合肥大學
HEFEI UNIVERSITY



Programming with Python

33. Zwischenspiel: Testen auf Ausnahmen

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. `pytest.raises`
3. Beispiele
4. Zusammenfassung



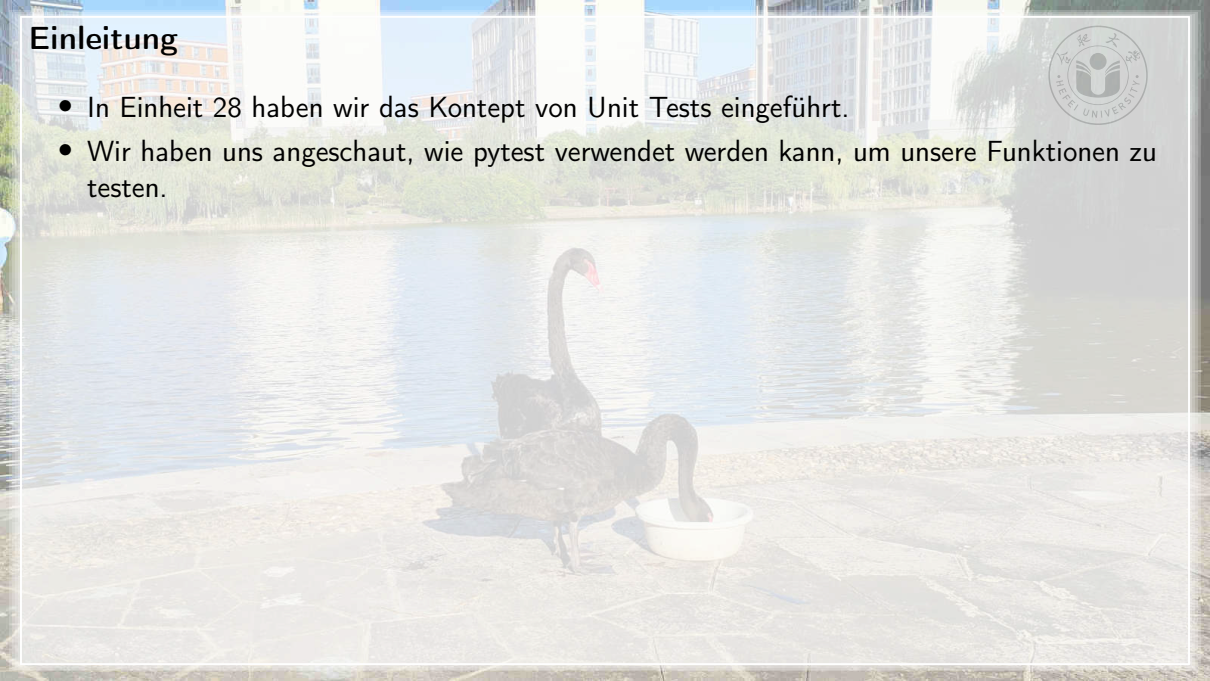
Einleitung

Einleitung

- In Einheit 28 haben wir das Kontept von Unit Tests eingeführt.



Einleitung



- In Einheit 28 haben wir das Kontept von Unit Tests eingeführt.
- Wir haben uns angeschaut, wie pytest verwendet werden kann, um unsere Funktionen zu testen.

Einleitung



- In Einheit 28 haben wir das Kontept von Unit Tests eingeführt.
- Wir haben uns angeschaut, wie pytest verwendet werden kann, um unsere Funktionen zu testen.
- Wir haben auch gesagt, dass wir idealerweise alle Zweige des Kontrollflusses in einer Funktion mit Unit Tests abdecken sollten.

Einleitung



- In Einheit 28 haben wir das Kontept von Unit Tests eingeführt.
- Wir haben uns angeschaut, wie pytest verwendet werden kann, um unsere Funktionen zu testen.
- Wir haben auch gesagt, dass wir idealerweise alle Zweige des Kontrollflusses in einer Funktion mit Unit Tests abdecken sollten.
- Bei der Abdeckung mit Tests werden Ausnahmen und der Kode zur Ausnahmebehandlung allerdings oft übersehen[39].

Einleitung



- In Einheit 28 haben wir das Kontept von Unit Tests eingeführt.
- Wir haben uns angeschaut, wie pytest verwendet werden kann, um unsere Funktionen zu testen.
- Wir haben auch gesagt, dass wir idealerweise alle Zweige des Kontrollflusses in einer Funktion mit Unit Tests abdecken sollten.
- Bei der Abdeckung mit Tests werden Ausnahmen und der Kode zur Ausnahmebehandlung allerdings oft übersehen[39].
- Wenn unsere Funktion eine bestimmte Ausnahme unter bestimmten Bedingungen auslösen soll, dann sollten wir Unit Tests haben, die prüfen, ob die Ausnahme auch tatsächlich ausgelöst wird.

Einleitung



- In Einheit 28 haben wir das Kontept von Unit Tests eingeführt.
- Wir haben uns angeschaut, wie pytest verwendet werden kann, um unsere Funktionen zu testen.
- Wir haben auch gesagt, dass wir idealerweise alle Zweige des Kontrollflusses in einer Funktion mit Unit Tests abdecken sollten.
- Bei der Abdeckung mit Tests werden Ausnahmen und der Kode zur Ausnahmebehandlung allerdings oft übersehen[39].
- Wenn unsere Funktion eine bestimmte Ausnahme unter bestimmten Bedingungen auslösen soll, dann sollten wir Unit Tests haben, die prüfen, ob die Ausnahme auch tatsächlich ausgelöst wird.
- Natürlich führt jede Ausnahme, die in einem Unit Test ausgelöst wird, dazu, dass der Unit Test fehlschlägt.

Einleitung

- In Einheit 28 haben wir das Kontept von Unit Tests eingeführt.
- Wir haben uns angeschaut, wie pytest verwendet werden kann, um unsere Funktionen zu testen.
- Wir haben auch gesagt, dass wir idealerweise alle Zweige des Kontrollflusses in einer Funktion mit Unit Tests abdecken sollten.
- Bei der Abdeckung mit Tests werden Ausnahmen und der Kode zur Ausnahmebehandlung allerdings oft übersehen[39].
- Wenn unsere Funktion eine bestimmte Ausnahme unter bestimmten Bedingungen auslösen soll, dann sollten wir Unit Tests haben, die prüfen, ob die Ausnahme auch tatsächlich ausgelöst wird.
- Natürlich führt jede Ausnahme, die in einem Unit Test ausgelöst wird, dazu, dass der Unit Test fehlschlägt.
- Dies scheint unserem Ziel, Ausnahmen *absichtlich* auszulösen, zu widersprechen.

Einleitung



- In Einheit 28 haben wir das Kontept von Unit Tests eingeführt.
- Wir haben uns angeschaut, wie pytest verwendet werden kann, um unsere Funktionen zu testen.
- Wir haben auch gesagt, dass wir idealerweise alle Zweige des Kontrollflusses in einer Funktion mit Unit Tests abdecken sollten.
- Bei der Abdeckung mit Tests werden Ausnahmen und der Kode zur Ausnahmebehandlung allerdings oft übersehen[39].
- Wenn unsere Funktion eine bestimmte Ausnahme unter bestimmten Bedingungen auslösen soll, dann sollten wir Unit Tests haben, die prüfen, ob die Ausnahme auch tatsächlich ausgelöst wird.
- Natürlich führt jede Ausnahme, die in einem Unit Test ausgelöst wird, dazu, dass der Unit Test fehlschlägt.
- Dies scheint unserem Ziel, Ausnahmen *absichtlich* auszulösen, zu widersprechen.
- Zum Glück bietet uns pytest dafür passende Werkzeuge an.

pytest.raises



pytest.raises

- Das Modul `pytest` bietet uns einen Context-Manager[16] namens `raises` an.

```python
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised > OR < if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Das Modul `pytest` bietet uns einen Context-Manager[16] namens `raises` an.
- Wir haben ja gerade eben gelernt, wie man Context-Managers mit dem `with`-Statement verwendet.

```python
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised > OR < if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Das Modul `pytest` bietet uns einen Context-Manager[16] namens `raises` an.
- Wir haben ja gerade eben gelernt, wie man Context-Managers mit dem `with`-Statement verwendet.
- Wenn wir prüfen wollen, ob eine Funktion wirklich eine Ausnahme vom Typ `ExceptionType` für einen bestimmten Input auslöst, dann packen wir den Funktionsaufruf in einen `with raises(ExceptionType):`-Block.

```python
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised > OR < if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Das Modul `pytest` bietet uns einen Context-Manager[16] namens `raises` an.
- Wir haben ja gerade eben gelernt, wie man Context-Managers mit dem `with`-Statement verwendet.
- Wenn wir prüfen wollen, ob eine Funktion wirklich eine Ausnahme vom Typ `ExceptionType` für einen bestimmten Input auslöst, dann packen wir den Funktionsaufruf in einen `with raises(ExceptionType):`-Block.
- Dieser Block sagt dem pytest-System, dass der folgende eingerückte Block eine Ausnahme vom Typ `ExceptionType` auslösen muss.

```
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised > OR < if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Wir haben ja gerade eben gelernt, wie man Context-Managers mit dem `with`-Statement verwendet.
- Wenn wir prüfen wollen, ob eine Funktion wirklich eine Ausnahme vom Typ `ExceptionType` für einen bestimmten Input auslöst, dann packen wir den Funktionsaufruf in einen `with raises(ExceptionType):`-Block.
- Dieser Block sagt dem pytest-System, dass der folgende eingerückte Block eine Ausnahme vom Typ `ExceptionType` auslösen muss.
- Wenn so eine Ausnahme nicht ausgelöst wird, dann schlägt der Test fehl.

```
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised > OR < if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Wenn wir prüfen wollen, ob eine Funktion wirklich eine Ausnahme vom Typ `ExceptionType` für einen bestimmten Input auslöst, dann packen wir den Funktionsaufruf in einen `with raises(ExceptionType):`-Block.
- Dieser Block sagt dem pytest-System, dass der folgende eingerückte Block eine Ausnahme vom Typ `ExceptionType` auslösen muss.
- Wenn so eine Ausnahme nicht ausgelöst wird, dann schlägt der Test fehl.
- Wenn sie ausgelöst wird, dann ist der Test erfolgreich.

```
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised, OR < if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Dieser Block sagt dem pytest-System, dass der folgende eingerückte Block eine Ausnahme vom Typ `ExceptionType` auslösen muss.
- Wenn so eine Ausnahme nicht ausgelöst wird, dann schlägt der Test fehl.
- Wenn sie ausgelöst wird, dann ist der Test erfolgreich.
- Wir haben auch gelernt, dass wir eine Fehlermeldung als Parameter beim Auslösen einer Ausnahme angeben können.

```
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised -> OR -> if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Wenn so eine Ausnahme nicht ausgelöst wird, dann schlägt der Test fehl.
- Wenn sie ausgelöst wird, dann ist der Test erfolgreich.
- Wir haben auch gelernt, dass wir eine Fehlermeldung als Parameter beim Auslösen einer Ausnahme angeben können.
- Mit `raises` können wir die String-Repräsentation der Ausnahme (welche diese Fehlermeldung beinhaltet) mit einer regular expression (regex) vergleichen.

```
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised -> OR <- if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Wenn sie ausgelöst wird, dann ist der Test erfolgreich.
- Wir haben auch gelernt, dass wir eine Fehlermeldung als Parameter beim Auslösen einer Ausnahme angeben können.
- Mit `raises` können wir die String-Repräsentation der Ausnahme (welche diese Fehlermeldung beinhaltet) mit einer regular expression (regex) vergleichen.
- Wir geben die regex dafür als Parameter `match` an.

```python
"""The syntax of the `raises` context manager offered by `pytest`."""

from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised -> OR -> if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

pytest.raises

- Mit `raises` können wir die String-Repräsentation der Ausnahme (welche diese Fehlermeldung beinhaltet) mit einer regular expression (regex) vergleichen.
- Wir geben die regex dafür als Parameter `match` an.
- Dann schlägt der Unit Test fehl, wenn entweder keine Ausnahme vom Typ `ExceptionType` ausgelöst wurde oder wenn so eine Ausnahme ausgelöst wurde, ihre String-Repräsentation aber nicht zur regex in `match` passt.

```python
"""The syntax of the `raises` context manager offered by `pytest`."""

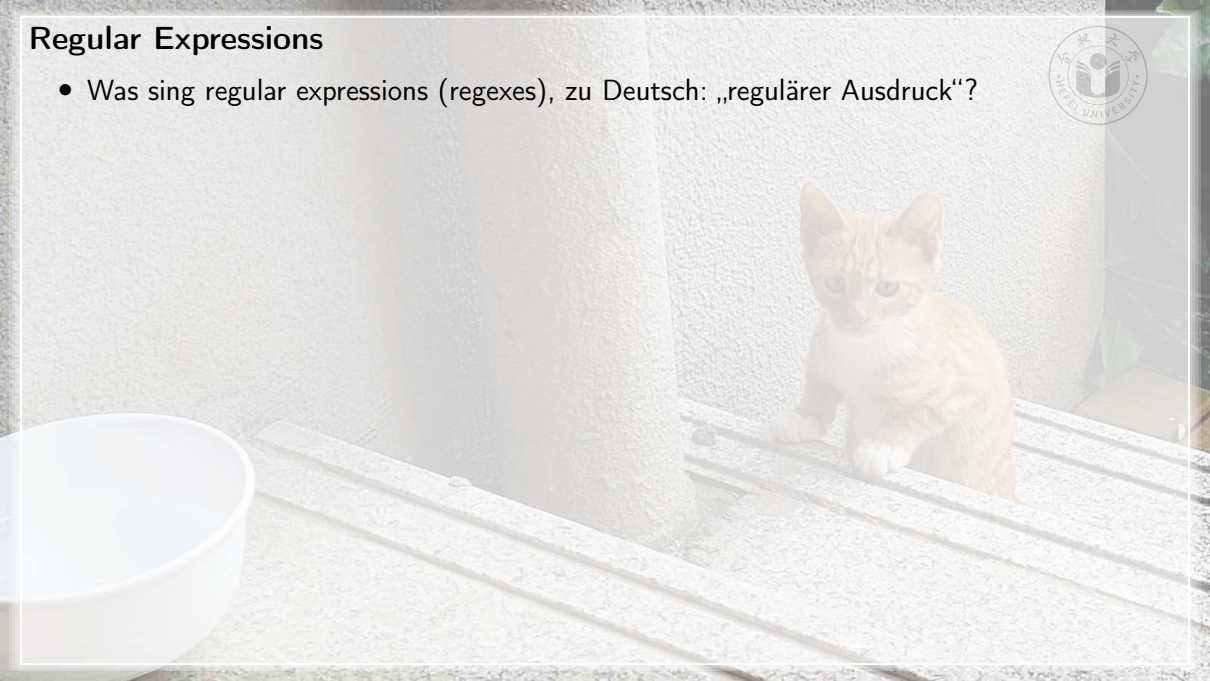
from pytest import raises  # Needed checking that exceptions are raised.

