Python: 12. Fouten en uitzonderingen

Dr. Cornelis Stal

April 27, 2022

1 Fouten en uitzonderingen afhandelen

1.1 Syntaxisfouten en uitzonderingen (exceptions)

Een Python-programma wordt beëindigd zodra er een fout optreedt. In Python kan een fout een syntaxisfout (`SyntaxError`) of een uitzondering (`exception`) zijn. In deze sectie worden beide zaken behandeld. Ook bekijken we hoe we fouten kunnen opvangen en op gepaste wijze kunnen afhandelen.

Syntaxisfouten treden op wanneer de parser een fouten in de code gevonden heeft:

[ ]:
```
print(0/0))
```

Als we in de Python-console code schrijven, geeft de pijl aan waar de parser de syntaxisfout is tegengekomen. We kunnen deze fout herstellen en de code nogmaals uitvoeren:

[ ]:
```
print(0/0)
```

In een IDE, zoals Spyder, lezen we voor dezelfde fout:

```
Traceback (most recent call last): File "D:\Desktop\untitled0.py", line 7, in
<module> print(0/0) ZeroDivisionError: division by zero
```

In dit geval zien we een `exception`, wat eveneens een fout is. Dit type fout treedt echter op wanneer syntactisch correcte Python-code resulteert in een fout. De laatste regel van het bericht vinden we het type uitzonderingsfout, zijnde een `ZeroDivisionError`, wat betekend dat we een getal niet kunnen delen door het cijfer nul. Python is in staat om verschillende standaard uitzonderingen te detecteren, maar we kunnen er ook zelf definiëren.

1.2 Een uitzondering opwerpen

We kunnen het `raise`-statement gebruiken een `exception` te genereren voor het geval een conditie optreedt. Dit statement kan worden aangevuld met een aangepaste `exception`. Als we een foutmelding willen genereren wanneer een bepaalde conditie optreedt, kun we dit met `raise` als volgt doen:

[ ]:
```
x = 10
if x > 5:
    raise Exception('x mag niet groter zijn dan 5, maar was {}'.format(x))
```

Wanneer we deze code uitvoeren, is de uitvoer de volgende:

```
Traceback (most recent call last):   File "<stdin>", line 2, in <module>
Exception: x mag niet groter zijn dan 5, maar was 10
```

Het programma stopt en toont de `exception` op het scherm, met aanwijzingen over wat er mis is gegaan.

1.2.1 De `AssertionError`-uitzondering

In plaats van te wachten tot een programma halverwege crasht, kunnen we in Python ook beginnen met een bewering. Wij beweren of `assert` dan dat aan een bepaalde voorwaarde is voldaan. Als deze toestand waar of `True` blijkt te zijn vervolgd de uitvoering. Als de voorwaarde echte onwaar of `False` blijkt te zijn kunnen we het programma een `AssertionError`-uitzondering op laten werpen. In het volgende voorbeeld wordt beweerd dat de code uitgevoerd wordt op een MS Windows system:

[ ]:
```
import sys
assert('win' in sys.platform), "Deze code is enkel voor MS Windows."
```

Als we deze code op een Windows-machine uitvoeren, gaat de bewering op en zal Python de code verder uitvoeren. Als we deze code op een Linux- (`linux`) of MacOS- (`darwin`) computer uit zouden voeren, krijgen we een foutmelding, aangezien de bewering een `False` terug geeft:

```
Traceback (most recent call last):   File "<stdin>", line 1, in <module>
AssertionError: Deze code is enkel voor MS Windows.
```

In dit voorbeeld zal het opwerpen van een `AssertionError`-uitzondering het laatste zijn wat het programma zal doen waarna het stopt. Vaak willen we echter dat het programma doorgaat met de uitvoering.

1.2.2 De `try`- en `except`-blokken: omgaan met uitzonderingen

Het `try`- en `except`-blok in Python wordt gebruikt om `exceptions` op te vangen en af te handelen. Python voert code uit na de `try`-instructie als een normaal onderdeel van het programma. De code die volgt op de `except`-instructie is de reactie van het programma op eventuele `exceptions` in de voorgaande `try`-clausule.

Zoals we eerder zagen zal Python een uitzonderingsfout genereren wanneer syntactisch correcte code een fout tegenkomt. Deze “exceptions” zullen het programma laten crashen als ze niet worden afgehandeld. De `except`-clausule bepaalt hoe het programma reageert op `exceptions`.

[ ]:
```
def windowsMethode():
    assert ('win' in sys.platform), "Deze code is enkel voor MS Windows."
    print("Voer iets uit...")
```

De `windowsMethode`-methode kan alleen op een Windows-systeem draaien. De `assert` in deze functie genereert een `AssertionError`-uitzondering als we de code op een ander besturingssysteem dan Windows op uit zouden voeren. We verwerken deze methode in een `try`-clausule:

[ ]:
```
try:
    windowsMethode()
except:
    pass
```

Als de code op een niet-Windows toestel uitgevoerd zal worden, resulteert de opgegeven `exception` in dit geval in een `pass`. Hierdoor krijgen we als output niets terug. Het programma is niet gecrasht, maar het zou handig zijn om te zien welk type fout er optrad bij de uitvoering van de code:

[ ]:
```
try:
    windowsMethode()
except:
    print("Windows-methode niet uitgevoerd.")
```

Voeren we deze code uit op een Linux- of MacOS computer, zien we de volgende boodschap:

```
Windows-methode niet uitgevoerd.
```

