# Programmeren in python

*090H99-2119*

# PROGRAMMEREN IN PYTHON

*090H99-2119*

[B5**]

# INHOUD

# 1 SOFTWAREBIBLIOTHEKEN

**In hoofdstuk 7 'Writing Functions' van het boek van Halterman hebt u al gezien hoe we code kunnen hergebruiken door middel van functies. Functies maken *modulair programmeren* in Python mogelijk. Bij modulair programmeren splitsen we een complexe taak of probleem in kleinere eenheden (bouwstenen), die gemakkelijker te testen en te beheren zijn. Deze bouwstenen kunnen we vervolgens samenvoegen tot een groter geheel: de Python-applicatie.**

**Naast functies kent de Python twee andere mechanismen voor modulair programmeren: *modules* en *packages*. Samen vallen deze onder de noemer *softwarebibliotheken*. In dit hoofdstuk bespreken we hoe u softwarebibliotheken binnen uw Python-applicatie kunt inzetten of zelf kunt ontwikkelen. Twee concepten die daarbij een belangrijke rol spelen, zijn *namespaces* en de *scope* van identifiers; ook deze zullen in dit hoofdstuk uitgebreid aan bod komen.**

## Wat is een module in Python?

Een module is een softwarebibliotheek die definities van functies, klassen en/of variabelen bevat. Een module kan eenvoudig aangemaakt worden door de Python-code voor deze definities op te slaan in een bestand met de extensie *.py*. Door de module met een *import*opdracht vanuit dit bestand in het geheugen te laden, kan een Python-applicatie de functies, klassen en variabelen uit de module gebruiken. Zoals alles is ook een module binnen Python een object.

Binnen Python is er een duidelijke één-op-één-relatie tussen een moduleobject, een modulebestand en een namespace: alle hebben dezelfde naam. In de volgende paragraaf leest u wat een namespace precies is.

We zetten de voordelen van het gebruik van modules voor u even op een rij:

a. *Modules bevorderen de eenvoud van code*: een module richt zich typisch op een gedeelte van een probleem of applicatie, waardoor een groot probleem in kleinere problemen opgedeeld wordt.
b. *Modules zijn gemakkelijk te onderhouden*: door het gebruik van modules wordt er een logische scheiding aangebracht tussen verschillende probleemgebieden binnen een applicatie. Wijzigingen in een module hebben daardoor minimale impact op andere onderdelen van de applicatie.
c. *Modules bevorderen hergebruik van code*: de functionaliteit zoals die binnen een module gedefinieerd is, kan eenvoudig in meer applicaties hergebruikt worden.

d. *Modules hebben een eigen scope*: doordat elke module een zelfstandige namespace heeft, wordt een botsing van identifiers op verschillende plaatsen in een applicatie voorkomen.

## Namespaces

Eerder hebt u al geleerd dat alles in Python een object is: getallen, functies en klassen, het zijn allemaal objecten van een ander type. Ook hebben we gezien dat we met een *identifier* naar een object kunnen verwijzen. Python houdt een boekhouding bij van de koppeling tussen identifiers en de bijbehorende objecten. Deze boekhouding wordt een *namespace* genoemd. Verschillende namespaces kunnen naast elkaar bestaan, maar zijn verder volledig van elkaar gescheiden. Gevolg hiervan is dat een zelfde identifier kan verwijzen naar verschillende objecten in verschillende namespaces.

In Python kunnen we drie verschillende categorieën namespaces onderscheiden:

a. De *local namespace*: deze namespace bevat alle identifiers die lokaal in een functie geldig zijn.
   Elke keer wanneer een functie opnieuw wordt aangeroepen, wordt er een nieuwe local namespace gecreëerd; deze namespace blijft in leven totdat de functie een waarde retourneert met de opdracht *return*.
b. De *global namespace*: deze namespace bevat alle identifiers die bij een module behoren.
   Wanneer een module in Python geladen wordt met de importopdracht, wordt ook een nieuwe namespace met dezelfde naam als de module gecreëerd. Het hoofdprogramma, dat geladen wordt bij het opstarten van Python of gestart wordt vanaf de command-prompt, heeft een eigen Global namespace *__main__*. Global namespaces behorende bij modules blijven bestaan totdat de Python-interpreter gestopt wordt.
c. De *built-in namespace*: deze namespace bevat de identifiers van alle built-in functies en built-in exceptions.
   Wanneer de Python-interpreter wordt opgestart, wordt deze namespace automatisch gecreëerd.

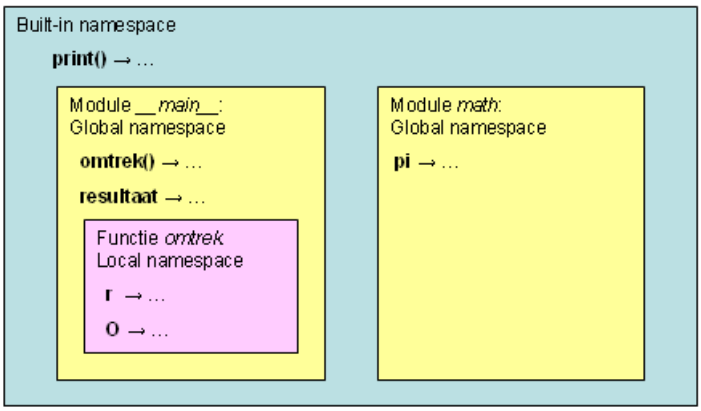
*Voorbeeld*
Bekijk de volgende Python-code:

```
import math

def omtrek (r):
    O = 2 * math.pi * r
    return (O)

resultaat = omtrek (5)
print (resultaat)
```

De functie *omtrek* berekent de omtrek van een cirkel. In afb. 1 is de verdeling van de identifiers in deze code over de verschillende namespaces weergegeven.




**Afb. 1** *Gebruik van namespaces.* *(Bron: OGN.)*

U ziet dat de variabele *pi* zich in een andere namespace bevindt dan de functie *omtrek*, namelijk in de namespace *math*. Daarom moeten we een *gekwalificeerde verwijzing* gebruiken vanuit de functie *omtrek* naar *pi*: we moeten *pi* voorzien met een prefix gelijk aan de modulenaam, gevolgd door een punt:

```
    O = 2 * math.pi * r
```

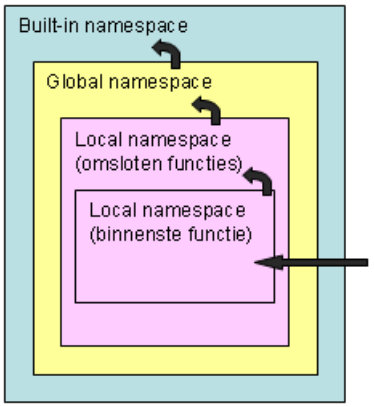
De variabelen *O* en *r* bevinden zich in de local namespace van de functie *omtrek*, dus zijn deze variabelen direct toegankelijk (d.w.z. er is geen prefix nodig bij het verwijzen naar deze variabelen).

De naam van de huidige module (en dus ook van de bijhorende global namespace) kan gevonden worden met de ingebouwde variabele *__name__*. In het script op het hoogste niveau, of in de Python-interpreter, wordt *__name__* gelijkgesteld aan *__main__*:

```
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 19:28:18) [MSC
v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print (__name__)
__main__
```

## De scope van een identifier

De *scope* van een identifier is dat gedeelte van een programma waarin het object dat bij de identifier hoort, *direct toegankelijk* is (dat wil zeggen, zonder prefix). Wanneer naar een variabele of object verwezen wordt zonder prefix, zal Python in volgorde een aantal namespaces doorzoeken om een definitie van het desbetreffende object te vinden. Deze volgorde is in afb. 2 weergegeven.




**Afb. 2** *De scope van een identifier.* *(Bron: OGN.)*

De volgorde waarin Python naar de definitie van een ongekwalificeerde identifier zoekt, is:

- eerst in de *local namespace van de binnenste functie* (d.w.z. de functie die als laatste is aangeroepen)
- daarna in de *local namespace van de omringende functies* (d.w.z. alle functies via welke de huidige functie is aangeroepen)
- vervolgens in de *global namespace*
- en ten slotte in de *built-in namespace*.

Als de identifier in geen van deze namespaces gevonden wordt, zal Python een foutmelding geven:

```
>>> import math
>>> print (pi)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    print (pi)
NameError: name 'pi' is not defined
```

*Voorbeeld*
Vanuit de Python-interpreter worden alleen de global namespace van de module *__main__* en de built-in namespace doorzocht en bevinden we ons dus buiten de scope van de variabele *pi*. Maar met een prefix kunnen we deze variabele wel benaderen:

```
>>> print (math.pi)
3.141592653589793
```

Met de built-in functie *dir()* (zonder argumenten) krijgen we een lijst van alle identifiers die in de huidige local namespace gedefinieerd zijn (in alfabetische volgorde):

```
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__',
'__spec__']
```

Wanneer de functie *dir()* aangeroepen wordt met een argument, worden alle identifiers in de desbetreffende module weergegeven:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum',
'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

### Het import-statement

In Python kunnen we verschillende vormen van het *import*-statement gebruiken, elk met een ander effect. De basisvorm van het import-statement is:

```
import <module_name>
```

Hiermee wordt alleen de module als object in de huidige global namespace (meestal *__main__*) geladen. Om de definities van bijvoorbeeld functies en klassen in de module te kunnen gebruiken, is een gekwalificeerde verwijzing (met een prefix gevolgd door een punt) nodig.

We kunnen echter ook elementen van een module in de huidige global namespace laden met de vorm:

```
from <module_name> import <name(s)>
```

*Voorbeeld*
Om de waarde van pi in de global namespace ***__main__***te laden:

```
>>> from math import pi
>>> print (pi)
3.141592653589793
```

Nu is geen gekwalificeerde verwijzing meer nodig om de variabele *pi* te kunnen benaderen.

In plaats van één voor één definities uit een module te laden, kunnen we ook met één instructie alle definities in de global namespace van de huidige module laden:

```
from <module_name> import *
```

Ten slotte kunnen we de objectdefinities uit de module ook laden met een alias:

```
from <module_name> import <name> as <alt_name>
```

Door een alias te gebruiken, voorkomen we een mogelijke botsing met bestaande identifiers in de global namespace.

*Voorbeeld*
Stel dat de variabele **e** in een programma al in gebruik is. Om het getal e (behorende bij exponentiële functies) te laden onder de naam *het_getal_e*:

```
>>> from math import e as het_getal_e
>>> print (het_getal_e)
2.718281828459045
```

### Zoeken naar modules

Wanneer een module geïmporteerd wordt met het *import*-statement, zal Python op de volgende plaatsen (en ook in deze volgorde) gaan zoeken naar het bestand met de extensie *.py*:

- de huidige folder
- de folders zoals gedefinieerd in de environment-variabele PYTHONPATH
- de standaardfolder (deze is installatieafhankelijk).

Met *path* uit de module *sys* kunt u zien welke folders inbegrepen zijn bij het zoekpad van Python:

```
>>> import sys
>>> sys.path
['', 'C:\\Program Files\\Python34\\Lib\\idlelib', 'C:\\WINDOWS\\sys-
tem32\\python34.zip', 'C:\\Program Files\\Python34\\DLLs', 'C:\\Pro-
gram Files\\Python34\\lib', 'C:\\Program Files\\Python34', 'C:\\Pro-
gram Files\\Python34\\lib\\site-packages']
```

Ook is het mogelijk om vanuit een Python-programma een folder aan het zoekpad toe te voegen.

*Voorbeeld*
Om de folder "C:\TEMP" toe te voegen aan het zoekpad:

```
>>> sys.path.append("C:\\TEMP")
>>> sys.path
['', 'C:\\Program Files\\Python34\\Lib\\idlelib', 'C:\\WINDOWS\\
system32\\python34.zip', 'C:\\Program Files\\Python34\\DLLs',
'C:\\Program Files\\Python34\\lib', 'C:\\Program Files\\Python34',
'C:\\Program Files\\Python34\\lib\\site-packages', 'C:\\TEMP']
```

## Packages gebruiken binnen Python

Met packages biedt Python de mogelijkheid om de code in uw softwareproject op een overzichtelijke en toegankelijke manier te organiseren. In deze paragraaf bekijken we wat packages zijn en hoe u de software verzameld in een package in uw softwareproject kunt inzetten.

### Wat is een package?

Op elke computer zijn, ongeacht het gebruikte besturingssysteem, bestanden op een hiërarchische manier gestructureerd in folders. De meeste gebruikers vinden het prettig om soortgelijke bestanden bij elkaar in één en dezelfde folder op te slaan (bijvoorbeeld alle MP3-bestanden in een folder 'muziek').

Op analoge wijze worden modules binnen Python gestructureerd in *packages*. Net zoals folders één of meer bestanden en subfolders kunnen bevatten, zo kunnen packages één of meer modules en subpackages hebben.

Om door Python herkend te worden als een package moet een folder een bestand met de naam *__init__.py* bevatten. Dit bestand bevat meestal de code voor de initialisatie van de package (bijvoorbeeld het automatisch laden van modules), maar kan ook leeg gelaten worden.

**Modules importeren uit een package**

Om een module uit een package te importeren, gebruiken we de naam van de package als prefix, gevolgd door de punt-operator (.) en de naam van de module:

```
import <package_name.module_name>
```

Om daarna een functie uit te module aan te roepen, hebben we een volledig gekwalificeerde naam nodig.