# `raises` is a context manager that will cause the test to fail if no
# exception of type `ExceptionType` is raised.
with raises(ExceptionType):
    code that should raise ExceptionType

# We can optionally provide a regular expression with parameter `match`.
# If either no exception of type `ExceptionType` is raised -> OR -> if the
# string-representation of the exception (usually corresponding to the
# error message) does not match to this regex, then the test will fail.
with raises(ExceptionType, match="error message regex"):
    code that should raise ExceptionType with fitting error message
```

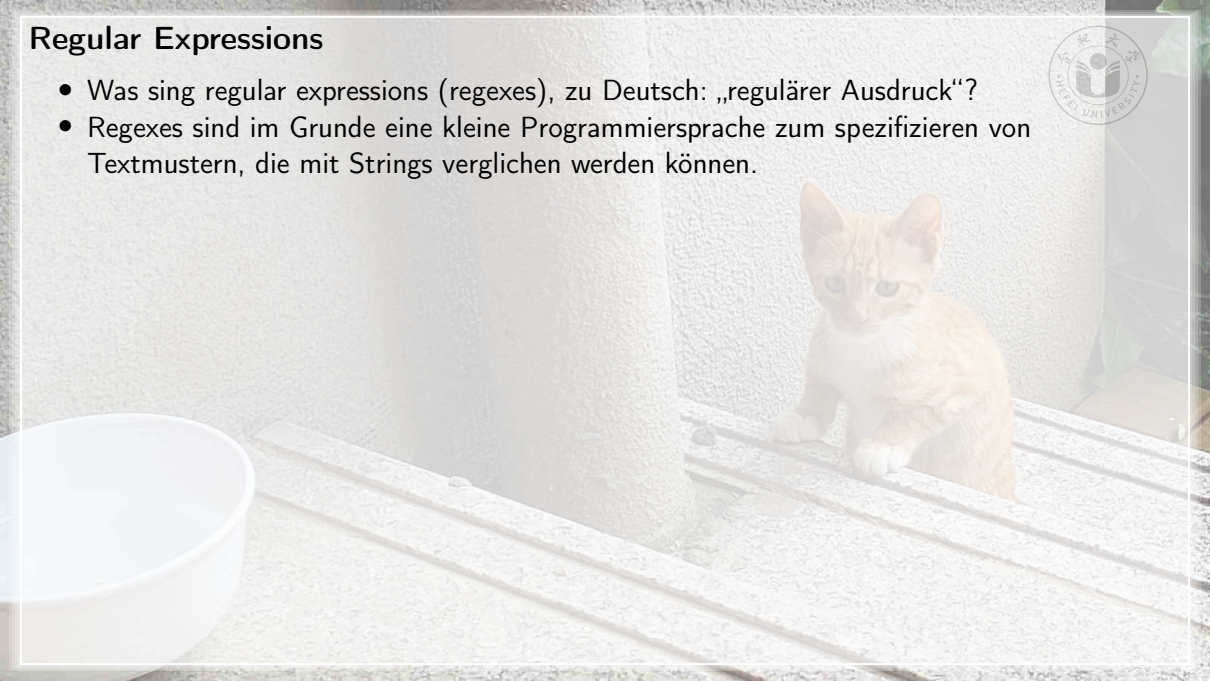
Regular Expressions

- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?



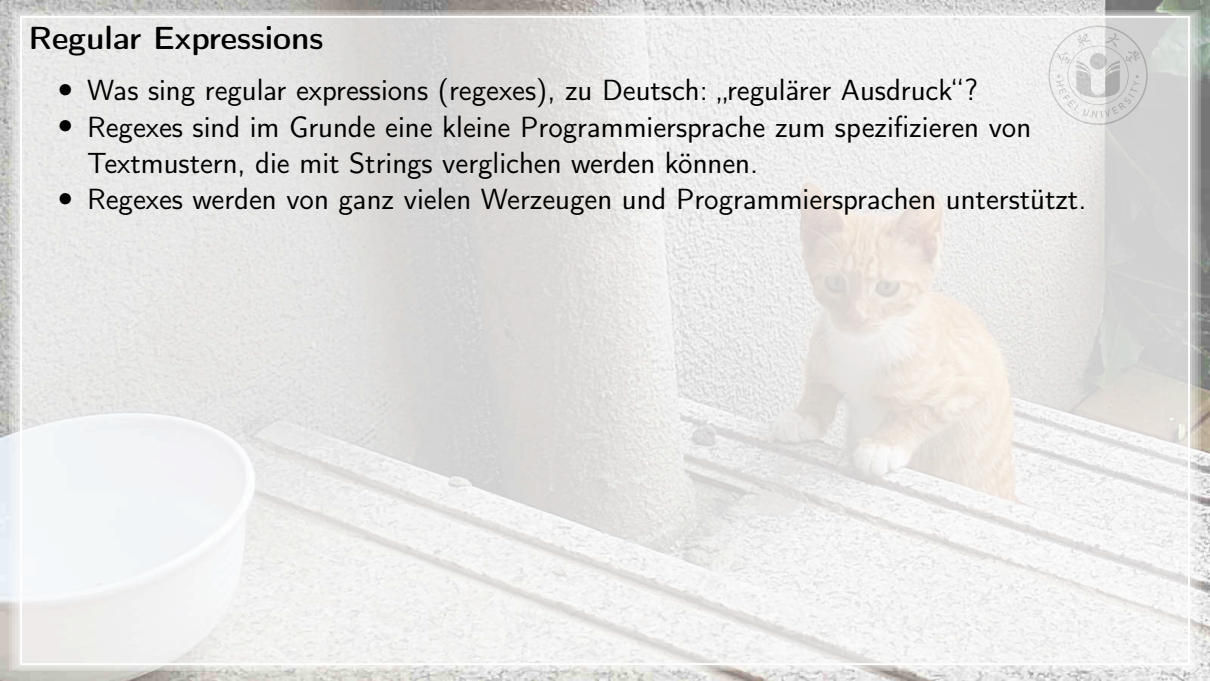
Regular Expressions

- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.



Regular Expressions

- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.



Regular Expressions

- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.

Regular Expressions



- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.

Regular Expressions



- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.

Regular Expressions



- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.

Regular Expressions



- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.

Regular Expressions



- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.

Regular Expressions



- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.

Regular Expressions



- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.

Regular Expressions



- Was sing regular expressions (regexes), zu Deutsch: „regulärer Ausdruck“?
- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.
- Im einfachsten Fall ist ein Unter-Muster ein einziges Zeichen.

Regular Expressions



- Regexes sind im Grunde eine kleine Programmiersprache zum spezifizieren von Textmustern, die mit Strings verglichen werden können.
- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.
- Im einfachsten Fall ist ein Unter-Muster ein einziges Zeichen.
- Die regex `"he*llo"` passt also zu `"hllo"`, `"hello"`, `"heello"`, `"heeello"`, usw.

Regular Expressions

- Regexes werden von ganz vielen Werzeugen und Programmiersprachen unterstützt.
- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.
- Im einfachsten Fall ist ein Unter-Muster ein einziges Zeichen.
- Die regex `"he*llo"` passt also zu `"hllo"`, `"hello"`, `"heello"`, `"heeello"`, usw.
- Die regex `"h.*llo"` passt zu `"hllo"`, `"hello"`, `"hallo"`, `"heeeXYZeeeello"`, usw.

Regular Expressions

- Wir können sie hier nicht tiefgehend diskutieren, aber zumindest ein paar ganz einfache Beispiele anschauen.
- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.
- Im einfachsten Fall ist ein Unter-Muster ein einziges Zeichen.
- Die regex `"he*llo"` passt also zu `"hllo"`, `"hello"`, `"heello"`, `"heeello"`, usw.
- Die regex `"h.*llo"` passt zu `"hllo"`, `"hello"`, `"hallo"`, `"heeeXYZeeeello"`, usw.
- Es gibt noch sehr viel mehr Muster, die wir mit regexes bauen können.

Regular Expressions

- Im einfachsten Fall ist eine regex ein ganz normaler String, z. B. `"hello"`.
- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.
- Im einfachsten Fall ist ein Unter-Muster ein einziges Zeichen.
- Die regex `"he*llo"` passt also zu `"hllo"`, `"hello"`, `"heello"`, `"heeello"`, usw.
- Die regex `"h.*llo"` passt zu `"hllo"`, `"hello"`, `"hallo"`, `"heeeXYZeeeello"`, usw.
- Es gibt noch sehr viel mehr Muster, die wir mit regexes bauen können.
- Dazu gibt es noch mehr Spezial-Zeichen.

Regular Expressions

- Wenn dieser String mit einem anderen String in einer Variable `x` verglichen wird, dann passt er nur, wenn `x` genau gleich `"hello"` ist. Dann spricht man von einem „match“.
- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.
- Im einfachsten Fall ist ein Unter-Muster ein einziges Zeichen.
- Die regex `"he*llo"` passt also zu `"hllo"`, `"hello"`, `"heello"`, `"heeello"`, usw.
- Die regex `"h.*llo"` passt zu `"hllo"`, `"hello"`, `"hallo"`, `"heeeXYZeeeello"`, usw.
- Es gibt noch sehr viel mehr Muster, die wir mit regexes bauen können.
- Dazu gibt es noch mehr Spezial-Zeichen.
- Wir müssen es hier aber erstmal dabei belassen.

Regular Expressions



- Ist `x` nicht gleich `"hello"` , dann passt er nicht.
- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.
- Im einfachsten Fall ist ein Unter-Muster ein einziges Zeichen.
- Die regex `"he*llo"` passt also zu `"hllo"`, `"hello"`, `"heello"`, `"heeello"`, usw.
- Die regex `"h.*llo"` passt zu `"hllo"`, `"hello"`, `"hallo"`, `"heeeXYZeeeello"`, usw.
- Es gibt noch sehr viel mehr Muster, die wir mit regexes bauen können.
- Dazu gibt es noch mehr Spezial-Zeichen.
- Wir müssen es hier aber erstmal dabei belassen.
- Sie können mehr Informationen in [33, 36, 45, 47, 59] finden.

Regular Expressions



- Es gibt Spezial-Zeichen, die regexes ziemlich mächtig machen.
- Ein Beispiel ist der Punkt `.`, der für ein beliebiges Zeichen steht.
- Der regex `"h.llo"` passt daher zu `"hello"`, `"hallo"`, und `"hXllo"`, oder `"h llo"`.
- Der Stern (`*`) spezifiziert, dass das Unter-Muster genau vor ihm nicht, einmal, oder mehrmals auftauchen kann.
- Im einfachsten Fall ist ein Unter-Muster ein einziges Zeichen.
- Die regex `"he*llo"` passt also zu `"hllo"`, `"hello"`, `"heello"`, `"heeello"`, usw.
- Die regex `"h.*llo"` passt zu `"hllo"`, `"hello"`, `"hallo"`, `"heeeXYZeeeello"`, usw.
- Es gibt noch sehr viel mehr Muster, die wir mit regexes bauen können.
- Dazu gibt es noch mehr Spezial-Zeichen.
- Wir müssen es hier aber erstmal dabei belassen.
- Sie können mehr Informationen in [33, 36, 45, 47, 59] finden.
- So oder so, ich denke, Sie sehen warum regexes eine sinnvolle Idee sind, um zu prüfen, ob Fehlermeldungen einer bestimmten Struktur entsprechen.

Beispiele

Beispiel: Verbesserte `sqrt`-Funktion



- In Einheit 28 hatten wir die Implementierungen unserer `sqrt`-Funktionen gestestet.

```python
"""Testing our third version of the `my_math` module."""

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

from my_math_3 import sqrt  # Get our 3rd square root implementation.

# test_factorial() is omitted for brevity

def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math_3`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Beispiel: Verbesserte `sqrt`-Funktion



- In Einheit 28 hatten wir die Implementierungen unserer `sqrt`-Funktionen gestestet.
- Das sah damals so aus.

```python
"""Testing our third version of the `my_math` module."""

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

from my_math_3 import sqrt  # Get our 3rd square root implementation.

# test_factorial() is omitted for brevity

def test_sqrt() -> None:
    """Test the function `sqrt` from module `my_math_3`."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10  # 1e10^2 = 1e10 * 1e10
    assert sqrt(inf) == inf  # The square root of +inf is +inf.
    assert isnan(sqrt(nan))  # The root of not-a-number is still nan.
```

Beispiel: Verbesserte `sqrt`-Funktion

- In Einheit 28 hatten wir die Implementierungen unserer `sqrt`-Funktionen gestestet.
- Das sah damals so aus.
- Dann hatten wir unsere `sqrt`-Funktion in Einheit 31weiterentwickelt.

```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: Verbesserte `sqrt`-Funktion

- In Einheit 28 hatten wir die Implementierungen unserer `sqrt`-Funktionen gestestet.
- Das sah damals so aus.
- Dann hatten wir unsere `sqrt`-Funktion in Einheit 31weiterentwickelt.
- Diese Funktion wird einen `TypeError` mit einer hartkodierten Fehlermeldung auslösen, wenn ihr argument kein `float` ist.

```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: Verbesserte `sqrt`-Funktion

- In Einheit 28 hatten wir die Implementierungen unserer `sqrt`-Funktionen gestestet.
- Das sah damals so aus.
- Dann hatten wir unsere `sqrt`-Funktion in Einheit 31weiterentwickelt.
- Diese Funktion wird einen `TypeError` mit einer hartkodierten Fehlermeldung auslösen, wenn ihr argument kein `float` ist.
- Sie löst einen `ArithmeticError` aus, wenn ihr Argument entweder negativ oder nicht endlich ist, wobei die Fehlermeldung den Wert des Arguments beinhaltet.

```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: Verbesserte `sqrt`-Funktion

- Das sah damals so aus.
- Dann hatten wir unsere `sqrt`-Funktion in Einheit 31weiterentwickelt.
- Diese Funktion wird einen `TypeError` mit einer hartkodierten Fehlermeldung auslösen, wenn ihr argument kein `float` ist.
- Sie löst einen `ArithmeticError` aus, wenn ihr Argument entweder negativ oder nicht endlich ist, wobei die Fehlermeldung den Wert des Arguments beinhaltet.
- Natürlich wollen wir sicher gehen, dass das auch passiert.

