Je eigen games maken met Python 3e editie

Geschreven door Al Sweigart

Vertaald door Marjo Hahn

Boekversie 3.0.1, ISBN 978-1503212305

Als je dit boek hebt gedownload vanaf een torrent, is het waarschijnlijk niet meer actueel. Ga maar naar http://inventwithpython.com om de laatste versie te downloaden.

Voor Caro, van wie ik meer houd

dan ik zelf voor mogelijk hield.

Bericht aan ouders en medeprogrammeurs

Bedankt dat je dit boek leest. De reden dat ik het heb geschreven is dat ik zag dat er een gebrek was aan boeken voor kinderen die willen leren programmeren. Ik begon zelf met programmeren in de programmeertaal BASIC na een soortgelijk boek gelezen te hebben.

Tijdens het schrijven van dit boek kwam ik tot de ontdekking dat een moderne taal als Python het programmeren veel makkelijker en veelzijdiger heeft gemaakt voor een hele nieuwe generatie programmeurs. Python is tamelijk gemakkelijk te leren, maar is toch een serieuze programmeertaal die door professionele programmeurs overal ter wereld wordt gebruikt.

De programmeerboeken die er vandaag de dag zijn, kun je onderverdelen in twee categorieën. Aan de ene kant zijn er boeken die niet echt programmeren leren maar het meer hebben over software voor het maken van spelletjes of die talen aanbevelen die zo simplistisch zijn dat het weinig meer met programmeren te maken heeft. Aan de andere kant zijn er boeken die progammeren onderwijzen alsof het een wiskundeboek is, alleen maar principes en begrippen zonder oog voor de lezer. Dit boek benadert het anders. Vanaf het allereerste begin laten we de broncode voor spelletjes zien en we leggen de programmeerbeginselen uit aan de hand van de voorbeelden.

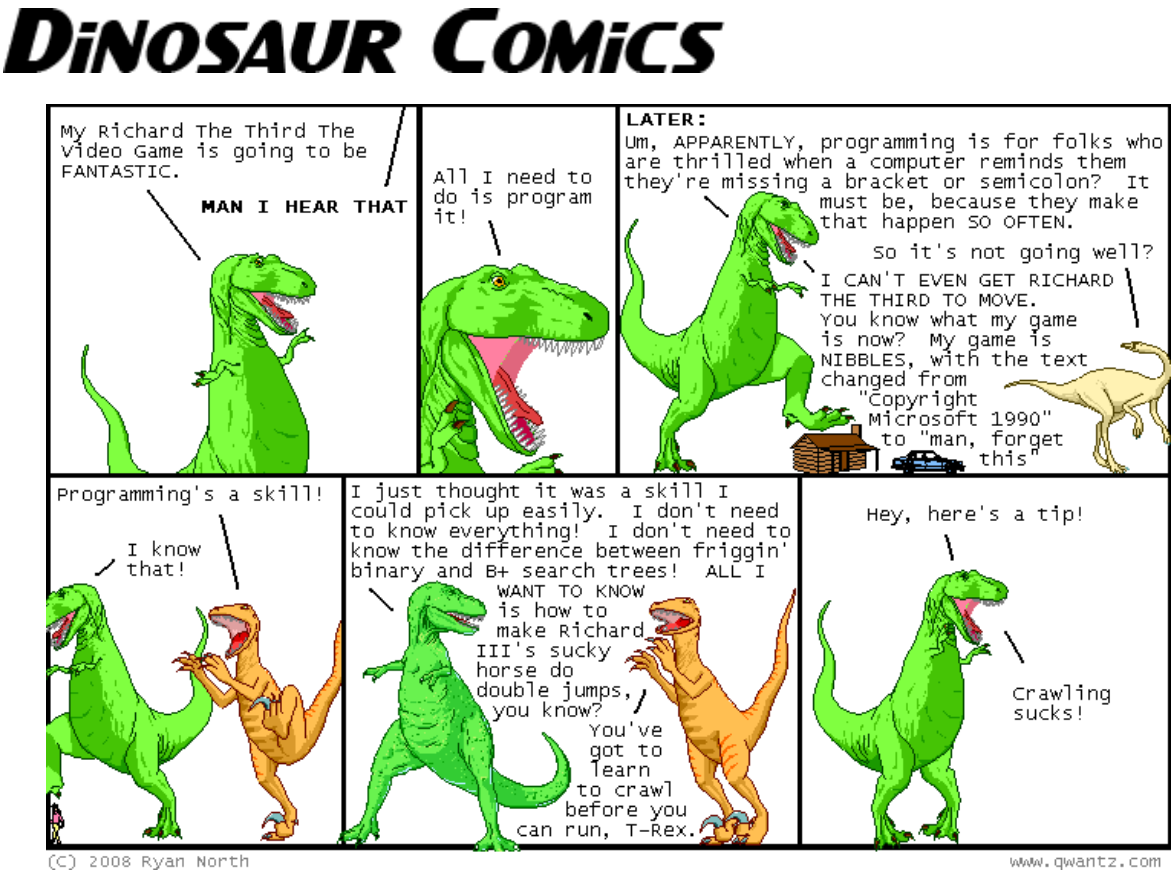
Het boek is beschikbaar onder de Creative Commons-licentie, wat betekent dat je kopieën mag maken en het boek (of onderdelen ervan) mag distribueren, mits ik genoemd blijf als auteur en je het niet gebruikt om zelf geld aan te verdienen. (Zie de copyrightpagina.) Dit boek is een geschenk aan de wereld die me zo veel goeds heeft gebracht.

Wat is er nieuw in de 3e editie?

De derde editie heeft geen nieuwe inhoud vergeleken met de 2e editie. De derde editie is wat meer gestroomlijnd, zodat dezelfde inhoud wordt behandeld op 20% minder pagina's. Waar nodig is de uitleg uitgebreid en zijn onduidelijkheden verduidelijkt.

Hoofdstuk 9 is gesplitst in de hoofdstukken 9 en 9½ om de hoofdstuknummering hetzelfde te houden.

De broncode is expres niet veranderd om geen verwarring te veroorzaken. Als je de tweede editie al hebt gelezen hoef je dit boek niet meer te lezen, tenzij je liever in het Nederlands leest. Maar als je een beginner bent met programmeren of een vriend of vriendin kennis wilt laten maken met programmeren, is dat met deze derde editie makkelijker en hopelijk nog leuker geworden.



Voor wie is dit boek bestemd?

Programmeren is niet moeilijk. Maar het is wel lastig om goed lesmateriaal te vinden dat je leert hoe je iets interessants kunt doen met programmeren. Andere computerboeken behandelen onderwerpen die de meeste beginnende programmeurs niet nodig hebben. Dit boek leert je hoe je je eigen computerspelletjes kunt programmeren. Je leert een nuttige vaardigheid en aan het eind van de lessen heb je er ook nog een paar leuke games bij! Dit boek is voor:

- Echte beginners die zichzelf programmeren willen leren, ook als ze nog geen enkele programmeerervaring hebben.
- Kinderen en tieners die willen leren programmeren door spelletjes te maken.
- Volwassenen en docenten die anderen willen leren programmeren.
- Iedereen, jong of oud, die wil leren programmeren met behulp van een professionele programmeertaal.

Hoofdstuk 1

PYTHON INSTALLEREN

In dit hoofdstuk behandelen we:
- Het downloaden en installeren van de Python-interpreter
- De interactieve shell van IDLE
- Hoe je dit boek moet gebruiken
- De website van het boek op http://inventwithpython.com

Hoi! Dit boek leert je programmeren doordat we samen videogames gaan maken. Zodra je begrijpt hoe de games in dit boek werken, kun je je eigen games gaan maken. Het enige wat je nodig hebt, is een computer, wat software – de zogenaamde Python-interpreter – en dit boek. De Python-interpreter kun je gratis downloaden van internet.

Toen ik klein was, kreeg ik ook zo'n soort boek als dit in handen en daarmee leerde ik om mijn eerste programma's en spelletjes te schrijven. Het was gemakkelijk en leuk om te doen. Nu, als volwassene, vind ik het nog steeds leuk en krijg ik er zelfs voor betaald. Maar ook als je later geen computerprogrammeur wordt, is het toch handig en leuk om te kunnen programmeren.

Computers zijn fantastische machines en het is niet moeilijk om ze te leren programmeren. Als je dit boek kunt lezen, kun je een computer programmeren. Een computer**programma** is gewoon een reeks instructies die door een computer worden uitgevoerd, net zoals een boek een hele reeks zinnen is die door een lezer worden gelezen. Videogames zijn ook gewoon computerprogramma's, dus ze bestaan ook uit een reeks instructies die worden uitgevoerd.

Om een instructie aan een computer te geven, moet je een programma schrijven in een taal die de computer begrijpt. In dit boek leer je een programmeertaal die Python heet. Er zijn veel verschillende programmeertalen, zoals bijvoorbeeld BASIC, Java, JavaScript, PHP en C++.

Toen ik jonger was, leerden de meeste mensen BASIC als eerste taal. Sinds die tijd zijn er echter een paar nieuwe programmeertalen bedacht, zoals Python. Python is nog makkelijker te leren dan BASIC! Maar toch is het ook een serieuze programmeertaal die door professionele programmeurs wordt gebruikt. Veel volwassenen gebruiken Python in hun werk en wanneer ze programmeren voor hun plezier.

De games die je met behulp van dit boek gaat maken, lijken eenvoudig, vergeleken met de games die je kent van de Xbox, Playstation of Wii. Onze games hebben geen vette graphics, maar dat komt gewoon omdat het de bedoeling is om de beginselen van het programmeren te leren. Ze zijn met opzet eenvoudig gehouden, zodat je je kunt richten op leren programmeren. En games hoeven niet ingewikkeld te zijn om toch leuk te zijn.

Python downloaden en installeren

De software die je moet installeren, is de zogenaamde Python-interpreter. Het **interpreter**-programma begrijpt de instructies die jij in de Python-taal schrijft. Van nu af aan noem ik de 'Python-interpreter-software' gewoon 'Python'.

Download Python 3.4, of een latere versie, van de officiële website van Python, http://www.python.org. Download de 32-bits versie van Python voor jouw besturingssysteem, ook als je een 64-bits computer hebt. De Pygame-module die we later in het boek gaan gebruiken, draait momenteel namelijk alleen op 32-bits Python.

> **Belangrijke opmerking!** Let erop dat je Python 3 installeert, en niet Python 2. De programma's in dit boek gebruiken Python 3, en als je ze probeert uit te voeren met Python 2, krijg je foutmeldingen. Het is zo belangrijk, dat ik een pinguïnstripfiguur heb toegevoegd in Afbeelding 1-1 die je vertelt Python 3 te installeren, zodat je dit bericht niet over het hoofd ziet.

Afbeelding 1-1: Een verdwaalde pinguïn zegt dat je Python 3 moet installeren.

IDLE starten

IDLE is de afkorting van **I**nteractive **D**eve**L**opment **E**nvironment (Interactieve ontwikkelomgeving). De ontwikkelomgeving is een soort wordprocessor voor het schrijven van Python-programma's. Hoe je IDLE start, is op elk besturingssysteem een beetje anders.

In Windows klik je op de Startknop in de linkerbenedenhoek van je scherm, typ je 'IDLE' en selecteer je **IDLE (Python GUI - 32 bit)**.

Op de Mac open je het Finder-venster en klik je op **Programma's**. Vervolgens klik je op **Python 3.4**. En dan op het pictogram IDLE.

Op Ubuntu of Linux open je een schermvenster en typ je 'idle3'. Je kunt mogelijk ook klikken op **Applications** boven aan het scherm. Klik vervolgens op **Programming** en **IDLE 3**.

Afbeelding 1-2: De interactieve shell van IDLE op Windows, OS X en Ubuntu Linux.

Het venster dat verschijnt als je IDLE voor het eerst start, heet de **interactieve shell**. Je kunt Python-instructies in de interactieve shell typen en Python voert ze dan uit. Python toont het resultaat van de instructies ook in de interactieve shell.

Hoe je dit boek moet gebruiken

De meeste hoofdstukken in dit boek beginnen met een voorbeeld van de uitvoer van het programma dat we schrijven. In dat voorbeeld kun je zien hoe het programma eruitziet als je het uitvoert. Wat de gebruiker intypt, wordt in het boek weergegeven met **vette** letters.

Je kunt de code van het programma beter zelf in IDLE typen dan het te downloaden. Als je de code zelf typt, onthoud je veel beter wat je leert over programmeren.

Regelnummers en spaties

Wanneer je de code typt, typ dan **niet** de regelnummers aan het begin van elke regel. Als je dit bijvoorbeeld ziet in het boek:

```
9. getal = random.randint(1, 20)
```

moet je '9.' niet typen en ook niet de spatie erna. Typ het gewoon zo:

```
getal = random.randint(1, 20)
```

Die nummers aan het begin van de regel gebruiken we alleen om tijdens de uitleg naar de regels te kunnen verwijzen. Ze horen niet bij het programma zelf.

Behalve de regelnummers moet je de code *precies* zo invoeren als het er staat. Zoals je kunt zien, worden sommige regels ingesprongen met vier of acht spaties. Die inspringingen moet je ook precies volgen, want die hebben een betekenis voor Python. Elk teken in IDLE heeft dezelfde breedte. Je kunt het aantal spaties dus tellen door het aantal tekens te tellen boven of onder de regel.

De ingesprongen ruimte hier wordt bijvoorbeeld aangegeven met zwarte vierkantjes ▪:

```
while aantalBeurten < 10:
▪▪▪▪if getal == 42:
▪▪▪▪▪▪▪▪print('Hallo')
```

Tekstdoorloop in dit boek

Sommige regels code zijn te lang om op één regel op de pagina te passen en lopen door op de volgende regel. Voer al deze code achter elkaar in, zonder op ENTER te drukken. Je kunt zien waar een

nieuwe regel begint door naar de regelnummers links van de code te kijken. Onderstaande code heeft bijvoorbeeld eigenlijk maar twee regels code:

```
1. print('Dit is de eerste regel! xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxx')
2. print('Dit is de tweede regel en niet de derde regel.')
```

De eerste regel loopt door en daardoor lijken het drie regels te zijn. Maar dat komt alleen maar omdat de pagina's van dit boek niet breed genoeg zijn om de code allemaal op die ene regel te laten passen.

Help online vinden

De website van dit boek is op http://inventwithpython.com. Je kunt daar verschillende informatiebronnen over dit boek vinden. Een aantal koppelingen in dit boek gebruiken de domeinnaam invpy.com voor kortere URL's. (URL's zijn internetadressen, zoals je misschien al weet.)

De subreddit website op http://reddit.com/r/inventwithpython is een prima plek om programmeervragen te stellen die te maken hebben met dit boek. Richt algemene vragen over Python aan de subreddits LearnProgramming en LearnPython op http://reddit.com/r/learnprogramming en http://reddit.com/r/learnpython. Dat moet wel in het Engels.

Je kunt me ook e-mailen (in het Engels) met je programmeervragen via *al@inventwithpython.com*.

Vergeet niet dat er slimme manieren zijn om programmeervragen te stellen zodat anderen je makkelijker kunnen helpen. Kijk even naar de sectie Frequently Asked Questions (Veelgestelde vragen) op deze websites voor tips over hoe je op een goede manier vragen stelt. Als je een programmeervraag hebt, doe dan eerst het volgende:

- Als je de programma's in dit boek intypt en een foutmelding krijgt, controleer dan eerst of je een typefout hebt gemaakt met behulp van de online diff tool op http://invpy.com/diff. Kopieer en plak je code in de diff tool en de verschillen tussen de code in het boek en jouw code worden vanzelf aangegeven.
- Als je over de fout vertelt, leg dan uit wat je probeert te doen. Dan weet jouw helper of je helemaal op de verkeerde weg bent of niet.
- Kopieer en plak de hele foutmelding en je code in je vraag.
- Zoek op het web of iemand jouw vraag misschien al heeft gesteld (of beantwoord).
- Vertel wat je al hebt geprobeerd om het probleem op te lossen. Dan weten anderen dat je zelf ook al het een en ander hebt geprobeerd.
- Wees beleefd. 'Eis' geen help en zet je helpers ook niet onder druk om snel antwoord te geven.

De programma's online traceren

Een programma traceren betekent stapsgewijs door de code gaan, regel voor regel, op dezelfde manier als een computer het programma zou uitvoeren. Ga naar http://invpy.com/traces om een tracering te zien van alle programma's in dit boek. De webpagina biedt commentaar en nuttige tips bij elke stap in de tracering om uit te leggen wat het programma aan het doen is, zodat je beter kunt begrijpen waarom de programma's doen wat ze doen.

Samenvatting

Dit hoofdstuk heeft je geholpen aan de slag te gaan met de Python-software door je te vertellen over de website http://python.org, waar je deze software gratis kunt downloaden. Na het installeren en starten van de Python IDLE-software, ben je er klaar voor om te gaan leren programmeren. Daar beginnen we mee in het volgende hoofdstuk.

De website van dit boek op http://inventwithpython.com heeft meer informatie over alle hoofdstukken, waaronder een website voor online tracering en een diff-tool, die je kunnen helpen de programma's in dit boek beter te begrijpen.

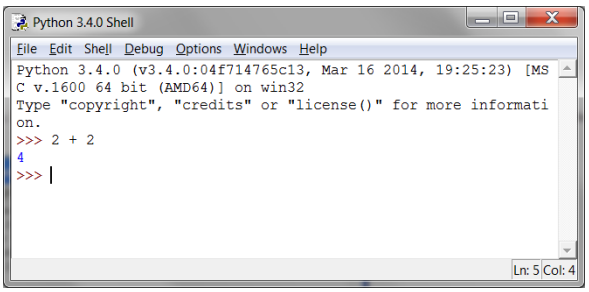
Hoofdstuk 2
DE INTERACTIEVE SHELL

In dit hoofdstuk behandelen we:
- Integers (gehele getallen) en getallen met drijvende komma
- Expressies
- Waarden
- Operatoren
- Het evalueren van expressies
- Het opslaan van waarden in variabelen

Voordat je kunt beginnen met het maken van games, moet je een paar elementaire begrippen in de programmeerkunst onder de knie krijgen. We gaan in dit hoofdstuk nog geen games maken, maar als we hebben geleerd wat deze begrippen betekenen, wordt het programmeren van videogames straks een stuk makkelijker. Laten we beginnen met hoe we de interactieve shell van Python kunnen gebruiken.

Een paar makkelijke rekensommetjes

Open IDLE zoals in hoofdstuk 1 werd uitgelegd en gebruik Python dan om wat makkelijke rekensommen voor je te maken. De interactieve shell kun je namelijk ook gebruiken als een rekenmachine. Typ `2 + 2` in de interactieve shell en druk op de ENTER-toets op je toetsenbord. (Op sommige toetsenborden is dit de RETURN-toets.) Afbeelding 2-1 laat zien hoe IDLE reageert: met het getal `4`.



Afbeelding 2-1: Typ 2+2 in de interactieve shell.

Dit rekensommetje is een simpele programmeerinstructie. Het plusteken (+) vertelt de computer dat de cijfers 2 en 2 bij elkaar moeten worden opgeteld. In Tabel 2-1 zie je de andere rekensymbolen die in Python worden gebruikt. Met het minteken (-) trek je getallen van elkaar af. Met het sterretje (*) vermenigvuldig je getallen. Met de slash (/) deel je getallen.

Tabel 2-1: De verschillende rekenkundige operatoren in Python.

Operator	Bewerking
+	optellen
-	aftrekken
*	vermenigvuldigen
/	delen

Als deze tekens (+, -, * en /) op deze manier worden gebruikt, worden ze **operatoren** genoemd. Operatoren vertellen Python welke rekenbewerking moet worden uitgevoerd op de omringende getallen.

Integers en getallen met drijvende komma

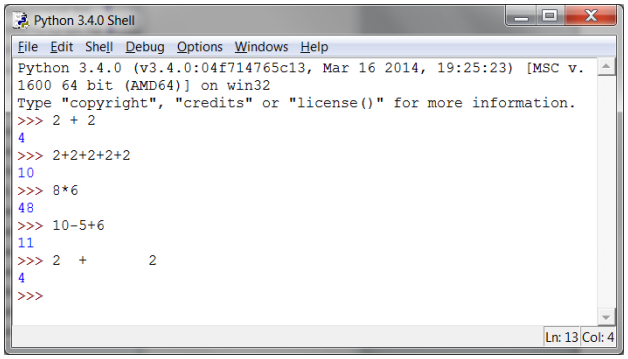
In het programmeren wordt een geheel getal, zoals 4, 99 en 0, een **integer** of **int** genoemd. Een breuk of een getal met een decimale komma, zoals 3,5 of 5,0 heet een **getal met drijvende komma** of, lekker kort, een **float**. In Python is het cijfer 5 een integer, maar 5,0 is een float.

Expressies

Deze rekenopgaven zijn voorbeelden van expressies. Computers kunnen miljoenen van dit soort opgaven oplossen in een paar seconden. Expressies bestaan uit **waarden** (de getallen) die met elkaar worden gecombineerd door middel van **operatoren** (de rekenkundige symbolen). We gaan hier even wat dieper op in. Probeer een paar van deze rekenopgaven in de interactieve shell te typen. Na elke expressie druk je op ENTER.

```
2+2+2+2+2
8*6
10-5+6
2  +       2
```

Als je bovenstaande instructies hebt ingetypt, ziet de interactieve shell eruit zoals in Afbeelding 2-2.



Afbeelding 2-2: Wat je in het IDLE-venster ziet als je de instructies hebt getypt.

Afbeelding 2-3: Een expressie bestaat uit waarden en operatoren.

Een **expressie** bestaat uit waarden (zoals de integers `8` en `6`) die met elkaar worden gecombineerd door een operator (zoals het sterretje `*`, het vermenigvuldigingsteken). Een waarde in zijn uppie is trouwens ook een expressie. In het voorbeeld `2 + 2` zie je dat er spaties mogen staan tussen de waarden en de operatoren. Dat is geen probleem. Je mag alleen geen spaties plaatsen aan het begin van deze instructie als je de instructie typt in de interactieve shell.

In het volgende hoofdstuk vertel ik je meer over het werken met tekst in expressies. Python beperkt zich namelijk niet tot alleen getallen, het is veel meer dan alleen een grote rekenmachine!

Het evalueren van expressies

Wanneer een computer de expressie `10 + 5` oplost en de waarde `15` meldt, heeft de computer de expressie **geëvalueerd**. Het evalueren van een expressie levert één enkele waarde op, net zoals het maken van een som resulteert in één getal, de uitkomst.

De expressies `10 + 5` en `10 + 3 + 2` hebben dezelfde waarde. Ze geven als resultaat allebei `15`. Het geven van een resultaat wordt in het programmeren ook wel 'retourneren' genoemd. En de uitkomst heet de 'retourwaarde'.

Expressies kunnen lang of kort zijn, maar ze retourneren altijd één waarde. Ook een waarde in zijn eentje is een expressie. Dan zeggen we: de expressie `15` retourneert de waarde `15`. De expressie `8 * 3 / 2 + 2 + 7 - 9` retourneert uiteindelijk de waarde `12,0`. Dit gaat via de volgende stappen:

```
8 * 3 / 2 + 2 + 7 - 9
                ▼
   24 / 2 + 2 + 7 - 9
                ▼
     12,0 + 2 + 7 - 9
                ▼
         14,0 + 7 - 9
                ▼
             21,0 - 9
                ▼
                12,0
```

Je ziet al die stappen niet voorbijkomen in de interactieve shell. De interactieve shell voert ze uit en toont jou alleen het resultaat:

```
>>> 8 * 3 / 2 + 2 + 7 - 9
12,0
```

Merk op dat de deeloperator `/` een floatwaarde retourneert: `24 / 2` retourneert `12,0`. Rekenbewerkingen met floatwaarden retourneren ook altijd een floatwaarde: `12,0 + 2` retourneert `14,0`.

Syntaxfouten

Als je `5 +` in de interactieve shell typt, krijg je een foutmelding.

```
>>> 5 +
SyntaxError: invalid syntax
```

Deze fout treedt op omdat `5 +` geen expressie is. Zoals je al weet, hebben expressies waarden die met elkaar worden gecombineerd door operatoren. Maar bij de operator + wordt een waarde verwacht na het plusteken. Omdat deze waarde ontbreekt, wordt er een fout geconstateerd.

`SyntaxError` (letterlijk: grammaticafout) betekent dat Python de instructie niet begrijpt omdat je hem verkeerd hebt getypt. Computers programmeren is niet alleen een kwestie van de computer vertellen wat hij moet doen; je moet ook precies weten hoe je dat moet vertellen.

Maar maak je geen zorgen over het maken van fouten. Fouten beschadigen de computer niet. Verbeter de instructie gewoon in de interactieve shell achter het volgende >>>-teken.

Waarden opslaan in variabelen

Als je programmeert, wil je vaak de waarde die een expressie heeft geretourneerd, bewaren, zodat je deze waarde later in het programma nog eens kunt gebruiken. Je kunt waarden opslaan in **variabelen**. Je kunt aan een variabele denken als een doosje waarin je iets van waarde kunt doen.

Met een **toewijzingsstatement** sla je een waarde (of het resultaat van een expressie) op in een variabele. Bedenk een naam voor de variabele, typ het teken = (ook wel de **toewijzingsoperator** genoemd) en typ vervolgens de waarde die je in de variabele wilt opslaan. Typ bijvoorbeeld `ham = 15` in de interactieve shell:

```
>>> ham = 15
>>>
```

Afbeelding 2-4: Variabelen zijn net doosjes waarin je waarden kunt stoppen.

Het variabeledoosje `ham` bevat de waarde 15, zoals je kunt zien in Afbeelding 2-4. De naam 'ham' is het etiket dat je op het doosje plakt (zodat Python de variabelen uit elkaar kan houden) en de waarde zie je op het kleine papiertje in het doosje.

Wanneer je nu op ENTER drukt, zie je geen reactie. Je kunt in Python aannemen dat de instructie goed wordt uitgevoerd, want als het niet goed gaat, krijg je een foutmelding. Het >>>-teken verschijnt, zodat je de volgende instructie kunt typen.

Anders dan expressies retourneren **'statements'** geen waarden. Daarom wordt er ook geen waarde weergegeven op de volgende regel in de interactieve shell. Misschien vind je het een beetje lastig om te weten welke instructies expressies zijn en welke instructies statements zijn. Onthoud gewoon dat

expressies een waarde retourneren. Alle andere soorten instructies zijn statements (het Engelse woord voor 'verklaring').

In variabelen worden waarden bewaard, geen expressies. Kijk bijvoorbeeld eens naar de statements `ham = 10 + 5` en `ham = 10 + 7 - 2`. Ze geven als resultaat allebei `15`. Het eindresultaat is hetzelfde: beide statements slaan de waarde `15` op in de variabele `ham`.

De eerste keer dat een variabele wordt gebruikt in een toewijzingsstatement, wordt die variabele door Python aangemaakt. Als je wilt zien welke waarde in de variabele 'zit', typ je de naam van de variabele in de interactieve shell:

```
>>> ham = 15
>>> ham
15
```

De expressie `ham` retourneert de waarde in de variabele `ham`: `15`. Je kunt in expressies ook variabelen gebruiken. Probeer het volgende eens te typen in de interactieve shell:

```
>>> ham = 15
>>> ham + 5
20
```

Je hebt de waarde van de variabele `ham` op `15` gezet, dus als je `ham + 5` schrijft, schrijf je eigenlijk de expressie `15 + 5`.

Je kunt een variabele pas gebruiken als je hem met een toewijzingsstatement hebt aangemaakt. Anders krijg je een foutmelding, want dan weet Python nog niet van het bestaan van die variabele. Als je de naam van de variabele verkeerd typt, krijg je ook een foutmelding:

```
>>> ham = 15
>>> hma
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    spma
NameError: name 'hma' is not defined
```

Deze fout treedt op omdat er wel al een variabele is met de naam `ham` maar `hma` wordt niet herkend.

Je kunt de waarde die in een variabele wordt bewaard, wijzigen door een andere toewijzingsstatement te typen. Probeer bijvoorbeeld eens het volgende te typen:

```
>>> ham = 15
>>> ham + 5
20
>>> ham = 3
>>> ham + 5
8
```

Wanneer je de eerste keer `ham + 5` typt, retourneert de expressie `20` omdat je `15` had opgeslagen in `ham`. Als je daarna echter `ham = 3` typt, wordt de `15` vervangen, of **overschreven**, met de waarde `3`. Als je nu `ham + 5` typt, retourneert de expressie `8` omdat de waarde van `ham` nu `3` was.

Je kunt de naam van de variabele `ham` zelfs gebruiken om een nieuwe waarde aan `ham` toe te wijzen:

```
>>> ham = 15
```

```
>>> ham = ham + 5
20
```

De toewijzingsstatement `ham = ham + 5` zegt eigenlijk: "de nieuwe waarde van de variabele `ham` wordt de huidige waarde van `ham` plus vijf". Blijf de waarde van `ham` een paar keer verhogen met `5`:

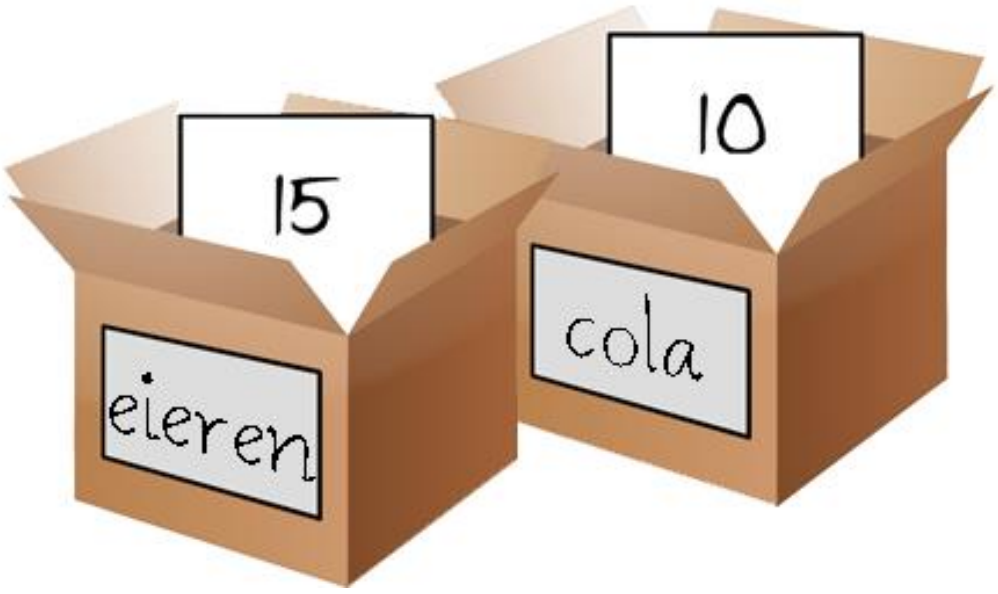
```
>>> ham = 15
>>> ham = ham + 5
>>> ham = ham + 5
>>> ham = ham + 5
>>> ham
30
```

Meer dan één variabele gebruiken

Maak zoveel variabelen als je nodig hebt in je programma's. Laten we bijvoorbeeld verschillende waarden toewijzen aan twee variabelen, genaamd `eieren` en `cola`, als volgt:

```
>>> cola = 10
>>> eieren = 15
```

Nu bevat de variabele `cola` de waarde `10` en `eieren` bevat de waarde `15`.



Afbeelding 2-5: De variabelen 'cola' en 'eieren' bevatten waarden.

Probeer `ham = cola + eieren` in de interactieve shell te typen en kijk dan wat de nieuwe waarde van `ham` wordt:

```
>>> cola = 10
>>> eieren = 15
>>> ham = cola + eieren
>>> ham
25
```

De waarde van `ham` is nu `25`. Toen je `cola` en `eieren` bij elkaar optelde, telde je eigenlijk hun waarden bij elkaar op, die respectievelijk 10 en 15 waren.

Samenvatting

In dit hoofdstuk heb je al een aantal beginselen geleerd van het schrijven van instructies in Python. Python begrijpt alleen wat je wilt als je het op een hele duidelijke en precieze manier vertelt. Computers zijn maar domme dingen en begrijpen alleen hele specifieke instructies.

Expressies zijn waarden (zoals `2` of `5`) die met elkaar worden gecombineerd door operatoren (zoals `+` of `-`). Python kan expressies evalueren (dat wil zeggen de expressie een waarde laten retourneren). Je kunt waarden opslaan in variabelen, zodat je programma de waarde kan onthouden en verderop in het programma kan gebruiken.

Er zijn veel andere typen operatoren en waarden in Python. In het volgende hoofdstuk gaan we het hebben over nog een paar begrippen en ga je je eerste programma schrijven!

Hoofdstuk 3
PROGRAMMA'S SCHRIJVEN

In dit hoofdstuk behandelen we:
- Hoe de verwerking van instructies verloopt
- Strings (tekenreeksen)
- Stringconcatenatie (oftewel strings aan elkaar plakken)
- Datatypen (zoals strings of integers)
- IDLE gebruiken om code te schrijven
- Programma's opslaan en uitvoeren in IDLE
- De functie `print()`
- De functie `input()`
- Commentaar
- Hoofdlettergebruik bij variabelen
- Hoofdlettergevoeligheid

Nou, we hebben nu wel genoeg gerekend voorlopig. Python is veel meer dan alleen een rekenmachine. We gaan nu ontdekken wat Python kan doen met tekst. In dit hoofdstuk ga je leren hoe je tekst kunt opslaan in variabelen, hoe je tekst met andere tekst kunt combineren en hoe je tekst kunt weergeven op het scherm. Bijna alle computerprogramma's kunnen tekst aan de gebruiker tonen en de gebruiker kan tekst in een programma typen door middel van het toetsenbord. Je gaat ook je eerste programma maken. Dit programma laat de groet 'Hallo wereld!' zien en vraagt hoe je heet.

Strings

In Python worden tekstwaarden **strings** genoemd. (Strings worden ook wel tekenreeksen genoemd.) Stringwaarden kunnen net zo worden gebruikt als integerwaarden of floatwaarden. Je kunt strings dus ook opslaan in variabelen. In programmeercode worden stringwaarden voorafgegaan en gevolgd door een enkel aanhalingsteken (`'`). Probeer deze code eens in de interactieve shell te typen:

```
>>> ham = 'hallo'
```

Door de enkele aanhalingstekens weet Python waar de string begint en eindigt en dat het hier om tekst gaat. Ze horen niet bij de waarde van de string. Als je nu `ham` in de interactieve shell typt, zie je de inhoud van de variabele `ham` (de string `'hallo'`.) Weet je nog? Python evalueert een variabele door de waarde te retourneren die in de variabele zit. In dit geval is dat de string `'hallo'`:

```
>>> ham = 'hallo'
>>> ham
'hallo'
```

Strings kunnen alle tekens bevatten die je op het toetsenbord ziet. Dit zijn allemaal voorbeelden van strings:

```
'hallo'
'Hee daar!'
'EENDJES'
'7 appels, 14 sinaasappels, 3 citroenen'
```

```
'Als achter vliegen vliegen vliegen, vliegen vliegen vliegen achterna.'
'Dikkertje Dap zat op de trap...'
'O*&#wY%*&OCfsdYO*&gfC%YO*&%3yc8r2'
```

Stringwaarden kunnen worden gecombineerd met operatoren om expressies te maken, net zoals dat kan met integers en floats.

Stringconcatenatie

Probeer twee strings eens met elkaar te combineren met de operator +. Dit heet **strings concateneren** (gewoon een duur woord voor 'aan elkaar plakken'). Typ `'Hallo' + 'wereld!'` in de interactieve shell:

```
>>> 'Hallo' + 'wereld!'
'Hallowereld!'
```

De expressie retourneert één stringwaarde, namelijk `'Hallowereld!'`. Als je de woorden uit elkaar wilt houden, moet je een spatie toevoegen aan het eind van de string `'Hallo'`, vóór het enkele aanhalingsteken, als volgt:

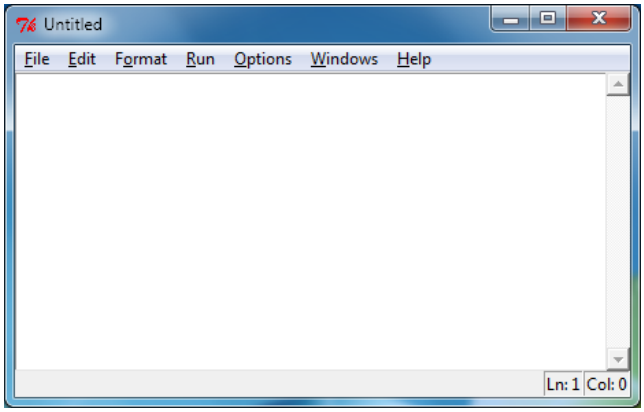
```
>>> 'Hallo ' + 'wereld!'
'Hallo wereld!'
```

De operator + heeft een andere werking op stringwaarden dan op integerwaarden omdat die waarden verschillende datatypen hebben. (Datatypen worden ook wel gegevenstypen genoemd.) Alle waarden hebben een datatype. Het datatype van de waarde `'Hallo'` is een string. Het datatype van de waarde `5` is een integer. Het datatype laat Python weten wat operatoren moeten doen tijdens het evalueren van expressies. De operator + plakt twee stringwaarden aan elkaar vast, maar zal twee integer- of floatwaarden bij elkaar optellen.

Programma's schrijven in de bestandseditor van IDLE

Tot nu toe heb je instructies in de interactieve shell van IDLE getypt, maar steeds één voor één. Als je een echt programma schrijft, schrijf je echter een heleboel instructies en worden ze allemaal achter elkaar uitgevoerd. We gaan nu ons eerste programma schrijven!

IDLE heeft ook nog een ander onderdeel, de **bestandseditor**. Klik op het menu **File** (Bestand) boven aan het shellvenster van Python. Selecteer vervolgens **New File** (Nieuw bestand). Er verschijnt een leeg, nieuw venster waarin je de code van je programma kunt gaan typen.



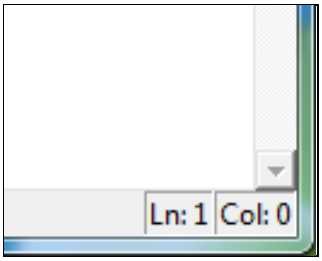
Afbeelding 3-1: Het venster van de bestandseditor.

De twee vensters lijken op elkaar maar er is een verschil: Het **venster van de interactieve shell** heeft het >>>-teken. Het **venster van de bestandseditor** heeft dat niet.

Hallo wereld!

Het is al jarenlang een traditie voor programmeurs om hun eerste programma de boodschap "Hallo wereld!" op het scherm te laten weergeven. Jij gaat nu jouw eigen Hallo wereld-programma maken.

Als je je programma gaat schrijven, typ je niet de nummers aan de linkerkant van de code. Zij dienen alleen als verwijzing naar de regels code. In de rechterbenedenhoek van het bestandseditorvenster zie je waar de knipperende cursor zich bevindt. In Afbeelding 3-2 zie je in de hoek een voorbeeld van waar de cursor staat.



Afbeelding 3-2: De rechteronderhoek van het bestandseditorvenster vertelt je op welke regel de cursor staat.

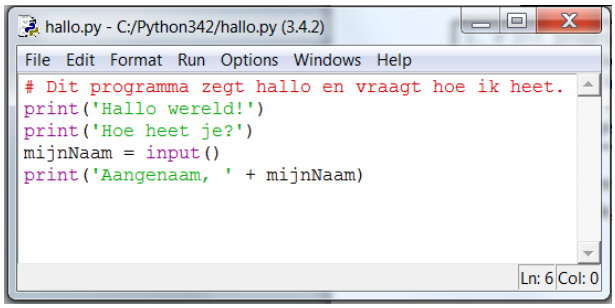
hallo.py

Typ de volgende tekst in het nieuwe venster van de bestandseditor. Dit is de **broncode** van het programma. Deze broncode bevat de instructies die Python gaat uitvoeren als het programma straks wordt gestart. (Niet vergeten: typ niet de regelnummers!)

> **BELANGRIJKE OPMERKING!** De programma's in dit boek kunnen alleen worden uitgevoerd in Python 3, niet in Python 2. Zodra het IDLE-venster start, zie je bovenaan zoiets als 'Python 3.4.0'. Als je Python 2 al had geïnstalleerd, kun je daarna ook Python 3 installeren op dezelfde computer. Om Python 3 te downloaden, ga je naar https://python.org/download/.

```
1. # Dit programma zegt hallo en vraagt hoe ik heet.
2. print('Hallo wereld!')
3. print('Hoe heet je?')
4. mijnNaam = input()
5. print('Aangenaam, ' + mijnNaam)
```

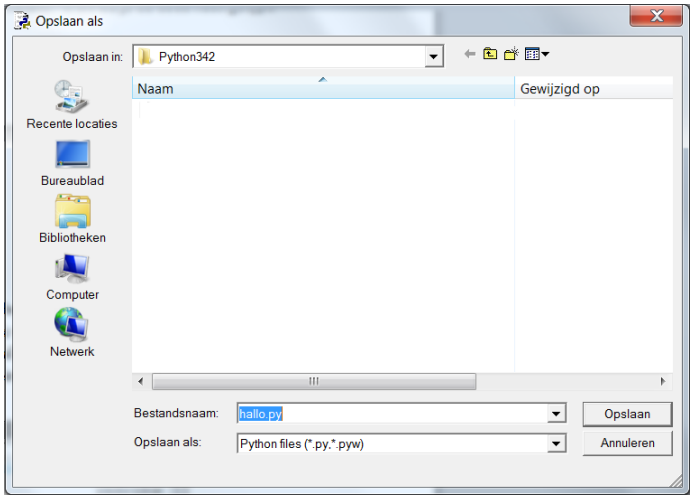
IDLE toont de verschillende soorten instructies in verschillende kleuren. Als je klaar bent met het typen van de code, zal het venster er als volgt uitzien:



Afbeelding 3-3: Zo ziet het bestandseditorvenster eruit als je de code hebt getypt.

Je programma opslaan

Zodra je je broncode hebt ingevoerd, moet je het bestand opslaan door op **File** (Bestand) te klikken boven aan het bestandseditorvenster. Klik vervolgens op **Save As** (Opslaan als). In Afbeelding 3-4 zie je het venster Opslaan als. Typ *hallo.py* in het tekstvak **Bestandsnaam** en klik op **Opslaan**. Je kunt ook gelijktijdig op Ctrl+S drukken op het toetsenbord om op een snelle manier op te slaan.



Afbeelding 3-4: Het programma opslaan.

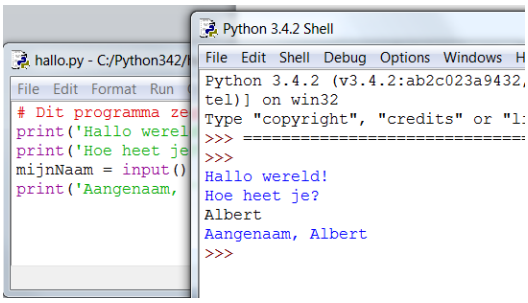
Je moet je programma's steeds even opslaan terwijl je ze aan het typen bent. Als de computer een keertje crasht of jij per ongeluk IDLE sluit, ben je dan niet al je werk kwijt.

De programma's die je hebt opgeslagen, weer openen

Als je een programma dat je hebt opgeslagen, weer wilt laden, klik je op **File ► Open** (Bestand > Openen). Kies *hallo.py* in het venster dat verschijnt en klik op de knop **Openen**. Je eerder opgeslagen programma *hallo.py* wordt geopend in het bestandseditorvenster.

Nu is het tijd om het programma uit te voeren. Klik op **Run ► Run Module** (Uitvoeren > Module uitvoeren) of druk gewoon op F5 in het bestandseditorvenster. Je programma wordt uitgevoerd in het venster van de interactieve shell.

Typ je naam wanneer je programma vraagt hoe je heet. Dit ziet eruit zoals in Afbeelding 3-5:



Afbeelding 3-5: De interactieve shell na het starten van hallo.py.

Zodra je je naam typt en op ENTER drukt, begroet het programma je met je naam. Gefeliciteerd! Je hebt je eerste programma geschreven en je mag jezelf nu een 'computerprogrammeur' noemen. Druk nogmaals op F5 en typ een andere naam.

Als je een foutmelding kreeg, moet je je code vergelijken met de code van dit boek, met behulp van de online diff tool op http://invpy.com/diff/nl/hallo.

Als je een foutmelding kreeg zoals hieronder:

```
Hallo wereld!
Hoe heet je?
Albert
Traceback (most recent call last):
  File "C:/Python26/test1.py", line 4, in <module>
    mijnNaam = input()
  File "<string>", line 1, in <module>
NameError: name 'Albert' is not defined
```

...gebruik je waarschijnlijk Python 2 in plaats van Python 3. Installeer versie 3 van Python vanaf http://www.python.org. Voer het programma opnieuw uit met Python 3.

Hoe het programma 'Hallo wereld' werkt

Elke regel code is een instructie die door Python wordt geïnterpreteerd (vertaald naar de computer). Deze instructies vormen samen het programma. De instructies in een computerprogramma zijn als de stappen in een recept voor het bakken van een taart. Elke instructie wordt op volgorde uitgevoerd, van boven in het programma naar beneden.

Als Python bezig is met een bepaalde stap in het recept van het programma, zeg je dat hij die instructie **verwerkt**. Wanneer het programma start, gaat de eerste regel verwerkt worden. Daarna wordt de volgende instructie verwerkt.

Laten we elke regel code eens bekijken om te zien wat er precies gebeurt. We beginnen met regel 1.

Commentaar

```
1. # Dit programma zegt hallo en vraagt hoe ik heet.
```

Deze regel is **commentaar**. Tekst achter het teken # (ook wel het **hekje** genoemd) is commentaar. Commentaar is niet bedoeld voor Python, maar voor jou als programmeur. Python negeert commentaar. Commentaar vertelt aan jou wat de code doet of aan een andere programmeur die de

broncode leest. Om het makkelijker te maken om de broncode te lezen, wordt commentaar in dit boek weergegeven in lichtgrijs.

Programmeurs zetten meestal in het commentaar bovenaan hun code hoe het programma heet. IDLE geeft commentaar weer in rode tekst zodat het beter opvalt.

Functies

Een **functie** is een soort miniprogrammaatje in jouw programma. Een functie bevat instructies die worden uitgevoerd zodra de functie wordt aangeroepen. Je kunt zelf functies schrijven, maar Python bevat ook al een aantal ingebouwde functies. Twee functies, `print()` en `input()`, worden nu uitgelegd. Wat zo fijn is bij die ingebouwde functies, is dat je alleen maar hoeft te weten wat de functie doet, maar niet hoe hij dat doet. Laat dat maar aan de functie over.

Een **functieaanroep** is een instructie aan Python om de code die in de functie staat, uit te voeren. Jouw programma roept bijvoorbeeld de functie `print()` aan om een string op het scherm weer te geven. De functie `print()` ziet de string die jij typt tussen de haakjes achter de naam van de functie, als invoer en geeft die tekst vervolgens weer op het scherm. Om `Hallo wereld!` op het scherm weer te geven, typ je de functienaam `print`, gevolgd door een haakje openen, dan de string `'Hallo wereld!'` en ten slotte een haakje sluiten.

De functie `print()`

```
2. print('Hallo wereld!')
3. print('Hoe heet je?')
```

Regels 2 en 3 zijn aanroepen naar de functie `print()`. Een waarde tussen de haakjes in een functieaanroep heet een **argument**. Het argument van de functieaanroep `print()` op regel 2 is `'Hallo wereld!'`. Het argument van de functieaanroep `print()` op regel 3 is `'Hoe heet je?'`. Dit heet het **doorgeven** van het argument aan de functie `print()`.

Als we het in dit boek over functienamen hebben, tonen we altijd de haakjes erbij. Hierdoor wordt het duidelijk dat het boek dan een functie bedoelt met de naam `print()`, en niet een variabele die toevallig ook `print` heet. Dit is een beetje zoals de aanhalingstekens om het getal `'42'` die Python laten weten dat je het hebt over de string `'42'` en niet over de integerwaarde `42`.

De functie `input()`

```
4. mijnNaam = input()
```

Deze regel bevat een toewijzingsstatement met een variabele (`mijnNaam`) en een functieaanroep `input()`. Wanneer `input()` wordt aangeroepen, wacht het programma tot de gebruiker tekst heeft ingevoerd. De tekststring die de gebruiker invoert, wordt de waarde die wordt geretourneerd door de functie. Functieaanroepen kunnen in expressies worden gebruikt op elke plek waar een waarde kan worden gebruikt.

De waarde die de functie oplevert (retourneert), wordt de **retourwaarde** genoemd. ('De waarde die een functie retourneert' betekent eigenlijk hetzelfde als 'de waarde die de functie oplevert of teruggeeft'.) In dit geval is de retourwaarde van de functie `input()` de string die de gebruiker heeft getypt: hun naam. Als de gebruiker 'Albert' heeft getypt, retourneert de functie `input()` de string `'Albert'`. Deze evaluatie ziet er als volgt uit:

```
mijnNaam = input()
           ▼
mijnNaam = 'Albert'
```

Zo wordt de stringwaarde 'Albert' opgeslagen in de variabele `mijnNaam`.

Expressies gebruiken in functieaanroepen

```
5. print('Aangenaam, ' + mijnNaam)
```

De laatste regel is weer een functieaanroep voor `print()`. De expressie `'Aangenaam, ' + mijnNaam` wordt doorgegeven aan `print()`. Maar, zoals we al zagen, argumenten zijn altijd een enkele waarde. Daarom moet Python deze expressie eerst evalueren en daarna de enkele waarde doorgeven als argument. Als `'Albert'` is opgeslagen in `mijnNaam`, ziet de evaluatie er als volgt uit:

```
print('Aangenaam, ' + mijnNaam)
                       ▼
print('Aangenaam, ' + 'Albert')
                       ▼
print('Aangenaam, Albert')
```

En zo begroet het programma de gebruiker met zijn naam.

Het programma beëindigen

Zodra het programma de laatste regel heeft uitgevoerd, wordt het **beëindigd** of **afgesloten**. Dit betekent dat het programma ophoudt met de uitvoering. Python vergeet direct alles over de variabelen, zoals de string die in `mijnNaam` was opgeslagen. Als je het programma opnieuw start, met een andere naam, zal het programma denken dat dat je naam is.

```
Hallo wereld!
Hoe heet je?
Carolien
Aangenaam, Carolien
```

Onthoud dat de computer precies doet wat jij programmeert. Meer kan hij niet. Computers zijn dom. Het kan de computer niet schelen of je jouw naam typt, die van iemand anders of gewoon iets stoms. Typ wat je wilt. De computer verwerkt alles op dezelfde manier:

```
Hallo wereld!
Hoe heet je?
poep
Aangenaam, poep
```

Namen van variabelen

Door variabelen een beschrijvende naam te geven, wordt het gemakkelijker om te begrijpen wat een programma doet. Stel dat je ging verhuizen en op elke doos die je inpakte, schreef je 'Spullen'. Dat zou niet handig zijn!

In plaats van `mijnNaam` had je ook `sinterKlaas` of `nAaM` kunnen gebruiken. Dat kan Python niets schelen. Python voert het programma op dezelfde manier uit.

Namen van variabelen zijn hoofdlettergevoelig. **Hoofdlettergevoeligheid** betekent dat dezelfde variabelenaam maar met een verschillende combinatie van hoofdletters en kleine letters, wordt gezien als een andere variabelenaam. Dus `ham`, `HAM`, `Ham` en `hAM` zijn vier verschillende variabelen volgens Python. En ze bevatten allemaal hun eigen waarden.

Het is geen goed idee om variabelenamen te gebruiken waarin alleen de hoofdletters en kleine letters van elkaar verschillen. Als je je eerste naam in de variabele `naam` had opgeslagen en je laatste naam in de variabele `NAAM`, weet je een paar weken later echt niet meer welke variabele welke waarde ook al weer had. Was `naam` nou de eerste naam en `NAAM` de laatste naam, of andersom?

Namen van variabelen worden meestal met kleine letters geschreven. Als er meer dan één woord in de naam voorkomt, schrijf je elk woord na het eerste woord, met een hoofdletter. Hierdoor wordt je code beter leesbaar. De variabelenaam `watIkVanmorgenAtVoorMijnOntbijt` is veel makkelijker te lezen dan `watikvanmorgenatvoormijnontbijt`. Dit is een **conventie** (een optionele, maar vaakgebruikte manier om iets te doen) in het programmeren met Python.