*Voorbeeld*

Numpy is een veelgebruikte package binnen de wiskunde met functionaliteit voor het oplossen van wiskundige problemen met numerieke methoden. De module *linalg* is onderdeel van deze package en bevat onder andere de functie *eigvals()*. Om deze functie aan te roepen, hebben we de volgende code:

```
>>> import numpy, numpy.linalg
>>> A = numpy.array([[2, 4], [5, -6]])
>>> numpy.linalg.eigvals(A)
array([ 4., -8.])
```

Om meer compacte code te krijgen, kunnen we ook een constructie in de vorm van *from ... import ... as ...* gebruiken:

```
>>> from numpy import array
>>> A = array([[2, 4], [5, -6]])
>>> from numpy import linalg as LA
>>> LA.eigvals(A)
array([ 4., -8.])
```

Numpy kan eenvoudig geïnstalleerd worden met behulp van de package-manager *pip*. Geef vanuit de installatiefolder van Python via de command-shell van uw besturingssysteem de opdracht 'Scripts\pip install numpy' en Numpy zal automatisch gedownload worden.

# 2 TESTEN EN DEBUGGEN

**Bij de professionele ontwikkeling van software speelt het programmeren zelf (onderdeel van de realisatie) maar een beperkte rol. Minstens zo belangrijk is het testen van programma's en systemen. Uit allerlei onderzoeken is gebleken dat hoe later een fout in de software ontdekt wordt, hoe groter de kosten voor aanpassing zijn. Het is dan ook niet voor niets dat het systematisch testen van software zich in de afgelopen decennia enorm ontwikkeld heeft. Testen is echter een vak apart; in deze module zullen we dan ook alleen wat algemene kaders van het testen schetsen. Voor u als cursist zijn vooral de praktische aspecten van het testen van belang, vandaar dat we ook voldoende aandacht zullen besteden aan het testen en debuggen in de praktijk.**

**In de eerste paragraaf van dit hoofdstuk analyseren we het proces van testen vanuit verschillende perspectieven. In de volgende paragraaf bespreken we hoe u systematisch testcases ontwerpt voor het functioneel testen van uw applicaties. Ook geven we enkele *do's* en *dont's* bij het testen. In de derde paragraaf illustreren we de theorie met een praktische casus. We sluiten dit hoofdstuk af met een bespreking van de debugger.**

## Wat is het testen van software?

Er zijn ontzettend veel definities van softwaretesten in omloop. In deze paragraaf bespreken we welke definities betekenisvol zijn en welke definities niet.

Een voor de hand liggende definitie van testen zou zijn: '*aantonen dat een computersysteem doet wat het moet doen*'. Een voorwaarde die onmiddellijk uit deze definitie volgt, is dat de eisen aan het systeem eenduidig en volledig gespecificeerd zijn. Helaas is dat in de praktijk niet altijd het geval; we komen hier later op terug.

Uit deze definitie volgt ook onmiddellijk een bezwaar: wat als een programma dingen doet die het niet moet doen (denk bijvoorbeeld aan het ongewenst verwijderen van bestanden op een harde schijf)? Duidelijk is dat de vorige definitie niet volstaat.

Een ander probleem met deze definitie is dat het praktisch onmogelijk is om aan te tonen dat een computerprogramma 'werkt'. Zelfs voor een klein algoritme vereist het heel veel logische en wiskundige toeren om te bewijzen dat het algoritme het gewenste resultaat oplevert. In de praktijk heeft men nooit voldoende tijd en middelen om aan te tonen dat een systeem naar behoren werkt.

Meer zinvol is een definitie van testen die de nadruk legt op het doel van het testen: het vergroten van de *betrouwbaarheid* van een computersysteem door het opsporen van

fouten. Hoe meer fouten er in het systeem gevonden en verwijderd worden, hoe groter de betrouwbaarheid van het systeem wordt.

Nu dienen we alleen nog vast te stellen wat een 'fout' precies inhoudt. We kunnen vaststellen dat een systeem foutief is als het werkelijke gedrag afwijkt van het gedrag dat in de *requirements* gespecificeerd is. De requirements (eisen) voor software kunnen onderverdeeld worden in verschillende categorieën:

- functionele eisen
- prestatie-eisen
- bruikbaarheidseisen
- zakelijke eisen
- veiligheidseisen
- wettelijke eisen.

Bij het testen *vergelijken* we dus het *werkelijke gedrag* van een systeem met de *requirements* voor dat systeem.

Met de vorenstaande analyse zouden we de volgende definitie van testen kunnen vinden:

> *Definitie*
> Testen is 'het opsporen van afwijkingen tussen het werkelijke gedrag van een systeem en de eisen aan dat systeem met als doel de betrouwbaarheid van het systeem te maximaliseren'.

We mogen gerust veronderstellen dat in de praktijk elk systeem fouten bevat. Daarom is het belangrijk te begrijpen dat hoe meer fouten er in het systeem opgespoord worden, hoe succesvoller de test is geweest. Een test die geen fouten blootlegt, heeft slechts een beperkte waarde: is het systeem echt foutvrij of was de test niet grondig genoeg?

## Het proces van testen

We kunnen op verschillende manieren naar het proces van testen kijken. Het proces van testen kunnen we vanuit drie verschillende perspectieven bekijken:

- het niveau van testen
- de methode van testen
- het type test dat uitgevoerd wordt.

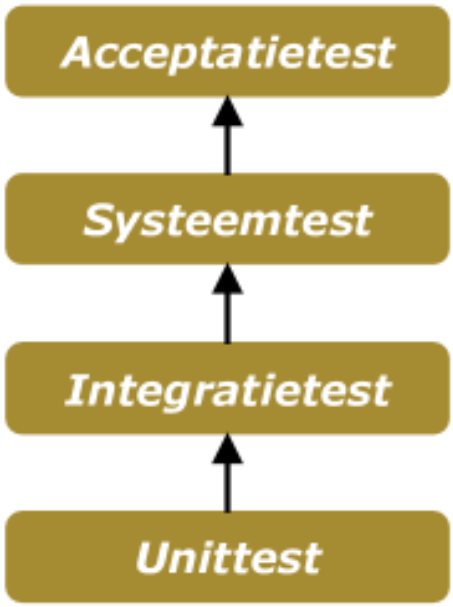
In deze paragraaf werken we elk van deze verschillende perspectieven nader uit.

## Niveaus van testen

Veel computersystemen zijn dermate complex dat het onmogelijk is alle functies en aspecten tegelijkertijd te testen. Daarom onderscheidt men vier verschillende niveaus van testen. Bij het onderscheiden van de verschillende niveaus speelt het concept *software-unit* (of kortweg *unit*) een essentiële rol.

> *Definitie*
> Een unit is de kleinst mogelijke eenheid programmatuur die een eigen functionaliteit heeft en apart getest kan worden.

Een unit heeft meestal maar een paar inputs en slechts één output. Voorbeelden van units zijn individuele functies en procedures (bij procedureel programmeren) of methoden van objectklassen (bij objectgeoriënteerd programmeren).




**Afb. 1** *Vier niveaus van testen.* *(Bron: OGN.)*

In het schema van afb. 1 zijn de vier verschillende niveaus van testen weergegeven. We lichten ze één voor één nog even toe:

De *unittest*: op dit niveau van testen worden individuele units en/of componenten van de software getest.

De *integratietest*: op dit niveau van testen worden individuele units samengevoegd en als groep getest. Het doel hiervan is om fouten in de interactie en/of interfaces van de verschillende units op te sporen.

De *systeemtest*: op dit niveau van testen wordt (nadat alle verschillende units geïntegreerd zijn) het complete systeem getest om te bepalen of het voldoet aan alle requirements.

De *acceptatietest*: op dit niveau van testen wordt het complete systeem door de eindgebruiker beoordeeld of het geaccepteerd en opgeleverd kan worden.

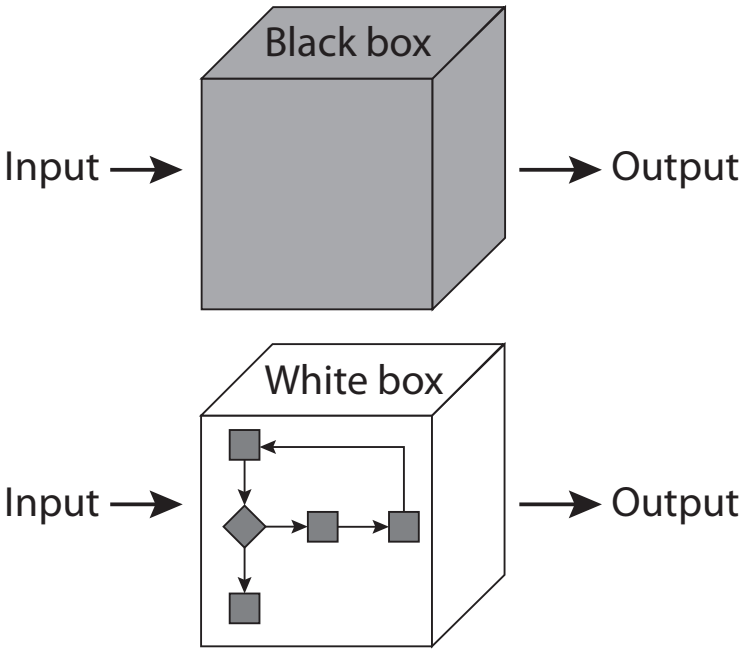
**Methoden van testen**

Op hoofdlijnen kan men twee verschillende methoden van testen onderscheiden: black-boxtesten en white-boxtesten.

*Black-boxtesten* is een methode van testen waarbij de interne werking van een computerprogramma voor de tester onzichtbaar is (vandaar ook de naam). Met deze methode van testen kunnen bijvoorbeeld fouten in de volgende categorieën opgespoord worden:

- foutieve of ontbrekende functies
- interface-fouten
- fouten in gegevensstructuren
- fouten bij de toegang tot externe databases
- fouten in de initialisatie of beëindiging.

*White-boxtesten* is een methode van testen waarbij de tester van een computerprogramma de interne structuur (zowel wat betreft ontwerp als implementatie) van het programma wél kent; de tester kan dan bepaalde invoer kiezen om het programma een bepaald pad door de code te laten lopen. De tester moet zowel bekend zijn met de implementatie van het programma als verstand van programmeren hebben.




**Afb. 2** *Het verschil tussen black-boxtesten en white-boxtesten.* *(Bron: OGN.)*

### Soorten tests

We kunnen vele verschillende soorten tests onderscheiden naar het doel van de test. Enkele vaak voorkomende soorten tests zijn de volgende:

*Functionele tests*: bij deze soort test wordt het gedrag van een computersysteem vergeleken met de functionele eisen aan het systeem.

*Prestatietest*: bij deze soort test wordt gekeken hoe het systeem presteert in termen van reactietijd en stabiliteit bij een bepaalde belasting.

*Bruikbaarheidstest*: bij deze soort test wordt vanuit het perspectief van de eindgebruiker beoordeeld of het systeem eenvoudig genoeg is in het gebruik.

*Veiligheidstest*: bij deze soort test wordt gekeken of het systeem voldoende beschermd is tegen eventuele aanvallers of indringers van buitenaf en wordt geprobeerd de kwetsbare plekken van het systeem bloot te leggen.

*Regressietest*: bij deze soort test wordt gekeken of wijzigingen in het systeem geen onverwachte bijwerkingen hebben op andere plaatsen in het systeem.

## Testen in de praktijk

Binnen de context van deze module hebben we vooral te maken met functioneel testen. In deze paragraaf bekijken we hoe functionele tests in de praktijk uitgevoerd worden. We bespreken aan welke eisen een goede testcase moet voldoen en welke zaken u voor elke testcase documenteert. Daarnaast geven we enkele praktische tips die voor u nuttig kunnen zijn wanneer u zelf aan de slag gaat met het functioneel testen van uw eigen programma's.

*Functioneel testen met testcases*

Functioneel testen kan zowel met black-boxtests als white-boxtests gebeuren. Een functionele test is meestal opgebouwd uit een aantal *testscenario's,* die elk bestaan uit een of meer *testcases*.

> *Definitie*
> Een testcase is een verzameling voorwaarden, variabelen en acties waarmee we kunnen bepalen of het geteste systeem voldoet aan de van tevoren gestelde eisen.

Belangrijk bij het functioneel testen is een goede voorbereiding: alle testcases worden van tevoren uitgedacht en gedocumenteerd.

Nadat alle testcases uitgedacht zijn, is het tijd om de testcases uit te voeren. Daarbij wordt voor elke testcase de werkelijke uitvoer vergeleken met de verwachte uitvoer. Dit proces wordt in het Engels *assertion* (bevestiging) genoemd. De uitvoer van een testcase moet steeds hetzelfde zijn; om toevallige resultaten uit te sluiten moet elke testcase dus ten minste tweemaal uitgevoerd worden.