```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: Verbesserte `sqrt`-Funktion



- Diese Funktion wird einen `TypeError` mit einer hartkodierten Fehlermeldung auslösen, wenn ihr argument kein `float` ist.
- Sie löst einen `ArithmeticError` aus, wenn ihr Argument entweder negativ oder nicht endlich ist, wobei die Fehlermeldung den Wert des Arguments beinhaltet.
- Natürlich wollen wir sicher gehen, dass das auch passiert.
- Daher wollen wir also auf diese `ArithmeticErrors` und `TypeErrors` hin testen.

```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: Verbesserte `sqrt`-Funktion Testen

- In der ersten Test-Funktion, `test_sqrt`, machen wir die „normalen“ tests.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- In der ersten Test-Funktion, `test_sqrt`, machen wir die „normalen“ tests.
- Wir müssen ja trotzdem prüfen, ob `sqrt` richtige Ergebnisse für normale Zahlen liefert.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- In der ersten Test-Funktion, `test_sqrt`, machen wir die „normalen“ tests.
- Wir müssen ja trotzdem prüfen, ob `sqrt` richtige Ergebnisse für normale Zahlen liefert.
- In der zweiten Funktion, `test_sqrt_raises_arithmetic_erro` prüfen wir Argumente, die zwar Fließkommazahlen sind, aber ungültige Werte haben.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- In der ersten Test-Funktion, `test_sqrt`, machen wir die „normalen“ tests.
- Wir müssen ja trotzdem prüfen, ob `sqrt` richtige Ergebnisse für normale Zahlen liefert.
- In der zweiten Funktion, `test_sqrt_raises_arithmetic_erro` prüfen wir Argumente, die zwar Fließkommazahlen sind, aber ungültige Werte haben.
- In einr `for`-Schleife lassen wir eine Variable `number` über die Werte `[-1.0, inf, -inf, nan]` iterieren.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- In der ersten Test-Funktion, `test_sqrt`, machen wir die „normalen“ tests.
- Wir müssen ja trotzdem prüfen, ob `sqrt` richtige Ergebnisse für normale Zahlen liefert.
- In der zweiten Funktion, `test_sqrt_raises_arithmetic_erro` prüfen wir Argumente, die zwar Fließkommazahlen sind, aber ungültige Werte haben.
- In einr `for`-Schleife lassen wir eine Variable `number` über die Werte `[-1.0, inf, -inf, nan]` iterieren.
- Der erste Wert ist negativ, die anderen sind nicht endlich.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- In der zweiten Funktion, `test_sqrt_raises_arithmetic_erro` prüfen wir Argumente, die zwar Fließkommazahlen sind, aber ungültige Werte haben.
- In einr `for`-Schleife lassen wir eine Variable `number` über die Werte `[-1.0, inf, -inf, nan]` iterieren.
- Der erste Wert ist negativ, die anderen sind nicht endlich.
- Die Überprüfung `(not isfinite(number)) or (number < 0.0)` in unserer Funktion sollte diese Werte abfangen – es sei denn, wir haben irgendwie falsch verstanden, wie er funktioniert.

```
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- In einr `for`-Schleife lassen wir eine Variable `number` über die Werte `[-1.0, inf, -inf, nan]` iterieren.
- Der erste Wert ist negativ, die anderen sind nicht endlich.
- Die Überprüfung `(not isfinite(number)) or (number < 0.0)` in unserer Funktion sollte diese Werte abfangen – es sei denn, wir haben irgendwie falsch verstanden, wie er funktioniert.
- Alle diese Werte sollten unsere `sqrt`-Funktion dazu bringen, einen `ArithmeticError` auszulösen.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Der erste Wert ist negativ, die anderen sind nicht endlich.
- Die Überprüfung `(not isfinite(number)) or (number < 0.0)` in unserer Funktion sollte diese Werte abfangen – es sei denn, wir haben irgendwie falsch verstanden, wie er funktioniert.
- Alle diese Werte sollten unsere `sqrt`-Funktion dazu bringen, einen `ArithmeticError` auszulösen.
- Die Fehlermeldung in dieser Ausnahme wird über den folgenden f-String gebaut: `f"sqrt({number})is not permitted."`.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Alle diese Werte sollten unsere `sqrt`-Funktion dazu bringen, einen `ArithmeticError` auszulösen.
- Die Fehlermeldung in dieser Ausnahme wird über den folgenden f-String gebaut: `f"sqrt({number})is not permitted."`.
- In der Schleife im Test packen wir den Funktionsaufruf also in einen `with`-Block der den `raises`-Context-Manager benutzt.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Alle diese Werte sollten unsere `sqrt`-Funktion dazu bringen, einen `ArithmeticError` auszulösen.
- Die Fehlermeldung in dieser Ausnahme wird über den folgenden f-String gebaut: `f"sqrt({number})is not permitted."`.
- In der Schleife im Test packen wir den Funktionsaufruf also in einen `with`-Block der den `raises`-Context-Manager benutzt.
- Wir geben die Ausnahmen-Klasse `ArithmeticError` an.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Die Fehlermeldung in dieser Ausnahme wird über den folgenden f-String gebaut: `f"sqrt({number})is not permitted."`.
- In der Schleife im Test packen wir den Funktionsaufruf also in einen `with`-Block der den `raises`-Context-Manager benutzt.
- Wir geben die Ausnahmen-Klasse `ArithmeticError` an.
- Als Muster für die String-Repräsentation (also die Fehlermeldung) der Ausnahme nehmen wir `"sqrt.* is not permitted."`.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- In der Schleife im Test packen wir den Funktionsaufruf also in einen `with`-Block der den `raises`-Context-Manager benutzt.
- Wir geben die Ausnahmen-Klasse `ArithmeticError` an.
- Als Muster für die String-Repräsentation (also die Fehlermeldung) der Ausnahme nehmen wir `"sqrt.* is not permitted."`.
- Dadurch wird verlangt, dass jeder `sqrt`-Aufruf in der Schleife einen `ArithmeticError` auslöst.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Wir geben die Ausnahmen-Klasse `ArithmeticError` an.
- Als Muster für die String-Repräsentation (also die Fehlermeldung) der Ausnahme nehmen wir `"sqrt.* is not permitted."`.
- Dadurch wird verlangt, dass jeder `sqrt`-Aufruf in der Schleife einen `ArithmeticError` auslöst.
- Die Fehlermeldung im Ausnahme-Objekt muss mit `sqrt` anfangen und mit `is not permitted.` enden.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Als Muster für die String-Repräsentation (also die Fehlermeldung) der Ausnahme nehmen wir `"sqrt.* is not permitted."`.
- Dadurch wird verlangt, dass jeder `sqrt`-Aufruf in der Schleife einen `ArithmeticError` auslöst.
- Die Fehlermeldung im Ausnahme-Objekt muss mit `sqrt` anfangen und mit `is not permitted.` enden.
- Zwischen diesen beiden Strings können beliebig viele beliebige Zeichen auftreten.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Die Fehlermeldung im Ausnahme-Objekt muss mit `sqrt` anfangen und mit `is not permitted.` enden.
- Zwischen diesen beiden Strings können beliebig viele beliebige Zeichen auftreten.
- In anderen Worten, `"sqrt.* is not permitted."` matched `"sqrt is not permitted."`, `"sqrt(1)is not permitted."`, `"sqrt(inf)is not permitted."`, aber nicht `"sqrt is wrong."`.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Zwischen diesen beiden Strings können beliebig viele beliebige Zeichen auftreten.
- In anderen Worten, `"sqrt.* is not permitted."` matched `"sqrt is not permitted."`, `"sqrt(1)is not permitted."`, `"sqrt(inf)is not permitted."`, aber nicht `"sqrt is wrong."`.
- Wenn auch nur für einen Wert `number` keine solche Ausnahme ausgelöst wird, schlägt der Test fehl.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- In anderen Worten, `"sqrt.* is not permitted."` matched `"sqrt is not permitted."`, `"sqrt(1)is not permitted."`, `"sqrt(inf)is not permitted."`, aber nicht `"sqrt is wrong."`.
- Wenn auch nur für einen Wert `number` keine solche Ausnahme ausgelöst wird, schlägt der Test fehl.
- In der dritten und letzten Testfunktion, `test_sqrt_raises_type_error`, prüfen wir, ob die `TypeErrors` ordentlich ausgelöst werden.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Wenn auch nur für einen Wert `number` keine solche Ausnahme ausgelöst wird, schlägt der Test fehl.
- In der dritten und letzten Testfunktion, `test_sqrt_raises_type_error`, prüfen wir, ob die `TypeErrors` ordentlich ausgelöst werden.
- Wir gehen dazu genauso vor wie in der vorherigen Testfunktion.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Wenn auch nur für einen Wert `number` keine solche Ausnahme ausgelöst wird, schlägt der Test fehl.
- In der dritten und letzten Testfunktion, `test_sqrt_raises_type_error`, prüfen wir, ob die `TypeErrors` ordentlich ausgelöst werden.
- Wir gehen dazu genauso vor wie in der vorherigen Testfunktion.
- In einer `for`-Schleife lassen wir die Variable `number` über die Werte `[True, "x", None]` iterieren.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Wenn auch nur für einen Wert `number` keine solche Ausnahme ausgelöst wird, schlägt der Test fehl.
- In der dritten und letzten Testfunktion, `test_sqrt_raises_type_error`, prüfen wir, ob die `TypeErrors` ordentlich ausgelöst werden.
- Wir gehen dazu genauso vor wie in der vorherigen Testfunktion.
- In einer `for`-Schleife lassen wir die Variable `number` über die Werte `[True, "x", None]` iterieren.
- Keiner dieser Werte ist vom Typ `float`.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- In der dritten und letzten Testfunktion, `test_sqrt_raises_type_error`, prüfen wir, ob die `TypeErrors` ordentlich ausgelöst werden.
- Wir gehen dazu genauso vor wie in der vorherigen Testfunktion.
- In einer `for`-Schleife lassen wir die Variable `number` über die Werte `[True, "x", None]` iterieren.
- Keiner dieser Werte ist vom Typ `float`.
- Die Überprüfung `isinstance(number, float)` sollte also verhindern, dass diese Werte in die Berechnung fließen.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Wir gehen dazu genauso vor wie in der vorherigen Testfunktion.
- In einer `for`-Schleife lassen wir die Variable `number` über die Werte `[True, "x", None]` iterieren.
- Keiner dieser Werte ist vom Typ `float`.
- Die Überprüfung `isinstance(number, float)` sollte also verhindern, dass diese Werte in die Berechnung fließen.
- Es sei denn natürlich dass wir irgendwie falsch verstanden haben, wie entering the actual computation, unless we misunderstood how `isinstance` funktioniert.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Die Überprüfung `isinstance(number, float)` sollte also verhindern, dass diese Werte in die Berechnung fließen.
- Es sei denn natürlich dass wir irgendwie falsch verstanden haben, wie entering the actual computation, unless we misunderstood how `isinstance` funktioniert.
- Wenn unsere `sqrt`-Funktion so funktioniert wie wir denken, dann sollte sie jeweils einen `TypeError` mit der Fehlernachricht `"number must be float!"` auslösen.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen

- Es sei denn natürlich dass wir irgendwie falsch verstanden haben, wie entering the actual computation, unless we misunderstood how `isinstance` funktioniert.
- Wenn unsere `sqrt`-Funktion so funktioniert wie wir denken, dann sollte sie jeweils einen `TypeError` mit der Fehlernachricht `"number must be float!"` auslösen.
- Deshalb setzen wir die Funktionsaufrufe in einen `with`-Block mit dem Context-Manager `raises(TypeError, match="number must be float!")`.

```python
"""Testing our sqrt function that raises an error for invalid inputs."""

from math import inf, nan  # some maths constants

from pytest import raises  # Needed checking that exceptions are raised.

from sqrt_raise_2 import sqrt  # Import our new sqrt function.


def test_sqrt() -> None:
    """Test the `sqrt` function on normal input values."""
    assert sqrt(0.0) == 0.0  # The square root of 0 is 0.
    assert sqrt(1.0) == 1.0  # The square root of 1 is 1.
    assert sqrt(4.0) == 2.0  # The square root of 4 is 2.
    s3: float = sqrt(3.0)    # Get the approximated square root of 3.
    assert abs(s3 * s3 - 3.0) <= 5e-16  # sqrt(3)^2 should be close to 3.
    assert sqrt(1e10 * 1e10) == 1e10    # 1e10^2 = 1e10 * 1e10


def test_sqrt_raises_arithmetic_error():
    """Check that `ArithmeticError` is properly raised."""
    for number in [-1.0, inf, -inf, nan]:  # negative or not finite...
        with raises(ArithmeticError, match="sqrt.* is not permitted."):
            sqrt(number)  # The square root of `number` is not defined.


def test_sqrt_raises_type_error():
    """Check that `TypeError` is properly raised."""
    for number in [True, "x", None]:  # all of these are NOT `float`s.
        with raises(TypeError, match="number must be float!"):
            sqrt(number)  # non-float values are not permitted.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Wenn unsere `sqrt`-Funktion so funktioniert wie wir denken, dann sollte sie jeweils einen `TypeError` mit der Fehlernachricht `"number must be float!"` auslösen.
- Deshalb setzen wir die Funktionsaufrufe in einen `with`-Block mit dem Context-Manager `raises(TypeError, match="number must be float!")`.
- Wir führen nun pytest aus wie gewöhnlich.

