合肥大學
HEFEI UNIVERSITY



Programming with Python

32. Ausnahmen verarbeiten

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. Der `try...except` Block
3. Verschachtelte Fehlerbehandlung
4. Der `try-except-else`-Block
5. Der `try-finally`-Block
6. `with`-Block und Context Manager
7. Zusammenfassung



Einleitung

Einleitung

- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.
- Natürlich wollen wir nicht dass jede unerwartete Fehlersituation immer sofort unser Programm zum Absturz bringt.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.
- Natürlich wollen wir nicht dass jede unerwartete Fehlersituation immer sofort unser Programm zum Absturz bringt.
- Nehmen wir an, dass wir ein Malprogramm entwickelt haben. Ein Benutzer hat ein Bild gemalt und will es speichern, gibt jedoch aus Versehen einen falschen Pfad an.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.
- Natürlich wollen wir nicht dass jede unerwartete Fehlersituation immer sofort unser Programm zum Absturz bringt.
- Nehmen wir an, dass wir ein Malprogramm entwickelt haben. Ein Benutzer hat ein Bild gemalt und will es speichern, gibt jedoch aus Versehen einen falschen Pfad an.
- Es wäre dann ärgerlich, wenn unser Programm sofort abstürzen würde.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.
- Natürlich wollen wir nicht dass jede unerwartete Fehlersituation immer sofort unser Programm zum Absturz bringt.
- Nehmen wir an, dass wir ein Malprogramm entwickelt haben. Ein Benutzer hat ein Bild gemalt und will es speichern, gibt jedoch aus Versehen einen falschen Pfad an.
- Es wäre dann ärgerlich, wenn unser Programm sofort abstürzen würde.
- Die Funktion zum Bilder-speichern würde mit einer Ausnahme fehlschlagen.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.
- Natürlich wollen wir nicht dass jede unerwartete Fehlersituation immer sofort unser Programm zum Absturz bringt.
- Nehmen wir an, dass wir ein Malprogramm entwickelt haben. Ein Benutzer hat ein Bild gemalt und will es speichern, gibt jedoch aus Versehen einen falschen Pfad an.
- Es wäre dann ärgerlich, wenn unser Programm sofort abstürzen würde.
- Die Funktion zum Bilder-speichern würde mit einer Ausnahme fehlschlagen.
- Der aktuelle Zweig es Kontrollflusses, der vielleicht das Bild als „gespeichert“ markieren und den Undo-Puffer leeren würde, muss sofort verlassen werden.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.
- Natürlich wollen wir nicht dass jede unerwartete Fehlersituation immer sofort unser Programm zum Absturz bringt.
- Nehmen wir an, dass wir ein Malprogramm entwickelt haben. Ein Benutzer hat ein Bild gemalt und will es speichern, gibt jedoch aus Versehen einen falschen Pfad an.
- Es wäre dann ärgerlich, wenn unser Programm sofort abstürzen würde.
- Die Funktion zum Bilder-speichern würde mit einer Ausnahme fehlschlagen.
- Der aktuelle Zweig es Kontrollflusses, der vielleicht das Bild als „gespeichert“ markieren und den Undo-Puffer leeren würde, muss sofort verlassen werden.
- Auf einer höheren Ebene sollte die Exception dann verarbeitet werden.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.
- Natürlich wollen wir nicht dass jede unerwartete Fehlersituation immer sofort unser Programm zum Absturz bringt.
- Nehmen wir an, dass wir ein Malprogramm entwickelt haben. Ein Benutzer hat ein Bild gemalt und will es speichern, gibt jedoch aus Versehen einen falschen Pfad an.
- Es wäre dann ärgerlich, wenn unser Programm sofort abstürzen würde.
- Die Funktion zum Bilder-speichern würde mit einer Ausnahme fehlschlagen.
- Der aktuelle Zweig es Kontrollflusses, der vielleicht das Bild als „gespeichert“ markieren und den Undo-Puffer leeren würde, muss sofort verlassen werden.
- Auf einer höheren Ebene sollte die Exception dann verarbeitet werden.
- Dem Benutzer wird ein Dialog mit der Fehlermeldung angezeigt. Dann kann er normal weiterarbeiten.

Einleitung



- Wir haben nun gelernt, wie wir Ausnahmen auslösen können.
- Die Ausnahmen, die wir ausgelöst haben, haben unsere Prozesse dazu gebracht, eine Fehlermeldung auf dem standard error stream (`stderr`) auszugeben und dann zu terminieren.
- Natürlich wollen wir nicht dass jede unerwartete Fehlersituation immer sofort unser Programm zum Absturz bringt.
- Nehmen wir an, dass wir ein Malprogramm entwickelt haben. Ein Benutzer hat ein Bild gemalt und will es speichern, gibt jedoch aus Versehen einen falschen Pfad an.
- Es wäre dann ärgerlich, wenn unser Programm sofort abstürzen würde.
- Die Funktion zum Bilder-speichern würde mit einer Ausnahme fehlschlagen.
- Der aktuelle Zweig es Kontrollflusses, der vielleicht das Bild als „gespeichert“ markieren und den Undo-Puffer leeren würde, muss sofort verlassen werden.
- Auf einer höheren Ebene sollte die Exception dann verarbeitet werden.
- Dem Benutzer wird ein Dialog mit der Fehlermeldung angezeigt. Dann kann er normal weiterarbeiten.
- Nun lernen wir, wie man das macht, wie man Ausnahmen verarbeitet.

Der `try...except` Block



Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
        statement 1    # Code that may raise an exception or that
        statement 2    # calls a function that may rise one.
        #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
        statements processing ex1
        # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
        statements processing exceptions of type ExceptionType2
        # Code the handles exceptions of type ExceptionType2 that are not
        # instances of ExceptionType1. Notice that we do not necessarily
        # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.
- Wir schreiben `try:` und danach platzieren wir den Kode, der eine Ausnahme auslösen kann, in einen eingerückten Block.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
    statement 1    # Code that may raise an exception or that
    statement 2    # calls a function that may rise one.
    #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.
- Wir schreiben `try:` und danach platzieren wir den Kode, der eine Ausnahme auslösen kann, in einen eingerückten Block.
- Nach dem Block kommen die Handler für spezifische Ausnahmetypen.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
    statement 1    # Code that may raise an exception or that
    statement 2    # calls a function that may rise one.
    #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.
- Wenn der `try`-Block einen `ArithmeticError` auslösen kann (aus einem Grund den wir vernünftig behandeln können!), dann schreiben wir `except ArithmeticError as ae:`.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
    statement 1    # Code that may raise an exception or that
    statement 2    # calls a function that may rise one.
    #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.
- Wenn der `try`-Block einen `ArithmeticError` auslösen kann (aus einem Grund den wir vernünftig behandeln können!), dann schreiben wir `except ArithmeticError as ae:`.
- Der Kode im `except`-Block wird *nur* dann ausgeführt, wenn tatsächlich irgendwo im `try`-Block ein `ArithmeticError` ausgelöst wurde.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
    statement 1    # Code that may raise an exception or that
    statement 2    # calls a function that may rise one.
    #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.
- Der Kode im `except`-Block wird *nur* dann ausgeführt, wenn tatsächlich irgendwo im `try`-Block ein `ArithmeticError` ausgelöst wurde.
- In diesem Fall steht der `ArithmeticError` dann als Variable `ae` in diesem Block bereit.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
    statement 1    # Code that may raise an exception or that
    statement 2    # calls a function that may rise one.
    #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.
- In diesem Fall steht der `ArithmeticError` dann als Variable `ae` in diesem Block bereit.
- Natürlich kann Kode beliebig viele `Exceptions` auslösen, weshalb wir auch mehrere `except`-Blöcke haben können.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
    statement 1    # Code that may raise an exception or that
    statement 2    # calls a function that may rise one.
    #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.
- Natürlich kann Kode beliebig viele `Exceptions` auslösen, weshalb wir auch mehrere `except`-Blöcke haben können.
- Wir müssen auch Ausnahmen nicht unbedingt mit „`as varName`“ in Variablen speichern.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
    statement 1    # Code that may raise an exception or that
    statement 2    # calls a function that may rise one.
    #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try...except` Block

- Der `try`-`except` Block existiert als primäre Methode, um bestimmte Fehler aufzufangen und zu verarbeiten.
- Wir müssen auch Ausnahmen nicht unbedingt mit „`as varName`“ in Variablen speichern.
- Wenn wir auf die `Exception`-Daten nicht zugreifen wollen, können wir das auch weglassen.

```
"""The syntax of a try-except statement in Python."""

try:    # Begin the try-except block.
    statement 1    # Code that may raise an exception or that
    statement 2    # calls a function that may rise one.
    #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
except ExceptionType2:    # Another exception type (optional).
    statements processing exceptions of type ExceptionType2
    # Code the handles exceptions of type ExceptionType2 that are not
    # instances of ExceptionType1. Notice that we do not necessarily
    # need to store the exceptions in variables with, like "as ex2".

next statement    # Executed only if there are no uncaught Exceptions.
```

Beispiel: String.index

- Probieren wir das mal aus.




```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Probieren wir das mal aus.
- Wir haben bereits gelernt, dass Strings `r` die Methode `r.find(s)` anbieten, die einen String `s` im String `r` suchen und den ersten Index, wo `s` auftaucht, zurückliefern.

```
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Probieren wir das mal aus.
- Wir haben bereits gelernt, dass Strings `r` die Methode `r.find(s)` anbieten, die einen String `s` im String `r` suchen und den ersten Index, wo `s` auftaucht, zurückliefern.
- Wenn `s` nicht in `r` gefunden wird, dann wird `-1` zurückgegeben.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Probieren wir das mal aus.
- Wir haben bereits gelernt, dass Strings `r` die Methode `r.find(s)` anbieten, die einen String `s` im String `r` suchen und den ersten Index, wo `s` auftaucht, zurückliefern.
- Wenn `s` nicht in `r` gefunden wird, dann wird `-1` zurückgegeben.
- Die Operation `r.index(s)` funktioniert im Grunde genauso, nur dass sie einen `ValueError` auslöst, wenn `s` nicht in `r` gefunden wird.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- Probieren wir das mal aus.
- Wir haben bereits gelernt, dass Strings `r` die Methode `r.find(s)` anbieten, die einen String `s` im String `r` suchen und den ersten Index, wo `s` auftaucht, zurückliefern.
- Wenn `s` nicht in `r` gefunden wird, dann wird `-1` zurückgegeben.
- Die Operation `r.index(s)` funktioniert im Grunde genauso, nur dass sie einen `ValueError` auslöst, wenn `s` nicht in `r` gefunden wird.
- Das ist sinnvoll in Fällen, wo wir wissen, dass `s` in `r` seien muss und wenn nicht, dann ist es ein Fehler.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Wenn `s` nicht in `r` gefunden wird, dann wird `-1` zurückgegeben.
- Die Operation `r.index(s)` funktioniert im Grunde genauso, nur dass sie einen `ValueError` auslöst, wenn `s` nicht in `r` gefunden wird.
- Das ist sinnvoll in Fällen, wo wir wissen, dass `s` in `r` seien muss und wenn nicht, dann ist es ein Fehler.
- Wir können auch das Ergebnis von `r.index(s)` direkt als Index in `r` verwenden, also sowas wie `r[r.index(s)]` machen.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Wenn `s` nicht in `r` gefunden wird, dann wird `-1` zurückgegeben.
- Die Operation `r.index(s)` funktioniert im Grunde genauso, nur dass sie einen `ValueError` auslöst, wenn `s` nicht in `r` gefunden wird.
- Das ist sinnvoll in Fällen, wo wir wissen, dass `s` in `r` seien muss und wenn nicht, dann ist es ein Fehler.
- Wir können auch das Ergebnis von `r.index(s)` direkt als Index in `r` verwenden, also sowas wie `r[r.index(s)]` machen.
- Mit `r.find(s)` können wir das nicht, weil `-1` ja auch ein gültiger Index für einen String ist...

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Das ist sinnvoll in Fällen, wo wir wissen, dass `s` in `r` seien muss und wenn nicht, dann ist es ein Fehler.
- Wir können auch das Ergebnis von `r.index(s)` direkt als Index in `r` verwenden, also sowas wie `r[r.index(s)]` machen.
- Mit `r.find(s)` können wir das nicht, weil `-1` ja auch ein gültiger Index für einen String ist...
- Im Beispiel `try_except_str_index.py` erforschen wir also diese Funktion.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- Das ist sinnvoll in Fällen, wo wir wissen, dass `s` in `r` seien muss und wenn nicht, dann ist es ein Fehler.
- Wir können auch das Ergebnis von `r.index(s)` direkt als Index in `r` verwenden, also sowas wie `r[r.index(s)]` machen.
- Mit `r.find(s)` können wir das nicht, weil `-1` ja auch ein gültiger Index für einen String ist...
- Im Beispiel `try_except_str_index.py` erforschen wir also diese Funktion.
- Unser String `r` ist `"Hello World!"`.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Wir können auch das Ergebnis von `r.index(s)` direkt als Index in `r` verwenden, also sowas wie `r[r.index(s)]` machen.
- Mit `r.find(s)` können wir das nicht, weil `-1` ja auch ein gültiger Index für einen String ist...
- Im Beispiel `try_except_str_index.py` erforschen wir also diese Funktion.
- Unser String `r` ist `"Hello World!"`.
- Im `try`-Block ist eine `for`-Schleife, die eine Variable `s` nacheinander drei Werte annehmen lässt.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- Mit `r.find(s)` können wir das nicht, weil `-1` ja auch ein gültiger Index für einen String ist...
- Im Beispiel `try_except_str_index.py` erforschen wir also diese Funktion.
- Unser String `r` ist `"Hello World!"`.
- Im `try`-Block ist eine `for`-Schleife, die eine Variable `s` nacheinander drei Werte annehmen lässt.
- In der ersten Iteration gilt `s = "Hello"` und `r.index(s)` ergibt `0`.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Im Beispiel `try_except_str_index.py` erforschen wir also diese Funktion.
- Unser String `r` ist `"Hello World!"`.
- Im `try`-Block ist eine `for`-Schleife, die eine Variable `s` nacheinander drei Werte annehmen lässt.
- In der ersten Iteration gilt `s = "Hello"` und `r.index(s)` ergibt `0`.
- Das wird auch ausgegeben.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Unser String `r` ist `"Hello World!"`.
- Im `try`-Block ist eine `for`-Schleife, die eine Variable `s` nacheinander drei Werte annehmen lässt.
- In der ersten Iteration gilt `s = "Hello"` und `r.index(s)` ergibt `0`.
- Das wird auch ausgegeben.
- In der zweiten Iteration ist `s = "world"` was nicht gefunden wird, weil String-Suche case-sensitive ist, also Groß- und Kleinschreibung beachtet.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Im `try`-Block ist eine `for`-Schleife, die eine Variable `s` nacheinander drei Werte annehmen lässt.
- In der ersten Iteration gilt `s = "Hello"` und `r.index(s)` ergibt `0`.
- Das wird auch ausgegeben.
- In der zweiten Iteration ist `s = "world"` was nicht gefunden wird, weil String-Suche case-sensitive ist, also Groß- und Kleinschreibung beachtet.
- `r.index(s)` löst also eine Ausnahme aus.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- In der ersten Iteration gilt `s = "Hello"` und `r.index(s)` ergibt `0`.
- Das wird auch ausgegeben.
- In der zweiten Iteration ist `s = "world"` was nicht gefunden wird, weil String-Suche case-sensitive ist, also Groß- und Kleinschreibung beachtet.
- `r.index(s)` löst also eine Ausnahme aus.
- Der `except`-Block nach dem `try`-Block wird ausgeführt, wenn irgendwo im `try`-Block ein `ValueError` ausgelöst wird.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Das wird auch ausgegeben.
- In der zweiten Iteration ist `s = "world"` was nicht gefunden wird, weil String-Suche case-sensitive ist, also Groß- und Kleinschreibung beachtet.
- `r.index(s)` löst also eine Ausnahme aus.
- Der `except`-Block nach dem `try`-Block wird ausgeführt, wenn irgendwo im `try`-Block ein `ValueError` ausgelöst wird.
- In diesem Fall wird der `ValueError` in der Variable `ve` gespeichert.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- In der zweiten Iteration ist `s = "world"` was nicht gefunden wird, weil String-Suche case-sensitive ist, also Groß- und Kleinschreibung beachtet.
- `r.index(s)` löst also eine Ausnahme aus.
- Der `except`-Block nach dem `try`-Block wird ausgeführt, wenn irgendwo im `try`-Block ein `ValueError` ausgelöst wird.
- In diesem Fall wird der `ValueError` in der Variable `ve` gespeichert.
- In diesem Block drucken wir einfach den Fehler aus.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- `r.index(s)` löst also eine Ausnahme aus.
- Der `except`-Block nach dem `try`-Block wird ausgeführt, wenn irgendwo im `try`-Block ein `ValueError` ausgelöst wird.
- In diesem Fall wird der `ValueError` in der Variable `ve` gespeichert.
- In diesem Block drucken wir einfach den Fehler aus.
- Nach dem Block gegen wir `"The program is now finished."`.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- Der `except`-Block nach dem `try`-Block wird ausgeführt, wenn irgendwo im `try`-Block ein `ValueError` ausgelöst wird.
- In diesem Fall wird der `ValueError` in der Variable `ve` gespeichert.
- In diesem Block drucken wir einfach den Fehler aus.
- Nach dem Block gegen wir `"The program is now finished."`.
- Dieser Kode wird nur ausgeführt, wenn keine unbearbeitete Ausnahme den `try`-`except`-Block verlässt.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- In diesem Fall wird der `ValueError` in der Variable `ve` gespeichert.
- In diesem Block drucken wir einfach den Fehler aus.
- Nach dem Block gegen wir `"The program is now finished."`.
- Dieser Kode wird nur ausgeführt, wenn keine unbearbeitete Ausnahme den `try`-`except`-Block verlässt.
- In der Ausgabe sehen wir zuerst das Ergebnis der erfolgreichen Suche, dann die Ausgabe für die fehlgeschlagene Suche, und zuletzt den text `The program is now finished.`.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- Nach dem Block gegen wir `"The program is now finished."`.
- Dieser Kode wird nur ausgeführt, wenn keine unbearbeitete Ausnahme den `try`-`except`-Block verlässt.
- In der Ausgabe sehen wir zuerst das Ergebnis der erfolgreichen Suche, dann die Ausgabe für die fehlgeschlagene Suche, und zuletzt den text `The program is now finished.`.
- Wenn wir den `except`-Block nicht geschrieben hätten, dann hätte der Kontrollfluss die Schleife verlassen, einen Stack-Trace ausgegeben, und das Programm abgebrochen.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index



- In der Ausgabe sehen wir zuerst das Ergebnis der erfolgreichen Suche, dann die Ausgabe für die fehlgeschlagene Suche, und zuletzt den text `The program is now finished.`.
- Wenn wir den `except`-Block nicht geschrieben hätten, dann hätte der Kontrollfluss die Schleife verlassen, einen Stack-Trace ausgegeben, und das Programm abgebrochen.
- Beachten Sie, dass der drite Wert in der Schleife – `"!"` – niemals der Variable `s` zugewiesen wird.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- Wenn wir den `except`-Block nicht geschrieben hätten, dann hätte der Kontrollfluss die Schleife verlassen, einen Stack-Trace ausgegeben, und das Programm abgebrochen.
- Beachten Sie, dass der drite Wert in der Schleife – `"!"` – niemals der Variable `s` zugewiesen wird.
- Der `try`-Block wird sofort abgebrochen, wenn die Ausnahme auftritt.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- Wenn wir den `except`-Block nicht geschrieben hätten, dann hätte der Kontrollfluss die Schleife verlassen, einen Stack-Trace ausgegeben, und das Programm abgebrochen.
- Beachten Sie, dass der drite Wert in der Schleife – `"!"` – niemals der Variable `s` zugewiesen wird.
- Der `try`-Block wird sofort abgebrochen, wenn die Ausnahme auftritt.
- Wenn die Ausnahme von einem passenden `except`-Block behandelt werden kann, dann wird dieser ausgeführt.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ `python3 try_except_str_index.py` ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- Beachten Sie, dass der drite Wert in der Schleife – `"!"` – niemals der Variable `s` zugewiesen wird.
- Der `try`-Block wird sofort abgebrochen, wenn die Ausnahme auftritt.
- Wenn die Ausnahme von einem passenden `except`-Block behandelt werden kann, dann wird dieser ausgeführt.
- Andernfalls wird der Prozess abgebrochen.

```
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Beispiel: String.index

- Beachten Sie, dass der drite Wert in der Schleife – `"!"` – niemals der Variable `s` zugewiesen wird.
- Der `try`-Block wird sofort abgebrochen, wenn die Ausnahme auftritt.
- Wenn die Ausnahme von einem passenden `except`-Block behandelt werden kann, dann wird dieser ausgeführt.
- Andernfalls wird der Prozess abgebrochen.
- So oder so, der Kode nach dem fehlgeschlagenen Befehl im `try`-Block wird nie erreicht.