### Wat is een goede testcase?

Het schrijven van goede testcases is geen sinecure, daarom volgen hier enkele eisen waaraan een goede testcase voldoet:

a. Testcases moeten identificeerbaar zijn, geef ze daarom een unieke testcase-id.
b. Testcases moeten eenvoudig en transparant zijn. Met een duidelijke, beknopte beschrijving kan de testcase door iedereen uitgevoerd worden (en niet alleen door de programmeur).
c. Elke testcase moet herhaalbaar zijn en iedere keer hetzelfde resultaat opleveren.
d. Met elke testcase wordt slechts één ding tegelijkertijd getest.
e. Met iedere testcase wordt iets nieuws getest, probeer dus herhaling van zetten te voorkomen.

#### *Vastlegging van testcases*

Het identificeren en uitdenken van goede testcases is een tijdrovende klus. Het is zonde al dat werk verloren te laten gaan, daarom worden de testcases voldoende gedocumenteerd. Bijkomend voordeel is dat alle testcases eenvoudig herhaald kunnen worden; bovendien maakt documentatie van de testcases regressietesten mogelijk.

Voor een serie testcases kunnen *precondities* van toepassing zijn, bijvoorbeeld bepaalde waarden in de database die vooraf aanwezig moeten zijn. Voor sommige testcases kunnen ook *postcondities* gelden: bepaalde voorwaarden die moeten gelden om de testcase af te kunnen sluiten (bijvoorbeeld het loggen van bepaalde gebeurtenissen).

Naast de precondities die gelden voor een serie testcases, leggen we van tevoren per testcase de volgende zaken vast:

a. Testcase-id.
b. Beknopte beschrijving van de testcase.
c. Uit te voeren teststappen.
d. Testgegevens die nodig zijn voor het uitvoeren van de testcase.
e. Verwachte resultaten.
f. Postcondities voor de testcase.

Hierna ziet u een lege tabel die gebruikt kan worden bij het ontwerpen van testcases.

| **Naam van de unit**<br>**Soort unit**<br>**Precondities** | | | | | |
|---|---|---|---|---|---|
| **Testcase-id** | **Omschrijving testcase** | **Uit te voeren teststappen** | **Testgegevens (invoer)** | **Verwachte resultaten** | **Postconditie** |
| 1 | | | | | |
| 2 | | | | | |
| 3 | | | | | |

**Afb. 3** *Vast te leggen zaken bij het ontwerpen van testcases.*

Bij het uitvoeren van de functionele test leggen we ook nog vast:

a. daadwerkelijke resultaten
b. testuitslag (test wel/niet geslaagd).

*Enkele praktische tips*
Hierna volgen enkele praktische tips die u bij het testen van uw eigen programma's kunnen helpen:

a. Begin uw test met de instelling om fouten op te sporen. Als u voor het testen al in de veronderstelling bent dat uw code foutloos zal zijn, zult u nooit de subtiele, zeldzame fouten kunnen vinden die er 10 tegen 1 toch inzitten.
b. Schrijf uw testcases voor zowel geldige invoer als ongeldige invoer. Probeer niet alleen de verwachte situaties te testen, maar vooral ook de onverwachte situaties. Wees alert voor allerlei dingen die het programma niet geacht wordt te doen.
c. Bekijk de uitkomsten van uw testcases zorgvuldig. Elke schrijver en programmeur is blind voor de eigen fouten, maar als u systematisch de testresultaten vergelijkt met de verwachte resultaten kunt u toch veel van uw eigen fouten corrigeren!
d. Een efficiënte en effectieve manier van white-boxtesten is het gebruik van print-statements in uw programma. Door op de juiste plaatsen print-statements tussen te voegen in uw code, kunt u op eenvoudige manier de waarden en het type van variabelen in het geheugen inspecteren. Vergeet echter niet alle code voor testdoeleinden te verwijderen voordat u uw code oplevert voor gebruik in productie! Testcode vormt een zeer vaak voorkomende categorie bugs in software.

## Automatisch testen

U zult merken dat, om een Python-applicatie van enige omvang grondig te testen, u al gauw enkele tientallen testcases nodig hebt. En in principe dienen deze testcases na elke wijziging allemaal opnieuw doorlopen te worden. Omdat handmatig te doen, is een tijdrovend karwei, maar gelukkig bestaan er tools die u dat werk uit handen kunnen nemen en met scripts testcases automatisch kunnen testen. U dient dan wel het testscript zelf te schrijven, maar daarna kan de testcase meermalen automatisch doorlopen worden.

Twee veelgebruikte tools voor het automatisch testen van Python-applicaties zijn *unittest* (deze maakt deel uit van de standaardbibliotheek van Python) en *pytest*. Het

valt buiten het bestek van deze module om deze tools in detail te bespreken; meer informatie kunt u vinden op de documentatiepagina's van unittest (https://docs.python.org/3/library/unittest.html) en pytest (https://docs.pytest.org/en/latest/).

### Voorbeeldcasus: testcases ontwerpen en uitvoeren

We bekijken nu hoe u voor een gegeven probleemstelling testcases ontwerpt.

> *Beschrijving van de casus*
> In de statistiek is de modus van een serie waarnemingen de waarde die het vaakst voorkomt.
>
> Wanneer er meer waarden zijn in de waarnemingen die allemaal even vaak als vaakst voorkomen, kunnen we stellen dat de modus niets bestaat (in zo'n geval moet een functie de waarde *None* teruggeven).
>
> Voorbeeld: voor een groep van 15 mannen worden de volgende schoenmaten gemeten: {43, 41, 42, 42, 41, 39, 44, 42, 43, 44, 41, 42, 40, 42, 41}. De modus van deze serie waarnemingen is 42, want deze schoenmaat komt het vaakst voor (namelijk 5 maal).

We willen nu een implementatie voor een *modus*-functie testen in Python. Daartoe ontwerpen we een functionele test met meer testcases (zie afb. 4).

| **Naam van de unit** | | **modus** | | | |
|---|---|---|---|---|---|
| **Soort unit** | | **functie** | | | |
| **Precondities** | | **geen** | | | |
| **Testcase-id** | **Omschrijving testcase** | **Uit te voeren teststappen** | **Testgegevens (invoer)** | **Verwachte resultaten** | **Postconditie** |
| A | Lege list | Functie aanroepen | [] | None | |
| B | Slechts één waarde | Functie aanroepen | [44] | 44 | |
| C | Tweemaal dezelfde waarde | Functie aanroepen | [44, 44] | 44 | |
| D | Een waarde komt vaker voor | Functie aanroepen | [44, 43, 43] | 43 | |
| E | Twee waarden komen even vaak voor | Functie aanroepen | [44, 43, 43, 44] | None | |
| F | Een grote verzameling getallen | Functie aanroepen | [43, 41, 42, 42, 41, 39, 44, 42, 43, 44, 41, 42, 40, 42, 41] | 42 | |
| G | Een grote verzameling woorden | Functie aanroepen | ["Hond", "Kat", "Hond", "Kanarie", "Kat", "Hond"] | "Hond" | |
| H | Geen list als argument | Functie aanroepen | 44 | Foutmelding | |

**Afb. 4** *Voorbeeld van testcases bij een functionele test.*

U ziet dat de testcases systematisch opgebouwd zijn: we beginnen met een lege list en breiden deze vervolgens met steeds meer waarden uit. Ook hebben we een testcase met tekst in plaats van getallen toegevoegd. De laatste testcase is een geval van ongeldige invoer; bij een volledige set testcases willen we niet alleen situaties met geldige invoer testen, maar vooral ook de uitzonderingssituaties.

## Het debuggen van een programma

Het woord *bug* komt uit het Engels; de definitie van een bug volgens de Webster's Collegiate Dictionary is: 'an unexpected defect, fault, flaw, or imperfection'. In programmeursjargon is debuggen het opsporen en verwijderen van fouten ('bugs') uit een computerprogramma. In deze paragraaf bekijken we welke soorten fouten men in de praktijk tegenkomt bij het programmeren in Python en hoe u een debugger kunt gebruiken bij het opsporen van deze fouten.

### Soorten bugs

Alle mogelijke fouten in een Python-programma kunnen geclassificeerd worden in drie categorieën:

*Syntax errors*: dit zijn fouten in de manier waarop de Python-instructies gecodeerd zijn (bijvoorbeeld een ontbrekende ":" na een *if*-statement. De Python-interpreter geeft een foutmelding wanneer hij een ongeldige instructie tegenkomt bij het vertalen van broncode naar bytecode.

*Runtime errors*: dit zijn fouten die tijdens het uitvoeren van een programma ontstaan (bijvoorbeeld het delen door 0 of het rekenen met strings). De Python-interpreter geeft, voordat het programma afgebroken wordt, een foutmelding met de aard en de plaats van de fout.

*Semantic errors*: dit zijn fouten waarbij het programma iets anders doet dan de programmeur bedoeld heeft. Dit zijn de lastigste fouten om op te sporen, omdat er door Python geen foutmelding gegenereerd wordt.

### Een debugger gebruiken

Een *debugger* is een hulpprogramma (meestal onderdeel van een *IDE = Integrated Development Environment*), waarmee het mogelijk is om een inkijk te krijgen in de status van een computerprogramma tijdens de uitvoering. U kunt de uitvoering stoppen op een willekeurige regelcode in het programma, instructies stap voor stap doorlopen en de waarden van variabelen en objecten in het geheugen inspecteren. Daarmee wordt het eenvoudiger om bugs in het programma te lokaliseren en de programmacode aan te passen.

Ook de door ons aanbevolen IDE, Wing 101, beschikt over een ingebouwde debugger. Hierna tonen wij u de functies van deze debugger; we doen dat aan de hand van voorbeeldcode die een bug bevat (zie de probleemstelling hierna).

*Probleemstelling*
Een beginnend programmeur heeft voor een dokterskliniek een klein Python-programma geschreven om de Body Mass Index (BMI) van enkele patiënten te berekenen. De BMI wordt berekend door het gewicht van de patiënt in kilogram te delen door het kwadraat van zijn/haar lengte in meters. De code van het programma is hierna gegeven.

```
patienten = [[1.81, 70], [1.90, 85], [1.75, 102]]

def bereken_bmi (g, l):
return g / (l ** 2)

for patient in patienten:
lengte, gewicht = patient
bmi = bereken_bmi (lengte, gewicht)
print("De BMI van de patient is: %f" % bmi)
```

Het programma levert de volgende uitvoer:

```
De BMI van de patient is: 0.000369
De BMI van de patient is: 0.000263
De BMI van de patient is: 0.000168
```

Het probleem van dit programma echter, is dat de berekende waarden van de BMI veel te klein zijn: normaal gesproken ligt de BMI tussen 18.5 en 30. Het is dus zaak om de bug in het programma op te sporen!

**Beginnen met debuggen**
Hoewel debuggers erg op elkaar lijken en meestal dezelfde functies hebben, werken ze allemaal net even anders. De volgende beschrijving is van toepassing op Wing 101.

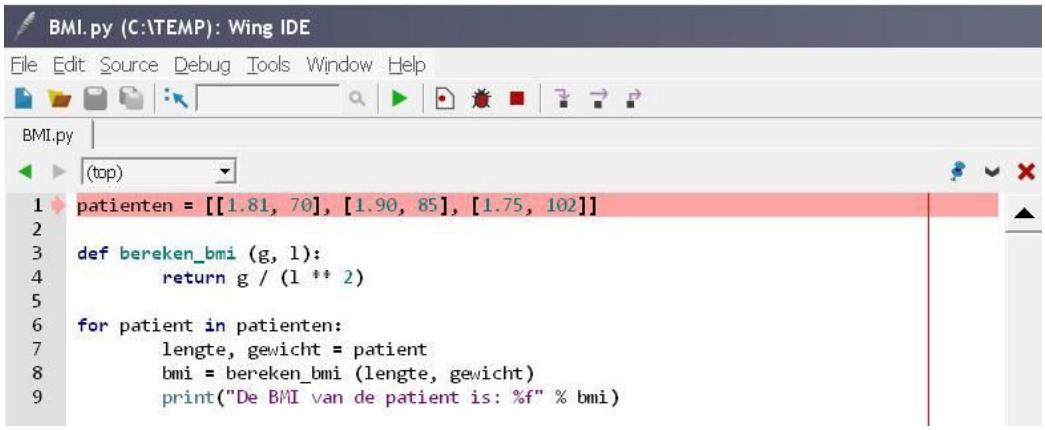
Als eerste laden we het programma dat we willen debuggen in de IDE. In ons geval is dat het programma *BMI.py*.

Met sneltoetsen kunnen we nu door de instructies van het programma stappen:

- met *F7* stapt de debugger naar de instructie waar de cursor op staat, zonder deze uit te voeren
- met *CTRL-F6* voert de debugger de huidige instructie uit en stapt door naar de eerstvolgende instructie
- met *F5* start de debugger met het uitvoeren van instructies, net zolang totdat een breakpoint of het einde van het programma bereikt wordt.