```
1 $ pytest --timeout=10 --no-header --tb=short test_sqrt_raise_2.py
2 ========================= test session starts
    ↪ ==========================
3 collected 3 items
4 test_sqrt_raise_2.py ...                                          [100%]
5 ========================== 3 passed in 0.01s
    ↪ ===========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Wenn unsere `sqrt`-Funktion so funktioniert wie wir denken, dann sollte sie jeweils einen `TypeError` mit der Fehlernachricht `"number must be float!"` auslösen.
- Deshalb setzen wir die Funktionsaufrufe in einen `with`-Block mit dem Context-Manager `raises(TypeError, match="number must be float!")`.
- Wir führen nun pytest aus wie gewöhnlich.
- Wie Sie sehen, sind alle drei Tests erfolgreich.

```
1 $ pytest --timeout=10 --no-header --tb=short test_sqrt_raise_2.py
2 ========================= test session starts
    ↪ ==========================
3 collected 3 items
4 test_sqrt_raise_2.py ...                                          [100%]
5 ========================== 3 passed in 0.01s
    ↪ ===========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Deshalb setzen wir die Funktionsaufrufe in einen `with`-Block mit dem Context-Manager `raises(TypeError, match="number must be float!")`.
- Wir führen nun pytest aus wie gewöhnlich.
- Wie Sie sehen, sind alle drei Tests erfolgreich.
- Das bedeutet, dass unsere `sqrt`-Funktion die erwarteten Ergebnisse für normale Eingabedaten zurückliefert.

```
1 $ pytest --timeout=10 --no-header --tb=short test_sqrt_raise_2.py
2 ========================= test session starts
    ↪ ==========================
3 collected 3 items
4 test_sqrt_raise_2.py ...                                          [100%]
5 ========================== 3 passed in 0.01s
    ↪ ===========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Wir führen nun pytest aus wie gewöhnlich.
- Wie Sie sehen, sind alle drei Tests erfolgreich.
- Das bedeutet, dass unsere `sqrt`-Funktion die erwarteten Ergebnisse für normale Eingabedaten zurückliefert.
- Es bedeutet auch, dass sie einen `ArithmeticError` mit einer passenden Nachricht für Fließkommazahlen auslöst, mit denen unsere Funktion nichts anfangen kann.

```
1 $ pytest --timeout=10 --no-header --tb=short test_sqrt_raise_2.py
2 ========================= test session starts
    ↪ ==========================
3 collected 3 items
4 test_sqrt_raise_2.py ...                                          [100%]
5 ========================== 3 passed in 0.01s
    ↪ ===========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Wie Sie sehen, sind alle drei Tests erfolgreich.
- Das bedeutet, dass unsere `sqrt`-Funktion die erwarteten Ergebnisse für normale Eingabedaten zurückliefert.
- Es bedeutet auch, dass sie einen `ArithmeticError` mit einer passenden Nachricht für Fließkommazahlen auslöst, mit denen unsere Funktion nichts anfangen kann.
- Und es bedeutet, dass sie einen `TypeError` mit passender Nachricht auslöst, wenn wir Argumente eingeben, die keine Fließkommazahlen sind.

```
1 $ pytest --timeout=10 --no-header --tb=short test_sqrt_raise_2.py
2 ========================= test session starts
    ↪ ==========================
3 collected 3 items
4 test_sqrt_raise_2.py ...                                          [100%]
5 ========================== 3 passed in 0.01s
    ↪ ===========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Beispiel: Verbesserte `sqrt`-Funktion Testen