```python
"""Demonstrate `try...except` by looking for text in a string."""

r: str = "Hello World!"  # This is the string we search inside.

try:  # If this block raises an error, we continue at `except`.
    for s in ["Hello", "world", "!"]:  # The strings we try to find.
        print(f"{s!r} is at index {r.index(s)}.")
except ValueError as ve:  # ValueError is raised if `s` isn't in `text`.
    print(f"Error: {ve}")  # Error, as "world" is not in "Hello World!".

print("The program is now finished.")  # We get here after except block.
```

↓ python3 try_except_str_index.py ↓

```
'Hello' is at index 0.
Error: substring not found
The program is now finished.
```

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.



Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.
- Solche Fehler sollten wir mit einer passenden `except`-Klausel abfangen und dann entsprechend verarbeiten.

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.
- Solche Fehler sollten wir mit einer passenden `except`-Klausel abfangen und dann entsprechend verarbeiten.
- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.
- Solche Fehler sollten wir mit einer passenden `except`-Klausel abfangen und dann entsprechend verarbeiten.
- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.
- So ein Fehler ist nicht OK in diesem Kontext.

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.
- Solche Fehler sollten wir mit einer passenden `except`-Klausel abfangen und dann entsprechend verarbeiten.
- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.
- So ein Fehler ist nicht OK in diesem Kontext.
- Wir sollten nur Fehler abfangen und behandeln, die im Kontext erwartbar sind.

Vernünftige Fehler



- Wir können nun alle möglichen `Exceptions` abfangen.
- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.
- Solche Fehler sollten wir mit einer passenden `except`-Klausel abfangen und dann entsprechend verarbeiten.
- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.
- So ein Fehler ist nicht OK in diesem Kontext.
- Wir sollten nur Fehler abfangen und behandeln, die im Kontext erwartbar sind.
- Jeder andere Fehler sollte dazu führen, dass unser Program abstürzt.

Vernünftige Fehler



- Es ist aber wichtig, nur solche Fehler abzufangen und zu behanden, die *vernünftig* sind.
- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.
- Solche Fehler sollten wir mit einer passenden `except`-Klausel abfangen und dann entsprechend verarbeiten.
- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.
- So ein Fehler ist nicht OK in diesem Kontext.
- Wir sollten nur Fehler abfangen und behandeln, die im Kontext erwartbar sind.
- Jeder andere Fehler sollte dazu führen, dass unser Program abstürzt.
- Ein abgestürztes Programm ist der klarste Hinweis, dass etwas Falsch ist, das Aktionen vom Benutzer erforderlich sind.

Vernünftige Fehler



- Nehmen wir an, Ihr Programm soll einen Text in eine Datei schreiben.
- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.
- Solche Fehler sollten wir mit einer passenden `except`-Klausel abfangen und dann entsprechend verarbeiten.
- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.
- So ein Fehler ist nicht OK in diesem Kontext.
- Wir sollten nur Fehler abfangen und behandeln, die im Kontext erwartbar sind.
- Jeder andere Fehler sollte dazu führen, dass unser Program abstürzt.
- Ein abgestürztes Programm ist der klarste Hinweis, dass etwas Falsch ist, das Aktionen vom Benutzer erforderlich sind.
- Die Aktion könnte ja auch sein, uns anzurufen und einen Bug zu melden.

Vernünftige Fehler



- Es ist akzeptabel, dass das mit einem System-bezogenen Fehler schief geht.
- Vielleicht ist nicht genug Platz auf dem Speicher vorhanden, oder der Benutzer hat ungenügende Zugriffsrechte, oder vielleicht war der Dateiname ungültig.
- Solche Fehler sollten wir mit einer passenden `except`-Klausel abfangen und dann entsprechend verarbeiten.
- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.
- So ein Fehler ist nicht OK in diesem Kontext.
- Wir sollten nur Fehler abfangen und behandeln, die im Kontext erwartbar sind.
- Jeder andere Fehler sollte dazu führen, dass unser Program abstürzt.
- Ein abgestürztes Programm ist der klarste Hinweis, dass etwas Falsch ist, das Aktionen vom Benutzer erforderlich sind.
- Die Aktion könnte ja auch sein, uns anzurufen und einen Bug zu melden.
- Dann können wir das Programm verbessern und den Bug fixen.

Vernünftige Fehler



- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.
- So ein Fehler ist nicht OK in diesem Kontext.
- Wir sollten nur Fehler abfangen und behandeln, die im Kontext erwartbar sind.
- Jeder andere Fehler sollte dazu führen, dass unser Program abstürzt.
- Ein abgestürztes Programm ist der klarste Hinweis, dass etwas Falsch ist, das Aktionen vom Benutzer erforderlich sind.
- Die Aktion könnte ja auch sein, uns anzurufen und einen Bug zu melden.
- Dann können wir das Programm verbessern und den Bug fixen.

Gute Praxis

Es dürfen nur die `Exceptions` von einem `except`-Block abgefangen werden, die auch vernünftig behandelt werden können[9,47].

Vernünftige Fehler



- Tritt dagegen ein `ZeroDivisionError` auf, dann bedeutet das, dass etwas völlig anderes sehr schlimm schief gegangen ist.
- So ein Fehler ist nicht OK in diesem Kontext.
- Wir sollten nur Fehler abfangen und behandeln, die im Kontext erwartbar sind.
- Jeder andere Fehler sollte dazu führen, dass unser Program abstürzt.
- Ein abgestürztes Programm ist der klarste Hinweis, dass etwas Falsch ist, das Aktionen vom Benutzer erforderlich sind.
- Die Aktion könnte ja auch sein, uns anzurufen und einen Bug zu melden.
- Dann können wir das Programm verbessern und den Bug fixen.

Gute Praxis

Es dürfen nur die `Exceptions` von einem `except`-Block abgefangen werden, die auch vernünftig behandelt werden können[9,47]. Ein `except`-Block darf nicht dafür verwendet werden, beliebige `Exceptions` abzufangen, GIGO zu implementieren, oder um Eingabedaten zu reparieren.

Kompliziertes Beispiel

- Benutzen wir noch einmal unsere `sqrt`-Funktion aus der vorigen Einheit.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Benutzen wir noch einmal unsere `sqrt`-Funktion aus der vorigen Einheit.
- Diese Funktion löst einen `ArithmeticError` aus, wenn ihr Argument nicht endlich oder negativ ist.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Benutzen wir noch einmal unsere `sqrt`-Funktion aus der vorigen Einheit.
- Diese Funktion löst einen `ArithmeticError` aus, wenn ihr Argument nicht endlich oder negativ ist.
- Dieses Mal wollen $\sqrt{\frac{1}{0}}$ berechnen, also versuchen, das Ergebnis von `sqrt(1 / 0)` in einer Variable `sqrt_of_1_div_0` zu speichern.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Benutzen wir noch einmal unsere `sqrt`-Funktion aus der vorigen Einheit.
- Diese Funktion löst einen `ArithmeticError` aus, wenn ihr Argument nicht endlich oder negativ ist.
- Dieses Mal wollen $\sqrt{\frac{1}{0}}$ berechnen, also versuchen, das Ergebnis von `sqrt(1 / 0)` in einer Variable `sqrt_of_1_div_0` zu speichern.
- Das ist natürlich Unsinn. Aber schauen wir, was passiert.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Benutzen wir noch einmal unsere `sqrt`-Funktion aus der vorigen Einheit.
- Diese Funktion löst einen `ArithmeticError` aus, wenn ihr Argument nicht endlich oder negativ ist.
- Dieses Mal wollen $\sqrt{\frac{1}{0}}$ berechnen, also versuchen, das Ergebnis von `sqrt(1 / 0)` in einer Variable `sqrt_of_1_div_0` zu speichern.
- Das ist natürlich Unsinn. Aber schauen wir, was passiert.
- Wir deklarieren zuerst die Variable als `float`.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Diese Funktion löst einen `ArithmeticError` aus, wenn ihr Argument nicht endlich oder negativ ist.
- Dieses Mal wollen $\sqrt{\frac{1}{0}}$ berechnen, also versuchen, das Ergebnis von `sqrt(1 / 0)` in einer Variable `sqrt_of_1_div_0` zu speichern.
- Das ist natürlich Unsinn. Aber schauen wir, was passiert.
- Wir deklarieren zuerst die Variable als `float`.
- Im `try`-`except`-Block machen wir die Berechnung und Wertzuweisung `sqrt_of_1_div_0 = sqrt(1 / 0)`.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Dieses Mal wollen $\sqrt{\frac{1}{0}}$ berechnen, also versuchen, das Ergebnis von `sqrt(1 / 0)` in einer Variable `sqrt_of_1_div_0` zu speichern.
- Das ist natürlich Unsinn. Aber schauen wir, was passiert.
- Wir deklarieren zuerst die Variable als `float`.
- Im `try-except`-Block machen wir die Berechnung und Wertzuweisung `sqrt_of_1_div_0 = sqrt(1 / 0)`.
- Wir wissen, dass `sqrt` einen `ArithmeticError` auslösen könnte definieren wir einen passenden `except`-Block.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Das ist natürlich Unsinn. Aber schauen wir, was passiert.
- Wir deklarieren zuerst die Variable als `float`.
- Im `try`-`except`-Block machen wir die Berechnung und Wertzuweisung `sqrt_of_1_div_0 = sqrt(1 / 0)`.
- Wir wissen, dass `sqrt` einen `ArithmeticError` auslösen könnte definieren wir einen passenden `except`-Block.
- Nun sieht `1 / 0` auch komisch aus, also fangen wir auch einen möglichen `ZeroDivisionError` ab.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Wir deklarieren zuerst die Variable als `float`.
- Im `try-except`-Block machen wir die Berechnung und Wertzuweisung `sqrt_of_1_div_0 = sqrt(1 / 0)`.
- Wir wissen, dass `sqrt` einen `ArithmeticError` auslösen könnte definieren wir einen passenden `except`-Block.
- Nun sieht `1 / 0` auch komisch aus, also fangen wir auch einen möglichen `ZeroDivisionError` ab.
- Wir haben zwei unabhängige `except`-Klausel.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Im `try-except`-Block machen wir die Berechnung und Wertzuweisung `sqrt_of_1_div_0 = sqrt(1 / 0)`.
- Wir wissen, dass `sqrt` einen `ArithmeticError` auslösen könnte definieren wir einen passenden `except`-Block.
- Nun sieht `1 / 0` auch komisch aus, also fangen wir auch einen möglichen `ZeroDivisionError` ab.
- Wir haben zwei unabhängige `except`-Klausel.
- Welche wird ausgeführt?

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Wir wissen, dass `sqrt` einen `ArithmeticError` auslösen könnte definieren wir einen passenden `except`-Block.
- Nun sieht `1 / 0` auch komisch aus, also fangen wir auch einen möglichen `ZeroDivisionError` ab.
- Wir haben zwei unabhängige `except`-Klausel.
- Welche wird ausgeführt?
- $\frac{1}{0}$ ist keine endliche Zahl, also würde `sqrt` eine Ausnahme auslösen.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Nun sieht `1 / 0` auch komisch aus, also fangen wir auch einen möglichen `ZeroDivisionError` ab.
- Wir haben zwei unabhängige `except`-Klausel.
- Welche wird ausgeführt?
- $\frac{1}{0}$ ist keine endliche Zahl, also würde `sqrt` eine Ausnahme auslösen.
- Andererseits kann $\frac{1}{0}$ vielleicht gar nicht berechnet werden und wir bekommen stattdessen einen `ZeroDivisionError`?

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Wir haben zwei unabhängige `except`-Klausel.
- Welche wird ausgeführt?
- $\frac{1}{0}$ ist keine endliche Zahl, also würde `sqrt` eine Ausnahme auslösen.
- Andererseits kann $\frac{1}{0}$ vielleicht gar nicht berechnet werden und wir bekommen stattdessen einen `ZeroDivisionError`?
- Wie wir sehen wird der `except`-Block für `ZeroDivisionError` ausgeführt.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Welche wird ausgeführt?
- $\frac{1}{0}$ ist keine endliche Zahl, also würde `sqrt` eine Ausnahme auslösen.
- Andererseits kann $\frac{1}{0}$ vielleicht gar nicht berechnet werden und wir bekommen stattdessen einen `ZeroDivisionError`?
- Wie wir sehen wird der `except`-Block für `ZeroDivisionError` ausgeführt.
- Um `sqrt(1 / 0)` aufzurufen, muss der Interpreter zuerst `1 / 0` berechnen.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- $\frac{1}{0}$ ist keine endliche Zahl, also würde `sqrt` eine Ausnahme auslösen.
- Andererseits kann $\frac{1}{0}$ vielleicht gar nicht berechnet werden und wir bekommen stattdessen einen `ZeroDivisionError`?
- Wie wir sehen wird der `except`-Block für `ZeroDivisionError` ausgeführt.
- Um `sqrt(1 / 0)` aufzurufen, muss der Interpreter zuerst `1 / 0` berechnen.
- Diese Berechnung löst einen `ZeroDivisionError` aus und `sqrt` wird nie aufgerufen.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Andererseits kann $\frac{1}{0}$ vielleicht gar nicht berechnet werden und wir bekommen stattdessen einen `ZeroDivisionError`?
- Wie wir sehen wird der `except`-Block für `ZeroDivisionError` ausgeführt.
- Um `sqrt(1 / 0)` aufzurufen, muss der Interpreter zuerst `1 / 0` berechnen.
- Diese Berechnung löst einen `ZeroDivisionError` aus und `sqrt` wird nie aufgerufen.
- Das bringt uns zu der Frage: *„Wenn `sqrt` nicht aufgerufen wird, was steht dann in `sqrt_of_1_div_0`?“*

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Wie wir sehen wird der `except`-Block für `ZeroDivisionError` ausgeführt.
- Um `sqrt(1 / 0)` aufzurufen, muss der Interpreter zuerst `1 / 0` berechnen.
- Diese Berechnung löst einen `ZeroDivisionError` aus und `sqrt` wird nie aufgerufen.
- Das bringt uns zu der Frage: *„Wenn `sqrt` nicht aufgerufen wird, was steht dann in `sqrt_of_1_div_0`?“*
- Genau genommen existiert `sqrt_of_1_div_0` gar nicht.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Um `sqrt(1 / 0)` aufzurufen, muss der Interpreter zuerst `1 / 0` berechnen.
- Diese Berechnung löst einen `ZeroDivisionError` aus und `sqrt` wird nie aufgerufen.
- Das bringt uns zu der Frage: *„Wenn `sqrt` nicht aufgerufen wird, was steht dann in `sqrt_of_1_div_0`?“*
- Genau genommen existiert `sqrt_of_1_div_0` gar nicht.
- Die Variable `sqrt_of_1_div_0` würde erst „entstehen“ wenn wir einen Wert darin speichern.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Diese Berechnung löst einen `ZeroDivisionError` aus und `sqrt` wird nie aufgerufen.
- Das bringt uns zu der Frage: *„Wenn `sqrt` nicht aufgerufen wird, was steht dann in `sqrt_of_1_div_0`?“*
- Genau genommen existiert `sqrt_of_1_div_0` gar nicht.
- Die Variable `sqrt_of_1_div_0` würde erst „entstehen“ wenn wir einen Wert darin speichern.
- Was wir nicht tun.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Das bringt uns zu der Frage: *„Wenn `sqrt` nicht aufgerufen wird, was steht dann in `sqrt_of_1_div_0`?“*
- Genau genommen existiert `sqrt_of_1_div_0` gar nicht.
- Die Variable `sqrt_of_1_div_0` würde erst „entstehen“ wenn wir einen Wert darin speichern.
- Was wir nicht tun.
- Um den Wert zuzuweisen, hätten wir das Ergebnis von `sqrt(1 / 0)` gebraucht.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Genau genommen existiert `sqrt_of_1_div_0` gar nicht.
- Die Variable `sqrt_of_1_div_0` würde erst „entstehen“ wenn wir einen Wert darin speichern.
- Was wir nicht tun.
- Um den Wert zuzuweisen, hätten wir das Ergebnis von `sqrt(1 / 0)` gebraucht.
- Und weil das niemals zur Verfügung steht, wird die Zuweisung nicht ausgeführt.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Die Variable `sqrt_of_1_div_0` würde erst „entstehen“ wenn wir einen Wert darin speichern.
- Was wir nicht tun.
- Um den Wert zuzuweisen, hätten wir das Ergebnis von `sqrt(1 / 0)` gebraucht.
- Und weil das niemals zur Verfügung steht, wird die Zuweisung nicht ausgeführt.
- Wenn wir versuchen, auf `print(sqrt_of_1_div_0)` nach dem `try-except`-Block zuzugreifen, dann gibt es den *Name* `sqrt_of_1_div_0` gar nicht.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Was wir nicht tun.
- Um den Wert zuzuweisen, hätten wir das Ergebnis von `sqrt(1 / 0)` gebraucht.
- Und weil das niemals zur Verfügung steht, wird die Zuweisung nicht ausgeführt.
- Wenn wir versuchen, auf `print(sqrt_of_1_div_0)` nach dem `try-except`-Block zuzugreifen, dann gibt es den *Name* `sqrt_of_1_div_0` gar nicht.
- Die Variable `sqrt_of_1_div_0` hat niemals einen Wert bekommen und existiert daher nicht.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Um den Wert zuzuweisen, hätten wir das Ergebnis von `sqrt(1 / 0)` gebraucht.
- Und weil das niemals zur Verfügung steht, wird die Zuweisung nicht ausgeführt.
- Wenn wir versuchen, auf `print(sqrt_of_1_div_0)` nach dem `try`-`except`-Block zuzugreifen, dann gibt es den *Name* `sqrt_of_1_div_0` gar nicht.
- Die Variable `sqrt_of_1_div_0` hat niemals einen Wert bekommen und existiert daher nicht.
- Der Zugriff darauf scheitert mit einem `NameError`.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Und weil das niemals zur Verfügung steht, wird die Zuweisung nicht ausgeführt.
- Wenn wir versuchen, auf `print(sqrt_of_1_div_0)` nach dem `try-except`-Block zuzugreifen, dann gibt es den *Name* `sqrt_of_1_div_0` gar nicht.
- Die Variable `sqrt_of_1_div_0` hat niemals einen Wert bekommen und existiert daher nicht.
- Der Zugriff darauf scheitert mit einem `NameError`.
- Wir haben die Variable mit einem Type-Hint deklariert.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Wenn wir versuchen, auf `print(sqrt_of_1_div_0)` nach dem `try`-`except`-Block zuzugreifen, dann gibt es den *Name* `sqrt_of_1_div_0` gar nicht.
- Die Variable `sqrt_of_1_div_0` hat niemals einen Wert bekommen und existiert daher nicht.
- Der Zugriff darauf scheitert mit einem `NameError`.
- Wir haben die Variable mit einem Type-Hint deklariert.
- Aber der Python-Interpreter ignoriert Type-Hints.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Die Variable `sqrt_of_1_div_0` hat niemals einen Wert bekommen und existiert daher nicht.
- Der Zugriff darauf scheitert mit einem `NameError`.
- Wir haben die Variable mit einem Type-Hint deklariert.
- Aber der Python-Interpreter ignoriert Type-Hints.
- Deshalb kennt er die Variable nicht, da sie keinen Wert hat.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Der Zugriff darauf scheitert mit einem `NameError`.
- Wir haben die Variable mit einem Type-Hint deklariert.
- Aber der Python-Interpreter ignoriert Type-Hints.
- Deshalb kennt er die Variable nicht, da sie keinen Wert hat.
- Das Programm bricht ab, da wir nirgendwo einen `NameError` verarbeiten.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Wir haben die Variable mit einem Type-Hint deklariert.
- Aber der Python-Interpreter ignoriert Type-Hints.
- Deshalb kennt er die Variable nicht, da sie keinen Wert hat.
- Das Programm bricht ab, da wir nirgendwo einen `NameError` verarbeiten.
- Es würde keinen Sinn ergeben, solch einen Fehler abzufangen, da er nur durch einen Bug im Kode entstehen kann.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Aber der Python-Interpreter ignoriert Type-Hints.
- Deshalb kennt er die Variable nicht, da sie keinen Wert hat.
- Das Programm bricht ab, da wir nirgendwo einen `NameError` verarbeiten.
- Es würde keinen Sinn ergeben, solch einen Fehler abzufangen, da er nur durch einen Bug im Kode entstehen kann.
- Und das war wirklich ein Bug.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

- Deshalb kennt er die Variable nicht, da sie keinen Wert hat.
- Das Programm bricht ab, da wir nirgendwo einen `NameError` verarbeiten.
- Es würde keinen Sinn ergeben, solch einen Fehler abzufangen, da er nur durch einen Bug im Kode entstehen kann.
- Und das war wirklich ein Bug.
- Deshalb wird der Stack-Trace und die Fehler-Information ausgegeben.