De voorbeelden in de interactieve shell in dit boek gebruiken soms maffe variabelenamen zoals `ham`, `kaas` en `cola`. Dat komt omdat de variabelenamen in deze voorbeelden er niet toe doen. De programma's in dit boek gebruiken echter allemaal beschrijvende namen. Jouw programma's zouden ook beschrijvende variabelenamen moeten bevatten.

Samenvatting

Nu je het een en ander weet over strings en functies, kun je beginnen met het schrijven van programma's die een interactie hebben met de gebruiker. Strings zijn belangrijk omdat de gebruiker en de computer voornamelijk via tekst met elkaar communiceren. Je kunt de gebruiker tekst laten invoeren via het toetsenbord met de functie `input()` en je kunt de computer tekst laten weergeven op het scherm met de functie `print()`.

Strings zijn gewoon waarden van een nieuw datatype. Alle waarden hebben een datatype en er zijn veel verschillende datatypen in Python. Met de operator + kun je strings concateneren (aan elkaar plakken).

In variabelen onthoud je waarden zodat je die later kunt gebruiken in het programma. Functies worden gebruikt om een bepaalde ingewikkelde instructie uit te voeren als onderdeel van je programma. Python heeft veel ingebouwde functies waar je over gaat leren in dit boek. Variabelen en functieaanroepen kunnen in expressies worden gebruikt, op elke plek waar een waarde wordt gebruikt.

Het uitvoeren door Python van een instructie in je programma, heet de verwerking van die instructie. In het volgende hoofdstuk leer je meer over hoe je de verwerking van instructies op andere manieren kunt laten plaatsvinden dan alleen maar van boven naar beneden in het programma. Zodra je dat hebt geleerd, ben je klaar om games te gaan maken.

Hoofdstuk 4

RAAD HET GETAL

In dit hoofdstuk behandelen we:

- `import`-statements
- Modules
- Argumenten
- `while`-statements
- Voorwaarden
- Blokken
- Boole-waarden
- Vergelijkingsoperatoren
- Het verschil tussen = en ==
- `if`-statements
- Het sleutelwoord `break`
- De functies `str()` en `int()`
- De functie `random.randint()`

Het spelletje 'Raad het getal'

In dit hoofdstuk ga je een spelletje maken dat 'Raad het getal' heet. De computer zal een willekeurig getal tussen 1 en 20 in gedachten houden en jij moet het raden. Bij elke beurt vertelt de computer of je te hoog of te laag hebt geraden. Als je het getal raadt binnen 6 beurten, heb jij gewonnen.

Dit is een goed spel om te coderen, omdat het gebruikmaakt van willekeurige getallen, lussen en invoer van de gebruiker, en dat allemaal in een tamelijk kort programma. Je leert ook hoe je waarden kunt omzetten in een ander datatype (en waarom je dit soms moet doen). Omdat dit programma een spelletje is, noemen we de gebruiker nu de **speler**, maar je kunt natuurlijk ook gewoon gebruiker zeggen.

Voorbeeld van uitvoering van 'Raad het getal'

Hier zie je hoe het programma eruitziet voor de speler wanneer het spel wordt gestart. De tekst die de speler intypt, wordt **vet** weergegeven.

```
Hoi! Hoe heet je?
Albert
OK Albert, ik denk aan een getal tussen 1 en 20.
Raad eens.
10
Je hebt te hoog geraden.
Raad eens.
2
Je hebt te laag geraden.
Raad eens.
4
```

```
Gaaf, Albert! Je hebt het getal geraden in 3 beurten!
```

Typ de code van dit programma precies zoals het hieronder staat afgebeeld en sla het op door te klikken op **File ► Save As** (Bestand > Opslaan als). Geef het bestand de naam *raden.py*. Druk dan op **F5** om het programma te starten. Maak je geen zorgen als je deze code nog niet begrijpt, ik ga het stap voor stap uitleggen.

Broncode van Raad het getal

Hier zie je de broncode van het spel 'Raad het getal'. Wanneer je deze code in de bestandseditor typt, let dan goed op dat je precies hetzelfde aantal spaties typt aan het begin van de regels als in het voorbeeld. Sommige regels springen vier of acht spaties in.

> **BELANGRIJKE OPMERKING!** De programma's in dit boek kunnen alleen worden uitgevoerd in Python 3, niet in Python 2. Zodra het IDLE-venster start, zie je bovenaan zoiets als 'Python 3.4.0'. Als je Python 2 al had geïnstalleerd, kun je daarna ook Python 3 installeren op dezelfde computer. Om Python 3 te downloaden, ga je naar https://www.python.org/download/.

Als je een foutmelding krijgt nadat je deze code hebt ingetypt, vergelijk jouw code dan met de code van het boek met behulp van de online diff tool op http://invpy.com/diff/nl/raden.

raden.py

```
 1. # Dit is een spelletje om een getal te raden.
 2. import random
 3.
 4. aantalBeurten = 0
 5.
 6. print('Hallo! Hoe heet je?')
 7. mijnNaam = input()
 8.
 9. getal = random.randint(1, 20)
10. print('OK ' + mijnNaam + ', ik denk aan een getal tussen 1 en 20.')
11.
12. while aantalBeurten < 6:
13.     print('Raad eens.') # Er staan vier spaties voor print.
14.     poging = input()
15.     poging = int(poging)
16.
17.     aantalBeurten = aantalBeurten + 1
18.
19.     if poging < getal:
20.         print('Je hebt te laag geraden.') # Er staan acht spaties voor print.
21.
22.     if poging > getal:
23.         print('Je hebt te hoog geraden.')
24.
25.     if poging == getal:
26.         break
27.
28. if poging == getal:
29.     aantalBeurten = str(aantalBeurten)
30.     print('Gaaf, ' + mijnNaam + '! Je hebt het getal geraden in ' +
aantalBeurten + ' beurten!')
```

```
31.
32. if poging != getal:
33.     getal = str(getal)
34.     print('Jammer. Het getal waar ik aan dacht was ' + getal + '.')
```

We gaan nu elke regel code bekijken zodat we begrijpen hoe dit programma werkt.

De `import`-statement

```
1. # Dit is een spelletje om een getal te raden.
2. import random
```

De eerste regel is commentaar. Je weet al dat Python alles dat achter het teken # komt, gewoon negeert. Commentaar is alleen voor ons, zodat we weten wat dit programma gaat doen.

De tweede regel is een **import-statement**. Je herinnert je misschien nog dat statements instructies zijn die een actie uitvoeren maar geen waarde retourneren zoals expressies dat doen. Je bent statements al tegengekomen: toewijzingsstatements slaan een waarde op in een variabele (maar de statement zelf geeft geen resultaat).

Hoewel Python heel veel ingebouwde functies bevat, zijn er ook nog functies die in aparte programma's staan, de zogenaamde modules. **Modules** zijn Python-programma's die extra functies bevatten. Als je deze functies wilt gebruiken, moet je ze in je programma importeren met behulp van de `import`-statement.

De `import`-statement bestaat uit het sleutelwoord `import` gevolgd door de modulenaam. Regel 2 importeert een module met de naam `random`. De module `random` bevat een aantal functies die te maken hebben met willekeurige ('random' in het Engels) getallen. Een van deze functies gaat een willekeurig getal verstrekken dat de speler moet raden.

```
4. aantalBeurten = 0
```

Deze regel maakt een nieuwe variabele met de naam `aantalBeurten`. In deze variabele bewaar je hoe vaak de speler al aan de beurt is geweest. Aangezien de speler aan het begin van het spel nog geen enkele keer heeft geraden, geven we deze variabele om te beginnen de waarde `0`.

```
6. print('Hallo! Hoe heet je?')
7. mijnNaam = input()
```

De regels 6 en 7 zijn hetzelfde als de regels in het programma 'Hallo wereld' dat je in Hoofdstuk 3 al hebt gezien. Programmeurs gebruiken vaak code uit andere programma's opnieuw om zichzelf werk te besparen.

Regel 6 is een aanroep naar de functie `print()`. Zoals je weet, is een functie een soort miniprogrammaatje in jouw programma. Zodra in jouw programma een functie wordt aangeroepen, wordt dit miniprogrammaatje uitgevoerd. De functie `print()` toont het stringargument dat je aan de functie hebt doorgegeven, op het scherm.

Regel 7 laat de gebruiker hun naam intypen en bewaart deze waarde in de variabele `mijnNaam`. (Vergeet niet, die string is misschien helemaal niet echt hoe de speler heet. Het is gewoon wat de speler heeft getypt. Computers zijn dom en doen gewoon wat het programma zegt.)

De functie `random.randint()`

```
9. getal = random.randint(1, 20)
```

Regel 9 doet twee dingen: hij roept een nieuwe functie aan met de naam `randint()` én hij bewaart de waarde die de functie retourneert, in de variabele `getal`. Je weet misschien nog wel dat functieaanroepen onderdeel mogen uitmaken van een expressie (en dat komt omdat ze een waarde retourneren).

Omdat de functie `randint()` wordt geleverd door de module `random`, moet je de functieaanroep even laten voorafgaan door `random.` (vergeet de punt niet) zodat Python weet dat hij de functie `randint()` in de module `random` kan vinden.

De functie `randint()` retourneert een willekeurige integer die ligt tussen de 2 integer-argumenten die je doorgeeft (inclusief die twee getallen). Regel 9 geeft `1` en `20` door tussen de haakjes achter de functienaam (gescheiden door een komma). De willekeurige integer die door de functie `randint()` wordt geretourneerd, wordt bewaard in de variabele met de naam `getal`; dit is het geheime getal dat de speler moet raden.

Ga eventjes terug naar de interactieve shell en typ `import random` om de module `random` te importeren. Typ dan `random.randint(1, 20)` om te zien welke waarde de functie retourneert. Er wordt een getal geretourneerd dat tussen `1` en `20` ligt. Herhaal de code en de functieaanroep retourneert waarschijnlijk een ander getal. Elke keer retourneert de functie `randint()` een willekeurig getal, net zoals je bij het gooien van een dobbelsteen elke keer een ander getal krijgt.

```
>>> import random
>>> random.randint(1, 20)
12
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
3
>>> random.randint(1, 20)
18
>>> random.randint(1, 20)
7
```

Gebruik de functie `randint()` wanneer je een element van willekeurigheid aan je games wilt toevoegen. En in de meeste games zul je daarvan gebruik willen maken. Denk maar eens aan hoeveel gezelschapsspellen er wel niet zijn met dobbelstenen.

Je kunt ook verschillende bereiken van getallen uitproberen, door de argumenten te veranderen. Typ bijvoorbeeld `random.randint(1, 4)` om alleen integers te krijgen tussen `1` en `4` (inclusief `1` en `4`). Of probeer `random.randint(1000, 2000)` om integers te krijgen tussen `1000` en `2000`. Typ bijvoorbeeld het volgende in de interactieve shell. Het resultaat dat jij krijgt als je de functie `random.randint()` aanroept, zal waarschijnlijk anders zijn dan het hier getoonde resultaat (het is tenslotte willekeurig).

```
>>> random.randint(1, 4)
3
>>> random.randint(1000, 2000)
1294
```

Je kunt de code van het spel een beetje wijzigen om het spel anders te laten verlopen. Probeer de regels 9 en 10 te wijzigen van dit:

```
 9. getal = random.randint(1, 20)
10. print('OK ' + mijnNaam + ', ik denk aan een getal tussen 1 en 20.')
```

…in dit:

```
 9. getal = random.randint(1, 100)
10. print('OK ' + mijnNaam + ', ik denk aan een getal tussen 1 en 100.')
```

Nu neemt de computer een integer in gedachten tussen 1 en 100 in plaats van 1 en 20. Als je regel 9 verandert, wordt het bereik van het willekeurige getal gewijzigd, maar vergeet dan niet ook regel 10 te veranderen, anders weet de speler niet dat er nu een groter bereik is.

De speler begroeten

```
10. print('OK ' + mijnNaam + ', ik denk aan een getal tussen 1 en 20.')
```

Op regel 10 begroet de functie `print()` de speler door zijn naam te noemen en wordt uitgelegd dat de computer een willekeurig getal in gedachten neemt.

Het lijkt misschien of er meer dan één stringargument in regel 10 staat, maar als je nauwkeurig kijkt, zie je dat het plusteken de drie strings aan elkaar plakt zodat er nog maar één string overblijft. En dat is het stringargument dat wordt doorgegeven aan de functie `print()`. Als je goed kijkt, zie je dat de komma binnen de aanhalingstekens staat en dus deel uitmaakt van de string zelf.

Lussen

```
12. while aantalBeurten < 6:
```

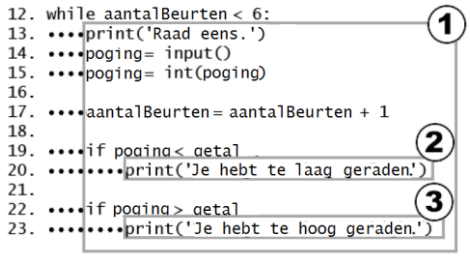
Regel 12 is een `while`-statement, die het begin van een `while`-lus aangeeft. Met **lussen** kun je dezelfde code steeds overnieuw uitvoeren. Maar voor we het over lussen gaan hebben, moet je eerst nog iets weten over een paar andere begrippen. Die begrippen zijn blokken, Boole-waarden, vergelijkingsoperatoren en de `while`-statement.

Blokken

Een aantal regels code die bij elkaar horen, worden gegroepeerd in een blok. Een **blok** code heeft dezelfde minimaal vereiste inspringing. Je kunt zien waar een blok begint en eindigt door naar het aantal spaties aan het begin van de regel te kijken. Dat is de **inspringing** van de regel.

Een blok begint waar de inspringing groter wordt (dat gaat meestal met vier spaties tegelijk). Alle volgende regels die dezelfde inspringing hebben, horen bij het blok. Het blok eindigt zodra er een regel is met hetzelfde aantal spaties als er stond voordat het blok begon. Dit betekent dat blokken ook kunnen voorkomen in andere blokken.

Afbeelding 4-1 is een diagram met code waarin de blokken zijn gemarkeerd en genummerd. De spaties worden weergegeven met zwarte vierkantjes, zodat je ze gemakkelijker kunt tellen.



Afbeelding 4-1: Blokken en hun inspringing. De zwarte vierkantjes duiden spaties aan.

In Afbeelding 4-1 heeft regel 12 geen inspringing en staat deze regel dus niet in een blok. Regel 13 springt met vier spaties in. Aangezien deze inspringing groter is dan de vorige, is er een nieuw blok begonnen. Dit blok heeft het label (1) in Afbeelding 4-1. Dit blok loopt door tot er een regel met nul spaties komt (de oorspronkelijke inspringing voor het blok begon). Lege regels worden genegeerd.

Regel 20 springt in met acht spaties. Acht spaties is meer dan vier spaties, dus hier begint weer een nieuw blok. Dit blok heeft het label (2) in Afbeelding 4-1. Dit blok staat in een ander blok.

Regel 22 heeft maar vier spaties. Omdat de inspringing kleiner is geworden, weet je dat het vorige blok is beëindigd. Regel 20 was de enige regel in dat blok. Regel 22 staat in hetzelfde blok als de andere regels met vier spaties.

Regel 23 verhoogt de inspringing tot acht spaties, dus hier is weer een nieuw blok begonnen. Dit blok heeft het label (3) in Afbeelding 4-1.

Samenvattend: regel 12 staat niet in een blok. De regels 13 tot 23 staan allemaal in één blok (gemarkeerd als blok 1). Regel 20 staat in een blok binnen een ander blok (gemarkeerd als blok 2). En regel 23 is de enige regel in een ander blok binnen een blok (gemarkeerd als blok 3).

Het datatype Boole

Het datatype Boole heeft slechts twee waarden: 'Waar' of 'Niet waar', oftewel in het Engels: `True` of `False`. Deze waarden moeten worden getypt met een hoofdletter 'T' en een hoofdletter 'F'. De rest van de naam van de waarde moet kleine letters hebben. Je kunt Boole-waarden gebruiken samen met vergelijkingsoperatoren om voorwaarden te testen. (Wat voorwaarden zijn, komt straks.)

De datatypen die we tot nu toe hebben besproken, zijn integer, float, string en nu dus Boole.

Vergelijkingsoperatoren

Regel 12 bevat een `while`-statement:

```
12. while aantalBeurten < 6:
```

De expressie na het sleutelwoord `while` (het stukje tekst `aantalBeurten < 6`) bevat twee waarden (de waarde in de variabele `aantalBeurten` en de integerwaarde `6`). Ze worden aan elkaar gekoppeld door een operator (het teken <, het kleiner-dan-teken). Het teken < wordt een **vergelijkingsoperator** genoemd.

Vergelijkingsoperatoren vergelijken twee waarden met elkaar en retourneren een van de twee Boole-waarden: `True` of `False`. In Tabel 4-1 zie je een lijst met alle vergelijkingsoperatoren.

Tabel 4-1: Vergelijkingsoperatoren.

Operatorteken	Operatornaam
<	Kleiner dan
>	Groter dan
<=	Kleiner dan of gelijk aan
>=	Groter dan of gelijk aan
==	Gelijk aan
!=	Niet gelijk aan

Je hebt al gelezen over de rekenkundige operatoren +, -, * en /. Net als deze operatoren, combineren vergelijkingsoperatoren waarden met elkaar en vormen ze zo expressies. Bijvoorbeeld: `aantalBeurten < 6`.

Voorwaarden

Een **voorwaarde** is een expressie die twee waarden combineert met behulp van een vergelijkingsoperator (zoals < of >), en een Boole-waarde retourneert. Een voorwaarde is dus gewoon een expressie waarvan je kunt zeggen of hij 'waar' is of 'niet waar' (`True` of `False`). Voorwaarden worden gebruikt in `while`-statements (en op nog een paar andere plekken, dat komt later.)

Bijvoorbeeld: de voorwaarde `aantalBeurten < 6` vraagt: "Is de waarde die in `aantalBeurten` zit, kleiner dan het getal `6`?" Als dat zo is, retourneert de voorwaarde de waarde `True` (Waar). Zo niet, is het resultaat `False` (Niet waar).

In het geval van het programma 'Raad het getal' heb je op regel 4 de waarde `0` in de variabele `aantalBeurten` gestopt. Omdat `0` kleiner is dan `6`, is het resultaat van deze voorwaarde de Boole-waarde `True`. De evaluatie verloopt als volgt:

```
aantalBeurten < 6
              ▼
            0 < 6
              ▼
             True
```

Experimenteren met Boole-waarden, vergelijkingsoperatoren en voorwaarden

Typ de volgende expressies maar eens in de interactieve shell om de resulterende Boole-waarden te onderzoeken:

```
>>> 0 < 6
True
>>> 6 < 0
False
>>> 50 < 10
False
>>> 10 < 11
True
>>> 10 < 10
False
```

De voorwaarde `0 < 6` retourneert de Boole-waarde `True` omdat het getal `0` kleiner is dan het getal `6`. Maar omdat `6` niet kleiner is dan `0`, retourneert de voorwaarde `6 < 0` de waarde `False`. `50` is niet kleiner dan `10`, dus `50 < 10` is `False`. `10` is kleiner dan `11`, dus `10 < 11` is `True`.

Merk op dat `10 < 10` resulteert in `False` omdat het getal `10` niet kleiner is dan het getal `10`. Ze zijn even groot. Als Els even lang zou zijn als Rob, zou je ook niet zeggen dat Els kleiner is dan Rob of dat Els langer is dan Rob. Beide statements zouden 'Niet waar' zijn.

Probeer nu deze expressies in de interactieve shell te typen:

```
>>> 10 == 10
True
>>> 10 == 11
False
>>> 11 == 10
False
>>> 10 != 10
False
>>> 10 != 11
True
>>> 'Hallo' == 'Hallo'
True
>>> 'Hallo' == 'Tot ziens'
False
>>> 'Hallo' == 'HALLO'
False
>>> 'Tot ziens' != 'Hallo'
True
```

Let op dat je de toewijzingsoperator (=) en de vergelijkingsoperator van gelijkheid (==) niet door elkaar haalt. Het teken = wordt in Python gebruikt in toewijzingsstatements om een waarde toe te wijzen aan een variabele, terwijl het isgelijkteken (==) wordt gebruikt in expressies om te zien of twee waarden aan elkaar gelijk zijn. (In rekenen en wiskunde op school wordt het teken = wél gebruikt als isgelijk-teken. Dat kan verwarrend zijn.) Het is gemakkelijk om de twee door elkaar te halen.

Een ezelsbruggetje kan zijn dat de vergelijkingsoperator voor gelijkheid (==) twee tekens bevat, net als de vergelijkingsoperator voor 'niet gelijk' ook twee tekens bevat (`!=`).

Let op: een stringwaarde en een integerwaarde kunnen nooit gelijk aan elkaar zijn (omdat ze verschillende datatypen zijn). Probeer bijvoorbeeld maar eens het volgende in de interactieve shell te typen:

```
>>> 42 == 'Hallo'
False
>>> 42 != '42'
True
```

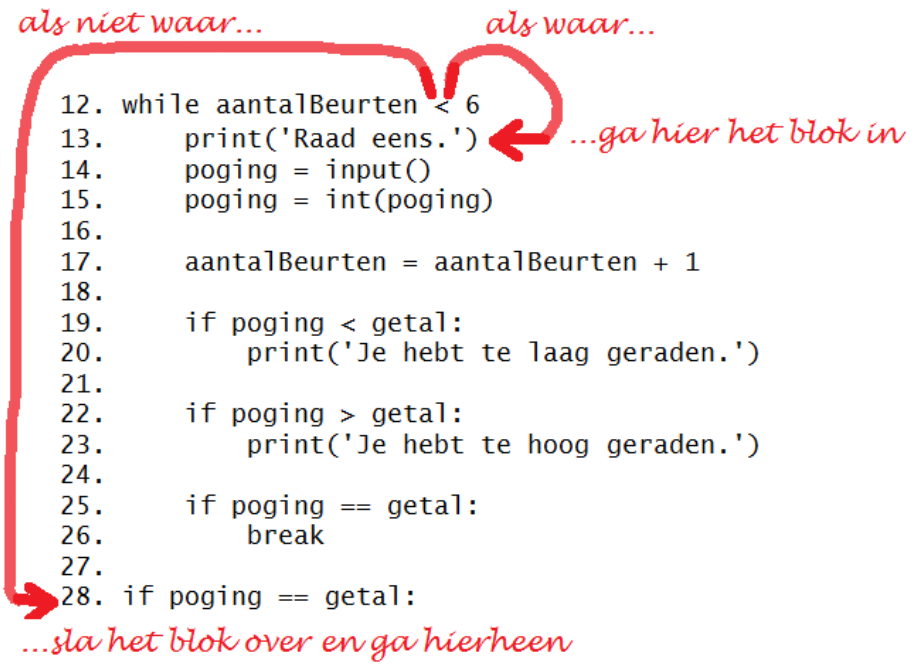
Een lus maken met een `while`-statement

De `while`-statement geeft het begin van een lus aan. Lussen voeren dezelfde code meerdere keren uit. Wanneer de verwerking een `while`-statement bereikt, wordt de voorwaarde achter het sleutelwoord `while` geëvalueerd. Als de voorwaarde de waarde `True` retourneert, gaat de verwerking het while-blok binnen. (In jouw programma begint het while-blok op regel 13.) Als de voorwaarde de waarde `False`

oplevert, slaat de verwerking het while-blok over. (In 'Raad het getal' is regel 28 de eerste regel na het while-blok.)

Een `while`-statement wordt altijd afgesloten met een dubbele punt (:) achter de voorwaarde.

```
12. while aantalBeurten < 6:
```



Afbeelding 4-2: De voorwaarde van de while-lus.

Afbeelding 4-2 laat zien hoe de verwerking verloopt, afhankelijk van de voorwaarde. Als de voorwaarde 'True' retourneert (dit gebeurt de eerste keer, omdat de waarde van `aantalBeurten` 0 is), stapt de verwerking het while-blok binnen op regel 13 en loopt hij door het blok heen. Zodra het programma het eind van het while-blok heeft bereikt, gaat het niet verder naar de volgende regel, maar gaat het programma als in een lus terug naar de regel met de `while`-statement (regel 12) en wordt de voorwaarde opnieuw getest. Net als eerder is de voorwaarde bij de tweede beurt weer `True`, dus de verwerking stapt het while-blok weer binnen. Elke keer dat de verwerking door de lus gaat, noemen we een **iteratie**. (Dat is programmeursjargon voor 'herhaling'.)

En zo werkt de lus. Zo lang als de voorwaarde `True` oplevert, wordt de code binnen het while-blok steeds weer uitgevoerd, totdat de voorwaarde niet meer waar is, dus `False` is. Denk maar aan de `while`-statement op deze manier: "Zo lang (While) deze voorwaarde waar is, blijf je door de code in dit blok gaan".

De code binnen het while-blok zorgt ervoor dat de poging van de speler om het getal te raden wordt geregistreerd en wordt gecontroleerd of het getal van de speler groter dan, kleiner dan of gelijk is aan het geheime getal van de computer. Je kunt de moeilijkheidsgraad van het spel aanpassen door het aantal beurten dat de speler krijgt, te veranderen. Als je de speler maar vier beurten wilt geven, wijzig je deze regel:

```
12. while aantalBeurten < 6:
```

in deze regel:

```
12. while aantalBeurten < 4:
```

Code verderop in het while-blok verhoogt de variabele `aantalBeurten` met `1` bij elke iteratie. Door de voorwaarde in te stellen op `aantalBeurten < 4`, zorg je ervoor dat de code in de lus maar vier keer wordt verwerkt, in plaats van zes keer. Hierdoor wordt het een stuk moeilijker om te winnen. Als je het makkelijker wilt maken, stel je de voorwaarde in op `aantalBeurten < 8` of `aantalBeurten < 10`. Hierdoor loopt de lus nog een paar keer door en worden er meer pogingen van de speler geaccepteerd.

Als je regel 17 helemaal zou verwijderen (`aantalBeurten = aantalBeurten + 1`) dan zou de waarde van `aantalBeurten` nooit oplopen. De voorwaarde van de `while`-lus zou dan altijd `True` blijven! Hierdoor zou de speler tot in alle eeuwigheid kunnen blijven raden. Dat vindt de speler misschien leuk, maar voor een programmeur zou dit gelden als een 'bug'!

De speler raadt

De regels 13 tot 17 vragen de speler te raden wat het geheime getal is en laten de speler zijn poging invoeren. Dat getal wordt opgeslagen in een variabele met de naam `poging`.

```
14.     print('Raad eens.') # Er staan vier spaties voor print.
15.     poging = input()
```

Strings omzetten in integers met de functie `int()`

```
16.     poging = int(poging)
```

In regel 15 roep je een nieuwe functie aan, met de naam `int()`. De functie `int()` 'neemt' één argument en geeft dat argument terug in de vorm van een integerwaarde. Probeer het volgende eens in de interactieve shell te typen:

```
>>> int('12')
12
>>> int(12)
12
>>> int('twaalf')
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
int('twaalf')
ValueError: invalid literal for int() with base 10: 'twaalf'
>>> 3 + int('2')
5
```

De functieaanroep `int('12')` retourneert de integerwaarde `12`. De functieaanroep `int(12)` doet hetzelfde (hoewel het een beetje nutteloos is om een integerwaarde om te zetten in een integerwaarde). Maar ook al kun je een string doorgeven aan de functie `int()`, je kunt niet zo maar elke string doorgeven. Als je `'twaalf'` doorgeeft aan `int()` krijg je een foutmelding. De computer snapt namelijk helemaal niet dat 12 hetzelfde is als twaalf. Daar moet je Nederlands voor kunnen spreken en dat kan de computer niet. De string die je doorgeeft aan `int()`, moet wel bestaan uit cijfertekens, anders begrijpt de computer je niet.

De regel `3 + int('2')` toont een expressie die de retourwaarde van int() gebruikt als onderdeel van een expressie. De expressie retourneert de integerwaarde `5`. Kijk maar:

```
3 + int('2')
  ▼
3 + 2
  ▼
  5
```

Vergeet niet dat de functie `input()` altijd een waarde retourneert van het datatype **string**, namelijk de tekst die de speler typte. Als de speler `5` typt, geeft de functie `input()` de stringwaarde `'5'` terug, niet de integerwaarde `5`. Python kan niet de vergelijkingsoperatoren `<` en `>` gebruiken om een string met een integer te vergelijken:

```
>>> 4 < '5'
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    4 < '5'
TypeError: unorderable types: int() < str()
```

Op regel 15 bevatte de variabele `poging` in het begin de stringwaarde van wat de speler had getypt. Regel 16 overschrijft (vervangt) de stringwaarde in `poging` met de integerwaarde die wordt geretourneerd door `int()`. Hierdoor kan de code verderop in het programma kijken of de variabele `poging` groter of kleiner is dan of gelijk is aan het geheime getal in de variabele `getal`. Python kan geen stringwaarde met een integerwaarde vergelijken, ook niet als de stringwaarde er (voor ons mensen) uitziet als een cijfer, zoals `'5'`.

Nog één laatste ding: Als je `int(poging)` aanroept, verander je niet de waarde van de variabele `poging`, alleen het datatype. De code `int(poging)` is een expressie die het datatype van de variabele `poging` omzet in het datatype integer. De variabele `poging` wordt gewijzigd in de toewijzingsstatement `poging = int(poging)`.

De waarde van variabelen verhogen

```
17.     aantalBeurten = aantalBeurten + 1
```

Zodra de speler een keer heeft geraden, moet het aantal beurten worden verhoogd met 1.

Bij de eerste iteratie van de lus heeft `aantalBeurten` de waarde `0`. Python neemt deze waarde en telt er `1` bij op. `0 + 1` resulteert in `1`, en deze waarde wordt opgeslagen als de nieuwe waarde van de variabele `aantalBeurten`. Denk aan regel 17 als: "De variabele `aantalBeurten` moet 1 meer worden dan hij nu is".

Als je de waarde van een integer- of floatwaarde met 1 verhoogt, heet dat **incrementeren**. Als je de waarde van een integer- of floatwaarde met 1 verlaagt, heeft dat **decrementeren**.

`if`-statements

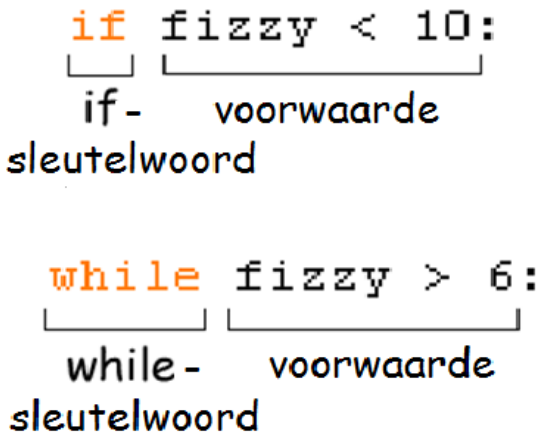
Heeft de speler te laag geraden?

```
19.     if poging < getal:
```

```
20.          print('Je hebt te laag geraden.') # Er staan acht spaties voor print.
```

Regel 19 controleert of het getal dat de speler heeft genoemd, kleiner is dan het geheime getal van de computer. Als dat zo is, gaat de verwerking naar regel 20 en wordt er een bericht weergegeven dat dit zo is. Regel 19 is een `if`-statement. (If betekent Als, zoals je wel zult weten.) Het programma voert de code in het volgende blok uit als de voorwaarde van de `if`-statement de waarde True retourneert. Als de voorwaarde `False` is, wordt de code in het if-blok overgeslagen. Door `if`-statements te gebruiken, kun je ervoor zorgen dat het programma bepaalde code alleen uitvoert in bepaalde gevallen.

De `if`-statement werkt bijna hetzelfde als de `while`-statement. Het enige verschil is dat de verwerking na het eind van het if-blok niet terugspringt naar het `if`-statement. Er wordt gewoon verdergegaan met de volgende regel. Met andere woorden, `if`-statements zijn geen lussen. Zie Afbeelding 4-3 voor een vergelijking van de twee statements.



Afbeelding 4-3: if- en `while`-statements.

Heeft de speler te hoog geraden?

```
22.      if poging > getal:
23.          print('Je hebt te hoog geraden.')
```

Regel 22 kijkt of de speler een getal heeft getypt dat hoger is dan het willekeurige getal van de computer. Als deze voorwaarde `True` is, vertelt de functieaanroep `print()` aan de speler dat ze te hoog hebben geraden.

Een lus vroegtijdig verlaten met de `break`-statement

```
25.      if poging == getal:
26.          break
```

De `if`-statement op regel 25 controleert of het getal van de speler gelijk is aan het willekeurige getal van de computer. Als dat zo, voert het programma de break-statement uit op regel 26.

Een **break-statement** geeft het programma de instructie om onmiddellijk het while-blok te verlaten en verder te gaan naar de eerste regel na het while-blok. (De `break`-statement kijkt niet nog eens of de voorwaarde in de `while`-lus waar is; het programma verlaat de lus meteen.)

De break-statement bestaat alleen uit het sleutelwoord break, zonder voorwaarde of dubbele punt erachter.

Als het getal dat de speler heeft getypt, niet gelijk is aan het willekeurige getal van de computer, bereikt de verwerking het einde van het while-blok. Dit betekent dat de verwerking weer terug gaat naar het begin van het while-blok (als in een lus), om de voorwaarde op regel 12 opnieuw te testen (aantalBeurten < 6). Na de verwerking van de regel aantalBeurten = aantalBeurten + 1 is de nieuwe waarde van de variabele aantalBeurten natuurlijk 1. En omdat 1 < 6 True is, gaat de verwerking de lus weer binnen.

Als de speler steeds te hoog of te laag blijft raden, wordt de waarde van aantalBeurten 2, daarna 3, 4, 5 en vervolgens 6. Zodra aantalBeurten de waarde 6 heeft, wordt de voorwaarde van de while-statement False, omdat 6 niet kleiner is dan 6. En omdat de voorwaarde van de while-statement nu False is, gaat de verwerking verder met de eerste regel na het while-blok.

Als de speler het getal goed heeft geraden, laat de break-statement de verwerking verder springen naar de eerste regel na het while-blok.

De resterende regels van de code worden uitgevoerd wanneer de speler klaar is met raden, ofwel omdat het juiste getal is geraden, of omdat er geen beurten meer over zijn.

Controleren of de speler heeft gewonnen

```
28. if poging == getal:
```

Regel 28 springt niet in, dus het while-blok is beëindigd en dit is de eerste regel na het while-blok. De verwerking heeft het while-blok verlaten ofwel omdat de voorwaarde aan het begin van de while-statement False was (als de speler alle beurten heeft opgebruikt) of omdat de break-statement is verwerkt (als de speler het getal goed heeft geraden). Regel 28 controleert of de speler het goed heeft geraden. Als dat zo is, gaat de verwerking het if-blok binnen, op regel 29.

```
29.     aantalBeurten = str(aantalBeurten)
30.     print('Prima, ' + mijnNaam + '! Je hebt het getal geraden in ' +
aantalBeurten + ' beurten!')
```

Regels 29 en 30 worden alleen verwerkt als de voorwaarde in de if-statement op regel 28 True was (als de speler het getal dus goed heeft geraden).

Regel 29 roept de nieuwe functie str() aan, die een argument omzet in een stringwaarde. Deze code zet de integer in aantalBeurten om in een stringwaarde, en dat is nodig omdat je alleen strings aan elkaar kunt plakken in een bericht aan de gebruiker.

De strings moeten aan elkaar worden geplakt om de speler te vertellen dat ze hebben gewonnen en hoeveel beurten ze nodig hadden. Dat gebeurt op regel 30. Daarom moest regel 29 de variabele aantalBeurten nog even omzetten in een stringwaarde. Anders zou Python een foutmelding hebben gegeven omdat je geen string aan een integer kunt plakken.

Controleren of de speler heeft verloren

```
32. if poging != getal:
```

Regel 32 gebruikt de vergelijkingsoperator `!=` om te controleren of de laatste poging van de speler ongelijk is aan het geheime getal. Als deze voorwaarde `True` is, gaat de verwerking het if-blok op regel 33 binnen.

De regels 33 en 34 staan in het if-blok en worden dus alleen verwerkt als de voorwaarde op regel 32 `True` was.

```
33.     getal = str(getal)
34.     print('Jammer. Het getal waar ik aan dacht was ' + getal + '.')
```

In dit blok vertelt het programma aan de speler wat het geheime getal was dat ze niet hebben geraden. Hiervoor moet je weer strings aan elkaar plakken, maar in de variabele `getal` staat een integerwaarde. Dus regel 33 vervangt de integerwaarde van `getal` met de stringwaarde van `getal` zodat het kan worden samengevoegd met `'Jammer. Het getal waar ik aan dacht was ' + getal + '.'` op regel 34.

Op dit punt heeft de verwerking het einde van de code bereikt en is het programma klaar. Gefeliciteerd! Je hebt zojuist je eerste echte game geprogrammeerd!

Besturingsstroom-statements

In eerdere hoofdstukken begon de verwerking van het programma bij de eerste instructie in het programma en ging de verwerking verder gewoon van boven naar beneden. Elke instructie werd uitgevoerd in de volgorde waarin deze in het programma stond. Maar met de `while`-, `if`- en `break`-statements kun je de verwerking stukjes laten overslaan of opnieuw laten doen. Dit soort statements worden ook wel **besturingsstroom-statements** genoemd omdat ze de 'stroom' van de programmaverwerking veranderen.

Samenvatting

Als iemand je zou vragen, **"Wat is programmeren nou eigenlijk?"**, wat zou je dan zeggen? Programmeren is gewoon het schrijven van code voor een computer, dus het maken van programma's die door een computer kunnen worden uitgevoerd.

"Maar wat is een programma dan precies?" Wanneer je iemand een computerprogramma ziet gebruiken (bijvoorbeeld iemand die ons 'Raad het getal'-spel speelt), zie je bepaalde tekst op het scherm. Het programma beslist welke tekst op het scherm moet worden weergegeven. Dat noemen we de **uitvoer** van het programma. De uitvoer is het resultaat van de instructies in het programma en de tekst die de speler op het toetsenbord heeft getypt (dat laatste is de **invoer** voor het programma). Een **programma** is gewoon een verzameling instructies die reageren op de invoer van een gebruiker.

"Wat voor soort instructies?" Er zijn maar een paar verschillende soorten instructies.

1. **Expressies.** Expressies zijn waarden die met elkaar worden gecombineerd door operatoren. Expressies geven als resultaat altijd één enkele waarde, zoals `2 + 2` de waarde `4` oplevert of `'Hallo' + ' ' + 'wereld'` de tekst `'Hallo wereld'` oplevert. Wanneer expressies naast de sleutelwoorden `if` en `while` staan, noem je ze ook voorwaarden.
2. **Toewijzingsstatements.** Toewijzingsstatements slaan een bepaalde waarde op in een variabele zodat je de waarden later in het programma nog weet.

3. **Besturingsstroom-statements**, zoals `if`**,** `while` en `break`. Besturingsstroom-statements wijzigen hoe het programma 'stroomt', door instructies over te slaan, instructies als in een lus meerdere keren uit te voeren of een lus voortijdig te verlaten. Functieaanroepen kunnen de stroom van de verwerking ook veranderen door naar het begin van een functie te springen die ergens anders staat.
4. **Functies**, zoals `print()` en `input()`**.** Deze functies geven tekst op het scherm weer en onthouden tekst die op het toetsenbord wordt getypt. Dit heet **I/O** (van Input/Output) omdat het betrekking heeft op de invoer en uitvoer van het programma.

En dat is het, gewoon die vier soorten. Maar er zijn natuurlijk wel nog veel meer details die je kunt leren over deze vier soorten instructies. In dit boek leer je over nieuwe datatypen en operatoren, nieuwe besturingsstroom-statements en nog veel meer functies die in Python zitten. Er zijn ook verschillende soorten I/O zoals invoer via de muis en uitvoer van geluid en graphics in plaats van alleen tekst.

De mensen die jouw programma's gebruiken, zijn alleen geïnteresseerd in dat laatste type, I/O. De gebruiker typt iets op het toetsenbord en ziet dan iets op het scherm of hoort iets uit de luidsprekers. Maar om de computer te laten weten welke beelden getoond moeten worden en welke geluiden moeten worden afgespeeld, is een programma nodig, en programma's zijn gewoon een reeks instructies die jij, als programmeur, hebt geschreven.

Hoofdstuk 5

MOPPEN

In dit hoofdstuk behandelen we:

- Het gebruik van het sleutelwoord `end` als argument in de `print`-functie om niet naar een nieuwe regel te gaan
- Escape-tekens
- Het gebruik van enkele en dubbele aanhalingstekens in strings

Meer doen met `print()`

De meeste games in dit boek maken gebruik van eenvoudige tekst voor invoer en uitvoer. De invoer wordt door de gebruiker op het toetsenbord getypt en op die manier in de computer ingevoerd. De uitvoer is de tekst die op het scherm wordt weergegeven. In Python wordt de functie `print()` gebruikt voor het weergeven van de tekstuitvoer op het scherm. Maar we kunnen nog meer leren over hoe strings en `print()` werken in Python.

Het programma in dit hoofdstuk vertelt een paar moppen aan de gebruiker.

Voorbeelduitvoer van Moppen

```
Hoe opent een skelet een deur?
Met zijn sleutelbeen!
Wat doen pizza's als ze ziek zijn?
Ze bellen Dr. Oetker!
Zeg eens: "Koe die in de rede valt".
Koe die in de rede v-BOE!
```

Broncode van Moppen

Typ de volgende broncode in de bestandseditor en sla het bestand op met de naam *moppen.py*.

> **BELANGRIJKE OPMERKING!** De programma's in dit boek kunnen alleen worden uitgevoerd in Python 3, niet in Python 2. Zodra het IDLE-venster start, zie je bovenaan zoiets als 'Python 3.4.2'. Als je Python 2 al had geïnstalleerd, kun je daarna ook Python 3 installeren op dezelfde computer. Om Python 3 te downloaden, ga je naar https://python.org/download/.

Als je een foutmelding krijgt nadat je deze code hebt ingetypt, vergelijk jouw code dan met de code van het boek met behulp van de online diff tool op http://invpy.com/diff/moppen.

moppen.py

```
1. print('Hoe opent een skelet een deur?')
2. input()
3. print('Met zijn sleutelbeen!')
4. print()
5. print('Wat doen pizza\'s als ze ziek zijn?')
```

```
 6. input()
 7. print('Ze bellen Dr. Oetker!')
 8. print()
 9. print('Zeg eens: "Koe die in de rede valt."')
10. input()
11. print('Koe die in de rede v', end='')
12. print('-BOE!')
```

Geen zorgen als je niet alle code begrijpt. Sla het programma gewoon op en voer het uit met F5. Als je bugs in je programma hebt, kun je de online diff tool gebruiken op http://invpy.com/chap5.

Hoe de code werkt

```
1. print('Hoe opent een skelet een deur?')
2. input()
3. print('Met zijn sleutelbeen!')
4. print()
```

De regels 1 t/m 4 bevatten drie functieaanroepen voor `print()`. Je wilt niet dat de speler de clou van de mop meteen al ziet, dus na de eerste `print()` wordt de functie `input()` aangeroepen. De speler kan de eerste regel van de mop dan lezen en daarna op Enter drukken, om vervolgens de clou te lezen.

De gebruiker kan ook een antwoord typen als hij wil, en dan op Enter drukken, maar wat de gebruiker typt, wordt niet in een variabele opgeslagen. Het programma negeert een eventueel antwoord van de speler dus gewoon en gaat naar de volgende regel code.

De laatste aanroep van `print()` heeft geen stringargument. Op deze manier krijgt het programma de opdracht alleen een lege regel weer te geven. Lege regels zijn handig om te zorgen dat de regels van een stukje tekst niet allemaal op een hoopje staan.

Escape-tekens

```
5. print('Wat doen pizza\'s als ze ziek zijn?')
6. input()
7. print('Ze bellen Dr. Oetker!')
8. print()
```

In de eerste `print()`-functie hierboven staat een backslash voor het enkele aanhalingsteken. (Het teken \ noemen we een backslash, en / heet een slash.) Deze backslash laat weten dat het teken dat er direct achter komt, een bijzonder teken is. Met een escape-teken kun je tekens weergeven die normaal een andere betekenis hebben dan binnen Python. In deze aanroep naar `print()` is de backslash het escape-teken.

Dit teken staat vóór het enkele aanhalingsteken omdat Python anders zou denken dat het einde van de string was gekomen. Maar dit aanhalingsteken maakt juist onderdeel uit van de string. Door de backslash weet Python dat het enkele aanhalingsteken deel uitmaakt van de string en niet het einde van de string aangeeft.

Een paar andere escape-tekens

Wat zou je moeten doen als je een backslash zou willen weergeven op het scherm? Deze regel code zou niet goed werken:

```
>>> print('Sla het bestand op in C:\test.')
```

Die `print()`-functie zou dit resultaat opleveren:

```
Sla het bestand op in C:  est.
```

Dat komt omdat de 't' van 'test' gezien wordt als een bijzonder teken omdat het na een backslash kwam. De combinatie \t simuleert het indrukken van de Tab-toets op het toetsenbord. Als je een echte backslash in je tekst wilt in plaats van een bepaalde opdracht in Python te geven, moet je nog een backslash typen.

Probeer deze regel eens:

```
>>> print('Sla het bestand op in C:\\test.')
Sla het bestand op in C:\test.
```

In Tabel 5-1 zie je een lijst met de escape-tekens in Python.

Tabel 5-1: Escape-tekens

Escape-teken	Wat op het scherm wordt weergegeven
\\	Backslash (\)
\'	Enkel aanhalingsteken (')
\"	Dubbel aanhalingsteken (")
\n	Een nieuwe regel
\t	Een tab

Enkele en dubbele aanhalingstekens

Strings hoeven niet per se tussen enkele aanhalingstekens te staan in Python. Je mag ze ook tussen dubbele aanhalingstekens zetten. Deze twee regels geven hetzelfde weer:

```
>>> print('Hallo wereld')
Hallo wereld
>>> print("Hallo wereld")
Hallo wereld
```

Maar je mag aanhalingstekens niet door elkaar gebruiken. Deze regel heeft een foutmelding als resultaat:

```
>>> print('Hallo wereld")
SyntaxError: EOL while scanning single-quoted string
```

Ik gebruik zelf altijd enkele aanhalingstekens. Dan hoef ik de Shift-toets niet ingedrukt te houden als ik een aanhalingsteken typ. Dat is gemakkelijker typen en Python maakt het niets uit.

Net als bij \' moet je een backslash gebruiken als je een dubbel aanhalingsteken wilt weergeven in een string die al tussen dubbele aanhalingstekens staat. Kijk bijvoorbeeld eens naar deze twee regels:

```
>>> print('Ik vroeg of ik Theo\'s auto een week mocht lenen. Hij zei: "Geen
probleem."')
Ik vroeg of ik Theo's auto een week mocht lenen. Hij zei: "Geen probleem."
>>> print("Hij zei: \"Niet te geloven dat je Theo's auto zo maar mocht lenen.\"")
Hij zei: "Niet te geloven dat je Theo's auto zo maar mocht lenen."
```

In de strings die beginnen en eindigen met enkele aanhalingstekens hoef je geen escape-tekens voor dubbele aanhalingstekens te zetten en in de strings die beginnen en eindigen met dubbele aanhalingstekens hoef je geen escape-teken voor enkele aanhalingstekens te zetten. De Python-interpreter is wel zo slim om te begrijpen dat als een string begint met het ene type aanhalingsteken, het andere type niet betekent dat dat het eind van de string is.

Het sleutelwoord `end` als argument van de functie `print()`

```
 9. print('Zeg eens: "Koe die in de rede valt."')
10. input()
11. print('Koe die in de rede v', end='')
12. print('-BOE!')
```

Zag je de tweede parameter op regel 11 bij `print()`? (Een parameter is trouwens een soort variabele, daar komen we later nog op terug.) Normaal is het zo dat de functie `print()` aan het eind van de string een nieuwe regel invoert. Dat wordt gedaan met een nieuweregelteken. Daarom krijg je een lege regel als je een `print()`-functie aanroept zonder tekst. Maar de `print()`-functie kan ook een tweede parameter bevatten (met de naam 'end'.)

De lege string die wordt doorgegeven, wordt een **sleutelwoordargument** genoemd. De parameter `end` heeft een specifieke naam en als je een argument wilt doorgeven aan deze specifieke parameter, moet je `end=` typen.

Door een lege string door te geven aan end, voegt de functie `print()` geen nieuweregelteken toe aan het eind van de regel, maar een lege string. Daarom wordt `'-BOE!'` uiteindelijk weergegeven achter de vorige regel en niet eronder. Er stond nu dus geen nieuweregelteken achter de string `'Koe die in de rede v'`.

Samenvatting

In dit hoofdstuk keken we naar een paar dingen die je met de functie `print()` kunt doen. Escape-tekens moet je gebruiken voor letters en tekens die een specifieke betekenis hebben in Python, bijvoorbeeld \', \", \\, \n, \t. Escape-combinaties worden getypt met een backslash \ en dan het teken met de speciale betekenis. \n gebruik je bijvoorbeeld als je een nieuwe regel wilt aanbrengen. Als je een backslash in je uitvoer wilt weergeven, moet je de backslash twee maal typen.

De functie `print()` voegt aan het einde van de string die wordt doorgegeven voor uitvoer op het scherm automatisch een nieuweregelteken toe. Dit is meestal handig. Maar soms wil je geen nieuweregelteken aan het einde van je zin. Dan kun je het sleutelwoordargument `end` doorgeven met

een lege string door end='' te typen. Als je bijvoorbeeld 'ham' wilt weergeven op het scherm zonder een nieuweregelteken, roep je de functie als volgt aan: `print('ham', end='')`.

Met deze kennis over hoe je de print-functie kunt gebruiken, kun je flexibeler zijn in hoe je tekst weergeeft op het scherm.

Hoofdstuk 6
DRAKENRIJK

In dit hoofdstuk behandelen we:
- De module `time`
- De functie `time.sleep()`
- Het sleutelwoord `return`
- Zelf functies maken met het sleutelwoord `def`
- De Boole-operatoren `and` en `or` en `not`
- Waarheidstabellen
- Het bereik van variabelen (globaal en lokaal)
- Parameters en argumenten
- Stroomdiagrammen

Functies

Je hebt al een paar functies gebruikt tot nu toe: `print()`, `input()`, `random.randint()`, `str()` en `int()`. Je hebt deze functies aangeroepen zodat de code in de functie kon worden uitgevoerd. Omdat dit ingebouwde functies zijn, heb je de code in de functie eigenlijk niet gezien. Die wordt 'achter de schermen' uitgevoerd. In dit hoofdstuk ga je je eigen functies schrijven die je dan in het programma aanroept. Zoals we weten, is een functie een soort miniprogramma binnen een programma. Met functies kun je dezelfde code meerdere keren uitvoeren zonder dat je de code steeds opnieuw moet typen of kopiëren en plakken. In plaats daarvan kun je de code eenmaal in een functie typen en vervolgens die functie zo vaak aanroepen als je wilt. Nog een voordeel is dat als de code in de functie een foutje bevat, je dat maar op één plek in het programma hoeft te verbeteren.

Het spel dat je in dit hoofdstuk maakt, heet 'Drakenrijk'. De speler moet beslissen welke van twee grotten hij binnengaat. De ene grot bevat een schat en de andere betekent een gruwelijk einde.

Hoe je Drakenrijk moet spelen

In dit spel bevindt de speler zich in een land vol draken. De draken wonen allemaal in grotten, waar ze hun enorme schatten bewaren. Sommige draken zijn aardig en willen hun schatten best met je delen. Andere draken zijn gemeen en vreten iedereen op die in de buurt komt. De speler staat voor twee grotten. In de ene woont een aardige draak en in de andere woont een hongerige draak. De speler moet kiezen welke grot hij binnengaat.

Open een nieuw venster in de bestandseditor door te klikken op **File ► New File** (Bestand > Nieuw bestand). In het nieuwe lege venster typ je de broncode en dit bestand sla je op als *draken.py*. Vervolgens voer je het programma uit met **F5**.

Voorbeelduitvoer van Drakenrijk

```
Je bent in een land waar veel draken wonen. Voor je
zie je twee grotten. In de ene grot is de draak aardig
en zal hij zijn schatten met je delen. De andere draak
```

```
is gemeen en hongerig en vreet je op met huid en haar.
Welke grot ga je binnen? (1 of 2)
1
Je nadert de grot...
Het is donker en griezelig...
Een grote draak springt tevoorschijn! Hij opent zijn bek en...
vreet je op en slikt je door!
Wil je nog een keer spelen? (ja of nee)
nee
```

Broncode van Drakenrijk

Wat je moet weten bij het lezen van de code hieronder: de blokken na de regels die beginnen met `def` definiëren een functie. Code definiëren is niet hetzelfde als code uitvoeren. Als je definieert, zeg je wat de functie gaat doen. Pas als je de functie aanroept, wordt de code ook uitgevoerd. Verderop in dit hoofdstuk ga je daar meer over lezen.