- Das bedeutet, dass unsere `sqrt`-Funktion die erwarteten Ergebnisse für normale Eingabedaten zurückliefert.
- Es bedeutet auch, dass sie einen `ArithmeticError` mit einer passenden Nachricht für Fließkommazahlen auslöst, mit denen unsere Funktion nichts anfangen kann.
- Und es bedeutet, dass sie einen `TypeError` mit passender Nachricht auslöst, wenn wir Argumente eingeben, die keine Fließkommazahlen sind.
- Wir können also zuversichtlich sein, dass unsere Implementierg korrekt ist.

```
1 $ pytest --timeout=10 --no-header --tb=short test_sqrt_raise_2.py
2 ========================= test session starts
     ↪ ==========================
3 collected 3 items
4 test_sqrt_raise_2.py ...                                          [100%]
5 ========================== 3 passed in 0.01s
     ↪ ===========================
6 # pytest 9.2.0 with pytest-timeout 2.4.0 succeeded with exit code 0.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Jetzt wollen wir noch ein paar weitere Aspekte des `raises` Context-Managers aus dem Modul `pytest` ausprobieren.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Jetzt wollen wir noch ein paar weitere Aspekte des `raises` Context-Managers aus dem Modul `pytest` ausprobieren.
- In der Datei `test_sqrt.py` testen wir im Grunde die Originalversion unserer `sqrt` die nicht selbst Exceptions auslöst.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Jetzt wollen wir noch ein paar weitere Aspekte des `raises` Context-Managers aus dem Modul `pytest` ausprobieren.
- In der Datei `test_sqrt.py` testen wir im Grunde die Originalversion unserer `sqrt` die nicht selbst Exceptions auslöst.
- Wir kopieren die Funktion der Einfachheit halber direkt in die selbe Datei wie die Tests.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Jetzt wollen wir noch ein paar weitere Aspekte des `raises` Context-Managers aus dem Modul `pytest` ausprobieren.
- In der Datei `test_sqrt.py` testen wir im Grunde die Originalversion unserer `sqrt` die nicht selbst Exceptions auslöst.
- Wir kopieren die Funktion der Einfachheit halber direkt in die selbe Datei wie die Tests.
- Der Test `test_raises_arithmetic_error_1` übergibt den Wert `-1.0` als Argument an die Funktion `sqrt`.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- In der Datei `test_sqrt.py` testen wir im Grunde die Originalversion unserer `sqrt` die nicht selbst Exceptions auslöst.
- Wir kopieren die Funktion der Einfachheit halber direkt in die selbe Datei wie die Tests.
- Der Test `test_raises_arithmetic_error_1` übergibt den Wert `-1.0` als Argument an die Funktion `sqrt`.
- Er erwartet, dass ein `ArithmeticError` ausgelöst wird.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Wir kopieren die Funktion der Einfachheit halber direkt in die selbe Datei wie die Tests.
- Der Test `test_raises_arithmetic_error_1` übergibt den Wert `-1.0` als Argument an die Funktion `sqrt`.
- Er erwartet, dass ein `ArithmeticError` ausgelöst wird.
- Wir spezifizieren keinen Wert für den Parameter `match` von `raises`, also wird die Fehlermeldung der Ausnahme nicht ausgewertet.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Wir kopieren die Funktion der Einfachheit halber direkt in die selbe Datei wie die Tests.
- Der Test `test_raises_arithmetic_error_1` übergibt den Wert `-1.0` als Argument an die Funktion `sqrt`.
- Er erwartet, dass ein `ArithmeticError` ausgelöst wird.
- Wir spezifizieren keinen Wert für den Parameter `match` von `raises`, also wird die Fehlermeldung der Ausnahme nicht ausgewertet.
- Von der Ausgabe sehen wir, dass dieser Test fehlschlägt.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt.py
========================== test session starts
   ↪ ==========================
collected 4 items
test_sqrt.py F..F                                                        [100%]
================================== FAILURES
   ↪ ==================================
____________________ test_raises_arithmetic_error_1
   ↪ ____________________
test_sqrt.py:26: in test_raises_arithmetic_error_1
    with raises(ArithmeticError):  # We can also test without `match`.
         ^^^^^^^^^^^^^^^^^^^^^^^
E   Failed: DID NOT RAISE ArithmeticError
____________________ test_raises_arithmetic_error_3
   ↪ ____________________
test_sqrt.py:44: in test_raises_arithmetic_error_3
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
E   AssertionError: Regex pattern did not match.
E     Expected regex: 'sqrt.* is not permitted.'
E     Actual message: 'int too large to convert to float'
======================== short test summary info
   ↪ ========================
FAILED test_sqrt.py::test_raises_arithmetic_error_1 - Failed: DID NOT
   ↪ RAISE ArithmeticError
FAILED test_sqrt.py::test_raises_arithmetic_error_3 - AssertionError:
   ↪ Regex pattern did not match.
  Expected regex: 'sqrt.* is not permitted.'
  Actual message: 'int too large to convert to float'
====================== 2 failed, 2 passed in 0.02s
   ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Der Test `test_raises_arithmetic_error_1` übergibt den Wert `-1.0` als Argument an die Funktion `sqrt`.
- Er erwartet, dass ein `ArithmeticError` ausgelöst wird.
- Wir spezifizieren keinen Wert für den Parameter `match` von `raises`, also wird die Fehlermeldung der Ausnahme nicht ausgewertet.
- Von der Ausgabe sehen wir, dass dieser Test fehlschlägt.
- Der Grund ist, dass unsere alte `sqrt`-Implementierung einfach `-1` für negative Argumente zurückliefert

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Er erwartet, dass ein `ArithmeticError` ausgelöst wird.
- Wir spezifizieren keinen Wert für den Parameter `match` von `raises`, also wird die Fehlermeldung der Ausnahme nicht ausgewertet.
- Von der Ausgabe sehen wir, dass dieser Test fehlschlägt.
- Der Grund ist, dass unsere alte `sqrt`-Implementierung einfach `-1` für negative Argumente zurückliefert
- Sie löse niemals selber explizit eine Ausnahme aus.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Wir spezifizieren keinen Wert für den Parameter `match` von `raises`, also wird die Fehlermeldung der Ausnahme nicht ausgewertet.
- Von der Ausgabe sehen wir, dass dieser Test fehlschlägt.
- Der Grund ist, dass unsere alte `sqrt`-Implementierung einfach `-1` für negative Argumente zurückliefert
- Sie löse niemals selber explizit eine Ausnahme aus.
- Weil der Funktionsaufruf im `with raises(ArithmeticError):` in Wirklichkeit also keinen `ArithmeticError` auslöst, schlägt dieser Test fehl.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Von der Ausgabe sehen wir, dass dieser Test fehlschlägt.
- Der Grund ist, dass unsere alte `sqrt`-Implementierung einfach `-1` für negative Argumente zurückliefert
- Sie löse niemals selber explizit eine Ausnahme aus.
- Weil der Funktionsaufruf im `with raises(ArithmeticError):` in Wirklichkeit also keinen `ArithmeticError` auslöst, schlägt dieser Test fehl.
- Im zweiten Test Case, `test_raises_overflow_error`, rufen wir dann `sqrt(10 ** 320)` auf.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Sie löse niemals selber explizit eine Ausnahme aus.
- Weil der Funktionsaufruf im `with raises(ArithmeticError):` in Wirklichkeit also keinen `ArithmeticError` auslöst, schlägt dieser Test fehl.
- Im zweiten Test Case, `test_raises_overflow_error`, rufen wir dann `sqrt(10 ** 320)` auf.
- Wir setzen diesen Kode in einen `with`-Block mit Context-Manager `raises(OverflowError, match="int too large.*")`.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Weil der Funktionsaufruf im `with raises(ArithmeticError):` in Wirklichkeit also keinen `ArithmeticError` auslöst, schlägt dieser Test fehl.
- Im zweiten Test Case, `test_raises_overflow_error`, rufen wir dann `sqrt(10 ** 320)` auf.
- Wir setzen diesen Kode in einen `with`-Block mit Context-Manager `raises(OverflowError, match="int too large.*")`.
- Das bedeutet, dass wir erwarten, dass der Kode einen `OverflowError` auslöst.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Im zweiten Test Case, `test_raises_overflow_error`, rufen wir dann `sqrt(10 ** 320)` auf.
- Wir setzen diesen Kode in einen `with`-Block mit Context-Manager `raises(OverflowError, match="int too large.*")`.
- Das bedeutet, dass wir erwarten, dass der Kode einen `OverflowError` auslöst.
- Das Ausnahme-Objekt muss eine Fehlermeldung beinhalten, die mit `int too large` anfängt und danach beliebigen weiteren Text enthalten can.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Wir setzen diesen Kode in einen `with`-Block mit Context-Manager `raises(OverflowError, match="int too large.*")`.
- Das bedeutet, dass wir erwarten, dass der Kode einen `OverflowError` auslöst.
- Das Ausnahme-Objekt muss eine Fehlermeldung beinhalten, die mit `int too large` anfängt und danach beliebigen weiteren Text enthalten can.
- Und genau das passiert auch.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt.py
========================== test session starts
   ↪ ===========================
collected 4 items
test_sqrt.py F..F                                                        [100%]
================================ FAILURES
   ↪ ================================
____________________ test_raises_arithmetic_error_1
   ↪ ____________________
test_sqrt.py:26: in test_raises_arithmetic_error_1
    with raises(ArithmeticError):  # We can also test without `match`.
         ^^^^^^^^^^^^^^^^^^^^^^^
E   Failed: DID NOT RAISE ArithmeticError
____________________ test_raises_arithmetic_error_3
   ↪ ____________________
test_sqrt.py:44: in test_raises_arithmetic_error_3
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
E   AssertionError: Regex pattern did not match.
E     Expected regex: 'sqrt.* is not permitted.'
E     Actual message: 'int too large to convert to float'
======================== short test summary info
   ↪ ========================
FAILED test_sqrt.py::test_raises_arithmetic_error_1 - Failed: DID NOT
   ↪ RAISE ArithmeticError
FAILED test_sqrt.py::test_raises_arithmetic_error_3 - AssertionError:
   ↪ Regex pattern did not match.
  Expected regex: 'sqrt.* is not permitted.'
  Actual message: 'int too large to convert to float'
====================== 2 failed, 2 passed in 0.02s
   ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Wir setzen diesen Kode in einen `with`-Block mit Context-Manager `raises(OverflowError, match="int too large.*")`.
- Das bedeutet, dass wir erwarten, dass der Kode einen `OverflowError` auslöst.
- Das Ausnahme-Objekt muss eine Fehlermeldung beinhalten, die mit `int too large` anfängt und danach beliebigen weiteren Text enthalten can.
- Und genau das passiert auch.
- Der Kode in der Funktion führt ja Fließkommaarithmetik durch.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Das bedeutet, dass wir erwarten, dass der Kode einen `OverflowError` auslöst.
- Das Ausnahme-Objekt muss eine Fehlermeldung beinhalten, die mit `int too large` anfängt und danach beliebigen weiteren Text enthalten can.
- Und genau das passiert auch.
- Der Kode in der Funktion führt ja Fließkommaarithmetik durch.
- Wenn wir eine Ganzzahl da hinein geben, dann wird diese irgendwann in einen `float` umgewandelt.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Das Ausnahme-Objekt muss eine Fehlermeldung beinhalten, die mit `int too large` anfängt und danach beliebigen weiteren Text enthalten can.
- Und genau das passiert auch.
- Der Kode in der Funktion führt ja Fließkommaarithmetik durch.
- Wenn wir eine Ganzzahl da hinein geben, dann wird diese irgendwann in einen `float` umgewandelt.
- Die Ganzzahl 10^{320} ist allerdings zu groß, um in einen `float` umgewandelt zu werden.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess    # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Und genau das passiert auch.
- Der Kode in der Funktion führt ja Fließkommaarithmetik durch.
- Wenn wir eine Ganzzahl da hinein geben, dann wird diese irgendwann in einen `float` umgewandelt.
- Die Ganzzahl 10^{320} ist allerdings zu groß, um in einen `float` umgewandelt zu werden.
- Dieser Fehler führt zu einem `OverflowError`.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Der Kode in der Funktion führt ja Fließkommaarithmetik durch.
- Wenn wir eine Ganzzahl da hinein geben, dann wird diese irgendwann in einen `float` umgewandelt.
- Die Ganzzahl 10^{320} ist allerdings zu groß, um in einen `float` umgewandelt zu werden.
- Dieser Fehler führt zu einem `OverflowError`.
- Die Fehlermeldung in diesem Objekt passt zu dem Textmuster, dass wir angegeben haben.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Wenn wir eine Ganzzahl da hinein geben, dann wird diese irgendwann in einen `float` umgewandelt.
- Die Ganzzahl 10^{320} ist allerdings zu groß, um in einen `float` umgewandelt zu werden.
- Dieser Fehler führt zu einem `OverflowError`.
- Die Fehlermeldung in diesem Objekt passt zu dem Textmuster, dass wir angegeben haben.
- Deshalb ist dieser Test erfolgreich.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt.py
========================== test session starts
    ↪ ==========================
collected 4 items
test_sqrt.py F..F                                                        [100%]
================================== FAILURES
    ↪ ==================================
____________________ test_raises_arithmetic_error_1
    ↪ _____________________
test_sqrt.py:26: in test_raises_arithmetic_error_1
    with raises(ArithmeticError):  # We can also test without `match`.
         ^^^^^^^^^^^^^^^^^^^^^^^
E   Failed: DID NOT RAISE ArithmeticError
____________________ test_raises_arithmetic_error_3
    ↪ _____________________
test_sqrt.py:44: in test_raises_arithmetic_error_3
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
E   AssertionError: Regex pattern did not match.
E     Expected regex: 'sqrt.* is not permitted.'
E     Actual message: 'int too large to convert to float'
======================== short test summary info
    ↪ ========================
FAILED test_sqrt.py::test_raises_arithmetic_error_1 - Failed: DID NOT
    ↪ RAISE ArithmeticError
FAILED test_sqrt.py::test_raises_arithmetic_error_3 - AssertionError:
    ↪ Regex pattern did not match.
  Expected regex: 'sqrt.* is not permitted.'
  Actual message: 'int too large to convert to float'
====================== 2 failed, 2 passed in 0.02s
    ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Die Ganzzahl 10^{320} ist allerdings zu groß, um in einen `float` umgewandelt zu werden.
- Dieser Fehler führt zu einem `OverflowError`.
- Die Fehlermeldung in diesem Objekt passt zu dem Textmuster, dass wir angegeben haben.
- Deshalb ist dieser Test erfolgreich.
- Dann, im Test `test_raises_arithmetic_error_2`, packen wir den selben Funktionsaufruf in ein `with raises(ArithmeticError):`.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Dieser Fehler führt zu einem `OverflowError`.
- Die Fehlermeldung in diesem Objekt passt zu dem Textmuster, dass wir angegeben haben.
- Deshalb ist dieser Test erfolgreich.
- Dann, im Test `test_raises_arithmetic_error_2`, packen wir den selben Funktionsaufruf in ein `with raises(ArithmeticError):`.
- Nun haben wir gerade festgestellt, dass `sqrt(10 ** 320)` einen `OverflowError` auslöst.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Die Fehlermeldung in diesem Objekt passt zu dem Textmuster, dass wir angegeben haben.
- Deshalb ist dieser Test erfolgreich.
- Dann, im Test `test_raises_arithmetic_error_2`, packen wir den selben Funktionsaufruf in ein `with raises(ArithmeticError):`.
- Nun haben wir gerade festgestellt, dass `sqrt(10 ** 320)` einen `OverflowError` auslöst.
- Trotzdem ist auch dieser Test erfolgreich.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Deshalb ist dieser Test erfolgreich.
- Dann, im Test `test_raises_arithmetic_error_2`, packen wir den selben Funktionsaufruf in ein `with raises(ArithmeticError):`.
- Nun haben wir gerade festgestellt, dass `sqrt(10 ** 320)` einen `OverflowError` auslöst.
- Trotzdem ist auch dieser Test erfolgreich.
- Der Grund ist, dass ein `OverflowError` ja ein Spezialfall von `ArithmeticError` ist.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Dann, im Test `test_raises_arithmetic_error_2`, packen wir den selben Funktionsaufruf in ein `with raises(ArithmeticError):`.
- Nun haben wir gerade festgestellt, dass `sqrt(10 ** 320)` einen `OverflowError` auslöst.
- Trotzdem ist auch dieser Test erfolgreich.
- Der Grund ist, dass ein `OverflowError` ja ein Spezialfall von `ArithmeticError` ist.
- Wir haben hier auch keine besondere Fehlermeldung verlangt, also kein Argument für `match` spezifiziert.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Nun haben wir gerade festgestellt, dass `sqrt(10 ** 320)` einen `OverflowError` auslöst.
- Trotzdem ist auch dieser Test erfolgreich.
- Der Grund ist, dass ein `OverflowError` ja ein Spezialfall von `ArithmeticError` ist.
- Wir haben hier auch keine besondere Fehlermeldung verlangt, also kein Argument für `match` spezifiziert.
- Darum verlangt der Test nur, das ein `ArithmeticError` ausgelöst wird.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Trotzdem ist auch dieser Test erfolgreich.
- Der Grund ist, dass ein `OverflowError` ja ein Spezialfall von `ArithmeticError` ist.
- Wir haben hier auch keine besondere Fehlermeldung verlangt, also kein Argument für `match` spezifiziert.
- Darum verlangt der Test nur, das ein `ArithmeticError` ausgelöst wird.
- Er sieht dann einen Spezialfall davon und ist zufrieden.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt.py
========================== test session starts
   ↪ ==========================
collected 4 items
test_sqrt.py F..F                                                        [100%]
================================== FAILURES
   ↪ ==================================
____________________ test_raises_arithmetic_error_1
   ↪ ____________________
test_sqrt.py:26: in test_raises_arithmetic_error_1
    with raises(ArithmeticError):  # We can also test without `match`.
         ^^^^^^^^^^^^^^^^^^^^^^^
E   Failed: DID NOT RAISE ArithmeticError
____________________ test_raises_arithmetic_error_3
   ↪ ____________________
test_sqrt.py:44: in test_raises_arithmetic_error_3
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
E   AssertionError: Regex pattern did not match.
E     Expected regex: 'sqrt.* is not permitted.'
E     Actual message: 'int too large to convert to float'
======================== short test summary info
   ↪ ========================
FAILED test_sqrt.py::test_raises_arithmetic_error_1 - Failed: DID NOT
   ↪ RAISE ArithmeticError
FAILED test_sqrt.py::test_raises_arithmetic_error_3 - AssertionError:
   ↪ Regex pattern did not match.
  Expected regex: 'sqrt.* is not permitted.'
  Actual message: 'int too large to convert to float'
====================== 2 failed, 2 passed in 0.02s
   ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Der Grund ist, dass ein `OverflowError` ja ein Spezialfall von `ArithmeticError` ist.
- Wir haben hier auch keine besondere Fehlermeldung verlangt, also kein Argument für `match` spezifiziert.
- Darum verlangt der Test nur, das ein `ArithmeticError` ausgelöst wird.
- Er sieht dann einen Spezialfall davon und ist zufrieden.
- Der dritte Test ist daher auch erfolgreich.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt.py
========================== test session starts
   ↪ ==========================
collected 4 items
test_sqrt.py F..F                                                        [100%]
================================ FAILURES
   ↪ ================================
____________________ test_raises_arithmetic_error_1
   ↪ ____________________
test_sqrt.py:26: in test_raises_arithmetic_error_1
    with raises(ArithmeticError):  # We can also test without `match`.
         ^^^^^^^^^^^^^^^^^^^^^^^
E   Failed: DID NOT RAISE ArithmeticError
____________________ test_raises_arithmetic_error_3
   ↪ ____________________
test_sqrt.py:44: in test_raises_arithmetic_error_3
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
E   AssertionError: Regex pattern did not match.
E     Expected regex: 'sqrt.* is not permitted.'
E     Actual message: 'int too large to convert to float'
======================== short test summary info
   ↪ ========================
FAILED test_sqrt.py::test_raises_arithmetic_error_1 - Failed: DID NOT
   ↪ RAISE ArithmeticError
FAILED test_sqrt.py::test_raises_arithmetic_error_3 - AssertionError:
   ↪ Regex pattern did not match.
  Expected regex: 'sqrt.* is not permitted.'
  Actual message: 'int too large to convert to float'
====================== 2 failed, 2 passed in 0.02s
   ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Wir haben hier auch keine besondere Fehlermeldung verlangt, also kein Argument für `match` spezifiziert.
- Darum verlangt der Test nur, das ein `ArithmeticError` ausgelöst wird.
- Er sieht dann einen Spezialfall davon und ist zufrieden.
- Der dritte Test ist daher auch erfolgreich.
- Im vierten Test, `test_raises_arithmetic_error_3`, benutzen wir wieder `sqrt(10 ** 320)`.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Darum verlangt der Test nur, das ein `ArithmeticError` ausgelöst wird.
- Er sieht dann einen Spezialfall davon und ist zufrieden.
- Der dritte Test ist daher auch erfolgreich.
- Im vierten Test, `test_raises_arithmetic_error_3`, benutzen wir wieder `sqrt(10 ** 320)`.
- Dieses Mal spezifizieren wir den Context-Manager `with raises(ArithmeticError, match="sqrt.* is not permitted.")`.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Er sieht dann einen Spezialfall davon und ist zufrieden.
- Der dritte Test ist daher auch erfolgreich.
- Im vierten Test, `test_raises_arithmetic_error_3`, benutzen wir wieder `sqrt(10 ** 320)`.
- Dieses Mal spezifizieren wir den Context-Manager `with raises(ArithmeticError, match="sqrt.* is not permitted.")`.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Der dritte Test ist daher auch erfolgreich.
- Im vierten Test, `test_raises_arithmetic_error_3`, benutzen wir wieder `sqrt(10 ** 320)`.
- Dieses Mal spezifizieren wir den Context-Manager `with raises(ArithmeticError, match="sqrt.* is not permitted.")`.
- Das ist genau die gleiche Bedingung, mit der wir unsere neue `sqrt` Implementierung, die selber Ausnahmen auslöst, in `test_sqrt_raise_2.py` getestet hatten.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Im vierten Test, `test_raises_arithmetic_error_3`, benutzen wir wieder `sqrt(10 ** 320)`.
- Dieses Mal spezifizieren wir den Context-Manager `with raises(ArithmeticError, match="sqrt.* is not permitted.")`.
- Das ist genau die gleiche Bedingung, mit der wir unsere neue `sqrt` Implementierung, die selber Ausnahmen auslöst, in `test_sqrt_raise_2.py` getestet hatten.
- Diese neue Implementierung hat genau so einen Fehler ausgelöst `ArithmeticErrors`.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Dieses Mal spezifizieren wir den Context-Manager `with raises(ArithmeticError, match="sqrt.* is not permitted.")`.
- Das ist genau die gleiche Bedingung, mit der wir unsere neue `sqrt` Implementierung, die selber Ausnahmen auslöst, in `test_sqrt_raise_2.py` getestet hatten.
- Diese neue Implementierung hat genau so einen Fehler ausgelöst `ArithmeticErrors`.
- Diese hier tut das aber nicht.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Das ist genau die gleiche Bedingung, mit der wir unsere neue `sqrt` Implementierung, die selber Ausnahmen auslöst, in `test_sqrt_raise_2.py` getestet hatten.
- Diese neue Implementierung hat genau so einen Fehler ausgelöst `ArithmeticErrors`.
- Diese hier tut das aber nicht.
- Natürlich wissen wir bereits, dass `sqrt(10 ** 320)` einen `OverflowError` ausführt.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Diese neue Implementierung hat genau so einen Fehler ausgelöst `ArithmeticErrors`.
- Diese hier tut das aber nicht.
- Natürlich wissen wir bereits, dass `sqrt(10 ** 320)` einen `OverflowError` ausführt.
- Wir wissen, dass ein `OverflowErrors` ein Spezialfall von `ArithmeticErrors` ist.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Diese hier tut das aber nicht.
- Natürlich wissen wir bereits, dass `sqrt(10 ** 320)` einen `OverflowError` ausführt.
- Wir wissen, dass ein `OverflowErrors` ein Spezialfall von `ArithmeticErrors` ist.
- Allerdings wissen wir auch, dass die Fehlermeldung im Ausnahme-Objekt nicht zu dem regex `"sqrt.* is not permitted."` passt.

```python
"""Testing the square root implementation that does not raise errors."""

from math import isclose   # Checks if two float numbers are similar.
from pytest import raises  # Expects that a certain Exception is raised.


def sqrt(number: float) -> float:
    """
    Compute the square root of a `number`, but do not raise errors.