```python
"""Demonstrate `try...except` with multiple exceptions and NameError."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Which error will this produce?
except ZeroDivisionError:  # Did a division by zero happen?
    print(f"We got a division-by-zero error!", flush=True)
except ArithmeticError as ae:  # Or an ArithmeticError?
    print(f"We got an arithmetic error: {ae}!", flush=True)

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_multi_except.py ↓

```
We got a division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
Traceback (most recent call last):
  File "{...}/exceptions/try_multi_except.py", line 16, in <module>
    print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not
        ↪ assigned
          ^^^^^^^^^^^^^^
NameError: name 'sqrt_of_1_div_0' is not defined
# 'python3 try_multi_except.py' failed with exit code 1.
```

Kompliziertes Beispiel

Gute Praxis

Beachten Sie, dass der Kontrollfluss sofort den aktuellen Block verlässt, wenn eine Ausnahme ausgelöst wird.

Kompliziertes Beispiel

Gute Praxis

Beachten Sie, dass der Kontrollfluss sofort den aktuellen Block verlässt, wenn eine Ausnahme ausgelöst wird. Der Befehl, in dem die Ausnahme ausgelöst wird, wird nicht zuende geführt, sondern sofort abgebrochen.

Kompliziertes Beispiel

Gute Praxis

Beachten Sie, dass der Kontrollfluss sofort den aktuellen Block verlässt, wenn eine Ausnahme ausgelöst wird. Der Befehl, in dem die Ausnahme ausgelöst wird, wird nicht zuende geführt, sondern sofort abgebrochen. Darum können dann keine Variablenzuweisungen mehr stattfinden und es ist möglich, das Variablen undefiniert bleiben.

Kompliziertes Beispiel



Gute Praxis

Beachten Sie, dass der Kontrollfluss sofort den aktuellen Block verlässt, wenn eine Ausnahme ausgelöst wird. Der Befehl, in dem die Ausnahme ausgelöst wird, wird nicht zuende geführt, sondern sofort abgebrochen. Darum können dann keine Variablenzuweisungen mehr stattfinden und es ist möglich, das Variablen undefiniert bleiben. Beachten Sie dass, wenn Sie auf Variablen zugreifen, die in einem `try`-block zugewiesen werden.

Verschachtelte Fehlerbehandlung



Fehler in der Fehlerbehandlung

- Was passiert, wenn eine Ausnahme in einem `except`-Block ausgelöst wird?

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                           ~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                           ~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Was passiert, wenn eine Ausnahme in einem `except`-Block ausgelöst wird?
- Wir probieren das in `try_except_nested_1.py` aus.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                           ~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                           ~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Was passiert, wenn eine Ausnahme in einem `except`-Block ausgelöst wird?
- Wir probieren das in `try_except_nested_1.py` aus.
- Wir versuchen einfach, `sqrt_of_1_div_0 = sqrt(1 / 0)` zweimal zu berechnen.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                           ~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                           ~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Was passiert, wenn eine Ausnahme in einem `except`-Block ausgelöst wird?
- Wir probieren das in `try_except_nested_1.py` aus.
- Wir versuchen einfach, `sqrt_of_1_div_0 = sqrt(1 / 0)` zweimal zu berechnen.
- Zuerst im `try`-Block und und dann nochmal im `except`-Block, der den `ZeroDivisionError` behandelt.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                           ~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                           ~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Was passiert, wenn eine Ausnahme in einem `except`-Block ausgelöst wird?
- Wir probieren das in `try_except_nested_1.py` aus.
- Wir versuchen einfach, `sqrt_of_1_div_0 = sqrt(1 / 0)` zweimal zu berechnen.
- Zuerst im `try`-Block und und dann nochmal im `except`-Block, der den `ZeroDivisionError` behandelt.
- In anderen Worten: In einem `except`-Block für `ZeroDivisionErrors` wird ein weiterer `ZeroDivisionError` ausgelöst.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                           ~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                           ~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Wir probieren das in `try_except_nested_1.py` aus.
- Wir versuchen einfach, `sqrt_of_1_div_0 = sqrt(1 / 0)` zweimal zu berechnen.
- Zuerst im `try`-Block und und dann nochmal im `except`-Block, der den `ZeroDivisionError` behandelt.
- In anderen Worten: In einem `except`-Block für `ZeroDivisionErrors` wird ein weiterer `ZeroDivisionError` ausgelöst.
- Der `except`-Block wird dadurch sofort abgebrochen und erklärender Text wird auf stderr ausgegeben.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                      ~~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                      ~~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Zuerst im `try`-Block und und dann nochmal im `except`-Block, der den `ZeroDivisionError` behandelt.
- In anderen Worten: In einem `except`-Block für `ZeroDivisionErrors` wird ein weiterer `ZeroDivisionError` ausgelöst.
- Der `except`-Block wird dadurch sofort abgebrochen und erklärender Text wird auf stderr ausgegeben.
- Während wir den originalen `ZeroDivisionError` behandelt haben, ist ein weiterer Fehler aufgetreten.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                      ~~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                      ~~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- In anderen Worten: In einem `except`-Block für `ZeroDivisionErrors` wird ein weiterer `ZeroDivisionError` ausgelöst.
- Der `except`-Block wird dadurch sofort abgebrochen und erklärender Text wird auf stderr ausgegeben.
- Während wir den originalen `ZeroDivisionError` behandelt haben, ist ein weiterer Fehler aufgetreten.
- Die Ausgabe zeigt uns zuerst den Stack-Trace der Ausnahme, die wir gerade versucht haben zu behandeln.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                      ~~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                      ~~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Der `except`-Block wird dadurch sofort abgebrochen und erklärender Text wird auf stderr ausgegeben.
- Während wir den originalen `ZeroDivisionError` behandelt haben, ist ein weiterer Fehler aufgetreten.
- Die Ausgabe zeigt uns zuerst den Stack-Trace der Ausnahme, die wir gerade versucht haben zu behandeln.
- Dann sagt es uns, dass *„During handling of the above exception, another exception occurred:“*.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                      ~~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                      ~~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Der `except`-Block wird dadurch sofort abgebrochen und erklärender Text wird auf stderr ausgegeben.
- Während wir den originalen `ZeroDivisionError` behandelt haben, ist ein weiterer Fehler aufgetreten.
- Die Ausgabe zeigt uns zuerst den Stack-Trace der Ausnahme, die wir gerade versucht haben zu behandeln.
- Dann sagt es uns, dass *„During handling of the above exception, another exception occurred:“*.
- Dann gibt es den Stack-Trace der neuen Ausnahme aus, die in unserem `except`-Block aufgetreten ist.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                      ~~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                      ~~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Fehler in der Fehlerbehandlung

- Während wir den originalen `ZeroDivisionError` behandelt haben, ist ein weiterer Fehler aufgetreten.
- Die Ausgabe zeigt uns zuerst den Stack-Trace der Ausnahme, die wir gerade versucht haben zu behandeln.
- Dann sagt es uns, dass *„During handling of the above exception, another exception occurred:“*.
- Dann gibt es den Stack-Trace der neuen Ausnahme aus, die in unserem `except`-Block aufgetreten ist.
- Weil es keinen Kode gab, der diesen Fehler behandeln kann, ist unser Prozess mit exit code 1 abgebrochen.

```python
"""Demonstrate `try...except` with an exception raised in `except`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Does not work: sqrt_of_1_div_0 is not assigned
print("The program is now finished.")  # We never get here.
```

↓ python3 try_except_nested_1.py ↓

```
We got a division-by-zero error: division by zero.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 9, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
                      ~~~~~
ZeroDivisionError: division by zero

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "{...}/exceptions/try_except_nested_1.py", line 12, in <module>
    sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
                      ~~~~~
ZeroDivisionError: division by zero
# 'python3 try_except_nested_1.py' failed with exit code 1.
```

Verschachtelte Fehlerbehandlung

- Natürlich ist es möglich, dass Kode, der einen Fehler behandelt, einen anderen Fehler auslöst.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ python3 try_except_nested_2.py ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- Natürlich ist es möglich, dass Kode, der einen Fehler behandelt, einen anderen Fehler auslöst.
- Wir können daher auch `try`-`except`-Blöcke verschachteln.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ python3 try_except_nested_2.py ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- Natürlich ist es möglich, dass Kode, der einen Fehler behandelt, einen anderen Fehler auslöst.
- Wir können daher auch `try`-`except`-Blöcke verschachteln.
- Program `try_except_nested_2.py` zeigt genau das.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ `python3 try_except_nested_2.py` ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- Natürlich ist es möglich, dass Kode, der einen Fehler behandelt, einen anderen Fehler auslöst.
- Wir können daher auch `try`-`except`-Blöcke verschachteln.
- Program `try_except_nested_2.py` zeigt genau das.
- In dem wir den zweiten `sqrt`-Aufruf, der bereits in einem `except`-Block ist, wiederum in einen neuen `try`-`except`-Block packen, können wir den zweiten `ZeroDivisionError` ebenfalls auffangen und bearbeiten.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ python3 try_except_nested_2.py ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- Wir können daher auch `try`-`except`-Blöcke verschachteln.
- Program `try_except_nested_2.py` zeigt genau das.
- In dem wir den zweiten `sqrt`-Aufruf, der bereits in einem `except`-Block ist, wiederum in einen neuen `try`-`except`-Block packen, können wir den zweiten `ZeroDivisionError` ebenfalls auffangen und bearbeiten.
- In diesem Beispiel zeigen wir auch zwei Methoden, um mit dem Problem der Variablenzuweisung in `try`-`except`-Blöcken umzugehen.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ `python3 try_except_nested_2.py` ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- Program `try_except_nested_2.py` zeigt genau das.
- In dem wir den zweiten `sqrt`-Aufruf, der bereits in einem `except`-Block ist, wiederum in einen neuen `try`-`except`-Block packen, können wir den zweiten `ZeroDivisionError` ebenfalls auffangen und bearbeiten.
- In diesem Beispiel zeigen wir auch zwei Methoden, um mit dem Problem der Variablenzuweisung in `try`-`except`-Blöcken umzugehen.
- Erstens können wir der Variable einen initialen Wert vor allen Berechnungen zuweisen.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ `python3 try_except_nested_2.py` ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- In dem wir den zweiten `sqrt`-Aufruf, der bereits in einem `except`-Block ist, wiederum in einen neuen `try-except`-Block packen, können wir den zweiten `ZeroDivisionError` ebenfalls auffangen und bearbeiten.
- In diesem Beispiel zeigen wir auch zwei Methoden, um mit dem Problem der Variablenzuweisung in `try-except`-Blöcken umzugehen.
- Erstens können wir der Variable einen initialen Wert vor allen Berechnungen zuweisen.
- Dieser Wert wird überschrieben, wenn die Berechnung doch erfolgreich abläuft.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ python3 try_except_nested_2.py ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- In diesem Beispiel zeigen wir auch zwei Methoden, um mit dem Problem der Variablenzuweisung in `try-except`-Blöcken umzugehen.
- Erstens können wir der Variable einen initialen Wert vor allen Berechnungen zuweisen.
- Dieser Wert wird überschrieben, wenn die Berechnung doch erfolgreich abläuft.
- Dadurch ist sichergestellt, dass die Variable garantiert einen Wert hat und existiert, gleichgültig, was später passiert.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ python3 try_except_nested_2.py ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- In diesem Beispiel zeigen wir auch zwei Methoden, um mit dem Problem der Variablenzuweisung in `try`-`except`-Blöcken umzugehen.
- Erstens können wir der Variable einen initialen Wert vor allen Berechnungen zuweisen.
- Dieser Wert wird überschrieben, wenn die Berechnung doch erfolgreich abläuft.
- Dadurch ist sichergestellt, dass die Variable garantiert einen Wert hat und existiert, gleichgültig, was später passiert.
- Wir nehmen dafür `nan` aus dem `math`-Module.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ python3 try_except_nested_2.py ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- Dieser Wert wird überschrieben, wenn die Berechnung doch erfolgreich abläuft.
- Dadurch ist sichergestellt, dass die Variable garantiert einen Wert hat und existiert, gleichgültig, was später passiert.
- Wir nehmen dafür `nan` aus dem `math`-Module.
- Die zweite und kompliziertere Methode ist, sicherzustellen, dass alle Zweige des Kontrollflusses, die zu dem Kode führen, wo auf die Variable zugegriffen wird, ihr auch garantiert einen Wert zuweisen.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ `python3 try_except_nested_2.py` ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- Dadurch ist sichergestellt, dass die Variable garantiert einen Wert hat und existiert, gleichgültig, was später passiert.
- Wir nehmen dafür `nan` aus dem `math`-Module.
- Die zweite und kompliziertere Methode ist, sicherzustellen, dass alle Zweige des Kontrollflusses, die zu dem Kode führen, wo auf die Variable zugegriffen wird, ihr auch garantiert einen Wert zuweisen.
- In dem Beispiel würden wir dafür im innersten `except`-Block einen Wert zuweisen müssen.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ python3 try_except_nested_2.py ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Verschachtelte Fehlerbehandlung

- Wir nehmen dafür `nan` aus dem `math`-Module.
- Die zweite und kompliziertere Methode ist, sicherzustellen, dass alle Zweige des Kontrollflusses, die zu dem Kode führen, wo auf die Variable zugegriffen wird, ihr auch garantiert einen Wert zuweisen.
- In dem Beispiel würden wir dafür im innersten `except`-Block einen Wert zuweisen müssen.
- Wir könnten dafür wieder `nan` nehmen.

```python
"""Demonstrate nested `try...except` blocks."""

from math import nan
from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float = nan  # Declare this variable and assign it.

try:  # If this block raises an error, we continue at `except`.
    # sqrt_of_1_div_0 only gets assigned if sqrt(1 / 0) succeeds...
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This produces a ZeroDivisionError.
except ZeroDivisionError as de:  # Catch an ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
    try:  # Nesting try-except blocks is totally fine.
        sqrt_of_1_div_0 = sqrt(1 / 0)  # Let's try it again!
    except ZeroDivisionError:  # Another ZeroDivisionError?
        print(f"Yet another division-by-zero error!", flush=True)
        # We could also assign a value to sqrt_of_1_div_0 here, which
        # would work, but we already chose the solution with an initial
        # value before all the try-except blocks.

print("Now we try to print the value of sqrt_of_1_div_0.", flush=True)
print(sqrt_of_1_div_0)  # Works, because we gave an initial value.
print("The program is now finished.")  # This time, we do get here.
```

↓ python3 try_except_nested_2.py ↓

```
We got a division-by-zero error: division by zero.
Yet another division-by-zero error!
Now we try to print the value of sqrt_of_1_div_0.
nan
The program is now finished.
```

Der `try-except-else`-Block



Der `try-except-else`-Block

- Was können wir machen, wenn wir das Ergebnis einer Berechnung in einem `try`-Block brauchen, aber nur dann, wenn der `try`-Block erfolgreich zuende läuft?

```
"""The syntax of a try-except-else statement in Python."""

try:    # Begin the try-except block.
        statement 1    # Code that may raise an exception or that
        statement 2    # calls a function that may rise one.
        #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
        statements processing ex1
        # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:    # The else block is optional.
        statement 3    # Code executed if and only if no Exception occurred.
        statement 4
        #...

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try-except-else`-Block

- Was können wir machen, wenn wir das Ergebnis einer Berechnung in einem `try`-Block brauchen, aber nur dann, wenn der `try`-Block erfolgreich zuende läuft?
- Eine mögliche Lösung ist der `try`-`except`-`else`-Block.

```
"""The syntax of a try-except-else statement in Python."""

try:    # Begin the try-except block.
        statement 1    # Code that may raise an exception or that
        statement 2    # calls a function that may rise one.
        #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
        statements processing ex1
        # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:    # The else block is optional.
        statement 3    # Code executed if and only if no Exception occurred.
        statement 4
        #...

next statement    # Executed only if there are no uncaught Exceptions.
```

Der `try-except-else`-Block

- Was können wir machen, wenn wir das Ergebnis einer Berechnung in einem `try`-Block brauchen, aber nur dann, wenn der `try`-Block erfolgreich zuende läuft?
- Eine mögliche Lösung ist der `try-except-else`-Block.
- Der einzige Unterschied zum `try-except`-Block ist, dass ein `else`-Block folgt, der nur aufgerufen wird, wenn keine Ausnahme aufgetreten ist.

```
"""The syntax of a try-except-else statement in Python."""

try:    # Begin the try-except block.
        statement 1    # Code that may raise an exception or that
        statement 2    # calls a function that may rise one.
        #...
except ExceptionType1 as ex1:    # One exception type that can be caught.
        statements processing ex1
        # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:    # The else block is optional.
        statement 3    # Code executed if and only if no Exception occurred.
        statement 4
        #...

next statement    # Executed only if there are no uncaught Exceptions.
```

Beispiel

- Wir schauen uns das am Beispiel von Program `try_except_else.py` an.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Wir schauen uns das am Beispiel von Program `try_except_else.py` an.
- Wir benutzen wieder unsere `sqrt`-Funktion, diesmal um $\sqrt{\frac{1}{0}}$ und danach $\sqrt{3}$ zu berechnen, jedesmal in einem eigenen `try`-Block.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Wir schauen uns das am Beispiel von Program `try_except_else.py` an.
- Wir benutzen wieder unsere `sqrt`-Funktion, diesmal um $\sqrt{\frac{1}{0}}$ und danach $\sqrt{3}$ zu berechnen, jedesmal in einem eigenen `try`-Block.
- Wir hatten ja früher das Problem, dass wir einen `NameError` bekommen haben, weil wir auf den Wert einer nicht-existierenden Variable zugreifen wollten.

```
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Wir schauen uns das am Beispiel von Program `try_except_else.py` an.
- Wir benutzen wieder unsere `sqrt`-Funktion, diesmal um $\sqrt{\frac{1}{0}}$ und danach $\sqrt{3}$ zu berechnen, jedesmal in einem eigenen `try`-Block.
- Wir hatten ja früher das Problem, dass wir einen `NameError` bekommen haben, weil wir auf den Wert einer nicht-existierenden Variable zugreifen wollten.
- Diese Variable wäre im `try`-Block erst durch eine Zuweisung entstanden.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Wir schauen uns das am Beispiel von Program `try_except_else.py` an.
- Wir benutzen wieder unsere `sqrt`-Funktion, diesmal um $\sqrt{\frac{1}{0}}$ und danach $\sqrt{3}$ zu berechnen, jedesmal in einem eigenen `try`-Block.
- Wir hatten ja früher das Problem, dass wir einen `NameError` bekommen haben, weil wir auf den Wert einer nicht-existierenden Variable zugreifen wollten.
- Diese Variable wäre im `try`-Block erst durch eine Zuweisung entstanden.
- Aber der `try`-Block war fehlgeschlagen.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Wir hatten ja früher das Problem, dass wir einen `NameError` bekommen haben, weil wir auf den Wert einer nicht-existierenden Variable zugreifen wollten.
- Diese Variable wäre im `try`-Block erst durch eine Zuweisung entstanden.
- Aber der `try`-Block war fehlgeschlagen.
- Wir hatten einen `ZeroDivisionError` im `except`-Block abgefangen, aber der Variable ist nie ein Wert zugewiesen worden.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Wir hatten ja früher das Problem, dass wir einen `NameError` bekommen haben, weil wir auf den Wert einer nicht-existierenden Variable zugreifen wollten.
- Diese Variable wäre im `try`-Block erst durch eine Zuweisung entstanden.
- Aber der `try`-Block war fehlgeschlagen.
- Wir hatten einen `ZeroDivisionError` im `except`-Block abgefangen, aber der Variable ist nie ein Wert zugewiesen worden.
- Auf die Variable später zuzugreifen führte zu dem Fehler.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Diese Variable wäre im `try`-Block erst durch eine Zuweisung entstanden.
- Aber der `try`-Block war fehlgeschlagen.
- Wir hatten einen `ZeroDivisionError` im `except`-Block abgefangen, aber der Variable ist nie ein Wert zugewiesen worden.
- Auf die Variable später zuzugreifen führte zu dem Fehler.
- Dieses Mal packen wir den Kode, der auf die Variable zugreift, in `else`-Blöcke.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Aber der `try`-Block war fehlgeschlagen.
- Wir hatten einen `ZeroDivisionError` im `except`-Block abgefangen, aber der Variable ist nie ein Wert zugewiesen worden.
- Auf die Variable später zuzugreifen führte zu dem Fehler.
- Dieses Mal packen wir den Kode, der auf die Variable zugreift, in `else`-Blöcke.
- Das ist also die dritte Variante, auf Variablen zuzugreifen, deren Werte in einem `try`-Block zugewiesen werden.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Wir hatten einen `ZeroDivisionError` im `except`-Block abgefangen, aber der Variable ist nie ein Wert zugewiesen worden.
- Auf die Variable später zuzugreifen führte zu dem Fehler.
- Dieses Mal packen wir den Kode, der auf die Variable zugreift, in `else`-Blöcke.
- Das ist also die dritte Variante, auf Variablen zuzugreifen, deren Werte in einem `try`-Block zugewiesen werden.
- Diese `else`-Blöcke werden nur ausgeführt, wenn der dazugehörige `try`-Block erfolgreich war.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Auf die Variable später zuzugreifen führte zu dem Fehler.
- Dieses Mal packen wir den Kode, der auf die Variable zugreift, in `else`-Blöcke.
- Das ist also die dritte Variante, auf Variablen zuzugreifen, deren Werte in einem `try`-Block zugewiesen werden.
- Diese `else`-Blöcke werden nur ausgeführt, wenn der dazugehörige `try`-Block erfolgreich war.
- Daher ist garantiert, dass die Variablen existieren und dass ihnen Werte zugewiesen wurden.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Dieses Mal packen wir den Kode, der auf die Variable zugreift, in `else`-Blöcke.
- Das ist also die dritte Variante, auf Variablen zuzugreifen, deren Werte in einem `try`-Block zugewiesen werden.
- Diese `else`-Blöcke werden nur ausgeführt, wenn der dazugehörige `try`-Block erfolgreich war.
- Daher ist garantiert, dass die Variablen existieren und dass ihnen Werte zugewiesen wurden.
- Im Falle von $\sqrt{\frac{1}{0}}$ wird der `else`-Block nicht ausgeführt.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Beispiel