Druk nu op *F7* om naar de eerstvolgende instructie te stappen. Het rode pijltje in de linkerkantlijn geeft aan welke instructie de debugger eerstvolgend zal uitvoeren. In ons geval is dat de regelcode:

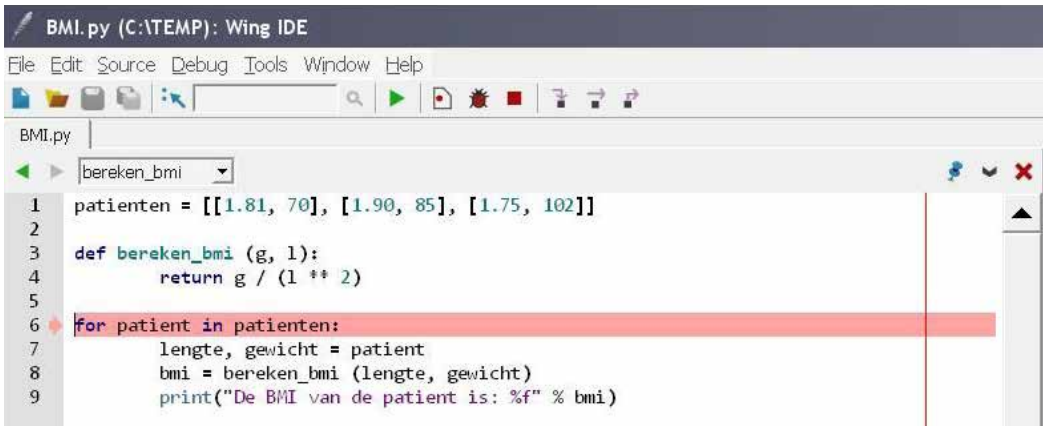
```
patiënten = [[1.81, 70], [1.90, 85], [1.75, 102]]
```




**Afb. 5** *Stappen naar de eerstvolgende instructie.* *(Bron: OGN.)*

Druk nu op *CTRL-F6* om deze instructie uit te voeren. Het pijltje springt naar de volgende instructie. Merk op dat de debugger eventuele regels met commentaar overslaat; deze bevatten geen instructies die uitgevoerd kunnen worden.

Na nogmaals op *CTRL-F6* gedrukt te hebben, komen we aan bij de *for*-loop.

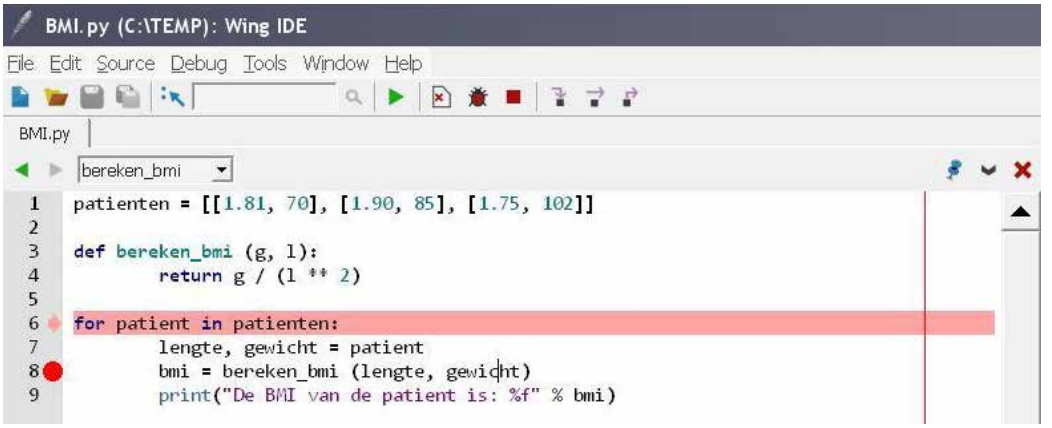



**Afb. 6** *De for-loop.* *(Bron OGN.)*

### Breakpoints

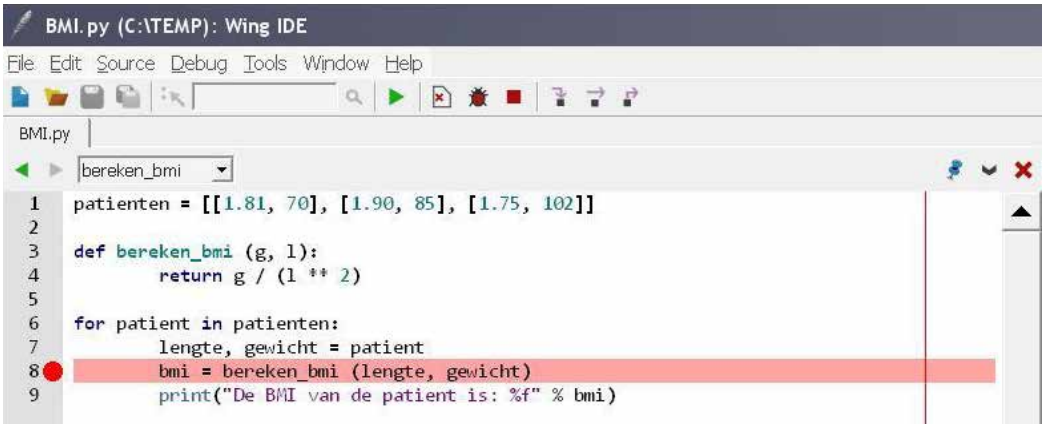
Het kan erg tijdrovend zijn om steeds door alle instructies van een iteratie te moeten stappen. Maar gelukkig is het mogelijk om een punt in te stellen waar de debugger moet stoppen, zodat we vanaf dat punt verder kunnen debuggen. Zo'n punt waar het programma moet stoppen, noemen we een *breakpoint*.

Zet de cursor op de regel met de aanroep van de functie *bereken_bmi* en druk op *F5*. U ziet nu een rode stip voor de regel verschijnen. Deze stip geeft aan dat u hier een breakpoint hebt geplaatst.



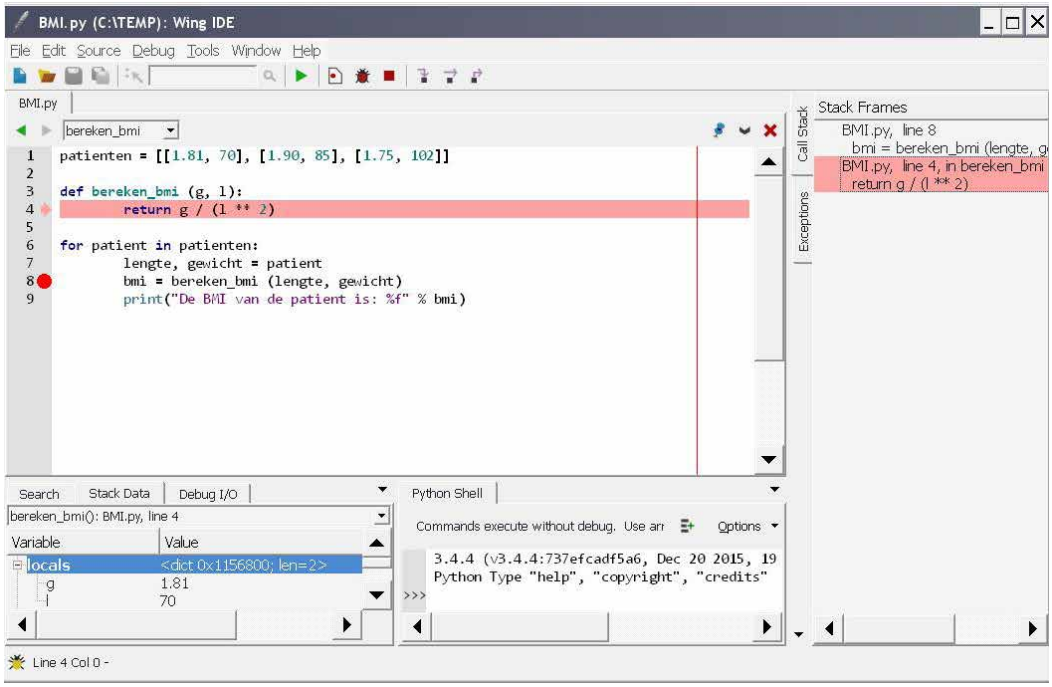

**Afb. 7** *Een breakpoint geplaatst.* *(Bron OGN.)*

Nadat u op *F5* gedrukt hebt, zal de debugger het programma herstarten en stoppen wanneer het breakpoint wordt bereikt.




**Afb. 8** *Het breakpoint bereikt.* *(Bron OGN.)*

Om te kunnen zien wat de waarden van de argumenten zijn bij het aanroepen van de functie, stappen we in de instructie met *F7*. Door op de tabbladen *Stack Data* en *Stack Frames* te klikken, komt extra informatie tevoorschijn (deze informatie zal in de volgende subparagraaf nader worden toegelicht). In afb. 9 is het volledige scherm weergegeven dat u dan te zien zult krijgen.




**Afb. 9** *Het tabblad met Stack Frames.* *(Bron OGN.)*

### Stack Frames

In grotere programma's hebben we vaak te maken met geneste functie-aanroepen. Wanneer we een breakpoint bij een instructie in een functie plaatsen, is het ook belangrijk om te weten van waaruit de functie is aangeroepen. Deze informatie is terug te vinden in het tabblad *Stack Frames*. Van boven naar beneden leest u vanaf welke regel in het programma een functie is aangeroepen (de onderste regel is de huidige functie).

### Variabelen inspecteren

Belangrijk bij het debuggen is om de waarden van variabelen en objecten te kunnen inspecteren. Daarvoor dient het tabblad *Stack Data*. Binnen een bepaalde scope (bijvoorbeeld *locals*) is af te lezen welke variabelen allemaal gedefinieerd zijn en wat de waarde van deze variabelen is.

Terug naar ons eigen programma. Op het tabblad *Stack Data* is af te lezen dat de waarde van *g* gelijk is aan 1.81, en de waarde van *l* gelijk is aan 70. Maar wacht even! *g* is vast een afkorting van gewicht, en *l* een afkorting van lengte. En een gewicht van 1.81 en een lengte van 70 lijken niet echt getallen uit de werkelijkheid. We zien hier dus dat de waarden van gewicht en lengte abusievelijk verwisseld zijn. We hebben de bug gevonden!

Nadere inspectie van de code wijst uit dat de volgorde van de argumenten van de functie *bereken_bmi* is: eerst gewicht en dan lengte. Bij het aanroepen van deze functie moeten we ook deze volgorde gebruiken!

De oplossing is de volgende regelcode aanpassen:

```
bmi = bereken_bmi (lengte, gewicht)
```

De juiste code is:

```
bmi = bereken_bmi (gewicht, lengte)
```

Wanneer we het programma aanpassen en opnieuw uitvoeren, krijgen we de volgende uitvoer:

```
De BMI van de patient is: 21.366869
De BMI van de patient is: 23.545706
De BMI van de patient is: 33.306122
```

# 3 GEGEVENS BENADEREN MET SQLITE

**Om gegevens die horen bij een Python-applicatie gestructureerd op te slaan, is een relationele database erg nuttig. Een van de meest gebruikte relationele databasemanagementsystemen wereldwijd is SQLite. SQLite bestaat uit een kleine softwarebibliotheek, die zonder verdere installatie of configuratie door een applicatie gebruikt kan worden. Alle onderdelen van een database (metadata, tabellen, indices enz.) worden opgeslagen in één enkel bestand. Vanaf Python 2.5 wordt SQLite met Python meegeleverd, zodat u SQLite direct in uw Python-applicatie kunt gebruiken, zonder dat verdere installatie noodzakelijk is.**

**In dit hoofdstuk behandelen we stap voor stap alle kennis die noodzakelijk is om SQLite effectief te kunnen gebruiken binnen Python. Allereerst bekijken we de essentie van relationele databases en de bijbehorende taal voor zoekopdrachten, SQL. Ook zullen we zien hoe de structuur van een relationele database weergegeven kan worden in een entiteit-relatiediagram (of ERD). Vervolgens bespreken we hoe u een SQLite-database direct kunt benaderen vanaf de SQLite-command-prompt of vanuit een GUI. Ten slotte laten we zien hoe u embedded SQL in uw Python-applicatie kunt gebruiken door middel van de sqlite3-module.**

## Wat is een relationele database?

Een relationele database is een database die gebaseerd is op het relationele model. Daarbij worden gegevens opgeslagen in tabellen, die door middel van sleutelvelden aan elkaar gekoppeld zijn.

De tabellen in een relationele database hebben de volgende eigenschappen:

a. elke tabel bestaat uit kolommen (behorende bij de velden) en rijen (deze worden ook wel *records* of *tuples* genoemd)
b. elke rij wordt uniek geïdentificeerd door één of meer velden die samen de *primaire sleutel* van de rij vormen
c. in principe komen er geen twee rijen voor met voor alle velden dezelfde waarden (zogenaamde *duplicates*)
d. tussen twee tabellen kan een *relatie* gelegd worden door een veld in de ene tabel (de zogenaamde *vreemde sleutel*) te laten verwijzen naar de primaire sleutel van een andere tabel
e. de rijen in een tabel hoeven niet gesorteerd te zijn.

Hierna ziet u als voorbeeld twee tabellen uit de database van een webwinkel (de tabellen *Artikelen* en *Bestelregels*).