    :param number: The number to compute the square root of.
    :return: A value `v` such that `v * v == number`.
    """
    if number <= 0.0:  # Fix for the special case `0`:
        return 0.0  # We return 0; for now, we ignore negative values.
    guess: float = 1.0       # This will hold the current guess.
    old_guess: float = 0.0  # 0.0 is just a dummy value != guess.
    while not isclose(old_guess, guess):   # Repeat until no change.
        old_guess = guess   # The current guess becomes the old guess.
        guess = 0.5 * (guess + number / guess)  # The new guess.
    return guess


def test_raises_arithmetic_error_1():
    """Check that `ArithmeticError` is raised for negative input."""
    with raises(ArithmeticError):  # We can also test without `match`.
        sqrt(-1.0)  # This is not permitted, but no Exception is raised.


def test_raises_overflow_error():
    """Check that large integers cause `OverflowError`s."""
    with raises(OverflowError, match="int too large.*"):  # This works.
        sqrt(10 ** 320)  # Raises OverflowError with right message.


def test_raises_arithmetic_error_2():
    """Check that large integers cause `ArithmeticError`."""
    with raises(ArithmeticError):  # ArithmeticError, any message
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors -> OK.


def test_raises_arithmetic_error_3():
    """Check that an Arithmetic error with (wrong) message is raise."""
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
        sqrt(10 ** 320)  # OverflowErrors are ArithmeticErrors.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Natürlich wissen wir bereits, dass `sqrt(10 ** 320)` einen `OverflowError` ausführt.
- Wir wissen, dass ein `OverflowErrors` ein Spezialfall von `ArithmeticErrors` ist.
- Allerdings wissen wir auch, dass die Fehlermeldung im Ausnahme-Objekt nicht zu dem regex `"sqrt.* is not permitted."` passt.
- Deshalb schlägt der Test fehl.

```
$ pytest --timeout=10 --no-header --tb=short test_sqrt.py
========================== test session starts
    ↪ ==========================
collected 4 items
test_sqrt.py F..F                                                        [100%]
================================ FAILURES
    ↪ ================================
____________________ test_raises_arithmetic_error_1
    ↪ ____________________
test_sqrt.py:26: in test_raises_arithmetic_error_1
    with raises(ArithmeticError):  # We can also test without `match`.
         ^^^^^^^^^^^^^^^^^^^^^^^
E   Failed: DID NOT RAISE ArithmeticError
____________________ test_raises_arithmetic_error_3
    ↪ ____________________
test_sqrt.py:44: in test_raises_arithmetic_error_3
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
E   AssertionError: Regex pattern did not match.
E     Expected regex: 'sqrt.* is not permitted.'
E     Actual message: 'int too large to convert to float'
======================== short test summary info
    ↪ ========================
FAILED test_sqrt.py::test_raises_arithmetic_error_1 - Failed: DID NOT
    ↪ RAISE ArithmeticError
FAILED test_sqrt.py::test_raises_arithmetic_error_3 - AssertionError:
    ↪ Regex pattern did not match.
  Expected regex: 'sqrt.* is not permitted.'
  Actual message: 'int too large to convert to float'
====================== 2 failed, 2 passed in 0.02s
    ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Beispiel: Alte `sqrt`-Funktion Testen

- Wir wissen, dass ein `OverflowErrors` ein Spezialfall von `ArithmeticErrors` ist.
- Allerdings wissen wir auch, dass die Fehlermeldung im Ausnahme-Objekt nicht zu dem regex `"sqrt.* is not permitted."` passt.
- Deshalb schlägt der Test fehl.
- Die Ausgabe erklärt uns das klar.

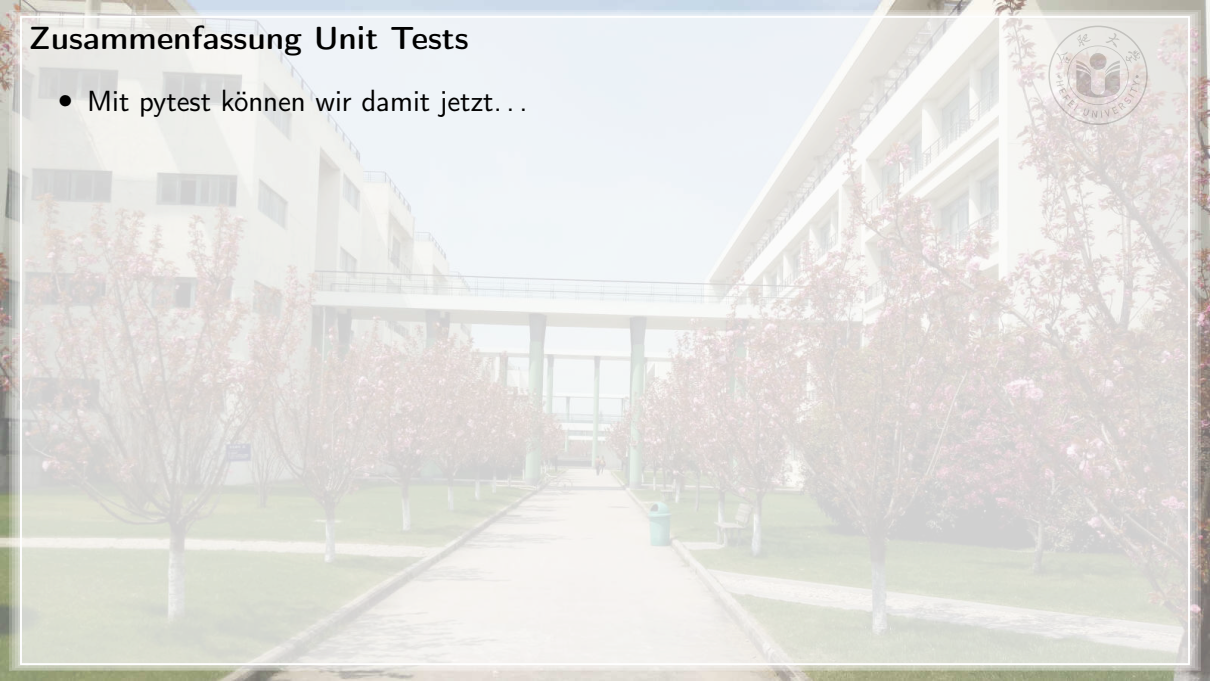
```
$ pytest --timeout=10 --no-header --tb=short test_sqrt.py
========================== test session starts
   ↪ ==========================
collected 4 items
test_sqrt.py F..F                                                        [100%]
================================== FAILURES
   ↪ ==================================
____________________ test_raises_arithmetic_error_1
   ↪ _____________________
test_sqrt.py:26: in test_raises_arithmetic_error_1
    with raises(ArithmeticError):  # We can also test without `match`.
         ^^^^^^^^^^^^^^^^^^^^^^^
E   Failed: DID NOT RAISE ArithmeticError
____________________ test_raises_arithmetic_error_3
   ↪ _____________________
test_sqrt.py:44: in test_raises_arithmetic_error_3
    with raises(ArithmeticError, match="sqrt.* is not permitted."):
         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
E   AssertionError: Regex pattern did not match.
E     Expected regex: 'sqrt.* is not permitted.'
E     Actual message: 'int too large to convert to float'
======================== short test summary info
   ↪ ========================
FAILED test_sqrt.py::test_raises_arithmetic_error_1 - Failed: DID NOT
   ↪ RAISE ArithmeticError
FAILED test_sqrt.py::test_raises_arithmetic_error_3 - AssertionError:
   ↪ Regex pattern did not match.
  Expected regex: 'sqrt.* is not permitted.'
  Actual message: 'int too large to convert to float'
====================== 2 failed, 2 passed in 0.02s
   ↪ ======================
# pytest 9.2.0 with pytest-timeout 2.4.0 failed with exit code 1.
```

Zusammenfassung



Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt...



Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet.

Zusammenfassung Unit Tests



- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet. Wenn ja: gut; wenn nein: Fehlschlag.

Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet. Wenn ja: gut; wenn nein: Fehlschlag.
  2. Testen, ob eine Funktion eine unerwartete Ausnahmen für ausgewählte korrekte Eingabedaten auslöst.

Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet. Wenn ja: gut; wenn nein: Fehlschlag.
  2. Testen, ob eine Funktion eine unerwartete Ausnahmen für ausgewählte korrekte Eingabedaten auslöst. Wenn ja: Fehlschlag; wenn nein: gut.

Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet. Wenn ja: gut; wenn nein: Fehlschlag.
  2. Testen, ob eine Funktion eine unerwartete Ausnahmen für ausgewählte korrekte Eingabedaten auslöst. Wenn ja: Fehlschlag; wenn nein: gut.
  3. Die erwarteten Ausnahmen für ausgewählte (falsche) Eingaben auslöst.

Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet. Wenn ja: gut; wenn nein: Fehlschlag.
  2. Testen, ob eine Funktion eine unerwartete Ausnahmen für ausgewählte korrekte Eingabedaten auslöst. Wenn ja: Fehlschlag; wenn nein: gut.
  3. Die erwarteten Ausnahmen für ausgewählte (falsche) Eingaben auslöst. Wenn ja: gut; wenn nein: Fehlschlag.

Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet. Wenn ja: gut; wenn nein: Fehlschlag.
  2. Testen, ob eine Funktion eine unerwartete Ausnahmen für ausgewählte korrekte Eingabedaten auslöst. Wenn ja: Fehlschlag; wenn nein: gut.
  3. Die erwarteten Ausnahmen für ausgewählte (falsche) Eingaben auslöst. Wenn ja: gut; wenn nein: Fehlschlag.
- Wir können also nun sowohl die erwarteten, korrekten Benutzung unserer Funktion testen, als auch prüfen, ob sie korrekt Ausnahmen bei falscher Benutzung generiert.

Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet. Wenn ja: gut; wenn nein: Fehlschlag.
  2. Testen, ob eine Funktion eine unerwartete Ausnahmen für ausgewählte korrekte Eingabedaten auslöst. Wenn ja: Fehlschlag; wenn nein: gut.
  3. Die erwarteten Ausnahmen für ausgewählte (falsche) Eingaben auslöst. Wenn ja: gut; wenn nein: Fehlschlag.
- Wir können also nun sowohl die erwarteten, korrekten Benutzung unserer Funktion testen, als auch prüfen, ob sie korrekt Ausnahmen bei falscher Benutzung generiert.
- Wir können zuversichtlich sein, dass unser Kode keinen Schaden anrichtet, weder durch Fehler, die wir beim Programmieren gemacht haben, noch durch falsche Benutzung durch andere Programmierer.

Zusammenfassung Unit Tests

- Mit pytest können wir damit jetzt:
  1. Testen ob eine Funktion für ausgewählte korrekte Eingabedaten die richtigen Ausgabedaten berechnet. Wenn ja: gut; wenn nein: Fehlschlag.
  2. Testen, ob eine Funktion eine unerwartete Ausnahmen für ausgewählte korrekte Eingabedaten auslöst. Wenn ja: Fehlschlag; wenn nein: gut.
  3. Die erwarteten Ausnahmen für ausgewählte (falsche) Eingaben auslöst. Wenn ja: gut; wenn nein: Fehlschlag.
- Wir können also nun sowohl die erwarteten, korrekten Benutzung unserer Funktion testen, als auch prüfen, ob sie korrekt Ausnahmen bei falscher Benutzung generiert.
- Wir können zuversichtlich sein, dass unser Kode keinen Schaden anrichtet, weder durch Fehler, die wir beim Programmieren gemacht haben, noch durch falsche Benutzung durch andere Programmierer.
- Gute Unit Tests gehen Hand-in-Hand mit guter Dokumentation, denn gute Docstrings reduzieren die Chance, das jemand unseren Kode überhaupt erst falsch verwendet.

Zusammenfassung Unit Tests



- Wir können also nun sowohl die erwarteten, korrekten Benutzung unserer Funktion testen, als auch prüfen, ob sie korrekt Ausnahmen bei falscher Benutzung generiert.
- Wir können zuversichtlich sein, dass unser Kode keinen Schaden anrichtet, weder durch Fehler, die wir beim Programmieren gemacht haben, noch durch falsche Benutzung durch andere Programmierer.
- Gute Unit Tests gehen Hand-in-Hand mit guter Dokumentation, denn gute Docstrings reduzieren die Chance, das jemand unseren Kode überhaupt erst falsch verwendet.

Gute Praxis

Es ist wichtig, sowohl die richtige Benutzung unserer Funktionen mit Test Cases zu prüfen als auch die falsche Benutzung mit z. B. falschen Argumenten.

Zusammenfassung Unit Tests



- Wir können also nun sowohl die erwarteten, korrekten Benutzung unserer Funktion testen, als auch prüfen, ob sie korrekt Ausnahmen bei falscher Benutzung generiert.
- Wir können zuversichtlich sein, dass unser Kode keinen Schaden anrichtet, weder durch Fehler, die wir beim Programmieren gemacht haben, noch durch falsche Benutzung durch andere Programmierer.
- Gute Unit Tests gehen Hand-in-Hand mit guter Dokumentation, denn gute Docstrings reduzieren die Chance, das jemand unseren Kode überhaupt erst falsch verwendet.

Gute Praxis

Es ist wichtig, sowohl die richtige Benutzung unserer Funktionen mit Test Cases zu prüfen als auch die falsche Benutzung mit z. B. falschen Argumenten. Bei falscher Benutzung sollte unsere Funktion Ausnahmen auslösen und unsere Unit Tests sollten Prüfen, ob sie das auch wirklich tut.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.
- Vielleicht wurden falsche Daten an unser Program übergeben.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.
- Vielleicht wurden falsche Daten an unser Program übergeben.
- In diesem Fall sollte unser Programm mit einer klaren Fehlermeldung abbrechen.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.
- Vielleicht wurden falsche Daten an unser Program übergeben.
- In diesem Fall sollte unser Programm mit einer klaren Fehlermeldung abbrechen.
- Vielleicht enstanden die Probleme aus einem Programmierfehler heraus.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.
- Vielleicht wurden falsche Daten an unser Program übergeben.
- In diesem Fall sollte unser Programm mit einer klaren Fehlermeldung abbrechen.
- Vielleicht enstanden die Probleme aus einem Programmierfehler heraus.
- Vielleicht hat jemand unsere Funktion verwendet und aus einem Mißverständnis heraus, falsche Argumente übergenen.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.
- Vielleicht wurden falsche Daten an unser Program übergeben.
- In diesem Fall sollte unser Programm mit einer klaren Fehlermeldung abbrechen.
- Vielleicht enstanden die Probleme aus einem Programmierfehler heraus.
- Vielleicht hat jemand unsere Funktion verwendet und aus einem Mißverständnis heraus, falsche Argumente übergenen.
- Dann sollte unsere Funktion mit einer klaren Fehlermeldung fehlschlagen.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.
- Vielleicht wurden falsche Daten an unser Program übergeben.
- In diesem Fall sollte unser Programm mit einer klaren Fehlermeldung abbrechen.
- Vielleicht enstanden die Probleme aus einem Programmierfehler heraus.
- Vielleicht hat jemand unsere Funktion verwendet und aus einem Mißverständnis heraus, falsche Argumente übergenen.
- Dann sollte unsere Funktion mit einer klaren Fehlermeldung fehlschlagen.
- Das Abbrechen des Kontrollflusses durch Ausnahmen ist sehr oft eine gute und wichtige Idee.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.
- Vielleicht wurden falsche Daten an unser Program übergeben.
- In diesem Fall sollte unser Programm mit einer klaren Fehlermeldung abbrechen.
- Vielleicht enstanden die Probleme aus einem Programmierfehler heraus.
- Vielleicht hat jemand unsere Funktion verwendet und aus einem Mißverständnis heraus, falsche Argumente übergenen.
- Dann sollte unsere Funktion mit einer klaren Fehlermeldung fehlschlagen.
- Das Abbrechen des Kontrollflusses durch Ausnahmen ist sehr oft eine gute und wichtige Idee.
- Es zeigt allen Benutzern und Programmierern an, dass etwas schief gegangen ist, das wir aktiv etwas ändern müssen, um den Fehler zu beheben.