- Das ist also die dritte Variante, auf Variablen zuzugreifen, deren Werte in einem `try`-Block zugewiesen werden.
- Diese `else`-Blöcke werden nur ausgeführt, wenn der dazugehörige `try`-Block erfolgreich war.
- Daher ist garantiert, dass die Variablen existieren und dass ihnen Werte zugewiesen wurden.
- Im Falle von $\sqrt{\frac{1}{0}}$ wird der `else`-Block nicht ausgeführt.
- Im Fall von $\sqrt{3}$ wird der dazugehörige `else`-Block mit seinem `print` ausgeführt.

```python
"""Demonstrate `try...except..else`."""

from sqrt_raise import sqrt  # Import our sqrt function.

sqrt_of_1_div_0: float  # Declare this variable, but do not assign it.

try:  # If this block raises DivisionByZero, we continue at `except`.
    sqrt_of_1_div_0 = sqrt(1 / 0)  # This is a DivisionByZero!
except ZeroDivisionError as de:  # Catch and print a ZeroDivisionError.
    print(f"We got a division-by-zero error: {de}.", flush=True)
else:  # This code is not executed because an exception occurred.
    print(f"\u221A(1/0)\u2248{sqrt_of_1_div_0}")  # Never reached.

sqrt_3: float  # Declare this variable, but do not assign it.
try:  # If this block raises ArithmeticError, we continue at `except`.
    sqrt_3 = sqrt(3.0)  # This will work just fine and raise no error.
except ZeroDivisionError:  # We catch and print the ZeroDivisionError.
    print(f"We got a another division-by-zero error!", flush=True)
else:  # This code is executed because no exception occurs.
    print(f"\u221A3\u2248{sqrt_3}")  # Always executed.

print("The program is now finished.")  # We do get here.
```

↓ `python3 try_except_else.py` ↓

```
We got a division-by-zero error: division by zero.
√3≈1.7320508075688772
The program is now finished.
```

Der try-finally-Block

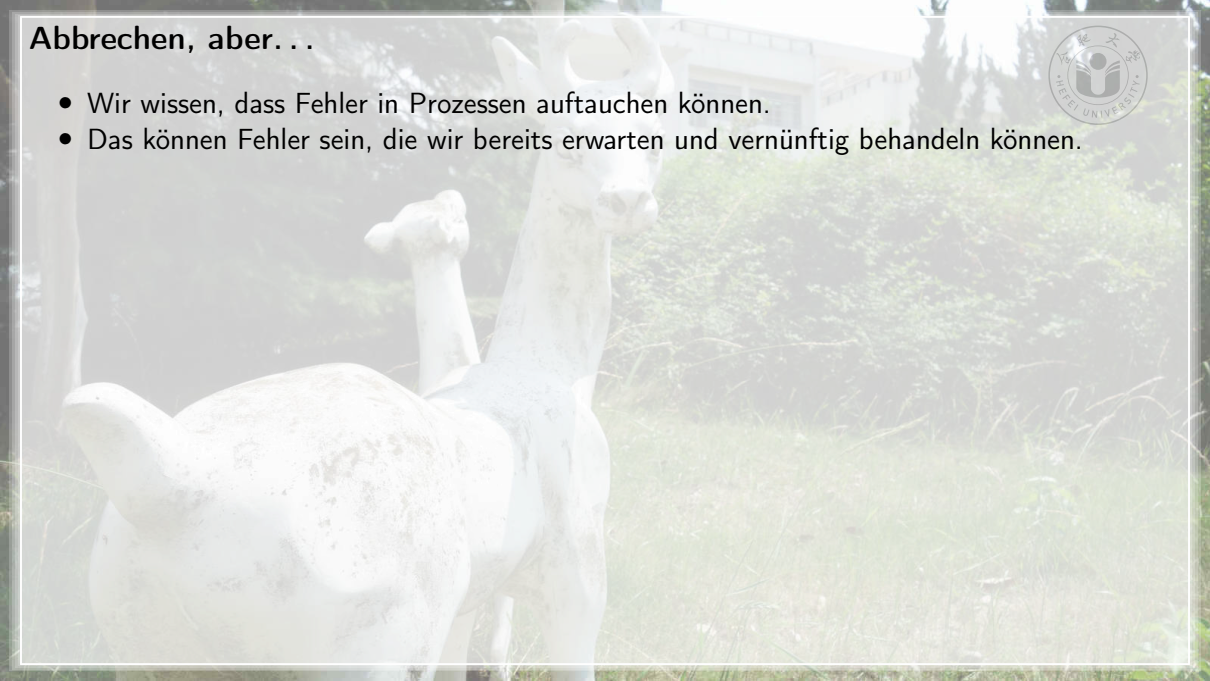


Abbrechen, aber...

- Wir wissen, dass Fehler in Prozessen auftauchen können.

Abbrechen, aber...

- Wir wissen, dass Fehler in Prozessen auftauchen können.
- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.



Abbrechen, aber...



- Wir wissen, dass Fehler in Prozessen auftauchen können.
- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.
- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.

Abbrechen, aber...



- Wir wissen, dass Fehler in Prozessen auftauchen können.
- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.
- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.
- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.

Abbrechen, aber...



- Wir wissen, dass Fehler in Prozessen auftauchen können.
- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.
- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.
- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.
- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.

Abbrechen, aber...



- Wir wissen, dass Fehler in Prozessen auftauchen können.
- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.
- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.
- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.
- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.
- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.

Abbrechen, aber...



- Wir wissen, dass Fehler in Prozessen auftauchen können.
- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.
- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.
- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.
- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.
- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.
- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.

Abbrechen, aber...



- Wir wissen, dass Fehler in Prozessen auftauchen können.
- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.
- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.
- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.
- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.
- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.
- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.
- Ein typisches Beispiel ist, wenn wir etwas in eine Datei schreiben.

Abbrechen, aber...



- Wir wissen, dass Fehler in Prozessen auftauchen können.
- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.
- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.
- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.
- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.
- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.
- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.
- Ein typisches Beispiel ist, wenn wir etwas in eine Datei schreiben.
- Sagen wir, wir schreiben eine Tabelle mit Daten Zeile-für-Zeile in eine Datei.

Abbrechen, aber...



- Das können Fehler sein, die wir bereits erwarten und vernünftig behandeln können.
- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.
- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.
- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.
- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.
- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.
- Ein typisches Beispiel ist, wenn wir etwas in eine Datei schreiben.
- Sagen wir, wir schreiben eine Tabelle mit Daten Zeile-für-Zeile in eine Datei.
- Plötzlich passiert ein völlig unerwarteter Fehler, sagen wir etwas eigenartiges wie ein `ReferenceError`.

Abbrechen, aber...



- Diese Fehler verarbeiten wir dann in entsprechenden `except`-Blöcken.
- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.
- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.
- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.
- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.
- Ein typisches Beispiel ist, wenn wir etwas in eine Datei schreiben.
- Sagen wir, wir schreiben eine Tabelle mit Daten Zeile-für-Zeile in eine Datei.
- Plötzlich passiert ein völlig unerwarteter Fehler, sagen wir etwas eigenartiges wie ein `ReferenceError`.
- Natürlich sollte unser Prozess dann mit einem Fehler beendet werden.

Abbrechen, aber...



- Es können aber auch unerwartete Fehler sein, für die wir keine vernünften `except`-Blöcke bereitstellen können.
- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.
- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.
- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.
- Ein typisches Beispiel ist, wenn wir etwas in eine Datei schreiben.
- Sagen wir, wir schreiben eine Tabelle mit Daten Zeile-für-Zeile in eine Datei.
- Plötzlich passiert ein völlig unerwarteter Fehler, sagen wir etwas eigenartiges wie ein `ReferenceError`.
- Natürlich sollte unser Prozess dann mit einem Fehler beendet werden.
- Wenn wir aber abbrechen, ohne die Datei vorher zu schließen, dann könnte der gesamte Inhalt der Datei verloren gehen.

Abbrechen, aber...



- In diesem Fall wird das Program abgebrochen und der Stack-Trace wird ausgegeben.
- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.
- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.
- Ein typisches Beispiel ist, wenn wir etwas in eine Datei schreiben.
- Sagen wir, wir schreiben eine Tabelle mit Daten Zeile-für-Zeile in eine Datei.
- Plötzlich passiert ein völlig unerwarteter Fehler, sagen wir etwas eigenartiges wie ein `ReferenceError`.
- Natürlich sollte unser Prozess dann mit einem Fehler beendet werden.
- Wenn wir aber abbrechen, ohne die Datei vorher zu schließen, dann könnte der gesamte Inhalt der Datei verloren gehen.
- Wenn wir die Datei noch schließen, bevor wir den Prozess abbrechen, dann werden wenigstens die bisher erfolgreich geschriebenen Daten gerettet.

Abbrechen, aber...



- Es gibt aber Fälle, wo wir nicht sofort unseren Prozess abbrechen wollen.
- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.
- Ein typisches Beispiel ist, wenn wir etwas in eine Datei schreiben.
- Sagen wir, wir schreiben eine Tabelle mit Daten Zeile-für-Zeile in eine Datei.
- Plötzlich passiert ein völlig unerwarteter Fehler, sagen wir etwas eigenartiges wie ein `ReferenceError`.
- Natürlich sollte unser Prozess dann mit einem Fehler beendet werden.
- Wenn wir aber abbrechen, ohne die Datei vorher zu schließen, dann könnte der gesamte Inhalt der Datei verloren gehen.
- Wenn wir die Datei noch schließen, bevor wir den Prozess abbrechen, dann werden wenigstens die bisher erfolgreich geschriebenen Daten gerettet.
- In dem wir den Prozess danach abbrechen, zeigen wir trotzdem dem Benutzer klar an, dass ein ernstes Problem aufgetreten ist.

Abbrechen, aber...



- Stattdessen wollen wir vielleicht noch ein paar wichtige Aktionen ausführen, selbst wenn ein unerwarteter Fehler auftritt.
- Ein typisches Beispiel ist, wenn wir etwas in eine Datei schreiben.
- Sagen wir, wir schreiben eine Tabelle mit Daten Zeile-für-Zeile in eine Datei.
- Plötzlich passiert ein völlig unerwarteter Fehler, sagen wir etwas eigenartiges wie ein `ReferenceError`.
- Natürlich sollte unser Prozess dann mit einem Fehler beendet werden.
- Wenn wir aber abbrechen, ohne die Datei vorher zu schließen, dann könnte der gesamte Inhalt der Datei verloren gehen.
- Wenn wir die Datei noch schließen, bevor wir den Prozess abbrechen, dann werden wenigstens die bisher erfolgreich geschriebenen Daten gerettet.
- In dem wir den Prozess danach abbrechen, zeigen wir trotzdem dem Benutzer klar an, dass ein ernstes Problem aufgetreten ist.
- Aber wenigstens vernichten wir nicht die korrekten Daten.

Der `try-finally`-Block

- Dafür gibt es den `try-finally` Block.

```
"""The syntax of a try-except-else-finally statement in Python."""

try:    # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # The except blocks are optional here.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:   # The else block is optional.
    statement 3  # Code executed if and only if no Exception occurred.
    statement 4
    # ...
finally:  # The optional finally block.
    statement 5  # This code will always be executed, even if there are
    statement 6  # uncaught exceptions.

next statement  # Executed only if there are no uncaught Exceptions.
```

Der `try-finally`-Block

- Dafür gibt es den `try`-`finally` Block.
- Wir können einen `finally`-Block definieren, der immer ausgeführt wird.

```
"""The syntax of a try-except-else-finally statement in Python."""

try:    # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # The except blocks are optional here.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:   # The else block is optional.
    statement 3  # Code executed if and only if no Exception occurred.
    statement 4
    # ...
finally:    # The optional finally block.
    statement 5  # This code will always be executed, even if there are
    statement 6  # uncaught exceptions.

next statement  # Executed only if there are no uncaught Exceptions.
```

Der `try-finally`-Block

- Dafür gibt es den `try`-`finally` Block.
- Wir können einen `finally`-Block definieren, der immer ausgeführt wird.
- Na gut. Immer gilt natürlich nur, so lange der Python-Interprer läuft. Wenn ich den Strom abschalte oder den Interpreter mit dem Task-Manager abschieße, dann kann man nicht garantieren, dass ein `finally`-Block ausgeführt wird...

```
"""The syntax of a try-except-else-finally statement in Python."""

try:    # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # The except blocks are optional here.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:   # The else block is optional.
    statement 3  # Code executed if and only if no Exception occurred.
    statement 4
    # ...
finally:  # The optional finally block.
    statement 5  # This code will always be executed, even if there are
    statement 6  # uncaught exceptions.

next statement  # Executed only if there are no uncaught Exceptions.
```

Der `try-finally`-Block

- Wir können einen `finally`-Block definieren, der immer ausgeführt wird.
- Na gut. Immer gilt natürlich nur, so lange der Python-Interprer läuft. Wenn ich den Strom abschalte oder den Interpreter mit dem Task-Manager abschieße, dann kann man nicht garantieren, dass ein `finally`-Block ausgeführt wird...
- Davon abgesehen.

```
"""The syntax of a try-except-else-finally statement in Python."""

try:    # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # The except blocks are optional here.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:   # The else block is optional.
    statement 3  # Code executed if and only if no Exception occurred.
    statement 4
    # ...
finally:  # The optional finally block.
    statement 5  # This code will always be executed, even if there are
    statement 6  # uncaught exceptions.

next statement  # Executed only if there are no uncaught Exceptions.
```

Der `try-finally`-Block

- Na gut. Immer gilt natürlich nur, so lange der Python-Interprer läuft. Wenn ich den Strom abschalte oder den Interpreter mit dem Task-Manager abschieße, dann kann man nicht garantieren, dass ein `finally`-Block ausgeführt wird...
- Davon abgesehen.
- Wir brauchen einen `try`-Block mit dem Kode der einen Fehler auslösen könnte.

```
"""The syntax of a try-except-else-finally statement in Python."""

try:    # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # The except blocks are optional here.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:  # The else block is optional.
    statement 3  # Code executed if and only if no Exception occurred.
    statement 4
    # ...
finally:  # The optional finally block.
    statement 5  # This code will always be executed, even if there are
    statement 6  # uncaught exceptions.

next statement  # Executed only if there are no uncaught Exceptions.
```

Der `try-finally`-Block

- Davon abgesehen.
- Wir brauchen einen `try`-Block mit dem Kode der einen Fehler auslösen könnte.
- Dann kommt der `finally`-Block, der ausgeführt wird gleichgültig ob im `try`-Block ein Fehler aufgetreten ist, oder nicht.

```
"""The syntax of a try-except-else-finally statement in Python."""

try:    # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # The except blocks are optional here.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:  # The else block is optional.
    statement 3  # Code executed if and only if no Exception occurred.
    statement 4
    # ...
finally:  # The optional finally block.
    statement 5  # This code will always be executed, even if there are
    statement 6  # uncaught exceptions.

next statement  # Executed only if there are no uncaught Exceptions.
```

Der `try-finally`-Block

- Wir brauchen einen `try`-Block mit dem Kode der einen Fehler auslösen könnte.
- Dann kommt der `finally`-Block, der ausgeführt wird gleichgültig ob im `try`-Block ein Fehler aufgetreten ist, oder nicht.
- Dazwischen können wir optional noch `except`-Blöcke zum Verarbeiten bestimmter Fehler und einen `else`-Block haben, der nur ausgeführt wird, wenn kein Fehler auftritt.

```
"""The syntax of a try-except-else-finally statement in Python."""

try:    # Begin the try-except block.
    statement 1  # Code that may raise an exception or that
    statement 2  # calls a function that may rise one.
    # ...
except ExceptionType1 as ex1:  # The except blocks are optional here.
    statements processing ex1
    # Code the handles exceptions of type ExceptionType1.
# ... maybe more `except` blocks
else:  # The else block is optional.
    statement 3  # Code executed if and only if no Exception occurred.
    statement 4
    # ...
finally:  # The optional finally block.
    statement 5  # This code will always be executed, even if there are
    statement 6  # uncaught exceptions.

next statement  # Executed only if there are no uncaught Exceptions.
```

Beispiel

- `try_except_else_finally.py` ist ein künstliches Beispiel für diese Struktur.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- `try_except_else_finally.py` ist ein künstliches Beispiel für diese Struktur.
- Wir erstellen die Funktion `divide_and_print`, welche die beiden Parameter `a` und `b` empfängt, die beide entweder Ganz- oder Fließkommazahlen seien können.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- `try_except_else_finally.py` ist ein künstliches Beispiel für diese Struktur.
- Wir erstellen die Funktion `divide_and_print`, welche die beiden Parameter `a` und `b` empfängt, die beide entweder Ganz- oder Fließkommazahlen seien können.
- In einem `try`-Block versucht die Funktion `a` durch `b` zu teilen und das Ergebnis mit Hilfe eines f-Strings via `print` auszugeben.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- `try_except_else_finally.py` ist ein künstliches Beispiel für diese Struktur.
- Wir erstellen die Funktion `divide_and_print`, welche die beiden Parameter `a` und `b` empfängt, die beide entweder Ganz- oder Fließkommazahlen seien können.
- In einem `try`-Block versucht die Funktion `a` durch `b` zu teilen und das Ergebnis mit Hilfe eines f-Strings via `print` auszugeben.
- Da wir nicht wissen, welchen Wert `b` haben wird, nehmen wir an, dass ein `ZeroDivisionError` auftreten könnte.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Wir erstellen die Funktion `divide_and_print`, welche die beiden Parameter `a` und `b` empfängt, die beide entweder Ganz- oder Fließkommazahlen seien können.
- In einem `try`-Block versucht die Funktion `a` durch `b` zu teilen und das Ergebnis mit Hilfe eines f-Strings via `print` auszugeben.
- Da wir nicht wissen, welchen Wert `b` haben wird, nehmen wir an, dass ein `ZeroDivisionError` auftreten könnte.
- Im dazugehörigen `except`-Block geben wir dann eine passende Erklärung aus.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- In einem `try`-Block versucht die Funktion `a` durch `b` zu teilen und das Ergebnis mit Hilfe eines f-Strings via `print` auszugeben.
- Da wir nicht wissen, welchen Wert `b` haben wird, nehmen wir an, dass ein `ZeroDivisionError` auftreten könnte.
- Im dazugehörigen `except`-Block geben wir dann eine passende Erklärung aus.
- In diesem Fall würde das Divisionsergebnis nicht ausgegeben, da der `try`-Block abbricht während der f-String interpoliert wird.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Da wir nicht wissen, welchen Wert `b` haben wird, nehmen wir an, dass ein `ZeroDivisionError` auftreten könnte.
- Im dazugehörigen `except`-Block geben wir dann eine passende Erklärung aus.
- In diesem Fall würde das Divisionsergebnis nicht ausgegeben, da der `try`-Block abbricht während der f-String interpoliert wird.
- Wir wollen auch mögliche `TypeErrors` abfangen.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
        ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Im dazugehörigen `except`-Block geben wir dann eine passende Erklärung aus.
- In diesem Fall würde das Divisionsergebnis nicht ausgegeben, da der `try`-Block abbricht während der f-String interpoliert wird.
- Wir wollen auch mögliche `TypeErrors` abfangen.
- Wenn so ein Fehler auftritt, dann weil unsere Funktion mit einem Argument aufgerufen wird, das weder `int` noch `float` ist und das Division nicht unterstützt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- In diesem Fall würde das Divisionsergebnis nicht ausgegeben, da der `try`-Block abbricht während der f-String interpoliert wird.
- Wir wollen auch mögliche `TypeErrors` abfangen.
- Wenn so ein Fehler auftritt, dann weil unsere Funktion mit einem Argument aufgerufen wird, das weder `int` noch `float` ist und das Division nicht unterstützt.
- Das ist ein typisches Beispiel für Fehler, die wir lieber nicht abfangen sollten.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
        ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Wir wollen auch mögliche `TypeErrors` abfangen.
- Wenn so ein Fehler auftritt, dann weil unsere Funktion mit einem Argument aufgerufen wird, das weder `int` noch `float` ist und das Division nicht unterstützt.
- Das ist ein typisches Beispiel für Fehler, die wir lieber nicht abfangen sollten.
- So ein Fehler kann nur passieren, wenn ein Programmierer unsere Funktion falsch aufruft.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
        ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Wenn so ein Fehler auftritt, dann weil unsere Funktion mit einem Argument aufgerufen wird, das weder `int` noch `float` ist und das Division nicht unterstützt.
- Das ist ein typisches Beispiel für Fehler, die wir lieber nicht abfangen sollten.
- So ein Fehler kann nur passieren, wenn ein Programmierer unsere Funktion falsch aufruft.
- Wir fangen den Fehler nur für das Beispiel ab und drucken eine entsprechende Nachricht im `except`-Block.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Das ist ein typisches Beispiel für Fehler, die wir lieber nicht abfangen sollten.
- So ein Fehler kann nur passieren, wenn ein Programmierer unsere Funktion falsch aufruft.
- Wir fangen den Fehler nur für das Beispiel ab und drucken eine entsprechende Nachricht im `except`-Block.
- Wir haben auch einen `else`-Block, in dem angezeigt wird, das kein Fehler aufgetreten ist.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- So ein Fehler kann nur passieren, wenn ein Programmierer unsere Funktion falsch aufruft.
- Wir fangen den Fehler nur für das Beispiel ab und drucken eine entsprechende Nachricht im `except`-Block.
- Wir haben auch einen `else`-Block, in dem angezeigt wird, das kein Fehler aufgetreten ist.
- Dieser Kode wird nur ausgeführt, wenn weder ein `ZeroDivisionError`, noch ein `TypeError`, und auch keine andere Ausnahme aufgetreten ist.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- So ein Fehler kann nur passieren, wenn ein Programmierer unsere Funktion falsch aufruft.
- Wir fangen den Fehler nur für das Beispiel ab und drucken eine entsprechende Nachricht im `except`-Block.
- Wir haben auch einen `else`-Block, in dem angezeigt wird, das kein Fehler aufgetreten ist.
- Dieser Kode wird nur ausgeführt, wenn weder ein `ZeroDivisionError`, noch ein `TypeError`, und auch keine andere Ausnahme aufgetreten ist.
- Zuletzt kommt noch ein `finally`-Block.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
        ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Wir fangen den Fehler nur für das Beispiel ab und drucken eine entsprechende Nachricht im `except`-Block.
- Wir haben auch einen `else`-Block, in dem angezeigt wird, das kein Fehler aufgetreten ist.
- Dieser Kode wird nur ausgeführt, wenn weder ein `ZeroDivisionError`, noch ein `TypeError`, und auch keine andere Ausnahme aufgetreten ist.
- Zuletzt kommt noch ein `finally`-Block.
- Dieser wird immer ausgeführt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
  ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
    ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
    ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Wir haben auch einen `else`-Block, in dem angezeigt wird, das kein Fehler aufgetreten ist.
- Dieser Kode wird nur ausgeführt, wenn weder ein `ZeroDivisionError`, noch ein `TypeError`, und auch keine andere Ausnahme aufgetreten ist.
- Zuletzt kommt noch ein `finally`-Block.
- Dieser wird immer ausgeführt.
- Selbst wenn ein Fehler innerhalb einer der `except`-Blöcke auftritt wird dieser Kode ausgeführt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
        ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Dieser Kode wird nur ausgeführt, wenn weder ein `ZeroDivisionError`, noch ein `TypeError`, und auch keine andere Ausnahme aufgetreten ist.
- Zuletzt kommt noch ein `finally`-Block.
- Dieser wird immer ausgeführt.
- Selbst wenn ein Fehler innerhalb einer der `except`-Blöcke auftritt wird dieser Kode ausgeführt.
- Er gibt nur aus, dass der Kode fertig ist.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
       ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Zuletzt kommt noch ein `finally`-Block.
- Dieser wird immer ausgeführt.
- Selbst wenn ein Fehler innerhalb einer der `except`-Blöcke auftritt wird dieser Kode ausgeführt.
- Er gibt nur aus, dass der Kode fertig ist.
- Nach den ganzen `try-except-else-finally`-Blöcken druckt eine normale Zeile Kode noch eine Nachricht aus.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
        ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Dieser wird immer ausgeführt.
- Selbst wenn ein Fehler innerhalb einer der `except`-Blöcke auftritt wird dieser Kode ausgeführt.
- Er gibt nur aus, dass der Kode fertig ist.
- Nach den ganzen `try`-`except`-`else`-`finally`-Blöcken druckt eine normale Zeile Kode noch eine Nachricht aus.
- Dieser Kode wird nur erreicht, wenn entweder kein Fehler aufgetreten ist oder ein Fehler auftrat, der von einem der `except`-Blöcke behandelt wurde (ohne einen weiteren Fehler auszulösen).