Wanneer er een uitzondering optreedt in een programma dat deze functie uitvoert, zal het programma doorgaan en ons informeren over het feit dat de functieaanroep niet succesvol was. Om precies te zien wat er mis is gegaan, moeten we de fout opvangen die de functie heeft veroorzaakt:

[ ]:
```
try:
    windowsMethode()
except AssertionError as error:
    print(error)
    print("Windows-methode niet uitgevoerd.")
```

Voeren we deze code uit op een Linux- of MacOS computer, zien we de volgende boodschap:

```
Deze code is enkel voor MS Windows. Windows-methode niet uitgevoerd.
```

Het eerste bericht is de `AssertionError`, die ons informeert dat de functie alleen op een Windows-machine kan worden uitgevoerd. Het tweede bericht vertelt ons welke functie niet is uitgevoerd. In dit voorbeeld is een functie aangeroepen we zelf geschreven hebben. Toen we de functie uitvoerden, is de `AssertionError`-uitzondering opgevangen en afgedrukt. We kunnen ook werken met *built-in* uitzonderingen:

[ ]:
```
try:
    with open('file.log') as file:
        leesData = file.read()
except:
    print('Kan file.log niet openen.')
```

Als `file.log` niet bestaat, geeft dit codeblok het volgende weer:

```
Kan file.log niet openen.
```

Dit is een informatief bericht en ons programma zal nog steeds worden uitgevoerd. In de Python-documenten valt te lezen dat er veel ingebouwde uitzonderingen bestaan. Enkele uitzondering die in deze sectie worden beschreven, zijn:

```
exception AssertionError Raised when an assert statement fails. (…) exception
FileNotFoundError Raised when a file or directory is requested but doesn't exist.
Corresponds to errno ENOENT.
```

We gebruiken de volgende code om een `FileNotFoundError`-uitzondering op te vangen en op het scherm af te drukken:

[ ]:
```
try:
    with open('file.log') as file:
        leesData = file.read()
except FileNotFoundError as fnfError:
    print(fnfError)
```

Als `file.log` niet bestaat, is de output de volgende:

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

We kunnen meer dan één functieaanroep in de `try`-clausule verwerken en anticiperen op het opvangen van verschillende `exceptions`. Merk hierbij wel op dat de code in de `try`-clausule stopt zodra een uitzondering wordt aangetroffen.

> **Opmerking:** Het opvangen van `exceptions` verwerkt alle andere fouten, zelfs fouten die volledig onverwacht zijn. Vermeid daarom lege `except`-clausules in de Python-code. In plaats daarvan verwijzen we naar specifieke uitzonderingsklassen die we willen opvangen en afhandelen.

We combineren bovenstaande nu in een code-reeks:

[ ]:
```
try:
    windowsMethode()
    with open('file.log') as file:
        leesData = file.read()
except FileNotFoundError as fnfError:
    print(fnfError)
except AssertionError as error:
    print(error)
    print("Windows-methode niet uitgevoerd.")
```

Zelfs als het bestand `file.log` niet bestaat, geeft het uitvoeren van deze code op een Linux- of MacOS-computer het volgende weer:

```
Deze code is enkel voor MS Windows. Windows-methode niet uitgevoerd.
```

Binnen de `try`-clausule veroorzaakt de `windowsMethode`-methode onmiddellijk fout, waardoor Python niet eens in de mogelijkheid is geweest om het bestand te laden. Bij afwezigheid van het bestand `file.log` lezen we op een Windows-computer:

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

1.3 De `else`-clausule

In Python kunnen we met behulp van de `else`-clausule een programma instrueren om een bepaald codeblok alleen uit te voeren als er geen uitzonderingen zijn:

[ ]:
```
try:
    windowsMethode()
```

```
except AssertionError as error:
    print(error)
else:
    print('De else clause wordt geprint.')
```

Op een Windows-systeem zou deze code de volgende output geven:

```
Voer iets uit. De else clause wordt geprint.
```

Omdat het programma geen enkele "exception" tegenkwam, werd de `else`-clausule uitgevoerd. Aan een `else`-clausule kunnen we ook nieuwe `try`- en `except`-blokken toevoegen:

[ ]:
```
try:
    windowsMethode()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            leedData = file.read()
    except FileNotFoundError as fnfError:
        print(fnfError)
```

Op een Windows-systeem zou deze code de volgende output geven:

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

Aan de output zien we dat de `windowsMethode`-methode succesvol is uitgevoerd. Omdat er geen uitzonderingen werden aangetroffen, werd vervolgens geprobeerd het bestand `file.log` te openen. Dat bestand bestond niet, en in plaats van het bestand te openen, is de `FileNotFoundError`-uitzondering opgevangen.

1.4 Afronden met de `finally`-clausule

Wanneer er binnen de code handelingen zijn die aan het einde hoe dan ook uitgevoerd moeten worden, zoals bijvoorbeeld het afsluiten van een geopend bestand, gebruiken we de "finally"-clausule:

[ ]:
```
try:
    windowsMethode()
except AssertionError as error:
    print(error)
else:
    try:
        with open('file.log') as file:
            leesData = file.read()
    except FileNotFoundError as fnfError:
        print(fnfError)
finally:
    print('Code afronden, ongeacht eventuele fouten.')
```

De `finally`-clausule wordt dus hoe dan ook uitgevoerd. Voeren we deze code uit op een Linux- of MacOS computer, zien we bijgevolg de volgende boodschap:

```
Deze code is enkel voor MS Windows. Code afronden, ongeacht eventuele fouten.
```