> **BELANGRIJKE OPMERKING!** De programma's in dit boek kunnen alleen worden uitgevoerd in Python 3, niet in Python 2. Zodra het IDLE-venster start, zie je bovenaan zoiets als 'Python 3.4.0'. Als je Python 2 al had geïnstalleerd, kun je daarna ook Python 3 installeren op dezelfde computer. Om Python 3 te downloaden, ga je naar https://python.org/download/.

Als je een foutmelding krijgt nadat je deze code hebt ingetypt, vergelijk jouw code dan met de code van het boek met behulp van de online diff tool op http://invpy.com/diff/draken.

draken.py

```
 1. import random
 2. import time
 3.
 4. def toonVerhaaltje():
 5.     print('Je bent in een land waar veel draken wonen. Voor je')
 6.     print('zie je twee grotten. In de ene grot is de draak aardig')
 7.     print('en zal hij zijn schatten met je delen. De andere draak')
 8.     print('is gemeen en hongerig en vreet je op met huid en haar.')
 9.     print()
10.
11. def kiesGrot():
12.     grot = ''
13.     while grot != '1' and grot != '2':
14.         print('Welke grot ga je binnen? (1 of 2)')
15.         grot = input()
16.
17.     return grot
18.
19. def controleerGrot(gekozenGrot):
20.     print('Je nadert de grot...')
21.     time.sleep(2)
22.     print('Het is donker en griezelig...')
23.     time.sleep(2)
24.     print('Een grote draak springt tevoorschijn! Hij opent zijn bek en...')
25.     print()
26.     time.sleep(2)
27.
```

```
28.     veiligeGrot = random.randint(1, 2)
29.
30.     if gekozenGrot == str(veiligeGrot):
31.         print('geeft jou al zijn schatten!!!!')
32.     else:
33.         print('vreet je op en slikt je door!')
34.
35. nogmaalsSpelen = 'ja'
36. while nogmaalsSpelen == 'ja' or nogmaalsSpelen == 'j':
37.
38.     toonVerhaaltje()
39.
40.     grotNummer = kiesGrot()
41.
42.     controleerGrot(grotNummer)
43.
44.     print('Wil je nog een keer spelen? (ja of nee)')
45.     nogmaalsSpelen = input()
```

Hoe de code werkt

We zullen de broncode eens van dichtbij bekijken.

```
1. import random
2. import time
```

Hiermee worden twee modules geïmporteerd in het programma. De module `random` levert de functie `random.randint()` die we nodig hebben voor het willekeurige getal net als bij ons 'Raad het getal'-spel. We hebben ook functies nodig die met tijd te maken hebben, dus we importeren de module `time`.

`def`-statements

```
4. def toonVerhaaltje():
5.     print('Je bevindt je in een land vol draken. Voor je')
6.     print('zie je twee grotten. In de ene grot is de draak aardig')
7.     print('en zal hij zijn schatten met je delen. De andere draak')
8.     print('is gemeen en hongerig en vreet je op met huid en haar.')
9.     print()
```

Regel 4 is een `def`-statement. De `def`-statement maakt een nieuwe functie aan, dat wil zeggen, **definieert** een nieuwe functie die je later in het programma kunt aanroepen. Zodra je deze functie hebt *gedefinieerd*, kun je de functie aanroepen, net zoals je dat doet met de ingebouwde functies van Python. Pas wanneer je deze functie *aanroept*, wordt de code in het def-blok uitgevoerd.

In Afbeelding 6-1 zie je de onderdelen van een `def`-statement. Je ziet het sleutelwoord `def` gevolgd door een functienaam met haakjes en daarachter een dubbele punt (het teken `:`). Het blok dat begint met de `def`-statement noemen we het def-blok.

Afbeelding 6-1: De onderdelen van een def-statement.

Vergeet niet dat de `def`-statement de code nog niet uitvoert. De def-statement vertelt alleen welke code moet worden uitgevoerd als je de functie aanroept. Als de programmaverwerking bij een `def`-statement aankomt, worden alle regels in het def-blok overgeslagen en gaat het programma verder met de eerste regel na het def-blok (het programma weet tot waar het def-blok doorloopt door het aantal spaties, weet je nog?).

Pas wanneer de functie `toonVerhaaltje()` wordt *aangeroepen* (op regel 38), gaat de verwerking de functie `toonVerhaaltje()` binnen, naar de eerste regel van het def-blok.

```
38.     toonVerhaaltje()
```

Alle `print()`-aanroepen worden uitgevoerd en het verhaaltje "Je bent in een land waar veel draken wonen..." wordt op het scherm weergegeven.

Waar je functiedefinities moet plaatsen

De `def`-statement en het def-blok voor een functie moeten in het programma staan *vóór* de plek waar je de functie aanroept. Dat is net zo als met variabelen, daar moet je ook eerst een waarde aan toewijzen voordat je de variabele kunt gebruiken. Als je eerst een functie aanroept en pas daarna vertelt wat er moet gebeuren in die functie, krijg je een foutmelding. Kijk naar deze code:

```
zegTotZiens()

def zegTotZiens():
    print('Tot ziens!')
```

Als je dit probeert uit te voeren, krijg je deze foutmelding:

```
Traceback (most recent call last):
  File "C:\Python34\ham.py", line 1, in <module>
zegTotZiens()
NameError: name 'zegTotZiens' is not defined
```

Dit los je op door de functiedefinitie vóór de functieaanroep te plaatsen:

```
def zegTotZiens():
    print('Tot ziens!')

zegTotZiens()
```

De functie `kiesGrot()` definiëren

```
11. def kiesGrot():
```

Hier definieer je een andere functie, namelijk `kiesGrot()`. De code van deze functie gaat de speler vragen welke grot ze binnen willen gaan, 1 of 2.

```
12.     grot = ''
13.     while grot != '1' and grot != '2':
```

Deze code controleert of de speler 1 of 2 heeft ingetypt, en niet iets anders. Als de speler '3' intypt of 'Draken stinken!' zorgt de While-lus ervoor dat het programma niet verder gaat, maar blijft vragen welke grot de speler binnen wil gaan.

Op regel 12 wordt een nieuwe variabele aangemaakt, met de naam `grot` en hierin wordt een lege string opgeslagen. Dan begint er een `while`-lus op regel 13. De voorwaarde van de while-lus bevat een nieuwe operator die je nog niet bent tegengekomen: `and`. We kennen `-` of `*` al, de wiskundige operatoren, en we kennen `==` of `!=`, de vergelijkingsoperatoren. Nou, `and` is ook een operator, een zogenaamde Boole-operator.

Boole-operatoren

Booleaanse logica houdt zich bezig met zaken die ofwel `True` (Waar) of `False` (Niet waar) zijn. (George Boole was een Britse wiskundige uit de 19e eeuw, vandaar de naam.) Boole-operatoren vergelijken twee Boole-waarden en retourneren één Boole-waarde. De Boole-operator `and` combineert twee Boole-waarden en produceert een nieuwe Boole-waarde.

Denk aan deze zin: "Katten hebben scherpe nagels en honden hebben een staart." Is die zin waar? "Katten hebben scherpe nagels" is waar en "honden hebben een staart" is ook waar, dus de hele zin "Katten hebben scherpe nagels **and** honden hebben een staart" is waar.

Maar de zin "Katten hebben scherpe nagels en honden hebben vleugels" zou niet waar zijn. Ook al is het stukje "katten hebben scherpe nagels" wel waar, het stukje over honden met vleugels is niet waar. In de Booleaanse logica kunnen dingen alleen maar helemaal waar of helemaal niet waar zijn. Omdat de operator 'and' er staat, is de hele zin alleen maar waar als **beide** delen waar zijn. Als één of beide delen niet waar zijn, is de hele zin niet waar.

De operatoren `and` en `or`

De `and`-operator in Python gedraagt zich hetzelfde. Als de Booleaanse waarden aan beide zijden van het sleutelwoord `and` waar zijn (`True`), is de hele expressie met de operator `and` ook `True`. Als een van de beide Boole-waarden `False` zijn (niet waar), of als ze allebei `False` zijn, dan wordt de hele expressie geëvalueerd als `False`.

Probeer de volgende expressies met de operator `and` eens in de interactieve shell te typen:

```
>>> True and True
True
>>> True and False
False
>>> False and True
False
>>> False and False
False
>>> 10 < 20 and 'Hallo' == 'Hallo'
True
```

De operator `or` (of) lijkt op de operator `and`, alleen levert hij ook `True` op als maar *één* van de twee Boole-waarden `True` is. De enige keer dat de operator `or` `False` oplevert, is als *beide* Boole-waarden `False` zijn.

Probeer het volgende eens in de interactieve shell te typen:

```
>>> True or True
True
>>> True or False
True
>>> False or True
True
>>> False or False
False
>>> 10 > 20 or 20 > 10
True
```

De operator `not`

De operator `not` werkt maar op één waarde tegelijk en combineert dus niet meerdere waarden met elkaar. De operator `not` retourneert de tegenovergestelde Boole-waarde. De expressie `not True` retourneert `False` en `not False` retourneert `True`.

Probeer het volgende eens in de interactieve shell te typen:

```
>>> not True
False
>>> not False
True
>>> not 'zwart' == 'wit'
True
```

Waarheidstabellen

Als je ooit vergeet hoe het ook al weer zat met die Boole-operatoren kun je naar dit diagram kijken. Dit heet een waarheidstabel:

Tabel 6-1: De waarheidstabel van de operator `and`.

A	And	B	is	Hele statement
True	And	True	is	True
True	And	False	is	False
False	And	True	is	False
False	And	False	is	False

Tabel 6-2: De waarheidstabel van de operator `or`.

A	Or	B	is	Hele statement
True	Or	True	is	True
True	Or	False	is	True
False	Or	True	is	True
False	Or	False	is	False

Tabel 6-3: De waarheidstabel van de operator `not`.

not A	is	Hele statement
not True	is	False
not False	is	True

Boole-operatoren evalueren

Kijk nog eens naar regel 13:

```
13.     while grot != '1' and grot != '2':
```

De voorwaarde heeft twee delen die met elkaar worden gecombineerd door de Boole-operator `and`. De voorwaarde is alleen `True` als beide onderdelen `True` zijn.

De eerste keer dat de voorwaarde van de `while`-statement wordt getest, wordt `grot` ingesteld op de lege string `''`. De lege string is niet gelijk aan de string `'1'`, dus de linkerzijde retourneert `True`. De lege string is ook niet gelijk aan de string `'2'`, dus de rechterzijde retourneert ook `True`.

De voorwaarde wordt dus `True and True`. En omdat beide Boole-waarden `True` zijn, wordt de hele voorwaarde geëvalueerd als `True`. Dus de programmaverwerking gaat het while-blok binnen.

Hier zie je hoe de evaluatie van de voorwaarde in zijn werk gaat (wanneer de waarde van `grot` de lege string is):

```
while grot != '1' and grot != '2':
                   ▼
while  ''  != '1' and grot != '2':
                   ▼
while     True    and grot != '2':
                   ▼
while     True    and  ''  != '2':
                   ▼
while     True    and     True:
                   ▼
while             True:
```

De invoer van de speler lezen

```
13.     while grot != '1' and grot != '2':
14.         print('Welke grot ga je binnen? (1 of 2)')
15.         grot = input()
```

Regel 14 vraagt de speler welke grot ze kiezen. Regel 15 laat de speler zijn antwoord typen. Dit antwoord wordt bewaard in de variabele `grot`. Zodra deze code is uitgevoerd, gaat de verwerking terug naar het begin van de `while`-statement en wordt de voorwaarde opnieuw getest.

Als de speler 1 of 2 heeft getypt, staat er in `grot` ofwel `'1'` of `'2'`. Hierdoor wordt de waarde als `False` geëvalueerd en gaat de programmaverwerking verder, voorbij de `while`-lus. Als de gebruiker bijvoorbeeld '1' heeft ingevoerd, ziet de evaluatie er als volgt uit:

```
while grot != '1' and grot != '2':
                    ▼
while  '1' != '1' and grot != '2':
                    ▼
while     False   and grot != '2':
                    ▼
while     False   and  '1' != '2':
                    ▼
while     False   and     True:
                    ▼
while             False:
```

Maar als de speler 3 of 4 of HALLO zou hebben getypt, zou dat antwoord ongeldig zijn. De voorwaarde zou dan `True` zijn en de verwerking zou het while-blok weer zijn binnengegaan om de speler opnieuw te vragen 1 of 2 te typen. Het programma blijft dezelfde vraag stellen, tot de gebruiker 1 of 2 typt. Hiermee garanderen we dat de variabele `grot` een geldig antwoord bevat.

Retourwaarden

```
17.     return grot
```

Dit is een `return`-statement, die alleen voorkomt binnen een def-blok. Weet je nog dat de functie `input()` een stringwaarde retourneert met de tekst die de speler heeft ingetypt? De functie `kiesGrot()` retourneert ook een waarde. Regel 17 retourneert de string die in `grot` is opgeslagen, ofwel `'1'` of `'2'`.

Zodra de `return`-statement wordt uitgevoerd, springt de programmaverwerking direct het def-blok uit. (Net zoals de `break`-statement de verwerking het while-blok uit laat springen, zoals we laatst al zagen.) De programmaverwerking gaat terug naar de regel met de functieaanroep. En de functieaanroep `kiesGrot` levert nu de waarde op van de variabele `grot`.

We slaan een stukje over en kijken even naar regel 40:

```
40.     grotNummer = kiesGrot()
```

Wanneer de functie `kiesGrot()` later wordt aangeroepen door het programma op regel 40, wordt de waarde die de functie retourneert, opgeslagen in de variabele `grotNummer`. De `while`-lus zorgt ervoor dat de functie `kiesGrot()` alleen `'1'` of `'2'` als retourwaarde oplevert.

Dus regel 17 retourneert een string die het resultaat is van de functie. Dit resultaat wordt vervolgens opgeslagen in de variabele `grotNummer`.

Globaal bereik en lokaal bereik van variabelen

De variabelen van een programma worden vergeten zodra het programma is afgelopen. Dat wist je al. De variabelen in ons programma, die worden aangemaakt terwijl de programmaverwerking met een functie bezig is, gedragen zich net zo. De variabelen worden gemaakt wanneer de functie wordt aangeroepen en ze worden vergeten zodra de functie zijn retourwaarde heeft gegeven.

En dat niet alleen, maar wanneer de programmaverwerking binnen een functie zit, kun je een variabele buiten de functie niet gaan veranderen, ook niet de variabelen die in andere functies zitten. Het programma zou niet op de hoogte zijn van die veranderingen. Dat komt omdat deze variabelen in een ander 'bereik' voorkomen. Alle variabelen bestaan ofwel in het globale bereik van een programma, of in het lokale bereik van een functie.

Het bereik buiten alle functies om wordt het **globale bereik** genoemd. Het bereik binnen een functie (zo lang als een bepaalde functie duurt) wordt het **lokale bereik** genoemd. In het gehele programma is maar één globaal bereik en elke functie heeft zijn eigen lokale bereik.

Variabelen die in het globale bereik zijn gedefinieerd, kunnen zowel buiten een functie als binnen een functie worden gelezen en gebruikt, maar hun waarde kan alleen buiten alle functies worden gewijzigd. Variabelen die binnen een functie zijn aangemaakt, kunnen alleen worden gelezen en gewijzigd binnen die functie.

Je kunt de waarde van globale variabelen lezen in een lokaal bereik, maar als je die globale variabele vanuit het lokale bereik probeert te wijzigen, zal dat niet lukken. Wat Python dan doet, is dat hij een lokale variabele aanmaakt, die toevallig **dezelfde naam** heeft als de globale variabele. Je kunt bijvoorbeeld een lokale variabele hebben met de naam `ham` en tegelijk een globale variabele met de naam `ham`. Python denkt dan dat het twee verschillende variabelen zijn.

Kijk naar dit voorbeeld om te zien wat er gebeurt als je een globale variabele probeert te veranderen vanuit een lokaal bereik. Het commentaar vertelt wat er gebeurt:

```
def spek():
    # Er wordt een lokale variabele met de naam 'ham'
    # gemaakt in plaats van dat de waarde van de globale variabele
    # 'ham' wordt gewijzigd:
    ham = 99
    # De naam 'ham' verwijst nu naar een lokale
    # variabele die alleen in deze functie
    # voorkomt.
    print(ham)    # 99

ham = 42 # Dit is een globale variabele met de naam 'ham':
print(ham) # 42
spek() # We roepen de functie spek() aan:
# De globale variabele is niet gewijzigd:
print(ham)    # 42
```

Wanneer deze code wordt uitgevoerd, zie je het volgende resultaat:

```
42
99
42
```

De plek waar een variabele wordt gemaakt, bepaalt in welk bereik hij voorkomt. Wanneer het programma Drakenrijk de volgende regel voor het eerst verwerkt:

```
12.     grot = ''
```

...wordt de variabele grot aangemaakt binnen de functie kiesGrot(). Dus hij wordt aangemaakt in het lokale bereik van de functie kiesGrot(). Wanneer de functie kiesGrot() zijn waarde heeft geretourneerd, wordt de variabele vergeten en als kiesGrot() nog een keer wordt aangeroepen, wordt

de variabele opnieuw aangemaakt. Tussen twee functieaanroepen wordt de waarde van een lokale variabele niet onthouden.

De functie `controleerGrot()` definiëren

```
19. def controleerGrot(gekozenGrot):
```

De volgende functie die wordt gedefinieerd, heet `controleerGrot()`. Let op de tekst `gekozenGrot` tussen de haakjes. Dit noemen we een **parameter**: een lokale variabele die de waarde (het argument) krijgt die aan de functie werd meegegeven toen hij werd aangeroepen.

We zijn argumenten en parameters al eerder tegengekomen. Weet je nog dat je bij een functieaanroep als `str()` of `randint()` een argument meegaf tussen de haakjes?

```
>>> str(5)
'5'
>>> random.randint(1, 20)
14
```

Op dezelfde manier geef je een argument door als je `controleerGrot()` aanroept. Dit argument wordt opgeslagen in een nieuwe variabele genaamd `gekozenGrot`. Deze variabelen noemen we ook parameters.

Parameters

Parameters zijn lokale variabelen die worden aangemaakt wanneer een functie wordt aangeroepen. Hier is bijvoorbeeld een kort programmaatje dat het definiëren van een functie met een parameter laat zien. De functie `len()` wordt gebruikt om de lengte van een string te retourneren.

```
def zegHallo(naam):
    print('Hallo, ' + naam + '. Jouw naam heeft ' + str(len(naam)) + 'letters.')

zegHallo('Ellen')
zegHallo('Rob')
ham = 'Carolien'
zegHallo(ham)
```

Als je dit programma uitvoert, ziet het er als volgt uit:

```
Hallo, Ellen. Jouw naam heeft 5 letters.
Hallo, Rob. Jouw naam heeft 3 letters.
Hallo, Carolien Jouw naam heeft 8 letters.
```

Wanneer je `zegHallo()` aanroept, wordt het argument ('Ellen', 'Rob' of de inhoud van de variabele ham) toegewezen aan de parameter `naam`. Parameters zijn gewoon lokale variabelen. Net als bij alle lokale variabelen worden de waarden in parameters vergeten als de functieaanroep is uitgevoerd. De naam van de parameter in de functieaanroep hoeft overigens niet hetzelfde te zijn als de naam van de parameter in de functiedefinitie. Als we controleerGrot() aanroepen op regel 42 is de parameternaam 'grotNummer'. In de definitie van controleerGrot() heet de parameter 'gekozenGrot'. Python begrijpt wel dat dat hetzelfde is.

Het resultaat van het spel weergeven

Terug naar de broncode van het spel:

```
20.     print('Je nadert de grot...')
21.     time.sleep(2)
```

Weet je nog dat om de functie `random.randint()` te kunnen gebruiken, we eerst de module random moesten importeren, met `import random`? In het spel Drakenrijk importeer je op regel 2 de module `time`. In de module `time` zit een functie genaamd `sleep()` waarmee je het programma even kunt pauzeren. Met deze functie wordt het programma een paar seconden lang stilgelegd. Regel 21 geeft de integerwaarde `2` door als argument, zodat met `time.sleep()` het programma 2 seconden stilstaat.

```
22.     print('Het is donker en griezelig...')
23.     time.sleep(2)
```

Hier geeft de code nog wat tekst weer en wordt er weer 2 seconden gewacht. Die korte pauzes bouwen wat spanning op tijdens het spel, anders zou alle tekst achter elkaar worden weergegeven; dat zou niet spannend zijn. In het Moppen-programma in het vorige hoofdstuk riep je de functie `input()` aan om te pauzeren tot de speler op de ENTER-toets drukte. Maar hier hoeft de speler niets in te voeren. Hij moet alleen een paar seconden wachten.

```
24.     print('Een grote draak springt tevoorschijn! Hij opent zijn bek en...
25.     print()
26.     time.sleep(2)
```

Wat gebeurt er nu? En hoe weet het programma wat er moet gebeuren? Dat leggen we uit in de volgende paragraaf.

Bepalen in welke grot de aardige draak woont

```
28.     veiligeGrot = random.randint(1, 2)
```

Het programma kiest willekeurig in welke grot de aardige draak woont. Regel 28 roept de functie `random.randint()` aan die ofwel `1` of `2` retourneert, helemaal willekeurig. Deze integerwaarde wordt bewaard in de variabele `veiligeGrot`.

```
30.     if gekozenGrot == str(veiligeGrot):
31.         print('geeft jou al zijn schatten!!!!')
```

Regel 30 controleert of de grot die de speler heeft gekozen en die wordt bewaard in de variabele `gekozenGrot` (`'1'` of `'2'`), gelijk is aan de veilige grot van de aardige draak.

De waarde in `gekozenGrot` is een stringwaarde, omdat de functie `input()` altijd stringwaarden retourneert. Maar de waarde in `veiligeGrot` is een integer omdat `random.randint()` integers retourneert. Je kunt strings en integers niet met elkaar vergelijken met de vergelijkingsoperator `==` omdat ze **altijd** ongelijk zijn aan elkaar (omdat ze ongelijke datatypen zijn). `'1'` is niet hetzelfde als `1` en `'2'` is niet hetzelfde als `2`.

Dus we geven `veiligeGrot` door aan de functie `str()` die de stringwaarde van `veiligeGrot` oplevert. Zo krijgen de waarden hetzelfde datatype en kunnen ze op een geldige manier met elkaar worden vergeleken. Je had ook deze code kunnen gebruiken:

```
    if int(gekozenGrot) == veiligeGrot:
```

De voorwaarde in de bovenstaande `if`-statement vergelijkt de integerwaarde die het resultaat is van de functie `int()` met de integerwaarde van `veiligeGrot`. Zo kun je het ook doen.

Als de voorwaarde `True` is, vertelt regel 31 dat de speler de schat heeft gewonnen.

```
32.     else:
33.         print('vreet je op en slikt je door!')
```

Regel 32 is een **else**-statement. Die hadden we nog niet gezien. Het sleutelwoord `else` zie je alleen na een if-blok. Het else-blok wordt uitgevoerd als de voorwaarde van de `if`-statement `False` oplevert. Je kunt het als volgt zeggen: "*Als* (If) deze voorwaarde waar is, voer je het if-blok uit en *anders* (Else) voer je het else-blok uit."

Vergeet de dubbele punt (:) niet achter het sleutelwoord `else`.

Waar het programma eigenlijk begint

```
35. nogmaalsSpelen = 'ja'
36. while nogmaalsSpelen == 'ja' or nogmaalsSpelen == 'j':
```

Regel 35 is de eerste regel die geen `def`-statement is of in een def-blok zit. Deze regel is waar het programma in feite begint. De voorgaande `def`-statements definieerden alleen die functies. De code binnen die functies werd nog niet uitgevoerd.

De regels 35 en 36 stellen een lus in waarin de rest van de code plaatsvindt. Aan het eind van het spel kan de speler zeggen of ze nog een keer willen spelen. Als dat zo is, gaat de verwerking de `while`-lus weer in om het hele spel nog eens te doorlopen. Als dat niet zo is, is de voorwaarde van de `while`-statement `False` en gaat de verwerking naar het eind van het programma en wordt het programma beëindigd.

De eerste keer dat de verwerking bij deze `while`-statement aankomt, heeft regel 35 net `'ja'` toegekend aan de variabele `nogmaalsSpelen`. Dat betekent dat de voorwaarde `True` is.

De functies in het programma aanroepen

```
38.     toonVerhaaltje()
```

Regel 38 roept de functie `toonVerhaaltje()` aan. Dit is geen ingebouwde Python-functie, het is jouw zelfgeschreven functie, die je eerder op regel 4 hebt gedefinieerd. Zodra deze functie wordt aangeroepen, springt de programmaverwerking naar de eerste regel van de functie `toonVerhaaltje()` op regel 5. Als alle regels in de functie zijn doorlopen, springt de verwerking weer terug naar regel 38 en verder naar beneden.

```
40.     grotNummer = kiesGrot()
```

Regel 40 bevat ook een functieaanroep naar een functie die jij hebt geschreven. De functie `kiesGrot()` laat de speler het nummer typen van de grot die ze willen binnengaan. Zodra `return grot` op regel 17 is verwerkt, springt het programma terug naar regel 40 en retourneert de aanroep van `kiesGrot()` de retourwaarde van die functie. Deze retourwaarde wordt weer opgeslagen in een nieuwe variabele met de naam `grotNummer`. Vervolgens gaat het programma naar regel 42.

```
42.     controleerGrot(grotNummer)
```

Deze regel roept de functie `controleerGrot()` aan en geeft de waarde van `grotNummer` door als argument. De programmaverwerking springt niet alleen naar regel 20, maar de waarde in `grotNummer` wordt ook gekopieerd naar de parameter `gekozenGrot` in de functie `controleerGrot()`. Dit is de functie die ofwel `'geeft je al zijn schatten!!!!'` of `'vreet je op en slikt je door!'` weergeeft, afhankelijk van de grot die de speler binnenging.

De speler vragen of ze nog een keertje willen spelen

```
44.     print('Wil je nog een keer spelen? (ja of nee)')
45.     nogmaalsSpelen = input()
```

Of de speler nu gewonnen heeft of verloren, er wordt toch gevraagd of ze nog een keertje willen spelen. De variabele `nogmaalsSpelen` bewaart de string die de speler heeft getypt. Regel 45 is de laatste regel van het while-blok, dus het programma springt terug naar regel 36 om de voorwaarde te testen: `nogmaalsSpelen == 'ja' or nogmaalsSpelen == 'j'`

Als de speler de string `'ja'` heeft getypt, of alleen `'j'`, gaat de programmaverwerking de lus nog een keer binnen op regel 38.

Als de speler `'nee'` of `'n'` of iets flauws als `'De groeten'` heeft getypt, is de voorwaarde `False`. Dan gaat het programma verder naar de regel na het while-blok. Maar omdat er geen regels meer zijn na het while-blok, wordt het programma beëindigd.

Let op: de string `'JA'` is niet gelijk aan de string `'ja'`. Als de speler de string `'JA'` heeft ingetypt, zou de voorwaarde van de `while`-statement worden geëvalueerd als `False` en zou het programma worden beëindigd. Dat zou eigenlijk natuurlijk niet moeten. In programma's verderop in dit boek gaan we kijken hoe we dat kunnen voorkomen.

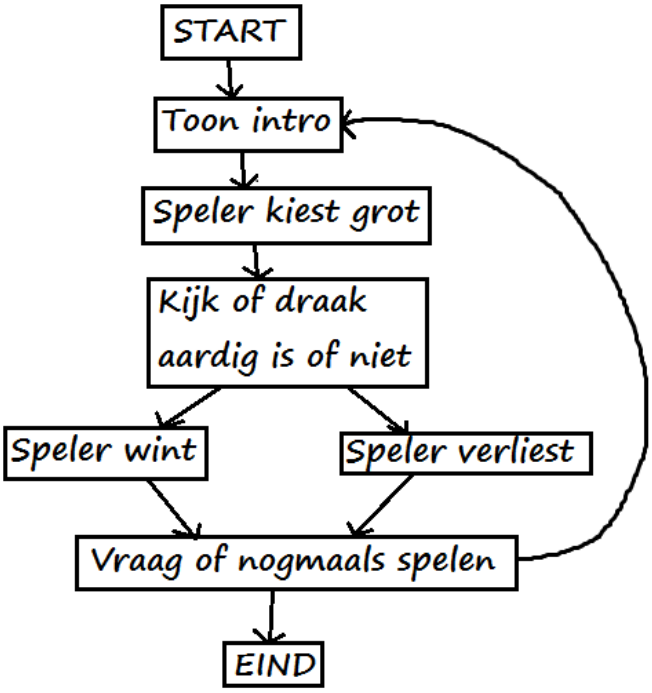
Je hebt zojuist je tweede spel voltooid! In Drakenrijk heb je veel van wat je eerder hebt geleerd bij 'Raad het getal' opnieuw toegepast en je hebt nog een paar trucjes geleerd. Als je wat we hier behandeld hebben, nog niet helemaal begrijpt, kun je de samenvatting aan het einde van dit hoofdstuk even doorlezen. Misschien wordt het dan duidelijker. Of doorloop elke regel van de broncode nog eens en probeer de broncode hier en daar een beetje te wijzigen en kijk dan hoe het programma reageert. Daar kun je veel van leren.

In het volgende hoofdstuk gaan we geen game maken maar leer je hoe je een handige voorziening van IDLE kunt gebruiken, de Debugger.

Het programma ontwerpen

Drakenrijk is een simpel spelletje. De andere spellen in dit boek zullen wat ingewikkelder zijn. Soms helpt het om voordat je begint met het schrijven van code, eerst een diagram te maken van wat je precies wilt doen. Dit noemen we het 'ontwerp van het programma'.

Het kan bijvoorbeeld heel nuttig zijn om een stroomdiagram te tekenen. Een **stroomdiagram** is een tekening die alle mogelijke acties weergeeft die in een spel kunnen voorkomen en welke acties naar welke andere acties leiden. Afbeelding 6-2 is een stroomdiagram voor Drakenrijk.



Afbeelding 6-2: Stroomdiagram voor het spel Drakenrijk.

Om te zien wat er in het spel gebeurt, zet je je vinger bij het vak 'Start'. Dan volg je een pijl van dit vak naar het volgende vak. Je vinger is de programmaverwerking. Het programma wordt beëindigd als je vinger bij EIND is aangekomen.

Als je bij het vak 'Kijk of draak aardig is of niet' bent aangekomen, kun je daarvandaan naar 'Speler wint' of 'Speler verliest' gaan. Dit vertakkingspunt laat zien hoe het programma verschillende keuzes kan maken. Maar uiteindelijk eindigen beide paden bij het vak 'Vragen of nogmaals spelen'.

Samenvatting

In het spel 'Drakenrijk' heb je je eigen functies gemaakt. Functies zijn miniprogramma's binnen je programma. De code in deze functies wordt pas uitgevoerd als de functie wordt *aangeroepen*. Door je code op te delen in functies, kun je je code in kleinere en duidelijkere onderdelen ordenen.

Argumenten zijn waarden die aan de code van de functie worden doorgegeven als de functie wordt aangeroepen. De functieaanroep zelf levert een waarde op, de retourwaarde, die je weer in een variabele kunt stoppen. Dat gebeurt bijvoorbeeld in een regel zoals grotNummer = kiesGrot().

Je weet nu ook iets over het bereik van variabelen. Variabelen die binnen een functie worden aangemaakt, 'leven' alleen in die functie en variabelen die buiten alle functies worden gemaakt, 'leven' in het globale bereik. Code in het globale bereik kan niet gebruikmaken van lokale variabelen. Als een lokale variabele dezelfde naam heeft als een globale variabele, denkt Python gewoon dat het twee verschillende variabelen zijn. Als je een nieuwe waarde aan de lokale variabele toewijst, gebeurt er niets met de waarde in de globale variabele.

Bereik van variabelen lijkt wat ingewikkeld, maar het is best heel handig dat het zo werkt, want functies blijven dan aparte stukjes code in je programma die zonder invloed van buitenaf hun werk blijven doen. Omdat elke functie zijn eigen lokale bereik heeft, weet je zeker dat de code in de ene functie geen bugs zal veroorzaken in een andere functie.

In bijna elk programma komen functies voor, omdat ze zo handig zijn. Als je begrijpt hoe functies werken en wat het nut is, kun je jezelf veel typewerk besparen en zijn bugs gemakkelijker te corrigeren

Hoofdstuk 7
DE DEBUGGER GEBRUIKEN

In dit hoofdstuk behandelen we:
- 3 verschillende soorten fouten
- De Debugger van IDLE
- Stappen in, over en uit
- Go en Quit
- Breekpunten

Bugs!

"Bij twee gelegenheden is mij gevraagd, 'Excuseer mijnheer Babbage, als u in uw machine onjuiste cijfers stopt, komen er dan wel juiste antwoorden uit?' Ik moet zeggen dat ik waarlijk niet de verwarde gedachtegang kan volgen die een dergelijke vraag opwerpt!"

-Charles Babbage, de 19e-eeuwse wiskundige die het idee opperde van een programmeerbare computer.

Natuurlijk is het zo dat als je de verkeerde code invoert, de uitvoer ook niet zal kloppen. Een computerprogramma zal altijd doen wat je zegt, maar wat je het computerprogramma soms zegt is niet altijd hetzelfde als wat je had *willen* zeggen. Dit soort fouten noemen we **bugs** in een computerprogramma. Bugs treden op wanneer de programmeur niet zorgvuldig genoeg heeft nagedacht over wat het programma precies moet doen. Er zijn drie soorten bugs die kunnen voorkomen in je programma:

- **Syntaxisfouten** zijn een soort bug die wordt veroorzaakt door typefouten. Wanneer de Python-interpreter een syntaxisfout ziet, komt dat omdat jouw code niet voldoet aan de regels van de Python-programmeertaal. Als er ook maar 1 syntaxisfoutje in een Python-programma zit, kan het programma niet worden uitgevoerd.
- **Runtime-fouten** zijn bugs die optreden terwijl het programma wordt uitgevoerd. Het programma wordt uitgevoerd tot het punt dat de regel code met de fout wordt bereikt, en dan wordt het programma beëindigd met een foutmelding (dit heet **crashen**). De Python-interpreter geeft een 'traceback' weer en laat de regel zien waar het probleem optrad.
- En dan zijn er nog **semantische fouten**, en die zijn het lastigst te 'fixen'. Deze bugs laten het programma niet crashen maar door deze bugs doet het programma niet precies wat de bedoeling van de programmeur was. Als de programmeur bijvoorbeeld wil dat de variabele `totaal` de som is van de waarden in de variabelen `a`, `b` en `c`, maar hij schreef `totaal = a * b * c`, is de waarde van `totaal` helemaal verkeerd. Het programma kan hierdoor verderop misschien ook nog crashen, maar het is niet direct duidelijk waar deze semantische fout optrad.

Bugs opsporen in een programma kan lastig zijn, en soms vind je ze misschien helemaal niet! Wanneer je je programma uitvoert, merk je soms misschien dat functies niet worden aangeroepen wanneer je dat verwachtte of misschien worden ze te vaak aangeroepen. Misschien heb je de voorwaarde van een `while`-lus verkeerd gecodeerd, zodat de lus een verkeerd aantal keer wordt

doorlopen. (Als je nooit uit een lus in je programma kunt komen, heet dat een **oneindige lus**. Om het programma dan te stoppen, kun je op **Ctrl-C** drukken in de interactieve shell.) Al deze dingen kunnen gebeuren in je code als je niet voorzichtig bent.

Weet je wat, we maken gewoon even een oneindige lus in de interactieve shell door de volgende code te typen (je moet tweemaal op Enter drukken om de interactieve shell te laten weten dat je klaar bent met het typen van code voor de lus):

```
>>> while True: print('Druk op Ctrl-C om deze oneindige lus te onderbreken!!!')
...
```

Houd nu de Ctrl-toets ingedrukt terwijl je op C drukt om het programma te stoppen. De interactieve shell ziet er als volgt uit:

```
Druk op Ctrl-C om deze oneindige lus te onderbreken!!!
Druk op Ctrl-C om deze oneindige lus te onderbreken!!!
Druk op Ctrl-C om deze oneindige lus te onderbreken!!!
Druk op Ctrl-C om deze oneindige lus te onderbreken!!!
Druk op Ctrl-C om deze oneindige lus te onderbreken!!!
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    while True: print('Druk op Ctrl-C om deze oneindige lus te onderbreken!!!')
KeyboardInterrupt
```

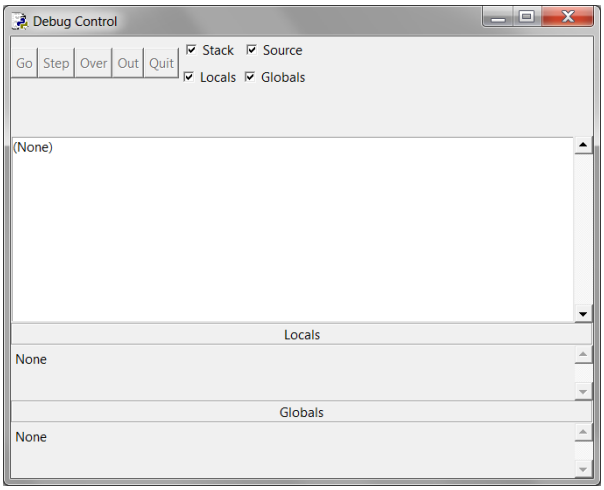
De Debugger

Soms kan het moeilijk zijn om uit te vogelen waar in je code een bug zit. De regels code worden razendsnel uitgevoerd en de waarden in variabelen veranderen zo vaak. Een **Debugger** is een speciaal programma dat je de mogelijkheid geeft om regel voor regel door je code te stappen in dezelfde volgorde als de Python-interpreter de regels verwerkt. De Debugger laat ook zien welke waarden bij elke stap in je variabelen zitten.

De Debugger starten

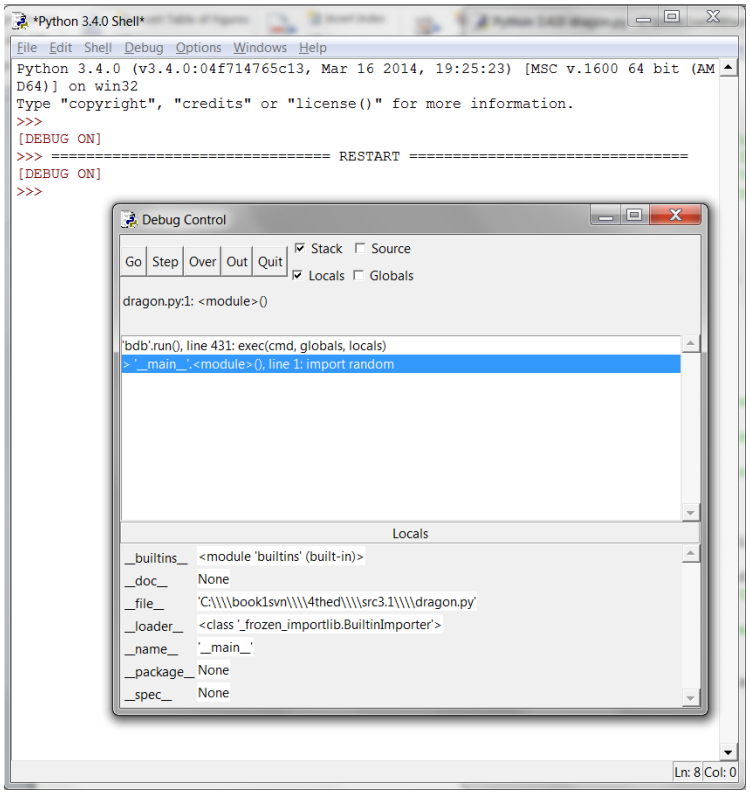
Open in IDLE het bestand Draken.py dat je in het vorige hoofdstuk hebt gemaakt. Klik in de interactieve shell op **File** en dan op **Open** en selecteer dan *draken.py* (of de naam die jij gaf aan het programma toen je het opsloeg).

Als je het bestand *draken.py* hebt geopend, klik je op **Debug ► Debugger** zodat het Debug Control-venster verschijnt (Afbeelding 7-1).



Afbeelding 7-1: Het Debug Control-venster.

Als je nu het spel Drakenrijk start, door op **F5** te drukken, wordt de Debugger van IDLE geactiveerd. 'Debuggen' zou je kunnen vertalen als 'Ontfouten'. In het Debug Control-venster schakel je de selectievakjes **Source** en **Globals** in.



Afbeelding 7-2: Het spel Drakenrijk uitvoeren met de Debugger.

Wanneer je Python-programma's uitvoert met de geactiveerde Debugger, stopt het programma voordat de eerste regel code wordt verwerkt. Als je op de titelbalk klikt in de bestandseditor (en het selectievakje Source hebt ingeschakeld in het controlevenster van de Debugger), wordt de eerste regel code gemarkeerd in grijs. Het Debug Control-venster laat zien dat de verwerking op regel 1 staat, de regel met `import random`.

Stappen

Met de Debugger kun je de code regel voor regel laten uitvoeren. Dit heet **stappen**. Om één instructie te laten uitvoeren, klik je op de knop Step in het Debug Control-venster. Probeer maar. Python voert de instructie `import random` uit en stopt dan vóór de volgende instructie. Het Debug Control-venster laat zien dat de verwerking nu op regel 2 staat, de regel met `import time`. Klik op de knop **Quit** (Afsluiten) om het programma nu even te sluiten.

Hier volgt een overzicht van wat er gebeurt als je op de knop Step klikt wanneer je Drakenrijk uitvoert met de geactiveerde Debugger. Druk op **F5** om Drakenrijk nogmaals te starten en volg dan deze instructies:

1. Klik tweemaal op de knop **Step** om de twee regels met `import` uit te voeren.
2. Klik nog driemaal op de knop **Step** om de drie `def`-statements uit te voeren.
3. Klik nogmaals op de knop **Step** om de variabele `nogmaalsSpelen` te definiëren.
4. Klik op **Go** om de rest van het programma uit te voeren of klik op **Quit** om het programma te beëindigen.

Het Debug Control-venster laat zien welke regel *op het punt staat* te worden uitgevoerd wanneer je op de knop Step klikt. De Debugger sloeg regel 3 over omdat dat een lege regel was. Je kunt overigens alleen vooruit stappen met de Debugger, niet achteruit.

De sectie Globals

In de sectie **Globals** in het Debug Control-venster worden alle globale variabelen getoond. Globale variabelen waren die variabelen die buiten functies werden aangemaakt, weet je nog? (Dus in het globale bereik.)

Omdat de drie `def`-statements functies definiëren, zie je deze terug in de sectie Globals van het controlevenster.

De tekst naast de functienamen in de sectie Globals ziet eruit als “<function controleerGrot at 0x012859B0>“. De modulenamen hebben ook vreemd uitziende tekst naast zich staan, zoals “<module 'random' from 'C:\\Python31\\lib\\random.pyc'>“. Dit is gedetailleerde informatie die van nut kan zijn aan gevorderde Python-programmeurs, maar deze informatie heb jij nu niet nodig om je programma's te kunnen debuggen. Het feit dat de functies en modules hier worden weergegeven in de sectie Global, is het bewijs dat de functie is gedefinieerd en de module goed is geïmporteerd.

Je kunt ook de regels `__builtins__`, `__doc__` en `__name__` negeren in de sectie Global. (Dit zijn variabelen die in elk Python-programma voorkomen.)

Zodra de variabele `nogmaalsSpelen` is aangemaakt, zie je hem terug in de sectie Global. Naast de variabelenaam zie je de huidige waarde van de variabele, de string `'ja'`. De Debugger toont je de waarden van alle variabelen in het programma terwijl het programma wordt uitgevoerd. Dit is handig bij het fixen van bugs in je programma.

De sectie Locals

Er is ook een sectie **Locals**, waarin je de variabelen ziet met lokaal bereik en hun waarden. De sectie Local toont alleen variabelen wanneer de programmaverwerking binnenin een functie bezig is. Als de verwerking in het globale bereik bezig is, is deze sectie leeg.

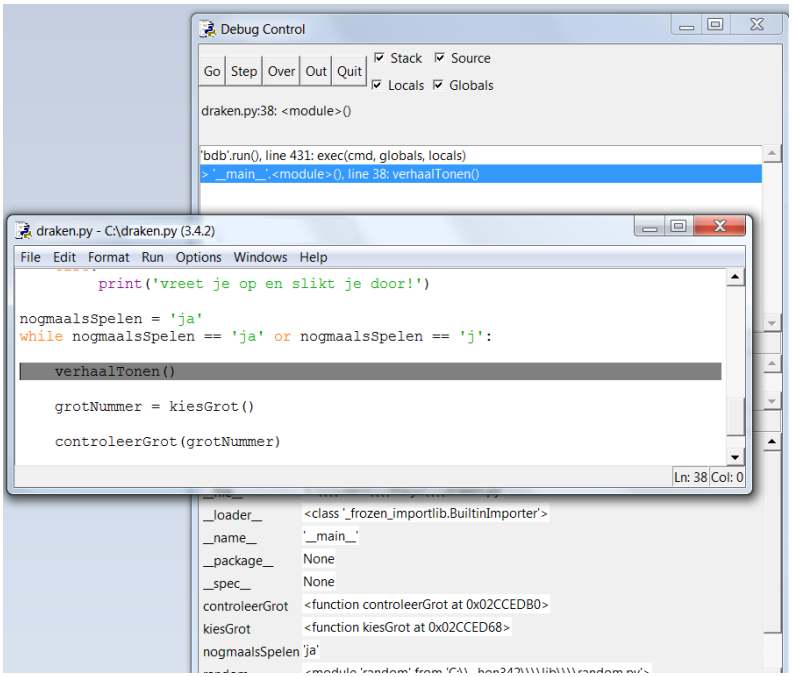
De knoppen Go en Quit

Als je genoeg hebt van het klikken op de knop **Step** en het programma gewoon verder wilt laten lopen, klik je op de knop **Go** bovenaan het Debug Control-venster. Dan gaat het programma gewoon verder in plaats van stap voor stap.

Als je het programma helemaal wilt beëindigen, klik je op de knop **Quit** bovenaan het Debug Control-venster. Dan wordt het programma meteen beëindigd. Dat is handig als je opnieuw moet debuggen vanaf de start van het programma.

Stappen in, over en uit

Start het programma Drakenrijk met de Debugger. Blijf stappen tot de Debugger bij regel 38 is aangekomen. Zoals je ziet in Afbeelding 7-3, is dit de regel met `toonVerhaaltje()`. Als je nog eens op **Step** klikt, springt de Debugger deze functie binnen en komt hij terecht op regel 5, de eerste regel in de functie `toonVerhaaltje()`. Wat je nu hebt gedaan heet **stappen in**, omdat de Debugger *in* de functie stapt wanneer deze wordt aangeroepen. Dit is iets anders dan 'stappen over', dat hierna wordt uitgelegd.



Afbeelding 7-3: Blijf stappen tot je bij regel 38 komt.

Als de verwerking is aangekomen bij regel 5 en je nog eens op **Step** klikt, zie je ineens code die je niet herkent. Dit is de code van de functie `print()`. De functie `print()` is een van Pythons ingebouwde functies, dus het is niet zo nuttig om hier doorheen te lopen. De eigen functies van Python, zoals `print()`, `input()`, `str()` of `random.randint()` zijn zeer zorgvuldig gecontroleerd op bugs. Je kunt ervan uitgaan dat zij niet de bugs in jouw programma veroorzaken.

Dus verdoe je tijd niet met het stappen door de interne code van de functie `print()`. In plaats van op **Step** te klikken om de code van de functie `print()` te doorlopen, kun je op **Over** klikken. Hiermee stap je over de code heen in de functie `print()`. De code binnen de functie `print()` wordt wél gewoon uitgevoerd en de Debugger pauzeert zodra de verwerking klaar is met `print()`.

Stappen over is een handige manier om iets sneller door de code in een functie te gaan. De Debugger wordt nu gepauzeerd op regel 40, de regel met `grotNummer = kiesGrot()`.

Klik nog een keer op **Step** om de functie `kiesGrot()` in te gaan. Blijf door de code stappen tot regel 15, de aanroep naar `input()`. Het programma wacht nu tot jij een antwoord typt in de interactieve shell, net als je deed toen het programma op de normale manier draaide. Als je nu klikt op **Step**, gebeurt er niets, omdat het programma wacht op de invoer van de gebruiker via het toetsenbord.

Klik weer op het venster van de interactieve shell en typ welk grot je binnen wilt gaan. De knipperende cursor moet op de onderste regel in de interactieve shell staan, voor je kunt typen. Anders wordt de tekst die je typt, niet weergegeven.

Zodra je op ENTER drukt, gaat de Debugger weer verder met het doorlopen van de stappen. Klik op de knop **Out** in het Debug Control-venster. Dit heet **'stappen uit'**, omdat de Debugger nu zoveel regels overslaat als nodig is om de verwerking de functie te laten verlaten waarin hij op dat moment zit. Zodra de verwerking de functie uit is, staat hij op de regel na de regel die de functie aanriep.

Als je bijvoorbeeld klikt op **Out** in de functie `toonVerhaaltje()` op regel 6 ga je verder naar de regel die volgt op de regel met de aanroep naar `toonVerhaaltje()`. 'Stappen uit' bespaart je een boel geklik op Step om een functie uit te komen.

Als je niet in een functie zit en op **Out** klikt, worden alle resterende regels in het programma achter elkaar uitgevoerd. Dat is hetzelfde als klikken op de knop **Go**.

Hier nog even een samenvatting van wat elke knop doet:

- **Go**: voert de rest van de code uit zoals gewoonlijk, of tot er een breekpunt komt. (Breekpunten komen straks.)
- **Step**: verwerkt één regel code. Als de regel een aanroep naar een functie is, stapt de Debugger de functie binnen.
- **Over**: slaat een regel code over. Als de regel een functieaanroep bevat, stapt de Debugger de functie *niet binnen* maar stapt hij *over* de aanroep heen.
- **Out**: blijft regels code overslaan tot de Debugger de functie verlaat waar hij in zat toen op Out werd geklikt. Let op: regels overslaan betekent hier niet dat de code niet wordt uitgevoerd, alleen dat het niet aan je wordt *getoond*.
- **Quit**: beëindigt het programma direct.

De bug vinden

De Debugger kan je helpen de oorzaak van bugs in je programma te vinden. Als voorbeeld is hier een klein programma dat een bug bevat. Het programma toont een willekeurige som aan de gebruiker die hij moet oplossen. Klik in de interactieve shell op File en vervolgens New File om een nieuw venster in de bestandseditor te openen. Typ het volgende programma in dat venster en sla het programma op als *buggy.py*.

```
1. import random
2. getal1 = random.randint(1, 10)
3. getal2 = random.randint(1, 10)
4. print('Wat is ' + str(getal1) + ' + ' + str(getal2) + '?')
5. antwoord = input()
6. if antwoord == getal1 + getal2:
7.     print('Goed zo!')
```

```
8. else:
9.     print('Nee hoor! Het juiste antwoord is ' + str(getal1 + getal2) + '.')
```

Typ het programma precies zoals het hierboven staat, ook als je nu al weet wat de bug is. Voer het programma dan uit door op **F5** te drukken. Dit is een simpele rekenopdracht die twee willekeurige cijfers toont en je vraagt ze bij elkaar op te tellen. Zo zou het eruit kunnen zien, als je het programma start:

```
Wat is 5 + 1?
6
Nee hoor! Het juiste antwoord is 6.
```

Dat is duidelijk een bug! Het programma crasht wel niet, maar het werkt ook niet goed. Het programma zegt dat de gebruiker het fout heeft als hij het goed heeft.

Door het programma uit te voeren met de Debugger erbij, kunnen we de oorzaak van de bug achterhalen. Boven aan het venster van de interactieve shell klik je op **Debug ► Debugger** om het Debug Control-venster te openen. Schakel in dat venster alle vier de selectievakjes in (Stack, Source, Locals en Globals). Hierdoor geeft het Debug Control-venster de meeste informatie weer. Druk dan op **F5** in de bestandseditor om het programma te starten. Het draait nu met de geactiveerde Debugger.