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Nach den ganzen `try`-`except`-`else`-`finally`-Blöcken druckt eine normale Zeile Kode noch eine Nachricht aus.
- Dieser Kode wird nur erreicht, wenn entweder kein Fehler aufgetreten ist oder ein Fehler auftrat, der von einem der `except`-Blöcke behandelt wurde (ohne einen weiteren Fehler auszulösen).
- Für `divide_and_print(10, 5)`, werden das Divisionsergebnis im `try`-Block und die Nachricht im `else`-Block, dem -Block, und vom Ende der Funktion ausgedruckt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
       ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Dieser Kode wird nur erreicht, wenn entweder kein Fehler aufgetreten ist oder ein Fehler auftrat, der von einem der `except`-Blöcke behandelt wurde (ohne einen weiteren Fehler auszulösen).
- Für `divide_and_print(10, 5)`, werden das Divisionsergebnis im `try`-Block und die Nachricht im `else`-Block, dem -Block, und vom Ende der Funktion ausgedruckt.
- `divide_and_print(3, 0)` löst einen `ZeroDivisionError` aus.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Für `divide_and_print(10, 5)`, werden das Divisionsergebnis im `try`-Block und die Nachricht im `else`-Block, dem -Block, und vom Ende der Funktion ausgedruckt.
- `divide_and_print(3, 0)` löst einen `ZeroDivisionError` aus.
- Deshalb wird der `print`-Befehl im `try`-Block nicht ausgeführt, weil ja schon die glslinkfstringf-String interpolation fehlschlägt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Für `divide_and_print(10, 5)`, werden das Divisionsergebnis im `try`-Block und die Nachricht im `else`-Block, dem -Block, und vom Ende der Funktion ausgedruckt.
- `divide_and_print(3, 0)` löst einen `ZeroDivisionError` aus.
- Deshalb wird der `print`-Befehl im `try`-Block nicht ausgeführt, weil ja schon die glslinkfstringf-String interpolation fehlschlägt.
- Der erste `except`-Block, der für `ZeroDivisionErrors` zuständig ist, wird ausgeführt und druckt seine Nachricht.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- `divide_and_print(3, 0)` löst einen `ZeroDivisionError` aus.
- Deshalb wird der `print`-Befehl im `try`-Block nicht ausgeführt, weil ja schon die glslinkfstringf-String interpolation fehlschlägt.
- Der erste `except`-Block, der für `ZeroDivisionErrors` zuständig ist, wird ausgeführt und druckt seine Nachricht.
- Der `else`-Block wird nicht erreicht, aber der `finally`-Block druckt ebenfalls seine Nachricht.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
       ↪ !
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- `divide_and_print(3, 0)` löst einen `ZeroDivisionError` aus.
- Deshalb wird der `print`-Befehl im `try`-Block nicht ausgeführt, weil ja schon die glslinkfstringf-String interpolation fehlschlägt.
- Der erste `except`-Block, der für `ZeroDivisionErrors` zuständig ist, wird ausgeführt und druckt seine Nachricht.
- Der `else`-Block wird nicht erreicht, aber der `finally`-Block druckt ebenfalls seine Nachricht.
- Weil der Fehler ordentlich behandelt wurde, wird auch der `print`-Befehl am Ende der Funktion ausgeführt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Der erste `except`-Block, der für `ZeroDivisionErrors` zuständig ist, wird ausgeführt und druckt seine Nachricht.
- Der `else`-Block wird nicht erreicht, aber der `finally`-Block druckt ebenfalls seine Nachricht.
- Weil der Fehler ordentlich behandelt wurde, wird auch der `print`-Befehl am Ende der Funktion ausgeführt.
- Der Versuch, `divide_and_print("3", 0)` aufzurufen, bedeutet, dass wir absichtlich den Type-Hint in der Funktionsdefinition ignorieren.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Der `else`-Block wird nicht erreicht, aber der `finally`-Block druckt ebenfalls seine Nachricht.
- Weil der Fehler ordentlich behandelt wurde, wird auch der `print`-Befehl am Ende der Funktion ausgeführt.
- Der Versuch, `divide_and_print("3", 0)` aufzurufen, bedeutet, dass wir absichtlich den Type-Hint in der Funktionsdefinition ignorieren.
- Der Python-Interpreter erlaubt uns das ohne Probleme.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Der `else`-Block wird nicht erreicht, aber der `finally`-Block druckt ebenfalls seine Nachricht.
- Weil der Fehler ordentlich behandelt wurde, wird auch der `print`-Befehl am Ende der Funktion ausgeführt.
- Der Versuch, `divide_and_print("3", 0)` aufzurufen, bedeutet, dass wir absichtlich den Type-Hint in der Funktionsdefinition ignorieren.
- Der Python-Interpreter erlaubt uns das ohne Probleme.
- Die Division `a / b` schlägt aber Fehl, weil der tring `"3"` den Divisionsoperator nicht unterstützt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Weil der Fehler ordentlich behandelt wurde, wird auch der `print`-Befehl am Ende der Funktion ausgeführt.
- Der Versuch, `divide_and_print("3", 0)` aufzurufen, bedeutet, dass wir absichtlich den Type-Hint in der Funktionsdefinition ignorieren.
- Der Python-Interpreter erlaubt uns das ohne Probleme.
- Die Division `a / b` schlägt aber Fehl, weil der tring `"3"` den Divisionsoperator nicht unterstützt.
- Das führt zu einem `TypeError`, welche von unserem zweiten `except`-Block eingefangen wird.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Der Versuch, `divide_and_print("3", 0)` aufzurufen, bedeutet, dass wir absichtlich den Type-Hint in der Funktionsdefinition ignorieren.
- Der Python-Interpreter erlaubt uns das ohne Probleme.
- Die Division `a / b` schlägt aber Fehl, weil der tring `"3"` den Divisionsoperator nicht unterstützt.
- Das führt zu einem `TypeError`, welche von unserem zweiten `except`-Block eingefangen wird.
- Der `else`-Block wird also wieder nicht erreicht.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Der Python-Interpreter erlaubt uns das ohne Probleme.
- Die Division `a / b` schlägt aber Fehl, weil der tring `"3"` den Divisionsoperator nicht unterstützt.
- Das führt zu einem `TypeError`, welche von unserem zweiten `except`-Block eingefangen wird.
- Der `else`-Block wird also wieder nicht erreicht.
- Der `finally`-Block und auch das `print` am Ende der Funktion werden aber ausgeführt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Die Division `a / b` schlägt aber Fehl, weil der tring `"3"` den Divisionsoperator nicht unterstützt.
- Das führt zu einem `TypeError`, welche von unserem zweiten `except`-Block eingefangen wird.
- Der `else`-Block wird also wieder nicht erreicht.
- Der `finally`-Block und auch das `print` am Ende der Funktion werden aber ausgeführt.
- Hätten wir Mypy auf den Kode angewendet, dann hätte es uns gewarnt. . .

```
$ mypy try_except_else_finally.py --no-strict-optional --check-untyped-defs
try_except_else_finally.py:26: error: Argument 1 to "divide_and_print" has incompatible type "str"; expected "int | float"  [arg-type]
Found 1 error in 1 file (checked 1 source file)
# mypy 2.4.1 failed with exit code 1.
```

Beispiel

- Das führt zu einem `TypeError`, welche von unserem zweiten `except`-Block eingefangen wird.
- Der `else`-Block wird also wieder nicht erreicht.
- Der `finally`-Block und auch das `print` am Ende der Funktion werden aber ausgeführt.
- Hätten wir Mypy auf den Kode angewendet, dann hätte es uns gewarnt...
- Zuletzt versuchen wir noch `divide_and_print(10 ** 313, 1.0)` aufzurufen, also $\frac{10^{313}}{1}$ zu berechnen.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
        ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Der `else`-Block wird also wieder nicht erreicht.
- Der `finally`-Block und auch das `print` am Ende der Funktion werden aber ausgeführt.
- Hätten wir Mypy auf den Kode angewendet, dann hätte es uns gewarnt. . .
- Zuletzt versuchen wir noch `divide_and_print(10 ** 313, 1.0)` aufzurufen, also $\frac{10^{313}}{1}$ zu berechnen.
- Auf den ersten Blick sieht das OK aus.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Der `finally`-Block und auch das `print` am Ende der Funktion werden aber ausgeführt.
- Hätten wir Mypy auf den Kode angewendet, dann hätte es uns gewarnt. . .
- Zuletzt versuchen wir noch `divide_and_print(10 ** 313, 1.0)` aufzurufen, also $\frac{10^{313}}{1}$ zu berechnen.
- Auf den ersten Blick sieht das OK aus.
- Nun haben wir aber gelernt, dass der Datentyp `float` einen begrenzten Wertebereich hat.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Hätten wir Mypy auf den Kode angewendet, dann hätte es uns gewarnt. . .
- Zuletzt versuchen wir noch `divide_and_print(10 ** 313, 1.0)` aufzurufen, also $\frac{10^{313}}{1}$ zu berechnen.
- Auf den ersten Blick sieht das OK aus.
- Nun haben wir aber gelernt, dass der Datentyp `float` einen begrenzten Wertebereich hat.
- Tatsächlich ist der `int` 10^{313} außerhalb dem Wertebereich, den ein `float` repräsentieren kann.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
       ↪ !
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Zuletzt versuchen wir noch `divide_and_print(10 ** 313, 1.0)` aufzurufen, also $\frac{10^{313}}{1}$ zu berechnen.
- Auf den ersten Blick sieht das OK aus.
- Nun haben wir aber gelernt, dass der Datentyp `float` einen begrenzten Wertebereich hat.
- Tatsächlich ist der `int` 10^{313} außerhalb dem Wertebereich, den ein `float` repräsentieren kann.
- In dem wir die Ganzzahl durch `1.0` dividieren, erzwingen wir eine Umwandlung in `float`.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Auf den ersten Blick sieht das OK aus.
- Nun haben wir aber gelernt, dass der Datentyp `float` einen begrenzten Wertebereich hat.
- Tatsächlich ist der `int` 10^{313} außerhalb dem Wertebereich, den ein `float` repräsentieren kann.
- In dem wir die Ganzzahl durch `1.0` dividieren, erzwingen wir eine Umwandlung in `float`.
- Das führt zu einem `OverflowError`.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Nun haben wir aber gelernt, dass der Datentyp `float` einen begrenzten Wertebereich hat.
- Tatsächlich ist der `int` 10^{313} außerhalb dem Wertebereich, den ein `float` repräsentieren kann.
- In dem wir die Ganzzahl durch `1.0` dividieren, erzwingen wir eine Umwandlung in `float`.
- Das führt zu einem `OverflowError`.
- Wir haben keinen `except`-Block für `OverflowErrors`.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Tatsächlich ist der `int` 10^{313} außerhalb dem Wertebereich, den ein `float` repräsentieren kann.
- In dem wir die Ganzzahl durch `1.0` dividieren, erzwingen wir eine Umwandlung in `float`.
- Das führt zu einem `OverflowError`.
- Wir haben keinen `except`-Block für `OverflowError`s.
- Dadurch wird keine Nachricht im `try`-Block ausgegeben und auch keiner unserer `except`-Blöcke wird erreicht.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- In dem wir die Ganzzahl durch `1.0` dividieren, erzwingen wir eine Umwandlung in `float`.
- Das führt zu einem `OverflowError`.
- Wir haben keinen `except`-Block für `OverflowErrors`.
- Dadurch wird keine Nachricht im `try`-Block ausgegeben und auch keiner unserer `except`-Blöcke wird erreicht.
- Der `else`-Block wird auch nicht ausgeführt.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Das führt zu einem `OverflowError`.
- Wir haben keinen `except`-Block für `OverflowErrors`.
- Dadurch wird keine Nachricht im `try`-Block ausgegeben und auch keiner unserer `except`-Blöcke wird erreicht.
- Der `else`-Block wird auch nicht ausgeführt.
- Der `finally`-Block hingegen schon und seine Nachricht erscheint im Output.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Wir haben keinen `except`-Block für `OverflowError`s.
- Dadurch wird keine Nachricht im `try`-Block ausgegeben und auch keiner unserer `except`-Blöcke wird erreicht.
- Der `else`-Block wird auch nicht ausgeführt.
- Der `finally`-Block hingegen schon und seine Nachricht erscheint im Output.
- Da wir den `OverflowError` aber nicht verarbeitet haben, wird der Kode nach unseren Blöcken am Ende der Funktion nicht erreicht.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ python3 try_except_else_finally.py ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Dadurch wird keine Nachricht im `try`-Block ausgegeben und auch keiner unserer `except`-Blöcke wird erreicht.
- Der `else`-Block wird auch nicht ausgeführt.
- Der `finally`-Block hingegen schon und seine Nachricht erscheint im Output.
- Da wir den `OverflowError` aber nicht verarbeitet haben, wird der Kode nach unseren Blöcken am Ende der Funktion nicht erreicht.
- Stattdessen wird unsere Funktion sofort abgebrochen, nachdem der `finally`-Block fertig ist.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Der `else`-Block wird auch nicht ausgeführt.
- Der `finally`-Block hingegen schon und seine Nachricht erscheint im Output.
- Da wir den `OverflowError` aber nicht verarbeitet haben, wird der Kode nach unseren Blöcken am Ende der Funktion nicht erreicht.
- Stattdessen wird unsere Funktion sofort abgebrochen, nachdem der `finally`-Block fertig ist.
- Da es keinen `try`-`except`-Block gibt, der den `OverflowErrors` fangen kann, wird der ganze Python-Interpreter-Prozess beendet.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)           # This works just fine.
divide_and_print(3, 0)            # This yields a ZeroDivisionError.
divide_and_print("3", 0)          # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

Beispiel

- Da wir den `OverflowError` aber nicht verarbeitet haben, wird der Kode nach unseren Blöcken am Ende der Funktion nicht erreicht.
- Stattdessen wird unsere Funktion sofort abgebrochen, nachdem der `finally`-Block fertig ist.
- Da es keinen `try-except`-Block gibt, der den `OverflowErrors` fangen kann, wird der ganze Python-Interpreter-Prozess beendet.
- Es erscheint wieder ein Stack-Trace, der uns zeigt, wo der Fehler aufgetreten ist und was passiert ist.

```python
"""Demonstrate the try-except-else-finally clause."""


def divide_and_print(a: int | float, b: int | float) -> None:
    """
    Divide the numbers `a` and `b` and print the result.

    :param a: the dividend
    :param b: the divisor
    """
    try:  # We try to do some computation that may cause an Exception.
        print(f"{a / b = }", flush=True)  # Divide and print.
    except ZeroDivisionError as zd:  # Is b == 0 ?
        print(f"We got a ZeroDivisionError when doing {a} / {b}: {zd}.")
    except TypeError as te:  # Has one of the values the wrong type?
        print(f"We got a TypeError when computing {a} / {b}: {te}.")
    else:  # Only called when no Exception occurred.
        print(f"No error occurred when computing {a} / {b}.")
    finally:  # This code is always called.
        print(f"We are finally done with division-by-{b}.", flush=True)
    print(f"This code comes after all the {a} by {b} division code.")


divide_and_print(10, 5)            # This works just fine.
divide_and_print(3, 0)             # This yields a ZeroDivisionError.
divide_and_print("3", 0)           # This yields a TypeError.
divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError!
```

↓ `python3 try_except_else_finally.py` ↓

```
a / b = 2.0
No error occurred when computing 10 / 5.
We are finally done with division-by-5.
This code comes after all the 10 by 5 division code.
We got a ZeroDivisionError when doing 3 / 0: division by zero.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We got a TypeError when computing 3 / 0: unsupported operand type(s) for
    ↪ /: 'str' and 'int'.
We are finally done with division-by-0.
This code comes after all the 3 by 0 division code.
We are finally done with division-by-1.0.
Traceback (most recent call last):
  File "{...}/exceptions/try_except_else_finally.py", line 27, in <
      ↪ module>
    divide_and_print(10 ** 313, 1.0)  # Causes an uncaught OverflowError
      ↪ !
    ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
  File "{...}/exceptions/try_except_else_finally.py", line 12, in
      ↪ divide_and_print
    print(f"{a / b = }", flush=True)  # Divide and print.
          ~~~~~~^~~
OverflowError: int too large to convert to float
# 'python3 try_except_else_finally.py' failed with exit code 1.
```

`with`-Block und Context Manager



Einleitendes Beispiel: Datei-I/O



- Der `try-finally`-Block erlaubt uns, sicherzustellen das bestimmte Aktionen ausgeführt werden, gleichgültig ob Kode fehlschlägt und Ausnahmen auslöst.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Der `try-finally`-Block erlaubt uns, sicherzustellen das bestimmte Aktionen ausgeführt werden, gleichgültig ob Kode fehlschlägt und Ausnahmen auslöst.
- Ein Use Case ist der Umgang mit Ressourcen, die explizit geschlossen oder freigegeben werden müssen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Der `try-finally`-Block erlaubt uns, sicherzustellen das bestimmte Aktionen ausgeführt werden, gleichgültig ob Kode fehlschlägt und Ausnahmen auslöst.
- Ein Use Case ist der Umgang mit Ressourcen, die explizit geschlossen oder freigegeben werden müssen.
- Ein typisches Beispiel dafür ist wiederum Datei-Input/Output (I/O).