Zusammenfassung: Ausnahmen auslösen



- Wir haben nun ein weiteres wichtiges Thema gelernt: Wie man mit Fehlern umgeht.
- Es kann viele Gründe für Fehler geben.
- Vielleicht wurden falsche Daten an unser Program übergeben.
- In diesem Fall sollte unser Programm mit einer klaren Fehlermeldung abbrechen.
- Vielleicht enstanden die Probleme aus einem Programmierfehler heraus.
- Vielleicht hat jemand unsere Funktion verwendet und aus einem Mißverständnis heraus, falsche Argumente übergenen.
- Dann sollte unsere Funktion mit einer klaren Fehlermeldung fehlschlagen.
- Das Abbrechen des Kontrollflusses durch Ausnahmen ist sehr oft eine gute und wichtige Idee.
- Es zeigt allen Benutzern und Programmierern an, dass etwas schief gegangen ist, das wir aktiv etwas ändern müssen, um den Fehler zu beheben.
- Ansätze, die Fehler ignorieren führen nur zu schlimmeren Fehlern später.

Zusammenfassung: Ausnahmen behandeln



- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.

Zusammenfassung: Ausnahmen behandeln



- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.
- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.

Zusammenfassung: Ausnahmen behandeln



- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.
- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.
- Wir sollten den Benutzer informieren, müssen aber nicht unseren ganzen Prozess abschießen.

Zusammenfassung: Ausnahmen behandeln



- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.
- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.
- Wir sollten den Benutzer informieren, müssen aber nicht unseren ganzen Prozess abschießen.
- Dafür gibt es z. B. den `except`-Blocks, mit dem wir bestimmte ausgewählte Ausnahmen abfangen und verarbeiten können.

Zusammenfassung: Ausnahmen behandeln



- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.
- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.
- Wir sollten den Benutzer informieren, müssen aber nicht unseren ganzen Prozess abschießen.
- Dafür gibt es z. B. den `except`-Blocks, mit dem wir bestimmte ausgewählte Ausnahmen abfangen und verarbeiten können.
- Der `finally`-Block erlaubt es uns, bestimmte Operationen durchzuführen, egal ob es einen Fehler gegeben hat oder nicht.

Zusammenfassung: Ausnahmen behandeln



- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.
- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.
- Wir sollten den Benutzer informieren, müssen aber nicht unseren ganzen Prozess abschießen.
- Dafür gibt es z. B. den `except`-Blocks, mit dem wir bestimmte ausgewählte Ausnahmen abfangen und verarbeiten können.
- Der `finally`-Block erlaubt es uns, bestimmte Operationen durchzuführen, egal ob es einen Fehler gegeben hat oder nicht.
- Eine spezielle und auch elegantere Variante davon ist im Grunde der `with`-Block, der besonders dafür geeignet ist, Resourcen auch im Fehlerfall zu schließen und freizugeben.

Zusammenfassung: Ausnahmen behandeln



- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.
- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.
- Wir sollten den Benutzer informieren, müssen aber nicht unseren ganzen Prozess abschießen.
- Dafür gibt es z. B. den `except`-Blocks, mit dem wir bestimmte ausgewählte Ausnahmen abfangen und verarbeiten können.
- Der `finally`-Block erlaubt es uns, bestimmte Operationen durchzuführen, egal ob es einen Fehler gegeben hat oder nicht.
- Eine spezielle und auch elegantere Variante davon ist im Grunde der `with`-Block, der besonders dafür geeignet ist, Resourcen auch im Fehlerfall zu schließen und freizugeben.
- Durch das Auslösen und Behandeln von Fehlern können wir robusten Kode bauen, der sich gegen falsche Benutzung schützen kann und der such auch im Fehlerfall vernünftig verhält.

Zusammenfassung: Ausnahmen behandeln



- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.
- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.
- Wir sollten den Benutzer informieren, müssen aber nicht unseren ganzen Prozess abschießen.
- Dafür gibt es z. B. den `except`-Blocks, mit dem wir bestimmte ausgewählte Ausnahmen abfangen und verarbeiten können.
- Der `finally`-Block erlaubt es uns, bestimmte Operationen durchzuführen, egal ob es einen Fehler gegeben hat oder nicht.
- Eine spezielle und auch elegantere Variante davon ist im Grunde der `with`-Block, der besonders dafür geeignet ist, Resourcen auch im Fehlerfall zu schließen und freizugeben.
- Durch das Auslösen und Behandeln von Fehlern können wir robusten Kode bauen, der sich gegen falsche Benutzung schützen kann und der such auch im Fehlerfall vernünftig verhält.
- Natürlich können wir keinen Kode robust nennen, den wir nicht getestet haben.

Zusammenfassung: Ausnahmen behandeln

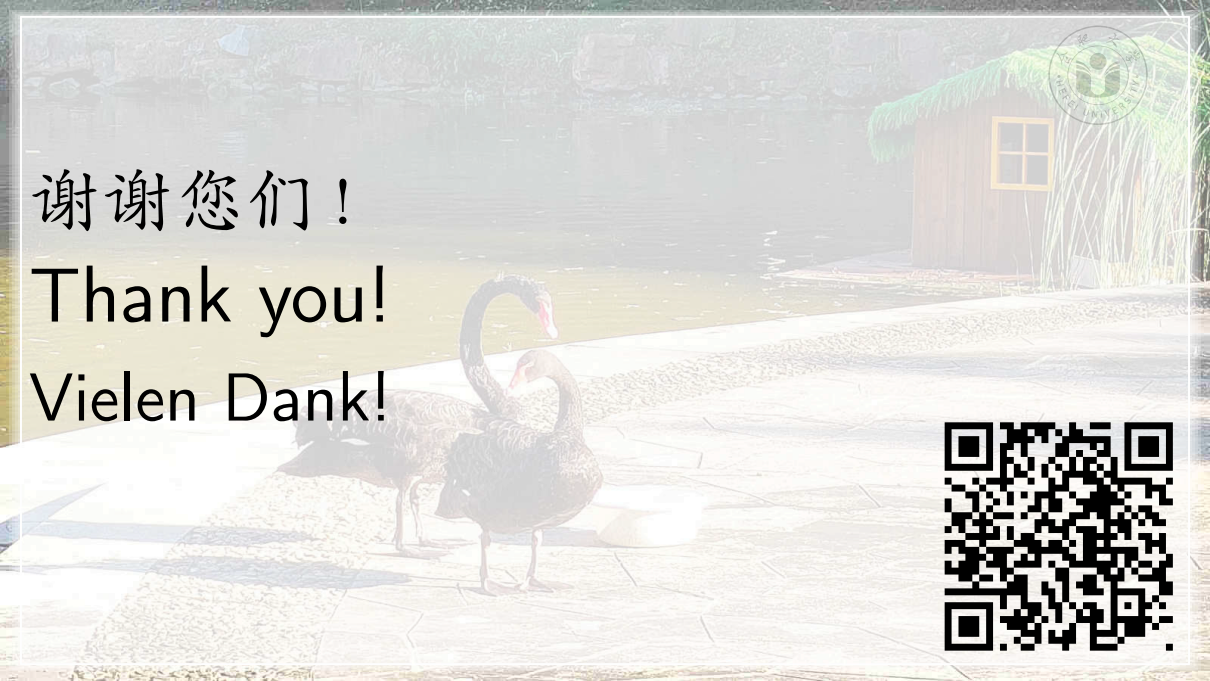


- Natürlich gibt es auch Situation, wo man Fehler elegant behandeln kann und ein Programmabsturz falsch wäre.
- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.
- Wir sollten den Benutzer informieren, müssen aber nicht unseren ganzen Prozess abschießen.
- Dafür gibt es z. B. den `except`-Blocks, mit dem wir bestimmte ausgewählte Ausnahmen abfangen und verarbeiten können.
- Der `finally`-Block erlaubt es uns, bestimmte Operationen durchzuführen, egal ob es einen Fehler gegeben hat oder nicht.
- Eine spezielle und auch elegantere Variante davon ist im Grunde der `with`-Block, der besonders dafür geeignet ist, Resourcen auch im Fehlerfall zu schließen und freizugeben.
- Durch das Auslösen und Behandeln von Fehlern können wir robusten Kode bauen, der sich gegen falsche Benutzung schützen kann und der such auch im Fehlerfall vernünftig verhält.
- Natürlich können wir keinen Kode robust nennen, den wir nicht getestet haben.
- Mit pytest können wir alle Aspekte der Fehlerbehandlung testen.

Zusammenfassung: Ausnahmen behandeln



- Wenn wir versuchen eine Datei zu löschen, die bereits gelöscht wurde, ist das kein Grund für einen Programmabsturz.
- Wir sollten den Benutzer informieren, müssen aber nicht unseren ganzen Prozess abschießen.
- Dafür gibt es z. B. den `except`-Blocks, mit dem wir bestimmte ausgewählte Ausnahmen abfangen und verarbeiten können.
- Der `finally`-Block erlaubt es uns, bestimmte Operationen durchzuführen, egal ob es einen Fehler gegeben hat oder nicht.
- Eine spezielle und auch elegantere Variante davon ist im Grunde der `with`-Block, der besonders dafür geeignet ist, Resourcen auch im Fehlerfall zu schließen und freizugeben.
- Durch das Auslösen und Behandeln von Fehlern können wir robusten Kode bauen, der sich gegen falsche Benutzung schützen kann und der such auch im Fehlerfall vernünftig verhält.
- Natürlich können wir keinen Kode robust nennen, den wir nicht getestet haben.
- Mit pytest können wir alle Aspekte der Fehlerbehandlung testen.
- Und damit sind wir am Ende der Behandlung des Fehler-bezogenen Kontrollflusses.


谢谢您们！
Thank you!
Vielen Dank!

References I

[1] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume I: Introduction to SQL Queries*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-4-0. See also[2] (siehe S. 155, 165).

[2] Adam Aspin und Karine Aspin. *Query Answers with MariaDB – Volume II: In-Depth Querying*. Tetras Publishing, Okt. 2018. ISBN: 978-1-9996172-5-7. See also[1] (siehe S. 155, 165).

[3] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 165, 166).

[4] Daniel Bartholomew. *Learning the MariaDB Ecosystem: Enterprise-level Features for Scalability and Availability*. New York, NY, USA: Apress Media, LLC, Okt. 2019. ISBN: 978-1-4842-5514-8 (siehe S. 165).

[5] Kent L. Beck. *JUnit Pocket Guide*. Sebastopol, CA, USA: O'Reilly Media, Inc., Sep. 2004. ISBN: 978-0-596-00743-0 (siehe S. 167).

[6] Tim Berners-Lee. *Re: Qualifiers on Hypertext links...* Geneva, Switzerland: World Wide Web project, European Organization for Nuclear Research (CERN) und Newsgroups: alt.hypertext, 6. Aug. 1991. URL: https://www.w3.org/People/Berners-Lee/1991/08/art-6484.txt (besucht am 2025-02-05) (siehe S. 167).

[7] Alex Berson. *Client/Server Architecture*. 2. Aufl. Computer Communications Series. New York, NY, USA: McGraw-Hill, 29. März 1996. ISBN: 978-0-07-005664-0 (siehe S. 164).

[8] Silvia Botros und Jeremy Tinley. *High Performance MySQL*. 4. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Nov. 2021. ISBN: 978-1-4920-8051-0 (siehe S. 165).

[9] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 165).

[10] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 164).

[11] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 164).

[12] Jason Cannon. *High Availability for the LAMP Stack*. Shelter Island, NY, USA: Manning Publications, Juni 2022 (siehe S. 164, 166).

References II

[13] Donald D. Chamberlin. "50 Years of Queries". *Communications of the ACM (CACM)* 67(8):110–121, Aug. 2024. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/3649887. URL: https://cacm.acm.org/research/50-years-of-queries (besucht am 2025-01-09) (siehe S. 166).

[14] 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. 166).

[15] Edgar Frank „Ted" Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM (CACM)* 13(6):377–387, Juni 1970. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/362384.362685. URL: https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf (besucht am 2025-01-05) (siehe S. 165).

[16] "`contextlib` – Utilities for `with`-Statement Contexts". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/contextlib.html (besucht am 2024-11-01) (siehe S. 14–17).

[17] *Database Language SQL*. Techn. Ber. ANSI X3.135-1986. Washington, D.C., USA: American National Standards Institute (ANSI), 1986 (siehe S. 166).

[18] Matt David und Blake Barnhill. *How to Teach People SQL*. San Francisco, CA, USA: The Data School, Chart.io, Inc., 10. Dez. 2019–10. Apr. 2023. URL: https://dataschool.com/how-to-teach-people-sql (besucht am 2025-02-27) (siehe S. 166).

[19] *Database Language SQL*. International Standard ISO 9075-1987. Geneva, Switzerland: International Organization for Standardization (ISO), 1987 (siehe S. 166).

[20] Paul Deitel, Harvey Deitel und Abbey Deitel. *Internet & World Wide WebW[: How to Program*. 5. Aufl. Hoboken, NJ, USA: Pearson Education, Inc., Nov. 2011. ISBN: 978-0-13-299045-5 (siehe S. 167).

[21] Alfredo Deza und Noah Gift. *Testing In Python*. San Francisco, CA, USA: Pragmatic AI Labs, Feb. 2020. ISBN: 979-8-6169-6064-1 (siehe S. 165).

[22] Russell J.T. Dyer. *Learning MySQL and MariaDB*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2015. ISBN: 978-1-4493-6290-4 (siehe S. 165).

References III

[23] Luca Ferrari und Enrico Pirozzi. *Learn PostgreSQL*. 2. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Okt. 2023. ISBN: 978-1-83763-564-1 (siehe S. 165).

[24] "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. 164).

[25] 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. 164).

[26] 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. 164).

[27] 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. 164).

[28] Terry Halpin und Tony Morgan. *Information Modeling and Relational Databases*. 3. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juli 2024. ISBN: 978-0-443-23791-1 (siehe S. 165).

[29] Jan L. Harrington. *Relational Database Design and Implementation*. 4. Aufl. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Apr. 2016. ISBN: 978-0-12-849902-3 (siehe S. 165).

[30] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 165).

[31] 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. 164, 166).

References IV

[32] 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. 165).

[33] *IEEE Standard for Information Technology--Portable Operating System Interfaces (POSIX(TM))--Part 2: Shell and Utilities*. IEEE Std 1003.2-1992. New York, NY, USA: Institute of Electrical and Electronics Engineers (IEEE), 23. Juni 1993. URL: https://mirror.math.princeton.edu/pub/oldlinux/Linux.old/Ref-docs/POSIX/all.pdf (besucht am 2025-03-27). Board Approved: 1992-09-17, ANSI Approved: 1993-04-05. See unapproved draft IEEE P1003.2 Draft 11.2 of 9 1991 at the url (siehe S. 24–42, 165).

[34] *Information Technology – Database Languages – SQL – Part 1: Framework (SQL/Framework), Part 1*. International Standard ISO/IEC 9075-1:2023(E), Sixth Edition, (ANSI X3.135). Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Juni 2023. URL: https://standards.iso.org/ittf/PubliclyAvailableStandards/ISO_IEC_9075-1_2023_ed_6_-_id_76583_Publication_PDF_(en).zip (besucht am 2025-01-08). Consists of several parts, see https://modern-sql.com/standard for information where to obtain them. (Siehe S. 166).

[35] Holger Krekel und pytest-Dev Team. *pytest Documentation*. Release 8.4. Freiburg, Baden-Württemberg, Germany: merlinux GmbH. URL: https://readthedocs.org/projects/pytest/downloads/pdf/latest (besucht am 2024-11-07) (siehe S. 165).

[36] Andrew M. Kuchling. *Python 3 Documentation. Regular Expression HOWTO*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/howto/regex.html (besucht am 2024-11-01) (siehe S. 24–42, 165).