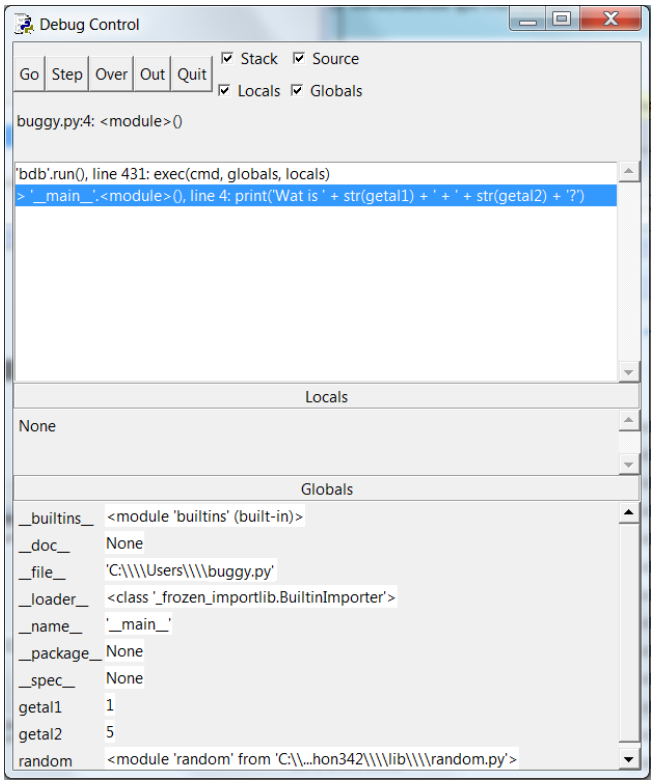
De Debugger start op de regel `import random`. Hier gebeurt niets bijzonders, dus klik gewoon op **Step** om de regel te verwerken. De module `random` wordt toegevoegd aan de sectie Globals, dus is nu geladen.

Klik nog een keer op **Step** om regel 2 te verwerken. Er verschijnt een nieuw bestandseditorvenster met het bestand *random.py*. Je bent nu de functie `randint()` binnengestapt in de module `random`. De ingebouwde functies van Python zullen geen bugs in jouw programma veroorzaken, dus klik op **Out** om de functie `randint()` te verlaten en terug te keren naar jouw programma. Sluit het venster van *random.py*.

De volgende keer kun je op **Over** klikken om de functie `randint()` helemaal over te slaan in plaats van erin te stappen. Regel 3 bevat ook een aanroep van `randint()`. Klik op **Over** om deze code over te slaan.

Regel 4 is een aanroep naar `print()` om de speler de willekeurige getallen te tonen. Je weet al welke getallen het programma gaat weergeven, zelfs voordat dat gebeurt! Kijk maar naar de sectie Globals van het Debug Control-venster. Je ziet de variabelen `getal1` en `getal2` en de integerwaarden die ze momenteel bevatten.

De variabele `getal1` heeft (bij mij) de waarde `4` en de variabele `getal2` heeft de waarde `8`. Bij jou staat waarschijnlijk iets anders. Wanneer je op **Step** klikt, geeft het programma deze waarden in de string van de aanroep naar `print()`. De functie `str()` zet de integerwaarde hiervan om in een string. Toen ik de Debugger liet draaien, zag het eruit zoals in Afbeelding 7-4. (Jouw willekeurige getallen zijn waarschijnlijk anders.)



Afbeelding 7-4: getal1 heeft de waarde 4 en getal2 heeft de waarde 8.

Als je klikt op **Step** op regel 5 wordt `input()` uitgevoerd. De Debugger wacht tot de speler een antwoord typt in het programma. Typ het juiste antwoord (in mijn geval zou dat 12 zijn) in de interactieve shell. De Debugger gaat verder naar regel 6.

Regel 6 is een `if`-statement. De voorwaarde is dat de waarde van `antwoord` gelijk moet zijn aan de som van `getal1` en `getal2`. Als de voorwaarde `True` is, gaat de Debugger naar regel 7. Als de voorwaarde `False` is, gaat de Debugger naar regel 9. Klik nog een keer op `Step` om te zien waar de Debugger heen gaat.

Hij gaat naar regel 9! Hoe kan dat? De voorwaarde in de `if`-statement wordt geëvalueerd als `False`. Kijk eens naar de waarden van de variabelen `getal1`, `getal2` en `antwoord`. Je ziet dat `getal1` en `getal2` integers zijn, dus de som van die twee zou ook een integer moeten zijn. Maar `antwoord` is een string.

Dat betekent dat `antwoord == getal1 + getal2` geëvalueerd zal zijn als `'12' == 12`. Maar we weten dat een stringwaarde en een integerwaarde nooit gelijk aan elkaar kunnen zijn. Vandaar dat de voorwaarde als `False` werd geëvalueerd.

En daar hebben we de bug te pakken. De bug is dat er `antwoord` in de code stond, terwijl er `int(antwoord)` had moeten staan. We waren vergeten de string om te zetten in een integer. Wijzig regel 6 met `int(antwoord) == getal1 + getal2` in plaats van `antwoord == getal1 + getal2` en start het programma opnieuw.

```
Wat is 2 + 3?
5
Goed zo!
```

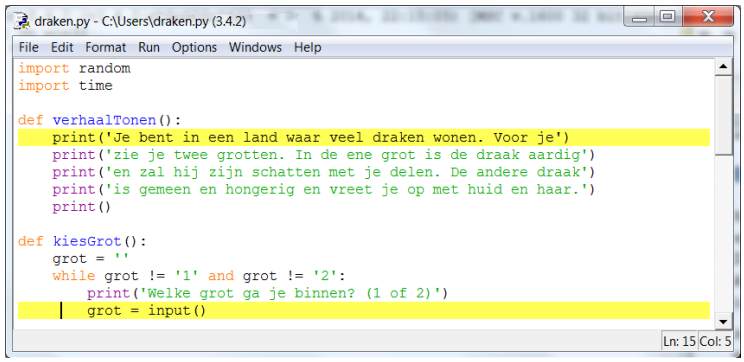
Deze keer werkte het programma goed. Start het nog een keertje en voer nu opzettelijk een verkeerd antwoord in. Nu heb je het programma goed getest. Je hebt het programma nu 'gedebugged'! Vergeet niet: de computer voert je programma's precies uit zoals je ze hebt getypt, ook als wat je hebt getypt niet helemaal is wat je had bedoeld.

Breekpunten

Stap voor stap door de code gaan, gaat soms gewoon te langzaam. Vaak wil je dat je programma op normale snelheid wordt uitgevoerd tot je bij een regel komt die je niet helemaal vertrouwt. Dan kun je een **breekpunt** instellen op de regel waar je de Debugger wilt inschakelen. Als je denkt dat er misschien een probleem is met je code op regel 17, kun je daar een breekpunt zetten (of misschien een paar regels eerder, voor de zekerheid).

Als de verwerking bij die regel komt, neemt de Debugger het over. Dan kun je verder stap voor stap door de regels gaan om te zien wat er precies gebeurt. Door op **Go** te klikken, gaat het programma weer gewoon verder, tot het bij een volgende breekpunt komt of tot het einde van het programma wordt bereikt.

Je stelt een breekpunt in door met de rechtermuisknop te klikken op de regel in de bestandseditor en **Set Breakpoint** (Breekpunt instellen) te selecteren in het menu dat verschijnt. In de bestandseditor wordt die regel dan geel gemarkeerd. Je kunt zoveel breekpunten instellen als je wilt. Als je het breekpunt weer wilt verwijderen, rechtsklik je op de regel en selecteer je **Clear Breakpoint** (Breekpunt wissen) in het menu dat verschijnt.



Afbeelding 7-5: De bestandseditor met twee ingestelde breekpunten.

Voorbeeld met breekpunten

Hier is een programma dat het opgooien van een muntje simuleert door `random.randint(0, 1)` aan te roepen. Als de functie de integer `1` retourneert, noemen we dat 'kop' en als 0 wordt geretourneerd, noemen we het 'munt'. De variabele `gegooid` houdt bij hoe vaak we een muntje hebben opgegooid. De variabele `kop` houdt bij hoe vaak het 'kop' was.

Het programma gaat duizend keer (!) een muntje opgooien. Een mens zou hier meer dan een uur voor nodig hebben, maar een computer kan het in 1 seconde! Typ de volgende code in de bestandseditor en sla het op als *muntGooi.py*. Je kunt deze code ook downloaden van http://invpy.com/muntGooi.py

Als je een foutmelding krijgt nadat je deze code hebt ingetypt, vergelijk jouw code dan met de code van het boek met behulp van de online diff tool op http://invpy.com/diff/muntGooi.

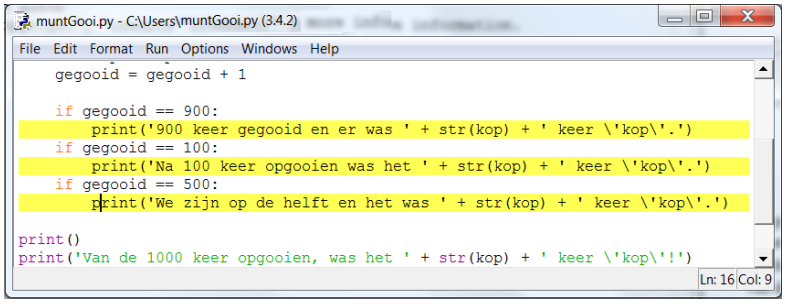
muntGooi.py

```
 1. import random
 2. print('Ik gooi 1000 keer een muntje op. Raad eens hoe vaak het \'kop\' zal
zijn? (Druk op Enter om te beginnen.)')
 3. input()
 4. gegooid = 0
 5. kop = 0
 6. while gegooid < 1000:
 7.     if random.randint(0, 1) == 1:
 8.         kop = kop + 1
 9.     gegooid = gegooid + 1
10.
11.     if gegooid == 900:
12.         print('900 keer gegooid en het was ' + str(kop) + ' keer \'kop\'.')
13.     if gegooid == 100:
14.         print('Na 100 keer opgooien was het ' + str(kop) + ' keer \'kop\'.')
15.     if gegooid == 500:
16.         print('We zijn op de helft en het was ' + str(kop) + ' keer \'kop\'.')
17.
18. print()
19. print('Van de 1000 keer opgooien, was het ' + str(kop) + ' keer \'kop\'!')
20. print('Had je het goed?')
```

Het programma gaat best snel. Het duurt langer voor de gebruiker op ENTER heeft gedrukt, dan om 1000 keer een muntje op te gooien voor de computer. Maar misschien wil je de muntjes één voor één opgegooid zien worden. Klik in de interactieve shell op **Debug ► Debugger** om het Debug Control-venster te openen. Druk dan op **F5** om het programma uit te voeren.

Het programma start in de Debugger op regel 1. Druk één keer op **Step** en twee keer op **Over** in het Debug Control-venster om de eerste drie regels te verwerken. Je ziet dat de knoppen grijs worden en niet meer beschikbaar zijn omdat de functie `input()` is aangeroepen en de interactieve shell op een reactie van de gebruiker wacht. Klik op het venster van de interactieve shell en druk op ENTER. (Let erop dat je onder de tekst klikt in de interactieve shell, anders ontvangt IDLE je toetsaanslagen misschien niet.)

Je kunt nog een paar keer op **Step** klikken, maar je zult wel inzien dat het tamelijk lang duurt om het hele programma te doorlopen. Je kunt beter een breekpunt instellen op de regels 12, 14 en 16. In de bestandseditor worden deze regels gemarkeerd zoals in Afbeelding 7-6.



Afbeelding 7-6: Er zijn drie breekpunten ingesteld.

Na het instellen van de breekpunten klik je op **Go** in het Debug Control-venster. Het programma wordt uitgevoerd op normale snelheid tot het bij het volgende breekpunt komt. Wanneer `gegooid` op 100 staat, is de voorwaarde van de `if`-statement op regel 13 `True`. Hierdoor wordt regel 14 (waar een

breekpunt staat) uitgevoerd, waardoor de Debugger de opdracht krijgt het programma te pauzeren en de controle over te nemen. Kijk naar het Debug Control-venster in de sectie Globals om te zien wat de waarden van `gegooid` en `kop` nu zijn.

Klik weer op **Go** en het programma gaat verder tot het bij het breekpunt op regel 16 komt. Opnieuw kun je bekijken hoe de waarden van `gegooid` en `kop` zijn veranderd.

Als je nog een keer op **Go** klikt, gaat de verwerking verder tot het volgende breekpunt, op regel 12.

Samenvatting

Programma's schrijven is maar één onderdeel van programmeren. Het andere deel is zorgen dat de code die je schreef, goed werkt. Met Debuggers kun je de code stap voor stap doorlopen. Je kunt onderzoeken welke regels worden verwerkt in welke volgorde en welke waarden er in de variabelen zitten. Als dit te langzaam gaat, kun je breekpunten instellen om de Debugger alleen op bepaalde regels die je wilt onderzoeken, te laten pauzeren.

Een Debugger is sowieso een goede manier om de werking van een programma te leren begrijpen. Dit boek biedt uitleg van alle code in alle games, maar de Debugger kan je helpen zelf nog meer te ontdekken.

Hoofdstuk 8
STROOMDIAGRAMMEN

In dit hoofdstuk behandelen we:
- Hoe je Galgje moet spelen
- ASCII-tekeningen
- Een programma ontwerpen met behulp van stroomdiagrammen

In dit hoofdstuk ga je het spel Galgje ontwerpen. Deze game is ingewikkelder dan ons vorige spel, maar ook leuker. Omdat de game moeilijker is, moet je het eerst zorgvuldig plannen aan de hand van een stroomdiagram. Dat gaan we hier doen. In het volgende hoofdstuk ga je dan de daadwerkelijke code voor Galgje schrijven.

Hoe je Galgje moet spelen

Galgje is een spel voor twee personen dat meestal met potlood en papier wordt gespeeld. De ene speler bedenkt een woord en zet streepjes neer op het papier voor elke letter in het woord. De andere speler probeert de letters te raden die in het woord voorkomen.

Als ze het goed raden, zet speler 1 de letter op de juiste plek. Als ze het fout raden, tekent speler 1 een lichaamsdeel aan de galg. Als speler 2 alle letters in het woord heeft geraden vóór de tekening van het lichaam aan de galg af is, hebben zij gewonnen. Maar als ze het niet op tijd raden, hangen ze!

Voorbeelduitvoer van Galgje

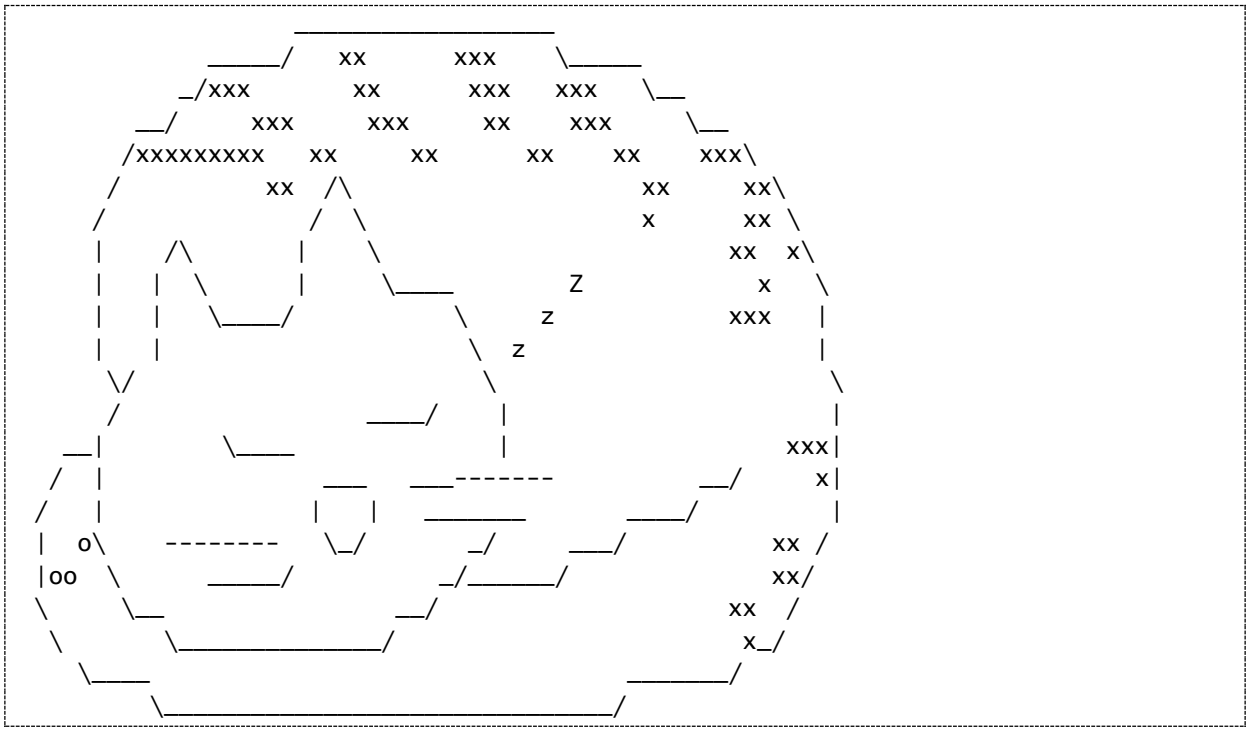
Hier zie je een voorbeeld van wat de speler zou zien als ze het spel Galgje doen, dat jij straks gaat schrijven. De tekst die de speler typt, wordt **vet** weergegeven.

```
G A L G J E
  +---+
  |   |
      |
      |
      |
      |
=========
Foute letters:

_ _ _
Raad een letter.
a
  +---+
  |   |
      |
      |
      |
      |
=========
Foute letters:
_ a _
```

```
Raad een letter.
o
  +---+
  |   |
  O   |
      |
      |
      |
=========
Foute letters: o
_ a _
Raad een letter.
r
  +---+
  |   |
  O   |
  |   |
      |
      |
=========
Foute letters: or
_ a _
Raad een letter.
t
  +---+
  |   |
  O   |
  |   |
      |
      |
=========
Foute letters: or
_ a t
Raad een letter.
a
Die letter heb je al gehad. Kies nog eens.
Raad een letter.
k
Ja! Het geheime woord is "kat"! Jij hebt gewonnen!
Wil je nog een keer spelen? (ja of nee)
nee
```

ASCII-tekeningen

De graphics voor galgje zijn tekens die je op het toetsenbord vindt, afgebeeld op het computerscherm. Deze soort graphics noemen we **ASCII-tekeningen** (ASCII spreek je uit als 'askie'). Hier is bijvoorbeeld een kat getekend met een ASCII-tekening:

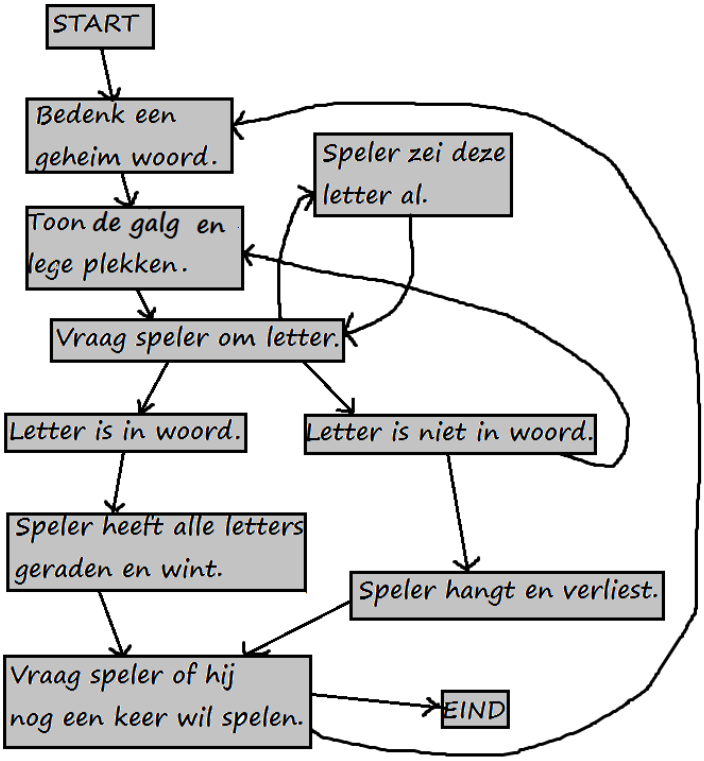


Een programma ontwerpen met een stroomdiagram

Deze game is een beetje ingewikkelder dan de spellen die we tot nu toe hebben geschreven, dus we moeten even goed nadenken hoe het in elkaar moet zitten. Eerst ga je daarom een stroomdiagram maken. (Je zag er al een aan het eind van het hoofdstuk over Drakenrijk). Hiermee kun je visualiseren wat het programma moet doen. In dit hoofdstuk leer je wat stroomdiagrammen zijn en waarom programmeurs die gebruiken. In het hoofdstuk hierna gaan we het hebben over de broncode van Galgje.

Een **stroomdiagram** is een diagram dat een serie stappen weergeeft als vakjes die door pijlen met elkaar zijn verbonden. Elk vakje vertegenwoordigt een stap en de pijlen laten zien welke stappen leiden tot welke andere stappen. Zet je vinger op het vakje 'Start' van het diagram en doorloop het programma door de pijlen te volgen naar de andere vakken tot je bij het vak 'Eind' komt.

In Afbeelding 8-1 zie je het volledige stroomdiagram voor Galgje. Je mag alleen in de richting van de pijl gaan, van het ene vak naar het andere. Je kunt niet terug, tenzij er een tweede pijl is die de andere kant op gaat, zoals in het vak 'Speler zei deze letter al'.

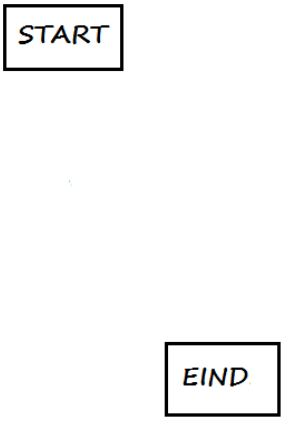


Afbeelding 8-1: Het volledige stroomdiagram dat weergeeft wat er gebeurt in het spel Galgje.

Het is natuurlijk niet *verplicht* om een stroomdiagram te maken. Je kunt ook gewoon beginnen met het schrijven van code. Maar heel vaak zul je ontdekken, terwijl je aan het programmeren bent, dat er dingen zijn die je moet toevoegen of wijzigen. Misschien moet je dan wel een heel stuk van je code weggooien. Dat zou zonde zijn van al je werk. Om dit te voorkomen, is het aan te raden om van te voren, vóór je begint te schrijven, je werk goed te plannen.

Het stroomdiagram maken

Jouw stroomdiagrammen hoeven er niet net zo uit te zien als dit diagram. Zo lang **jij** je stroomdiagram maar begrijpt, is dat genoeg om van nut te zijn voor het coderen. Hier zie je een stroomdiagram dat begint met alleen een vak 'Start' en een vak 'Eind', zoals getoond in Afbeelding 8-2:



Afbeelding 8-2: Begin je stroomdiagram met een vak Start en een vak Eind.

Bedenk nu wat er gebeurt als je Galgje speelt. Eerst bedenkt de computer een geheim woord. Dan gaat de speler letters raden. Teken vakjes voor deze gebeurtenissen, zoals in Afbeelding 8-3. De vakken die nieuw zijn in een stroomdiagram worden meestal getekend met een stippellijn eromheen.

De pijlen tonen de volgorde in het programma. Dus eerst bedenkt het programma een geheim woord en daarna wordt de speler gevraagd een letter te raden.

Afbeelding 8-3: Teken de eerste twee stappen van Galgje als vakken met een beschrijving erin.

Maar het spel eindigt natuurlijk niet als de speler een letter heeft geraden. Dan moet er gecontroleerd worden of die letter in het woord voorkomt of niet.

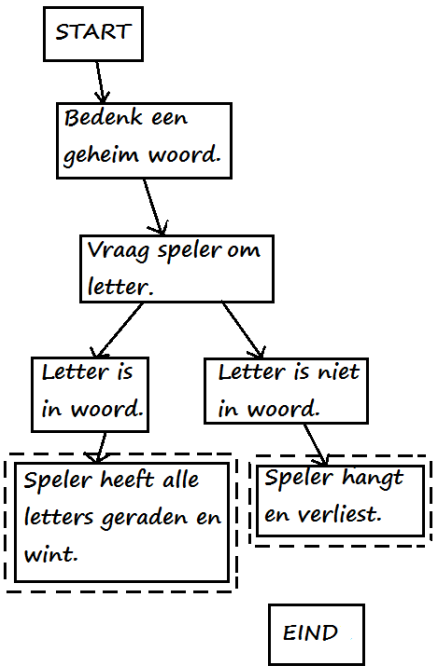
Vertakking uit een vak in een stroomdiagram

Er zijn twee mogelijkheden: ofwel de letter komt in het woord voor, of niet. Je voegt nu twee nieuwe vakken toe aan het stroomdiagram, een voor elke mogelijkheid. Nu heb je een vertakking in het stroomdiagram gemaakt, zoals je ziet in Afbeelding 8-4:

Afbeelding 8-4: De vertakking wordt gevormd door twee pijlen die naar verschillende vakken gaan.

Als de letter voorkomt in het geheime woord, moet je controleren of de speler alle letters al heeft geraden en het spel heeft gewonnen. Als de letter niet in het geheime woord voorkomt, wordt er een lichaamsdeel toegevoegd aan de galg. Voeg vakken toe voor deze gebeurtenissen.

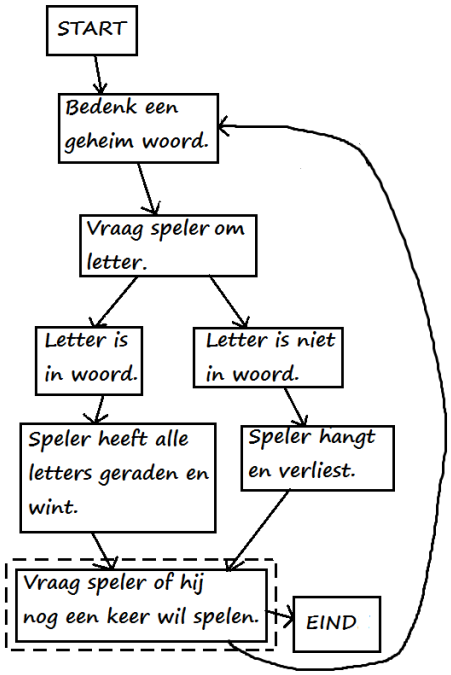
Je hebt **geen** pijl nodig van 'Letter is in woord' naar 'Speler hangt en verliest', omdat het niet mogelijk is om te verliezen als de speler een goede letter raadt. Het is ook niet mogelijk om te winnen als de speler fout raadt, dus die pijl hoef je ook niet te tekenen. Het stroomdiagram ziet er nu uit zoals in Afbeelding 8-5.



Afbeelding 8-5: Na de vertakking gaan de stappen verder op hun aparte pad.

Het spel beëindigen of opnieuw starten

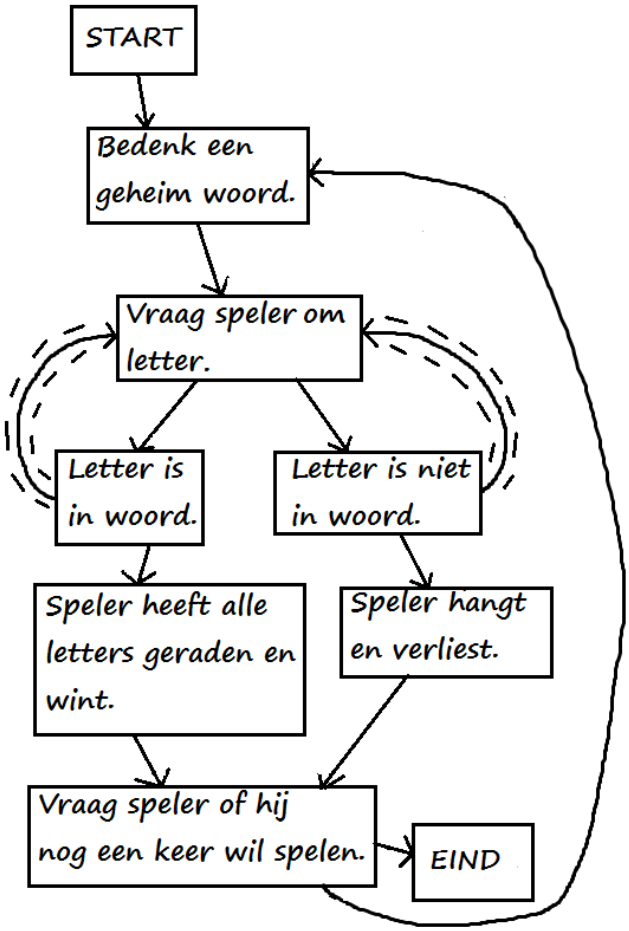
Zodra de speler heeft gewonnen of verloren, vraag je hun of ze nog een keer willen spelen. Als de speler nee zegt, stopt het programma. Als het programma niet stopt, wordt er een nieuw woord verzonnen. Dit zie je in Afbeelding 8-6.



Afbeelding 8-6: Het stroomdiagram vertakt weer als de speler wordt gevraagd of ze nog een keer willen spelen.

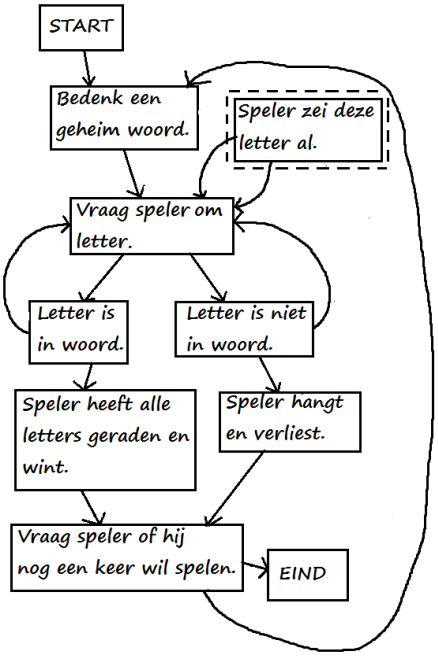
Opnieuw raden

De speler raadt natuurlijk niet maar één keer. Ze moeten blijven raden, tot ze winnen of verliezen. Je tekent nu twee nieuwe pijlen, zoals in Afbeelding 8-7.



Afbeelding 8-7: De nieuwe pijlen (met stippellijnen eromheen) laten zien dat de speler nog een keer kan raden.

Maar wat gebeurt er als de speler dezelfde letter nog een keer noemt? In plaats van dat ze dan verliezen, is het aardiger om ze een andere letter te laten kiezen. Dit nieuwe vak zie je in Afbeelding 8-8.

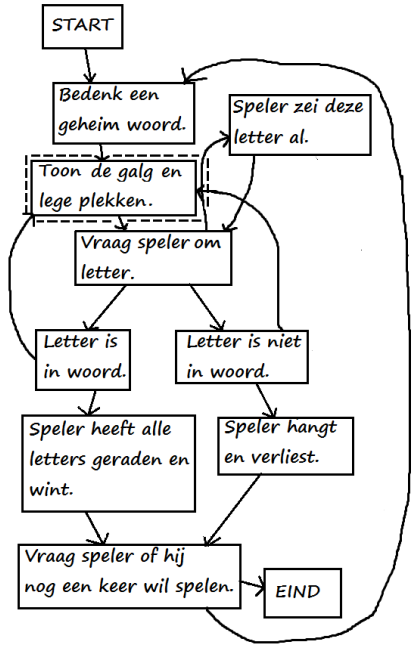


Afbeelding 8-8: Een stap toevoegen voor het geval de speler een letter noemt die hij al gehad heeft.

Feedback aan de speler bieden

De speler moet natuurlijk weten of ze een letter goed of fout raden. Het programma moet de galg tonen en het geheime woord met streepjes voor de letters die ze nog niet hebben geraden. Deze illustratie laat zien of ze het spel gaan winnen of verliezen.

Elke keer dat de speler aan letter raadt, moet deze informatie worden bijgewerkt. Voeg een vak aan je stroomdiagram toe met 'Toon de galg en streepjes aan de speler' tussen de vakken 'Bedenk het geheime woord' en 'Vraag de speler een letter te raden'. Deze vakken zie je in Afbeelding 8-9.



Afbeelding 8-9: Voeg 'Toon de galg en lege plekken' toe om feedback te geven aan de speler.

Dat ziet er goed uit! Dit stroomdiagram laat alles zien wat er in Galgje kan gebeuren, en in welke volgorde het gaat. Wanneer je je eigen games ontwerpt, kan een stroomdiagram helpen op een rijtje te zetten wat er precies moet gebeuren in je code.

Samenvatting

Misschien denk je dat het wel een beetje veel werk is om een stroomdiagram te gaan tekenen voor je gaat programmeren. Mensen willen tenslotte gamen, niet naar stroomdiagrammen kijken! Maar bedenk dat het veel makkelijker is om wijzigingen te maken en problemen op te merken als je eerst goed nadenkt over hoe het programma in elkaar moet zitten, vóór je de code gaat schrijven.

Als je gewoon pardoes in het schrijven van code springt, kom je later misschien problemen tegen waar je nog niet aan dacht en moet je de code die je hebt geschreven helemaal veranderen. En elke keer dat je je code verandert, loop je het risico dat je weer nieuwe bugs introduceert omdat je te veel of juist te weinig hebt veranderd. Het is een beter idee om goed te weten wat je gaat maken, voordat je aan de slag gaat.

Hoofdstuk 9
GALGJE

In dit hoofdstuk behandelen we:
- Methoden
- De methode `append()` voor lijsten
- De methoden `lower()`, `upper()` en `split()` voor strings
- De methode `reverse()` voor lijsten
- De functies `range()` en `list()`
- `for`-lussen
- `elif`-statements
- De methoden `startswith()` en `endswith()` voor strings

De game in dit hoofdstuk introduceert allerlei nieuwe begrippen, maar maak je geen zorgen. Je gaat eerst een beetje stoeien met deze programmatische begrippen in de interactieve shell. Je gaat leren over methoden. Dit zijn functies die zijn gekoppeld aan waarden. Je leert ook een nieuw type lus, de `for`-lus, en een nieuw datatype, de Lijst. Als je weet wat deze begrippen inhouden, wordt het programmeren van Galgje veel makkelijker.

Broncode van Galgje

De game in dit hoofdstuk is een stuk langer dan de vorige games, maar veel ervan gaat over de ASCII-art voor de tekening van de galg. Typ de volgende tekst in de bestandseditor en sla het op als *galgje.py*.

galgje.py

```
 1. import random
 2. GALGTEKENING = ['''
 3.
 4.   +---+
 5.   |   |
 6.       |
 7.       |
 8.       |
 9.       |
10. =========''', '''
11.

12.   +---+
```

```
  |   |
  O   |
      |
      |
      |
=========''', '''

  +---+
  |   |
  O   |
  |   |
      |
      |
=========''', '''

  +---+
  |   |
  O   |
 /|   |
      |
      |
=========''', '''

  +---+
  |   |
  O   |
 /|\  |
      |
      |
=========''', '''

  +---+
  |   |
  O   |
 /|\  |
 /    |
      |
=========''', '''

  +---+
  |   |
  O   |
 /|\  |
 / \  |
      |
=========''']
```

```
 59. woorden = 'aap adelaar baviaan beer bever cobra cougar coyote das duif
eend eland forel fret gans geit haai hagedis havik hert hond kalkoen kameel kat
kikker konijn kraai lama leeuw luiaard mier mol mossel muilezel muis neushoorn
ooievaar otter pad panda papegaai python raaf ram rat salamander schaap
schildpad slang spin stinkdier tijger uil vleermuis vos walvis wezel wolf
wombat zalm zeehond zwaan zebra'.split()
 60.
 61. def haalWillekeurigWoord(woordenLijst):
 62.     # Deze functie retourneert een willekeurige string uit de doorgegeven
lijst met strings.
 63.     woordIndex = random.randint(0, len(woordenLijst) - 1)
 64.     return woordenLijst[woordIndex]
 65.
 66. def toonGalg(GALGTEKENING, fouteLetters, goedeLetters, geheimWoord):
 67.     print(GALGTEKENING[len(fouteLetters)])
 68.     print()
 69.
 70.     print('Foute letters:', end=' ')
 71.     for letter in fouteLetters:
 72.         print(letter, end=' ')
 73.     print()
 74.
 75.     streepjes = '_' * len(geheimWoord)
 76.
 77.     for i in range(len(geheimWoord)): # vervang streepjes door de goed
geraden letters
 78.         if geheimWoord[i] in goedeLetters:
 79.             streepjes = streepjes[:i] + geheimWoord[i] + streepjes[i+1:]
 80.
 81.     for letter in streepjes: # toon het geheime woord met spaties tussen
elke letter
 82.         print(letter, end=' ')
 83.     print()
 84.
 85. def haalPoging(alGeraden):
 86.     # Retourneert de letter die de speler heeft getypt. Deze functie zorgt
ervoor dat de speler één letter heeft getypt, en niet iets anders.
 87.     while True:
 88.         print('Raad een letter.')
 89.         poging = input()
 90.         poging = poging.lower()
 91.         if len(poging) != 1:
 92.             print('Je mag maar 1 letter typen.')
 93.         elif poging in alGeraden:
 94.             print('Die letter heb je al gehad. Probeer een andere
letter.')
 95.         elif poging not in 'abcdefghijklmnopqrstuvwxyz':
```

```
 96.             print('Je moet een LETTER typen.')
 97.         else:
 98.             return poging
 99.
100. def nogmaalsSpelen():
101.     # Deze functie retourneert True als de speler nog een keer wil spelen,
anders is het False.
102.     print('Wil je nog een keer spelen? (ja of nee)')
103.     return input().lower().startswith('j')
104.
105.
106. print('G A L G J E')
107. fouteLetters = ''
108. goedeLetters = ''
109. geheimWoord = haalWillekeurigWoord(woorden)
110. klaarMetSpel = False
111.
112. while True:
113.     toonGalg(GALGTEKENING, fouteLetters, goedeLetters, geheimWoord)
114.
115.     # Laat de speler een letter typen.
116.     poging = haalpoging(fouteLetters + goedeLetters)
117.
118.     if poging in geheimWoord:
119.         goedeLetters = goedeLetters + poging
120.
121.         # Controleer of de speler heeft gewonnen
122.         alleLettersGevonden = True
123.         for i in range(len(geheimWoord)):
124.             if geheimWoord[i] not in goedeLetters:
125.                 alleLettersGevonden = False
126.                 break
127.         if alleLettersGevonden:
128.             print('Ja! Het geheime woord was "' + geheimWoord + '"! Jij
hebt gewonnen!')
129.             klaarMetSpel = True
130.     else:
131.         fouteLetters = fouteLetters + poging
132.
133.         # Controleer of de speler te vaak heeft geraden en dus verloren
134.         if len(fouteLetters) == len(GALGTEKENING) - 1:
135.             toonGalg(GALGTEKENING, fouteLetters, goedeLetters,
geheimWoord)
136.             print('Je kunt niet meer raden!\nNa ' + str(len(fouteLetters))
+ ' keer fout geraden te hebben en ' + str(len(goedeLetters)) + ' keer goed
geraden te hebben, was het woord "' + geheimWoord + '"')
137.             klaarMetSpel = True
```

```
138.
139.     # Vraag de speler of ze het nog een keer willen proberen (maar alleen
als het spel klaar is).
140.     if klaarMetSpel:
141.         if nogmaalsSpelen():
142.             fouteLetters = ''
143.             goedeLetters = ''
144.             klaarMetSpel = False
145.             geheimWoord = haalWillekeurigWoord(woorden)
146.         else:
147.             break
```

Hoe de code werkt

```
 1. import random
```

Het programma Galgje selecteert een willekeurig geheim woord uit een lijst geheime woorden. Deze functionaliteit wordt geboden door de module `random`.

```
 2. GALGTEKENING = ['''
 3.
 4.   +---+
 5.   |   |
 6.       |
 7.       |
 8.       |
 9.       |
10. =========''', '''

...de rest van de code is te veel om hier te tonen...
```

Deze toewijzingsstatement strekt zich uit over meerdere regels in de broncode. De statement eindigt pas op regel 58. Om je te helpen begrijpen wat deze code betekent, moet je eerst iets weten over multiregel-strings.

Multiregel-strings

Tot nu toe pasten alle strings op één regel en werden ze voorafgegaan en gevolgd door een aanhalingsteken. Maar als je drie enkele aanhalingstekens aan het begin en eind van de string plaatst, geef je daarmee aan dat de string meer dan één regel beslaat:

```
>>> briefje = '''Lieve Ellen,
Aan het eind van deze maand kom ik weer naar huis. Tot dan!
```

```
Groetjes,
Rob'''
>>> print(briefje)
Lieve Ellen,
Aan het eind van deze maand kom ik weer naar huis. Tot dan!
Groetjes,
Rob
```

Dit zijn **multiregel-strings**. In een multiregel-string zijn de nieuweregeltekens al in de string opgenomen. Je hoeft het escapeteken `\n` dus niet te gebruiken. Hierdoor is de code gemakkelijker leesbaar als er veel tekst is.

Constanten

De variabelenaam `GALGTEKENING` heeft allemaal hoofdletters. Dit is een conventie in het programmeren en betekent dat de variabele een constante is. **Constanten** zijn variabelen die één waarde krijgen die nooit meer verandert. De waarde blijft 'constant'. Hoewel je in principe de waarde `GALGTEKENING` wel zou kunnen wijzigen, net zoals dat bij alle andere variabelen kan, herinneren de hoofdletters je eraan dat je dat niet gaat doen. Omdat de variabele `GALGTEKENING` nooit hoeft te veranderen, wordt hij aangemerkt als constante.

Zoals bij alle conventies geldt: je *hoeft* je hier niet aan te houden. Maar als je het wel doet, is het makkelijker voor andere programmeurs om je code te lezen.

Lijsten

Een **lijst**waarde is een soort waarde die meerdere andere waarden kan bevatten. Probeer dit eens te typen in de interactieve shell: `['appels', 'sinaasappels', 'HALLO WERELD']`.

```
>>> ham = ['appels', 'sinaasappels', 'HALLO WERELD']
>>> ham
['appels', 'sinaasappels', 'HALLO WERELD']
```

Deze lijstwaarde bevat drie stringwaarden. Net zoals dat bij andere soorten waarden gaat, kun je deze hele lijst opslaan in een variabele. Als je de lijstwaarde in je code typt, begin je met een vierkante haak openen [en eindig je met de vierkante haak sluiten].

De waarden scheid je van elkaar met een komma. Deze waarden noemen we ook wel de **items**.

Typ `dieren = ['aardvarken', 'albert', 'antiloop', 'miereneter']` maar eens in de interactieve shell om een lijst op te slaan in de variabele `dieren`. De vierkante haken kunnen ook worden gebruikt om naar een bepaald item in de lijst te gaan. Typ `dieren[0]` of `dieren[1]` of `dieren[2]` of `dieren[3]` in de interactieve shell en kijk wat het resultaat is.

```
>>> dieren = ['aardvarken', 'albert', 'antiloop', 'miereneter']
>>> dieren[0]
'aardvarken'
>>> dieren[1]
'albert'
>>> dieren[2]
'antiloop'
>>> dieren[3]
'miereneter'
```

Het cijfer tussen de vierkante haken is het **indexnummer**. Een indexnummer wijst naar een item op volgorde. In Python heeft het eerste item in een lijst altijd het indexnummer `0`. Het tweede item heeft de index `1`, het derde item heeft het indexnummer `2` enzovoort. Dat is iets waar je even aan moet wennen!

Lijsten zijn handig omdat je er meerdere waarden in kunt opslaan, zonder dat je voor elk item een aparte variabele nodig hebt. Je gaat gewoon naar de variabele die je wilt hebben via het indexnummer. Als we geen lijsten zouden hebben, zou de code er als volgt moeten uitzien:

```
>>> dieren1 = 'aardvarken'
>>> dieren2 = 'albert'
>>> dieren3 = 'antiloop'
>>> dieren4 = 'miereneter'
```

Als je honderden of duizenden strings hebt, zou je heel lang bezig zijn. Een lijst kan veel gemakkelijker een aantal waarden bevatten. Met behulp van de vierkante haken kun je met de items in je lijst net zo omgaan als met andere waarden. Probeer maar eens `dieren[0] + dieren[2]` in de interactieve shell te typen:

```
>>> dieren[0] + dieren[2]
'aardvarkenalbert'
```

De evaluatie gaat als volgt:

```
dieren[0] + dieren[2]
          ▼
'aardvarken' + dieren[2]
          ▼
'aardvarken' + 'albert'
          ▼
  'aardvarkenalbert'
```

Als je een indexnummer noemt dat te hoog is, krijg je een **indexfout** die je programma laat crashen. Probeer `dieren[9999]` in de interactieve shell te typen:

```
>>> dieren = ['aardvarken', 'albert', 'antiloop', 'miereneter']
>>> dieren[9999]
Traceback (most recent call last):
File "", line 1, in
dieren[99]
IndexError: list index out of range
```

De waarden van items in een lijst wijzigen met indextoewijzing

Je kunt met de vierkante haken ook de waarde wijzigen van een item in een lijst. Typ maar eens `dieren[1] = 'MIERENETER'`. En typ dan `dieren` om de lijst te bekijken.

```
>>> dieren = ['aardvarken', 'albert', 'antiloop', 'miereneter']
>>> dieren[1] = 'ALBERT'
>>> dieren
['aardvarken', 'ALBERT', 'antiloop', 'miereneter']
```

De nieuwe string `'ALBERT'` overschrijft het tweede item in de lijst `dieren`. Dus `dieren[1]` geeft je niet alleen het tweede item in de lijst als resultaat, het is ook mogelijk om het aan de linkerkant van een toewijzingsstatement te gebruiken om een bepaalde waarde aan `dieren[1]` (het tweede item in de lijst) toe te wijzen.

Lijsten concateneren

Je kunt meerdere lijsten tot één lijst samenvoegen met behulp van de operator `+`, net zoals je dat bij strings kunt doen. Lijsten aan elkaar plakken met de operator `+` heet **lijsten concateneren**. Typ `[1, 2, 3, 4] + ['appels', 'sinaasappels'] + ['Ellen', 'Rob']` maar eens in de interactieve shell:

```
>>> [1, 2, 3, 4] + ['appels', 'sinaasappels'] + ['Ellen', 'Rob']
[1, 2, 3, 4, 'appels', 'sinaasappels', 'Ellen', 'Rob']
```

`['appels'] + ['sinaasappels']` retourneert `['appels', 'sinaasappels']`. Maar `['appels'] + 'sinaasappels'` levert een fout op. Je kunt wel twee lijstwaarden aan elkaar toevoegen, maar je kunt niet een lijstwaarde aan een stringwaarde toevoegen. Als je waarden aan een lijst wilt toevoegen die zelf niet in een lijst zitten, moet je de methode `append()` gebruiken (waar we het later over gaan hebben).

De operator `in`

De operator `in` vertelt je of een waarde voorkomt in een lijst of niet. Expressies waar de operator `in` in voorkomt, retourneren een Boole-waarde: `True` als de waarde in de lijst voorkomt en `False` als dat niet zo is. Probeer `'antiloop' in dieren` te typen in de interactieve shell:

```
>>> dieren = ['aardvarken', 'albert', 'antiloop', 'miereneter']
>>> 'antiloop' in dieren
True
```

De expressie `'antiloop' in dieren` retourneert `True` omdat de string `'antiloop'` voorkomt in de lijst `dieren`. Dit woord heeft het indexnummer `2`.

Maar als je de expressie `'mier' in dieren` typt, is het resultaat `False` omdat de string `'mier'` niet in de lijst staat.

```
>>> dieren = ['aardvarken', 'albert', 'antiloop', 'miereneter']
>>> 'antiloop' in dieren
True
>>> 'mier' in dieren
False
```

De operator `in` werkt ook voor strings. Je kunt kijken of een string voorkomt in een andere string. Typ `'hallo' in 'Ellen zei hallo tegen Rob.'` in de interactieve shell. Deze expressie retourneert `True`.

```
>>> 'hallo' in 'Ellen zei hallo tegen Rob.'
True
```

Items uit een lijst verwijderen met `del`-statements

De `del`-statement kan een item uit een lijst verwijderen als je de indexpositie weet. 'del' is een afkorting van 'delete', oftewel 'verwijderen'. Probeer eens een lijst met nummers te maken door het volgende te typen: `ham = [2, 4, 6, 8, 10]` en dan `del ham[1]`. Typ `ham` om de inhoud van de lijst te bekijken:

```
>>> ham = [2, 4, 6, 8, 10]
>>> del ham[1]
>>> ham
[2, 6, 8, 10]
```

Als je een item op indexnummer 1 verwijdert, krijgt het item dat op indexnummer 2 stond, nu de waarde indexnummer 1. Het item dat op indexnummer 3 stond, krijgt de nieuwe waarde indexnummer 2. Alle volgende nummers gaan dus 1 nummer naar beneden.

Je kunt nog een keer `del ham[1]` typen en nog eens en nog eens om steeds items uit de lijst te verwijderen

```
>>> ham = [2, 4, 6, 8, 10]
>>> del ham[1]
```

```
>>> ham
[2, 6, 8, 10]
>>> del ham[1]
>>> ham
[2, 8, 10]
>>> del ham[1]
>>> ham
[2, 10]
```

De `del`-statement is een statement, geen functie of operator. Er zijn geen haakjes en de statement retourneert ook geen waarde.

Lijsten met lijsten

Lijsten zijn een datatype dat andere waarden kan bevatten. Maar lijsten kunnen ook zelf een item zijn in een andere lijst. Stel dat je een boodschappenlijst hebt en een lijst met klusjes die je moet doen en een lijst met je lievelingstaarten. Je kunt die drie lijsten weer in een andere lijst stoppen. Probeer dit eens te typen in de interactieve shell:

```
>>> boodschappen = ['eieren', 'melk', 'soep', 'appels', 'brood']
>>> klusjes = ['schoonmaken', 'gras maaien', 'boodschappen doen']
>>> lievelingstaarten = ['appeltaart', 'chocoladetaart']
>>> lijstMetLijsten = [boodschappen, klusjes, lievelingstaarten]
>>> lijstMetLijsten
[['eieren', 'melk', 'soep', 'appels', 'brood'], ['schoonmaken', 'grasmaaien',
'boodschappen doen'], ['appeltaart', 'chocoladetaart']]
```

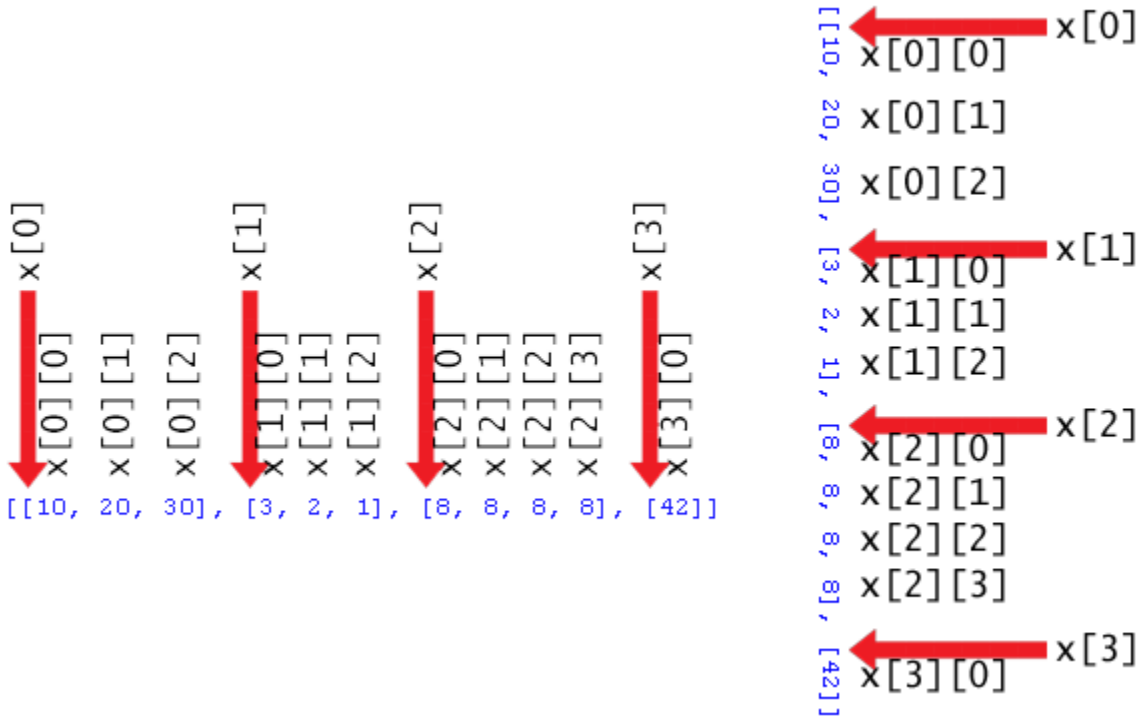
Om een bepaald item in de lijst met lijsten op te halen, gebruik je twee paar vierkante haken, als volgt: `lijstMetLijsten[1][2]` resulteert in de string `'boodschappen doen'`.