*Tabel Artikelen*

| Artikelnummer | Artikel | Prijs | Voorraad |
|---|---|---|---|
| 503000 | T-shirt | 7,49 | 20 |
| 503010 | Poloshirt | 12,49 | 8 |
| 511000 | Japon | 15,00 | 3 |
| 511800 | Kaftan | 12,50 | 0 |
| 512000 | Vest | 17,50 | 5 |

*Tabel Bestelregels*

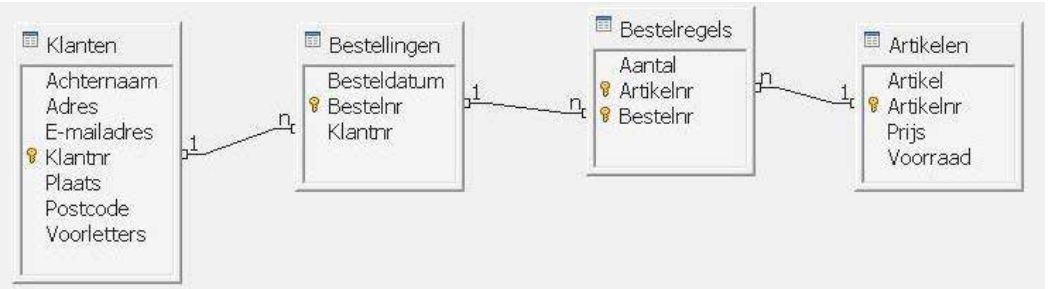
| Bestelnr | Artikelnr | Aantal |
|---|---|---|
| 1 | 503000 | 2 |
| 1 | 503010 | 5 |
| 2 | 503000 | 3 |
| 2 | 511800 | 1 |
| 2 | 512000 | 1 |

Elke bestelregel verwijst door middel van het veld *Artikelnr* (de vreemde sleutel) naar een rij in de tabel *Artikelen* (met *Artikelnr* als de primaire sleutel).

Door in het voorbeeld beide tabellen te combineren (door middel van de waarde van de sleutel *Artikelnr*) zien we dat bij bestelling 1 twee T-shirts en vijf poloshirts besteld zijn.

## Entiteit-relatiediagrammen

De structuur van een relationele database kunnen we weergeven in de vorm van een *entiteit-relatiediagram* (afgekort *ERD*). Elk object uit de werkelijkheid waarvoor we gegevens in de database willen vastleggen, wordt een *entiteit* genoemd. Voorbeelden van entiteiten zijn *Artikelen* en *Bestelregels*. De eigenschappen van entiteiten worden attributen genoemd. Een artikel heeft bijvoorbeeld de attributen *Artikelnr*, *Prijs* en *Voorraad*. In afb. 1 ziet u het ERD voor de webwinkeldatabase.




**Afb. 1** *Voorbeeld van een entiteit-relatiediagram.* *(Bron: OGN.)*

In de afbeelding zijn de attributen die als sleutel fungeren voorzien van een sleutel-icoontje; zo is bijvoorbeeld het attribuut *Klantnr* de primaire sleutel die hoort bij de entiteit *Klanten*. De lijnen tussen de entiteiten stellen de *relaties* tussen de entiteiten voor; zo zijn bijvoorbeeld *Bestellingen* gekoppeld aan *Klanten* door middel van *Klantnr*.

## SQL (Structured Query Language)

SQL is een wereldwijd gestandaardiseerde taal die gebruikt wordt voor het opvragen en/of bewerken van gegevens in een relationele database. SQL is geen procedurele taal, maar een taal die het gewenste resultaat beschrijft. Een relationeel database-managementsysteem (RDBMS) interpreteert SQL-opdrachten en voert vervolgens de opgedragen taken uit. Voorbeelden van een RDBMS zijn (naast SQLite) Oracle, Microsoft Access en MySQL. Al deze systemen gebruiken dezelfde taal voor het benaderen van de database, namelijk SQL.

### DDL, DML, DCL en TCL

Opdrachten in SQL kunnen onderverdeeld worden in vier verschillende categorieën:

a. *DDL (Data Definition Language)*: De DDL bestaat uit opdrachten waarmee de structuur van de database gedefinieerd wordt. De beschrijving van objecten in de databases wordt ook wel de *data dictionary* genoemd. Voorbeelden van DDL-opdrachten zijn:
   - CREATE – wordt gebruikt voor het aanmaken van de database of objecten in de database (zoals tabellen, indices en views)
   - DROP – wordt gebruikt om objecten uit de database te verwijderen
   - ALTER – wordt gebruikt om de structuur van de database aan te passen
   - RENAME – wordt gebruikt om de naam te wijzigen van een bestaand object in de database.

b. *DML (Data Manipulation Language)*: Met de opdrachten uit de DML kunnen gegevens uit de database opgehaald, ingevoerd, gewijzigd of verwijderd worden. Voorbeelden van DML-opdrachten zijn:
   - SELECT – wordt gebruikt voor het opzoeken en ophalen van bestaande records in de databasetabel
   - INSERT – wordt gebruikt voor het invoegen van nieuwe records in de databasetabel
   - UPDATE – wordt gebruikt voor het wijzigen van bestaande records in de databasetabel
   - DELETE – wordt gebruikt voor het verwijderen van bestaande records in de databasetabel.

c. *DCL (Data Control Language)*: Met opdrachten uit de DCL kunnen de toegangsrechten tot de database beheerd worden. Voorbeelden van DCL-opdrachten zijn:
   - GRANT – wijst bepaalde toegangsrechten toe aan een gebruiker van de database
   - REVOKE – neemt bepaalde rechten (die eerder met GRANT toegekend zijn) af van een gebruiker van de database.

d. *TCL (Transaction Control Language)*: Om te zorgen dat de gegevens in de database altijd consistent zijn, kunnen een aantal databasebewerkingen gegroepeerd worden in een zogeheten *transactie*. Dat betekent dat een groep bewerkingen alleen in zijn geheel wordt uitgevoerd, of helemaal niet. De opdrachten die betrekking hebben op transacties vormen samen de TCL. Voorbeelden van TCL-opdrachten zijn:
   - COMMIT – maakt alle wijzigingen uit een transactie definitief in de database, wanneer alle opdrachten in de transactie foutloos zijn uitgevoerd
   - ROLLBACK – draait alle wijzigingen uit de transactie terug als er een fout is opgetreden in een van de opdrachten in de transactie.

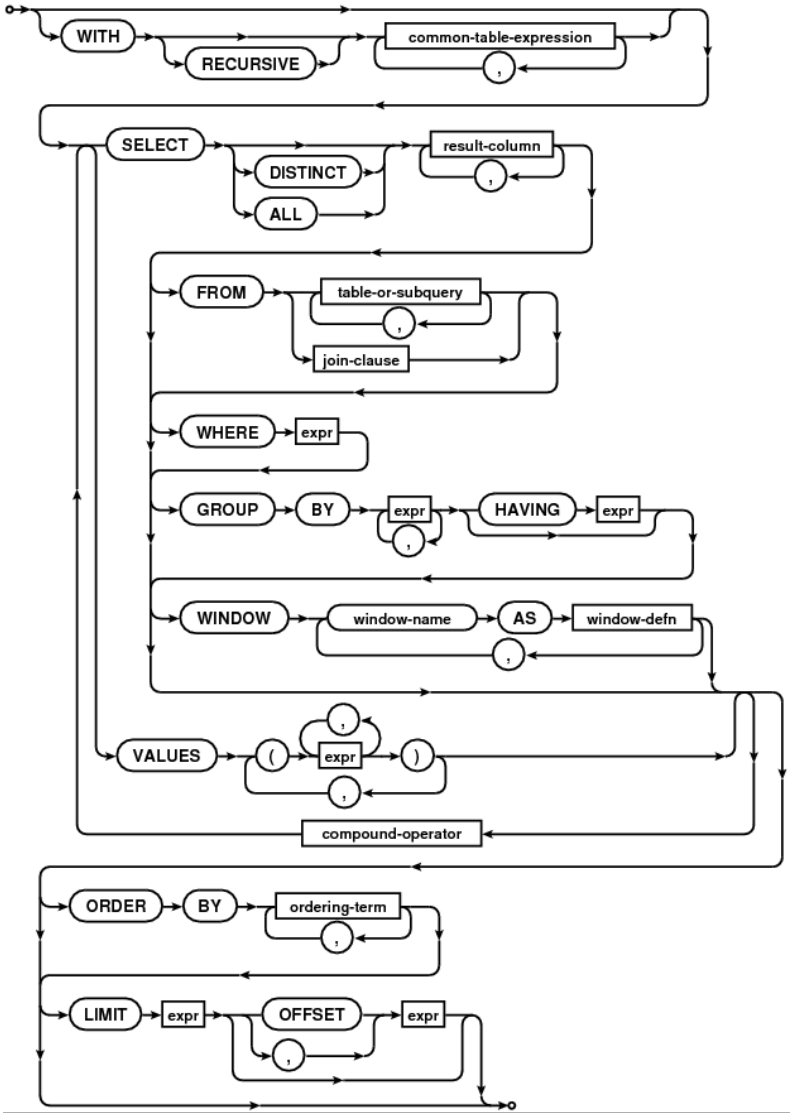
Opdrachten uit de DDL en DCL hebben betrekking op de structuur en configuratie van de database. Daarom worden deze opdrachten niet zo frequent gebruikt en is het meestal praktischer om deze opdrachten rechtstreeks aan de database te geven (via een command-prompt of GUI) en niet vanuit Python.

Opdrachten uit de DML en TCL hebben betrekking op de gegevens in de database zelf en worden meestal alleen vanuit programma's gegeven. Wanneer SQL-opdrachten opgenomen zijn in een computerprogramma (bijvoorbeeld geschreven in Python), spreken we van *embedded SQL*.

### DML-opdrachten

Een complete bespreking van SQL-opdrachten valt buiten het bestek van deze module; daarom zullen we hier volstaan met het geven van een syntaxisdiagram en een voorbeeld voor de opdrachten uit de DML (SELECT, INSERT, UPDATE en DELETE).
U vindt een goede uitleg van SQL-opdrachten op de website http://www.sqlitetutorial.net/; scroll direct door naar *Basic SQLite tutorial*. Daarnaast is een uitgebreide beschrijving van de syntaxis van SQL-opdrachten te vinden op https://www.sqlite.org/lang.html.

## De SELECT-opdracht



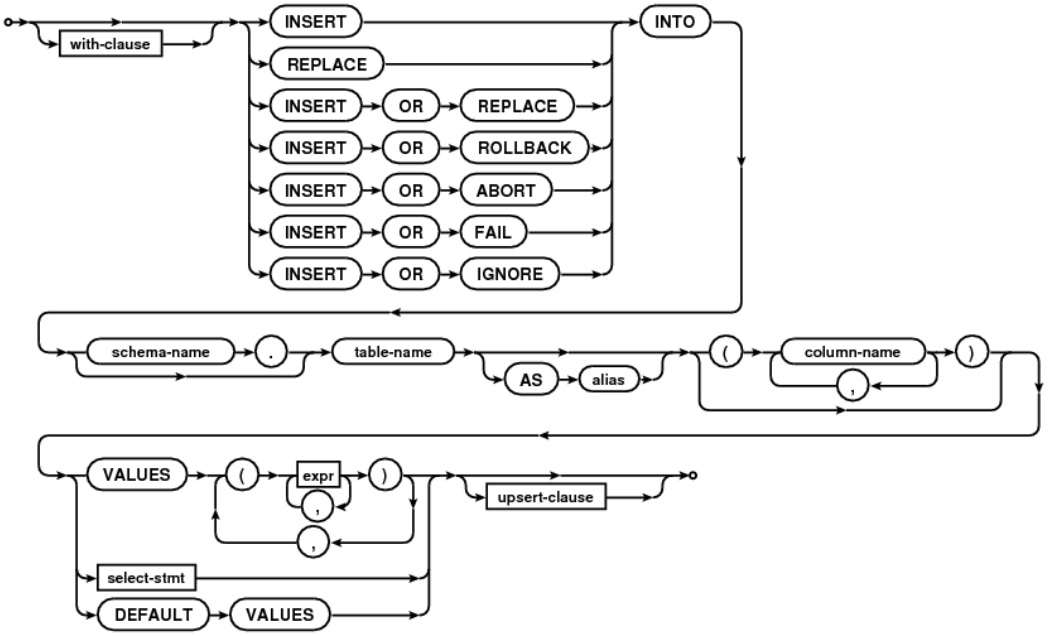

**Afb. 2** *Syntaxisdiagram voor de SELECT-opdracht.* *(Bron: SQLite.)*

*Voorbeeld*

Met de volgende query vinden we alle artikelen die uit voorraad zijn:

```
SELECT *
FROM Artikelen
WHERE Voorraad = 0;
```

## De INSERT-opdracht

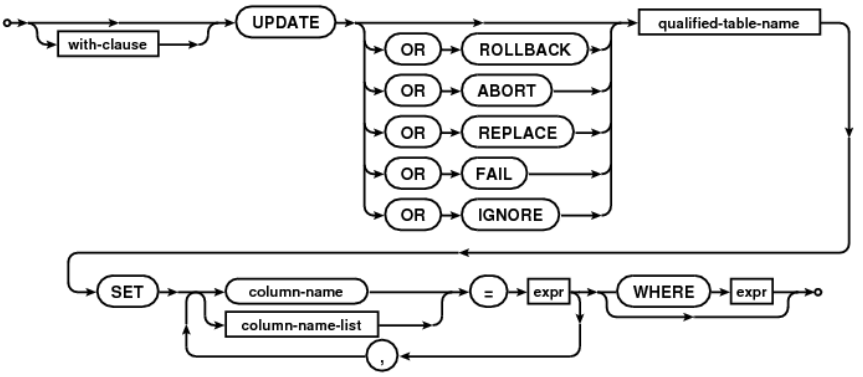



**Afb. 3** *Syntaxisdiagram voor de INSERT-opdracht.* *(Bron: SQLite.)*

*Voorbeeld*
In het assortiment van de webwinkel wordt een nieuw artikel opgenomen. Met de volgende opdracht voegen we een nieuw artikel toe aan de database:

```
INSERT INTO Artikelen
VALUES (512050, "Trui", 19.50, 0);
```

