Python: Werken met bestanden en mappen

D. Leeuw

10 november 2025
0.0.0

1 Bestanden: Leerdoelen

Na het lezen van dit document weet de lezer:

1. wat bestanden zijn en hoe Python ze opent/sluit.
2. hoe je met Python tekstbestanden leest en schrijft.
3. hoe je werkt met bestands- en folderpaden.
4. hoe je werkt met os en pathlib modules.

2 Bestanden: File handles

Met Python kan je data van bestanden lezen en data naar bestanden schrijven dit onderdeel gaat over deze mogelijkheden. We gaan eenvoudige data lezen en schrijven van en naar een bestand.

Het werken met bestanden bestaat eruit dat we een bestand eerst moeten openen, daarna moeten we ervan lezen of naar schrijven, waarna we het bestand weer moeten sluiten. Om niet elke keer te hoeven aangeven om welk bestand het gaat gebruiken we een zogenaamde file handle. Je geeft in je script één keer aan welk bestand je wilt openen en koppelt daaraan een file handle en daarna gebruik je alleen nog deze file handle om te lezen of te schrijven. Tot slot moet je die file handle sluiten.

In Python ziet dat er ongeveer zo uit:

```
# File handle korten we af als fh
fh = open('testbestand.txt')
fh.close()
```

In dit voorbeeld is `fh` de file handle en die heeft functies zoals `write` (schrijven) naar en `close` (sluiten) van een bestand.

Bij het uitvoeren van deze code zal je een error melding krijgen:

```
FileNotFoundError: [Errno 2] No such file or directory: 'testbestand.txt'
```

Deze foutmelding is een runtime error en kan afgevangen worden met een try en except.

Python neemt standaard aan dat je een tekstbestand wilt openen om het te lezen (read). Je kan aan `open()` ook een 2de parameter meegeven, waarmee je zegt wat er gebeuren moet met een bestand:

parameter	functie	werking
r	Read	Opent een bestand om het te lezen, geeft een error als de file niet bestaat. Dit is de default.
w	Write	Opent een bestand om het te schrijven, maakt het bestand aan als het niet bestaat en overschrijft eventueel bestaande inhoud
a	Append	Opent een bestand om er data aan toe te voegen, maakt het bestand aan als het niet bestaat.
x	Create	Maakt een nieuw, leeg, bestand aan, geeft een error als het bestand bestaat.

Tabel 1: Logical operators

Naast deze functies kunnen we in dezelfde 2de parameter ook meegeven of het om een tekst (t) of een binair (b) bestand gaat. Binaire bestanden zijn bestanden zoals plaatjes of filmpjes. Tekst bestanden zijn bestanden zoals HTML-pagina's of configuratiebestanden. Met de parameter `rb` gaat Python ervanuit dat je een binair bestand wilt gaan lezen. En met `wt` dat je een tekstbestand wilt gaan schrijven.

Tenslotte is er nog een 3de parameter die we meekunnen geven en dat is de encoding, hiermee kunnen we aangeven wat voor character encoding er gebruikt is in een bestand:

```
# File handle korten we af als fh
fh = open('testbestand.txt', 'rt', encoding="utf-8")
fh.close()
```

2.1 Schrijven naar een bestand

We beginnen met het schrijven van een bestand zodat we iets hebben om mee te werken als we een bestand willen gaan lezen:

```
try:
    fh = open("testbestand.txt", "wt")
except fh.FileNotFoundError:
    print("Bestand niet gevonden")

fh.write("Dit is de eerste regel van ons nieuwe bestand.")
fh.write("Dan is dit dus de tweede regel van ons nieuwe bestand.")
fh.write("En tot slotte de derde en laatste van ons nieuwe bestand.")
fh.close()
```

Runnen we de code en bekijken we daarna de inhoud van ons nieuwe bestand dan zien we dat het 1 lange regel geworden is en helemaal geen 3 verschillende

regels.