Dat komt omdat `lijstMetLijsten[1]` resulteert in `['schoonmaken', 'gras maaien', 'boodschappen doen'][2]`. En dat resulteert uiteindelijk in `'boodschappen doen'`:

```
lijstMetLijsten[1][2]
         ▼
[['eieren', 'melk', 'soep', 'appels', 'brood'], ['schoonmaken', 'grasmaaien',
'boodschappen doen'], ['appeltaart', 'chocoladetaart']][1][2]
         ▼
['schoonmaken', 'gras maaien', 'boodschappen doen'][2]
         ▼
'boodschappen doen'
```

Afbeelding 9-1 toont nog een voorbeeld van een lijst met lijsten, samen met de indexnummers die naar de items wijzen. De pijlen wijzen naar de indexnummers van de binnenste lijsten. De

afbeelding staat ook op zijn kant zodat het duidelijker is. Vergeet niet dat het eerste item indexnummer 0 heeft:



Afbeelding 9-1: De indexnummers van een lijst met lijsten.

Methoden

Methoden zijn functies die van toepassing zijn op een bepaalde waarde. Alle stringwaarden kennen bijvoorbeeld de methode `lower()`, die de stringwaarde omzet in kleine letters. Je kunt `lower()` niet in zijn eentje aanroepen en je kunt ook geen stringargument aan `lower()` doorgeven (zoals `lower('Hallo')` of zo). Je moet de methodeaanroep koppelen aan een bepaalde stringwaarde. Dit doe je met een punt. In de volgende paragraaf kijken we naar twee stringmethoden als voorbeelden.

De stringmethoden `lower()` en `upper()`

Probeer eens `'Hallo wereld!'.lower()` te typen in de interactieve shell. Je ziet dan een voorbeeld van deze methode.

```
>>> 'Hallo wereld!'.lower()
'hallo wereld!'
```

Er is ook een methode `upper()` voor strings, die een string retourneert met alle tekens in hoofdletters. Typ maar `'Hallo wereld!'.upper()` in de interactieve shell:

```
>>> 'Hallo wereld!'.upper()
'HALLO WERELD! '
```

Omdat de methode `upper()` een string retourneert, kun je op die string ook weer een methode aanroepen. Probeer maar eens `'Hallo wereld!'.upper().lower()` in de interactieve shell te typen:

```
>>> 'Hallo wereld!'.upper().lower()
'hallo wereld!'
```

`'Hallo wereld!'.upper()` retourneert de string `'HALLO WERELD!'`, en vervolgens wordt op die string de methode `lower()` aangeroepen. Dan krijg je `'hallo wereld!'` als resultaat.

```
'Hallo wereld!'.upper().lower()
                       ▼
       'HALLO WERELD!'.lower()
                       ▼
                 'hallo wereld!'
```

De volgorde is natuurlijk belangrijk. `'Hallo wereld!'.lower().upper()` levert niet hetzelfde resultaat als `'Hallo wereld!'.upper().lower()`:

```
>>> 'Hallo wereld!'.lower().upper()
'HALLO WERELD!'
```

Deze evaluatie gaat als volgt:

```
'Hallo wereld!'.lower().upper()
                       ▼
        'hallo wereld!'.upper()
                       ▼
                'HALLO WERELD!'
```

Als een string in een variabele wordt bewaard, kun je de stringmethode ook op die variabele aanroepen. Kijk maar naar dit voorbeeld:

```
>>> ham = 'Hallo wereld!'
>>> ham.upper()
'HALLO WERELD!'
```

Merk op dat de waarde in `ham` hierdoor niet wordt gewijzigd. De variabele `ham` bevat nog steeds `'Hallo wereld!'`.

De lijstmethoden `reverse()` en `append()`

Het datatype Lijst kent ook methoden. De methode `reverse()` draait de volgorde van de items in een lijst om. Typ maar `ham = [1, 2, 3, 4, 5, 6, 'miauw', 'woef']`, en dan `ham.reverse()` om de lijst om te draaien. Typ dan `ham` om de inhoud van de variabele weer te geven.

```
>>> ham = [1, 2, 3, 4, 5, 6, 'miauw', 'woef']
>>> ham.reverse()
>>> ham
['woef', 'miauw', 6, 5, 4, 3, 2, 1]
```

De meest gebruikte lijstmethode is `append()`. Met deze methode wordt de waarde die je meegeeft als argument achteraan in de lijst toegevoegd. Probeer het volgende eens in de interactieve shell te typen:

```
>>> eieren = []
>>> eieren.append('veerboot')
>>> eieren
['veerboot']
>>> eieren.append('paling')
>>> eieren
['veerboot', 'paling']
>>> eieren.append(42)
>>> eieren
['veerboot', 'paling', 42]
```

Er bestaan trouwens geen methoden voor integers en floats.

De lijstmethode `split()`

Regel 59 is een lange regel code, maar eigenlijk is dit gewoon een simpele toewijzingsstatement. In de regel staat ook de methode `split()`, waarmee je een bewerking kunt uitvoeren op het datatype String, net zoals je dat kunt doen met de methoden `lower()` en `upper()`.

```
59. woorden = 'aap adelaar baviaan beer bever cobra cougar coyote das duif eend
eland forel fret gans geit haai hagedis havik hert hond kalkoen kameel kat
kikker konijn kraai lama leeuw luiaard mier mol mossel muilezel muis neushoorn
ooievaar otter pad panda papegaai python raaf ram rat salamander schaap
```

```
schildpad slang spin stinkdier tijger uil vleermuis vos walvis wezel wolf
wombat zalm zeehond zwaan zebra'.split()
```

Deze toewijzingsstatement bevat gewoon één lange string van woorden die van elkaar zijn gescheiden door spaties. Aan het eind van de string zie je de methode `split()`. De methode `split()` resulteert in een lijst met elk woord in de string als apart lijstitem. De 'splitsing' van de string in aparte onderdelen vindt plaats waar een spatie staat.

Het is minder werk om de code te typen met behulp van de methode `split()`. Als je een lijst zou moeten typen, zou je het als volgt moeten doen: `['aap', 'adelaar', 'baviaan',...` enzovoort, met bij elk woord aanhalingstekens en een komma.

Als je nog een voorbeeld wilt zien van de stringmethode `split()` typ je dit in de interactieve shell:

```
>>> zin = input()
Mijn zeer energieke moeder gaf ons net een bakje chips.
>>> zin.split()
['Mijn', 'zeer', 'energieke', 'moeder', 'gaf', 'ons', 'net', 'een', 'bakje',
'chips.']
```

Het resultaat is een lijst met tien strings, één string voor elk woord in de oorspronkelijke string. De spaties maken geen deel uit van de items in de lijst.

Je kunt ook je eigen woorden aan de string op regel 59 toevoegen, of er een paar weghalen die je niet leuk vindt. Onze code zal straks kunnen omgaan met lijsten van variërende lengte. Je moet er alleen voor zorgen dat de woorden van elkaar worden gescheiden door een spatie.

Hoe de code werkt

Regel 61 definieert de functie `haalWillekeurigWoord()`, met de parameter `woordenLijst`. Met deze functie wordt één willekeurig geheim woord opgehaald uit de `woordenLijst`.

```
 61. def haalWillekeurigWoord(woordenLijst):
 62.     # Deze functie retourneert een willekeurige string uit een doorgegeven
lijst met strings.
 63.     woordIndex = random.randint(0, len(woordenLijst) - 1)
 64.     return woordenLijst[woordIndex]
```

Regel 63 slaat het willekeurige indexnummer voor deze lijst op in de variabele `woordIndex`. Je doet dit door `randint()` aan te roepen met twee argumenten. Het eerste argument is `0` en het tweede argument is de waarde die het resultaat is van de expressie `len(woordenLijst) - 1`.

Lijstindexen beginnen altijd bij `0`, niet bij `1`. Als je een lijst hebt met drie items, is het indexnummer van het eerste item `0`, het indexnummer van het tweede item is `1` en het indexnummer van het derde item is `2`. De lengte van deze lijst is 3 (er zijn drie items) maar het laatste indexnummer is niet 3, maar 2. (3-1 dus). Daarom zie je dat er op regel 63 1 wordt afgetrokken van de lengte van de lijst.

De variabele `woordIndex` krijgt de waarde van een willekeurig indexnummer uit de lijst die werd doorgegeven in de parameter `woordenLijst`. Regel 64 retourneert het element in `woordenLijst` dat zich bevindt op de indexpositie die staat in de variabele `woordIndex`.

Laten we even doen of `['appel', 'sinaasappel', 'druif']` was doorgegeven als het argument van `haalWillekeurigWoord()` en dat `randint(0, 2)` de integer `2` retourneerde. Dan zou regel 64 `return woordenLijst[2]` worden, wat uiteindelijk de retourwaarde `'druif'` zou opleveren. En zo retourneert de functie `haalWillekeurigWoord()` een willekeurige string uit de lijst `woordenLijst`.

Dus de invoer aan `haalWillekeurigWoord()` is een lijst met strings en de uitvoer van deze functie is één willekeurig gekozen string uit die lijst. En dat hebben we nodig bij Galgje.

De galg aan de speler laten zien

Nu heb je een functie nodig die de galg op het scherm weergeeft. Daarbij moet ook staan hoeveel letters de speler al goed (en fout) heeft geraden. En het geheime woord moet worden getoond, met streepjes voor de ontbrekende letters.

```
66. def toonGalg(GALGTEKENING, fouteLetters, goedeLetters, geheimWoord):
67.     print(GALGTEKENING[len(fouteLetters)])
68.     print()
```

Deze code definieert een nieuwe functie met de naam `toonGalg()`. Deze functie heeft 4 parameters:

- `GALGTEKENING`: een lijst met multiregel-strings die de galg tekenen als ASCII-art. (De globale variabele `GALGTEKENING` wordt doorgegeven als parameter.)
- `fouteLetters`: een string met de letters die de speler heeft genoemd maar die niet in het woord voorkomen. Bij elke foute letter wordt deze string langer.
- `goedeLetters`: een string met de letters die de speler goed heeft geraden.
- `geheimWoord`: een string met het geheime woord dat de speler probeert te raden.

De eerste functieaanroep `print()` laat de galg zien. `GALGTEKENING` is een lijst met strings die alle mogelijke galgen tekenen. `GALGTEKENING[0]` toont een lege galg, `GALGTEKENING[1]` toont een galg met alleen een hoofd (de speler heeft dan 1 letter fout geraden), `GALGTEKENING[2]` toont een

galg met een hoofd en een romp (als de speler 2 letters fout heeft geraden) enzovoort tot aan `GALGTEKENING[6]` waar je de galg ziet met het hele lijf eraan hangend.

Het aantal letters dat in `fouteLetters` zit, geeft aan hoe vaak de speler al fout heeft geraden. Dus je roept `len(fouteLetters)` aan om te kijken hoe lang deze string is (hoeveel letters de string bevat). Dus als de waarde van `fouteLetters` `'aetr'` is, dan retourneert `len('aetr')` de waarde `4`. Als je dan `GALGTEKENING[4]` laat zien, wordt de juiste galg getoond voor 4 foute letters. En dat is het resultaat van `GALGTEKENING[len(fouteLetters)]`.

```
70.     print('Foute letters:', end=' ')
71.     for letter in fouteLetters:
72.         print(letter, end=' ')
73.     print()
```

Regel 70 geeft de string `'Foute letters:'` weer, met een spatieteken aan het eind in plaats van een nieuweregelteken. Wist je nog dat het sleutelwoordargument `end=' '` één teken = gebruikt, niet twee zoals ==?

Regel 71 bevat een nieuw type lus, de `for`-lus. Een `for`-lus wordt vaak gebruikt met de functie `range()`. We behandelen beide in de volgende twee paragrafen.

De functies `range()` en `list()`

Wanneer de functie `range()` wordt aangeroepen met één argument, wordt een 'bereik' van integers geretourneerd dat loopt van 0 tot het meegegeven argument (*tot*, dus niet *tot en met*). Van dit bereik kun je gemakkelijk een lijst maken met de functie `list()`. Typ `list(range(10))` in de interactieve shell:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list('Hallo')
['H', 'a', 'l', 'l', 'o']
```

De functie `list()` lijkt op de functie `str()` of `int()`. De functie zet het object dat het meekrijgt, gewoon om in een lijst. Je kunt met de functie `range()` heel makkelijk enorm lange lijsten maken. Typ maar `list(range(10000))` in de interactieve shell:

```
>>> list(range(10000))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...
      ...we slaan maar even een stukje over, anders duurt het zo lang...
...9989, 9990, 9991, 9992, 9993, 9994, 9995, 9996, 9997, 9998, 9999]
```

De lijst is zo enorm dat hij niet eens op het scherm past. Je kunt de lijst in een variabele opslaan:

```
>>> ham = list(range(10000))
```

Als je twee argumenten doorgeeft aan `range()` in plaats van één, loopt de lijst met items van het eerste argument tot aan het tweede argument. Probeer `list(range(10, 20))` te typen in de interactieve shell:

```
>>> list(range(10, 20))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

De methode `range()` wordt vaak gebruikt in een `for`-lus (een lus die lijkt op een `while`-lus die je al kent).

`for`-lussen

De `for`-lus is handig als je een bepaald aantal keer iets moet doen, bijvoorbeeld iets met een aantal items in een lijst. Deze lus verschilt daarin van de `while`-lus, want die loopt door zo lang een bepaalde waarde `True` is. Een `for`-statement begint met het sleutelwoord `for`, gevolgd door de naam van een nieuwe variabele, gevolgd door het sleutelwoord `in`, gevolgd door een reeks, en hij eindigt met een dubbele punt.

De reeks is vaak een bereik dat wordt geretourneerd door de functie `range()` maar het kan ook een lijst zijn.

Elke keer dat de programmaverwerking door de lus gaat, krijgt de nieuwe variabele in de `for`-statement de waarde van het betreffende item in de lijst.

```
>>> for i in range(5):
...     print('i krijgt de waarde ' + str(i))
...
i krijgt de waarde 0
i krijgt de waarde 1
i krijgt de waarde 2
i krijgt de waarde 3
i krijgt de waarde 4
```

Het bereik dat werd geretourneerd door de functie range(5) is de lijst [0, 1, 2, 3, 4]. De eerste keer dat de verwerking door de code in het for-blok gaat, heeft de variabele i de waarde 0. Bij de volgende iteratie heeft i de waarde 1 gekregen, enzovoort.

De `for`-statement zet het bereik dat wordt geretourneerd door de functie `range()` automatisch om in een lijst, dus je hoeft niet `list(range(5))` te zeggen in de `for`-statement. `range(5)` is genoeg.

Lijsten en strings zijn ook een reeksachtig datatype. Daarom kun je lijsten en strings gebruiken in `for`-statements. Probeer dit eens te typen in de interactieve shell:

```
>>> for ding in ['katten', 'pasta', 'programmeren', 'ham']:
...     print('Ik hou veel van ' + ding)
...
Ik hou veel van katten
Ik hou veel van pasta
Ik hou veel van programmeren
Ik hou veel van ham

>>> for i in 'Hallo':
...     print(i)
...
H
a
l
l
o
```

Een `while`-lus gebruiken of een `for`-lus

De `for`-lus lijkt op een `while`-lus, maar als je iets moet doen met items in een lijst, is het veel handiger om een `for`-lus te gebruiken. Dan hoef je veel minder te typen. Je zou om hetzelfde te bereiken ook een `while`-lus kunnen gebruiken, maar dan moet je meer code typen:

```
>>> reeks = ['katten', 'pasta', 'programmeren', 'ham']
>>> indexnummer = 0
>>> while (indexnummer < len(reeks)):
...     nieuweVariabele = reeks[indexnummer]
...     print('Ik hou veel van ' + nieuweVariabele)
...     indexnummer = indexnummer + 1
...
Ik hou veel van katten
Ik hou veel van pasta
Ik hou veel van programmeren
Ik hou veel van ham
```

Als je de `for`-statement gebruikt, wordt al deze extra code al automatisch gedaan, en hoef je dit dus niet allemaal te typen.

De rest van de functie `toonGalg()` laat de foute letters zien en laat de string van het geheime woord zien met alle nog niet geraden letters weergegeven als het teken _.

```
70.     print('Foute letters:', end=' ')
71.     for letter in fouteLetters:
72.         print(letter, end=' ')
73.     print()
```

De `for`-lus op regel 71 doorloopt elk teken in de string `fouteLetters` en geeft het teken weer op het scherm. En het doet dat achter elkaar en niet onder elkaar omdat we het sleutelwoord `end=' '` toevoegden, zodat het nieuweregelteken van de functie print() wordt vervangen door een spatie.

Als `fouteLetters` bijvoorbeeld `'ajtw'` zou zijn, geeft de `for`-lus dit weer als `a j t w`.

Slicen

Door een **lijst te 'slicen'** (zeg maar: 'op te knippen') kun je een nieuwe lijst maken met een subset van de items van de oude lijst. In de code geef je twee indexen op (het begin en het eind) en een dubbele punt in de vierkante haken achter een lijstnaam. Typ bijvoorbeeld het volgende in de interactieve shell:

```
>>> ham = ['appels', 'bananen', 'wortels', 'dadels']
>>> ham[1:3]
['bananen', 'wortels']
```

De expressie `ham[1:3]` wordt geëvalueerd als een lijst met de items vanaf indexnummer `1` tot aan (niet tot en met!) indexnummer `3` van `ham`.

Als je het eerste indexnummer weglaat, denkt Python automatisch dat je indexnummer 0 bedoelt als eerste indexnummer:

```
>>> ham = ['appels', 'bananen', 'wortels', 'dadels']
>>> ham[:2]
['appels', 'bananen']
```

Als je het tweede indexnummer weglaat, denkt Python automatisch dat je de rest van de lijst wilt zien:

```
>>> ham = ['appels', 'bananen', 'wortels', 'dadels']
>>> ham[2:]
['wortels', 'dadels']
```

Slicen is een gemakkelijke manier om een subset (onderdeel) te krijgen van de items in een lijst. Je kunt strings op dezelfde manier slicen als je dat met lijsten kunt doen. Elk teken in een string is als een item in een lijst. Probeer het volgende eens in de interactieve shell te typen:

```
>>> mijnNaam = 'Darius de dikke kat'
>>> mijnNaam[4:12]
'us de di'
>>> mijnNaam[:10]
'Darius de '
>>> mijnNaam[7:]
```

```
'de dikke kat'
```

In het volgende onderdeel van de code in Galgje wordt slicen gebruikt.

Het geheime woord weergeven met streepjes

Nu heb je code nodig waarmee je het geheime woord kunt weergeven, met streepjes voor de letters die nog niet zijn geraden. Als streepje gebruiken we het onderstrepingsteken (_). Je maakt eerst een string met alleen onderstrepingstekens voor alle letters in het geheime woord. Daarna kun je de streepjes vervangen door elke letter in `goedeLetters`.

Dus als het geheime woord `'otter'` was, ziet de string met lege plekken eruit als `'_____'` (vijf onderstrepingstekens). Als `juisteLetters` de string `'rt'` bevat, verander je de string in `'_tt_r'`. Dit gebeurt in de code van de regels 75 t/m 79.

```
 75.     streepjes = '_' * len(geheimWoord)
```

Regel 75 maakt de variabele `streepjes` aan waarin onderstrepingstekens worden bewaard. Om het juiste aantal onderstrepingstekens te krijgen, gebruiken we de operator `*`. Die kunnen we namelijk ook met strings gebruiken, niet alleen met integers. De expressie `'_' * 5` resulteert dan in `'_____'`. Nu heeft `streepjes` net zoveel onderstrepingstekens als er letters zitten in `geheimWoord`. (En hoeveel letters daar in zitten, hebben we uitgevonden door de functie len().)

```
 77.     for i in range(len(geheimWoord)): # vervang streepjes door de goed
geraden letters
 78.         if geheimWoord[i] in goedeLetters:
 79.             streepjes = streepjes[:i] + geheimWoord[i] + streepjes[i+1:]
```

Regel 77 bevat een `for`-loop om langs elke letter in `geheimWoord` te gaan en het onderstrepingsteken te vervangen door de juiste letter, als die letter ten minste voorkomt in `goedeLetters`.

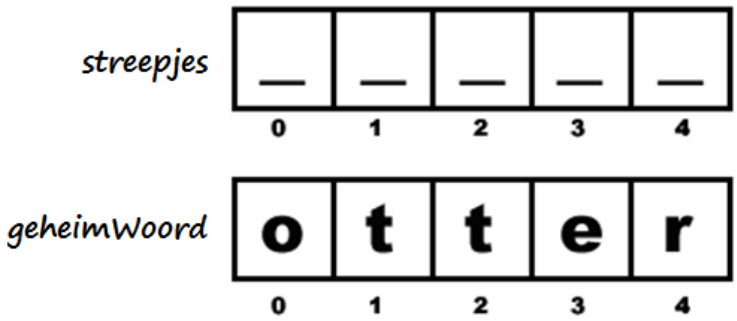
Stel dat de waarde van `geheimWoord` `'otter'` is en de waarde van `goedeLetters` is `'tr'`. Dan wil je de string `'_tt_r'` laten zien aan de speler. We gaan even uitvogelen hoe we dit voor elkaar moeten krijgen.

De aanroep `len(geheimWoord)` op regel 77 retourneert `5`. De aanroep `range(len(geheimWoord))` wordt geëvalueerd als `range(5)` en vervolgens als de lijst `[0, 1, 2, 3, 4]`.

Omdat de waarde van `i` elke waarde in `[0, 1, 2, 3, 4]` aan gaat nemen, gebeurt er in de code in de `for`-lus eigenlijk het volgende:

```
if geheimWoord[0] in goedeLetters:
    streepjes = streepjes[:0] + geheimWoord[0] + streepjes[1:]

if geheimWoord[1] in goedeLetters:
    streepjes = streepjes[:1] + geheimWoord[1] + streepjes[2:]

if geheimWoord[2] in goedeLetters:
    streepjes = streepjes[:2] + geheimWoord[2] + streepjes[3:]

if geheimWoord[3] in goedeLetters:
    streepjes = streepjes[:3] + geheimWoord[3] + streepjes[4:]

if geheimWoord[4] in goedeLetters:
    streepjes = streepjes[:4] + geheimWoord[4] + streepjes[5:]
```

Als het je een beetje duizelt over wat de waarde is van zoiets als `geheimWoord[0]` of `streepjes[3:]`, kijk dan naar Afbeelding 9-2. Daar zie je alle waarden van `geheimWoord` en `streepjes` die worden aangeduid door het indexnummer.



Afbeelding 9-2: De indexnummers van de strings `streepjes` en `geheimWoord`.

Als je de lijstslices en lijstindexnummers vervangt door de waarden die zij momenteel hebben, ziet de code van de lus er als volgt uit:

```
if 'o' in 'tr': # False
    streepjes = '' + 'o' + '____' # Deze regel wordt overgeslagen want niet
waar.

if 't' in 'tr': # True
    streepjes = '_' + 't' + '___' # De regel wordt verwerkt want waar.

if 't' in 'tr': # True
    streepjes = '_t' + 't' + '__' # Deze regel wordt verwerkt want waar.
```

```
if 'e' in 'tr': # False
    streepjes = '_tt' + 'e' + '_' # Deze regel wordt overgeslagen want niet
waar.

if 'r' in 'tr': # True
    streepjes = '_tt_' + 'r' + '' # Deze regel wordt verwerkt want waar.

# streepjes bevat nu de waarde '_tt_r'
```

Bovenstaande codevoorbeelden zijn van toepassing wanneer `geheimWoord 'otter'` is en `goedeLetters 'tr'` is. De volgende paar regels code geven de nieuwe waarde van `streepjes` weer, met spaties tussen elke letter.

```
 81.     for letter in streepjes: # toon het geheime woord met spaties tussen
elke letter
 82.         print(letter, end=' ')
 83.     print()
```

Deze `for`-lus laat elk teken in de string `streepjes` zien. Vergeet niet dat tegen deze tijd, sommige van de onderstrepingstekens in `streepjes` al kunnen zijn vervangen door de letters van `geheimWoord`. Het sleutelwoordargument `end` in de aanroep naar `print()` op regel 82 zorgt ervoor dat de functie `print()` een spatie toevoegt aan het einde van elke string in plaats van een nieuweregelteken. En zo zijn we bij het einde van de functie `toonGalg()` beland.

De letter die de speler noemt, ophalen

De functie `haalpoging()` wordt aangeroepen wanneer de speler een letter typt als poging. De functie retourneert de letter die de speler typt, als een string. Verder zorgt `haalpoging()` ervoor dat de speler een geldige letter typt.

```
 85. def haalPoging(alGeraden):
 86.     # Retourneert de letter die de speler heeft getypt. Deze functie zorgt
ervoor dat de speler één letter heeft getypt, en niet iets anders.
```

De functie `haalpoging()` heeft een stringparameter met de naam `alGeraden` waarin een string wordt doorgegeven die de letters bevat die de speler al heeft geraden. Daarna wordt de speler gevraagd één letter te typen. Deze letter wordt de retourwaarde van deze functie.

```
 87.     while True:
 88.         print('Raad een letter.')
 89.         poging = input()
 90.         poging = poging.lower()
```

Op regel 87 blijft de `while`-lus de speler om een letter vragen, totdat zij tekst invoeren die:

1. een enkele letter is;
2. een letter is die ze nog niet hebben geraden.

De voorwaarde in de `while`-lus is gewoon de Boole-waarde `True`. Dat betekent dat de enige manier waarop deze lus ooit kan worden verlaten, is om een `break`-statement uit te voeren (waardoor de lus onmiddellijk wordt verlaten) of door een `return`-statement uit te voeren (waarmee de hele functie wordt verlaten).

De code in de lus vraagt de speler een letter te typen, die wordt opgeslagen in de variabele `poging`. Als de speler toevallig een hoofdletter invoerde, wordt deze omgezet in een kleine letter met de code op regel 90.

`elif` ("Else If")-statements

Bekijk de volgende code eens:

```
if katNaam == 'Haarbal':
    print('Jouw kat is nogal harig.')
elif katNaam == 'Stippel'
        print('Jouw kat heeft stippen.')
else:
    print('Jouw kat is helemaal niet zo harig.')
```

Als de variabele `katNaam` gelijk is aan de string `'Haarbal'`, is de voorwaarde van de `if`-statement `True` en krijgt de gebruiker te horen dat hun kat harig is. Maar als deze voorwaarde False zou zijn, test Python vervolgens de voorwaarde van de elif-statement ('else if' oftewel 'anders als'). Als `katNaam` `'Stippel'` is, wordt de string `'Jouw kat heeft stippen.'` op het scherm weergegeven. Als beide statements False zijn, krijgt de gebruiker te horen dat hun kat helemaal niet zo harig is.

Je kunt aan de `if-elif-else`-statements denken als: "Als dit waar is, doet dit. Of anders, als de volgende voorwaarde waar is, doe dat. Of anders, als geen van alle waar zijn, doe dit laatste."

Je kunt zo veel `elif`-statements hebben als je wilt:

```
if katNaam == 'Haarbal':
    print('Jouw kat is nogal harig.')
elif katNaam == 'Stippel'
    print('Jouw kat heeft stippen.')
elif katNaam == 'Dikkie'
    print('Jouw kat is dik.')
elif katNaam == 'Duffie'
```

```
    print('Jouw kat is een beetje duf.')
else:
    print('Jouw kat is geen haarbal, heeft geen stippen, is niet dik en ook
niet duf.')
```

Wanneer een van de voorwaarden van de `elif`-statement `True` is, wordt die code uitgevoerd en springt de verwerking naar de eerste regel voorbij het else-blok. Dus *slechts één* van de blokken in deze `if-elif-else`-statement wordt uitgevoerd. Je kunt het else-blok ook weglaten als je het niet nodig hebt en gewoon een `if-else`-statement gebruiken.

Zorgen dat de speler een geldige letter noemt

```
 91.         if len(poging) != 1:
 92.             print('Je mag maar 1 letter typen.')
 93.         elif poging in alGeraden:
 94.             print('Die letter heb je al gehad. Probeer een andere
letter.')
 95.         elif poging not in 'abcdefghijklmnopqrstuvwxyz':
 96.             print('Je moet een LETTER typen.')
 97.         else:
 98.             return poging
```

De variabele `poging` bevat de letter die de speler heeft getypt. Het programma moet controleren of er één letter is getypt in kleine letters. Als dat niet zo is, moet het programma teruggaan en de speler opnieuw vragen.

De voorwaarde op regel 91 controleert of `poging` langer is dan 1 teken. De voorwaarde op regel 93 controleert of de waarde van `poging` al voorkomt in de variabele `alGeraden`. De voorwaarde op regel 95 controleert of `poging` geen kleine letter is.

Als al deze voorwaarden False opleveren, wordt het `else`-blok uitgevoerd en retourneert de functie `haalpoging()` op regel 98 de waarde die in de variabele `poging` zit.

Vergeet niet dat slechts één van de blokken in de `if-elif-else`-statements wordt uitgevoerd.

De speler vragen of ze nog een keertje willen spelen

```
100. def NogmaalsSpelen():
101.     # Deze functie retourneert True als de speler nog een keer wil spelen,
anders is het False.
102.     print('Wil je nog een keer spelen? (ja of nee)')
103.     return input().lower().startswith('j')
```

De functie `nogmaalsSpelen()` bevat alleen een functieaanroep naar `print()` en een `return`-statement. De `return`-statement bevat een expressie die er ingewikkeld uitziet, maar je kunt hem ontleden. Hier zie je stap voor stap hoe Python deze expressie evalueert als de gebruiker JA intypt.

```
input().lower().startswith('j')
           ▼
  'JA'.lower().startswith('j')
           ▼
         'ja'.startswith('j')
           ▼
          True
```

De bedoeling van de functie `nogmaalsSpelen()` is om de speler ja of nee te laten typen om te zeggen of ze nog een rondje Galgje willen spelen. Als de speler JA typt, is de retourwaarde van `input()` de string `'JA'`. En `'JA'.lower()` retourneert de string in kleine letters. Dus de retourwaarde van `'JA'.lower()` is `'ja'`.

Maar je ziet een tweede methodeaanroep, `startswith('j')`. Deze functie retourneert `True` als de bijbehorende string begint met de stringparameter tussen de haakjes, en `False` als dat niet zo is. De retourwaarde van `'ja'.startswith('j')` is dus `True`.

Nu heb je deze expressie geëvalueerd! De speler typt dus een antwoord en dit antwoord wordt geretourneerd in kleine letters en moet beginnen met de letter `'j'`, als dat zo is, wordt de waarde `True` en anders wordt de waarde `False`.

Tussen twee haakjes, er bestaat ook een stringmethode `endswith(eenString)` die `True` oplevert als de string eindigt op de string in `eenString` en `False` als dat niet het geval is. `endswith()` is zeg maar het tegenovergestelde van `startswith()`.

Overzicht van de functies in Galgje

En dat zijn alle functies die we voor dit spel maken!

- `haalWillekeurigWoord(woordenLijst)` krijgt een lijst met strings doorgegeven als parameter, en retourneert één willekeurige string uit die lijst. Zo wordt een woord gekozen dat de speler moet raden.
- `toonGalg(GALGTEKENING, fouteLetters, goedeLetters, geheimWoord)` toont de huidige toestand van de galg en hoeveel letters van het woord al geraden zijn en hoeveel foute letters zijn genoemd. Deze functie heeft vier parameters nodig om goed te kunnen werken. `GALGTEKENING` is een lijst met strings die de ASCII-tekeningen vormen voor elke mogelijke galg. `goedeLetters` en `fouteLetters` zijn strings die bestaan uit de letter die de speler heeft genoemd en die respectievelijk wel en niet in het geheime woord

voorkomen. En `geheimWoord` is het geheime woord dat de speler moet raden. Deze functie heeft geen retourwaarde.

- `haalpoging(alGeraden)` krijgt een string mee met de letters die de speler al heeft genoemd, en blijft de speler vragen om een letter tot de speler een letter noemt die nog niet eerder is genoemd. (Dus een letter die niet voorkomt in `alGeraden`.) Deze functie retourneert de string van de geldige letter die de speler heeft genoemd.
- `nogmaalsSpelen()` is een functie die de speler vraagt of ze nog een rondje Galgje willen doen. Deze functie retourneert `True` als de speler nog een keer wil spelen en `False` als dat niet zo is.

Na de functies komt de code voor het hoofdgedeelte van het programma. Het hoofdgedeelte van de code roept alle bovenstaande functies aan in de vereiste volgorde.

De hoofdcode voor Galgje

Het hoofdonderdeel van de code begint op regel 106. Alles wat daarvoor kwam, waren gewoon functiedefinities en een hele lange toewijzingsstatement voor de variabele `GALGTEKENING`.

De variabelen instellen

```
106. print('G A L G J E')
107. fouteLetters = ''
108. goedeLetters = ''
109. geheimWoord = haalWillekeurigWoord(woorden)
110. klaarMetSpel = False
```

Regel 106 is de eerste functieaanroep naar `print()` die wordt uitgevoerd zodra het spel wordt gestart. Begin door een lege string toe te wijzen aan `fouteLetters` en `goedeLetters` omdat de speler nog geen goede of foute letters heeft genoemd.

De functieaanroep `haalWillekeurigWoord(woorden)` resulteert in een willekeurig gekozen woord uit de lijst `woorden`. Dit woord wordt geplaatst in de variabele geheimWoord.

Regel 110 stelt een variabele met de naam `klaarMetSpel` in op `False`. De code zal `klaarMetSpel` op `True` zetten als je wilt aangeven dat het spel over is en het programma zal dan vragen of de speler nog eens wil spelen.

De galg aan de speler laten zien

```
112. while True:
113.     toonGalg(GALGTEKENING, fouteLetters, goedeLetters, geheimWoord)
```

De voorwaarde van de `while`-lus blijft altijd op `True` staan, wat betekent dat de lus eeuwig duurt totdat de `break`-statement wordt tegengekomen.

Regel 113 roept onze functie `toonGalg()` aan en geeft de lijst met ASCII-tekeningen mee en de drie variabelen die zijn ingesteld op de regels 107, 108 en 109. Afhankelijk van hoeveel letters de gebruiker goed en hoeveel hij fout heeft geraden, geeft deze functie de juiste galg weer aan de speler.

De speler een letter laten invoeren

```
115.     # Laat de speler een letter typen.
116.     poging = haalpoging(fouteLetters + goedeLetters)
```

De functie `haalpoging()` heeft alle letters in zowel `fouteLetters` als `goedeLetters` nodig om te weten welke letters al genoemd zijn, dus we concateneren de strings in deze variabelen met de operator + en geven het resultaat door als argument. Dit argument heeft `haalpoging()` nodig om te kunnen controleren of de speler een letter typt die ze al gehad hebben.

Controleren of de letter in geheimWoord voorkomt

```
118.     if poging in geheimWoord:
119.         goedeLetters = goedeLetters + poging
```

Als de string in `poging` in `geheimWoord` voorkomt, voegen we de inhoud van `poging` toe achteraan de string in `goedeLetters`. Deze string wordt dan de nieuwe waarde van `goedeLetters`.

Controleren of de speler heeft gewonnen

```
121.         # Controleer of de speler heeft gewonnen
122.         alleLettersGevonden = True
123.         for i in range(len(geheimWoord)):
124.             if geheimWoord[i] not in goedeLetters:
125.                 alleLettersGevonden = False
126.                 break
```

Hoe weet het programma of de speler elke letter in het woord heeft geraden? Nou, `goedeLetters` bevat elke letter die de speler goed heeft geraden en `geheimWoord` bevat het geheime woord. Je kunt niet gewoon controleren of `goedeLetters == geheimWoord`, want in `geheimWoord` kan de string `'otter'` staan en in `goedeLetters` kan de string `'orte'` staan (als de speler de letters in die volgorde noemde), dus `goedeLetters == geheimWoord` zou dan `False` zijn, ook al had de speler alle letters in het geheime woord *wel* geraden.

De enige manier om zeker uit te vinden of de speler heeft gewonnen, is om letter voor letter te controleren of de letters in `geheimWoord` en `goedeLetters` overeenkomen. Als (en alleen als) elke letter in `geheimWoord` voorkomt in `goedeLetters`, heeft de speler gewonnen.

Je kunt elke letter in `geheimWoord` met een lus doorlopen en als je een letter vindt die niet in `goedeLetters` voorkomt. weet je dat de speler nog **niet** alle letters heeft geraden. De nieuwe variabele `alleLettersGevonden` wordt ingesteld op `True` op regel 122 voor de lus begint. De lus begint dus met de aanname dat alle letters in het geheime woord zijn geraden. Maar de code van de lus wijzigt `alleLettersGevonden` in `False` zodra het een letter vindt in `geheimWoord` die niet voorkomt in `goedeLetters` (regel 125).

```
127.         if alleLettersGevonden:
128.             print('Ja! Het geheime woord was "' + geheimWoord + '"! Jij
hebt gewonnen!')
129.             klaarMetSpel = True
```

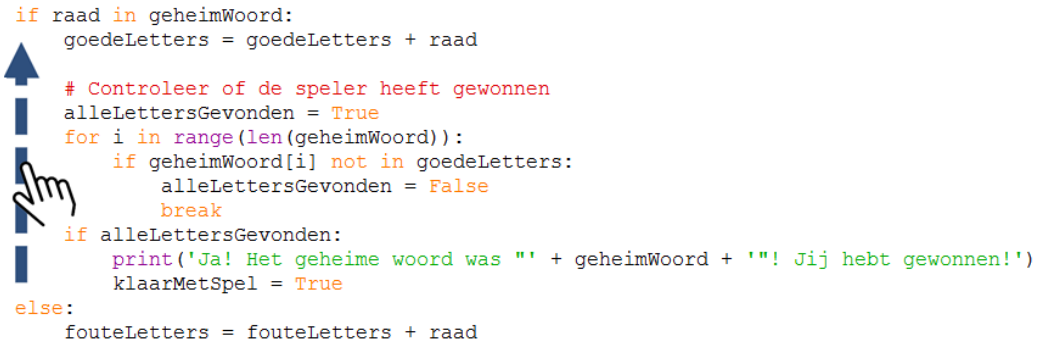
Als alle letters in het geheime woord zijn gevonden, wordt de speler verteld dat ze hebben gewonnen en wordt `klaarMetSpel` op `True` gezet.

Wanneer de speler het fout raadt

```
130.     else:
131.         fouteLetters = fouteLetters + poging
```

Dit is de start van het else-blok. Weet je nog dat de code in dit blok wordt uitgevoerd als de voorwaarde `False` is? Maar welke voorwaarde is dat ook al weer? Om dat uit te vinden, kun je je

vinger aan het begin van het sleutelwoord `else` zetten en recht omhoog gaan zoals in Afbeelding 9-3. Dan zie je dat de inspringing van het sleutelwoord `else` gelijk is aan de inspringing van het sleutelwoord `if` op regel 118.



Afbeelding 9-3: De `else`-statement hoort bij deze `if`-statement die dezelfde mate van inspringing heeft.

Dus als de voorwaarde op regel 118 (`poging in geheimWoord`) `False` was, gaat de verwerking dit else-blok binnen. Anders wordt het else-blok overgeslagen en komen we bij regel 140.

Verkeerd geraden letters worden vastgemaakt aan de string `fouteLetters`. Dit deed je ook al op regel 119, voor goed geraden letters.

```
133.         # Controleer of de speler te vaak heeft geraden en dus verloren
134.         if len(fouteLetters) == len(GALGTEKENING) - 1:
135.             toonGalg(GALGTEKENING, fouteLetters, goedeLetters,
geheimWoord)
136.             print('Je kunt niet meer raden!\nNa ' + str(len(fouteLetters))
+ ' keer fout geraden te hebben en ' + str(len(goedeLetters)) + ' keer goed
geraden te hebben, was het woord "' + geheimWoord + '"')
137.             klaarMetSpel = True
```

Elke keer dat de speler het verkeerd heeft geraden, wordt de verkeerde letter toegevoegd aan de string in `fouteLetters`. De lengte van `fouteLetters` (of, in programmeertaal: `len(fouteLetters)`) is ook het aantal fout geraden letters.

De lijst `GALGTEKENING` bevat 7 ASCII-tekeningen. Dus wanneer `len(fouteLetters)` gelijk is aan `6`, weet je dat de speler heeft verloren, omdat de galg dan helemaal is getekend. Vergeet niet dat `GALGTEKENING[0]` het eerste item in de lijst is, en `GALGTEKENING[6]` het laatste item is.

Dus wanneer de lengte van de string `fouteLetters` gelijk is aan `len(GALGTEKENING) - 1` (dus 6), heeft de speler geen beurten meer over. Het geheime woord wordt getoond en de variabele `klaarMetSpel` wordt op `True` gezet.

```
139.     # Vraag de speler of ze het nog een keer willen proberen (maar alleen
als het spel klaar is).
140.     if klaarMetSpel:
141.         if nogmaalsSpelen():
142.             fouteLetters = ''
143.             goedeLetters = ''
144.             klaarMetSpel = False
145.             geheimWoord = haalWillekeurigWoord(woorden)
```

Of de speler nu heeft gewonnen of verloren, het spel moet in beide gevallen vragen of ze nog een keer willen spelen. De functie `nogmaalsSpelen()` zorgt voor een ja of nee van de speler, dus wordt aangeroepen op regel 141.

Als de speler nog eens wil spelen, moeten de waarden in `fouteLetters` en `goedeLetters` opnieuw op lege strings gezet, `klaarMetSpel` op `False`, en moet er een nieuw geheim woord worden gekozen voor `geheimWoord`. Zo komt er weer een nieuw spel voor de speler wanneer de verwerking teruggaat naar het begin van de lus op regel 112.

```
146.         else:
147.             break
```

Als de speler `'nee'` heeft getypt op de vraag of ze nog eens willen spelen, wordt de waarde van de voorwaarde op regel 141 `False` en wordt het else-blok uitgevoerd. De `break`-statement zorgt ervoor dat de verwerking naar de eerste regel na de lus springt. Maar omdat er na de lus geen code meer is, eindigt het programma.

Samenvatting

Dit was een heel lang hoofdstuk en je hebt heel veel nieuwe dingen geleerd. Galgje was tot nu toe ons ingewikkeldste spel. Nu je games ingewikkelder gaan worden, is het een goed idee om steeds een stroomdiagram op papier te tekenen zodat je goed kunt plannen wat er in je programma moet gebeuren.

Hoofdstuk 9 ½

GALGJE VERDER UITBREIDEN

In dit hoofdstuk behandelen we:
- Het datatype Dictionary
- Trefwoord-waarde-paren
- De methoden `keys()` en `values()` voor dictionary's
- Meervoudige variabeletoewijzing

Nieuwe wijzigingen aanbrengen aan Galgje

Dit programma was veel langer dan het programma Drakenrijk en het was ook veel ingewikkelder. Het helpt echt om een stroomdiagram te tekenen of een schets te maken om goed te bedenken hoe je wilt dat alles werkt.

Nu je een basisversie van het spel Galgje hebt gemaakt, kunnen we kijken of we het nog verder kunnen verbeteren met nieuwe functies...

Als je Galgje een paar keer hebt gespeeld, ben je misschien tot de conclusie gekomen dat zes beurten niet genoeg is om veel van de woorden te raden. Je kunt de speler gemakkelijk meer beurten geven door meer multiregel-strings toe te voegen aan de lijst met `GALGTEKENINGEN`.

Sla je programma *galgje.py* op als *galgje2.py* en voeg dan het volgende toe:

```
58. ==========''', '''
59.   +----+
60.   |    |
61.  [O    |
62.  /|\   |
63.  / \   |
64.        |
65. ==========''', '''
66.   +----+
67.   |    |
68.  [O]   |
69.  /|\   |
70.  / \   |
```

```
 71.        |
 72. ==========''']
```

Er worden twee nieuwe multiregel-strings toegevoegd aan de lijst GALGTEKENING, een met het linkeroor aan het hoofd getekend en een met beide oren getekend. Omdat het programma de speler vertelt dat ze hebben verloren als het aantal beurten gelijk is aan het aantal strings in de lijst GALGTEKENING (min 1), is dit de enige wijziging die je hoeft te maken.

Je kunt ook de lijst met woorden veranderen door de woorden te wijzigen op regel 59. In plaats van dieren kun je bijvoorbeeld kleuren nemen:

```
59. woorden = 'rood oranje geel groen blauw turquoise lila wit zwart
bruin'.split()
```

Of vormen:

```
59. woorden = 'vierkant driehoek rechthoek cirkel ellips ruit trapezium chevron
pentagon hexagon septagon octagon'.split()
```

Of fruit:

```
59. woorden = 'appel sinaasappel citroen limoen peer watermeloen druif
grapefruit kers banaan cantaloupe mango aardbei framboos'.split()
```

Dictionary's

Door de code een beetje aan te passen, kun je het spel zo veranderen dat je al deze woorden kunt gebruiken in het spel als aparte sets. Het programma kan de speler dan vertellen uit welke set (dier, kleur, vorm of fruit) het geheime woord afkomstig is.

Als je dit wilt doen, heb je een nieuw datatype nodig: de **dictionary**. Een dictionary (woordenboek) is een verzameling waarden net zoals in een lijst. Maar in plaats van naar de items te verwijzen met een indexnummer, kun je naar de items verwijzen met een index van een ander datatype. Voor dictionary's worden die indexen **trefwoorden** genoemd.

Bij het definiëren van een dictionary gebruik je accolades { en } in plaats van de vierkante haken [en] die je bij het definiëren van lijsten gebruikt. Probeer het volgende eens in de interactieve shell te typen:

```
>>> dingen = {'hallo':'Hallo daar, hoe gaat het?', 4:'spek',
'boterhamworst':9999 }
```

De waarden tussen de accolades heten **trefwoord-waarde-paren**. De trefwoorden staan links van de dubbele punt en de waarde staat rechts. Je kunt naar de waarden verwijzen aan de hand van het trefwoord. Je zet het trefwoord dan tussen vierkante haken. Typ het volgende in de interactieve shell:

```
>>> dingen = {'hallo':'Hallo daar, hoe gaat het?', 4:'spek',
'boterhamworst':9999}
>>> dingen['hallo']
'Hallo daar, hoe gaat het?'
>>> dingen[4]
'spek'
>>> dingen['boterhamworst']
'9999'
```

In plaats van een integer tussen de vierkante haken te zetten zoals je dat met lijsten deed, gebruik je een trefwoord. Dat levert dan de waarde voor dat trefwoord op.

De lengte van dictionary's uitvinden met de functie `len()`

Met behulp van de functie `len()` kun je uitvinden hoeveel trefwoord-waarde-paren er in de dictionary zitten. Probeer het volgende eens in de interactieve shell te typen:

```
>>> dingen = {'hallo':'Hallo daar, hoe gaat het?', 4:'spek',
'boterhamworst':9999}
>>> len(dingen)
3
```

Het verschil tussen dictionary's en lijsten

Dictionary's kunnen trefwoorden hebben van elk datatype, niet alleen strings. Maar onthoud wel dat aangezien `0` en `'0'` verschillende waarden zijn, ze ook verschillende trefwoorden zijn. Probeer dit eens te typen in de interactieve shell:

```
>>> dingen = {'0':'een string', 0:'een integer'}
>>> dingen[0]
'een integer'
>>> dingen['0']
'een string'
```

Je kunt ook door de trefwoorden in dictionary's gaan met behulp van een `for`-lus. Probeer het volgende in de interactieve shell te typen:

```
>>> favorieten = {'fruit':'appels', 'dieren':'katten', 'nummer':42}
>>> for i in favorieten:
```

```
...     print(i)
dieren
fruit
nummer
>>> for i in favorieten:
...     print(favorieten[i])
katten
appels
42
```

Het verschil tussen dictionary's en lijsten is dat de waarden in dictionary's niet in een bepaalde volgorde staan. Het eerste item in een lijst met de naam `lijstInhoud` zou `lijstInhoud[0]` zijn. Maar er is geen 'eerste' item in een dictionary, omdat de items niet op volgorde staan. Probeer het volgende eens in de interactieve shell te typen:

```
>>> favorieten1 = {'fruit':'appels', 'nummer':42, 'dieren':'katten'}
>>> favorieten2 = {'dieren':'katten', 'nummer':42, 'fruit':'appels'}
>>> favorieten1 == favorieten2
True
```

De expressie met de voorwaarde `favorieten1 == favorieten2` retourneert `True` omdat dictionary's geen volgorde hebben en ze gezien worden als gelijk aan elkaar als ze dezelfde trefwoord-waarde-paren bevatten. Lijsten daarentegen, hebben altijd een volgorde, dus twee lijsten met dezelfde items maar in een andere volgorde worden niet geëvalueerd als gelijk aan elkaar. Probeer dit eens te typen in de interactieve shell:

```
>>> lijstFavorieten1 = ['appels', 'katten', 42]
>>> lijstFavorieten2 = ['katten', 42, 'appels']
>>> lijstFavorieten1 == lijstFavorieten2
False
```

Dictionary's kennen twee nuttige methoden, `keys()` en `values()`. Deze methoden retourneren respectievelijk waarden van het type `dict_keys` en `dict_values`. Net zoals dat het geval is bij reeksen (zie hoofdstuk 9) kunnen de waarden van die datatypen gemakkelijk worden omgezet in lijsten met de functie `list()`. Probeer het volgende eens in de interactieve shell te typen:

```
>>> favorieten = {'fruit':'appels', 'dieren':'katten', 'nummer':42}
>>> list(favorieten.keys())
['fruit', 'nummer', 'dieren']
>>> list(favorieten.values())
['appels', 42, 'katten']
```

Sets met woorden voor Galgje

Laten we de code voor het spel Galgje een beetje wijzigen om verschillende sets met geheime woorden te kunnen gebruiken. Eerst wijzigen we de waarde in `woorden` in een dictionary waarvan de trefwoorden strings zijn en de waarden lijsten met strings zijn. De methode `split()` voor strings zet de string om in een lijst met strings die elk één woord hebben.

```
 59. woorden = {'Kleuren':'rood oranje geel groen blauw turquoise lila wit
zwart bruin'.split(),
 60. 'Vormen':'vierkant driehoek rechthoek cirkel ellips ruit trapezium chevron
pentagon hexagon septagon octagon'.split(),
 61. 'Fruit':'appel sinaasappel citroen limoen peer watermeloen druif
grapefruit kers banaan cantaloupe mango aardbei framboos'.split(),
 62. 'Dieren':'baviaan bever cobra cougar duif eend eland forel gans geit haai
hagedis havik hond kalkoen kikker konijn kraai leeuw luiaard mier mossel
muilezel neushoorn ooievaar otter panda papegaai python ram salamander
schildpad slang stinkdier tijger vleermuis vos walvis wezel wombat zalm
zebra'.split()}
```

Deze code staat op meer dan één regel in het bestand, maar Python ziet het gewoon als één regel. (De regel code eindigt pas met de laatste accolade.)

De functie `random.choice()`

De functie `choice()` in de module random krijgt een aantal lijstargumenten doorgegeven en retourneert daar een willekeurige waarde van, net zoals dat ging met je functie `haalWillekeurigWoord()`. Je gaat `random.choice()` gebruiken in de nieuwe versie van de functie `haalWillekeurigWoord()`.