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Der `try-finally`-Block erlaubt uns, sicherzustellen das bestimmte Aktionen ausgeführt werden, gleichgültig ob Kode fehlschlägt und Ausnahmen auslöst.
- Ein Use Case ist der Umgang mit Ressourcen, die explizit geschlossen oder freigegeben werden müssen.
- Ein typisches Beispiel dafür ist wiederum Datei-Input/Output (I/O).
- Schauen wir uns genau dieses Beispiel in Programm `file_try_finally.py` einmal an.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                  # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                  # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Der `try-finally`-Block erlaubt uns, sicherzustellen das bestimmte Aktionen ausgeführt werden, gleichgültig ob Kode fehlschlägt und Ausnahmen auslöst.
- Ein Use Case ist der Umgang mit Ressourcen, die explizit geschlossen oder freigegeben werden müssen.
- Ein typisches Beispiel dafür ist wiederum Datei-Input/Output (I/O).
- Schauen wir uns genau dieses Beispiel in Programm `file_try_finally.py` einmal an.
- In diesem Programm erstellen und öffnen wir eine Textdatei namens `example.txt`.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                  # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                  # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Ein Use Case ist der Umgang mit Ressourcen, die explizit geschlossen oder freigegeben werden müssen.
- Ein typisches Beispiel dafür ist wiederum Datei-Input/Output (I/O).
- Schauen wir uns genau dieses Beispiel in Programm `file_try_finally.py` einmal an.
- In diesem Programm erstellen und öffnen wir eine Textdatei namens `example.txt`.
- Wir scheiben eine Zeile Text in die Datei und dann schließen wir sie.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Ein typisches Beispiel dafür ist wiederum Datei-Input/Output (I/O).
- Schauen wir uns genau dieses Beispiel in Programm `file_try_finally.py` einmal an.
- In diesem Programm erstellen und öffnen wir eine Textdatei namens `example.txt`.
- Wir scheiben eine Zeile Text in die Datei und dann schließen wir sie.
- Danach öffnen wir sie wieder, lesen den Text ein, und geben ihn wieder aus.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                  # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                  # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Schauen wir uns genau dieses Beispiel in Programm `file_try_finally.py` einmal an.
- In diesem Programm erstellen und öffnen wir eine Textdatei namens `example.txt`.
- Wir scheiben eine Zeile Text in die Datei und dann schließen wir sie.
- Danach öffnen wir sie wieder, lesen den Text ein, und geben ihn wieder aus.
- Zum Schluss löschen wir die Datei wieder, damit sie nicht nutzlos herumliegt.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- In diesem Programm erstellen und öffnen wir eine Textdatei namens `example.txt`.
- Wir scheiben eine Zeile Text in die Datei und dann schließen wir sie.
- Danach öffnen wir sie wieder, lesen den Text ein, und geben ihn wieder aus.
- Zum Schluss löschen wir die Datei wieder, damit sie nicht nutzlos herumliegt.
- Um diese Schritte zu implementieren, importieren wir zuerst die notwendigen Funktionen und Typen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O

- Wir scheiben eine Zeile Text in die Datei und dann schließen wir sie.
- Danach öffnen wir sie wieder, lesen den Text ein, und geben ihn wieder aus.
- Zum Schluss löschen wir die Datei wieder, damit sie nicht nutzlos herumliegt.
- Um diese Schritte zu implementieren, importieren wir zuerst die notwendigen Funktionen und Typen.
- Wir importieren den Typ `IO` aus dem `typing`-Modul.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                              # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                         # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Danach öffnen wir sie wieder, lesen den Text ein, und geben ihn wieder aus.
- Zum Schluss löschen wir die Datei wieder, damit sie nicht nutzlos herumliegt.
- Um diese Schritte zu implementieren, importieren wir zuerst die notwendigen Funktionen und Typen.
- Wir importieren den Typ `IO` aus dem `typing`-Modul.
- ist der Basistyp für alle text-basierten I/O-Ströme und wir benutzen ihn später als Type-Hint.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                  # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                  # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Zum Schluss löschen wir die Datei wieder, damit sie nicht nutzlos herumliegt.
- Um diese Schritte zu implementieren, importieren wir zuerst die notwendigen Funktionen und Typen.
- Wir importieren den Typ `IO` aus dem `typing`-Modul.
- ist der Basistyp für alle text-basierten I/O-Ströme und wir benutzen ihn später als Type-Hint.
- Wir werden auch die Funktion `remove` aus dem Modul `os` brauchen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Um diese Schritte zu implementieren, importieren wir zuerst die notwendigen Funktionen und Typen.
- Wir importieren den Typ `IO` aus dem `typing`-Modul.
- ist der Basistyp für alle text-basierten I/O-Ströme und wir benutzen ihn später als Type-Hint.
- Wir werden auch die Funktion `remove` aus dem Modul `os` brauchen.
- Wir beginnen, in dem wir die Datei `example.txt` zum Schreiben öffnen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir importieren den Typ `IO` aus dem `typing`-Modul.
- ist der Basistyp für alle text-basierten I/O-Ströme und wir benutzen ihn später als Type-Hint.
- Wir werden auch die Funktion `remove` aus dem Modul `os` brauchen.
- Wir beginnen, in dem wir die Datei `example.txt` zum Schreiben öffnen.
- Dafür benutzen wir die built-in Funktion `open`.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O

- ist der Basistyp für alle text-basierten I/O-Ströme und wir benutzen ihn später als Type-Hint.
- Wir werden auch die Funktion `remove` aus dem Modul `os` brauchen.
- Wir beginnen, in dem wir die Datei `example.txt` zum Schreiben öffnen.
- Dafür benutzen wir die built-in Funktion `open`.
- Wir übergeben den Dateiname `"example.txt"` als ersten Parameter.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir werden auch die Funktion `remove` aus dem Modul `os` brauchen.
- Wir beginnen, in dem wir die Datei `example.txt` zum Schreiben öffnen.
- Dafür benutzen wir die built-in Funktion `open`.
- Wir übergeben den Dateiname `"example.txt"` als ersten Parameter.
- Der zweite Parameter ist der Modus `mode`, welchen wir auf `"w"` setzen, was „Öffnen um zu Schreiben“ bedeutet.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir beginnen, in dem wir die Datei `example.txt` zum Schreiben öffnen.
- Dafür benutzen wir die built-in Funktion `open`.
- Wir übergeben den Dateiname `"example.txt"` als ersten Parameter.
- Der zweite Parameter ist der Modus `mode`, welchen wir auf `"w"` setzen, was „Öffnen um zu Schreiben“ bedeutet.
- Wenn die Datei noch nicht existiert, dann führt `"w"` dazu, dass sie erstellt wird.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir übergeben den Dateiname `"example.txt"` als ersten Parameter.
- Der zweite Parameter ist der Modus `mode`, welchen wir auf `"w"` setzen, was „Öffnen um zu Schreiben“ bedeutet.
- Wenn die Datei noch nicht existiert, dann führt `"w"` dazu, dass sie erstellt wird.
- Existiert sie bereits, wird ihr Inhalt verworfen und wir fangen am Anfang mit dem Schreiben an.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O

- Der zweite Parameter ist der Modus `mode`, welchen wir auf `"w"` setzen, was „Öffnen um zu Schreiben“ bedeutet.
- Wenn die Datei noch nicht existiert, dann führt `"w"` dazu, dass sie erstellt wird.
- Existiert sie bereits, wird ihr Inhalt verworfen und wir fangen am Anfang mit dem Schreiben an.
- Der Parameter `encoding` wird auf `"UTF-8"` gesetzt, womit festgelegt wird, der unser Text über das UTF-8-Encoding[17,52] in Binärdaten übersetzt wird.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wenn die Datei noch nicht existiert, dann führt `"w"` dazu, dass sie erstellt wird.
- Existiert sie bereits, wird ihr Inhalt verworfen und wir fangen am Anfang mit dem Schreiben an.
- Der Parameter `encoding` wird auf `"UTF-8"` gesetzt, womit festgelegt wird, der unser Text über das UTF-8-Encoding[17,52] in Binärdaten übersetzt wird.
- Alle gespeicherten Daten sind letztendlich binär und das ist das häufigste Format im Internet, in dem Text gespeichert wird.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Existiert sie bereits, wird ihr Inhalt verworfen und wir fangen am Anfang mit dem Schreiben an.
- Der Parameter `encoding` wird auf `"UTF-8"` gesetzt, womit festgelegt wird, der unser Text über das UTF-8-Encoding[17,52] in Binärdaten übersetzt wird.
- Alle gespeicherten Daten sind letztendlich binär und das ist das häufigste Format im Internet, in dem Text gespeichert wird.
- Wenn das öffnen der Textdatei glückt, dann haben wir nun eine Instanz von `IO` in der Variablen `stream_out`.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                  # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                  # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Der Parameter `encoding` wird auf `"UTF-8"` gesetzt, womit festgelegt wird, der unser Text über das UTF-8-Encoding[17,52] in Binärdaten übersetzt wird.
- Alle gespeicherten Daten sind letztendlich binär und das ist das häufigste Format im Internet, in dem Text gespeichert wird.
- Wenn das öffnen der Textdatei glückt, dann haben wir nun eine Instanz von `IO` in der Variablen `stream_out`.
- Wir müssen diesen so genannten Text-Stream auf jeden Fall schließen, gleichgültig, was von jetzt an passiert.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Alle gespeicherten Daten sind letztendlich binär und das ist das häufigste Format im Internet, in dem Text gespeichert wird.
- Wenn das öffnen der Textdatei glückt, dann haben wir nun eine Instanz von `IO` in der Variablen `stream_out`.
- Wir müssen diesen so genannten Text-Stream auf jeden Fall schließen, gleichgültig, was von jetzt an passiert.
- Wir wissen, dass das mit einem `try-finally`-Statement geht.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wenn das öffnen der Textdatei glückt, dann haben wir nun eine Instanz von `IO` in der Variablen `stream_out`.
- Wir müssen diesen so genannten Text-Stream auf jeden Fall schließen, gleichgültig, was von jetzt an passiert.
- Wir wissen, dass das mit einem `try`-`finally`-Statement geht.
- Wir packen daher den Aufruf `stream_out.close()` in den `finally`-Block.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir müssen diesen so genannten Text-Stream auf jeden Fall schließen, gleichgültig, was von jetzt an passiert.
- Wir wissen, dass das mit einem `try`-`finally`-Statement geht.
- Wir packen daher den Aufruf `stream_out.close()` in den `finally`-Block.
- Er wird auf jeden Fall aufgerufen und `stream_out` somit definitiv geschlossen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir wissen, dass das mit einem `try-finally`-Statement geht.
- Wir packen daher den Aufruf `stream_out.close()` in den `finally`-Block.
- Er wird auf jeden Fall aufgerufen und `stream_out` somit definitiv geschlossen.
- In den `try`-Block schreiben wir `stream_out.write("Hello world!")`

```
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir packen daher den Aufruf `stream_out.close()` in den `finally`-Block.
- Er wird auf jeden Fall aufgerufen und `stream_out` somit definitiv geschlossen.
- In den `try`-Block schreiben wir `stream_out.write("Hello world!")`
- Diese Zeile schreibt den String `"Hello world!"` in die Datei.

```
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Er wird auf jeden Fall aufgerufen und `stream_out` somit definitiv geschlossen.
- In den `try`-Block schreiben wir `stream_out.write("Hello world!")`
- Diese Zeile schreibt den String `"Hello world!"` in die Datei.
- Das könnte natürlich schief gehen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- In den `try`-Block schreiben wir `stream_out.write("Hello world!")`
- Diese Zeile schreibt den String `"Hello world!"` in die Datei.
- Das könnte natürlich schief gehen.
- Vielleicht ist ja auf unserer Festplatte nicht mehr genug Platz frei, um diesen String zu speichern.

```
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Diese Zeile schreibt den String `"Hello world!"` in die Datei.
- Das könnte natürlich schief gehen.
- Vielleicht ist ja auf unserer Festplatte nicht mehr genug Platz frei, um diesen String zu speichern.
- Aber selbst wenn es schief geht, der `finally`-Block wird auf jeden Fall ausgeführt und wird die Datei schließen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O

- Das könnte natürlich schief gehen.
- Vielleicht ist ja auf unserer Festplatte nicht mehr genug Platz frei, um diesen String zu speichern.
- Aber selbst wenn es schief geht, der `finally`-Block wird auf jeden Fall ausgeführt und wird die Datei schließen.
- Nach dem Block ist die Datei also wieder geschlossen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Vielleicht ist ja auf unserer Festplatte nicht mehr genug Platz frei, um diesen String zu speichern.
- Aber selbst wenn es schief geht, der `finally`-Block wird auf jeden Fall ausgeführt und wird die Datei schließen.
- Nach dem Block ist die Datei also wieder geschlossen.
- Wir könnten sie in einem Text-Editor öffnen und gucken, ob der Text auch wirklich angekommen ist.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Aber selbst wenn es schief geht, der `finally`-Block wird auf jeden Fall ausgeführt und wird die Datei schließen.
- Nach dem Block ist die Datei also wieder geschlossen.
- Wir könnten sie in einem Text-Editor öffnen und gucken, ob der Text auch wirklich angekommen ist.
- Wir wollen das stattdessen mit Kode machen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Nach dem Block ist die Datei also wieder geschlossen.
- Wir könnten sie in einem Text-Editor öffnen und gucken, ob der Text auch wirklich angekommen ist.
- Wir wollen das stattdessen mit Kode machen.
- Dafür öffnen wir die Datei wieder, diesmal zum Lesen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir könnten sie in einem Text-Editor öffnen und gucken, ob der Text auch wirklich angekommen ist.
- Wir wollen das stattdessen mit Kode machen.
- Dafür öffnen wir die Datei wieder, diesmal zum Lesen.
- Das geht genau wie vorhin, mit der Funktion `open`.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir wollen das stattdessen mit Kode machen.
- Dafür öffnen wir die Datei wieder, diesmal zum Lesen.
- Das geht genau wie vorhin, mit der Funktion `open`.
- Anstatt nun den Wert `mode="w"` zu übergeben, würden wir schreiben `mode="r"`, was „Öffnen zum Lesen“ bedeutet.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Dafür öffnen wir die Datei wieder, diesmal zum Lesen.
- Das geht genau wie vorhin, mit der Funktion `open`.
- Anstatt nun den Wert `mode="w"` zu übergeben, würden wir schreiben `mode="r"`, was „Öffnen zum Lesen“ bedeutet.
- Aber `"r"` ist bereits der Default Wert für Parameter `mode`, so dass wir es einfach weglassen können.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Das geht genau wie vorhin, mit der Funktion `open`.
- Anstatt nun den Wert `mode="w"` zu übergeben, würden wir schreiben `mode="r"`, was „Öffnen zum Lesen“ bedeutet.
- Aber `"r"` ist bereits der Default Wert für Parameter `mode`, so dass wir es einfach weglassen können.
- Wir machen also `stream_in = open("example.txt", encoding="UTF-8")`.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Anstatt nun den Wert `mode="w"` zu übergeben, würden wir schreiben `mode="r"`, was „Öffnen zum Lesen“ bedeutet.
- Aber `"r"` ist bereits der Default Wert für Parameter `mode`, so dass wir es einfach weglassen können.
- Wir machen also `stream_in = open("example.txt", encoding="UTF-8")`.
- Nachdem die Datei geöffnet ist, müssen wir natürlich sicherstellen, dass sie wieder geschlossen wird.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Aber `"r"` ist bereits der Default Wert für Parameter `mode`, so dass wir es einfach weglassen können.
- Wir machen also `stream_in = open("example.txt", encoding="UTF-8")`.
- Nachdem die Datei geöffnet ist, müssen wir natürlich sicherstellen, dass sie wieder geschlossen wird.
- Das machen wir mit einem `try-finally`-Statement, wobei `stream_in.close()` wieder in den `finally`-Block kommt.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Wir machen also `stream_in = open("example.txt", encoding="UTF-8")`.
- Nachdem die Datei geöffnet ist, müssen wir natürlich sicherstellen, dass sie wieder geschlossen wird.
- Das machen wir mit einem `try-finally`-Statement, wobei `stream_in.close()` wieder in den `finally`-Block kommt.
- Die Textzeile, die wir vorhin geschrieben haben, können wir nun im `try`-Block wieder lesen.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Nachdem die Datei geöffnet ist, müssen wir natürlich sicherstellen, dass sie wieder geschlossen wird.
- Das machen wir mit einem `try`-`finally`-Statement, wobei `stream_in.close()` wieder in den `finally`-Block kommt.
- Die Textzeile, die wir vorhin geschrieben haben, können wir nun im `try`-Block wieder lesen.
- Das geht via `stream_in.readline()`.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                  # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                  # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Das machen wir mit einem `try-finally`-Statement, wobei `stream_in.close()` wieder in den `finally`-Block kommt.
- Die Textzeile, die wir vorhin geschrieben haben, können wir nun im `try`-Block wieder lesen.
- Das geht via `stream_in.readline()`.
- Wir geben das Ergebnis direkt mit `print` wieder aus.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Die Textzeile, die wir vorhin geschrieben haben, können wir nun im `try`-Block wieder lesen.
- Das geht via `stream_in.readline()`.
- Wir geben das Ergebnis direkt mit `print` wieder aus.
- Am Ende des Programms löschen wir die Datei `example.txt` mit `remove("example.txt")`.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Das geht via `stream_in.readline()`.
- Wir geben das Ergebnis direkt mit `print` wieder aus.
- Am Ende des Programms löschen wir die Datei `example.txt` mit `remove("example.txt")`.
- Dafür hatten wir ja `remove` aus dem Modul `os` importiert.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Das geht via `stream_in.readline()`.
- Wir geben das Ergebnis direkt mit `print` wieder aus.
- Am Ende des Programms löschen wir die Datei `example.txt` mit `remove("example.txt")`.
- Dafür hatten wir ja `remove` aus dem Modul `os` importiert.
- Das Programm funktioniert genau wie erwartet.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Einleitendes Beispiel: Datei-I/O



- Das geht via `stream_in.readline()`.
- Wir geben das Ergebnis direkt mit `print` wieder aus.
- Am Ende des Programms löschen wir die Datei `example.txt` mit `remove("example.txt")`.
- Dafür hatten wir ja `remove` aus dem Modul `os` importiert.
- Das Programm funktioniert genau wie erwartet.
- Alles ist wunderbar.

```python
"""Use the `try-finally` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.
from typing import IO  # IO is the text stream object

# We open the text file "example.txt" for writing.
stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
try:  # If we succeed opening the file for writing, then...
    stream_out.write("Hello world!")  # ...we write one line of text
finally:                # ...and we will definitely get here....
    stream_out.close()  # and close the file again.

# We now open the text file "example.txt" for reading.
stream_in: IO = open("example.txt", encoding="UTF-8")
try:  # If we succeed opening the file for reading, then...
    print(stream_in.readline())  # ...we read one line of text
finally:                # ...and we will definitely get here....
    stream_in.close()  # and close the file again.

remove("example.txt")  # Delete the file "example.txt".
```

↓ `python3 file_try_finally.py` ↓

```
Hello world!
```

Was sagen Linter dazu?

- Wir führen natürlich immer eine statische Kodeanalyse mit Lintern durch.

Was sagen Linter dazu?

- Wir führen natürlich immer eine statische Kodeanalyse mit Lintern durch.
- Das sagt Ruff zu unserem Programm.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
  ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
  ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
  ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
  ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
  ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
  ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
  ↪ T201,TRY003,UP035,W --line-length 79 file_try_finally.py
SIM115 Use a context manager for opening files
 --> file_try_finally.py:7:18
6 | # We open the text file "example.txt" for writing.
7 | stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
  |                  ^^^^
8 | try:  # If we succeed opening the file for writing, then...
9 |     stream_out.write("Hello world!")  # ...we write one line of text
SIM115 Use a context manager for opening files
  --> file_try_finally.py:14:17
13 | # We now open the text file "example.txt" for reading.
14 | stream_in: IO = open("example.txt", encoding="UTF-8")
   |                 ^^^^
15 | try:  # If we succeed opening the file for reading, then...
16 |     print(stream_in.readline())  # ...we read one line of text
Found 2 errors.
# ruff 0.17.3 failed with exit code 1.
```

Was sagen Linter dazu?

- Wir führen natürlich immer eine statische Kodeanalyse mit Lintern durch.
- Das sagt Ruff zu unserem Programm.
- Und das sagt Pylint zu unserem Programm.

