Fehlerbehandlung

Wir haben bereits mehrfach gesehen, dass Python uns darauf hinweist, wenn unser Code einen Fehler bzw. eine *Exception* enthält. Diese Fehlermeldungen sorgen oft für Frustration, sind aber tatsächlich sehr hilfreich. Die Fehlermeldung enthält nämlich Informationen, die zum Beheben des Fehlers hilfreich sind:

- die Datei, in der der Fehler ist
- die Zeile, in der der Fehler ist
- bei Laufzeitfehlern: das Modul, in dem der Fehler ist
- bei Syntaxfehlern: die Position in der Zeile, wo der Fehler ist
- den Fehlertyp
- und eine etwas genauere Beschreibung des Fehlers

Deshalb ist es auch sinnvoll, Code regelmäßig zum Testen auszuführen, auch wenn das Programm noch nicht fertig ist. Gute Programmierer testen regelmäßig ihren Code, auch wenn es nur kleine Teile eines Programms sind. Gerade, wenn man nicht weiter kommt, kann es hilfreich sein, einfach mal auszuprobieren, was man schon hat. Dann kann man auch gut sehen, was noch gemacht werden muss, was noch verbessert werden kann, und was schon gut läuft.

In [5]:

```
# Beispiel 1: Runtime
l = [1, 2, 3]
for i in range(4):
    print(l[i])

print('Programmende')
```

```
1
2
3
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-5-c038f3f3a94f> in <module>
      2 l = [1, 2, 3]
      3 for i in range(4):
----> 4     print(l[i])
      5
      6 print('Programmende')

IndexError: list index out of range
```

In [2]:

```
# Beispiel 2: Syntax
for i in range(3)
    print('x')
```

```
  File "<ipython-input-2-1a9c5d11b123>", line 1
    for i in range(3)
                   ^
SyntaxError: invalid syntax
```

Bei Fehlern wird unterschieden zwischen Laufzeit- oder *Runtime*fehlern und Syntaxfehlern.

Bei **Syntaxfehlern** handelt es sich um Fehler, die beim Schreiben von Code auftreten. Die Syntax ist ja wie die Grammatik einer Programmiersprache, z.B. werden bei Python Schleifen immer mit einem Doppelpunkt geschrieben. Wenn diese Syntax misachtet wird, kann der Code nicht ausgeführt werden. Bevor der Python Interpreter also irgendetwas macht, gibt es direkt eine Fehlermeldung, wie oben in Beispiel 2. Diese Fehler sind unproblematisch, da sie einfach behoben werden können und dann nicht mehr auftreten. Viele IDEs weisen noch vor dem ersten Ausführen des Codes bereits auf eventuelle Syntaxfehler hin.

Laufzeitfehler sind da etwas kritischer. Diese Fehler treten beim Ausführen des Programmes auf und führen zu einem, in der Regel unerwünschten, Programmabbruch. In Beispiel 1 liegt ein `IndexError` vor, es wird versucht, auf einen Index der Liste zuzgreifen, der nicht existiert. Da Python nicht weiß, wie es danach weitergehen soll oder zu welchen Folgefehlern das führen könnte, wird das Programm abgebrochen. Die Ausgabe `'Programmende'` wird nicht mehr erreicht. In diesem Fall ist das nicht weiter schlimm, in komplexeren Programmen kann das aber zum Einen sehr ärgerlich sein, wenn man ständig neu starten muss, zum Anderen kann es auch zu Datenverlust führen. Dieses Problem hat bestimmt jeder schon mal erlebt, z.B. wenn Word abstürzt und alles, was man geschrieben hat, plötzlich weg ist; oder wenn ein Spiel abstürzt bevor man gespeichert hat.
Deshalb ist es die Verantwortung des Programmieres, solche Fehler zu verhindern. Dies kann manuell gemacht werden, oder automatisch. Eine manuelle Fehlerbehandlung ist oft die bessere Lösung. Im Beispiel 1 könnte der Fehler mit `len(l)` umgangen werden. Ein anderes Beispiel wäre ein Programm, das zwei Zahlen dividiert. Bei diesen Zahlen handelt es sich um Benutzereingaben, der Benutzer kann also auch eine `0` für den Nenner eingeben. Da dies nicht geht, muss dieser Fehler abgefangen werden, es muss also vor der Division geprüft werden, ob der Nenner `b` gleich `0` ist, und wenn ja, dann sollte der Benutzer einen entsprechenden Hinweis bekommen und eine neue Zahl eingeben.

Setze diese manuelle Fehlerbehandlung in Übung 1 um.

In [13]:

```
# Übung 1: manuelle Fehlerbehandlung
a = float(input('Zahl 1 (Zähler): '))
b = float(input('Zahl 2 (Nenner): '))
c = a / b
print(str(a) + ' geteilt durch ' + str(b) + ' ist gleich ' + str(c))
```

```
Zahl 1 (Zähler): 5
Zahl 2 (Nenner): 0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-13-0a63f4c5a9b4> in <module>
      2 a = float(input('Zahl 1 (Zähler): '))
      3 b = float(input('Zahl 2 (Nenner): '))
----> 4 c = a / b
      5 print(str(a) + ' geteilt durch ' + str(b) + ' ist gleich ' + str(c))

ZeroDivisionError: float division by zero
```

try und except

Manche Fehler lassen sich nicht manuell behandeln. Nehmen wir wieder die Division als Beispiel:

In [8]:

```
# Beispiel 3
a = float(input('Zahl 1 (Zähler): '))
b = float(input('Zahl 2 (Nenner): '))
c = a / b
print(str(a) + ' geteilt durch ' + str(b) + ' ist gleich ' + str(c))
```

```
Zahl 1 (Zähler): hallo
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-8-6746e81d5cf1> in <module>
      1 # Beispiel 3
----> 2 a = float(input('Zahl 1 (Zähler): '))
      3 b = float(input('Zahl 1 (Nenner): '))
      4 c = a / b
      5 print(str(a) + ' geteilt durch ' + str(b) + ' ist gleich ' + str(c))

ValueError: could not convert string to float: 'hallo'
```

Hier wurde statt einer Zahl ein Wort eingegeben. Der Input wird mit der `float()` Funktion in einen numerischen Wert umgewandelt, was mit einem Wort aber nicht möglich ist. Deshalb wird ein Fehler geworfen. Es muss also sichergestellt werden, dass nur Zahlen eingegeben werden. Dies ist manuell aber nicht (einfach) möglich. Deshalb gibt es das `try - except` Statement. Dies ist aufgeteilt in zwei Blöcke: im `try` Block wird versucht, Code auszuführen; der `except` Block wird ausgeführt, wenn es im `try` Block einen Fehler gibt. Wie bei Python üblich müssen wir auch hier wieder auf den Doppelpunkt und die Einrückung achten. Schauen wir uns das in einem Beispiel an:

In [16]:

```python
# Beispiel 4: try and except
def div():
    try:
        a = float(input('Zahl 1 (Zähler): '))
        b = float(input('Zahl 2 (Nenner): '))
        c = a / b
        print(str(a) + ' geteilt durch ' + str(b) + ' ist gleich ' + str(c))
    except ValueError:
        print('Ups. Das ist keine Zahl.')

div()
print('---------------------')
div()
```

```
Zahl 1 (Zähler): 5
Zahl 2 (Nenner): hallo
Ups. Das ist keine Zahl.
---------------------
Zahl 1 (Zähler): 5
Zahl 2 (Nenner): 7
5.0 geteilt durch 7.0 ist gleich 0.7142857142857143
```

In diesem Beispiel wird standardmäßig der `try` Block ausgeführt. Wenn es im `try` Block zu einem Fehler kommt, hier haben wir ein Wort statt einer Zahl eingegeben, bricht der `try` Block ab und es wird der entsprechende `except` Block ausgeführt. Wenn im `try` Block kein Fehler auftritt, dann wird auch kein `except` Block ausgeführt.

Ein `except` Block kann durch den Fehlertyp spezifiziert werden. In Beispiel 4 steht `except ValueError`, das heißt dieser `except` Block behandelt nur Fehler vom Typ ValueError. Wenn ein anderer Fehler auftritt, gibt es eine Fehlermeldung mit Programmabbruch. Dementsprechend kann ein `try - except` Statement mehrere `except` Blöcke haben, die von oben nach unten abgearbeitet werden. Ein `except` Block kann auch mehrere Fehlertypen behandeln, diese müssen dann als Tuple angegeben werden. Ein `except` Block mit dem allgemeinen `Exception` behandelt alle Fehler.

In [17]:

```python
# Beispiel 5: mehrere except Blöcke
def div():
    try:
        a = float(input('Zahl 1 (Zähler): '))
        b = float(input('Zahl 2 (Nenner): '))
        c = a / b
        print(str(a) + ' geteilt durch ' + str(b) + ' ist gleich ' + str(c))
    except ValueError:
        print('Ups. Das ist keine Zahl.')
    except ZeroDivisionError:
        print('Durch 0 kann nicht geteilt werden.')
    except Exception:
        print('Es ist ein unbekannter Fehler aufgetreten.')

div()
print('--------------------')
div()
```

```
Zahl 1 (Zähler): 5
Zahl 2 (Nenner): hallo
Ups. Das ist keine Zahl.
--------------------
Zahl 1 (Zähler): 5
Zahl 2 (Nenner): 0
Durch 0 kann nicht geteilt werden.
```

In [19]:

```python
# Beispiel 6: mehrere Fehlertypen in einem except Block
def div():
    try:
        a = float(input('Zahl 1 (Zähler): '))
        b = float(input('Zahl 2 (Nenner): '))
        c = a / b
        print(str(a) + ' geteilt durch ' + str(b) + ' ist gleich ' + str(c))
    except (ValueError, ZeroDivisionError):
        print('Es ist ein Fehler aufgetreten.')

div()
print('--------------------')
div()
```

```
Zahl 1 (Zähler): 5
Zahl 2 (Nenner): hallo
Es ist ein Fehler aufgetreten.
--------------------
Zahl 1 (Zähler): 5
Zahl 2 (Nenner): 0
Es ist ein Fehler aufgetreten.
```

In Beispiel 6 wurden zwei Fehlertypen mit einem `except` Block abgefangen. Das verkürzt zwar den Code, die Fehlerbehandlung wird dadurch aber unspezifischer. In Beispiel 5 hat der Benutzer eine Information darüber bekommen, was der Fehler ist, in Beispiel 6 fehlt diese Information.

Hier ist noch eine Übung.

Übung 2

Gegeben ist die Funktion `return_item()` , die ein Element aus einer Liste zurück gibt. Untersuche, welche Fehler auftreten können und fange diese Fehler ab.

In [ ]:

```python
# Übung 2
import random


def return_item(idx):
    k = random.randint(3, 10)  # length of random list
    my_list = []
    for i in range(k):  # fill random list with random numbers
        my_list.append(random.random())
    return my_list[idx]


idx = int(input())
print(return_item(idx))
```