Om de functie `random.choice()` aan het werk te zien, kun je het volgende typen in de interactieve shell:

```
>>> import random
>>> random.choice(['kat', 'hond', 'muis'])
'muis'
>>> random.choice(['kat', 'hond', 'muis'])
'kat'
>>> random.choice([2, 22, 222, 223])
2
>>> random.choice([2, 22, 222, 223])
222
```

Wijzig de functie `haalWillekeurigWoord()` zodat de parameter ervan een dictionary van lijsten met strings is, in plaats van alleen een lijst met strings. De functie zag er eerst zo uit:

```
def haalWillekeurigWoord(woordenLijst):
    # Deze functie retourneert een willekeurige string uit een doorgegeven
lijst met strings.
    woordIndex = random.randint(0, len(woordenLijst) - 1)
    return woordenLijst[woordIndex]
```

Wijzig de code in deze functie zodat het er als volgt uitziet:

```
def haalWillekeurigWoord(woordenDict):
    # Deze functie retourneert een willekeurige string van de doorgegeven
dictionary met lijsten met strings, en ook het trefwoord.
    # Selecteer eerst een willekeurig trefwoord uit de dictionary:
    trefwoord = random.choice(list(woordenDict.keys()))
    # Selecteer daarna een willekeurig woord uit de lijst met trefwoorden in de
dictionary:
    woordIndex = random.randint(0, len(woordenDict[trefwoord]) - 1)
    return [woordenDict[trefwoord][woordIndex], trefwoord]
```

De naam van de parameter veranderen we in `woordDict` zodat deze de inhoud beter beschrijft. In plaats van een willekeurig woord te kiezen uit een lijst met strings, kiest de functie nu eerst een willekeurig trefwoord in de dictionary `woordenDict` door `random.choice()` aan te roepen.

In plaats van de string `woordenLijst[woordIndex]` te retourneren, retourneert de functie nu een lijst met twee items. Het eerste item is `woordenDict[trefwoord][woordIndex]`. Het tweede item is `trefwoord`.

Een dictionary met lijsten evalueren

De expressie `woordenDict[trefwoord][woordIndex]` ziet er misschien ingewikkeld uit, maar het is gewoon een expressie die je stap voor stap kunt evalueren, net zoals we dat al eerder hebben gedaan. Bedenk eerst dat `trefwoord` de waarde `'Fruit'` had (wat werd gekozen op regel 65) en `woordIndex` heeft de waarde 5 (gekozen op regel 68).
`woordenDict[trefwoord][woordIndex]` wordt als volgt geëvalueerd:

```
woordenDict[trefwoord][woordIndex]
           ▼
woordenDict['Fruit'][woordIndex]
           ▼
['appel sinaasappel citroen limoen peer watermeloen druif grapefruit kers
banaan cantaloupe mango aardbei framboos'][woordIndex]
           ▼
```

```
['appel sinaasappel citroen limoen peer watermeloen druif grapefruit kers
banaan cantaloupe mango aardbei framboos'][5]
           ▼
      'watermeloen'
```

In het bovenstaande geval zou het item in de lijst dat door deze functie wordt geretourneerd, de string `'watermeloen'` zijn. (Vergeet niet dat indexen altijd bij 0 beginnen, dus `[5]` verwijst naar het 6e item in de lijst.)

Omdat de functie `haalWillekeurigWoord()` nu een lijst retourneert met twee items in plaats van een string, krijgt `geheimWoord` een lijst toegewezen, niet een string. Je kunt deze twee items in twee aparte variabelen opslaan met een trucje voor meervoudige toewijzing.

Meervoudige toewijzing

Je kunt meerdere variabelen opgeven, van elkaar gescheiden door een komma, aan de linkerkant van een toewijzingsstatement. Typ het volgende in de interactieve shell:

```
>>> a, b, c = ['appels', 'katten', 42]
>>> a
'appels'
>>> b
'katten'
>>> c
42
```

Het trucje werkt alleen als je even veel variabelen links van het = teken zet als waarden aan de rechterkant. Python wijst automatisch de eerste waarde toe aan de eerste variabele, de tweede waarde aan de tweede variabele, enzovoort. Maar als je niet evenveel variabelen hebt als waarden, geeft Python een foutmelding.

```
>>> a, b, c, d = ['appels', 'katten', 42, 10, 'hallo']
Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    a, b, c, d = ['appels', 'katten', 42, 10, 'hallo']
ValueError: too many values to unpack

>>> a, b, c, d = ['appels', 'katten']
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    a, b, c = ['appels', 'katten']
ValueError: need more than 2 values to unpack
```

Wijzig je code in Galgje om dit trucje te doen met de retourwaarde van `haalWillekeurigWoord()`:

```
108. goedeLetters = ''
109. geheimWoord, geheimTrefwoord = haalWillekeurigWoord(woorden)
110. klaarMetSpel = False
...
144. klaarMetSpel = False
145. geheimWoord, geheimTrefwoord = haalWillekeurigWoord(woorden)
146. else:
```

De woordcategorie laten zien aan de speler

De laatste wijziging die je gaat maken, is dat je de speler gaat vertellen uit welke categorie ze moeten raden. Dan weet de speler dat het geheime woord een dier, kleur, vorm of fruitsoort is. Voeg deze regel toe achter regel 112. Hier is de oorspronkelijke code:

```
112. while True:
113.     toonGalg(GALGTEKENING, fouteLetters, goedeLetters, geheimWoord)
```

Voeg de regel toe zodat de code er nu als volgt uitziet:

```
112. while True:
113.     print('Het geheime woord komt uit de categorie: ' + geheimTrefwoord)
114.     toonGalg(GALGTEKENING, fouteLetters, goedeLetters, geheimWoord)
```

Nu ben je klaar met je wijzigingen aan Galgje. In plaats van één lijst met strings, wordt het geheime woord nu gekozen uit een paar verschillende lijsten met strings. Het programma vertelt de speler ook uit welke categorie het geheime woord komt. Speel deze versie eens. Je kunt de dictionary `woorden` op regel 59 gemakkelijk wijzigen met nog meer categorieën met woorden.

Samenvatting

We zijn klaar met Galgje. Zelfs als je klaar bent met een programma, kun je daarna nog meer functies toevoegen, als je meer leert over programmeren.

Dictionary's lijken op lijsten maar kunnen elke soort waarde gebruiken als index. Met een index wijs je naar een bepaald item. Bij lijsten doe je dat met een integer die de volgorde in de lijst aanwijst; bij dictionary's doe je dat met een trefwoord. Trefwoorden kunnen strings of integers zijn of elke andere waarde van elk ander datatype.

De truc van meervoudig toewijzen is handig als je meerdere waarden aan meerdere variabelen moet toewijzen.

Galgje was behoorlijk ingewikkeld vergeleken met de eerdere spelletjes in dit boek. Maar nu je op dit punt bent aangekomen, weet je al heel veel van de basisbegrippen in het schrijven van programma's: je weet nu over variabelen, lussen, functies en de datatypen in Python, zoals lijsten en dictionary's. De programma's die hierna volgen zijn ook best ingewikkeld, maar je hebt het 'steilste stuk van de programmeerberg' nu eigenlijk achter de rug!

Hoofdstuk 10

BOTER-KAAS-EN-EIEREN

In dit hoofdstuk behandelen we:
- Kunstmatige intelligentie
- Lijstverwijzingen
- Evalueren met kortsluiting
- De waarde `None` (Geen)

In dit hoofdstuk spelen we Boter-kaas-en-eieren tegenover een simpele vorm van kunstmatige intelligentie. **Kunstmatige intelligentie** (dit gaan we afkorten als **KI**) is een computerprogramma dat op een intelligente manier kan reageren op de zetten van een speler. In dit spel komen geen ingewikkelde nieuwe begrippen aan de orde. De kunstmatige intelligentie die Boter-kaas-en-eieren met je gaat spelen, is afkomstig uit slechts een paar regeltjes code.

Normaal spelen twee mensen Boter-kaas-en-eieren met papier en potlood. De ene speler is X en de andere speler is O. De spelers zetten om beurten hun teken op een bord met 9 vakjes. Als een speler drie van zijn tekens op een rij krijgt op het bord, heeft hij gewonnen. Wanneer het bord helemaal is gevuld en niemand heeft gewonnen, is het gelijkspel.

Wij maken in dit hoofdstuk gebruik van de kennis die we in de eerdere hoofdstukken hebben opgedaan en we oefenen daar nog een beetje mee. Laten we om te beginnen eens kijken naar de voorbeelduitvoer van dit programma. De spelers doen hun zet door het getal te typen van het vakje waar ze in willen gaan staan. De vakjes zijn genummerd zoals op het toetsenblok van je toetsenbord (zie Afbeelding 10-2).

Voorbeelduitvoer van Boter-kaas-en-eieren

```
Welkom bij Boter-kaas-en-eieren!
Wil je X zijn of O?
X
De computer mag eerst.
   |   |
 O |   |
   |   |
-----------
   |   |
   |   |
   |   |
```

```
-----------
   |   |
   |   |
   |   |
Wat is je volgende zet? (1-9)
3
   |   |
 O |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
 O |   | X
   |   |
Wat is je volgende zet? (1-9)
4
   |   |
 O |   | O
   |   |
-----------
   |   |
 X |   |
   |   |
-----------
   |   |
 O |   | X
   |   |
Wat is je volgende zet? (1-9)
5
   |   |
 O | O | O
   |   |
-----------
   |   |
 X | X |
   |   |
-----------
   |   |
 O |   | X
   |   |
De computer heeft gewonnen! Volgende keer beter...
Wil je nog een keer spelen? (ja of nee)
nee
```

Broncode van Boter-kaas-en-eieren

Typ de volgende broncode in een nieuw bestand in de bestandseditor van Python en sla het op als *boter.py*. Voer het spel dan uit door op **F5** te drukken.

boter.py

```
 1. # Boter-kaas-en-eieren
 2.
 3. import random
 4.
 5. def tekenBord(bord):
 6.     # Deze functie tekent het bord dat aan de functie is doorgegeven.
 7.
 8.     # "bord" is een lijst met 10 strings die het bord weergeeft (we
negeren indexnummer 0)
 9.     print('   |   |')
10.     print(' ' + bord[7] + ' | ' + bord[8] + ' | ' + bord[9])
11.     print('   |   |')
12.     print('-----------')
13.     print('   |   |')
14.     print(' ' + bord[4] + ' | ' + bord[5] + ' | ' + bord[6])
15.     print('   |   |')
16.     print('-----------')
17.     print('   |   |')
18.     print(' ' + bord[1] + ' | ' + bord[2] + ' | ' + bord[3])
19.     print('   |   |')
20.
21. def kiesLetter():
22.     # Laat de speler de letter typen die ze willen zijn.
23.     # Retourneert een lijst, met de letter van de speler als eerste item
en de letter van de computer als tweede item.
24.     letter = ''
25.     while not (letter == 'X' or letter == 'O'):
26.         print('Wil je X zijn of O?')
27.         letter = input().upper()
28.
29.     # het eerste element in de lijst is de letter van de speler, het
tweede item is de letter van de computer.
30.     if letter == 'X':
31.         return ['X', 'O']
32.     else:
33.         return ['O', 'X']
34.
35. def wieMagEerst():
36.     # Wie het eerst mag, wordt willekeurig gekozen in deze functie.
37.     if random.randint(0, 1) == 0:
```

```
 38.         return 'computer'
 39.     else:
 40.         return 'speler'
 41.
 42. def nogmaalsSpelen():
 43.     # Deze functie retourneert True als de speler nog een keer wil spelen,
anders is het False.
 44.     print('Wil je nog een keer spelen? (ja of nee)')
 45.     return input().lower().startswith('j')
 46.
 47. def doeZet(bord, letter, zet):
 48.     bord[zet] = letter
 49.
 50. def isWinnaar(bo, le):
 51.     # Gegeven het bord en de letter van de speler, retourneert deze
functie True als de speler heeft gewonnen.
 52.     # We korten bord en letter af tot bo en le om minder te hoeven typen.
 53.     return ((bo[7] == le and bo[8] == le and bo[9] == le) or # bovenste
rij
 54.     (bo[4] == le and bo[5] == le and bo[6] == le) or # middelste rij
 55.     (bo[1] == le and bo[2] == le and bo[3] == le) or # onderste rij
 56.     (bo[7] == le and bo[4] == le and bo[1] == le) or # verticaal links
 57.     (bo[8] == le and bo[5] == le and bo[2] == le) or # verticaal midden
 58.     (bo[9] == le and bo[6] == le and bo[3] == le) or # verticaal rechts
 59.     (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonaal
 60.     (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonaal
 61.
 62. def haalKopieBord(bord):
 63.     # We maken een kopie van de bordlijst en retourneren die kopie.
 64.     kopieBord = []
 65.
 66.     for i in bord:
 67.         kopieBord.append(i)
 68.
 69.     return kopieBord
 70.
 71. def isRuimteVrij(bord, zet):
 72.     # Retourneert True als de zet die is doorgegeven, mogelijk is op het
bord dat is doorgegeven.
 73.     return bord[zet] == ' '
 74.
 75. def haalSpelerZet(bord):
 76.     # We laten de speler zijn zet invoeren.
 77.     zet = ' '
 78.     while zet not in '1 2 3 4 5 6 7 8 9'.split() or not isRuimteVrij(bord,
int(zet)):
 79.         print('Wat is je volgende zet? (1-9)')
```

```
 80.         zet = input()
 81.     return int(zet)
 82.
 83. def kiesWillekeurigeZetUitLijst(bord, zettenLijst):
 84.     # Retourneert een geldige zet uit de lijst die is doorgegeven op het
bord dat is doorgegeven.
 85.     # Retourneert None als er geen geldige zet is.
 86.     mogelijkeZetten = []
 87.     for i in zettenLijst:
 88.         if isRuimteVrij(bord, i):
 89.             mogelijkeZetten.append(i)
 90.
 91.     if len(mogelijkeZetten) != 0:
 92.         return random.choice(mogelijkeZetten)
 93.     else:
 94.         return None
 95.
 96. def haalComputerZet(bord, computerLetter):
 97.     # Gegeven een bord en de letter van de computer, bepaalt deze functie
welke zet de computer gaat doen en wordt die zet geretourneerd.
 98.     if computerLetter == 'X':
 99.         spelerLetter = 'O'
100.     else:
101.         spelerLetter = 'X'
102.
103.     # Hier is ons algoritme voor onze KI in Boter-kaas-en-eieren:
104.     # Eerst kijken we of we in de volgende zet kunnen winnen
105.     for i in range(1, 10):
106.         kopie = haalKopieBord(bord)
107.         if isRuimteVrij(kopie, i):
108.             doeZet(kopie, computerLetter, i)
109.             if isWinnaar(kopie, computerLetter):
110.                 return i
111.
112.     # Controleert of de speler bij zijn volgende zet kan winnen, en
blokkeert die zet.
113.     for i in range(1, 10):
114.         kopie = haalKopieBord(bord)
115.         if isRuimteVrij(kopie, i):
116.             doeZet(kopie, spelerLetter, i)
117.             if isWinnaar(kopie, spelerLetter):
118.                 return i
119.
120.     # Probeert een van de hoeken te bezetten, als ze leeg zijn.
121.     zet = kiesWillekeurigeZetUitLijst(bord, [1, 3, 7, 9])
122.     if zet != None:
123.         return zet
```

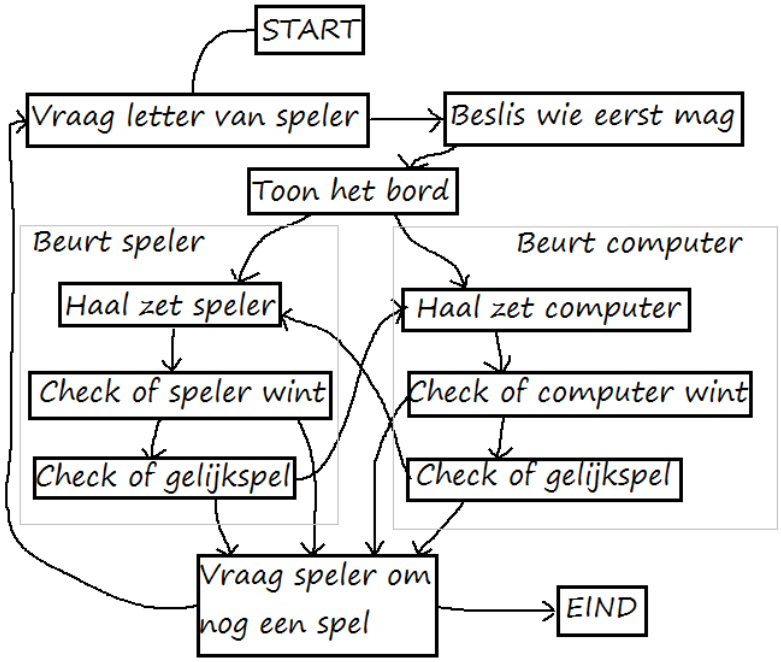
```
124.
125.     # Probeert het middelste vak te bezetten als het nog leeg is.
126.     if isRuimteVrij(bord, 5):
127.         return 5
128.
129.     # Zet op een van de zijkanten.
130.     return kiesWillekeurigeZetUitLijst(bord, [2, 4, 6, 8])
131.
132. def isBordVol(bord):
133.     # Retourneert True als elk vakje op het bord is bezet. Zo niet,
retourneert de functie False.
134.     for i in range(1, 10):
135.         if isRuimteVrij(bord, i):
136.             return False
137.     return True
138.
139.
140. print('Welkom bij Boter-kaas-en-eieren!')
141.
142. while True:
143.     # Wist het bord
144.     hetBord = [' '] * 10
145.     spelerLetter, computerLetter = kiesLetter()
146.     beurt = wieMagEerst()
147.     print('De ' + beurt + ' mag eerst.')
148.     spelBezig = True
149.
150.     while spelBezig:
151.         if beurt == 'speler':
152.             # De speler is aan de beurt.
153.             tekenBord(hetBord)
154.             zet = haalSpelerZet(hetBord)
155.             doeZet(hetBord, spelerLetter, zet)
156.
157.             if isWinnaar(hetBord, spelerLetter):
158.                 tekenBord(hetBord)
159.                 print('Hoera! Jij hebt gewonnen!')
160.                 spelBezig = False
161.             else:
162.                 if isBordVol(hetBord):
163.                     tekenBord(hetBord)
164.                     print('Het is gelijkspel!')
165.                     break
166.                 else:
167.                     beurt = 'computer'
168.
169.         else:
```

```
170.             # De computer is aan de beurt..
171.             zet = haalComputerZet(hetBord, computerLetter)
172.             doeZet(hetBord, computerLetter, zet)
173.
174.             if isWinnaar(hetBord, computerLetter):
175.                 tekenBord(hetBord)
176.                 print('De computer heeft gewonnen! Volgende keer
beter...')
177.                 spelBezig = False
178.             else:
179.                 if isBordVol(hetBord):
180.                     tekenBord(hetBord)
181.                     print('Het is gelijkspel!')
182.                     break
183.                 else:
184.                     beurt = 'speler'
185.
186.     if not nogmaalsSpelen():
187.         break
```

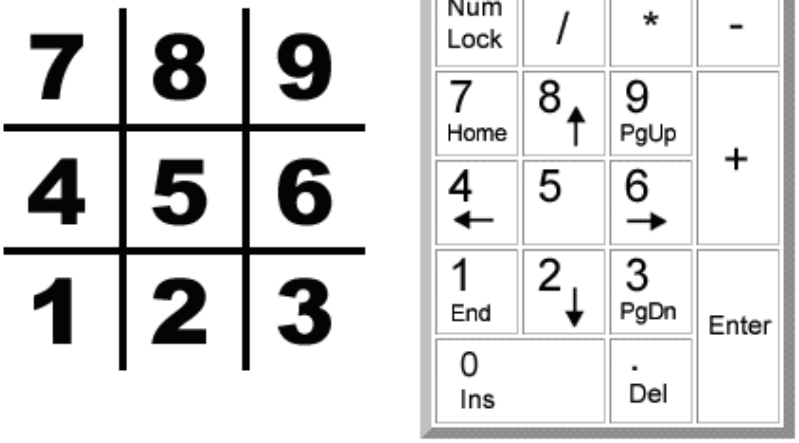
Het programma ontwerpen

In Afbeelding 10-1 zie je een voorbeeld van een mogelijk stroomdiagram voor Boter-kaas-en-eieren. In het programma Boter-kaas-en-eieren kiest de speler of hij X of O wil zijn. Er wordt willekeurig gekozen wie het eerst mag. Vervolgens mogen de speler en de computer om beurten een zet doen.

De vakken links in het stroomdiagram tonen wat er gebeurt tijdens de beurt van de speler. De rechterkant laat zien wat er gebeurt tijdens de beurt van de computer. Zodra de speler of de computer een zet hebben gedaan, controleert het programma of ze hebben gewonnen of dat het gelijkspel is, en dan is de ander aan de beurt. Zodra het spel over is, vraagt het programma aan de speler of ze nog een keer willen spelen.



Afbeelding 10-1: Stroomdiagram voor Boter-kaas-en-eieren



Afbeelding 10-2: Het bord is genummerd zoals op het toetsenblok van je toetsenbord.

Het bord weergeven als gegevens

Eerst moet je verzinnen hoe je een Boter-kaas-en-eieren-bord kunt omvormen tot gegevens in variabelen. Op papier wordt een Boter-kaas-en-eieren-bord getekend als twee parallelle horizontale lijnen en twee parallelle verticale lijnen, met ofwel een X, een O of een lege plek in alle negen vakjes.

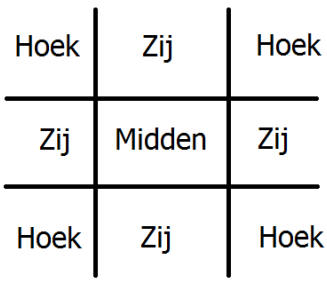
In het programma wordt het Boter-kaas-en-eieren-bord weergegeven als een lijst met strings. Elke string stelt een van de negen vakjes op het bord voor. Om het gemakkelijker te maken te onthouden welk cijfer voor welk vakje staat, gebruiken we dezelfde layout als op het toetsenblok van je toetsenbord. Dit zie je in Afbeelding 10-2.

De strings zijn ofwel `'X'` voor de speler die X is, of `'O'` voor de speler die O is, of een lege ruimte `' '` als er nog niks in het vakje staat.

Dus als in een variabele met de naam `bord` een lijst met tien strings wordt bewaard, bedoelen we met `bord[7]` het vakje linksboven op het bord. En op dezelfde manier zou `bord[5]` het middelste vakje zijn, `bord[4]` het vakje midden links, enzovoort. Waarom een lijst met tien strings en niet negen? We hebben toch maar 9 vakjes? Nou, er is geen 0 op het bord voor ons spel, dus we laten indexnummer `0` in dit programma voor het gemak buiten beschouwing. Maar omdat indexnummers in lijsten in Python nou eenmaal altijd met 0 beginnen, hebben we dus wel 10 items in onze lijst. De speler voert een getal in van 1 tot 9 om te zeggen welk vakje ze willen bezetten.

De KI in het spel

De KI moet naar het bord kunnen kijken en beslissen naar welk vakje het moet gaan. Voor de duidelijkheid noemen we de drie soorten vakjes op het Boter-kaas-en-eieren-bord: hoekvakjes, zijvakjes en het middenvakje. In Afbeelding 10-3 zie je een diagram van elk soort vakje.



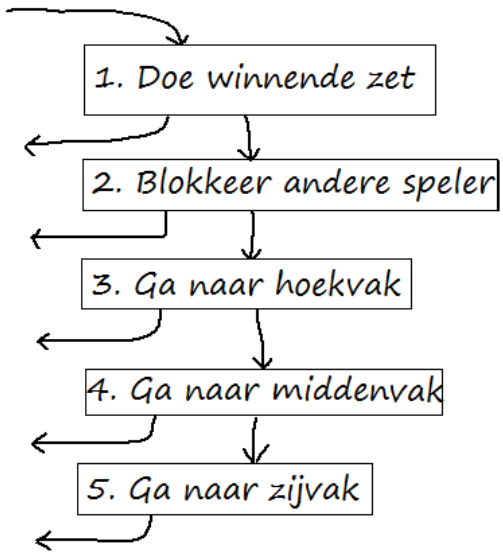
Afbeelding 10-3: Locaties van de zijvakjes, hoekvakjes en het middenvakje.

De KI speelt Boter-kaas-en-eieren volgens een eenvoudig algoritme. Een **algoritme** is een eindig aantal stappen waarmee je een bepaald gewenst resultaat kunt verkrijgen. In een computerprogramma kunnen meerdere verschillende algoritmen worden gebruikt. Je kunt een algoritme weergeven met een stroomdiagram. Het algoritme dat KI gebruikt, probeert de beste zet te bedenken, zoals je kunt zien in Afbeelding 10-4.

Het algoritme van KI bevat de volgende stappen:

1. Kijk eerst of er een zet is die KI kan doen om het spel te winnen. Als dat zo is, doe die zet. Zo niet, ga naar stap 2.
2. Kijk of er een zet is die de speler kan doen waarmee KI het spel zou verliezen. Als dat zo, ga naar dat vakje om de speler te blokkeren. Zo niet, ga naar stap 3.
3. Kijk of een van de hoeken vrij is (de vakken 1, 3, 7 of 9). (Een hoekvak is altijd beter dan het midden of een zijvak.) Als er geen hoekvak vrij is, ga naar stap 4.
4. Kijk of het midden vrij is. Als dat zo is, ga daarheen. Als dat niet zo is, ga naar stap 5.
5. Ga naar een van zijvakjes (de vakjes 2, 4, 6 of 8). Er zijn geen stappen meer, want als de uitvoering bij stap 5 is gekomen, zijn de zijvakjes de enige vakjes die nog over zijn.

Deze stappen vinden allemaal plaats in het vak 'Haal computerzetten op' in het stroomdiagram. Je zou deze informatie op de volgende manier aan het stroomdiagram kunnen toevoegen:



Afbeelding 10-4: De vijf stappen van het algoritme voor het algoritme 'Haal computerzetten op'. De pijlen die uit dit vak komen, gaan naar het vak 'Controle of computer heeft gewonnen'.

Dit algoritme is verwerkt in de functie `haalComputerZet()` en in de andere functies die worden aangeroepen in `haalComputerZet()`.

Het begin van het programma

```
 1. # Boter-kaas-en-eieren
 2.
 3. import random
```

De eerste paar regels zijn commentaar en dienen voor het importeren van de module `random` zodat je straks de functie `randint()` kunt aanroepen.

Het bord op het scherm weergeven

```
 5. def tekenBord(bord):
 6.     # Deze functie tekent het bord dat aan de functie is doorgegeven.
 7.
 8.     # "bord" is een lijst met 10 strings die het bord weergeeft (we
negeren indexnummer 0)
 9.     print('   |   |')
10.     print(' ' + bord[7] + ' | ' + bord[8] + ' | ' + bord[9])
11.     print('   |   |')
12.     print('-----------')
13.     print('   |   |')
14.     print(' ' + bord[4] + ' | ' + bord[5] + ' | ' + bord[6])
15.     print('   |   |')
16.     print('-----------')
17.     print('   |   |')
18.     print(' ' + bord[1] + ' | ' + bord[2] + ' | ' + bord[3])
19.     print('   |   |')
```

De functie `tekenBord()` toont het spelbord, dat wordt voorgesteld door de parameter `bord`, op het scherm. Het bord is dus een lijst met tien strings, waarbij de string met indexnummer `1` in de lijst ook naar vakje 1 op het bord wijst. We negeren de string op indexnummer `0` dit keer. Veel van de functies in dit spel werken via het doorgeven van het bord als een lijst met tien strings aan die functies.

Let erop dat je de spaties goed typt in de strings, anders ziet het bord er raar uit op het scherm. Hier zie je een paar voorbeeldaanroepen (met een argument voor de parameter `bord`) naar `tekenBord()` en wat de functie dan zou weergeven op het scherm:

```
>>> tekenBord([' ', ' ', ' ', ' ', 'X', 'O', ' ', 'X', ' ', 'O']
   |   |
 X |   | O
   |   |
-----------
   |   |
 X | O |
   |   |
-----------
   |   |
   |   |
   |   |
>>> [' ', 'O', 'O', ' ', ' ', 'X', ' ', ' ', ' ', ' ']
   |   |
   |   |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
 O | O |
   |   |
>>> [' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
-----------
   |   |
   |   |
   |   |
```

De speler laten kiezen of ze X of O willen zijn

```
21. def kiesLetter():
22.     # Laat de speler de letter typen die ze willen zijn.
23.     # Retourneert een lijst, met de letter van de speler als eerste item
en de letter van de computer als tweede item.
24.     letter = ''
25.     while not (letter == 'X' or letter == 'O'):
26.         print('Wil je X zijn of O?')
```

```
27.         letter = input().upper()
```

De functie `kiesLetter()` vraagt of de speler X of O wil zijn. Door middel van een lus blijft de functie dit vragen, tot de speler een X of O typt. Regel 27 wijzigt de strings die worden geretourneerd door de functie `input()`, automatisch in hoofdletters met de stringmethode `upper()`.

De voorwaarde van de `while`-lus heeft haakjes, dus de expressie binnen de haakjes wordt eerst geëvalueerd. Als de variabele `letter` op `'X'` gezet zou zijn, zou de expressie als volgt worden geëvalueerd:

```
not (letter == 'X' or letter == 'O')
      ▼
not ('X' == 'X'    or 'X' == 'O')
      ▼
not (   True       or    False)
      ▼
not (True)
      ▼
not True
      ▼
    False
```

Als `letter` de waarde `'X'` of `'O'` heeft, is de voorwaarde van de while-lus `False` en gaat het programma dus verder, voorbij het while-blok.

```
29.     # het eerste element in de lijst is de letter van de speler, het
tweede item is de letter van de computer.
30.     if letter == 'X':
31.         return ['X', 'O']
32.     else:
33.         return ['O', 'X']
```

De functie retourneert een lijst met twee items. Het eerste item (de string op indexnummer `0` in deze lijst) is de letter van de speler en het tweede item (de string op indexnummer `1`) is de letter van de computer. Deze `if-else`-statement kiest welke lijst moet worden geretourneerd.

Bepalen wie eerst mag

```
35. def wieMagEerst():
36.     # Wie het eerst mag, wordt willekeurig gekozen in deze functie.
37.     if random.randint(0, 1) == 0:
38.         return 'computer'
39.     else:
```

```
40.         return 'speler'
```

De functie `wiemagEerst()` gooit een virtueel muntje op om te bepalen wie eerst mag. Het muntje wordt opgegooid door het aanroepen van `random.randint(0, 1)`. Als deze functieaanroep een `0` retourneert, retourneert de functie `wieMagEerst()` de string `'computer'`. In het andere geval retourneert de functie `'speler'`. De code die deze functie aanroept, gebruikt de retourwaarde van de functie om te zeggen wie het eerst mag.

De speler vragen of ze nog een keertje willen spelen

```
42. def nogmaalsSpelen():
43.     # Deze functie retourneert True als de speler nog een keer wil spelen,
anders is het False.
44.     print('Wil je nog een keer spelen? (ja of nee)')
45.     return input().lower().startswith('j')
```

De functie `nogmaalsSpelen()` vraagt de speler of ze nog een spelletje willen spelen. De functie levert `True` op als de speler `'ja'`, `'JA'`, `'j'`, of iets anders dat met de letter J begint, typt. Bij elk ander antwoord van de speler retourneert de functie `False`.

De volgorde van de methodeaanroepen op regel 45 is belangrijk. De retourwaarde van de aanroep naar de functie `input()` is een string waarop de methode `lower()` wordt toegepast. De methode `lower()` retourneert weer een andere string (met kleine letters) en op die string wordt weer de methode `startswith()` toegepast, dus dit levert uiteindelijk het argument `'j'` op, dat wordt doorgegeven.

Er is geen lus hier omdat het programma aanneemt dat als de gebruiker iets anders typt dan een string die met `'j'` begint, dat betekent dat ze niet meer willen spelen.

Een teken op het bord plaatsen

```
47. def doeZet(bord, letter, zet):
48.     bord[zet] = letter
```

De functie `doeZet()` is maar één regel lang en heel makkelijk. De parameters die de functie meekrijgt, zijn een lijst met tien strings genaamd `bord`, de letter van de speler (`'X'` of `'O'`) genaamd `letter` en het veld op het bord waar de speler heen wil (een integer van `1` tot `9`) genaamd `zet`.

Maar wacht eens even... Deze code lijkt een van de items in de lijst `bord` te wijzigen in de waarde die `letter` heeft. Maar deze code bevindt zich in een functie, dus de parameter `bord` wordt

vergeten zodra de functie zijn retourwaarde heeft gegeven. We hebben immers geleerd dat variabelen in een functie alleen een lokaal bereik hebben. Zou de wijziging aan `bord` dan ook niet worden vergeten?

Nou, dat is niet zo. En dat komt omdat lijsten iets speciaals hebben als je ze doorgeeft als argumenten aan functies. Wat je eigenlijk doet, is dat je een verwijzing naar de lijst meegeeft en niet de lijst zelf. Laten we even kijken naar de verschillen tussen lijsten en verwijzingen naar lijsten.

Verwijzingen

Probeer het volgende in de interactieve shell te typen:

```
>>> ham = 42
>>> kaas = ham
>>> ham = 100
>>> ham
100
>>> kaas
42
```

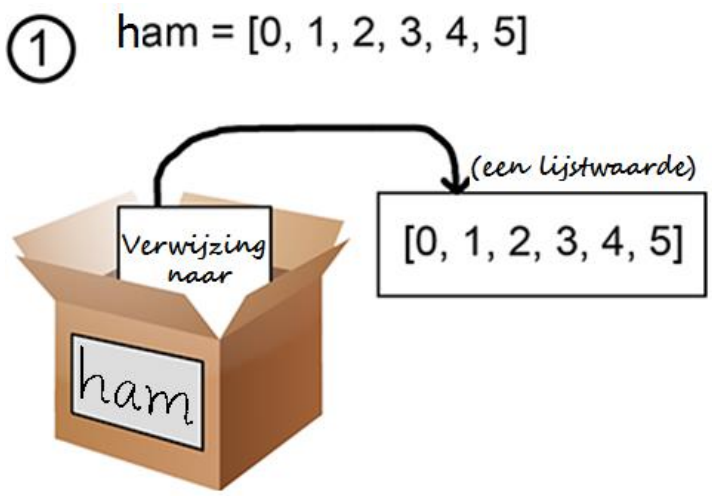
Nou, gebaseerd op wat je tot nu toe hebt geleerd, zal dit je niet verbazen. Je wijst `42` toe aan de variabele `ham` en wijst dan de waarde in de variabele `ham` toe aan de variabele met de naam `kaas`. Wanneer je `ham` daarna wijzigt in `100`, heeft dit geen effect op de waarde in `kaas`. Dat komt omdat `ham` en `kaas` verschillende variabelen zijn die verschillende waarden bevatten.

Maar lijsten werken op een andere manier. Wanneer je een lijst aan een variabele toewijst met het teken `=`, wijs je eigenlijk een lijstverwijzing toe aan de variabele. Een **verwijzing** is een waarde die *wijst* naar een stukje gegeven. Hier is wat code die helpt dit te begrijpen. Typ dit eens in de interactieve shell:

```
>>> ham = [0, 1, 2, 3, 4, 5]
>>> kaas = ham
>>> kaas[1] = 'Hallo!'
>>> ham
[0, 'Hallo!', 2, 3, 4, 5]
>>> kaas
[0, 'Hallo!', 2, 3, 4, 5]
```

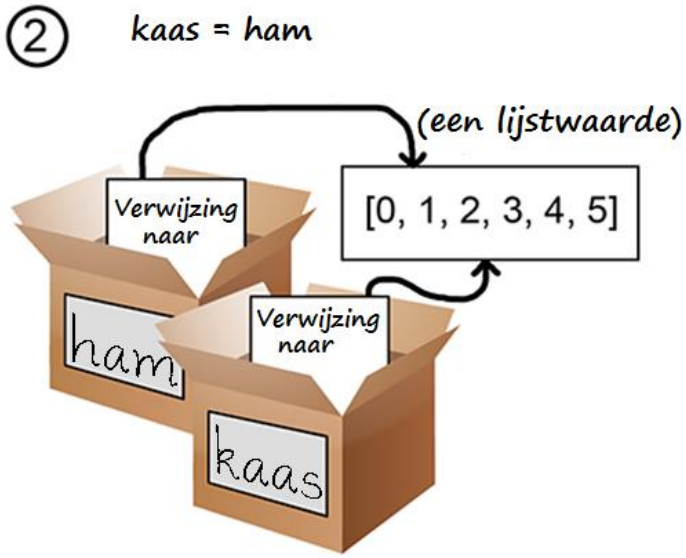
Dat ziet er best gek uit. De code wijzigde alleen maar de lijst `kaas`, maar toch zijn zowel de `kaas`- als de `ham`-lijst gewijzigd.

De variabele `ham` bevat niet de lijstwaarde zelf, maar alleen een verwijzing naar de lijst. Dit zie je in Afbeelding 10-5.



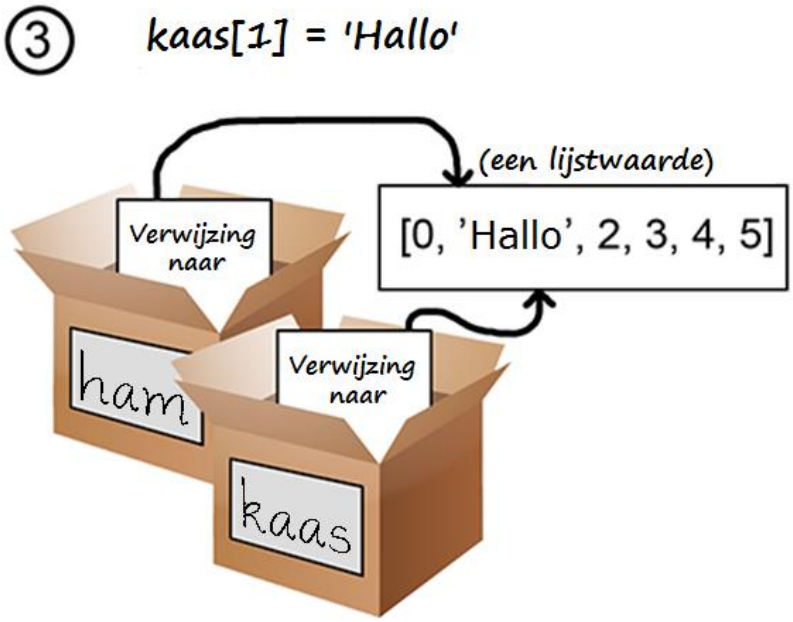
Afbeelding 10-5: Variabelen bevatten geen lijsten, maar alleen verwijzingen naar lijsten.

Zoals je ziet, kopieert de toewijzingsstatement `kaas = ham` de *lijstverwijzing* in `ham` naar `kaas`, zodat ze nu allebei naar hetzelfde item wijzen. Zowel `ham` als `kaas` bevatten nu een verwijzing die naar dezelfde lijstwaarde wijst. Er is maar één lijst. De lijst wordt niet gekopieerd, alleen de verwijzing naar de lijst wordt gekopieerd. In Afbeelding 10-6 zie je dit kopiëren.



Afbeelding 10-6: Twee variabelen bevatten twee verwijzingen naar dezelfde lijst.

Dus de regel `kaas[1] = 'Hallo!'` wijzigt dezelfde lijst als de lijst waar `ham` naar wijst. Daarom zie je dat `ham` dezelfde lijstwaarde heeft als `kaas`. Ze verwijzen allebei naar dezelfde lijst, zoals je ziet in Afbeelding 10-7.



Afbeelding 10-7: Als je de lijst wijzigt, wijzig je alle variabelen die verwijzingen naar die lijst bevatten.

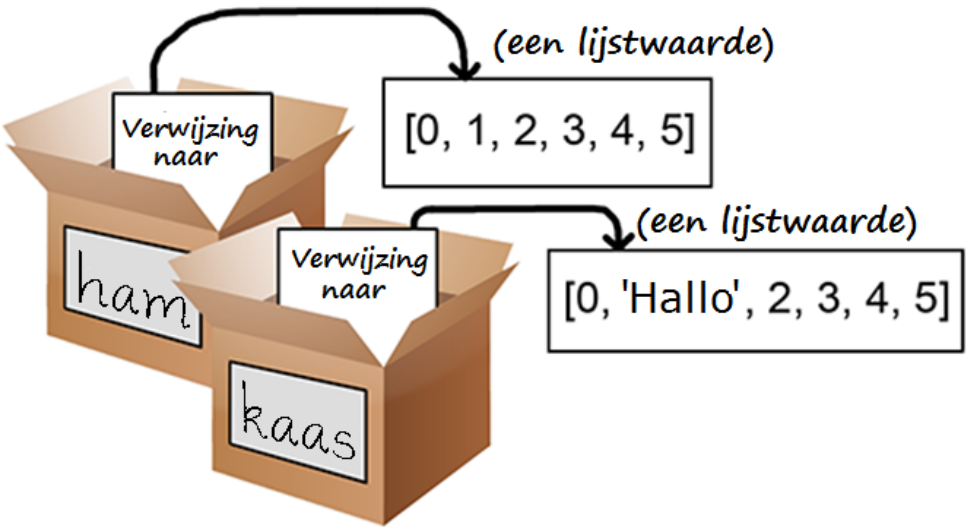
Als je wilt dat `ham` en `kaas` juist wél twee verschillende lijsten bevatten, moet je ook twee verschillende lijsten maken in plaats van een verwijzing te kopiëren:

```
>>> ham = [0, 1, 2, 3, 4, 5]
>>> kaas = [0, 1, 2, 3, 4, 5]
```

In het bovenstaande voorbeeld bevatten `ham` en `kaas` nu twee verschillende lijsten (ook al is de inhoud van deze lijsten identiek). Als je nu een van deze lijsten wijzigt, heeft dat geen effect op de andere lijst omdat `ham` en `kaas` nu verwijzingen bevatten naar twee verschillende lijsten:

```
>>> ham = [0, 1, 2, 3, 4, 5]
>>> kaas = [0, 1, 2, 3, 4, 5]
>>> kaas[1] = 'Hallo!'
>>> ham
[0, 1, 2, 3, 4, 5]
>>> kaas
[0, 'Hallo!', 2, 3, 4, 5]
```

Afbeelding 10-8 laat zien dat de twee verwijzingen nu naar verschillende lijsten wijzen.



Afbeelding 10-8: Twee variabelen die elk een verwijzing bevatten naar een andere lijst.

Dictionary's werken op dezelfde manier. Variabelen bevatten geen dictionary, maar verwijzingen naar dictionary's.

Lijstverwijzingen gebruiken en doeZet()

Laten we even teruggaan naar de functie `doeZet()`:

```
47. def doeZet(veld, letter, zet):
48.     bord[zet] = letter
```

Wanneer een lijstwaarde wordt doorgegeven voor de parameter `bord`, is de lokale variabele in de functie eigenlijk een kopie van de verwijzing naar de lijst, niet een kopie van de lijst. Maar een kopie van de verwijzing verwijst nog steeds naar dezelfde lijst als de oorspronkelijke verwijzing dat doet. Dus wijzigingen in `bord` in deze functie worden ook doorgevoerd in de oorspronkelijke lijst. En dat is hoe de functie `doeZet()` de lijst wijzigt waarvoor een verwijzing naar die lijst wordt doorgegeven.

De parameters `letter` en `zet` zijn kopieën van de string- en integerwaarden die je doorgeeft. Omdat ze kopieën zijn van de waarden, worden de oorspronkelijke waarden die je gebruikte toen je `doeZet()` aanriep, niet gewijzigd.

Controleren of de speler heeft gewonnen

```
 50. def isWinnaar(bo, le):
 51.     # Gegeven het bord en de letter van de speler, retourneert deze
functie True als de speler heeft gewonnen.
 52.     # We korten bord en letter af tot bo en le om minder te hoeven typen.
 53.     return ((bo[7] == le and e[8] == le and e[9] == le) or # bovenste rij
 54.     (bo[4] == le and bo[5] == le and bo[6] == le) or # middelste rij
 55.     (bo[1] == le and bo[2] == le and bo[3] == le) or # onderste rij
 56.     (bo[7] == le and bo[4] == le and bo[1] == le) or # verticaal links
 57.     (bo[8] == le and bo[5] == le and bo[2] == le) or # verticaal midden
 58.     (bo[9] == le and bo[6] == le and bo[3] == le) or # verticaal rechts
 59.     (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonaal
 60.     (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonaal
```

Regel 53 tot en met 60 in de functie `isWinnaar()` zijn eigenlijk één lange `return`-statement. De namen `bo` en `le` zijn afkortingen voor de parameters `bord` en `letter`. Deze kortere namen besparen je wat typewerk. Vergeet niet dat het Python niet kan schelen hoe je je variabelen noemt.

Er zijn acht mogelijke manieren om te winnen bij Boter-kaas-en-eieren. Je kunt een lijn krijgen op de bovenste, middelste of onderste rij. Of je kunt een lijn krijgen op de linker-, middelste of rechterkolom. Of je kunt een lijn krijgen op een van de twee diagonalen.

Elke regel van de voorwaarde controleert of de drie vakjes gelijk zijn aan de letter die is doorgegeven (gecombineerd met de operator `and`) en je gebruikt de operator `or` om de acht verschillende manieren om te winnen te combineren. Dit betekent dat slechts één van de acht manieren 'waar' moet zijn voordat we kunnen zeggen dat de speler die de letter in `le` bezit, de winnaar is.

Laten we doen alsof `le 'O'` is en `bo [' ', 'O', 'O', 'O', ' ', 'X', ' ', 'X', ' ', ' ']` is. Het bord ziet er dan als volgt uit:

```
   |   |
 X |   |
   |   |
-----------
   |   |
   | X |
   |   |
-----------
   |   |
 O | O | O
   |   |
```

Hier zie je hoe de expressie na het sleutelwoord `return` op regel 53 wordt geëvalueerd:

```
53.     return ((bo[7] == le and bo[8] == le and bo[9] == le) or # bovenste rij
54.     (bo[4] == le and bo[5] == le and bo[6] == le) or # middelste rij
55.     (bo[1] == le and bo[2] == le and bo[3] == le) or # onderste rij
56.     (bo[7] == le and bo[4] == le and bo[1] == le) or # verticaal links
57.     (bo[8] == le and bo[5] == le and bo[2] == le) or # verticaal midden
58.     (bo[9] == le and bo[6] == le and bo[3] == le) or # verticaal rechts
59.     (bo[7] == le and bo[5] == le and bo[3] == le) or # diagonaal
60.     (bo[9] == le and bo[5] == le and bo[1] == le)) # diagonaal
```

Eerst vervangt Python de variabele `bo` door de waarde die de variabele bevat:

```
    return (('X' == 'O' and ' ' == 'O' and ' ' == 'O') or
    (' ' == 'O' and 'X' == 'O' and ' ' == 'O') or
    ('O' == 'O' and 'O' == 'O' and 'O' == 'O') or
    ('X' == 'O' and ' ' == 'O' and 'O' == 'O') or
    (' ' == 'O' and 'X' == 'O' and 'O' == 'O') or
    (' ' == 'O' and ' ' == 'O' and 'O' == 'O') or
    ('X' == 'O' and 'X' == 'O' and 'O' == 'O') or
    (' ' == 'O' and 'X' == 'O' and 'O' == 'O'))
```

Vervolgens evalueert Python al die ==-vergelijkingen binnen de haakjes tot een Boole-waarde:

```
    return ((False and False and False) or
    (False and False and False) or
    (True and True and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True) or
    (False and False and True))
```

Daarna evalueert de Python-interpreter alle expressies binnen de haakjes:

```
    return ((False) or
    (False) or
    (True) or
    (False) or
    (False) or
    (False) or
    (False) or
    (False))
```

Aangezien er nu nog maar één waarde binnen de haakjes staat, kun je die haakjes wegdoen:

```
    return (False or
    False or
    True or
    False or
    False or
    False or
    False or
    False)
```

Nu wordt de expressie geëvalueerd die met al die `or`-operators is verbonden:

```
    return (True)
```

Doe de haakjes weer weg, en wat overblijft is:

```
    return True
```

Dus met deze waarden voor `bo` en `le`, zou de expressie de waarde `True` opleveren. Op die manier weet het programma of een van de twee spelers het spel heeft gewonnen of niet.

De bordgegevens kopiëren

```
62. def haalKopieBord(bord):
63.     # Maakt een kopie van de bordlijst en retourneert de kopie.
64.     kopieBord = []
65.
66.     for i in bord:
67.         kopieBord.append(i)
68.
69.     return kopieBord
```

Het nut van de functie `haalKopieBord()` is dat je zo een kopie kunt maken van de gegeven lijst met 10 strings die staat voor het Boter-kaas-en-eieren-bord in het spel. Soms wil je namelijk dat het algoritme van KI tijdelijke wijzigingen aan kan brengen op een tijdelijke kopie van het bord zonder het echte bord te veranderen. Dat is hoe KI kan 'nadenken'. In dat geval roep je deze functie aan om een kopie van de lijst van het bord te maken. De nieuwe lijst wordt gemaakt op regel 64, met vierkante haken zonder inhoud `[]`.

De lijst die wordt opgeslagen in `kopieBord` op regel 64 is eerst gewoon een lege lijst. De `for`-lus gaat door de parameter `bord` heen en voegt een kopie van de stringwaarden van het oorspronkelijke bord, toe achteraan de lijst van het kopiebord. Als de lus klaar is, wordt `kopieBord` geretourneerd. De functie `haalKopieBord()` bouwt een kopie op van het oorspronkelijke bord en retourneert een verwijzing naar dit nieuwe bord (niet naar het oorspronkelijke bord).

Controleren of een vakje op het bord nog vrij is

```
 71. def isRuimteVrij(bord, zet):
 72.     # Retourneert True als de zet die is doorgegeven, mogelijk is op het
bord dat is doorgegeven.
 73.     return bord[zet] == ' '
```

Dit is een simpele functie die bij een bepaald Boter-kaas-en-eieren-bord en een mogelijke zet, controleert of die zet mogelijk is of niet. Vergeet niet dat vrije vakjes op de bordlijsten worden gemarkeerd als een string van één spatie. Als het item op het indexnummer van het lege vakje niet gelijk is aan de string ' ', is de plaats al bezet.

De speler zijn zet laten doen

```
 75. def haalSpelerZet(bord):
 76.     # We laten de speler zijn zet invoeren.
 77.     zet = ' '
 78.     while zet not in '1 2 3 4 5 6 7 8 9'.split() or not isRuimteVrij(bord,
int(zet)):
 79.         print('Wat is je volgende zet? (1-9)')
 80.         zet = input()
 81.     return int(zet)
```

De functie `haalSpelerZet()` vraagt de speler een nummer in te voeren voor het vakje waar ze heen willen. De lus zorgt ervoor dat de verwerking pas verder gaat, als de speler een integer heeft

getypt in het bereik 1-9. Ook wordt gecontroleerd of het vakje dat wordt getypt, niet al bezet is, en daarvoor wordt het bord doorgegeven in de parameter `bord`.

De twee regels code in de `while`-lus vragen de speler gewoon een nummer in te voeren tussen 1 en 9. De voorwaarde op regel 78 is `True` als één van de expressies *links* of *rechts* van de operator `or` `True` is.

De expressie aan de *linker*kant controleert of de zet die de speler deed, gelijk is aan `'1'`, `'2'`, `'3'`, enzovoort tot `'9'` door een lijst te maken met deze strings (via de methode `split()`) en te controleren of `zet` in die lijst voorkomt.

`'1 2 3 4 5 6 7 8 9'.split()` is precies hetzelfde als `['1', '2', '3', '4', '5', '6', '7', '8', '9']`, maar makkelijker te typen.

De expressie aan de *rechter*kant controleert of de zet die de speler deed, een vrij vakje betreft op het bord. Dit wordt gedaan door het aanroepen van de functie `isRuimteVrij()`. Onthoud dat `isRuimteVrij()` `True` zal retourneren als de zet die je doorgeeft, toegestaan is op het bord. De functie `isRuimteVrij()` verwacht een integer in de variabele `zet`, dus je moet de functie `int()` nog even gebruiken om de string in `move` om te zetten in een integer.

De operatoren `not` worden aan beide zijden toegevoegd zodat de voorwaarde `True` zal opleveren als aan een van deze vereisten niet wordt voldaan. Hierdoor zal de lus de speler steeds blijven vragen een geldige zet te doen.

Ten slotte retourneert regel 81 de integervorm van de zet die de speler invoerde. Je was hopelijk niet vergeten dat `input()` altijd een string oplevert dus de functie `int()` was nodig om van de string een integer te maken.

Evaluatie met kortsluiting

Misschien is je al opgevallen dat er een mogelijk probleem kan zijn met de functie `haalSpelerZet()`. Wat zou er gebeuren als de speler `'X'` intypte, of een andere string die geen cijfer is? De expressie `zet not in '1 2 3 4 5 6 7 8 9'.split()` aan de linkerkant van `or` zou `False` opleveren, zoals verwacht, en vervolgens zou Python de expressie aan de rechterkant van de operator `or` evalueren.