[37] Jay LaCroix. *Mastering Ubuntu Server*. 4. Aufl. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2022. ISBN: 978-1-80323-424-3 (siehe S. 166).

[38] 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. 165).

[39] Luan P. Lima, Lincoln S. Rocha, Carla I. M. Bezerra und Matheus Paixão. "Assessing Exception Handling Testing Practices in Open-Source Libraries". *Empirical Software Engineering: An International Journal* 26(5:85), Juni–Sep. 2021. London, England, UK: Springer Nature Limited. ISSN: 1382-3256. doi:10.1007/s10664-021-09983-3. URL: https://arxiv.org/abs/2105.00500 (besucht am 2024-10-29). See also arXiv:2105.00500v1 [cs.SE] 2 May 2021 (siehe S. 5–12).

References V

[40] Gloria Lotha, Aakanksha Gaur, Erik Gregersen, Swati Chopra und William L. Hosch. "Client-Server Architecture". In: *Encyclopaedia Britannica*. Hrsg. von The Editors of Encyclopaedia Britannica. Chicago, IL, USA: Encyclopædia Britannica, Inc., 3. Jan. 2025. URL: https://www.britannica.com/technology/client-server-architecture (besucht am 2025-01-20) (siehe S. 164).

[41] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 165).

[42] *MariaDB Server Documentation*. Milpitas, CA, USA: MariaDB, 2025. URL: https://mariadb.com/kb/en/documentation (besucht am 2025-04-24) (siehe S. 165).

[43] 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. 164).

[44] Jim Melton und Alan R. Simon. *SQL: 1999 – Understanding Relational Language Components*. The Morgan Kaufmann Series in Data Management Systems. Burlington, MA, USA/San Mateo, CA, USA: Morgan Kaufmann Publishers, Juni 2001. ISSN: 1046-1698. ISBN: 978-1-55860-456-8 (siehe S. 166).

[45] Zsolt Nagy. *Regex Quick Syntax Reference: Understanding and Using Regular Expressions*. New York, NY, USA: Apress Media, LLC, Aug. 2018. ISBN: 978-1-4842-3876-9 (siehe S. 24–42, 165).

[46] 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. 164).

[47] Thomas Nield. *An Introduction to Regular Expressions*. Sebastopol, CA, USA: O'Reilly Media, Inc., Juni 2019. ISBN: 978-1-4920-8255-2 (siehe S. 24–42, 165).

[48] Regina O. Obe und Leo S. Hsu. *PostgreSQL: Up and Running*. 3. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., Okt. 2017. ISBN: 978-1-4919-6336-4 (siehe S. 165).

[49] 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. 167).

References VI

[50] Brian Okken. *Python Testing with pytest*. Flower Mound, TX, USA: Pragmatic Bookshelf by The Pragmatic Programmers, L.L.C., Feb. 2022. ISBN: 978-1-68050-860-4 (siehe S. 165).

[51] 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. 167).

[52] Robert Orfali, Dan Harkey und Jeri Edwards. *Client/Server Survival Guide*. 3. Aufl. Chichester, West Sussex, England, UK: John Wiley and Sons Ltd., 25. Jan. 1999. ISBN: 978-0-471-31615-2 (siehe S. 164).

[53] 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. 165, 167).

[54] "POSIX Regular Expressions". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. 9.7.3. URL: https://www.postgresql.org/docs/17/functions-matching.html#FUNCTIONS-POSIX-REGEXP (besucht am 2025-02-27) (siehe S. 165).

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

[56] *PostgreSQL Essentials: Leveling Up Your Data Work*. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2024 (siehe S. 165).

[57] Abhishek Ratan, Eric Chou, Pradeeban Kathiravelu und Dr. M.O. Faruque Sarker. *Python Network Programming*. Birmingham, England, UK: Packt Publishing Ltd, Jan. 2019. ISBN: 978-1-78883-546-6 (siehe S. 164).

[58] Federico Razzoli. *Mastering MariaDB*. Birmingham, England, UK: Packt Publishing Ltd, Sep. 2014. ISBN: 978-1-78398-154-0 (siehe S. 165).

[59] "`re` – Regular Expression Operations". In: *Python 3 Documentation. The Python Standard Library*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. URL: https://docs.python.org/3/library/re.html#module-re (besucht am 2024-11-01) (siehe S. 24–42, 165).

References VII

[60] Mike Reichardt, Michael Gundall und Hans D. Schotten. "Benchmarking the Operation Times of NoSQL and MySQL Databases for Python Clients". In: *47th Annual Conference of the IEEE Industrial Electronics Society (IECON'2021*. 13.–15. Okt. 2021, Toronto, ON, Canada. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE), 2021, S. 1–8. ISSN: 2577-1647. ISBN: 978-1-6654-3554-3. doi:10.1109/IECON48115.2021.9589382 (siehe S. 165).

[61] Mark Richards und Neal Ford. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2020. ISBN: 978-1-4920-4345-4 (siehe S. 164).

[62] 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. 167).

[63] 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. 165).

[64] 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. 164).

[65] John Miles Smith und Philip Yen-Tang Chang. "Optimizing the Performance of a Relational Algebra Database Interface". *Communications of the ACM (CACM)* 18(10):568–579, Okt. 1975. New York, NY, USA: Association for Computing Machinery (ACM). ISSN: 0001-0782. doi:10.1145/361020.361025 (siehe S. 165).

[66] "SQL Commands". In: *PostgreSQL Documentation*. 17.4. The PostgreSQL Global Development Group (PGDG), 20. Feb. 2025. Kap. Part VI. Reference. URL: https://www.postgresql.org/docs/17/sql-commands.html (besucht am 2025-02-25) (siehe S. 166).

[67] Ryan K. Stephens und Ronald R. Plew. *Sams Teach Yourself SQL in 21 Days*. 4. Aufl. Sams Teach Yourself. Indianapolis, IN, USA: SAMS Technical Publishing und Hoboken, NJ, USA: Pearson Education, Inc., Okt. 2002. ISBN: 978-0-672-32451-2 (siehe S. 161, 166).

[68] Ryan K. Stephens, Ronald R. Plew, Bryan Morgan und Jeff Perkins. *SQL in 21 Tagen. Die Datenbank-Abfragesprache SQL vollständig erklärt (in 14/21 Tagen)*. 6. Aufl. Burgthann, Bayern, Germany: Markt+Technik Verlag GmbH, Feb. 1998. ISBN: 978-3-8272-2020-2. Translation of [67] (siehe S. 166).

References VIII

[69] Allen Taylor. *Introducing SQL and Relational Databases*. New York, NY, USA: Apress Media, LLC, Sep. 2018. ISBN: 978-1-4842-3841-7 (siehe S. 165, 166).

[70] Alkin Tezuysal und Ibrar Ahmed. *Database Design and Modeling with PostgreSQL and MySQL*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2024. ISBN: 978-1-80323-347-5 (siehe S. 165).

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

[72] 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. 167).

[73] 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. 165).

[74] 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. 164).

[75] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 165).

[76] Thomas Weise (汤卫思). *Databases*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2025. URL: https://thomasweise.github.io/databases (besucht am 2025-01-05) (siehe S. 164, 165).

[77] 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. 165).

References IX

[78] *What is a Relational Database?* Armonk, NY, USA: International Business Machines Corporation (IBM), 20. Okt. 2021–12. Dez. 2024. URL: https://www.ibm.com/think/topics/relational-databases (besucht am 2025-01-05) (siehe S. 165).

[79] Ulf Michael „Monty“ Widenius, David Axmark und Uppsala, Sweden: MySQL AB. *MySQL Reference Manual – Documentation from the Source*. Sebastopol, CA, USA: O'Reilly Media, Inc., 9. Juli 2002. ISBN: 978-0-596-00265-7 (siehe S. 165).

[80] Kevin Wilson. *Python Made Easy*. Birmingham, England, UK: Packt Publishing Ltd, Aug. 2024. ISBN: 978-1-83664-615-0 (siehe S. 165).

[81] Kinza Yasar und Craig S. Mullins. *Definition: Database Management System (DBMS)*. Newton, MA, USA: TechTarget, Inc., Juni 2024. URL: https://www.techtarget.com/searchdatamanagement/definition/database-management-system (besucht am 2025-01-11) (siehe S. 164).

[82] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 164).

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[10,46,82]. Learn more at `https://www.gnu.org/software/bash`.

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

client-server architecture is a system design where a central server receives requests from one or multiple clients[7,40,52,57,61]. These requests and responses are usually sent over network connections. A typical example of such a system is the World Wide Web (WWW), where web servers host websites and make them available to web browsers, the clients. Another typical example is the structure of database (DB) software, where a central server, the DBMS, offers access to the DB to the different clients. Here, the client can be some terminal software shipping with the DBMS, such as psql, or the different applications that access the DBs.

DB A *database* is an organized collection of structured information or data, typically stored electronically in a computer system. Databases are discussed in our book *Databases*[76].

DBMS A *database management system* is the software layer located between the user or application and the DB. The DBMS allows the user/application to create, read, write, update, delete, and otherwise manipulate the data in the DB[81].

docstring Docstrings are special string constants in Python that contain documentation for modules or functions[26]. They must be delimited by `"""..."""`[26,74].

f-string let you include the results of expressions in strings[11,24,25,27,43,64]. 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"..."`.

IT information technology

LAMP Stack A system setup for web applications: Linux, Apache (a web server), MySQL, and the server-side scripting language PHP[12,31].

Glossary (in English) II



Linux is the leading open source operating system, i.e., a free alternative to Microsoft Windows[3,30,63,73,75]. We recommend using it for this course, for software development, and for research. Learn more at https://www.linux.org. Its variant Ubuntu is particularly easy to use and install.

MariaDB An open source relational database management system that has forked off from MySQL[1,2,4,22,42,58]. See https://mariadb.org for more information.

Microsoft Windows is a commercial proprietary operating system[9]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

MySQL An open source relational database management system[8,22,60,70,79]. MySQL is famous for its use in the LAMP Stack. See https://www.mysql.com for more information.

PostgreSQL An open source object-relational DBMS[23,48,56,70]. See https://postgresql.org for more information.

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

pytest is a framework for writing and executing unit tests in Python[21,35,50,53,80]. Learn more at https://pytest.org.

Python The Python programming language[32,38,41,77], i.e., what you will learn about in our book[77]. Learn more at https://python.org.

regex A *Regular Expression*, often called „regex“ for short, is a sequence of characters that defines a search pattern for text strings[33,36,45,47]. In Python, the `re` module offers functionality work with regular expressions[36,59]. In PostgreSQL, regex-based pattern matching is supported as well[54].

relational database A relational DB is a database that organizes data into rows (tuples, records) and columns (attributes), which collectively form tables (relations) where the data points are related to each other[15,28,29,65,69,76,78].

Glossary (in English) III



server In a client-server architecture, the server is a process that fulfills the requests of the clients. It usually waits for incoming communication carring the requests from the clients. For each request, it takes the necessary actions, performs the required computations, and then sends a response with the result of the request. Typical examples of servers are web servers[12] in the internet as well as DBMSes. It is also common to refer to the computer running the server processes as server as well, i.e., to call it the „server computer“[37].

SQL The *Structured Query Language* is basically a programming language for querying and manipulating relational databases[13,17–19,34,44,66–69]. It is understood by many DBMSes. You find the Structured Query Language (SQL) commands supported by PostgreSQL in the reference[66].

(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[3,14]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

Ubuntu is a variant of the open source operating system Linux[14,31]. We recommend that you use this operating system to follow this class, for software development, and for research. Learn more at https://ubuntu.com. If you are in China, you can download it from https://mirrors.ustc.edu.cn/ubuntu-releases.

Glossary (in English) IV



unit test Software development is centered around creating the program code of an application, library, or otherwise useful system. A *unit test* is an *additional* code fragment that is not part of that productive code. It exists to execute (a part of) the productive code in a certain scenario (e.g., with specific parameters), to observe the behavior of that code, and to compare whether this behavior meets the specification[5,49,51,53,62,72]. 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.

WWW World Wide Web[6,20]