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 file_try_finally.py
SIM115 Use a context manager for opening files
 --> file_try_finally.py:7:18
6 | # We open the text file "example.txt" for writing.
7 | stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
  |                  ^^^^
8 | try:  # If we succeed opening the file for writing, then...
9 |     stream_out.write("Hello world!")  # ...we write one line of text
SIM115 Use a context manager for opening files
  --> file_try_finally.py:14:17
13 | # We now open the text file "example.txt" for reading.
14 | stream_in: IO = open("example.txt", encoding="UTF-8")
   |                 ^^^^
15 | try:  # If we succeed opening the file for reading, then...
16 |     print(stream_in.readline())  # ...we read one line of text
Found 2 errors.
# ruff 0.17.3 failed with exit code 1.
```

```
$ pylint file_try_finally.py --disable=C0103,C0302,C0325,R0801,R0901,
    ↪ R0902,R0903,R0911,R0912,R0913,R0914,R0915,R1702,R1728,W0212,W0238,
    ↪ W0703
************* Module file_try_finally
file_try_finally.py:7:17: R1732: Consider using 'with' for resource-
    ↪ allocating operations (consider-using-with)
file_try_finally.py:14:16: R1732: Consider using 'with' for resource-
    ↪ allocating operations (consider-using-with)
-----------------------------------
Your code has been rated at 8.18/10
# pylint 4.1.3 failed with exit code 8.
```

Was sagen Linter dazu?

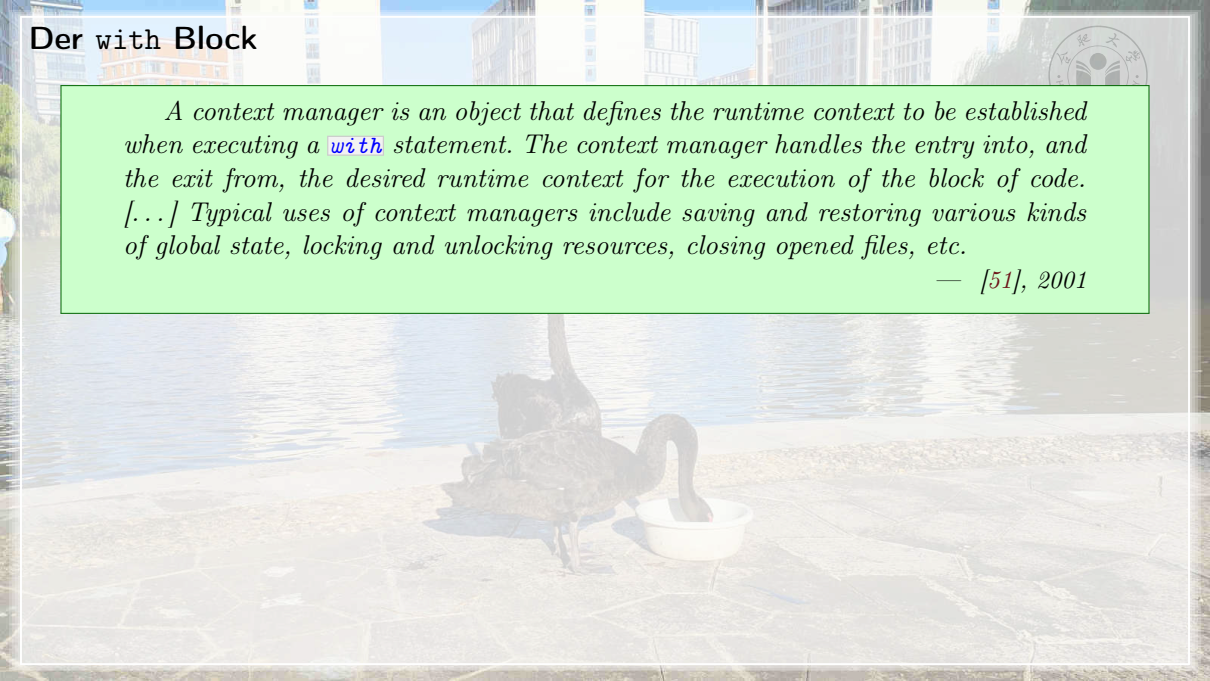
- Wir führen natürlich immer eine statische Kodeanalyse mit Lintern durch.
- Das sagt Ruff zu unserem Programm.
- Und das sagt Pylint zu unserem Programm.
- Beide meinen das selbe.

```
$ pylint file_try_finally.py --disable=C0103,C0302,C0325,R0801,R0901,
    ↪ R0902,R0903,R0911,R0912,R0913,R0914,R0915,R1702,R1728,W0212,W0238,
    ↪ W0703
************* Module file_try_finally
file_try_finally.py:7:17: R1732: Consider using 'with' for resource-
    ↪ allocating operations (consider-using-with)
file_try_finally.py:14:16: R1732: Consider using 'with' for resource-
    ↪ allocating operations (consider-using-with)
-----------------------------------
Your code has been rated at 8.18/10
# pylint 4.1.3 failed with exit code 8.
```

```
$ ruff check --target-version py312 --select=A,AIR,ANN,ASYNC,B,BLE,C,C4,
    ↪ COM,D,DJ,DTZ,E,ERA,EXE,F,FA,FIX,FLY,FURB,G,I,ICN,INP,ISC,INT,LOG,N
    ↪ ,NPY,PERF,PIE,PLC,PLE,PLR,PLW,PT,PYI,Q,RET,RSE,RUF,S,SIM,T,T10,TD,
    ↪ TID,TRY,UP,W,YTT --ignore=A005,ANN001,ANN002,ANN003,ANN204,ANN401,
    ↪ B008,B009,B010,C901,D203,D208,D212,D401,D407,D413,INP001,N801,
    ↪ PLC2801,PLR0904,PLR0911,PLR0912,PLR0913,PLR0914,PLR0915,PLR0916,
    ↪ PLR0917,PLR1702,PLR2004,PLR6301,PT011,PT012,PT013,PYI041,RUF100,S,
    ↪ T201,TRY003,UP035,W --line-length 79 file_try_finally.py
SIM115 Use a context manager for opening files
 --> file_try_finally.py:7:18
6 | # We open the text file "example.txt" for writing.
7 | stream_out: IO = open("example.txt", mode="w", encoding="UTF-8")
  |                  ^^^^
8 | try:  # If we succeed opening the file for writing, then...
9 |     stream_out.write("Hello world!")  # ...we write one line of text
SIM115 Use a context manager for opening files
  --> file_try_finally.py:14:17
13 | # We now open the text file "example.txt" for reading.
14 | stream_in: IO = open("example.txt", encoding="UTF-8")
   |                 ^^^^
15 | try:  # If we succeed opening the file for reading, then...
16 |     print(stream_in.readline())  # ...we read one line of text
Found 2 errors.
# ruff 0.17.3 failed with exit code 1.
```

Der `with` Block

> *A context manager is an object that defines the runtime context to be established when executing a* `with` *statement. The context manager handles the entry into, and the exit from, the desired runtime context for the execution of the block of code. [. . .] Typical uses of context managers include saving and restoring various kinds of global state, locking and unlocking resources, closing opened files, etc.*
>
> — *[51], 2001*



Der `with` Block

- Das `with`-Statement hat die folgende Syntax.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

Der `with` Block

- Das `with`-Statement hat die folgende Syntax.
- `expression` ist ein Ausdruck der ein so genanntes Context-Manager-Objekt[6,41,45,51] zurückliefert.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1 # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3 # block of code
    statement 4
```

Der `with` Block

- Das `with`-Statement hat die folgende Syntax.
- `expression` ist ein Ausdruck der ein so genanntes Context-Manager-Objekt[6,41,45,51] zurückliefert.
- Wir haben noch nicht Klassen behandelt, deshalb fehlt uns noch etwas Hintergrundwissen, um genau zu verstehen, wie ein Context-Manager funktioniert.

```python
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

Der `with` Block

- Das `with`-Statement hat die folgende Syntax.
- `expression` ist ein Ausdruck der ein so genanntes Context-Manager-Objekt[6,41,45,51] zurückliefert.
- Wir haben noch nicht Klassen behandelt, deshalb fehlt uns noch etwas Hintergrundwissen, um genau zu verstehen, wie ein Context-Manager funktioniert.
- Vereinfacht ausgedrückt ist ein Context-Manager ein Objekt mit zwei speziellen Methoden.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

Der `with` Block

- Wir haben noch nicht Klassen behandelt, deshalb fehlt uns noch etwas Hintergrundwissen, um genau zu verstehen, wie ein Context-Manager funktioniert.
- Vereinfacht ausgedrückt ist ein Context-Manager ein Objekt mit zwei speziellen Methoden.
- Die erste wird am Anfang des `with`-Blocks aufgerufen und ihr Ergebnis wird in der Variable `variable` gespeichert, wenn der `with`-Block die Form `with expression as variable:` hat.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

Der `with` Block

- Vereinfacht ausgedrückt ist ein Context-Manager ein Objekt mit zwei speziellen Methoden.
- Die erste wird am Anfang des `with`-Blocks aufgerufen und ihr Ergebnis wird in der Variable `variable` gespeichert, wenn der `with`-Block die Form `with expression as variable:` hat.
- Die zweite Methode wird nach dem Ende des `with`-Blocks aufgerufen.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

Der `with` Block

- Vereinfacht ausgedrückt ist ein Context-Manager ein Objekt mit zwei speziellen Methoden.
- Die erste wird am Anfang des `with`-Blocks aufgerufen und ihr Ergebnis wird in der Variable `variable` gespeichert, wenn der `with`-Block die Form `with expression as variable:` hat.
- Die zweite Methode wird nach dem Ende des `with`-Blocks aufgerufen.
- Das passiert immer, gleichgültig ob eine Ausnahme im Block passiert ist oder nicht.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

Der `with` Block

- Die erste wird am Anfang des `with`-Blocks aufgerufen und ihr Ergebnis wird in der Variable `variable` gespeichert, wenn der `with`-Block die Form `with expression as variable:` hat.
- Die zweite Methode wird nach dem Ende des `with`-Blocks aufgerufen.
- Das passiert immer, gleichgültig ob eine Ausnahme im Block passiert ist oder nicht.
- Die Syntax ist daher in etwa äquivalent dazu, die erste Methode vor einem `try`-Block aufzurufen und die zweite im `finally`-Block.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

Der `with` Block

- Die zweite Methode wird nach dem Ende des `with`-Blocks aufgerufen.
- Das passiert immer, gleichgültig ob eine Ausnahme im Block passiert ist oder nicht.
- Die Syntax ist daher in etwa äquivalent dazu, die erste Methode vor einem `try`-Block aufzurufen und die zweite im `finally`-Block.
- Sie ist aber kürzer und sieht schöner aus.

```
"""The syntax of with-blocks in Python."""

# The 'expression' is some expression that usually acquires a resource
# which must eventually be released.
# The 'with' block does this automatically, even if an uncaught
# exception is raised within it.
# It is basically a fancy try-finally block.

with expression as variable:
    # 'variable' here usually stores the acquired resource.
    # It could be a handle to a file opened for writing, for example.
    statement 1  # block of code that works with variable.
    statement 2
    #...

# or

with expression:
    # Here we do not explicitly work with the resource that was
    # acquired. This makes sense, for example, if the resource is a lock
    # to a critical section.
    statement 3  # block of code
    statement 4
```

Was sagen Linter dazu?



- Viele von Python's Ressourcen-bezogenen APIs wurden als Context-Managers realisiert.

Was sagen Linter dazu?

- Viele von Python's Ressourcen-bezogenen APIs wurden als Context-Managers realisiert.
- Das trifft auch auf die Datei-I/O-API zu, wie uns die Linter ja mitgeteilt haben.

```python
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?



- Viele von Python's Ressourcen-bezogenen APIs wurden als Context-Managers realisiert.
- Das trifft auch auf die Datei-I/O-API zu, wie uns die Linter ja mitgeteilt haben.
- Wir schreiben jetzt das Programm unter Benutzung von `with`-Blöcken neu als `file_with.py`.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?



- Viele von Python's Ressourcen-bezogenen APIs wurden als Context-Managers realisiert.
- Das trifft auch auf die Datei-I/O-API zu, wie uns die Linter ja mitgeteilt haben.
- Wir schreiben jetzt das Programm unter Benutzung von `with`-Blöcken neu als `file_with.py`.
- Das Erste, was uns auffällt, ist dass das Programm viel kürzer ist.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Das trifft auch auf die Datei-I/O-API zu, wie uns die Linter ja mitgeteilt haben.
- Wir schreiben jetzt das Programm unter Benutzung von `with`-Blöcken neu als `file_with.py`.
- Das Erste, was uns auffällt, ist dass das Programm viel kürzer ist.
- Es sind jetzt 13 Zeilen Kode anstatt von 20.
- Wir müssen die `close`-Methoden der Textströme nicht mehr selbst aufrufen.

```python
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Das trifft auch auf die Datei-I/O-API zu, wie uns die Linter ja mitgeteilt haben.
- Wir schreiben jetzt das Programm unter Benutzung von `with`-Blöcken neu als `file_with.py`.
- Das Erste, was uns auffällt, ist dass das Programm viel kürzer ist.
- Es sind jetzt 13 Zeilen Kode anstatt von 20.
- Wir müssen die `close`-Methoden der Textströme nicht mehr selbst aufrufen.
- Sie werden automatisch am Ende der `with`-Blöcke aufgerufen.

```python
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?



- Wir schreiben jetzt das Programm unter Benutzung von `with`-Blöcken neu als `file_with.py`.
- Das Erste, was uns auffällt, ist dass das Programm viel kürzer ist.
- Es sind jetzt 13 Zeilen Kode anstatt von 20.
- Wir müssen die `close`-Methoden der Textströme nicht mehr selbst aufrufen.
- Sie werden automatisch am Ende der `with`-Blöcke aufgerufen.
- Aber eins nach dem Anderen.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Das Erste, was uns auffällt, ist dass das Programm viel kürzer ist.
- Es sind jetzt 13 Zeilen Kode anstatt von 20.
- Wir müssen die `close`-Methoden der Textströme nicht mehr selbst aufrufen.
- Sie werden automatisch am Ende der `with`-Blöcke aufgerufen.
- Aber eins nach dem Anderen.
- Wenn wir unser neues Programm lesen, fällt uns auf, dass wir den Typ `IO` nicht mehr importieren.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Wir müssen die `close`-Methoden der Textströme nicht mehr selbst aufrufen.
- Sie werden automatisch am Ende der `with`-Blöcke aufgerufen.
- Aber eins nach dem Anderen.
- Wenn wir unser neues Programm lesen, fällt uns auf, dass wir den Typ `IO` nicht mehr importieren.
- Der Grund ist, dass die Syntax des `with`-Blocks, dessen erste Zeile mit einem Doppelpunkt `:` ended, es uns nicht erlaubt, einen Type-Hint zu spezifizieren.

```python
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Sie werden automatisch am Ende der `with`-Blöcke aufgerufen.
- Aber eins nach dem Anderen.
- Wenn wir unser neues Programm lesen, fällt uns auf, dass wir den Typ `IO` nicht mehr importieren.
- Der Grund ist, dass die Syntax des `with`-Blocks, dessen erste Zeile mit einem Doppelpunkt `:` ended, es uns nicht erlaubt, einen Type-Hint zu spezifizieren.
- Dann brauchen wir den Datentyp auch nicht importieren.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?



- Aber eins nach dem Anderen.
- Wenn wir unser neues Programm lesen, fällt uns auf, dass wir den Typ `IO` nicht mehr importieren.
- Der Grund ist, dass die Syntax des `with`-Blocks, dessen erste Zeile mit einem Doppelpunkt `:` ended, es uns nicht erlaubt, einen Type-Hint zu spezifizieren.
- Dann brauchen wir den Datentyp auch nicht importieren.
- Wir würden uns darauf verlassen, dass Mypy den richtigen Typ selber herausbekommt...

```python
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Wenn wir unser neues Programm lesen, fällt uns auf, dass wir den Typ `IO` nicht mehr importieren.
- Der Grund ist, dass die Syntax des `with`-Blocks, dessen erste Zeile mit einem Doppelpunkt `:` ended, es uns nicht erlaubt, einen Type-Hint zu spezifizieren.
- Dann brauchen wir den Datentyp auch nicht importieren.
- Wir würden uns darauf verlassen, dass Mypy den richtigen Typ selber herausbekommt...
- Wir müssen aber immer noch `remove` zum löschen von Dateien importieren.

```python
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?



- Der Grund ist, dass die Syntax des `with`-Blocks, dessen erste Zeile mit einem Doppelpunkt `:` ended, es uns nicht erlaubt, einen Type-Hint zu spezifizieren.
- Dann brauchen wir den Datentyp auch nicht importieren.
- Wir würden uns darauf verlassen, dass Mypy den richtigen Typ selber herausbekommt...
- Wir müssen aber immer noch `remove` zum löschen von Dateien importieren.
- Dann beginnen wir schon unseren ersten `with`-Block.

```python
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?



- Dann brauchen wir den Datentyp auch nicht importieren.
- Wir würden uns darauf verlassen, dass Mypy den richtigen Typ selber herausbekommt...
- Wir müssen aber immer noch `remove` zum löschen von Dateien importieren.
- Dann beginnen wir schon unseren ersten `with`-Block.
- Sein Sinn ist es, die Datei zu erstellen und den Text hineinzuschreiben.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Wir würden uns darauf verlassen, dass Mypy den richtigen Typ selber herausbekommt...
- Wir müssen aber immer noch `remove` zum löschen von Dateien importieren.
- Dann beginnen wir schon unseren ersten `with`-Block.
- Sein Sinn ist es, die Datei zu erstellen und den Text hineinzuschreiben.
- Wir verwenden den selben `open`-Aufruf wie vorher und speichern sein Ergebnis in einer Variable `stream_out`, in dem wir `as stream_out` schreiben.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Wir müssen aber immer noch `remove` zum löschen von Dateien importieren.
- Dann beginnen wir schon unseren ersten `with`-Block.
- Sein Sinn ist es, die Datei zu erstellen und den Text hineinzuschreiben.
- Wir verwenden den selben `open`-Aufruf wie vorher und speichern sein Ergebnis in einer Variable `stream_out`, in dem wir `as stream_out` schreiben.
- In den `with`-Block tuen wir nun den Kode, der vorher im `try`-Block war, also das `stream_out.write(...`.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Dann beginnen wir schon unseren ersten `with`-Block.
- Sein Sinn ist es, die Datei zu erstellen und den Text hineinzuschreiben.
- Wir verwenden den selben `open`-Aufruf wie vorher und speichern sein Ergebnis in einer Variable `stream_out`, in dem wir `as stream_out` schreiben.
- In den `with`-Block tuen wir nun den Kode, der vorher im `try`-Block war, also das `stream_out.write(...`.
- Wir können uns darauf verlassen, dass der `with`-Block den Text-Strom an seinem Ende selber schließt.

```
1  """Use the `with` statement to write to and read from a file."""
2
3  from os import remove  # Needed to delete a file.
4
5  # We open the text file "example.txt" for writing.
6  with open("example.txt", mode="w", encoding="UTF-8") as stream_
7      stream_out.write("Hello world!")  # Write one line of text.
8
9  # We now open the text file "example.txt" for reading.
10 with open("example.txt", encoding="UTF-8") as stream_in:
11     print(stream_in.readline())  # Read one line of text.
12
13 remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
1  Hello world!
```

Was sagen Linter dazu?

- Sein Sinn ist es, die Datei zu erstellen und den Text hineinzuschreiben.
- Wir verwenden den selben `open`-Aufruf wie vorher und speichern sein Ergebnis in einer Variable `stream_out`, in dem wir `as stream_out` schreiben.
- In den `with`-Block tuen wir nun den Kode, der vorher im `try`-Block war, also das `stream_out.write(...`.
- Wir können uns darauf verlassen, dass der `with`-Block den Text-Strom an seinem Ende selber schließt.
- Deshalb können wir direkt mit dem zweiten `with`-Block weitermachen.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?



- Wir verwenden den selben `open`-Aufruf wie vorher und speichern sein Ergebnis in einer Variable `stream_out`, in dem wir `as stream_out` schreiben.
- In den `with`-Block tuen wir nun den Kode, der vorher im `try`-Block war, also das `stream_out.write(...`.
- Wir können uns darauf verlassen, dass der `with`-Block den Text-Strom an seinem Ende selber schließt.
- Deshalb können wir direkt mit dem zweiten `with`-Block weitermachen.
- Hier wollen wir eine einzelne Zeile Text aus der Datei lesen.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- In den `with`-Block tuen wir nun den Kode, der vorher im `try`-Block war, also das `stream_out.write(...`.
- Wir können uns darauf verlassen, dass der `with`-Block den Text-Strom an seinem Ende selber schließt.
- Deshalb können wir direkt mit dem zweiten `with`-Block weitermachen.
- Hier wollen wir eine einzelne Zeile Text aus der Datei lesen.
- Wir spezifizieren denn passenden `open`-Aufruf und merken uns das Ergebnis in `stream_in`.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Wir können uns darauf verlassen, dass der `with`-Block den Text-Strom an seinem Ende selber schließt.
- Deshalb können wir direkt mit dem zweiten `with`-Block weitermachen.
- Hier wollen wir eine einzelne Zeile Text aus der Datei lesen.
- Wir spezifizieren denn passenden `open`-Aufruf und merken uns das Ergebnis in `stream_in`.
- Der Körper des `with`-Statements beinhaltet also das `stream_in.readline()`.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Wir können uns darauf verlassen, dass der `with`-Block den Text-Strom an seinem Ende selber schließt.
- Deshalb können wir direkt mit dem zweiten `with`-Block weitermachen.
- Hier wollen wir eine einzelne Zeile Text aus der Datei lesen.
- Wir spezifizieren denn passenden `open`-Aufruf und merken uns das Ergebnis in `stream_in`.
- Der Körper des `with`-Statements beinhaltet also das `stream_in.readline()`.
- An seinem Ende wird der Text-Strom automatisch geschlossen.

```
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Was sagen Linter dazu?

- Deshalb können wir direkt mit dem zweiten `with`-Block weitermachen.
- Hier wollen wir eine einzelne Zeile Text aus der Datei lesen.
- Wir spezifizieren denn passenden `open`-Aufruf und merken uns das Ergebnis in `stream_in`.
- Der Körper des `with`-Statements beinhaltet also das `stream_in.readline()`.
- An seinem Ende wird der Text-Strom automatisch geschlossen.
- Und dann löschen wir die Datei via `remove`.