Maar als je `int('X')` aanroept, zou dit een fout opleveren! Python constateert een fout omdat de functie `int()` alleen maar kan werken op strings van cijfertekens, zoals `'9'` of `'0'`, niet op strings van letters, zoals `'X'`.

Als voorbeeld van dit soort fout kun je het volgende in de interactieve shell typen:

```
>>> int('42')
42
>>> int('X')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    int('X')
ValueError: invalid literal for int() with base 10: 'X'
```

Maar als je Boter-kaas-en-eieren speelt en `'X'` invoert als zet, krijg je deze fout helemaal niet te zien. Hoe kan dat? Dat komt omdat we de voorwaarde van de `while`-lus laten kortsluiten.

Kortsluiten betekent het volgende: omdat het deel links van de operator `or` (`zet not in '1 2 3 4 5 6 7 8 9'.split()`) `True` oplevert, weet de Python-interpreter dat de hele expressie `True` zal opleveren. Het maakt niets uit of de expressie rechts van `or` `True` of `False` oplevert, want één (of meer) waarden aan één kant van `or` hoeven maar `True` te zijn om de hele expressie True te laten zijn.

Kijk maar: De expressie `True or False` retourneert `True` en de expressie `True or True` retourneert ook `True`. Als de waarde aan de linkerkant `True` is, is wat er rechts staat, niet meer van belang:

`False and <<<iets>>>` retourneert altijd `False`

`True or <<<iets>>>` retourneert altijd `True`

Dus Python kijkt niet meer naar de rest van de expressie en doet geen moeite meer om het gedeelte `not isRuimteVrij(bord, int(zet))` te evalueren. Dit betekent dat de functies `int()` en `isRuimteVrij()` niet meer worden aangeroepen als `zet not in '1 2 3 4 5 6 7 8 9'.split()` `True` is.

Dat komt wel goed uit voor ons programma want als de rechterkant ook `True` zou zijn, is `zet` geen string in cijfervorm. En dan zou `int()` een fout veroorzaken. De enige keer dat `zet not in '1 2 3 4 5 6 7 8 9'.split()` `False` zou opleveren, is wanneer `zet` geen string met één cijfer zou zijn, maar met meerdere cijfers. En in dat geval zou de aanroep naar `int()` geen fout opleveren.

Hier is een voorbeeld van een evaluatie met kortsluiting

Hier is een kort programmaatje dat een goed voorbeeld laat zien van kortsluiting. Probeer het volgende in de interactieve shell te typen:

```
>>> def RetourneerTrue():
       print('RetourneerTrue() is aangeroepen.')
       return True

>>> def RetourneerFalse():
       print('RetourneerFalse() is aangeroepen.')
       return False

>>> RetourneerTrue()
RetourneerTrue() is aangeroepen.
True
>>> RetourneerFalse()
RetourneerFalse() is aangeroepen.
False
```

Deze twee functies geven op het scherm weer dat ze zijn aangeroepen en ze retourneren een Boole-waarde. De functieaanroep `print()` in de functie laat je weten of Python de functie daadwerkelijk heeft aangeroepen of niet. Als `RetourneerTrue()` wordt aangeroepen, toont IDLE `'RetourneerTrue() is aangeroepen.'` en geeft ook de retourwaarde van `RetourneerTrue()` weer. Hetzelfde bij `RetourneerFalse()`.

Probeer nu het volgende te typen.

```
>>> RetourneerFalse() or RetourneerTrue()
RetourneerFalse() is aangeroepen.
RetourneerTrue() is aangeroepen.
True
>>> RetourneerTrue() or RetourneerFalse()
RetourneerTrue() is aangeroepen.
True
```

Het eerste stuk snappen we: De expressie `RetourneerFalse() or RetourneerTrue()` roept beide functies aan, dus je ziet beide berichten weergegeven.

Maar de tweede expressie toont alleen `'RetourneerTrue() is aangeroepen.'` maar niet `'RetourneerFalse() is aangeroepen.'`. En dat komt omdat Python `RetourneerFalse()` helemaal niet heeft aangeroepen. Omdat de linkerkant van de operator `or` al `True` opleverde, maakt het niet meer uit `RetourneerFalse()` oplevert en wordt dit niet meer aangeroepen.

Hetzelfde is van toepassing op de operator `and`. Probeer het volgende in de interactieve shell te typen:

```
>>> RetourneerTrue() and RetourneerTrue()
RetourneerTrue() is aangeroepen.
RetourneerTrue() is aangeroepen.
```

```
True
>>> RetourneerFalse() and RetourneerFalse()
RetourneerFalse() is aangeroepen.
False
```

Als de linkerkant van de operator `and` `False` oplevert, is de hele expressie `False`. Het maakt niet uit of de rechterkant van de operator `and` `True` of `False` is, dus Python kijkt er niet verder naar. Zowel `False and True` als `False and False` leveren `False` op, dus Python sluit de evaluatie kort.

Een zet kiezen uit een lijst met zetten

```
 83. def kiesWillekeurigeZetUitLijst(bord, zettenLijst):
 84.     # Retourneert een geldige zet uit de lijst die is doorgegeven op het
bord dat is doorgegeven.
 85.     # Retourneert None als er geen geldige zet is.
 86.     mogelijkeZetten = []
 87.     for i in zettenLijst:
 88.         if isRuimteVrij(bord, i):
 89.             mogelijkeZetten.append(i)
```

De functie `kiesWillekeurigeZetUitLijst()` is handig voor de KI-code. De parameter `bord` is een lijst met strings die het Boter-kaas-en-eieren-bord voorstelt. De tweede parameter `zettenLijst` is een lijst met integers van mogelijke vakjes om uit te kiezen. Als `zettenLijst` bijvoorbeeld `[1, 3, 7, 9]` is, betekent dat dat `kiesWillekeurigeZetUitLijst()` de integer moet retourneren voor een van de hoekvakjes.

Maar `kiesWillekeurigeZetUitLijst()` moet eerst controleren of het vakje wel een geldige keuze zou zijn voor de zet. De lijst `mogelijkeZetten` begint als een lege lijst. De `for`-lus itereert over de zetten die staan in `zettenLijst`. De zetten waarmee `isRuimteVrij()` `True` oplevert, worden toegevoegd aan `mogelijkeZetten` met de methode `append()` (de methode waarmee je iets achteraan in een lijst toevoegt).

```
 91.     if len(mogelijkeZetten) != 0:
 92.         return random.choice(mogelijkeZetten)
 93.     else:
 94.         return None
```

Op dit punt bevat de lijst `mogelijkeZetten` alle zetten die in `zettenLijst` stonden en ook vrije vakjes zijn. Als de lijst niet leeg is, staat er ten minste één mogelijke zet in die op het bord kan worden gedaan.

Maar de lijst zou ook leeg kunnen zijn. Als `zettenLijst` bijvoorbeeld `[1, 3, 7, 9]` was, maar op het bord dat wordt voorgesteld door de parameter `bord` alle hoekvakjes al bezet waren, zou de lijst `mogelijkeZetten` leeg zijn. In dat geval retourneert de functie `len(mogelijkeZetten)` de waarde `0` en retourneert de functie de waarde `None`. De volgende paragraaf vertelt meer over de waarde `None`.

De waarde `None` (Geen)

De waarde **None** stelt het ontbreken van een waarde voor. `None` is de enige waarde die het datatype `NoneType` kan hebben. Net zoals het datatype Boolean maar twee waarden kan hebben, heeft het datatype `NoneType` er maar één: `None`. Het kan handig zijn om de waarde `None` te gebruiken als je iets nodig hebt wat zoiets als 'bestaat niet', 'komt niet voor' of 'geen van alle' betekent.

Stel, bijvoorbeeld, dat je een variabele had met de naam `quizAntwoord` waarin het antwoord van de gebruiker op een quizvraag stond. De variabele zou `Waar` of `Niet waar` kunnen bevatten als antwoord van de gebruiker. Maar als de gebruiker de vraag nou had overgeslagen en helemaal geen antwoord had gegeven? Dan zou je `quizAntwoord` de waarde `None` kunnen geven. Dat zou een betere optie zijn dan iets anders, omdat dat de indruk zou wekken dat de gebruiker de vraag had beantwoord terwijl dat niet zo was.

Functies die iets retourneren doordat ze aan het einde van de functie zijn gekomen (en niet doordat de retourwaarde is afgegeven via een `return`-statement), leveren de waarde `None` als retourwaarde. De waarde `None` wordt geschreven zonder aanhalingstekens en met een hoofdletter N. De overige letters zijn kleine letters.

Als terzijde: `None` wordt niet weergegeven als retourwaarde in de interactieve shell, zoals dat met andere waarden wel gebeurt:

```
>>> 2 + 2
4
>>> 'Dit is een stringwaarde.'
>>> 'Dit is een stringwaarde.'
>>> None
>>>
```

Functies die niets lijken te retourneren, retourneren eigenlijk de waarde `None`. De functie `print()`, bijvoorbeeld, retourneert ook `None`. Kijk maar:

```
>>> spam = print('Hallo wereld!')
Hallo wereld!
>>> spam == None
True
```

De kunstmatige intelligentie voor de computer maken

```
 96. def haalComputerZet(bord, computerLetter):
 97.     # Gegeven een bord en de letter van de computer, bepaalt deze functie
welke zet de computer gaat doen en wordt die zet geretourneerd.
 98.     if computerLetter == 'X':
 99.         spelerLetter = 'O'
100.     else:
101.         spelerLetter = 'X'
```

De functie `haalComputerZet()` bevat de code voor de kunstmatige intelligentie (KI). De argumenten zijn een Boter-kaas-en-eieren-bord (in de parameter `bord`) en de letter van de computer (`'X'` of `'O'` in de parameter `computerLetter`). De eerste paar regels wijzen de andere letter toe aan een variabele met de naam `spelerLetter`. Op deze manier kun je dezelfde code gebruiken, of de computer nu X is of O.

De functie retourneert een integer van `1` tot `9`, die het vakje voorstelt waar de computer heen wil gaan.

Even herhalen hoe het Boter-kaas-en-eieren algoritme voor KI ook al weer werkt:

- Kijk eerst of er een zet is die KI kan doen om het spel te winnen. Als dat zo is, doe die zet. Zo niet, ga naar de tweede stap.
- Stap 2: kijk of er een zet is die de speler kan doen waarmee de computer het spel zou verliezen. Als dat zo, ga naar dat vakje om de speler te blokkeren. Zo niet, ga naar de derde stap.
- Stap 3: kijk of een van de hoekvakken vrij is (vakjes 1, 3, 7 of 9). Als er geen hoekvak vrij is, ga naar stap 4.
- Stap 4: controleer of het middenvak vrij is. Als dat zo is, ga daarheen. Als dat niet zo is, ga naar stap 5.
- Stap 5: ga naar een van de zijvakjes (vakjes 2, 4, 6 of 8). Er zijn geen stappen meer, want als de uitvoering bij deze stap is gekomen, waren de zijvakjes de enige vakjes die nog over waren.

De computer kijkt of hij in één zet kan winnen

```
103.     # Hier is ons algoritme voor onze KI in Boter-kaas-en-eieren:
104.     # Eerst kijken we of KI met de volgende zet kan winnen
105.     for i in range(1, 10):
106.         kopie = haalKopieBord(bord)
```

```
107.         if isRuimteVrij(kopie, i):
108.             doeZet(kopie, computerLetter, i)
109.             if isWinnaar(kopie, computerLetter):
110.                 return i
```

Als de computer bij de volgende zet zou kunnen winnen, moet hij die zet natuurlijk doen. De `for`-lus die op regel 105 begint, itereert langs elke mogelijke zet tussen 1 en 9. De code in die lus simuleert wat er zou gebeuren als de computer die zet zou doen.

Daarvoor hebben we die kopie van het bord nodig, om het even uit te proberen. De eerste regel in de lus (regel 106) maakt een kopie van de lijst `bord`. We willen natuurlijk niet dat de gesimuleerde zet het echte Boter-kaas-en-eieren-bord in de variabele `bord` meteen verandert. De functie `haalBordKopie()` retourneert een identieke, maar aparte lijst met de bordgegevens.

Regel 107 controleert of het vakje vrij is en zo ja, wordt de zet gesimuleerd op de kopie van het bord (nog niet op het echte bord). Als deze zet zou leiden tot de overwinning van de computer, is de retourwaarde van de functie de integer die hoort bij die zet.

Als geen van de vakjes leidt tot de overwinning, eindigt de lus en gaat de verwerking van het programma verder naar regel 112.

De computer controleert of de speler in één zet kan winnen

```
112.     # Controleert of de speler bij zijn volgende zet kan winnen, en
blokkeert die zet.
113.     for i in range(1, 10):
114.         kopie = haalKopieBord(bord)
115.         if isRuimteVrij(kopie, i):
116.             doeZet(kopie, spelerLetter, i)
117.             if isWinnaar(kopie, spelerLetter):
118.                 return i
```

Vervolgens simuleert de code de menselijke speler die naar elk van de vakjes gaat. De code lijkt op de vorige lus, behalve dat de letter van de speler op de kopie van het bord wordt geplaatst. Als de functie `isWinnaar()` aantoont dat de speler met deze zet zou winnen, kiest de computer dat vakje gauw, om de speler te kunnen blokkeren.

Als de menselijke speler niet kan winnen in een volgende zet, komt de `for`-lus aan het einde en gaat de verwerking verder op regel 120.

Controleren van de hoekvakjes, het middenvak en de zijvakjes (in die volgorde)

```
120.     # Probeert een van de hoeken te bezetten, als ze leeg zijn.
121.     zet = kiesWillekeurigeZetUitLijst(bord, [1, 3, 7, 9])
122.     if zet != None:
123.         return zet
```

De aanroep naar `kiesWillekeurigeZetUitLijst()` met de lijst `[1, 3, 7, 9]` zorgt ervoor dat de integer van een van de hoekvakken wordt geretourneerd: `1`, `3`, `7` of `9`. Als alle hoekvakken bezet zijn, retourneert de functie `kiesWillekeurigeZetUitLijst()` die handige waarde `None` en gaat de verwerking verder op regel 125.

```
125.     # Probeert het middelste vak te bezetten als het nog leeg is.
126.     if isRuimteVrij(bord, 5):
127.         return 5
```

Als geen van de hoekvakken beschikbaar zijn, proberen we naar het middenvak te gaan, als dat vrij is. Als het middenvak ook niet vrij is, gaat de verwerking naar regel 129.

```
129.     # Zet op een van de zijkanten.
130.     return kiesWillekeurigeZetUitLijst(bord, [2, 4, 6, 8])
```

Deze code doet ook een aanroep naar `kiesWillekeurigeZetUitLijst()`, maar geeft een lijst mee van de zijvakken (`[2, 4, 6, 8]`). De functie zal niet de waarde `None` opleveren, omdat de zijvakken de enige vakken zijn die nog over kunnen zijn. Hiermee eindigt de functie `haalComputerZet()` en het algoritme voor onze kunstmatige intelligentie.

Controleren of het bord vol is

```
132. def isBordVol(bord):
133.     # Retourneert True als elk vakje op het bord is bezet. Zo niet,
retourneert de functie False.
134.     for i in range(1, 10):
135.         if isRuimteVrij(bord, i):
136.             return False
137.     return True
```

De laatste functie is `isBordVol()`. Deze functie retourneert `True` als het meegegeven argument met de lijst van 10 strings die het bord voorstelt, een `'X'` of `'O'` bevat op elke positie (behalve op positie `0`, weet je nog, want die negeerden we in dit programma). Als er ten minste één vakje in `bord` voorkomt met als waarde een enkele spatie `' '` retourneert de functie `False`.

De `for`-lus laat ons de posities 1 tot en met 9 controleren in de lijst `bord`. Zodra een lege ruimte op het bord wordt aangetroffen (dat wil zeggen, zodra `isRuimteVrij(bord, i)` `True` retourneert), retourneert de functie `isBordVol()` de waarde `False`.

Als de verwerking door elke iteratie van de lus kan gaan, zijn er geen vakjes meer vrij. Regel 137 voert `return True` uit.

Het begin van het programma

```
140. print('Welkom bij Boter-kaas-en-eieren!')
```

Regel 140 is de eerste regel die zich niet binnen een functiedefinitie bevindt, dus het is de eerste regel code die wordt uitgevoerd wanneer je dit programma start. De speler wordt begroet.

```
142. while True:
143.     # Wist het bord
144.     hetBord = [' '] * 10
```

De `while`-lus op regel 142 heeft als voorwaarde gewoon `True` dus blijft doorgaan totdat de verwerking bij de `break`-statement komt. Regel 144 plaatst het hele Boter-kaas-en-eieren-bord in een variabele met de naam `hetBord`. Dit is een lijst met 10 strings, waarin elke string een lege spatie `' '` is.

We zouden deze hele lijst kunnen typen, maar het kan sneller met 'lijstreplicatie'. Dit gebeurt op regel 144. Het gaat sneller om `[' '] * 10` te typen dan `[' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ']`.

Bepalen welke letter de speler krijgt en wie het eerst mag

```
145.     spelerLetter, computerLetter = kiesLetter()
```

De functie `kiesLetter()` laat de speler kiezen of ze X of O willen zijn. De functie retourneert een lijst met twee strings, ofwel `['X', 'O']` of `['O', 'X']`. Deze meervoudige toewijzingstruc wijst `spelerLetter` toe aan het eerste item in de lijst en `computerLetter` aan het tweede item.

```
146.     beurt = wieMagEerst()
147.     print('De ' + beurt + ' mag eerst.')
148.     spelBezig = True
```

De functie `wieMagEerst()` bepaalt willekeurig wie er eerst mag en retourneert ofwel de string `'speler'` of de string `'computer'` en regel 147 vertelt de speler wie er eerst mag De variabele `spelBezig` houdt bij of het spel nog bezig is, of iemand al heeft gewonnen of dat het gelijkspel is.

De beurt van de speler uitvoeren

```
150.     while spelBezig:
```

De lus op regel 150 gaat heen en weer tussen de code voor de beurt van de speler en de beurt van de computer, zo lang `spelBezig` de waarde `True` heeft.

```
151.         if beurt == 'speler':
152.             # De speler is aan de beurt.
153.             tekenBord(hetBord)
154.             zet = haalSpelerZet(hetBord)
155.             doeZet(hetBord, spelerLetter, zet)
```

De variabele `beurt` werd oorspronkelijk ingesteld bij de aanroep van `wieMagEerst()` op regel 146. De variabele is ingesteld op `'speler'` of `'computer'`. Als `beurt` de string `'computer'` bevat, is de voorwaarde op regel 151 `False` en springt de verwerking naar regel 169.

Regel 153 roept `tekenBord()` aan en geeft de variabele `hetBord` mee om het Boter-kaas-en-eieren-bord op het scherm te tonen. Vervolgens laat de functie `haalSpelerZet()` de speler zijn zet typen (en checkt ook dat het wel een geldige zet is). De functie `doeZet()` voegt de X of O van de speler toe aan `hetBord`.

```
157.             if isWinnaar(hetBord, spelerLetter):
158.                 tekenBord(hetBord)
159.                 print('Hoera! Jij hebt gewonnen!')
160.                 spelBezig = False
```

Nu de speler zijn zet heeft gedaan, moet de computer controleren of ze met deze zet misschien hebben gewonnen. Als de functie `isWinnaar()` `True` retourneert, geeft de code van het if-blok het winnende bord weer en wordt een bericht getoond dat de speler heeft gewonnen.

De variabele `spelBezig` wordt ook op `False` gezet, zodat de verwerking niet meer verdergaat naar de beurt van de computer.

```
161.             else:
162.                 if isBordVol(hetBord):
163.                     tekenBord(hetBord)
164.                     print('Het is gelijkspel!')
```

```
165.                     break
```

Als de speler met zijn laatste zet niet won, zou het kunnen zijn dat het bord nu helemaal vol is en het gelijkspel is. In dit else-blok retourneert de functie `isBordVol()` de waarde `True` als er geen zetten meer mogelijk zijn. In dat geval geeft het if-blok dat begint op regel 163 het bord met gelijkspel weer en wordt dat aan de speler meegedeeld. De verwerking breekt uit de `while`-lus en springt naar regel 186.

```
166.                 else:
167.                     beurt = 'computer'
```

Als de speler niet heeft gewonnen of gelijkgespeeld, wordt de variabele `beurt` op regel 167 op `'computer'` gezet zodat de code voor de beurt van de computer wordt uitgevoerd tijdens de volgende iteratie.

De beurt van de computer uitvoeren

Als de variabele `beurt` niet `'speler'` is in de voorwaarde op regel 151, moet de computer aan de beurt zijn. De code in dit else-blok lijkt op de code voor de beurt van de speler.

```
169.         else:
170.             # Beurt van de computer.
171.             zet = haalComputerZet(hetBord, computerLetter)
172.             doeZet(hetBord, computerLetter, zet)
174.             if isWinnaar(hetBord, computerLetter):
175.                 tekenBord(hetBord)
176.                 print('De computer heeft gewonnen! Volgende keer
beter...')
177.                 spelBezig = False
178.             else:
179.                 if isBordVol(hetBord):
180.                     tekenBord(hetBord)
181.                     print('Het is gelijkspel!')
182.                     break
183.                 else:
184.                     beurt = 'speler'
```

Regels 170 t/m 184 zijn bijna gelijk aan de code voor de beurt van de speler op de regels 152 t/m 167. Het enige verschil is dat deze code de letter van de computer gebruikt en `haalComputerZet()` aanroept.

Als het spel niet wordt gewonnen of er geen gelijkspel is, stelt regel 184 `beurt` in op de beurt van de speler. Er zijn geen regels code meer in de `while`-lus, dus de verwerking springt terug naar de `while`-statement op regel 150.

```
186.     if not nogmaalsSpelen():
187.         break
```

Deze regels code komen direct na het while-blok dat begon met de `while`-statement op regel 150. Vergeet niet dat de verwerking alleen die `while`-lus zou verlaten als de voorwaarde (de variabele `spelBezig`) `False` zou zijn. `spelBezig` wordt op `False` gezet wanneer het spel is geëindigd, dus op dit punt vraagt het spel of de speler nog een keer wil spelen.

Als `nogmaalsSpelen()` `False` retourneert, is de voorwaarde van de `if`-statement `True` (omdat de operator `not` de Boole-waarde omdraait) en wordt de `break`-statement uitgevoerd. En daarmee breekt de verwerking uit de `while`-lus die begon op regel 142. Maar aangezien er geen code meer is na dat while-blok, is het programma aan het einde gekomen.

Samenvatting

Een programma schrijven voor een spel waarin de computer kan meedoen als speler, is een kwestie van zorgvuldig alle mogelijke situaties overdenken waarin de computer zich kan bevinden en aangeven hoe de computer in die situaties moet handelen. Dit is een vorm van het ontwikkelen van kunstmatige intelligentie (KI). De KI van Boter-kaas-en-eieren is niet al te moeilijk omdat er niet veel mogelijke zetten zijn in Boter-kaas-en-eieren. De KI voor een spel als schaken of dammen zou natuurlijk veel ingewikkelder zijn.

Onze KI checkt of bij een mogelijke zet de uitkomst 'winnen' zou zijn. Als dat niet zo is, wordt gekeken of de andere speler met een volgende zet zou kunnen winnen. Zo ja, wordt de speler geblokkeerd. Vervolgens kiest de KI gewoon een beschikbaar hoekvakje, een middenvakje of een zijvakje, in die volgorde. Dit is het simpele algoritme dat de computer volgt.

En om de computer te kunnen laten nadenken, maken we kopieën van de bordgegevens zodat we zetten kunnen uitproberen. Op die manier kan KI inschatten of een zet kan resulteren in de overwinning of in verlies. En vervolgens doet KI die zet op het echte bord. Dit soort simulatie is een effectieve manier om te kunnen voorspellen of een mogelijke zet een goede zet zal zijn of niet.

Hoofdstuk 11

CODE KRAKEN

In dit hoofdstuk behandelen we:
- 'Hard-coding'
- De toewijzingsoperators `+=`, `-=`, `*=`, `/=`
- De functie `random.shuffle()`
- De methode `sort()` voor lijsten
- De methode `join()` voor lijsten
- Stringinterpolatie (ook wel stringopmaak genoemd)
- De conversiespecificatie `%s`
- Geneste lussen

In dit hoofdstuk leer je een paar nieuwe methoden en functies die in Python zijn ingebouwd. Ook leer je over een paar nieuwe toewijzingsoperators en over stringinterpolatie. Wat je hiermee kunt doen, kon je eerder ook al wel doen, maar dit zijn gemakkelijke kortere manieren om te coderen.

Code kraken is een spelletje in logisch redeneren dat je met een vriend of vriendin kunt spelen. Je vriend(in) neemt een willekeurig getal van 3 cijfers in gedachten en jij moet het raden. De cijfers moeten allemaal anders zijn. Na elke keer dat je raadt, geeft de ander je een aanwijzing. Dit kunnen 3 soorten zijn:

- **Niks** – Geen van de drie cijfers die je hebt geraden, komt voor in het getal.
- **Wit** – Eén van de cijfers komt voor in het geheime getal maar de positie die je noemde, is fout.
- **Zwart** – Je hebt het juiste cijfer op de juiste plaats geraden.

Het is wel zo dat als je de aanwijzing 'Wit' of 'Zwart' krijgt, je nog niet weet welk cijfer in het getal dat je noemde, goed is. Je krijgt de aanwijzingen in alfabetische volgorde.

Je kunt meerdere aanwijzingen krijgen na elke poging. Als het geheime getal 456 is en jij zegt 465, is de aanwijzing 'Wit Wit Zwart'. De 4 levert 'Zwart' en de 6 en 5 leveren 'Wit Wit' op.

Voorbeelduitvoer van Code kraken

```
Ik denk aan een getal met 3 cijfers. Probeer het te raden.
Hier wat aanwijzingen:
Wanneer ik zeg:    Betekent dat:
  Wit           Eén cijfer is goed, maar staat op de verkeerde plek
  Zwart        Eén cijfer is goed en staat op de juiste plek.
  Niks           Alle cijfers zijn fout.
Ik neem een getal in gedachten. Je mag 10 keer raden.
Poging 1:
123
Zwart
Poging 2:
453
Wit
Poging 3:
425
Zwart
Poging 4:
326
Niks
Poging 5:
489
Niks
Poging 6:
075
Zwart Zwart
Poging 7:
015
Wit Zwart
Poging 8:
175
Goed geraden!
Wil je nog een keer spelen? (ja of nee)
nee
```

Broncode van Code kraken

Als je een foutmelding krijgt nadat je deze code hebt ingetypt, vergelijk jouw code dan met de code van het boek met behulp van de online diff tool op http://invpy.com/diff/kraken.

kraken.py

```
 1. import random
 2. def haalGeheimGetal(aantalCijfers):
 3.     # Retourneert een string die aantalCijfers lang is en bestaat uit unieke
willekeurige cijfers.
 4.     getallen = list(range(10))
 5.     random.shuffle(getallen)
 6.     geheimGetal = ''
 7.     for i in range(aantalCijfers):
 8.         geheimGetal+= str(getallen[i])
 9.     return geheimGetal
10.
11. def haalHulp(poging, geheimGetal):
```

```
12.     # Retourneert een string met de aanwijzingen niks, wit of zwart aan de
gebruiker.
13.     if poging == geheimGetal:
14.         return 'Goed geraden!'
15.
16.     hulp = []
17.
18.     for i in range(len(poging)):
19.         if poging[i] == geheimGetal[i]:
20.             hulp.append('Zwart')
21.         elif poging[i] in geheimGetal:
22.             hulp.append('Wit')
23.     if len(hulp) == 0:
24.         return 'Niks'
25.
26.     hulp.sort()
27.     return ' '.join(hulp)
28.
29. def isAlleenCijfers(getal):
30.     # Retourneert True als getal een string is die alleen uit cijfers bestaat.
Anders retourneert dit False.
31.     if getal == '':
32.         return False
33.
34.     for i in getal:
35.         if i not in '0 1 2 3 4 5 6 7 8 9'.split():
36.             return False
37.
38.     return True
39.
40. def nogmaalsSpelen():
41.     # Deze functie retourneert True als de speler nog een keer wil spelen,
anders is het False.
42.     print('Wil je nog een keer spelen? (ja of nee)')
43.     return input().lower().startswith('j')
44.
45. AANTALCIJFERS = 3
46. MAXPOGING = 10
47.
48. print('Ik denk aan een getal met %s cijfers. Probeer het te raden.' %
(AANTALCIJFERS))
49. print('Hier wat aanwijzingen:')
50. print('Wanneer ik zeg:    Betekent dat:')
51. print('Wit                   Eén cijfer is goed, maar staat op de verkeerde
plek.')
52. print('Zwart               Eén cijfer is goed en staat op de juiste plek.')
53. print('Niks                  Alle cijfers zijn fout.')
54.
55. while True:
56.     geheimGetal = haalGeheimGetal(AANTALCIJFERS)
57.     print('Ik neem een getal in gedachten. Je mag %s keer raden.' %
(MAXPOGING))
58.
59.     aantalPogingen = 1
60.     while aantalPogingen <= MAXPOGING:
61.         poging = ''
62.         while len(poging) != AANTALCIJFERS or not isAlleenCijfers(poging):
```

```
63.             print('Poging %s: ' % (aantalPogingen ))
64.             poging = input()
65.
66.         hulp = haalHulp(poging, geheimGetal)
67.         print(hulp)
68.         aantalPogingen += 1
69.
70.         if poging == geheimGetal:
71.             break
72.         if aantalPogingen > MAXPOGING:
73.             print('Je mag niet meer raden. De code was %s.' % (geheimGetal))
74.
75.     if not nogmaalsSpelen():
76.         break
```

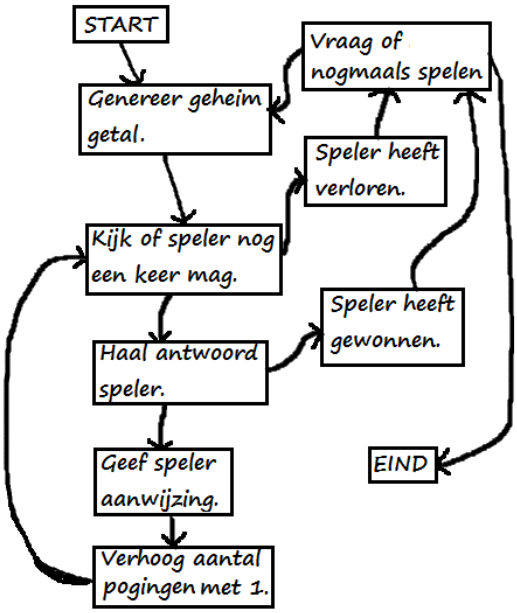
Het programma ontwerpen

Het stroomdiagram in Afbeelding 11-1 laat zien wat er gebeurt in dit spel en in welke volgorde.

Hoe de code werkt

```
 1. import random
 2. def haalGeheimGetal(aantalCijfers):
 3.     # Retourneert een string die aantalCijfers lang is en bestaat uit unieke
willekeurige cijfers.
```

Bij het begin van het programma importeren we de module `random`. Vervolgens definiëren we een functie genaamd `haalGeheimGetal()`. De functie verzint een geheim getal met drie unieke cijfers. In plaats van een getal met slechts 3 cijfers, kun je met behulp van de parameter `aantalCijfers` ook geheime getallen verzinnen met een ander aantal cijfers. Je kunt bijvoorbeeld een geheim getal maken met vier of zes cijfers. Dan hoef je alleen maar `4` of `6` door te geven als `aantalCijfers`.



Afbeelding 11-1: Stroomdiagram voor het spel Code kraken.

Een unieke set cijfers 'schudden'

```
4.      getallen = list(range(10))
5.      random.shuffle(getallen)
```

Op regel 4 staat `list(range(10))`. Dit levert altijd `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`. `list(range(10))` zeggen is gewoon makkelijker dan al die cijfers typen. De variabele `getallen` bevat een lijst met al deze tien cijfers.

De functie `random.shuffle()`

De functie `random.shuffle()` wijzigt de volgorde van de items in een lijst in een willekeurige andere volgorde. De functie retourneert geen waarde, maar verandert hier gewoon de lijst die je aan de functie doorgeeft. Dit lijkt op de manier waarop de functie `doeZet()` in het hoofdstuk over Boter-kaas-en-eieren de lijst wijzigde die aan de functie werd meegegeven, in plaats van een nieuwe lijst te maken die de wijziging bevatte. Dit is de reden waarom we hier **niet** code schrijven zoals `getallen = random.shuffle(getallen)`, zoals je misschien zou verwachten.

Probeer de functie `random.shuffle()` eens uit door de volgende code in de interactieve shell te typen. De cijfers die jij krijgt, zullen verschillen van de cijfers die ik kreeg, maar je snapt het principe van willekeurigheid!

```
>>> import random
>>> ham = list(range(10))
>>> print(ham)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> random.shuffle(ham)
>>> print(ham)
[3, 0, 5, 9, 6, 8, 2, 4, 1, 7]

>>> random.shuffle(ham)
>>> print(ham)
[1, 2, 5, 9, 4, 7, 0, 3, 6, 8]

>>> random.shuffle(ham)
>>> print(ham)
[9, 8, 3, 5, 4, 7, 1, 2, 0, 6]
```

Je wilt dat het geheime getal in dit programma unieke cijfers heeft. Dit spel is leuker als je niet dezelfde cijfers kunt krijgen in het geheime getal, zoals `'244'` of `'333'`. Met de functie `shuffle()` krijg je dit voor elkaar.

Het geheime nummer halen uit de 'geschudde' cijfers

```
6.      geheimGetal = ''
7.      for i in range(aantalCijfers):
8.          geheimGetal += str(getallen[i])
9.      return geheimGetal
```

Het geheime getal is een string met de eerste `aantalCijfers` cijfers (3 in ons geval) van de geschudde lijst met integers. Als de geschudde lijst in `getallen` bijvoorbeeld `[9, 8, 3, 5, 4, 7, 1, 2, 0,`

6] is en aantalGetallen is 3, moet de string die door haalgeheimGetal() wordt geretourneerd, dus '983' zijn.

Om dit voor elkaar te krijgen, begint de variabele geheimGetal als een lege string. De for-lus op regel 7 itereert het aantal keren dat in aantalCijfers zit. Bij elke iteratie door de lus wordt de integer die op indexpositie i staat, uit de geschudde lijst gehaald, omgezet in een string en achteraan geheimGetal geplakt.

Als getallen bijvoorbeeld verwijst naar de lijst [9, 8, 3, 5, 4, 7, 1, 2, 0, 6], wordt bij de eerste iteratie getallen[0] (dat is dus 9) doorgegeven aan str(). De functie str() maakt van de integer 9 de string '9' en die wordt achteraan gezet in geheimGetal. Bij de tweede iteratie gebeurt hetzelfde. met getallen[1] (dat is dus 8) en bij de derde iteratie gebeurt hetzelfde met getallen[2] (dat is dus 3). De uiteindelijke waarde van geheimGetal die wordt geretourneerd, is '983'.

geheimGetal in de functie bevat dus een string, geen integer! Dit lijkt misschien gek, maar komt omdat je integers niet kunt concateneren (aan elkaar kunt plakken). De expressie 9 + 8 + 3 levert 20 op, maar wat je wilt, is '9' + '8' + '3', dat '983' oplevert.

Nog meer toewijzingsoperators

De operator += op regel 8 is een nieuwe toewijzingsoperator. Normaliter, als je een waarde aan een variabele wilt toevoegen of vastplakken, gebruik je dit soort code:

```
ham = 42
ham = ham + 10

kaas = 'Hallo '
kaas = kaas + 'wereld!'
```

De toewijzingsoperator += biedt een kortere manier om dit te doen. Je hoeft de variabelenaam niet tweemaal te typen. De volgende code doet hetzelfde als bovenstaande code:

```
ham = 42
ham += 10       # is gelijk aan ham = ham + 10

kaas = 'Hallo '
kaas += 'wereld!' # is gelijk aan kaas = kaas + 'wereld!'
```

Er zijn ook nog andere verkorte toewijzingsoperators. Probeer het volgende in de interactieve shell te typen:

```
>>> ham = 42
>>> ham -= 2
>>> ham
40
>>> ham *= 3
>>> ham
120
>>> ham /= 10
>>> ham
12,0
```

Bepalen welke hulp je moet geven

```
11. def haalHulp(poging, geheimGetal):
12.     # Retourneert een string met de aanwijzingen niks, wit of zwart aan de
gebruiker.
13.     if poging == geheimGetal:
14.         return 'Goed geraden!'
```

De functie `haalHulp()` retourneert een string met de aanwijzingen 'Wit', 'Zwart' of 'Niks', afhankelijk van de inhoud van de parameters `poging` en `geheimGetal`. Wat natuurlijk het meest voor de hand ligt, is te controleren of de poging gelijk is aan het geheime getal. Als dat zo is, retourneert regel 14 de tekenreeks `'Goed geraden!'`.

```
16.     hulp = []
17.
18.     for i in range(len(poging)):
19.         if poging[i] == geheimGetal[i]:
20.             hulp.append('Zwart')
21.         elif poging[i] in geheimGetal:
22.             hulp.append('Wit')
```

Als de poging niet gelijk is aan het geheime getal, moet de code uitvogelen welke aanwijzingen aan de speler moeten worden gegeven. De lijst in `hulp` begint leeg en de strings `'Zwart'` en `'Wit'` worden toegevoegd waar nodig.

Je doet dit door elke mogelijke index te doorlopen in `poging` en `geheimGetal`. (De strings in beide variabelen zijn even lang, dus de code kan zowel `len(poging)` als `len(geheimGetal)` zijn. Dat werkt hetzelfde.) Terwijl de waarde van `i` verandert van `0` in `1`, `2` enzovoort, controleert regel 19 of het eerste, tweede, derde teken van `poging` gelijk is aan het getal op dezelfde indexpositie van `geheimGetal`. Als dat zo is, voegt regel 20 de string `'Zwart'` toe aan de variabele `hulp`.

Anders controleert regel 21 of het getal op de positie `i` in `hulp` op een andere plek in `geheimGetal` voorkomt. Als dat zo is, weet je dat het cijfer ergens voorkomt in het geheime getal, alleen niet op dezelfde plek. Regel 22 voegt dan `'Wit'` toe aan `hulp`.

```
23.     if len(hulp) == 0:
24.         return 'Niks'
```

Als de lijst `hulp` leeg is na de lus, weet je dat er geen goede cijfers zaten in `poging`. Dan is `'Niks'` de enige aanwijzing die de speler krijgt.

De methode `sort()` voor lijsten

```
26.     hulp.sort()
```

Lijsten hebben een methode ter beschikking genaamd `sort()`, waarmee de items in de lijst worden geordend in alfabetische of numerieke volgorde. Probeer het volgende in de interactieve shell te typen:

```
>>> ham = ['kat', 'aardvarken', 'vleermuis', 'miereneter']
>>> ham.sort()
>>> ham
```

```
['aardvarken', 'kat', 'miereneter', 'vleermuis']

>>> ham = [9, 8, 3, 5, 4, 7, 1, 2, 0, 6]
>>> ham.sort()
>>> ham
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

De methode `sort()` *retourneert* niet zozeer een gesorteerde lijst, maar sorteert de lijst 'ter plaatse'. Dat is net zoals de methode `reverse()` werkt.

Je moet lijsten niet sorteren met zoiets als: `return ham.sort()` omdat je dan de waarde `None` zou krijgen (dat is namelijk de 'retourwaarde' van `sort()`). In plaats daarvan moet je een aparte regel `ham.sort()` maken en dan de regel `return ham`.

En waarom moet je lijst `hulp` überhaupt sorteren? Nou, om het de speler niet al te makkelijk te maken, natuurlijk! Als `hulp ['Wit', 'Zwart, 'Wit']` was, zou de speler direct weten dat het middelste cijfer op de juiste positie staat. Aangezien de andere twee aanwijzingen allebei 'Wit' zijn, zou de speler weten dat hij het eerste en derde cijfer alleen maar hoefde om te wisselen.

Als de aanwijzingen altijd op alfabetische volgorde worden gesorteerd, weet de speler niet op welk cijfer de aanwijzing 'Zwart' slaat. En dat willen we natuurlijk voor ons spel.

De methode `join()` voor strings

```
27.     return ' '.join(hulp)
```

De methode `join()` voor strings neemt een lijst met strings en retourneert één enkele string die bestaat uit alle strings in de lijst. De string waarop de methode wordt aangeroepen (op regel 27 is dit een enkele spatie, `' '`), verschijnt tussen elke string in de lijst. Typ bijvoorbeeld het volgende in de interactieve shell.

```
>>> ' '.join(['Mijn', 'naam', 'is', 'Sofie'])
'Mijn naam is Sofie'
>>> 'x'.join(['hallo', 'wereld'])
'halloxwereld'
```

De string die wordt geretourneerd op regel 27 is dus elke string in `hulp` met een enkele spatie tussen elke string. De methode `join()` voor strings kun je zien als het tegenovergestelde van de methode `split()` die wel al kennen. De methode `split()` neemt een string, splitst die op in verschillende items en maakt daar een lijst van en `join()` neemt een lijst met aparte strings en maakt daar één string van.

Controleren of een string alleen cijfers bevat

```
29. def isAlleenCijfers(getal):
30.     # Retourneert True als getal een string is die alleen uit cijfers bestaat.
Anders retourneert dit False.
31.     if getal == '':
32.         return False
```

De functie `isAlleenCijfers()` helpt vast te stellen of de speler wel een geldig antwoord geeft. Regel 31 controleert of `getal` een lege string is en retourneert `False` als dat zo is.

```
34.     for i in getal:
35.         if i not in '0 1 2 3 4 5 6 7 8 9'.split():
36.             return False
37.
38.     return True
```

De `for`-lus itereert over elk teken in de string `getal`. De waarde van `i` is bij elke iteratie steeds één teken. Binnen het for-blok controleert de code of `i` niet voorkomt in de lijst die wordt geretourneerd door `'0 1 2 3 4 5 6 7 8 9'.split()`. (De retourwaarde van `split()` is natuurlijk `['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']` maar '0 1 2 3 4 5 6 7 8 9'.split() is gemakkelijker om te typen.) Als `i` niet voorkomt, weet je dat er een niet-numeriek teken in `getal` staat. En de waarde `False` wordt dan geretourneerd.

Als de verwerking voorbij de `for`-lus is gekomen, weet je dat elk teken in `getal` een cijfer is. In dat geval wordt `True` geretourneerd.

Uitvinden of de speler nog een keer wil spelen

```
40. def nogmaalsSpelen():
41.     # Deze functie retourneert True als de speler nog een keer wil spelen,
anders is het False.
42.     print('Wil je nog een keer spelen? (ja of nee)')
43.     return input().lower().startswith('j')
```

De functie `nogmaalsSpelen()` is dezelfde functie die je al eerder hebt gebruikt, in Galgje en Boter-kaas-en-eieren. De lange expressie op regel 43 retourneert ofwel `True` of `False` afhankelijk van het antwoord van de speler.

Het begin van het programma

```
45. AANTALCIJFERS = 3
46. MAXPOGING = 10
47.
48. print('Ik denk aan een getal met %s cijfers. Probeer het te raden.' %
(AANTALCIJFERS))
49. print('Hier wat aanwijzingen:')
50. print('Wanneer ik zeg:    Betekent dat:')
51. print('Wit                    Eén cijfer is goed, maar staat op de verkeerde
plek.')
52. print('Zwart                Eén cijfer is goed en staat op de juiste plek.')
53. print('Niks                   Alle cijfers zijn fout.')
```

Dit is het feitelijke begin van het programma. In plaats van in je programma vast te leggen dat je geheime getal altijd uit 3 cijfers moet bestaan, kun je een constante maken met de naam `AANTALCIJFERS`. In plaats van in je programma vast te leggen dat een speler altijd 10 keer mag raden, kun je een constante maken met de naam `MAXPOGINGEN`.

Als je nu het aantal pogingen of het aantal cijfers in het geheime getal wilt wijzigen, kan dat heel gemakkelijk. Dan hoef je alleen maar regel 45 en/of 46 aan te passen en de rest van het programma werkt zonder dat je er iets aan hoeft te veranderen. Schrijf de naam van de variabele met hoofdletters, om aan te geven dat de variabele hier een constante is. Een programma wijzigt nooit de waarde die in een constante staat.

De functieaanroepen naar `print()` vertellen de speler wat de regels van het spel zijn en wat de aanwijzingen 'Wit', 'Zwart' en 'Niks' betekenen. De functieaanroep naar `print()` op regel 48 eindigt met `% (AANTALCIJFERS)` en bevat `%s` in de string. Dit is een techniek die stringinterpolatie heet.

Stringinterpolatie

Stringinterpolatie is een snellere manier om te coderen. Normaliter, als je de stringwaarden in variabelen in een andere string wilt gebruiken, gebruik je de samenvoegingsoperator +:

```
>>> naam = 'Els'
>>> evenement = 'feestje'
>>> waar = 'het zwembad'
>>> dag = 'zaterdag'
>>> tijd = '18:00 uur'

>>> print('Hoi ' + naam + ', Ga jij aanstaande ' + dag + ' om ' + tijd + ' naar het
' + evenement + ' in ' + waar  + '?')
Hoi Els, Ga jij aanstaande zaterdag om 18:00 uur naar het feestje in het zwembad?
```

Zoals je kunt zien, kan het best lastig zijn om een regel te typen waarin meerdere strings worden geconcateneerd. In plaats daarvan kun je gebruikmaken van **stringinterpolatie**. Daarbij kun je tijdelijke aanduidingen zoals `%s` gebruiken (deze aanduidingen heten **conversiespecificaties**), en vervolgens alle variabelenamen aan het eind neerzetten. Elke `%s` wordt vervangen door de waarde van de variabele aan het eind van de string. De volgende code doet bijvoorbeeld hetzelfde als de code hierboven:

```
>>> naam = 'Els'
>>> evenement = 'feestje'
>>> waar = 'het zwembad'
>>> dag = 'zaterdag'
>>> tijd = '18:00 uur'

>>> print('Hoi %s, Ga jij aanstaande %s om %s naar het %s in %s?' % (naam, dag,
tijd, evenement, waar))
Hoi Els, Ga jij aanstaande zaterdag om 18:00 uur naar het feestje in het zwembad?
```

Stringinterpolatie maakt je code gemakkelijker om te typen. De laatste regel bevat de aanroep `print()` met een string met conversiespecificaties, gevolgd door het teken %, gevolgd door een paar haakjes met daartussen de variabelen. De eerste variabelenaam wordt gebruikt voor de eerste `%s`, de tweede variabele voor de tweede `%s` enzovoort. Je moet net zoveel `%s`-conversiespecificaties hebben als er variabelen zijn.

Nog een voordeel van het gebruik van stringinterpolatie in plaats van stringconcatenatie is dat interpolatie met elk datatype werkt, niet alleen met strings. Alle waarden worden automatisch geconverteerd naar het datatype string. Als je een integer met een string zou concateneren, zou je deze fout krijgen:

```
>>> ham = 42
>>> print('ham == ' + ham)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

Met stringconcatenatie kun je alleen twee strings aan elkaar vastplakken, maar `ham` is een integer. Je zou er aan moeten denken om `str(ham)` te schrijven in plaats van `ham`. Met stringinterpolatie wordt deze stringconversie automatisch voor je gedaan. Probeer dit maar in de interactieve shell:

```
>>> ham = 42
>>> print('Ham is %s' % (ham))
Ham is 42
```

Stringinterpolatie wordt ook wel **stringopmaak** genoemd.

Het geheime getal maken

```
55. while True:
56.     geheimGetal = haalGeheimGetal(AANTALCIJFERS)
57.     print('Ik neem een getal in gedachten. Je mag %s keer raden.' %
(MAXPOGING))
58.
59.     aantalPogingen = 1
60.     while aantalPogingen <= MAXPOGINGEN:
```

Regel 55 is een oneindige `while`-lus die de voorwaarde `True` heeft, dus eeuwig door blijft gaan tot er een `break`-statement komt. Binnen deze oneindige lus krijg je een geheim getal van de functie `haalGeheimGetal()` (waarbij je `AANTALCIJFERS` doorgeeft om te zeggen uit hoeveel cijfers het geheime getal moet bestaan). Dit wordt toegewezen aan `geheimGetal`. Vergeet niet dat de waarde van `geheimGetal` een string is en geen integer.

Regel 57 vertelt de speler hoeveel cijfers er in het geheime getal staan met behulp van stringinterpolatie in plaats van stringconcatenatie. Zet de variabele `aantalPogingen` op `1` om dit te markeren als de eerste poging. Vervolgens bevat regel 60 een nieuwe `while`-lus die doorloopt zolang `aantalPogingen` kleiner is dan of gelijk is aan `MAXPOGINGEN`.

Het antwoord van de speler halen

Zoals je ziet, staat deze tweede `while`-lus op regel 60 binnen een andere `while`-lus die op regel 55 begon. Zo'n lus binnen een andere lus heet een **geneste lus**. Als er een `break`- of `continue`-statement staat, heeft dat alleen effect op de binnenste lus, niet op de buitenste lus.

```
61.         poging = ''
62.         while len(poging) != AANTALCIJFERS or not isAlleenCijfers(poging):
63.             print('Poging %s: ' % (aantalPogingen))
64.             poging = input()
```

De variabele `poging` bevat de string die de speler heeft getypt en die wordt geretourneerd door de functie `input()`. De code blijft door de lus gaan en de speler vragen iets te typen, totdat de speler een geldig antwoord heeft gegeven. Een geldig antwoord bevat alleen cijfers, en hetzelfde aantal cijfers als er in het geheime getal staat. Dat is wat de `while`-lus die op regel 62 begint, doet.

De variabele `poging` wordt ingesteld op de lege string op regel 61, dus de voorwaarde van de `while`-lus is `False`, de eerste keer dat de voorwaarde wordt getest. Dit moet om er zeker van te zijn dat de verwerking de lus ten minste eenmaal binnengaat.

De aanwijzingen halen voor de speler

```
66.         hulp = haalHulp(poging, geheimGetal)
67.         print(hulp)
68.         aantalPogingen += 1
```

Als de verwerking voorbij de `while`-lus is gekomen die begon op regel 62, bevat `poging` een geldig antwoord. Dit antwoord en `geheimGetal` geef je door aan de functie `haalHulp()`. De functie retourneert een string met de aanwijzingen, die op regel 67 aan de speler worden getoond. Regel 68 verhoogt `aantalPogingen` met behulp van de verkorte toewijzingsoperator voor optelling.

Controleren of de speler heeft gewonnen of verloren

```
70.         if poging == geheimGetal:
71.             break
72.         if aantalPogingen > MAXPOGINGEN:
73.             print('Je mag niet meer raden. De code was %s.' % (geheimGetal))
```

Als `poging` dezelfde waarde heeft als `geheimGetal`, heeft de speler het geheime getal geraden en regel 71 breekt dan uit de `while`-lus die begon op regel 60.

Zo niet, gaat de verwerking door naar regel 72 waar wordt gecontroleerd of de speler nog een poging over heeft. Als het aantal pogingen dat de speler heeft gedaan, groter is dan `MAXPOGINGEN`, laat het programma de speler weten dat ze hebben verloren.

Op dit punt springt de verwerking terug naar de `while`-lus op regel 60 waar de speler nog een keertje mag raden. Als de speler geen pogingen meer over heeft (of er uit de lus is gebroken met de `break`-statement op regel 71), gaat de verwerking voorbij de lus en verder naar regel 75.

De speler vragen of ze nog een keertje willen spelen

```
75.     if not nogmaalsSpelen():
76.         break
```

Regel 75 vraagt de speler of ze nogmaals willen spelen door de functie `nogmaalsSpelen()` aan te roepen. Als `nogmaalsSpelen()` `False` retourneert, breek je uit de `while`-lus die begon op regel 55. Aangezien er geen code meer is na deze lus, eindigt het programma.

Als `nogmaalsSpelen()` `True` retourneert, voert de verwerking de `break`-statement niet uit en springt de verwerking terug naar regel 55. Het programma genereert dan een nieuw geheim getal en de speler kan opnieuw spelen.

Samenvatting

Code kraken is niet moeilijk te programmeren, maar wel moeilijk om te winnen. Maar als je het een paar keer hebt gespeeld, ontdek je betere manieren om te raden en gebruik te maken van de aanwijzingen die je krijgt. Dit is een beetje hetzelfde als beter worden met programmeren als je het blijft doen.