## De UPDATE-opdracht

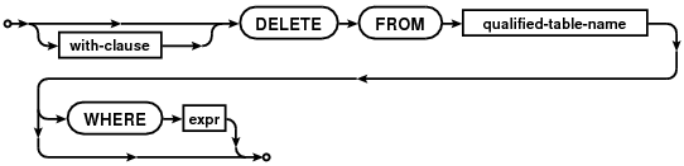



**Afb. 4** *Syntaxisdiagram voor de UPDATE-opdracht.* *(Bron: SQLite.)*

*Voorbeeld*
In het magazijn van de webwinkel zijn 4 nieuwe kaftans binnengekomen. Met de volgende opdracht kunnen we de voorraad in de database bijwerken:

```
UPDATE Artikelen
SET Voorraad = Voorraad + 4
WHERE Artikel = "Kaftan";
```

### De DELETE-opdracht




**Afb. 5** *Syntaxisdiagram voor de DELETE-opdracht.* *(Bron: SQLite.)*

*Voorbeeld*
Per abuis is er bij de bestelling met Bestelnr 2 een vest (met Artikelnr 512000) ingevoerd. Met de volgende opdracht verwijderen we het vest van de bestelling:

```
DELETE FROM Bestelregels
WHERE Bestelnr = 2
AND Artikelnr = 512000;
```

### JOIN van tabellen

Zoals eerder al in de paragraaf 'Wat is een relationele database?' beschreven, zijn de gegevens in verschillende tabellen aan elkaar gekoppeld door middel van sleutelvelden (dit zijn de relaties tussen entiteiten). Om deze koppeling in een query te realiseren, maken we gebruik van een zogenaamde *JOIN*.
SQL biedt twee manieren om deze koppeling te formuleren: met een WHERE-clausule of met een INNER JOIN-clausule. In beide gevallen wordt de *JOIN* gerealiseerd door twee sleutelvelden aan elkaar gelijk te stellen.

*Voorbeeld*
We willen voor alle bestelregels uit bestelling 1, naast de beschrijving van het artikel en het bestelde aantal, ook de prijs van het artikel en het subtotaal van de bestelregel weten. Daarvoor gebruiken we de volgende query:

```
SELECT B.Artikelnr, A.Artikel, B.Aantal, A.Prijs, B.Aantal * A.Prijs
FROM Bestelregels as B, Artikelen as A
WHERE Bestelnr = 1
AND B.Artikelnr = A.Artikelnr;
```

Een alternatieve formulering die hetzelfde resultaat oplevert is:

```
SELECT Bestelregels.Artikelnr, Artikel, Aantal, Prijs, Aantal *
Prijs
FROM Bestelregels
INNER JOIN Artikelen ON Bestelregels.Artikelnr = Artikelen.Ar-
tikelnr
WHERE Bestelnr = 1;
```

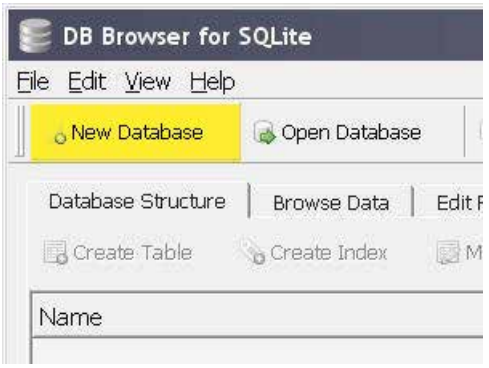
## Rechtstreekse toegang tot een SQLite-database

De snelste manier om toegang tot een SQLite-database te krijgen, is via de SQLite-command-prompt. Vanaf deze prompt kunnen SQL-opdrachten direct uitgevoerd worden. Nadeel is echter het vele typewerk; bovendien dient men de syntaxis van alle SQL-opdrachten te kennen.

Gelukkig bestaan er ook grafische gebruikersinterfaces (GUI's) om toegang tot een SQLite-database te krijgen. Een voorbeeld daarvan is *DB Browser for SQLite*; dit programma kan gratis gedownload worden vanaf https://sqlitebrowser.org/ en is beschikbaar voor alle gangbare platforms (Windows, Mac OS X en Linux).

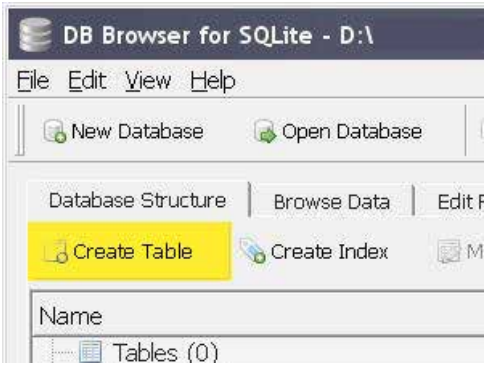
We laten nu stap voor stap zien hoe u vanuit DB Browser for SQLite de tabel *Artikelen* aan een nieuwe database *Webwinkel* toevoegt:

- Klik op *New database* om een nieuwe database te openen.




**Afb. 6** *Een nieuwe database openen.* *(Bron: SQLite.)*

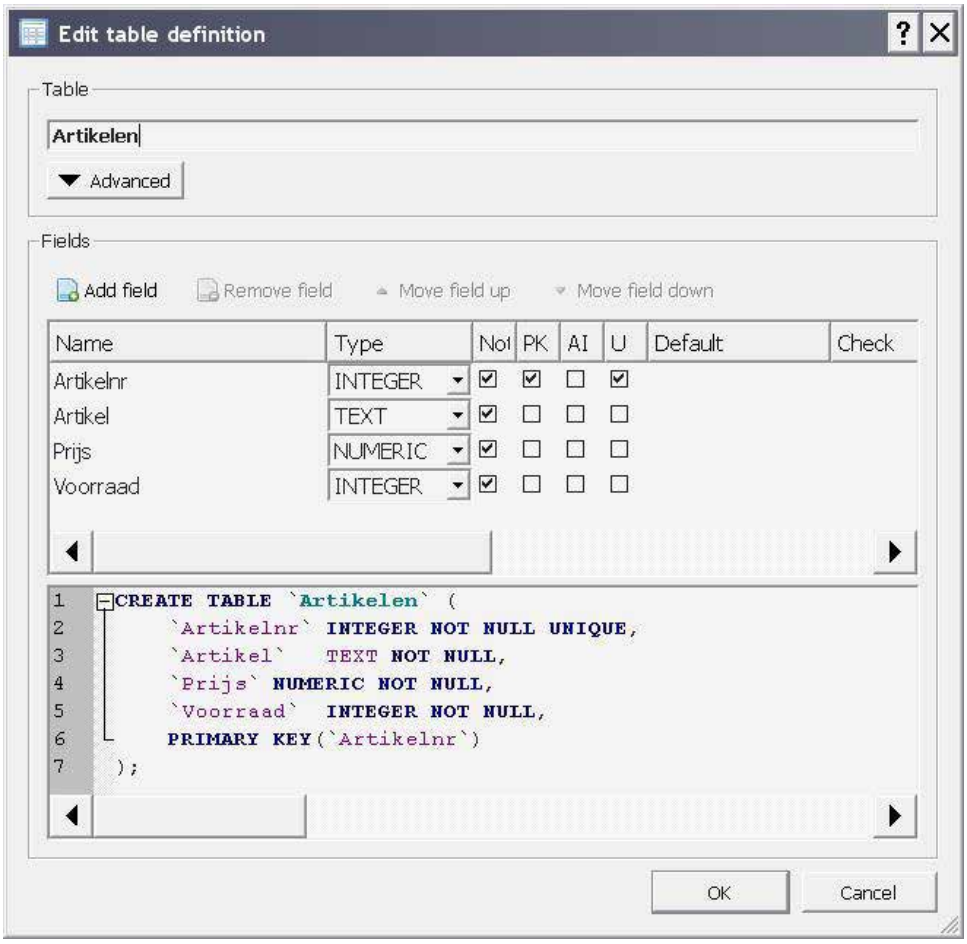
- Geef deze de naam "Webwinkel.db".
- Klik op *Create table* om een nieuwe tabel aan te maken.




**Afb. 7** *Een nieuwe tabel toevoegen.* *(Bron: SQLite.)*

- Het scherm *Edit table definition* verschijnt. In dit scherm definieert u de kolommen van de tabel (de attributen van een entiteit). Ook definieert u of een attribuut verplicht (Not Null) is, een primaire sleutel (PK) is en/of uniek (U) is. In een goed databaseontwerp is een primaire sleutel altijd verplicht en uniek.

In afbeelding 8 ziet u de definitie van de attributen van de tabel *Artikelen*.




**Afb. 8** *De definitie van attributen bekijken.* *(Bron: SQLite.)*

Onder in dit scherm wordt automatisch ook de DDL-opdracht gegenereerd (*CREATE TABLE ...)* die vanaf de SQLite-prompt of vanuit Python uitgevoerd kan worden.

## De sqlite3-module

Databasetoegang vanuit Python wordt geregeld door speciale modules (*Python Extensions*). Om te zorgen dat de interface naar verschillende RDBMsen bijna altijd hetzelfde is, is er een standaard opgesteld voor database-interfaces binnen Python: de *Python Database API Specification v2.0* (PEP 249), afgekort *DB-API 2.0*.

De module die toegang regelt tot een SQLite-database is de *sqlite3-module*. Deze module voldoet aan de specificaties voor DB-API 2.0 en wordt standaard bij Python 3.x meegeleverd. In deze paragraaf bekijken we welke functies de sqlite3-module allemaal biedt en hoe we deze functies vanuit Python moeten aanroepen.

### Import van de sqlite3-module

Om de sqlite3-module in een Python-programma te kunnen gebruiken, moeten we eerst een importopdracht aan het programma toevoegen:

```
import sqlite3
```

### Connection- en cursorobject

Voordat we een SQLite-database kunnen benaderen, maken we een verbinding met de database. Dit doen we door een connection-object met de naam *db* te creëren:

```
db = sqlite3.connect("C:\\temp\Webwinkel.db")
```

We hebben nu een verbinding gemaakt naar de database in het bestand “C:\ temp\ Webwinkel.db”.

Om een SQLite-database vanuit Python te kunnen benaderen met SQL-opdrachten hebben we een cursorobject nodig. Het resultaat van een query levert een resultaattabel op; met de cursor kunnen we ook door deze tabel scrollen. De cursor werkt dan als een wijzer naar de huidige rij in de resultaattabel.

Ook voor INSERT-, UPDATE- en DELETE-opdrachten wordt een cursor gebruikt; alleen leveren deze opdrachten geen resultaattabel op.

Met de volgende opdracht creëren we een cursorobject met de naam *cur*:

```
cur = db.cursor()
```

### Gegevens ophalen vanuit Python met een SELECT-opdracht

Nadat we een cursorobject gecreëerd hebben, kunnen we de cursor gebruiken voor het ophalen of bewerken van gegevens. Het ophalen van gegevens gebeurt in twee stappen:

- eerst voeren we de *execute*-methode van de cursor uit, waarbij we de SQL-opdracht opgeven die we willen laten uitvoeren
- daarna voeren we de *fetch*-methode van de cursor uit, waarmee de gegevens binnen het geheugen van Python geladen worden (als een list van tuples).

Het beste kunnen we het voorgaande verduidelijken aan de hand van een voorbeeld.

*Voorbeeld: gegevens ophalen met SELECT en een cursor*
Stel dat we in een Python-applicatie alle bestelregels willen ophalen voor de bestelling met nummer 6.

Eerst voeren we de SQL-opdracht uit met execute:

```
cur.execute(''' SELECT Bestelnr, Artikelnr, Aantal
FROM Bestelregels
WHERE Bestelnr = 6;''')
```

Daarna 'fetchen' we de data met *fetchall()*:

```
bestelregels = cur.fetchall()
```

De variabele *bestelregels* bevat nu de resultaattabel in de vorm van een list van tuples:

```
>>> print (bestelregels)
[(6, 503010, 4), (6, 514000, 5), (6, 514350, 3)]
```

De tuples zijn te herkennen aan de haakjes, de list aan de rechte haken.

We kunnen nu met een *for...in*-lus door de resultaattabel lopen. De volgende code berekent het totaalaantal artikelen in de bestelling met bestelnummer 6:

```
# Bereken het totale aantal artikelen voor deze bestelling
totaal = 0
for regel in bestelregels:
 aantal = regel[2]
 totaal += aantal
```

Bij de code uit het vorige voorbeeld zijn een aantal belangrijke opmerkingen te maken:

- In dit voorbeeld is de *fetchall()*-methode gebruikt. Daarmee wordt de gehele resultaattabel in één keer in het geheugen geladen. Bij grote databases kan dat echter problemen opleveren; dan verdient de *fetchone()*-methode de voorkeur. Bij de *fetchone()*-methode wordt slechts één rij ingelezen.
- Voor de leesbaarheid is de SQL-opdracht over meer regels verdeeld. Daarvoor is een multiline-string gebruikt, die omsloten is door drie apostroffen.
- Ook weer omwille van de leesbaarheid zijn alle SQL-elementen van de opdracht met hoofdletters geschreven.
- De juiste kolom van de resultaattabel kan geselecteerd worden door middel van de tuple-index. In het voorbeeld bevat de derde kolom het *Aantal*; deze kolom wordt geselecteerd met de index [2].
- Het gebruik van "*SELECT **" dient voorkomen te worden, omdat het aantal kolommen in de tabel van een database in de loop van de tijd kan wijzigen (er komen nieuwe kolommen bij of sommige kolommen worden verwijderd). Door de namen van de kolommen expliciet te benoemen, weet u zeker welke velden in de query worden teruggegeven en op welke positie deze velden staan.

