2. Python-Exceptions

Inhalt

Ein Python-Programm wird beendet, sobald es auf einen Fehler stösst. In Python kann ein Fehler ein Syntaxfehler oder eine Ausnahme sein. In diesem Artikel erfahren Sie, was eine Ausnahme ist und wie sie sich von einem Syntaxfehler unterscheidet. Danach erfahren Sie, wie Sie Ausnahmen auslösen und Zusicherungen aufstellen. Dann werden Sie mit einer Demonstration des `try...except` Blocks abschliessen.

Ausnahmen versus Syntaxfehler

Syntaxfehler treten auf, wenn der Parser eine falsche Anweisung erkennt. Beachten Sie folgendes Beispiel:

```
>>> print( 0 / 0 ))
  File "<stdin>", line 1
    print( 0 / 0 ))
                  ^
SyntaxError: invalid syntax
```

Der Pfeil zeigt an, wo der Parser auf den **Syntaxfehler** gestossen ist. In diesem Beispiel war eine Klammer zu viel. Entfernen Sie diese und führen Sie Ihren Code erneut aus:

```
>>> print( 0 / 0 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

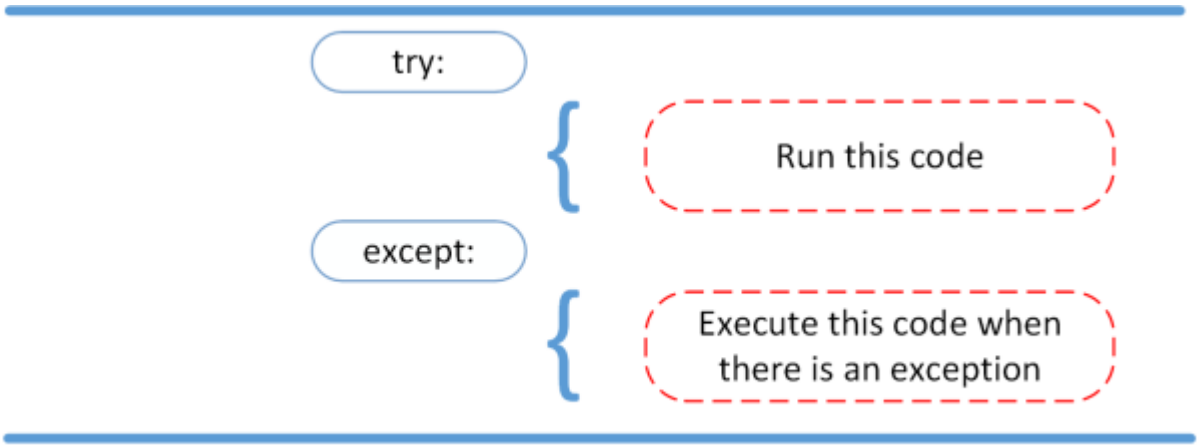
Dieses Mal ist ein **Ausnahmefehler** aufgetreten. Dieser Fehlertyp tritt immer dann auf, wenn

syntaktisch korrekter Python-Code zu einem Fehler führt. Die letzte Zeile der Meldung gibt an, auf welche Art von Ausnahmefehler Sie gestossen sind.

Anders als beim Syntaxfehler gibt Python an, welche Art von Ausnahmefehler aufgetreten ist. In diesem Fall war es ein `ZeroDivisionError`. Python verfügt über verschiedene eingebaute Ausnahmen sowie die Möglichkeit, selbst definierte Ausnahmen zu erstellen.

Der try...except Block: Umgang mit Ausnahmen

Der `try...except` Block in Python wird zum Abfangen und Behandeln von Ausnahmen verwendet. Python führt Code nach der `try` Anweisung als „normalen“ Teil des Programms aus. Der Code, der der `except` Anweisung folgt, ist die Antwort des Programms auf alle Ausnahmen in der vorhergehenden `try` Klausel.



Wie Sie bereits gesehen haben, gibt Python einen Ausnahmefehler aus, wenn syntaktisch korrekter Code auf einen Fehler stösst. Dieser Ausnahmefehler führt zum Absturz des Programms, wenn er nicht behandelt wird. Die `except` Klausel bestimmt, wie Ihr Programm auf Ausnahmen reagiert.

Die folgende Funktion kann Ihnen helfen, den `try...except` Block zu verstehen:

```
def process_orders(quantity: int):
    if quantity < 100:
        # hier wird ein Fehler erkannt und das passende Fehlerobjekt - im Beispiel ProcessingError - erzeugt
        raise ProcessingError("Not enough orders to start processing.")
    else:
        print('Processing orders.')
```

Die Funktion `process_orders()` kann nur erfolgreich mit Werten grösser oder gleich 100 ausgeführt werden. Andernfalls löst `raise` in dieser Funktion eine `ProcessingError` Exception aus.

Mit `try` können Sie dennoch einen Versuch wagen, die Funktion aufzurufen:

```
try:
    process_orders(99)
except:
```

```
    pass
```

Der Fehler wird hier mit `pass` behandelt. Wenn Sie diesen Code ausführen würden, erhalten Sie die folgende Ausgabe:

```

```

Nichts. Das Gute dabei ist, dass das Programm nicht abgestürzt ist. Aber es wäre schön zu sehen, ob bei der Ausführung Ihres Codes eine Ausnahme aufgetreten ist. Zu diesem Zweck können Sie `pass` wie folgt durch die Ausgabe einer Nachricht ersetzen:

```
try:
    process_orders(99)
except:
    print('Function was not executed')
```

Führen Sie diesen Code aus:

```
Function was not executed
```

Wenn in einem Programm, das diese Funktion ausführt, eine Ausnahme auftritt, wird das Programm fortfahren und Sie darüber informieren, dass der Funktionsaufruf nicht erfolgreich war.

Was Sie nicht gesehen haben, war die Art des Fehlers, der als Ergebnis des Funktionsaufrufs ausgelöst wurde. Um genau zu sehen, was schief gelaufen ist, müssten Sie den Fehler abfangen, den die Funktion ausgelöst hat.

Der folgende Code ist ein Beispiel, in dem Sie die Nachricht aus dem `ProcessingError` extrahieren und auf dem Bildschirm ausgeben:

```
try:
    process_orders(99)
except ProcessingError as error:
    print(error)
    print('Function was not executed')
```

Folgendes wird ausgegeben:

```
Not enough orders to start processing.
Function was not executed
```

Die erste Meldung ist der `ProcessingError`. Die zweite Meldung gibt an, dass die Funktion nicht ausgeführt wurde.

Im vorherigen Beispiel haben Sie eine selbst geschriebene Funktion aufgerufen. Als Sie die Funktion ausgeführt haben, haben Sie die `ProcessingError` Ausnahme abgefangen und auf dem Bildschirm ausgegeben.

Hier ist ein weiteres Beispiel, in dem Sie eine Datei öffnen und dabei auf Exceptions prüfen:

```
try:
```

```
    with open('file.log') as file:
        read_data = file.read()
except:
    print('Could not open file.log')
```

Wenn *file.log* nicht vorhanden ist, gibt dieser Codeblock folgendes aus:

```
Could not open file.log
```

Dies ist eine informative Nachricht, und unser Programm wird weiterhin ausgeführt. In der Python-Dokumentation können Sie sehen, dass Python viele sog. *Built-in Exceptions* anbietet, die Sie verwenden können. Eine auf dieser Seite beschriebene Ausnahme ist diese:

> Ausnahme `FileNotFoundError`
>
> Wird ausgelöst, wenn eine Datei oder ein Verzeichnis angefordert wird, aber nicht vorhanden ist. Entspricht `errno ENOENT`.

Um diese Art von Ausnahme abzufangen und auf dem Bildschirm auszugeben, können Sie den folgenden Code verwenden:

```
try:
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
```

Wenn in diesem Fall *file.log* nicht vorhanden ist, lautet die Ausgabe wie folgt:

```
[Errno 2] No such file or directory: 'file.log'
```

Sie können mehr als einen Funktionsaufruf in Ihrer `try` Klausel haben und sollten damit rechnen, verschiedene Ausnahmen abzufangen. Hier ist zu beachten, dass der Code in der `try` Klausel beendet wird, sobald eine Ausnahme auftritt.

Warnung: Catching `Exception` verbirgt alle Fehler ... auch die völlig unerwarteten. Aus diesem Grund sollten Sie in Ihren Python-Programmen allgemeine Überprüfungen auf `Exception` vermeiden und stattdessen spezifische Ausnahmeklassen verwenden, die Sie einzeln abfangen und behandeln. In diesem Tutorial erfahren Sie mehr darüber, warum dies eine gute Idee ist.

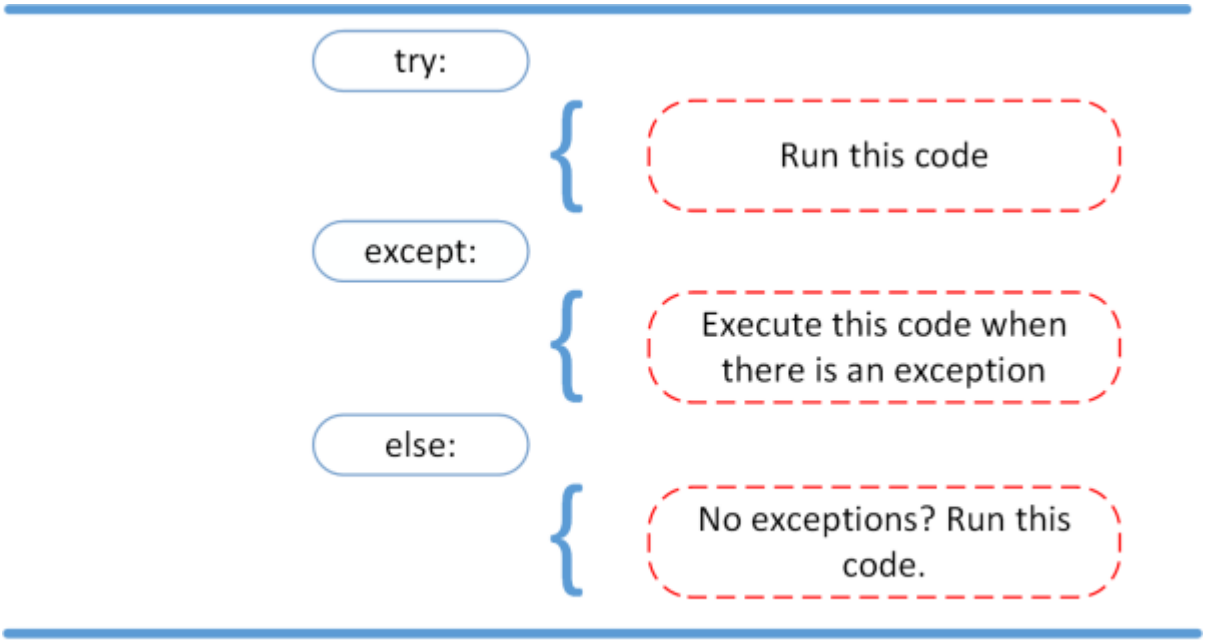
Hier sind die wichtigsten Erkenntnisse:

- Eine `try` Klausel wird bis zu dem Punkt ausgeführt, an dem die erste Ausnahme auftritt.
- Innerhalb der `except` Klausel bestimmen Sie, wie das Programm auf die Ausnahme reagiert.
- Dabei können Sie verschiedene Ausnahmen prüfen und jeweils anders darauf reagieren, so wie

es erforderlich ist.
- Verwenden Sie keine leeren `except` Blöcke und benennen Sie die erwarteten und behandelten Ausnahmen konkret!

Die else Klausel

In Python können Sie mit der `else`-Anweisung ein Programm anweisen, einen bestimmten Codeblock nur ohne Ausnahmen auszuführen.



Sehen Sie sich das folgende Beispiel an:

```
try:
    process_orders()
except ProcessingError as error:
    print(error)
else:
    print('Executing the else clause.')
```

Wenn Sie diesen Code ausführen und kein Fehler auftritt, würde die Ausgabe wie folgt aussehen:

```
Doing something. Executing the else clause.
```

Da das Programm auf keine Ausnahmen gestossen ist, wurde die `else` Klausel ausgeführt.

Sie können auch `try` Code innerhalb der `else` Klausel ausführen und dort auch mögliche Ausnahmen abfangen:

```
try:
    process_orders()
except ProcessingError as error:
    print(error)
else:
```

```
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
```

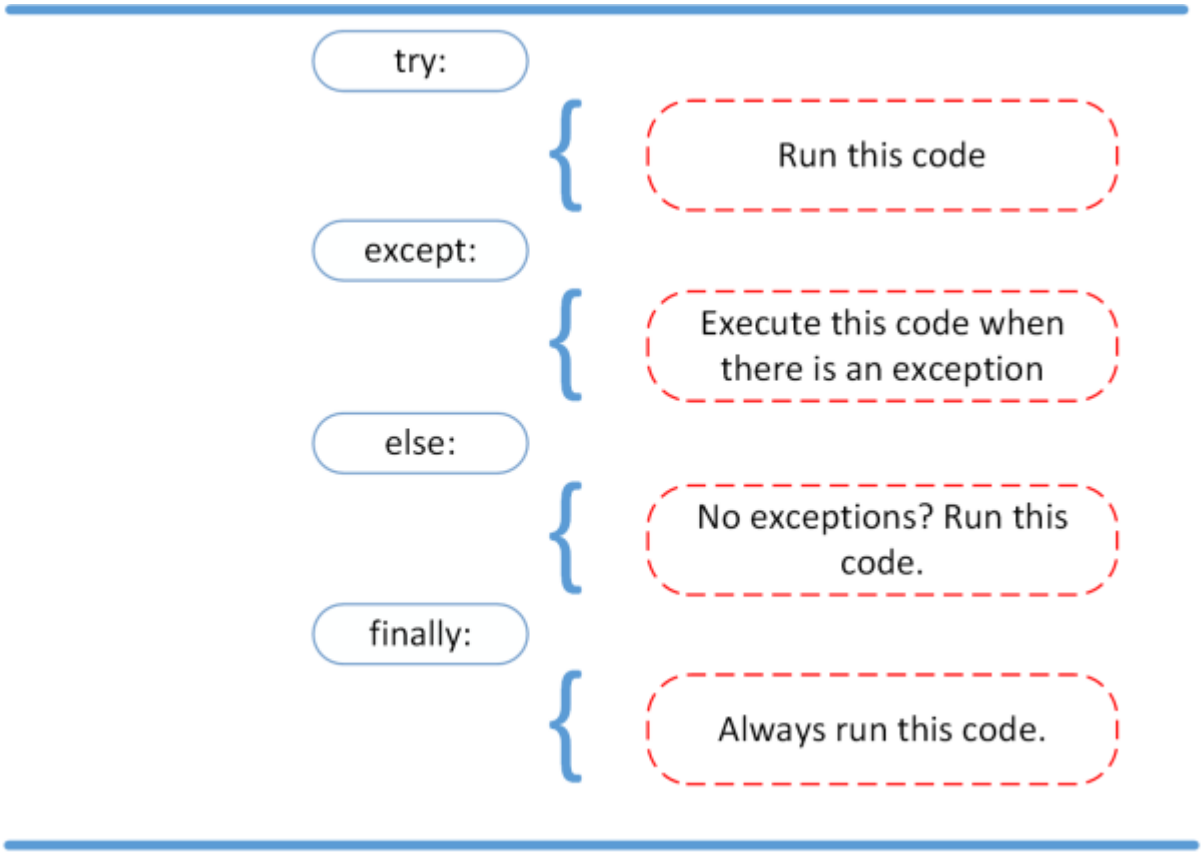
Wenn die Datei nicht gelesen werden kann, würden Sie das folgende Ergebnis erhalten:

```
Doing something.
[Errno 2] No such file or directory: 'file.log'
```

Aus der Ausgabe können Sie sehen, dass die `process_orders()` Funktion ausgeführt wurde. Da keine Ausnahmen aufgetreten sind, wurde versucht, *file.log* zu öffnen. Diese Datei war nicht vorhanden und, anstatt die Datei zu öffnen, haben Sie die `FileNotFoundError` Ausnahme abgefangen.

Aufräumen nach der finally Verwendung

Stellen Sie sich vor, Sie müssten nach der Ausführung Ihres Codes immer eine Aktion zum Aufräumen implementieren. Python ermöglicht Ihnen dies mit der `finally` Klausel.



Sehen Sie sich das folgende Beispiel an:

```
try:
    process_orders()
```

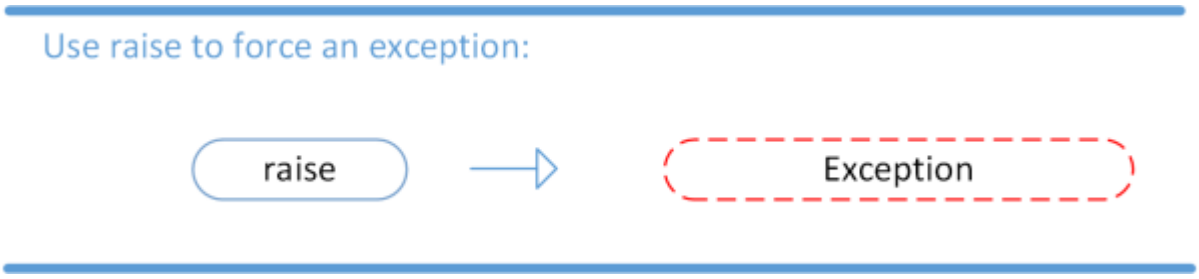
```
except ProcessingError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            read_data = file.read()
    except FileNotFoundError as fnf_error:
        print(fnf_error)
finally:
    print('Cleaning up, irrespective of any exceptions.')
```

Die `finally` Klausel wird immer ausgeführt. Dabei spielt es keine Rolle, ob Sie irgendwo in den `try` oder `else` Klauseln auf eine Ausnahme stossen. Das Unter der Annahme, dass beim Verarbeiten der Bestellungen ein Fehler auftritt, würde folgendes ausgegeben:

```
Processing Error occured.
Cleaning up, irrespective of any exceptions.
```

Auslösen einer Ausnahme

Falls eine bestimmte Bedingung eintritt, können Sie mit `raise` eine Ausnahme auslösen. Die Anweisung kann durch eine benutzerdefinierte Ausnahme ergänzt werden.



Um einen Fehler zu werfen, können Sie folgendes tun:

```
x = 10
if x > 5:
    raise Exception(f"x should not exceed 5. The value of x was: {x}")
```

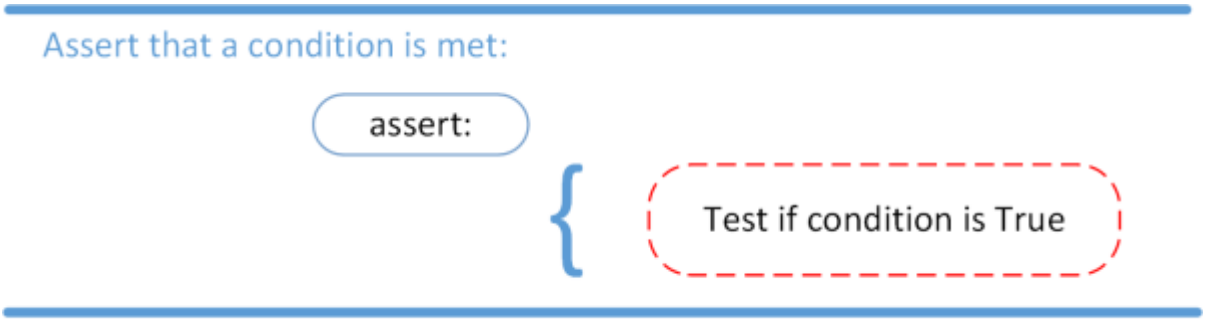
Wenn Sie den Code ausführen, erhalten Sie folgende Ausgabe:

```
Traceback (most recent call last):
  File "<input>", line 4, in <module>
Exception: x should not exceed 5. The value of x was: 10
```

Das Programm hält an und präsentiert unsere Ausnahme auf dem Bildschirm, zusammen mit Hinweisen darüber, was schief gelaufen ist.

Bedingungen sicherstellen

Mit `assert` stellen Sie sicher, dass eine bestimmte Bedingung erfüllt ist. Wenn sich herausstellt, dass dieser Zustand `True` ist, dann ist das ausgezeichnet! Das Programm kann fortgesetzt werden. Wenn sich herausstellt, dass die Bedingung `False` lautet, wirde eine `AssertionError` Ausnahme ausgelöst und das Programm beendet.



Schauen Sie sich das folgende Beispiel an:

```
x = -1

#if condition returns False, AssertionError is raised:
assert x >= 0, "The value x should be positive."
```

Wenn Sie diesen Code ausführen, ist das Ergebnis der Assertion `False` und angezeigt wird:

```
Traceback (most recent call last):
  File "<input>", line 4, in <module>
    assert x >= 0, "The value x should be positive."
AssertionError: The value x should be positive.
```

In diesem Beispiel ist das Auslösen einer `AssertionError` Ausnahme das Letzte, was das Programm tun wird. Das Programm wird beendet.

Achtung: `assert` ist in Python dazu gedacht, um *während der Entwicklungsphase* sicherzustellen, dass gewisse Erwartungen erfüllt werden (z.B. dass Funktionen vernünftige Werte zurückgeben). Dabei ist darauf zu achten, dass ein `assert` frühzeitig fehlschlägt (an der Stelle, an der der Fehler erkannt wurde). Daher sind `AssertErrors` nicht dazu gedacht, programmgesteuert erfasst oder behandelt zu werden mit `try…except`. Am besten verwenden Sie `assert` überhaupt *nicht in produktivem Code*, sondern ausschliesslich in Unit Tests

Zusammenfassung

Nachdem Sie den Unterschied zwischen Syntaxfehlern und Ausnahmen gesehen haben, haben Sie verschiedene Möglichkeiten zum Auslösen, Abfangen und Behandeln von Ausnahmen in Python kennengelernt und gesehen wie man die folgenden Schlüsselwörter verwendet:

- In der `try` Klausel werden alle Anweisungen ausgeführt, bis eine Ausnahme auftritt.
- `except` wird verwendet, um die in der try-Klausel auftretenden Ausnahmen abzufangen und zu behandeln.
- `else` lässt Sie Abschnitte codieren, die nur ausgeführt werden sollen, wenn in der try-Klausel keine Ausnahmen auftreten.
- `finally` ermöglicht es Ihnen, Codeabschnitte auszuführen, die immer ausgeführt werden sollten, mit oder ohne zuvor aufgetretene Ausnahmen.
- `raise` ermöglicht es Ihnen, jederzeit eine Ausnahme auszulösen.
- Mit `assert` können Sie überprüfen, ob eine bestimmte Bedingung erfüllt ist, und mit einer Ausnahme das Programm beenden, sollte dies nicht der Fall sein. Empfehlung: `assert` am besten überhaupt **nicht im produktiven Code verwenden!**

Hoffentlich hat Ihnen dieser Artikel geholfen, die grundlegenden Tools zu verstehen, die Python beim Umgang mit Ausnahmen zu bieten hat.

Credits: Der Inhalt dieses Artikels wurde grösstenteils direkt aus dem Englischen übernommen und nur leicht modifiziert. Das Original wurde hauptsächlich von Said van de Klundert verfasst und via Real Python veröffentlich.