```python
"""Use the `with` statement to write to and read from a file."""

from os import remove  # Needed to delete a file.

# We open the text file "example.txt" for writing.
with open("example.txt", mode="w", encoding="UTF-8") as stream_
    stream_out.write("Hello world!")  # Write one line of text.

# We now open the text file "example.txt" for reading.
with open("example.txt", encoding="UTF-8") as stream_in:
    print(stream_in.readline())  # Read one line of text.

remove("example.txt")  # Delete the file "example.txt".
```

↓ python3 file_with.py ↓

```
Hello world!
```

Zusammenfassung



Zusammenfassung



- Jetzt können wir Ausnahmen sowohl auslösen als auch behandeln.

Zusammenfassung



- Jetzt können wir Ausnahmen sowohl auslösen als auch behandeln.
- Das erlaubt uns, Kode zu schreiben der sicherstellt, dass seine Ein- und Ausgaben korrekt sind.

Zusammenfassung



- Jetzt können wir Ausnahmen sowohl auslösen als auch behandeln.
- Das erlaubt uns, Kode zu schreiben der sicherstellt, dass seine Ein- und Ausgaben korrekt sind.
- Gleichzeitig können wir voraussehbare Fehler elegant behandeln.

Zusammenfassung



- Jetzt können wir Ausnahmen sowohl auslösen als auch behandeln.
- Das erlaubt uns, Kode zu schreiben der sicherstellt, dass seine Ein- und Ausgaben korrekt sind.
- Gleichzeitig können wir voraussehbare Fehler elegant behandeln.
- Das Wichtige ist, das wir beides *explizit* und *klar* machen.

Zusammenfassung



- Jetzt können wir Ausnahmen sowohl auslösen als auch behandeln.
- Das erlaubt uns, Kode zu schreiben der sicherstellt, dass seine Ein- und Ausgaben korrekt sind.
- Gleichzeitig können wir voraussehbare Fehler elegant behandeln.
- Das Wichtige ist, das wir beides *explizit* und *klar* machen.
- Wir können klar ausdrücken, welche Ausnahmen unser Kode auslöst, z. B. in Docstrings.

Zusammenfassung



- Jetzt können wir Ausnahmen sowohl auslösen als auch behandeln.
- Das erlaubt uns, Kode zu schreiben der sicherstellt, dass seine Ein- und Ausgaben korrekt sind.
- Gleichzeitig können wir voraussehbare Fehler elegant behandeln.
- Das Wichtige ist, das wir beides *explizit* und *klar* machen.
- Wir können klar ausdrücken, welche Ausnahmen unser Kode auslöst, z. B. in Docstrings.
- Wir können explizit schreiben, welche Ausnahmen wir behandeln können.

Zusammenfassung



- Jetzt können wir Ausnahmen sowohl auslösen als auch behandeln.
- Das erlaubt uns, Kode zu schreiben der sicherstellt, dass seine Ein- und Ausgaben korrekt sind.
- Gleichzeitig können wir voraussehbare Fehler elegant behandeln.
- Das Wichtige ist, das wir beides *explizit* und *klar* machen.
- Wir können klar ausdrücken, welche Ausnahmen unser Kode auslöst, z. B. in Docstrings.
- Wir können explizit schreiben, welche Ausnahmen wir behandeln können.
- Und wir können unseren Kode robust machen, so dass Ressourcen richtig geschlossen und freigegeben werden, selbst dann, wenn unerwartete Fehler auftreten.

谢谢您们！

Thank you!

Vielen Dank!



References I

[1] Daniel J. Barrett. *Efficient Linux at the Command Line*. Sebastopol, CA, USA: O'Reilly Media, Inc., Feb. 2022. ISBN: 978-1-0981-1340-7 (siehe S. 312, 313).

[2] Ed Bott. *Windows 11 Inside Out*. Hoboken, NJ, USA: Microsoft Press, Pearson Education, Inc., Feb. 2023. ISBN: 978-0-13-769132-6 (siehe S. 312).

[3] Ron Brash und Ganesh Naik. *Bash Cookbook*. Birmingham, England, UK: Packt Publishing Ltd, Juli 2018. ISBN: 978-1-78862-936-2 (siehe S. 311).

[4] Florian Bruhin. *Python f-Strings*. Winterthur, Switzerland: Bruhin Software, 31. Mai 2023. URL: https://fstring.help (besucht am 2024-07-25) (siehe S. 311).

[5] 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. 313, 314).

[6] "`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. 263–267).

[7] Slobodan Dmitrović. *Modern C for Absolute Beginners: A Friendly Introduction to the C Programming Language*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0224-9 (siehe S. 311).

[8] "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. 311).

[9] Pablo Galindo und Brett Cannon. *No More Bare Excepts*. Python Enhancement Proposal (PEP) 760. Beaverton, OR, USA: Python Software Foundation (PSF), 2.–9. Okt. 2024. URL: https://peps.python.org/pep-0760 (besucht am 2025-09-10). Status: Withdrawn, due to backward compatibility breakage, see https://discuss.python.org/t/pep-760-no-more-bare-excepts visited on 2024-09-10. But it has many good references on why we should not just `except` any possible `ExceptionS`. (Siehe S. 51–65).

References II

[10] 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. 311).

[11] 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. 311).

[12] Michael Goodwin. *What is an API?* Armonk, NY, USA: International Business Machines Corporation (IBM), 9. Apr. 2024. URL: https://www.ibm.com/topics/api (besucht am 2024-12-12) (siehe S. 311).

[13] 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. 311).

[14] Michael Hausenblas. *Learning Modern Linux*. Sebastopol, CA, USA: O'Reilly Media, Inc., Apr. 2022. ISBN: 978-1-0981-0894-6 (siehe S. 312).

[15] 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. 314).

[16] 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. 312).

[17] *Information Technology – Universal Coded Character Set (UCS)*. International Standard ISO/IEC 10646:2020. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Dez. 2020 (siehe S. 212–233, 310, 314).

[18] Stephen Curtis Johnson. *Lint, a C Program Checker*. Computing Science Technical Report 78-1273. New York, NY, USA: Bell Telephone Laboratories, Incorporated, 25. Okt. 1978. URL: https://wolfram.schneider.org/bsd/7thEdManVol2/lint/lint.pdf (besucht am 2024-08-23) (siehe S. 312).

References III

[19] .“exit – Terminate a Process”. In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Hrsg. von Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) und San Francisco, CA, USA: The Open Group, 8. Aug. 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799/functions/exit.html (besucht am 2024-10-30) (siehe S. 311).

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

[21] .“stderr, stdin, stdout – Standard I/O Streams”. In: *POSIX.1-2024: The Open Group Base Specifications Issue 8, IEEE Std 1003.1™-2024 Edition*. Hrsg. von Andrew Josey. Piscataway, NJ, USA: Institute of Electrical and Electronics Engineers (IEEE) und San Francisco, CA, USA: The Open Group, 8. Aug. 2024. URL: https://pubs.opengroup.org/onlinepubs/9799919799/functions/stdin.html (besucht am 2024-10-30) (siehe S. 313).

[22] Donald Ervin Knuth. *Fundamental Algorithms*. 3. Aufl. Bd. 1 der Reihe The Art of Computer Programming. Reading, MA, USA: Addison-Wesley Professional, 1997. ISBN: 978-0-201-89683-1 (siehe S. 312).

[23] Łukasz Langa. *Literature Overview for Type Hints*. Python Enhancement Proposal (PEP) 482. Beaverton, OR, USA: Python Software Foundation (PSF), 8. Jan. 2015. URL: https://peps.python.org/pep-0482 (besucht am 2024-10-09) (siehe S. 313).

[24] 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. 312).

[25] Jukka Lehtosalo, Ivan Levkivskyi, Jared Hance, Ethan Smith, Guido van Rossum, Jelle „JelleZijlstra“ Zijlstra, Michael J. Sullivan, Shantanu Jain, Xuanda Yang, Jingchen Ye, Nikita Sobolev und Mypy Contributors. *Mypy – Static Typing for Python*. San Francisco, CA, USA: GitHub Inc, 2024. URL: https://github.com/python/mypy (besucht am 2024-08-17) (siehe S. 312).

[26] Mark Lutz. *Learning Python*. 6. Aufl. Sebastopol, CA, USA: O'Reilly Media, Inc., März 2025. ISBN: 978-1-0981-7130-8 (siehe S. 312).

[27] Charlie Marsh. “Ruff”. In: URL: https://pypi.org/project/ruff (besucht am 2025-08-29) (siehe S. 312).

References IV

[28] Charlie Marsh. *ruff: An Extremely Fast Python Linter and Code Formatter, Written in Rust*. New York, NY, USA: Astral Software Inc., 28. Aug. 2022. URL: https://docs.astral.sh/ruff (besucht am 2024-08-23) (siehe S. 312).

[29] 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. 311).

[30] 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. 311).

[31] Yasset Pérez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J. Eglen, Daniel S. Katz, Tom J. Pollard, Alexander Konovalov, Robert M. Flight, Kai Blin und Juan Antonio Vizcaíno. "Ten Simple Rules for Taking Advantage of Git and GitHub". *PLOS Computational Biology* 12(7), 14. Juli 2016. San Francisco, CA, USA: Public Library of Science (PLOS). ISSN: 1553-7358. doi:10.1371/JOURNAL.PCBI.1004947 (siehe S. 311).

[32] Amit Phalgun, Cory Kissinger, Margaret M. Burnett, Curtis R. Cook, Laura Beckwith und Joseph R. Ruthruff. "Garbage in, Garbage out? An Empirical Look at Oracle Mistakes by End-User Programmers". In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'2005)*. 21.–24. Sep. 2005, Dallas, TX, USA. Hrsg. von Martin Erwig und Andy Schürr. Los Alamitos, CA, USA: IEEE Computer Society, 2005, S. 45–52. ISSN: 1943-6092. ISBN: 978-0-7695-2443-6. doi:10.1109/VLHCC.2005.40 (siehe S. 311).

[33] *Programming Languages – C, Working Document of SC22/WG14*. International Standard ISO/IEC9899:2017 C17 Ballot N2176. Geneva, Switzerland: International Organization for Standardization (ISO) und International Electrotechnical Commission (IEC), Nov. 2017. URL: https://files.lhmouse.com/standards/ISO%20C%20N2176.pdf (besucht am 2024-06-29) (siehe S. 311).

[34] Pylint Contributors. *Pylint*. Toulouse, Occitanie, France: Logilab, 2003–2024. URL: https://pylint.readthedocs.io/en/stable (besucht am 2024-09-24) (siehe S. 312).

[35] Yeonhee Ryou, Sangwoo Joh, Joonmo Yang, Sujin Kim und Youil Kim. "Code Understanding Linter to Detect Variable Misuse". In: *37th IEEE/ACM International Conference on Automated Software Engineering (ASE'2022)*. 10.–14. Okt. 2022, Rochester, MI, USA. New York, NY, USA: Association for Computing Machinery (ACM), 2022, 133:1–133:5. ISBN: 978-1-4503-9475-8. doi:10.1145/3551349.3559497 (siehe S. 312).

References V

[36] 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. 312).

[37] Anna Skoulikari. *Learning Git*. Sebastopol, CA, USA: O'Reilly Media, Inc., Mai 2023. ISBN: 978-1-0981-3391-7 (siehe S. 311).

[38] 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. 311).

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

[40] *The Unicode Standard, Version 15.1: Archived Code Charts*. South San Francisco, CA, USA: The Unicode Consortium, 25. Aug. 2023. URL: https://www.unicode.org/Public/15.1.0/charts/CodeCharts.pdf (besucht am 2024-07-26) (siehe S. 314).

[41] "The `with` Statement". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 8.5. URL: https://docs.python.org/3/reference/compound_stmts.html#with (besucht am 2024-12-15) (siehe S. 263–267).

[42] 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. 312).

[43] Mariot Tsitoara. *Beginning Git and GitHub: Version Control, Project Management and Teamwork for the New Developer*. New York, NY, USA: Apress Media, LLC, März 2024. ISBN: 979-8-8688-0215-7 (siehe S. 311, 314).

[44] *Unicode® 15.1.0*. South San Francisco, CA, USA: The Unicode Consortium, 12. Sep. 2023. ISBN: 978-1-936213-33-7. URL: https://www.unicode.org/versions/Unicode15.1.0 (besucht am 2024-07-26) (siehe S. 314).

[45] Guido van Rossum und Alyssa Coghlan. *The „`with`" Statement*. Python Enhancement Proposal (PEP) 343. Beaverton, OR, USA: Python Software Foundation (PSF), 13. Mai 2005–30. Juli 2006. URL: https://peps.python.org/pep-0343 (besucht am 2024-12-15) (siehe S. 263–267).

[46] Guido van Rossum und Łukasz Langa. *Type Hints*. Python Enhancement Proposal (PEP) 484. Beaverton, OR, USA: Python Software Foundation (PSF), 29. Sep. 2014. URL: https://peps.python.org/pep-0484 (besucht am 2024-08-22) (siehe S. 313).

References VI

[47] 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. 51–65, 311).

[48] Sander van Vugt. *Linux Fundamentals*. 2. Aufl. Hoboken, NJ, USA: Pearson IT Certification, Juni 2022. ISBN: 978-0-13-792931-3 (siehe S. 312).

[49] Thomas Weise (汤卫思). *Programming with C*. Hefei, Anhui, China (中国安徽省合肥市): Hefei University (合肥大学), School of Artificial Intelligence and Big Data (人工智能与大数据学院), 2026. URL: https://thomasweise.github.io/programmingWithC (besucht am 2026-05-10) (siehe S. 311).

[50] 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. 312).

[51] "With Statement Context Managers". In: *Python 3 Documentation. The Python Language Reference*. Beaverton, OR, USA: Python Software Foundation (PSF), 2001–2025. Kap. 3.3.9. URL: https://docs.python.org/3/reference/datamodel.html#with-statement-context-managers (besucht am 2024-12-15) (siehe S. 263–267).

[52] François Yergeau. *UTF-8, A Transformation Format of ISO 10646*. Request for Comments (RFC) 3629. Wilmington, DE, USA: Internet Engineering Task Force (IETF), Nov. 2003. URL: https://www.ietf.org/rfc/rfc3629.txt (besucht am 2025-02-05). See Unicode and[17] (siehe S. 212–233, 314).

[53] Giorgio Zarrelli. *Mastering Bash*. Birmingham, England, UK: Packt Publishing Ltd, Juni 2017. ISBN: 978-1-78439-687-9 (siehe S. 311).

Glossary (in English) I



API An *Application Programming Interface* is a set of rules or protocols that enables one software application or component to use or communicate with another[12].

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[3,30,53]. Learn more at `https://www.gnu.org/software/bash`.

C is a programming language, which is very successful in system programming situations[7,33,49].

docstring Docstrings are special string constants in Python that contain documentation for modules or functions[11]. They must be delimited by `"""..."""`[11,47].

exit code When a process terminates, it can return a single integer value (the exit status code) to indicate success or failure[19]. Per convention, an exit code of 0 means success. Any non-zero exit code indicates an error. Under Python, you can terminate the current process at any time by calling `exit` and optionally passing in the exit code that should be returned. If `exit` is not explicitly called, then the interpreter will return an exit code of 0 once the process normally terminates.If the process was terminated by an uncaught `Exception`, a non-zero exit code, usually 1, is returned.

f-string let you include the results of expressions in strings[4,8,10,13,29,38]. They can contain expressions (in curly braces) like `f"a{6-1}b"` that are then transformed to text via (string) interpolation, which turns the string to `"a5b"`. F-strings are delimited by `f"..."`.

GIGO Garbage In–Garbage Out, see, e.g., [32]

Git is a distributed Version Control System (VCS) which allows multiple users to work on the same code while preserving the history of the code changes[37,43]. Learn more at `https://git-scm.com`.

GitHub is a website where software projects can be hosted and managed via the Git VCS[31,43]. Learn more at `https://github.com`.

Glossary (in English) II



I/O In computer science, the I/O stands for *input/output*, which refers to receiving and transmitting information. The term is commonly used as reading data from files (input) and writing data to files (output). Another common usage is receiving data over a network (input) or sending data over a network (output).

IT information technology

linter A linter is a tool for analyzing program code to identify bugs, problems, vulnerabilities, and inconsistent code styles[18,35]. Ruff is an example of a linter used in the Python world.

Linux is the leading open source operating system, i.e., a free alternative to Microsoft Windows[1,14,36,42,48]. 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.

Microsoft Windows is a commercial proprietary operating system[2]. It is widely spread, but we recommend using a Linux variant such as Ubuntu for software development and for our course. Learn more at https://www.microsoft.com/windows.

Mypy is a static type checking tool for Python[25] that makes use of type hints. Learn more at https://github.com/python/mypy and in [50].

Pylint is a linter for Python that checks for errors, enforces coding standards, and that can make suggestions for improvements[34]. Learn more at https://www.pylint.org.

Python The Python programming language[16,24,26,50], i.e., what you will learn about in our book[50]. Learn more at https://python.org.

Ruff is a linter and code formatting tool for Python[27,28]. Learn more at https://docs.astral.sh/ruff or in [50].

stack trace A stack trace gives information the way in which one function invoked another. The term comes from the fact that the data needed to implement function calls is stored in a stack data structure[22]. The data for the most recently invoked function is on top, the data of the function that called is right below, the data of the function that called that one comes next, and so on. Printing a stack trace can be very helpful when trying to find out where an `Exception` occurred.

Glossary (in English) III



stderr The *standard error stream* is one of the three pre-defined streams of a console process (together with the standard input stream (`stdin`) and the standard output stream (`stdout`))[21]. It is the text stream to which the process writes information about errors and exceptions. If an uncaught `Exception` is raised in Python and the program terminates, then this information is written to `stderr`. If you run a program in a terminal, then the text that a process writes to its `stderr` appears in the console.

stdin The *standard input stream* is one of the three pre-defined streams of a console process (together with the `stdout` and the `stderr`)[21]. It is the text stream from which the process reads its input text, if any. The Python instruction `input` reads from this stream. If you run a program in a terminal, then the text that you type into the terminal while the process is running appears in this stream.

stdout The *standard output stream* is one of the three pre-defined streams of a console process (together with the `stdin` and the `stderr`)[21]. It is the text stream to which the process writes its normal output. The `print` instruction of Python writes text to this stream. If you run a program in a terminal, then the text that a process writes to its `stdout` appears in the console.

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

terminal A terminal is a text-based window where you can enter commands and execute them[1,5]. Knowing what a terminal is and how to use it is very essential in any programming- or system administration-related task. If you want to open a terminal under Microsoft Windows, you can Druck auf ⊞+R, dann Schreiben von `cmd`, dann Druck auf ↵. Under Ubuntu Linux, Ctrl+Alt+T opens a terminal, which then runs a Bash shell inside.

type hint are annotations that help programmers and static code analysis tools such as Mypy to better understand what type a variable or function parameter is supposed to be[23,40]. Python is a dynamically typed programming language where you do not need to specify the type of, e.g., a variable. This creates problems for code analysis, both automated as well as manual: For example, it may not always be clear whether a variable or function parameter should be an integer or floating point number. The annotations allow us to explicitly state which type is expected. They are *ignored* during the program execution. They are a basically a piece of documentation.

Glossary (in English) IV



Ubuntu is a variant of the open source operating system Linux[5,15]. 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.

UCS Universal Coded Character Set, see Unicode

Unicode A standard for assigning characters to numbers[17,40,44]. The Unicode standard supports basically all characters from all languages that are currently in use, as well as many special symbols. It is the predominantly used way to represent characters in computers and is regularly updated and improved.

UTF-8 The *UCS Transformation Format 8* is one standard for encoding Unicode characters into a binary format that can be stored in files[17,52]. It is the world wide web's most commonly used character encoding, where each character is represented by one to four bytes. It is backwards compatible with ASCII.

VCS A *Version Control System* is a software which allows you to manage and preserve the historical development of your program code[43]. A distributed VCS allows multiple users to work on the same code and upload their changes to the server, which then preserves the change history. The most popular distributed VCS is Git.