We zetten voor u nog even op een rij welke stappen u dient te doorlopen om een query op de database uit te voeren:

- Eerst een verbinding maken met de SQLite-database door een *Connection*-object aan te maken.
- Vervolgens een *Cursor*-object aanmaken met de methode *cursor* van het *Connection*-object.
- Dan de SELECT-opdracht uitvoeren met de methode *execute* van het *Cursor*-object.
- Daarna de gegevens ophalen naar het geheugen met de methode *fetchall()* van het *Cursor*-object.
- Ten slotte door de list lopen van de resultaattabel.

### Geparametriseerde zoekopdrachten

In het voorbeeld uit de vorige subparagraaf was het bestelnummer waarop gezocht werd een constante waarde. In een Python-applicatie heeft men echter bijna altijd te maken met variabelen die in een zoekopdracht gebruikt worden. SQLite biedt een manier om waarden in een zoekopdracht variabel te maken, met behulp van zogenaamde *dynamic binding*. Op de plaats van de variabele waarde zet u een ?; dit is de zogenaamde plaatsvervanger (Engels: *placeholder*). Als tweede argument bij het aanroepen van de methode *execute* geeft u een tuple met de echte waarden die op de plaats van de vraagtekens moeten komen.

Ook nu kan een voorbeeld het principe van een geparametriseerde zoekopdracht het beste verduidelijken.

*Voorbeeld: een geparametriseerde zoekopdracht*
Het volgende programma vraagt de gebruiker om een bestelnummer, zoekt vervolgens de bestelregels in de database en print deze op het scherm.

```
db = sqlite3.connect("C:\\temp\Webwinkel.db")
cur = db.cursor()
# Vraag gebruiker om het bestelnummer
bestelnr = int( input("Geef het bestelnummer: "))
# Bestelregels ophalen uit database
cur.execute(''' SELECT Bestelnr, Artikelnr, Aantal
FROM Bestelregels
  WHERE Bestelnr = ?;''', (bestelnr,))
bestelregels = cur.fetchall()
print (bestelregels)
```

*Let op!* Bij gebruik van een plaatsvervanger in de *execute*-methode dient het tweede argument een tuple te zijn. In het voorbeeld is één plaatsvervanger gebruikt (het vraagteken), de tuple moet dus uit één element bestaan. Volgens de syntaxisregels van Python moet er na de identifier een komma volgen, om als tuple herkend te worden (dus: < (*identifier*,)>).

### Uitvoeren van INSERT, UPDATE en DELETE vanuit Python

Het uitvoeren van INSERT, UPDATE en DELETE gaat op dezelfde manier als het uitvoeren van SELECT. Bij deze opdrachten hebben we bijna altijd te maken met gegevens in variabelen. Het werken met variabelen gaat op dezelfde manier als hiervoor uitgelegd voor geparametriseerde zoekopdrachten, namelijk met vraagtekens als plaatsvervangers voor de waarden van de variabelen.

Er is echter wel een belangrijk ding waarop u moet letten bij SQL-opdrachten die wijzigingen aanbrengen in de database. Deze wijzigingen worden pas definitief nadat u een COMMIT hebt uitgevoerd met de methode *commit* van het *Connection*-object. Zonder een COMMIT zullen de wijzigingen verloren gaan!

We zullen nu volstaan met het geven van drie voorbeelden.

*Voorbeeld: INSERT met plaatsvervangers*

```
bestelnr = 15
artikelnr = 512050
aantal = 3
cur.execute(''' INSERT INTO Bestelregels
VALUES (?, ?, ?);''', (bestelnr, artikelnr, aantal))
db.commit()
```

*Voorbeeld: UPDATE met plaatsvervangers*

```
mutatie = 5
artikelnr = 512050
cur.execute(''' UPDATE Artikelen
SET Voorraad = Voorraad + ?
WHERE Artikel = ?;''', (mutatie, artikelnr))
db.commit()
```

*Voorbeeld: DELETE met plaatsvervanger*

```
bestelnr = 15
cur.execute(''' DELETE FROM BESTELREGELS
WHERE BESTELNR = ?;''', (bestelnr,))
db.commit()
```

# 4 AANVULLINGEN BIJ HET BOEK VAN HALTERMAN

**Bij een aantal onderwerpen dekt de tekst van het boek van Halterman de materie niet volledig. Daarom vindt u in dit hoofdstuk bij deze onderwerpen een korte aanvulling. Deze aanvullingen zijn:**

- **A1. Alles in Python is een object**
- **A2. Gestructureerd programmeren**
- **A3. Werken met binaire bestanden.**

## A1. Alles in Python is een object

Hiervoor hebt u geleerd wat variabelen en identifiers zijn. In het boek van Halterman is echter een zeer belangrijk concept van Python buiten beschouwing gebleven. Python is een objectgeoriënteerde programmeertaal, net als C++ en Java. Een belangrijk verschil met andere programmeertalen, echter, is dat binnen Python *alles een object is*: variabelen, functies, strings, bestanden: het zijn allemaal objecten van een verschillend type. Een identifier wijst dus naar een object in het geheugen; Python slaat daar het type object en de inhoud van het object op. Met de functie *id()* vinden we het geheugenadres waar een object is opgeslagen; met de functie *type()* kunnen we het type object bepalen.

*Voorbeeld*

```
a = 3
print (id(a))
print (type(a))
```

De uitvoer bij deze code:

```
505996176
<class 'int'>
```

Het hiervoor beschreven objectmodel van Python verschilt sterk van andere programmeertalen, waarbij de compiler een identifier vertaalt naar een geheugenadres. Twee verschillende identifiers van hetzelfde type wijzen dan altijd naar twee verschillende geheugenadressen. In Python echter, kunnen twee verschillende identifiers naar hetzelfde geheugenadres verwijzen! Bekijk maar eens het volgende voorbeeld.

*Voorbeeld*
We vergelijken twee stukken code (één stuk geschreven in C++ en één stuk geschreven in Python). De stukken code lijken op het eerste gezicht equivalent.

Code in C++:

```
void main()
{
int a, b;
a = 3;
b = 3;
cout << "Adres van variabele a: " << &a << "\n";
cout << "Adres van variabele b: " << &b << "\n";
}
```

De uitvoer van dit programma is:

```
Adres van variabele a: 0013FF60
Adres van variabele b: 0013FF54
```

We zien hier dat a en b verwijzen naar verschillende geheugenlocaties.
Code in Python:

```
a = 3
b = 3
print ("Adres van object a: ", id(a))
print ("Adres van object b: ", id(b))
```

Dit programma in Python levert de uitvoer:

```
Adres van object a: 505996176
Adres van object b: 505996176
```

We zien dus dat a en b naar hetzelfde object verwijzen! Anders gezegd: a en b zijn *aliassen* voor hetzelfde object.

Er zijn nog andere verstrekkende gevolgen van de manier waarop Python met objecten omgaat. We komen hierop terug bij de bespreking van lists in Python.

# A2. Gestructureerd programmeren

Programmeren is niet alleen een vaardigheid, maar ook een kunst. En code krijgen die werkt, is niet eens de grootste uitdaging. De grootste uitdaging bij het programmeren is om code te schrijven die ook door andere programmeurs goed te volgen is, en daardoor eenvoudiger (en vooral ook goedkoper!) te onderhouden is. Zulke code bevat vaak ook minder bugs en is gemakkelijker te testen.

Een beproeft recept voor het schrijven van betere code is het *gestructureerd programmeren*. In deze aanvulling bij de module Programmeren in Python bespreken we welke logische structuren men in een computerprogramma kan herkennen. Ook leert u hoe u deze logische structuren in schema's kunt weergeven met programmastructuurdiagrammen.

### Wat is gestructureerd programmeren?

Sinds de opkomst van computers en computerprogramma's heeft men gezocht naar manieren om betere code voor computerprogramma's te krijgen. Met name in de eerste- en tweedegeneratieprogrammeertalen had de programmeur de vrijheid een programma letterlijk alle kanten op te laten springen (met de beruchte *GOTO*-opdracht). Het gevolg: spaghetti-code waar geen touw meer aan vast te knopen was. Deze programma's bleken zeer kostbaar in het onderhoud te zijn, omdat een kleine wijziging op één plaats in het programma tot heel veel fouten op andere plaatsen kon leiden.

Twee paradigma's die populair geworden zijn voor het schrijven van betere programmacode zijn *gestructureerd programmeren* (geïntroduceerd in de jaren zeventig) en *objectgeoriënteerd programmeren* (geïntroduceerd halverwege de jaren negentig).

In de praktijk kunnen beide benaderingen naast elkaar gebruikt worden (ze zijn complementair):

- gestructureerd programmeren is belangrijk op kleine schaal (binnen een blok code)
- objectgeoriënteerd programmeren wordt gebruikt om de complexiteit van grotere applicaties en systemen te beheersen.

Bij gestructureerd programmeren worden een aantal regels en principes gevolgd voor de *flow* van een programma. Het basisprincipe van gestructureerd programmeren is dat blokken programmalogica opgesplitst kunnen worden in kleinere blokken (zogeheten *decompositie*). Bij het opsplitsen van een blok logica zijn er slechts vier structuren die gebruikt kunnen worden:

a. de *elementaire actie* (deze kan niet verder in kleinere delen opgesplitst worden)
b. de *sequentie* (oftewel opeenvolging) van acties
c. de *selectie* (oftewel keuze) tussen acties
d. de *iteratie* (oftewel herhaling) van acties.

De structuur van een programma is onafhankelijk van de gekozen programmeertaal. Er is zelfs een grafische methode bedacht om de structuur van een programma weer te geven, namelijk met een zogeheten *programmastructuurdiagram*. Deze schemavorm is in 1973 geïntroduceerd door Isaac Nassi en Ben Shneiderman, vandaar dat het programmastructuurdiagram ook wel een *Nassi-Shneidermandiagram* genoemd wordt.

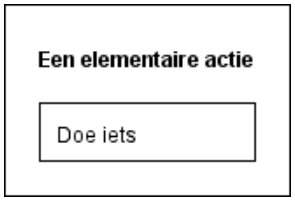
Het fraaie van het programmastructuurdiagram is dat als de logica van een programma in een programmastructuurdiagram weergegeven kan worden, het programma altijd goed gestructureerd is. Nog belangrijker is dat het omgekeerde ook geldt: als de logica van een programma *niet* met een programmastructuurdiagram weergegeven kan worden, is het programma *niet goed gestructureerd*. Zo'n programma heeft ongecontroleerde sprongen in de flow van de logica, waardoor het programma slecht leesbaar wordt en moeilijker te onderhouden.

## Programmastructuurdiagrammen

In deze paragraaf werpen we een nadere blik op de structuren binnen een programmastructuurdiagram (afgekort PSD). We doen dit aan de hand van een voorbeeld: het recept voor het bakken van een omelet. We zullen stap voor stap de verschillende structuren uitbouwen, totdat we uiteindelijk het recept voor het bakken van een omelet in een PSD gegoten hebben. Daarmee zouden we een robotkok kunnen programmeren!

### De elementaire actie

Een elementaire actie geven we in een PSD weer met een rechthoek.

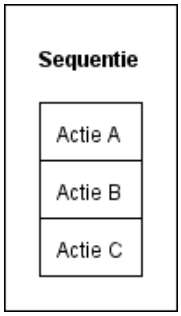



**Afb. 1** *Schemavorm voor een elementaire actie.* *(Bron: OGN.)*

Een elementaire actie moet uitgevoerd kunnen worden in de programmeertaal die u gebruikt; dat kan een statement zijn (zoals *teller = 0*) of een functie-aanroep. De vorm van een elementaire actie is altijd *werkwoord + object*.

### De sequentie

Een sequentie wordt in een PSD weergegeven met een aantal rechthoeken onder elkaar (zie afb. 2). De acties A, B en C worden *in volgorde* na elkaar uitgevoerd (van boven naar onder gelezen in het PSD).




**Afb. 2** *Schemavorm voor een sequentie.* *(Bron: OGN.)*

*Voorbeeld*
Het verwerken van een ei kunnen we opdelen in een aantal opeenvolgende acties (zie afb. 3).

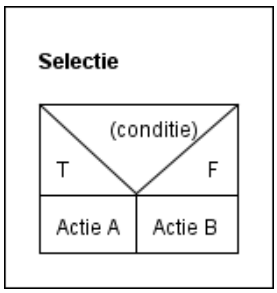

**Afb. 3** *Voorbeeld van een sequentie.* *(Bron: OGN.)*

Uit het diagram lezen we af dat de volgorde van handelingen is:

- eerst een ei pakken
- daarna het ei breken
- ten slotte het ei in een kom doen.