Python voegt kennelijk letterlijk de regels toe aan ons bestand zonder rekening te houden met dat wij willen werken met regels. Op de één of andere manier zullen we Python dus moeten gaan vertellen dat er een regel einde is. Op Unix (Linux, Mac OS X) systemen is het voldoende om aan te geven dat er een new-line is, op Windows systemen geldt de regel dat er een return + newline moet zijn. Deze extra elementen kunnen we toevoegen in onze code voor een new-line gebruiken we \n en voor een return gebruiken we \r.

```
try:
    fh = open("testbestand.txt", "wt")
except fh.FileNotFoundError:
    print("Bestand niet gevonden")
    exit()

fh.write("Dit is de eerste regel van ons nieuwe bestand.\r\n")
fh.write("Dan is dit dus de tweede regel van ons nieuwe bestand.\r\n")
fh.write("En tot slotte de derde en laatste van ons nieuwe bestand.\r\n")
fh.close()
```

We hebben in de voorgaande code gezien dat na elke open er ook een close **moet** zijn. Het risico bestaat dat als er ergens een fout optreedt en het script crashed dat het dan nooit bij de `close` gekomen is. Hierbij kan er data verloren gaan. Om dit te voorkomen is er een speciale constructie en dat is de `with`:

```
with open('testbestand2.txt', 'wt', encoding="utf-8") as fh:
    fh.write("We springen in bij het gebruik van with.\r\n")
    fh.write("Zodra het inspringen over is zal het bestand gesloten
    worden.\r\n")
# Rest van de code
```

Het advies is dan ook om altijd de constructie `with open(...) as fh:` te gebruiken.

2.2 Lezen van een bestand

Nu we een bestand geschreven hebben, gaan we kijken hoe we dit bestand weer kunnen inlezen.

```
file='testbestand.txt'

try:
        with open(file, 'rt', encoding="utf-8") as fh:
                read_data = fh.read()
except FileNotFoundError:
```

```
        print(f"Bestand {file} niet gevonden")
        exit()
except PermissionError:
        print(f"Geen rechten on {file} te lezen")
        exit()

print(type(read_data))
print(read_data, end='')
```

De variabele `data_read` bevat het volledige document. Dit kan een probleem veroorzaken als het bestand groter is dan de maximaal beschikbare vrije geheugenruimte! We gaan later zien hoe we dit kunnen voorkomen.

Het boven getoonde script toont een paar nieuwe zaken die we nog niet kennen van Python, dus we lopen het script eerst even door. Het script met het maken van een variabele die de bestandsnaam bevat. Door dit in een variabele te doen kunnen we het script later makkelijker wijzigen als we een ander bestand willen lezen.

Via "try/except" openen we een bestand en met `read` lezen we alle data uit het bestand in het geheugen. Als dit allemaal goed verlopen is dan verlaten we de "try/except" constructie. Omdat `with` hebben gebruikt hoeven het bestand niet te sluiten, daar zorgt Python voor. Omdat Python het bestand voor ons sluit bestaat de file handle niet meer als we bij de `except` aankomen. De `except` checked dan ook direct op de error en niet op de fh.error.

Tot slot printen we eerst wat `read_data` eigenlijk is voor variabele en het blijkt een string te zijn. Daarna printen we de variabele met `print`. We gebruiken daarbij een speciale constructie. We vertellen print namelijk dat deze moet stoppen als er een lege regel voorbij komt. Dit is een echte lege regel, dus hij mag ook geen new line of return bevatten. Het is de zogenaamde NULL regel. Verwijder maar eens de `end=''` constructie en zie dat er dan een extra lege regel geprint wordt bij het printen van het bestand.

2.3 Lezen van regels uit een bestand

Als eerste is er `readline` om een bestand regel voor regel te lezen. Het leest één regel van een bestand. Een tweede readline opdracht leest de volgende regel:

```
file='testbestand.txt'

with open(file, 'rt', encoding="utf-8") as fh:
        print(fh.readline())
        print(fh.readline())
        print(fh.readline())
        print(fh.readline())
```

```
        print(fh.readline())
```

Geven we meer `readline` opdrachten dan er regels in een bestand zitten dan zal python geen error geven, maar extra lege regels geven.

We kunnen ook met een `for` loop data uit een bestand lezen:

```
file='testbestand.txt'

with open(file, 'rt', encoding="utf-8") as fh:
        for line in fh:
                print(f"{line}\n", end='')
```

Als we met de data willen werken is het misschien handiger als elke regel uit een bestand als een item in een **list** terecht komt. Zo kunnen we per regel met het bestand werken:

```
file='testbestand.txt'

with open(file, 'rt', encoding="utf-8") as fh:
        file = fh.readlines()

for line in file:
        print(line)
```

2.4 Bestanden verwijderen

Om een bestand weg te gooien (delete) hebben we de `os`-module nodig waaruit we de `remove` functie gebruiken:

```
import os
os.remove("demofile.txt")
```

3 Bestanden: Opdrachten

3.1 Maken van een notitieblok

Maak een programma dat een bestand (notes.txt) kan openen, en een nieuwe regel toevoegt met datum + tekst.

1. Open een bestand notes.txt (maak het aan als het niet bestaat).
2. Vraag de gebruiker om een notitie.
3. Voeg de notitie + datum/tijd toe aan het bestand. Met als format eerst de datum dan een komma, dan de tijd en een komma en dan de notitie.

4. Bonus: toon na afloop alle notities in de console.

4 Bestanden en directories

Om te zien of een pad of en bestand bestaat op een systeem kunnen we gebruik maken van de `os`-module of van de `pathlib`-module. We zullen van beide een voorbeeld geven. Het heeft de voorkeur om de moderne versie met `pathlib` te gebruiken.

De versie met de `os`-module:

```
import os

pad = "C:/Users/Public/Documents/bestand.txt"

if os.path.exists(pad):
    print("Pad bestaat")
    if os.path.isfile(pad):
        print("Het is een bestand")
    elif os.path.isdir(pad):
        print("Het is een map")
else:
    print("Pad bestaat niet")
```

De versie met de pathlib-module:

```
import pathlib

pad = Path("C:/Users/Public/Documents/bestand.txt")

if pathlib.path.exists():
    print("Pad bestaat")
    if pathlib.path.is_file():
        print("Het is een bestand")
    elif pathlib.path.is_dir():
        print("Het is een map")
else:
    print("Pad bestaat niet")
```

We zullen in de rest van dit document alleen nog `pathlib` gebruiken.

4.1 Analyseer de inhoud van een map

Laat het programma een map met bestanden inlezen en een overzicht maken in report.txt.

1. Vraag aan de gebruiker om een map

2. Test of de map bestaat
3. Haal informatie op van alle bestanden in de map
4. Schrijf in report.txt: naam van de map,bestandsnaam,grootte,datum
5. Bonus: Maak een notitie in notes.txt met de naam van de map die geanalyseerd is, met natuurlijk de datum en tijd.

5 Werken met CSV bestanden

We hebben tot nu toe elke keer een regel weggeschreven in rapport.txt of notities.txt die bestond uit de verzamelde data gescheiden door een komma. Een bestand met "Comma Separated Values" heet een CSV-bestand. Voor CSV bestanden gelden meer regels dan alleen dat de waarden gescheiden moeten zijn door een komma. Om te voldoen aan al die regels is er een Python module met de naam `csv` die ervoor zorgt dat de weggeschreven data echt een CSV bestand vormt.

Voor nadere uitleg over deze module zie het `python_csv.pdf` bestand.

1. Installeer de CSV-module
2. Maak een kopie van je notities applicatie
3. Herschrijf de kopie van de applicatie aan zodat de output een echt CSV bestand is en noem dit output bestand notities.csv.
4. Maak een kopie van je rapportage applicatie
5. Herschrijf de kopie van de applicatie zodat de output een echt CSV bestand is en noem dit output bestand rapport.csv.