## De selectie

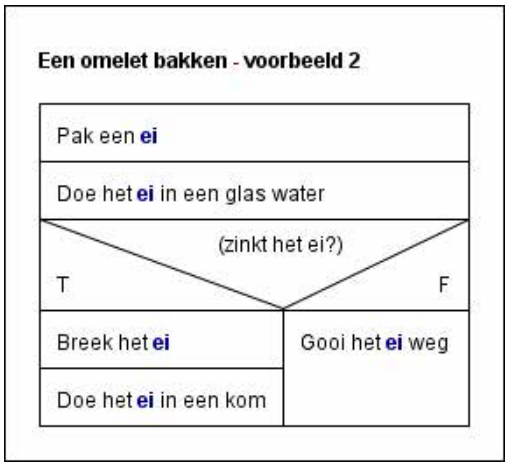
Bij een selectie maken we een keuze tussen twee of meer alternatieve acties op basis van een voorwaarde (*conditie*). We kunnen een selectie tussen de alternatieve acties A en B op basis van een conditie weergeven in een PSD als in afbeelding 4.




**Afb. 4** *Schemavorm voor een selectie.* *(Bron OGN.)*

De letters *T* en *F* staan voor respectievelijk *waar* (in het Engels: *True*) en *onwaar* (in het Engels: *False*).

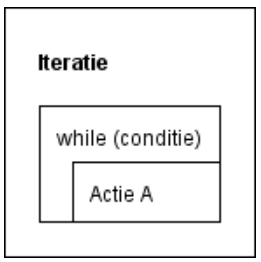
*Voorbeeld*
Een echte chef-kok werkt natuurlijk alleen met echt verse producten. Daarom test hij van tevoren de versheid van de eieren. Wanneer hij een ei in een glas water doet en het ei zinkt, is het ei vers. Gaat het ei zweven of stijgt het ei op? Dan is het ei niet meer vers en gooit hij het weg. Deze procedure kan worden weergegeven in een PSD als in afbeelding 5.




**Afb. 5** *Voorbeeld van een selectie.* *(Bron OGN.)*

### De iteratie

Soms is het in een programma nodig om een actie (of een blok logica) een aantal malen te herhalen. Zo'n herhaling wordt een *iteratie* genoemd en als wordt weergegeven in een PSD als in afbeelding 6.




**Afb. 6** *Schemavorm voor een iteratie.* *(Bron: OGN.)*

De actie A wordt herhaald zolang de conditie waar is. Pas als de conditie niet meer waar is, wordt deze lus onderbroken. Het is daarbij belangrijk dat de conditie kan veranderen (bijvoorbeeld in de omgeving van het programma of als gevolg van actie A). Zo niet, dan blijft het programma in een eeuwige lus zitten, wat bijna nooit de bedoeling is.

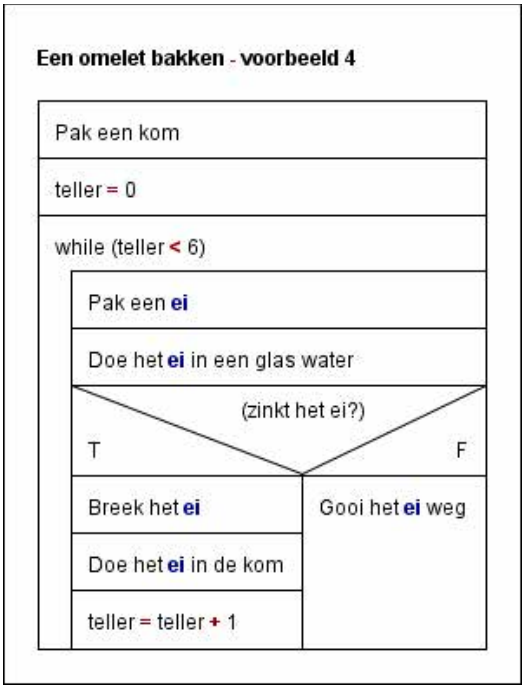
*Voorbeeld*
Onze chef-kok heeft om een omelet te bakken meer eieren nodig. Stel dat hij een half dozijn eieren nodig heeft. In het PSD van afb. 7 wordt de procedure (sequentie) uit afb. 3 zesmaal herhaald.


**Afb. 7** *Voorbeeld van een iteratie.* *(Bron: OGN.)*

Let in dit voorbeeld goed op de initialisatie van de teller (0) en het testen van de teller (<6). De teller loopt dus van 0 t/m 5 (wat overeenkomt met 6 eieren); wanneer de teller de waarde 6 bereikt, wordt de herhalingslus onderbroken.

We kunnen de structuren sequentie, selectie en iteratie ook combineren om meer complexe logica te bouwen. Bekijk maar eens het volgende voorbeeld.

*Voorbeeld*
De chef-kok controleert nu ook elk ei van de 6 stuks die hij nodig heeft voor de omelet. We krijgen dan het PSD uit afb. 8.




**Afb. 8** *Voorbeeld van een combinatie van structuren.* *(Bron: OGN.)*

## Een compleet programma

Nu we alle structuren van een PSD gezien hebben, kunnen we ook een compleet programma in een PSD weergeven.

*Voorbeeld*

De complete procedure voor het bakken van een omelet ziet u weergegeven in het PSD van afb. 9. Eet smakelijk!


**Afb. 9** *Voorbeeld van een compleet programma.* *(Bron: OGN.)*

In vorenstaand voorbeeld ziet u een andere variant van de iteratie, namelijk de until-lus. Het blok binnen de lus wordt net zo lang uitgevoerd totdat de conditie waar wordt.

# A3. Werken met binaire bestanden

In de tekstmodus worden bij het lezen van bestanden de platformspecifieke einde-regelmarkeringen (\n in Unix, \r\n in Windows) omgezet naar de standaardeinde-regelmarkeringen. Bij het schrijven naar bestanden gebeurt het omgekeerde. Deze bewerkingen, die Python achter de schermen uitvoert, zijn echter ongewenst voor bestanden met binaire data (zoals JPEG of MP3). Daarom heeft Python ook een binaire modus voor het lezen/schrijven van/naar bestanden. In deze aanvulling bespreken we hoe u met bestanden in de binaire modus werkt.

### De binaire modus selecteren

Bij het openen van een bestand dient u als het tweede argument van de *open()*-functie ook de *modus* op te geven. De syntaxis van het openen van een bestand is:

```
file_object = open(<bestandsnaam>, <modus>)
```

In afbeelding 10 ziet u alle mogelijke waarden voor modus. Door als tweede teken 'b' op te geven, wordt de binaire modus geselecteerd.

NB: Wordt het tweede teken weggelaten, dan wordt als default de tekstmodus geselecteerd.

| modus | tekstmodus | binaire modus |
|---|---|---|
| Lezen (read) | 'rt' (default) | 'rb' |
| Schrijven (write) | 'wt' | 'wb' |
| Schrijven aan het einde (append) | 'at' | 'ab' |
| Schrijven maar niet óverschrijven | 'xt' | 'xb' |
| Lezen en schrijven (update) | 'rt+' | 'rb+' |

**Afb. 10** *Mogelijke waarden voor het argument modus van de open()-functie.*

### Een bestand lezen met read()

Nadat een bestand geopend is in de binaire leesmodus, kunnen gegevens uit het bestand opgehaald worden met *read()*. Zonder argument wordt het gehele bestand ingelezen, maar wees daarmee voorzichtig! Het bestand kan zo groot zijn dat het geheugen volloopt en het programma niet verder kan. Als argument kan het aantal bytes opgegeven worden dat ingelezen moet worden.

```
>>> f = open ("loi_logo.bmp", 'rb')
>>> bytes = f.read(16)
>>> f.close()
```

Wanneer een binair bestand ingelezen wordt, geeft Python het resultaat terug in de vorm van een *bytes*-object: een serie van hexadecimale getallen (die wel of niet

overeen kunnen komen met een alfanumeriek teken) die elk de waarde van één byte voorstellen.

```
>>> print (bytes)
b'BMF;\x00\x00\x00\x00\x00\x006\x00\x00\x00(\x00'
```

Een *bytes*-object werkt bijna hetzelfde als een tekststring; zo kunnen we ook elementen van een bytes-object selecteren met slicing:

```
>>> print (bytes[8:12])
b'\x00\x006\x00'
```

Merk op dat het *bytes*-object in het vorenstaande voorbeeld bestaat uit 4 bytes; de waarde van het derde byte is hexadecimaal '\x36', wat overeenkomt met het symbool *6*:

```
>>> byte = '\x36'
>>> print (byte)
6
```

'Gewoon' afdrukbare en niet-afdrukbare tekens worden in een *bytes*-object dus verschillend weergegeven.

Met de *bytes()*-functie kunnen we een leeg *bytes*-object creëren. Een *bytes*-object kan niet gewijzigd worden (het is *immutable*).

```
>>> lege_bytes = bytes(4)
>>> lege_bytes [0] = b'\xff'
Traceback (most recent call last):
 File "<pyshell#2>", line 1, in <module>
 lege_bytes [0] = b'\xff'
TypeError: 'bytes' object does not support item assignment
```

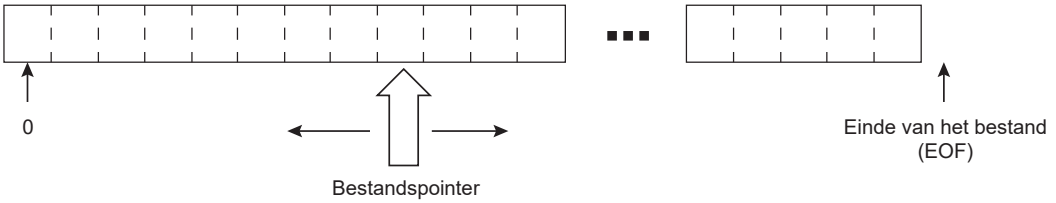
Een objecttype dat wel *mutable* is en waarin ook bytes opgeslagen kunnen worden, is *bytearray*.

```
>>> lege_bytes = bytearray(4)
>>> lege_bytes [0] = 255
>>> print (lege_bytes)
bytearray(b'\xff\x00\x00\x00')
```

Merk op dat aan een element van een *bytearray* een waarde toegekend wordt in de vorm van een integer-constante in plaats van een bytestring.

## De bestandspointer

Een bestandspointer in Python is een offset-getal van het type *integer*, dat de plaats aanwijst van de volgende byte die gelezen of geschreven moet worden.




**Afb. 11** *Een bestandspointer.* *(Bron: OGN.)*

De huidige waarde van een bestandspointer van een bestandsobject kan opgevraagd worden met de functie *tell()*.

```
>>> f = open ("loi_logo.bmp", 'rb')
>>> bytes = f.read()
>>> print(f.tell())
15174
>>> f.close()
```

In het vorenstaande voorbeeld zijn dus 15.174 bytes ingelezen.

We kunnen de bestandspointer ook naar een specifieke locatie in het bestand verplaatsen met de functie *seek()*. Deze functie heeft als syntaxis:

```
file_object.seek(<offset>, <origin>)
```

*Offset* is het aantal bytes ten opzichte van een positie bepaald door *origin*. De mogelijke waarden van *origin* zijn:

- 0 (default) het begin van het bestand (positie 0)
- 1 de huidige positie
- 2 het einde van het bestand.

Om de 64ste byte uit het bestand uit te lezen:

```
>>> f = open ("loi_logo.bmp", 'rb')
>>> f.seek(64)
64
>>> byte = f.read(1)
>>> print (byte)
b'\xf4'
>>> f.close()
```

### Een bestand in blokken inlezen

Een bestand kan dermate groot zijn dat het niet in zijn geheel in het RAM-geheugen van de computer past. Dan is het verstandiger om het bestand in blokken in te lezen. Het einde van het bestand (de *EOF-markering*) kunnen we herkennen aan b'' (*empty binary*). We lezen net zolang blokken vanuit het bestand totdat we deze EOF-markering tegenkomen.

```
blokgrootte = 256  # Blokgrootte bepaald door de programmeur

f = open ("loi_logo.bmp", 'rb')
volgende_blok = f.read(blokgrootte)

while not (volgende_blok == b''):      # EOF?
# Doe iets met het blok
volgende_blok = f.read(blokgrootte)

print(f.tell())    # Aantal ingelezen bytes
f.close()
```

### Naar een bestand schrijven met write()

Schrijven naar een bestand doen we met de functie *write()*. We moeten het bestand natuurlijk wel eerst openen in de schrijfmodus.

Het volgende programma maakt een kopie van het bestand "*loi_logo.bmp*", waarin alle pixels in de kleur lichtblauw vervangen zijn door rode pixels:

```
blokgrootte = 3

lichtblauw = b'\xe3\x9e\x00'
rood = b'\x00\x00\xff'

f_in = open ("loi_logo.bmp", 'rb')
f_uit = open ("logo_rood.bmp", 'wb')

header = f_in.read(54)
f_uit.write(header)

volgende_blok = f_in.read(blokgrootte)

while not (volgende_blok == b''):      # EOF?
if volgende_blok == lichtblauw:
f_uit.write(rood)
else:
f_uit.write(volgende_blok)
volgende_blok = f_in.read(blokgrootte)

f_in.close()
f_uit.close()
```