In dit hoofdstuk maakte je kennis met een paar nieuwe functies en methoden (`random.shuffle()`, `sort()` en `join()`), en een paar handige trucs om minder te hoeven typen. Door de verkorte toewijzingsoperators te gebruiken, hoef je iets minder te typen wanneer je de waarde van een variabele ten opzichte van zichzelf wilt wijzigen (zoals in `ham = ham + 1`, dat kan worden verkort tot `ham +=`

1). Stringinterpolatie maakt je code gemakkelijker leesbaar, door een `%s` (een conversiespecificatie) binnen in de string te plaatsen in plaats van heel veel stringconcatenaties te doen.

De methode `join()` voor strings krijgt een lijst met strings doorgegeven die worden geconcateneerd, met de string waarop de methode wordt uitgevoerd als scheidingsteken tussen de strings. De methode `sort()` voor lijsten herordent de items in een lijst in alfabetische volgorde. De methode `append()` voor lijsten voegt een waarde toe aan het eind van de betreffende lijst.

Het volgende hoofdstuk gaat niet direct over programmeren, maar het bevat inhoud die belangrijk is voor de games die we in de verdere hoofdstukken in dit boek gaan maken. We gaan iets leren over de wiskundige begrippen Cartesische coördinaten en negatieve getallen. Deze begrippen worden gebruikt in de games Sonar, Reversi en Dodger, maar ook in heel veel andere games. Als je al iets weet over deze begrippen (van wiskunde op school misschien), lees het dan toch nog even door, om je kennis op te frissen.

Hoofdstuk 12

CARTESISCHE COÖRDINATEN

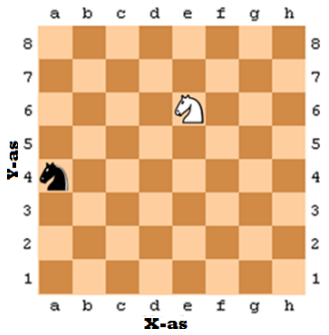
In dit hoofdstuk behandelen we:

- Het Cartesisch coördinatenstelsel
- De X-as en de Y-as
- De commutatieve eigenschap van optellen
- Absolute waarden en de functie `abs()`

In dit hoofdstuk schrijven we geen nieuwe game. In plaats daarvan gaan we het even hebben over een paar eenvoudige wiskundige begrippen die je gaat gebruiken in de rest van dit boek. In 2D-games kunnen de graphics op het scherm naar links en rechts bewegen en omhoog en omlaag. Deze twee richtingen bevinden zich in een tweedimensionale ruimte, 2D. Games met objecten die op een tweedimensionaal scherm rondbewegen, hebben een manier nodig om de plek op het scherm te vertalen in cijfers, zodat het programma ermee kan werken.

En daar is het Cartesische coördinatenstelsel, dat je wel kent van je wiskundelessen op school, handig voor. De coördinaten zijn getallen die een specifiek punt op het scherm aanduiden. Deze getallen kunnen worden opgeslagen als integers in de variabelen van je programma.

Rasters en Cartesische coördinaten

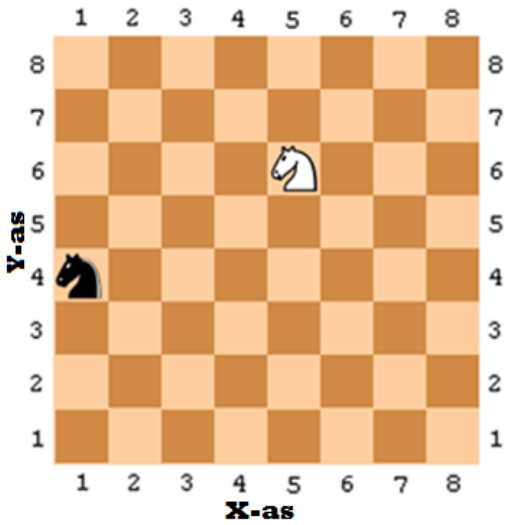


Afbeelding 12-1: Een voorbeeld van een schaakbord met een zwart paard op a4 en een wit paard op e6.

De vakjes op een schaakbord worden meestal aangeduid door de rij en de kolom aan te geven met letters en cijfers. Afbeelding 12-1 is een schaakbord waar elke rij en kolom van een letter en cijfer zijn voorzien.

Een 'coördinaat' voor een vakje op het schaakbord is een combinatie van een rij en kolom. Hier zien we een voorbeeld met het paard. Het witte paard in figuur 12-1 staat op punt e6 en het zwarte paard staat op punt a4

Dit schaakbord kun je dus eigenlijk zien als een Cartesisch coördinatenstelsel. Door de rij en de kolom te noemen, geef je een coördinaat dat één specifiek vakje aanduidt. Misschien heb je het Cartesische coördinatenstelsel al gehad bij grafieken van wiskunde op school. Dan weet je ook dat er meestal cijfers worden gebruikt voor zowel de rijen als de kolommen. Dat schaakbord ziet eruit zoals in Afbeelding 12-2:



Afbeelding 12-2: Hetzelfde schaakbord, maar nu met cijfercoördinaten voor zowel rijen als kolommen.

De getallen die van links naar rechts lopen en de kolommen aanduiden, maken deel uit van de **X-as**. De getallen die van onder naar boven lopen en de rijen aanduiden, maken deel uit van de **Y-as**. Coördinaten worden altijd aangegeven met eerst de X-coördinaat en dan de Y-coördinaat. In Afbeelding 12-2 staat het witte paard op coördinaat 5,6 en niet op 6,5. Het zwarte paard staat op coördinaat 1,4 en dat is dus iets heel anders dan 4,1.

Als je het zwarte paard op de positie van het witte paard zou willen zetten, moet het zwarte paard dus twee vakjes omhoog en vier vakjes naar rechts gaan. Maar je hoeft niet naar het bord te

kijken om dit uit te vinden Als je weet dat het witte paard op 5,6 staat en het zwarte paard op 1,4, dan kun je die twee getallen van elkaar aftrekken om te weten hoe het paard moet bewegen.

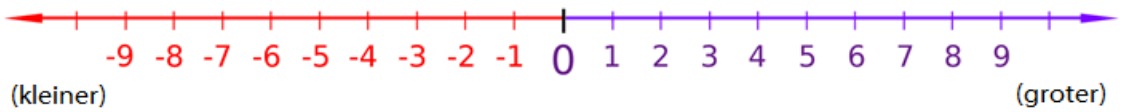
Trek de X-coördinaat van het zwarte paard af van de X-coördinaat van het witte paard: 5 - 1 = 4. Het zwarte paard moet dus vier vakjes opzij bewegen langs de X-as.

Trek de Y-coördinaat van het zwarte paard af van de Y-coördinaat van het witte paard: 6 - 4 = 2. Het zwarte paard moet dus 2 vakjes omhoog langs de Y-as bewegen.

Door gewoon even te rekenen met de coördinaten, kun je de afstand tussen twee coördinaten uitvinden.

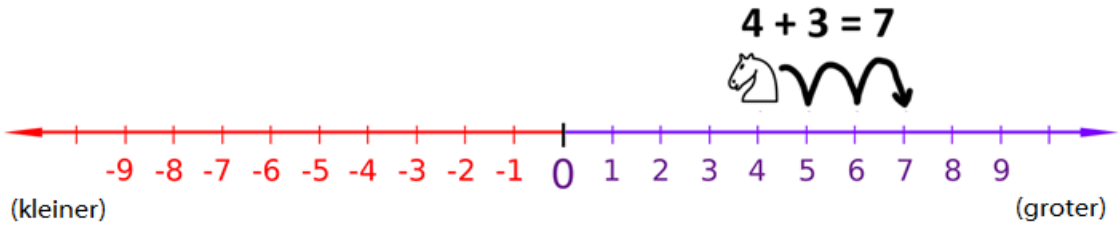
Negatieve getallen

Cartesische coördinaten gebruiken negatieve getallen. **Negatieve getallen** zijn getallen die kleiner zijn dan nul. Voor een negatief getal staat altijd een minteken. -1 is kleiner dan 0. En -2 is kleiner dan -1. Als je aan gewone getallen (de **positieve getallen**) denkt als beginnend bij 1 en groter wordend, kun je aan negatieve getallen denken als beginnend bij -1 en kleiner wordend. 0 is niet positief en niet negatief. In Afbeelding 12-3 zie je op een getallenlijn dat de positieve getallen naar rechts toe steeds groter worden en de negatieve getallen naar links toe steeds kleiner worden.



Afbeelding 12-3: Een getallenlijn

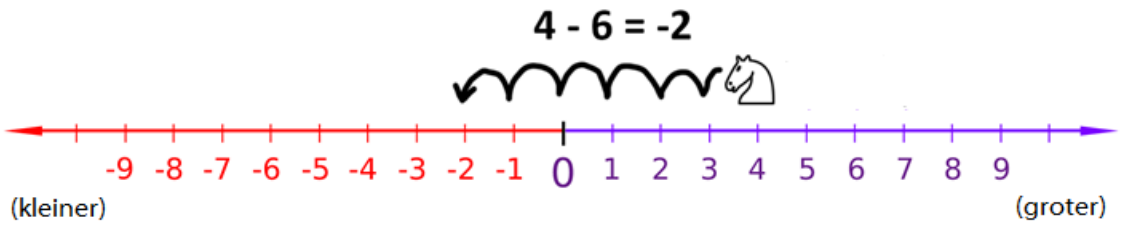
Een getallenlijn is een handig hulpmiddel om op te tellen en af te trekken met negatieve getallen. De expressie 4 + 3 kun je zien als het witte paard dat op positie 4 staat en 3 vakjes naar rechts gaat (optellen betekent groter worden, dus naar rechts gaan).



Afbeelding 12-4: Als je het witte paard naar rechts verplaatst, wordt er bij de coördinaat opgeteld.

Zoals je kunt zien in Afbeelding 12-4, gaat het witte paard naar positie 7. Nogal wiedes hè, want 4 + 3 is 7.

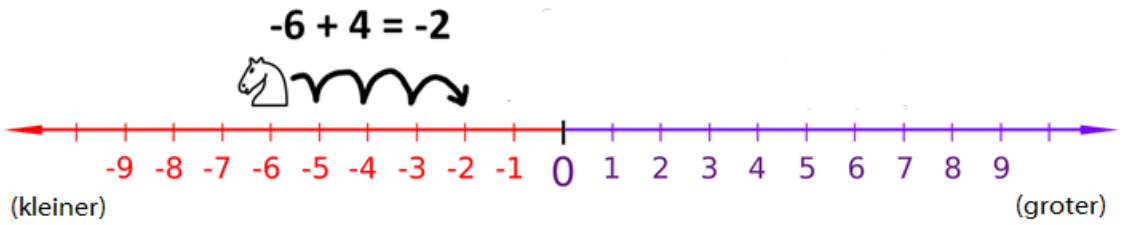
Aftrekken doe je door het paard naar links te verplaatsen. Aftrekken betekent kleiner worden, dus naar links gaan. 4 - 6 zou betekenen dat het witte paard op positie 4 begint en 6 vakjes naar links gaat, zoals in Afbeelding 12-5.



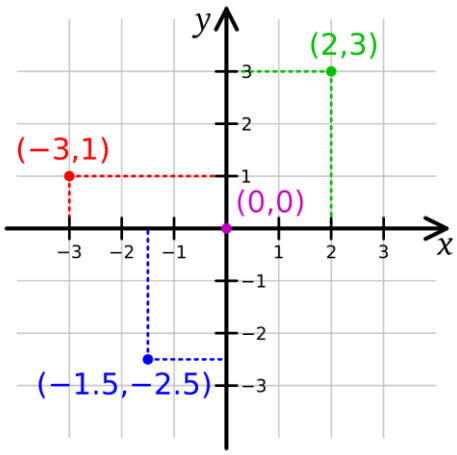
Afbeelding 12-5: Als je het witte paard naar links verplaatst, wordt er van de coördinaat afgetrokken.

Het witte paard gaat naar positie -2. Dus 4 - 6 is gelijk aan -2.

Als je een negatief getal erbij optelt of aftrekt, zou het witte paard in de *tegenovergestelde* richting gaan. Als je een negatief getal erbij optelt, gaat het paard naar *links*. Als je een negatief getal ervan aftrekt, gaat het paard naar *rechts*. De expressie -6 - -4 is gelijk aan -2. Het paard begint op -6 en gaat 4 vakjes naar *rechts*. Zoals je ziet is -6 - -4 dus hetzelfde als -6 + 4.



Afbeelding 12-6: Zelfs als het witte paard op een negatieve coördinaat begint, wordt bij naar rechts gaan de coördinaat groter.



Afbeelding 12-7: Twee getallenlijnen haaks op elkaar vormen een Cartesisch coördinatenstelsel.

Je kunt een X-as zien als een getallenlijn. Voeg een tweede getallenlijn van boven naar beneden toe als de Y-as. Als je deze twee getallenlijnen haaks op elkaar zet, heb je een Cartesisch coördinatenstelsel zoals in Afbeelding 12-7.

Als je een positief getal optelt (of een negatief getal aftrekt) gaat het paard omhoog langs de getallenlijn en als je een positief getal aftrekt (of een negatief getal optelt) gaat het paard omlaag langs de getallenlijn.

De coördinaat 0,0 heet de **oorsprong**.

Handige rekentrucjes

Negatieve getallen optellen en aftrekken is gemakkelijk als je naar een getallenlijn kunt kijken. Maar als je die niet voor je hebt, zijn er andere handigheidjes. Hier zijn drie trucjes die helpen bij het optellen en aftrekken van negatieve getallen.

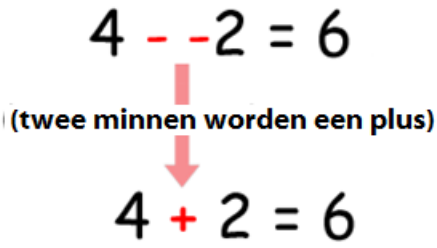
Truc 1: "Een minteken eet het plusteken links ervan op"

Als je een minteken ziet met een plusteken links ervan, kun je het plusteken vervangen door het minteken. Fantaseer erbij dat de min de plus 'opeet'. Het antwoord is hetzelfde want als je een negatieve waarde ergens bij optelt, is dat hetzelfde als een positieve waarde ervan aftrekken. 4 + -2 en 4 - 2 zijn allebei gelijk aan 2.

Afbeelding 12-8: Truc 1 - Een positief en een negatief getal bij elkaar optellen.

Truc 2: " Twee minnen worden een plus"

Als je twee mintekens naast elkaar ziet, zonder getal ertussen, worden ze samen een plus. Het antwoord is hetzelfde want als je een negatieve waarde ergens van aftrekt, is dat hetzelfde als een positieve waarde erbij optellen.

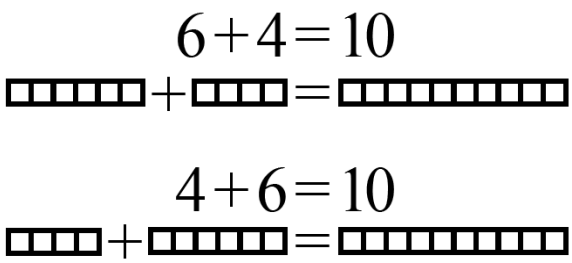


Afbeelding 12-9: Truc 2 - Een positief en een negatief getal van elkaar aftrekken.

Truc 3: De commutatieve eigenschap van optellen

In een optelsom kun je de getallen die je bij elkaar optelt, altijd omwisselen. Dit heet de **commutatieve eigenschap** van optellen. Of je nu 6 + 4 zegt, of 4 + 6, maakt niets uit voor het antwoord.

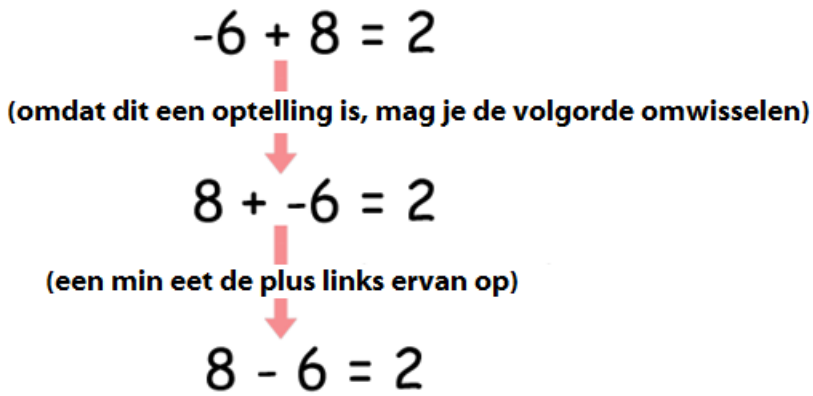
Als je de vakjes telt in Afbeelding 12-10, kun je zien dat het niet uitmaakt of je de getallen voor een optelsom omwisselt.



Afbeelding 12-10: Truc 3 - De commutatieve eigenschap van optellen.

Stel dat je een negatief getal en een positief getal bij elkaar optelt, bijvoorbeeld -6 + 8. Omdat je getallen *optelt*, mag je de volgorde van de getallen omwisselen, zonder dat het antwoord verandert. -6 + 8 is hetzelfde als 8 + -6.

Als je dan kijkt naar 8 + -6, weet je dat het minteken het plusteken links ervan opeet, en wordt de som 8 - 6 = 2. Maar dit betekent dat -6 + 8 dus ook 2 is! Je hebt de som door elkaar gehaald en aangetoond dat de som dan dezelfde uitkomst heeft, en op deze manier is het makkelijker geworden voor ons om de som op te lossen zonder een rekenmachine of computer te gebruiken.



Afbeelding 12-11: De rekentrucjes samen gebruiken.

Absolute waarden en de functie abs()

De **absolute waarde** van een getal is dat getal zonder plus- of minteken ervoor. Positieve getallen zie je dan niet veranderen, maar negatieve getallen worden positief. De absolute waarde van -4 is bijvoorbeeld 4. De absolute waarde van -7 is 7. De absolute waarde van 5 (dat positief is) is gewoon 5.

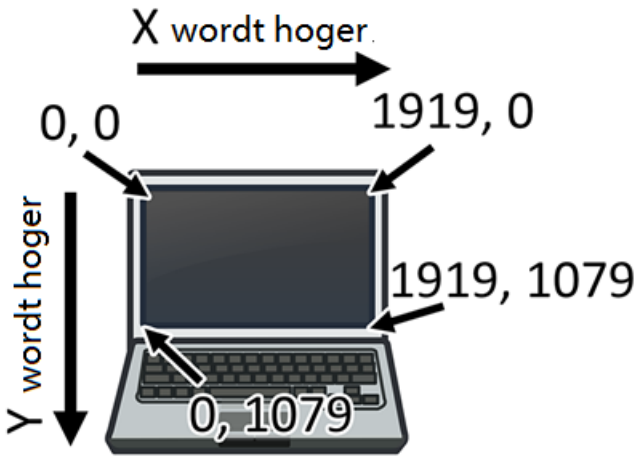
Je kunt de afstand tussen twee object uitrekenen door hun posities van elkaar af te trekken en de uitkomst om te zetten in een absolute waarde. Stel dat het witte paard op positie 4 staat en het zwarte paard op positie -2. De afstand zou 6 zijn, omdat 4 - -2 gelijk is aan 6, en de absolute waarde van 6 is 6.

Dit werkt ongeacht de volgorde van de getallen. -2 - 4 (dus negatieve twee min vier) is -6, en de absolute waarde van -6 is ook 6.

De functie `abs()` in Python retourneert de absolute waarde van een integer. Typ het volgende in de interactieve shell:

```
>>> abs(-5)
5
>>> abs(42)
42
>>> abs(-10,5)
10,5
```

Coördinatenstelsel van een computerscherm



Afbeelding 12-12: Het Cartesische coördinatenstelsel op een computerscherm.

Computerschermen hebben meestal een coördinatenstelsel waarbij de oorsprong (0,0) zich in de linkerbovenhoek bevindt. De coördinatengetallen worden hoger als je naar rechts en naar beneden gaat. Dit zie je in Afbeelding 12-12. Er zijn geen negatieve coördinaten. De meeste computergraphics gebruiken dit coördinatenstelsel en jij gaat het ook gebruiken in de verdere games in dit boek.

Samenvatting

Voor programmeren hoef je echt niet veel te weten over wiskunde, hoor. Tot aan dit hoofdstuk, konden we prima toe met wat optellen en wat vermenigvuldigen.

Je moet wel iets weten over coördinaten om te weten hoe je op een tweedimensionaal computerscherm een bepaalde positie kunt aanwijzen. Coördinaten hebben twee getallen: de X-coördinaat en de Y-coördinaat. De X-as loopt van links naar rechts en de Y-as gaat van boven naar beneden. Op een computerscherm staat de oorsprong (het punt 0,0) in de linkerbovenhoek en lopen de coördinaten op naar rechts en naar beneden.

Met de drie trucjes die je in dit hoofdstuk hebt geleerd, kun je gemakkelijk positieve en negatieve integers bij elkaar optellen. Het eerste trucje is dat een minteken een plusteken links ervan 'opeet'. Het tweede trucje is dat twee mintekens naast elkaar, een plusteken worden. Het derde trucje is dat je de twee termen in een optelsom altijd mag omwisselen. De uitkomst blijft dan hetzelfde.

In de rest van het boek zullen we deze begrippen gaan gebruiken in onze games omdat we gaan spelen met tweedimensionale ruimten. Voor alle games met graphics moet je een beetje begrijpen hoe Cartesische coördinaten werken.

Hoofdstuk 13

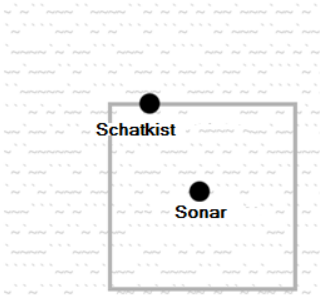
SCHATZOEKEN MET SONAR

In dit hoofdstuk behandelen we:
- Datastructuren
- De methode `remove()` voor lijsten
- De methode `isdigit()` voor strings
- De functie `sys.exit()`

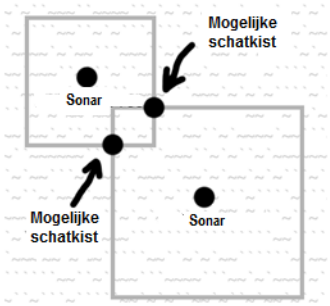
Bij de game in dit hoofdstuk gaan we voor het eerst gebruikmaken van het coördinatenstelsel waarover je in Hoofdstuk 12 hebt geleerd. De game heeft ook **datastructuren** (en dat is gewoon een dure naam voor complexe variabelen zoals lijsten met lijsten). Nu de games die je programmeert, wat ingewikkelder worden, moet je je gegevens gaan organiseren in datastructuren.

In de game in dit hoofdstuk plaatst de speler sonars op verschillende plaatsen in de oceaan om gezonken schatkisten te vinden. Sonar is een technologie die door schepen wordt gebruikt om objecten onder het wateroppervlak te vinden. De sonars in dit spel vertellen de speler hoe ver weg de dichtstbijzijnde schatkist is, maar vertellen er niet de richting bij. Maar door meerdere sonars in het water te laten zakken, kan de speler uitvinden waar de schatkist ligt.

Er zijn drie schatkisten die kunnen worden opgediept, maar de speler heeft maar 16 sonars tot zijn beschikking. Stel je voor dat je de schatkist niet zou kunnen zien in de volgende afbeelding. Omdat elke sonar alleen de afstand uit kan vinden, maar niet de richting, bevinden de mogelijke plaatsen zich binnen een vierkante kring rondom de sonar (zie Afbeelding 13-1).



Afbeelding 13-1: De vierkante kring rondom de sonar raakt de verborgen schatkist.

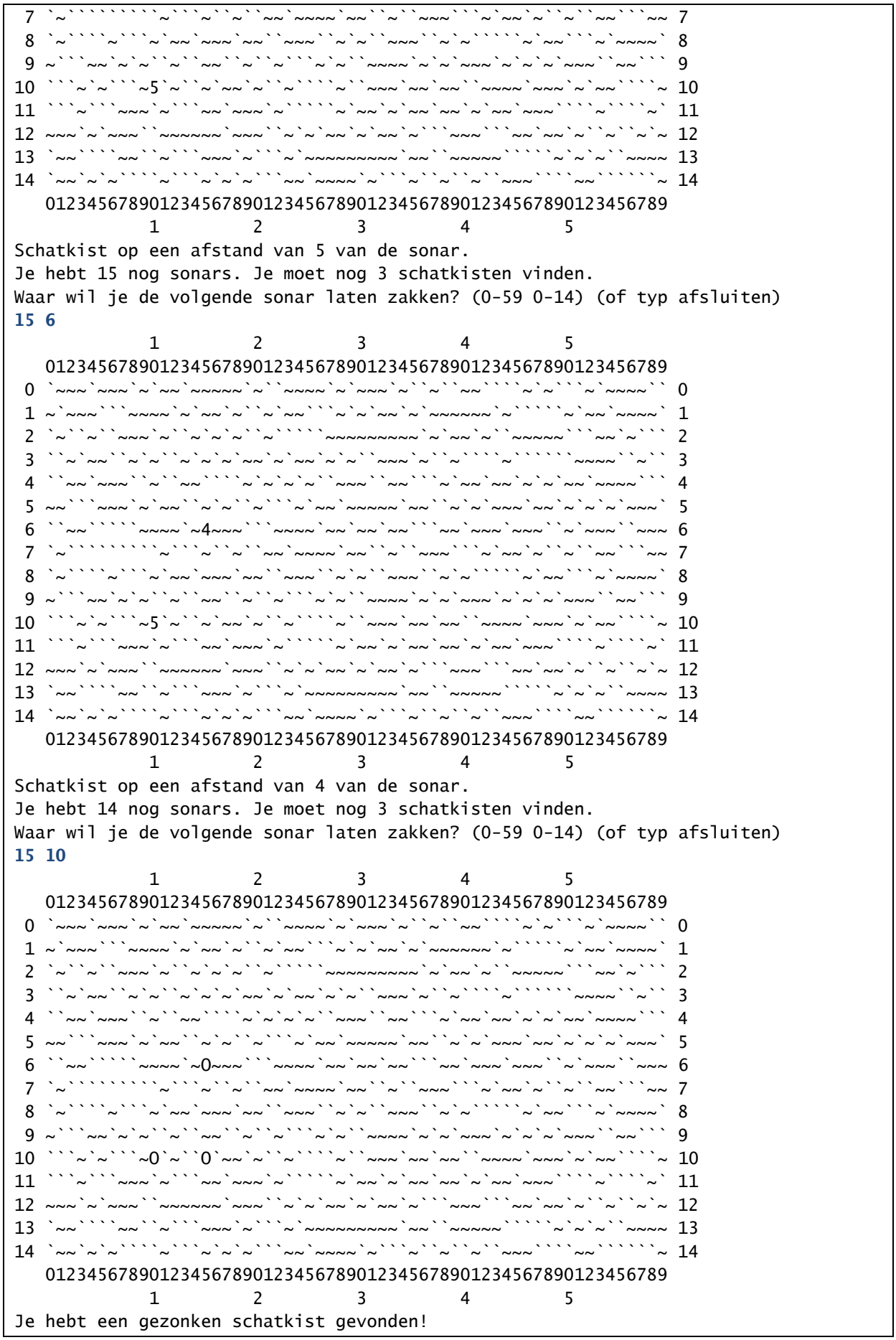


Afbeelding 13-2: Door meerdere vierkante kringen te combineren, kun je uitvinden waar de schatkisten liggen.

Maar als je meerdere sonars laat samenwerken, kun je de precieze plek vinden waar de kringen elkaar raken. Zie Afbeelding 13-2. (Normaliter horen deze kringen cirkels te zijn, maar in dit spel gebruiken we vierkanten, dat maakt het programmeren wat makkelijker.)

Voorbeelduitvoer van Schatzoeken met sonar

```
S O N A R !
Wil je weten hoe je het moet spelen? (ja/nee)
nee
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890123456789
 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~``~~````~`~```~`~~~~`` 0
 1 ~`~~~```~~~~`~`~~`~```~`~~```~`~`~~`~`~~~~~~`~`````~`~~`~~~~` 1
 2 `~``~``~~~`~``~`~`~``~`````~~~~~~~~~`~`~~`~``~~~~~```~~`~``` 2
 3 ``~`~~``~`~``~`~`~~`~`~~`~`~``~~~`~``~````~``````~~~~``~`` 3
 4 ``~~`~~~``~``~~````~`~`~`~``~~~``~~```~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~``~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~`~~~` 5
 6 ``~~`````~~~~`~`~~~```~~~~`~~`~~`~~```~~`~~~`~~~``~`~~~``~~~ 6
 7 `~`````````~```~``~``~~`~~~~`~~``~``~~~```~`~~`~``~``~~```~~ 7
 8 `~````~```~`~~`~~~`~~``~~~``~`~``~~~``~`~`````~`~~```~`~~~~` 8
 9 ~```~~`~`~``~``~~``~`~`~```~`~``~~~~`~`~`~~~`~`~`~`~~~``~~``` 9
10 ```~`~```~``~`~~`~``~````~``~~~`~~`~~``~~~~`~~~`~`~~````~ 10
11 ```~```~~~`~```~~`~~~`~`````~`~~`~`~~`~~`~`~~`~~~````~````~` 11
12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~`~``~``~`~ 12
13 `~~````~~``~```~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~``~~~~ 13
14 `~~`~`~````~```~`~`~```~~`~~~~`~```~``~``~``~~~````~~``````~ 14
    012345678901234567890123456789012345678901234567890123456789
              1         2         3         4         5
Je hebt 16 nog sonars. Je moet nog 3 schatkisten vinden.
Waar wil je de volgende sonar laten zakken? (0-59 0-14) (of typ afsluiten)
10 10
              1         2         3         4         5
    012345678901234567890123456789012345678901234567890123456789
 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~``~~````~`~```~`~~~~`` 0
 1 ~`~~~```~~~~`~`~~`~```~`~~```~`~`~~`~`~~~~~~`~`````~`~~`~~~~` 1
 2 `~``~``~~~`~``~`~`~``~`````~~~~~~~~~`~`~~`~``~~~~~```~~`~``` 2
 3 ``~`~~``~`~``~`~`~~`~`~~`~`~``~~~`~``~````~``````~~~~``~`` 3
 4 ``~~`~~~``~``~~````~`~`~`~``~~~``~~```~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~``~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~`~~~` 5
 6 ``~~`````~~~~`~`~~~```~~~~`~~`~~`~~```~~`~~~`~~~``~`~~~``~~~ 6
```


Schatkist op een afstand van 5 van de sonar.
Je hebt 15 nog sonars. Je moet nog 3 schatkisten vinden.
Waar wil je de volgende sonar laten zakken? (0-59 0-14) (of typ afsluiten)
15 6
Schatkist op een afstand van 4 van de sonar.
Je hebt 14 nog sonars. Je moet nog 3 schatkisten vinden.
Waar wil je de volgende sonar laten zakken? (0-59 0-14) (of typ afsluiten)
15 10
Je hebt een gezonken schatkist gevonden!

```
Je hebt 13 nog sonars. Je moet nog 2 schatkisten vinden.
Waar wil je de volgende sonar laten zakken? (0-59 0-14) (of typ afsluiten)

...slaan we even over, anders wordt het zo'n lang verhaal...

            1         2         3         4         5
   012345678901234567890123456789012345678901234567890123456789
 0 `~~~`~~~`~`~~`~~~~~`~``~~~~`~`~~~`~``~``~~````~`~```~`~~~~`` 0
 1 ~`~~~```~~~~`~`~~`~`~`~~```~0~`~~`~`~~~~~~`~`````~`~~`~~~~` 1
 2 `~```~``~~~`~```~`~`~``~`````~~0~~~0~~`~`~~`~``~~~~~```~~`~``` 2
 3 ``~3~~``8`~``~`~`~~`~`~~`~`~``~~~`~`0~````~``````~~~~``~`` 3
 4 ``~~`~~~``~``~~````~`~`~`~0`~~0``~~```~`~~`~~`~`~`~~`~~~~``` 4
 5 ~~```~~~`~`~~``~`~``~```~`~~`~~~~~`~~``~`~`~~~`~~`~`~`~`~~~` 5
 6 ``~~`````~~~~`~0~~~```~~~~`~~`~~`~~```~~`~~~`~~~``0`~~~``~~~ 6
 7 `~`````````~```~``~``~~`~~~~`~~``~``~~~```~`~~`~``~``~~```~~ 7
 8 `~````~```~`~~`~~~`~~``~~~``~`~``~~~``~`0```0`~`~~```~`~~~~` 8
 9 ~```~~`~`~``~``~~``~``~`~~~`~0~``~~~~`~`~`~~~`~`~`~`~~~``~~``` 9
10 ```~`~```~0`~``0`~~`~``~````~``~~~`~~`~~``~~~~`~~~`~`~~````~ 10
11 ```~```~~~`~```~~`~~~`~`````~`~~`~`~~`~~`~`~~`~~~````~````~` 11
12 ~~~`~`~~~``~~~~~~`~~~``~`~`~~`~`~~`~```~~~```~~`~~`~`~``~``~`~ 12
13 `~~````~~``~```~~~`~```~`~~~~~~~~~`~~``~~~~~`````~`~`~``~~~~ 13
14 `~~`~`~````~```~`~`~```~~`~~~~`~```~``~``~``~~~````~~``````~ 14
   012345678901234567890123456789012345678901234567890123456789
            1         2         3         4         5
Schatkist op een afstand van 4 van de sonar.
We hebben geen sonars meer! Nu moeten we terug naar de haven
terwijl er nog schatkisten liggen! Game over.
    De overige schatkisten lagen hier:
    0, 4
Wil je nog een keer spelen? (ja of nee)
nee
```

Broncode van Schatzoeken met sonar

Hier staat de broncode voor het spel. Typ het in een nieuw bestand, sla het bestand op als *sonar.py* en voer het uit door op **F5** te drukken. Als je een foutmelding krijgt nadat je deze code hebt ingetypt, vergelijk jouw code dan met de code van het boek met behulp van de online diff tool op http://invpy.com/diff/sonar.

sonar.py

```
 1. # Sonar
 2.
 3. import random
 4. import sys
 5.
 6. def tekenBord(bord):
 7.     # We tekenen de datastructuur voor het bord.
 8.
 9.     hregel = '    ' # eerste ruimte voor de getallen langs de linkerkant van
het bord
10.     for i in range(1, 6):
11.         hregel += (' ' * 9) + str(i)
12.
13.     # de getallen weergeven langs de bovenkant
14.     print(hregel)
```

```
 15.     print('   ' + ('0123456789' * 6))
 16.     print()
 17.
 18.     # alle 15 rijen weergeven
 19.     for i in range(15):
 20.         # getallen met 1 cijfer krijgen een extra spatie
 21.         if i < 10:
 22.             extraSpatie = ' '
 23.         else:
 24.             extraSpatie = ''
 25.         print('%s%s %s %s' % (extraSpatie, i, haalRij(bord, i), i))
 26.
 27.     # de getallen weergeven langs de onderkant
 28.     print()
 29.     print('   ' + ('0123456789' * 6))
 30.     print(hregel)
 31.
 32.
 33. def haalRij(bord, rij):
 34.     # een string retourneren van de datastructuur van het bord op een bepaalde
rij.
 35.     bordRij = ''
 36.     for i in range(60):
 37.         bordRij += bord[i][rij]
 38.     return bordRij
 39.
 40. def haalNieuwBord():
 41.     # een nieuwe datastructuur maken voor het 60x15-bord
 42.     bord = []
 43.     for x in range(60): # de hoofdlijst is een lijst met 60 lijsten
 44.         bord.append([])
 45.         for y in range(15): # elke lijst in de hoofdlijst heeft 15 strings van
1 teken
 46.             # verschillende tekens gebruiken voor de oceaan om hem leesbaarder
te maken.
 47.             if random.randint(0, 1) == 0:
 48.                 bord[x].append('~')
 49.             else:
 50.                 bord[x].append('`')
 51.     return bord
 52.
 53. def haalWillekeurigeSchatkisten(aantalKisten):
 54.     # een lijst maken met datastructuren voor een schatkist (lijsten met
dubbele items van x,y-coördinaten)
 55.     kisten = []
 56.     for i in range(aantalKisten):
 57.         kisten.append([random.randint(0, 59), random.randint(0, 14)])
 58.     return kisten
 59.
 60. def isGeldigeZet(x, y):
 61.     # retourneert True als de coördinaten op het bord voorkomen, anders False.
 62.     return x >= 0 and x <= 59 and y >= 0 and y <= 14
 63.
 64. def doeZet(bord, kisten, x, y):
 65.     # de datastructuur van het bord veranderen met een sonarteken. Schatkisten
verwijderen
```

```
 66.     # van de schatkistenlijsten zodra ze worden gevonden. False retourneren
als dit een ongeldige zet is.
 67.     # Anders de string retourneren van het resultaat van deze zet.
 68.     if not isGeldigeZet(x, y):
 69.         return False
 70.
 71.     kleinsteAfstand = 100 # alle kisten zijn dichterbij dan 100.
 72.     for cx, cy in kisten:
 73.         if abs(cx - x) > abs(cy - y):
 74.             afstand = abs(cx - x)
 75.         else:
 76.             afstand = abs(cy - y)
 77.
 78.         if afstand < kleinsteAfstand: # we willen de dichtstbijzijnde
schatkist.
 79.             kleinsteAfstand = afstand
 80.
 81.     if kleinsteAfstand == 0:
 82.         # xy is de locatie van een schatkist!
 83.         kisten.remove([x, y])
 84.         return 'Je hebt een gezonken schatkist gevonden!'
 85.     else:
 86.         if kleinsteAfstand < 10:
 87.             bord[x][y] = str(kleinsteAfstand)
 88.             return 'Schatkist op een afstand van %s van de sonar.' %
(kleinsteAfstand)
 89.         else:
 90.             bord[x][y] = 'O'
 91.             return 'Sonar heeft niets gevonden. Alle schatkisten buiten
bereik.'
 92.
 93.
 94. def zetSpelerInvoeren():
 95.     # We laten de speler zijn zet invoeren. Een lijst met dubbele items van
x,y-coördinaten retourneren.
 96.     print('Waar wil je de volgende sonar laten zakken? (0-59 0-14) (of typ
afsluiten)')
 97.     while True:
 98.         zet = input()
 99.         if zet.lower() == 'afsluiten':
100.             print('Leuk dat je meedeed!')
101.             sys.exit()
102.
103.         zet = zet.split()
104.         if len(zet) == 2 and zet[0].isdigit() and zet[1].isdigit() and
isGeldigeZet(int(zet[0]), int(zet[1])):
105.             return [int(zet[0]), int(zet[1])]
106.         print('Typ een getal van 0 tot 59, een spatie, en dan een getal van 0
tot 14.')
107.
108.
109. def nogmaalsSpelen():
110.     # Deze functie retourneert True als de speler nog een keer wil spelen,
anders is het False.
111.     print('Wil je nog een keer spelen? (ja of nee)')
112.     return input().lower().startswith('j')
113.
```

```
114.
115. def toonUitleg():
116.     print('''Uitleg:
117. Jij bent de kapitein van de Simon, een schip dat naar schatten zoekt. Je
huidige
118. missie is het vinden van drie gezonken schatkisten die zich ergens in de
oceaan
119. bevinden.
120.
121. Typ de coördinaten van het punt in de oceaan waar je een sonar wilt laten
122. zakken. De sonar kan meten hoe ver weg de dichtstbijzijnde schatkist ligt.
123. Als de schatkist bijvoorbeeld op locatie x zou staan
124. op de afbeelding hieronder, zie je hier aangegeven wat de
125. zouden zijn als je een van de omringende
126. vakjes zou kiezen om de sonar te laten zakken.
127. vakjes zou kiezen om de sonar te laten zakken.     444444444
128.     4       4
129.     4 22222 4
130.     4 2   2 4
131.     4 2 d 2 4
132.     4 2   2 4
133.     4 22222 4
134.     4       4
135.     444444444
136. Druk op enter om door te gaan...''')
137.     input()
138.
139.     print('''Hier is bijvoorbeeld een schatkist (de c) op een afstand van 2
vanaf
140. de sonar (de d):
141.
142.     22222
143.     c   2
144.     2 d 2
145.     2   2
146.     22222
147.
148. Het punt waar de sonar is uitgezet, wordt hier aangegeven met een 2.
149.
150. De schatkisten worden niet verplaatst. Sonars kunnen schatkisten opsporen
151. tot een afstand van 9. Als alle schatkisten buiten het bereik van een sonar
152. zijn, wordt het punt gemarkeerd met een O.
153.
154. Als een sonar wordt neergelaten bovenop een schatkist, heb je
155. de plaats van de schatkist gevonden, en wordt deze opgehaald. De sonar
156. blijft daar dan.
157.
158. Wanneer je een schatkist hebt opgehaald, worden alle sonars bijgewerkt om de
volgende dichtstbijzijnde
159. schatkist op te sporen.
160. Druk op enter om door te gaan...''')
161.     input()
162.     print()
163.
164.
165. print('S O N A R !')
166. print()
```

```
167. print('Wil je weten hoe je het moet spelen? (ja/nee)')
168. if input().lower().startswith('j'):
169.     toonUitleg()
170.
171. while True:
172.     # spel opstellen
173.     sonars = 16
174.     hetBord = haalNieuwBord()
175.     deSchatkisten = haalWillekeurigeSchatkisten(3)
176.     tekenBord(hetBord)
177.     vorigeZetten = []
178.
179.     while sonars > 0:
180.         # Begin van de beurt:
181.
182.         # sonars/schatkiststatus laten zien
183.         if sonars > 1: extraSsonar = 's'
184.         else: extraSsonar = ''
185.         if len(deSchatkisten) > 1: extraEnSchatkist = 'en'
186.         else: extraEnSchatkist = ''
187.         print('Je hebt nog %s sonar%s. Je moet nog %s schatkist%s vinden.' %
(sonars, extraSsonar, len(deSchatkisten), extraEnSchatkist))
188.
189.         x, y = zetSpelerInvoeren()
190.         vorigeZetten.append([x, y]) # we moeten alle zetten bijhouden zodat de
sonars kunnen worden bijgewerkt.
191.
192.         zetResultaat = doeZet(hetBord, deSchatkisten, x, y)
193.         if zetResultaat == False:
194.             continue
195.         else:
196.             if zetResultaat == 'Je hebt een gezonken schatkist gevonden!':
197.                 # alle sonars op de kaart moeten worden bijgewerkt.
198.                 for x, y in vorigeZetten:
199.                     doeZet(hetBord, deSchatkisten, x, y)
200.             tekenBord(hetBord)
201.             print(zetResultaat)
202.
203.         if len(deSchatkisten) == 0:
204.             print('Je hebt alle gezonken schatkisten gevonden! Gefeliciteerd,
kapitein!')
205.             break
206.
207.         sonars -= 1
208.
209.     if sonars == 0:
210.         print('We hebben geen sonars meer! Nu moeten we terug naar de haven')
211.         print('terwijl er nog schatkisten liggen! Game over.')
212.         print('    De overige schatkisten lagen hier:')
213.         for x, y in deSchatkisten:
214.             print('    %s, %s' % (x, y))
215.
216.     if not nogmaalsSpelen():
217.         sys.exit()
```

Het programma ontwerpen

Voordat we de broncode gaan uitleggen, moet je het spel eerst maar even een paar keer spelen, om te begrijpen hoe het werkt. Het spel Sonar maakt gebruik van lijsten met lijsten en andere ingewikkelde variabelen, die we **datastructuren** noemen. Datastructuren zijn variabelen die waarden gebruiken die op een bepaalde manier zijn geordend. In het hoofdstuk Boter-kaas-en-eieren zagen we al eerder een datastructuur, namelijk een lijst met strings waarmee we het bord aanduidden. De string stelde een X, O, of een leeg vak aan en het indexnummer van de string in de lijst duidde het vakje op het bord aan. Het spel Sonar heeft soortgelijke datastructuren waarmee we de locatie van de schatkisten en de sonars gaan aangeven.

Hoe de code werkt

```
1. # Sonar
2.
3. import random
4. import sys
```

De regels 3 en 4 importeren de modules `random` en `sys`. De module `sys` bevat de functie `exit()` waarmee het programma direct kan worden afgesloten. Deze functie wordt verderop in het programma gebruikt.

Het bord tekenen

```
6. def tekenBord(bord):
```

Het spelbord van Sonar is een oceaan in een ASCII-tekening met daaromheen coördinaten op een X-as en een Y-as. Het accent grave (`) en de tilde (~), links naast de toets 1 op je toetsenbord, worden gebruikt om golven aan te geven. Het ziet er zo uit:

```
            1         2         3         4         5
  012345678901234567890123456789012345678901234567890123456789
 0 ~~~`~``~~~``~~~~``~`~`~`~`~~`~~~`~~`~``````~~`~``~`~~```~`~` 0
 1 `~`~````~~``~`~```~```~```~`~~~``~~`~~~``````~`~``~~``~~`~~` 1
 2 ```~~~~`~`~~```~~~``~````~~`~`~~`~`~`~```~~`~``~~`~`~~~~~~`~ 2
 3 ~~~~`~~~``~```~``~~`~`~~`~`~~``~````~`~````~```~`~`~`~`````~ 3
 4 ~```~~~~~`~~````~~~~```~~~`~`~`~````~`~~`~`~~``~~`~``~`~``~~ 4
 5 `~```~`~`~~`~~~```~~``~``````~~``~`~`~~~~`~~``~~~~~~`~```~~` 5
 6 ``~~`~~`~``~`````~````~~``~`~~~~`~~```~~~``~`~`~~``~~~```~~~ 6
 7 ``~``~~~~~~```~`~```~~~``~`~``~`~~~~~~``````~~~`~~`~~`~~`~~ 7
 8 ~~`~`~~```~``~~``~~~``~~`~`~~`~`~```~```~~~```~~~~~~`~`~~~~` 8
 9 ```~```~`~~~`~~```~``~``~~~```~````~```~`~~`~~~~~`~``~~~~~``` 9
10 `~~~~```~`~````~`~`~~``~`~~~~`~``~``~```~~``````~`~``~````` 10
11 ~~`~`~~`~``~`~~~`````````````~~`````~`~~``~`~~~~`~~~`~~`~ 11
12 ~~`~~~~```~~~`````~~``~`~`~~`````~`~~``~`````~~``~~~`~~`~ 12
13 `~`````~~``~`~~~```~~~~```~~`~`~~~`~`````~~`~```~``~`~~~~ 13
14 ~~~``~```~`````~~`~`~``~~`~``~`~~`~`~``~`~``~~``~`~``~```~~~ 14
  012345678901234567890123456789012345678901234567890123456789
            1         2         3         4         5
```

De tekening in de functie `tekenBord()` bestaat uit vier stappen.

- Eerst maak je een stringvariabele van de regel met 1, 2, 3, 4 en 5 met veel ruimte daartussen (om de coördinaten voor 10, 20, 30, 40 en 50 op de X-as aan te geven).
- Ten tweede gebruik je die string om de coördinaten op de X-as aan de bovenkant van het scherm aan te geven.
- De derde stap is het weergeven van elke rij van de oceaan met de coördinaten op de Y-as aan beide zijkanten van het scherm.
- De laatste stap is het nogmaals weergeven van de X-as onderaan het scherm. Doordat we de coördinaten aan alle vier de zijden van het bord tonen, is het gemakkelijker om te zien waar je een sonar kunt plaatsen.

De X-coördinaten aan de bovenkant tekenen

```
  7.     # We tekenen de datastructuur voor het bord.
  8.
  9.     hregel = '    ' # eerste ruimte voor de getallen langs de linkerkant van
het bord
 10.     for i in range(1, 6):
 11.         hregel += (' ' * 9) + str(i)
```

Kijk nog eens naar de bovenkant van het bord in Afbeelding 13-3. Er staan plustekens in plaats van spaties, zodat je de lege plekken gemakkelijker kunt tellen:

```
++++++++++++1+++++++++2+++++++++3 # eerste regel
+++0123456789012345678901234567890123456789 # tweede regel

+0 ~~~`~`` ~~~`` ~~~~` `~`~`~`~`~ ~~`~~~~` ``~`~` 0 # derde regel
```

Afbeelding 13-3: De spatietekens voor het weergeven van de bovenkant van het spelbord.

Tussen de cijfers op de eerste regel die de tientallen aangeven, staan steeds 9 spaties, en er staan dertien spaties vóór de 1. Op de regels 9 t/m 11 maken we deze string voor deze regel en we slaan de string op in een variabele met de naam `hregel`.

```
 13.     # de getallen weergeven langs de bovenkant
 14.     print(hregel)
 15.     print('    ' + ('0123456789' * 6))
 16.     print()
```

Om de cijfers langs de bovenkant van het sonarbord weer te geven, printen we eerst de inhoud van de variabele `hregel`. Vervolgens printen we eerst drie spaties op de volgende regel (zodat deze rij precies onder de vorige rij komt) en daarna de string `'0123456789012345678901234567890123456789012345678901234567890123456789'`. Maar om onszelf wat getyp te besparen, gebruiken we `('0123456789' * 6)`, wat hetzelfde resultaat heeft.

De 'rijen' van de oceaan tekenen

```
18.     # alle 15 rijen weergeven
19.     for i in range(15):
20.         # getallen met 1 cijfers krijgen een extra spatie
21.         if i < 10:
22.             extraSpatie = ' '
23.         else:
24.             extraSpatie = ''
25.         print('%s%s %s %s' % (extraSpatie, i, haalRij(bord, i), i))
```

De regel 19-25 printen alle rijen met golven in de oceaan met de cijfers aan beide zijde om de Y-as aan te geven. De `for`-lus print de rijen 0 t/m 14, samen met de rijnummers aan beide zijden van het bord.

Er is een klein probleempje. Getallen met maar één cijfer (zoals 0, 1, 2, enzovoort) nemen maar één spatie in beslag als ze worden getoond, maar getallen met twee cijfers (zoals 10, 11 en 12) nemen twee spaties in beslag. De regels zouden niet goed boven elkaar staan als de coördinaten niet evenveel ruimte in beslag nemen. Het zou er als volgt uitzien:

```
8 ~~`~`~~```~``~~``~~~``~~`~`~~`~`~```~```~~~```~~~~~~`~`~~~~` 8
9 ```~``~`~~~`~~```~``~``~~~```~````~```~`~~`~~~~~`~``~~~~~``` 9
10 `~~~~```~`~````~`~`~~``~`~~~~`~``~``~~`~~~~~~~`~`~``~````` 10
11 ~~`~`~~`~``~`~~~`````````````````~~`````~`~~``~`~~~~`~~~`~~`~ 11
```

Gelukkig is de oplossing gemakkelijk. We voegen een spatie toe aan alle getallen met maar één cijfer. Op de regels 21-24 stellen we de variabele `extraSpatie` in op ofwel een spatie of een lege string. De variabele `extraSpatie` wordt altijd geprint, maar alleen bij de getallen met één cijfer, bevat de variabele een spatieteken. Bij de andere getallen is de string leeg. Op deze manier staan alle rijen netjes onder elkaar.

De functie `haalRij()` krijgt een rijnummer doorgegeven en retourneert een string die de golven weergeeft van die rij. De twee parameters van deze functie zijn de datastructuur voor het bord in de variabele `bord` en een rijnummer. We gaan deze functie even bekijken.

De X-coördinaten aan de onderkant tekenen

```
27.     # de getallen weergeven langs de onderkant
28.     print()
29.     print('   ' + ('0123456789' * 6))
30.     print(hregel)
```

De regels 27-30 zijn gelijk aan de regels 13-16. Ze printen de coördinaten van de X-as onder aan het scherm.

De status van een rij in de oceaan halen

```
33. def haalRij(bord, rij):
34.     # een string retourneren van de datastructuur van het bord op een bepaalde
rij.
35.     bordRij = ''
36.     for i in range(60):
```

```
37.         bordRij += bord[i][rij]
38.     return bordRij
```

De parameter `bord` is een datastructuur voor alle golven in de oceaan, maar de functie `haalRij()` maakt een string aan voor één enkele rij.

Eerst stellen we `bordRij` in op een lege string. De coördinaat van de Y-as wordt doorgegeven als de parameter `rij`. De string wordt gemaakt door concatenatie van `bord[0][rij]`, `bord[1][rij]`, `bord[2][rij]` enzovoort tot aan `bord[59][rij]`. Dat gaat zo omdat de rij 60 tekens bevat, van indexnummer `0` t/m indexnummer `59`.

De `for`-lus op regel 36 itereert langs de integers `0` t/m `59`. Bij elke iteratie wordt het volgende teken in de datastructuur voor het bord achteraan `bordRij` gekopieerd. Wanneer de lus klaar is bevat `bordRij` de hele ASCII-tekening voor die rij en wordt dit geretourneerd.

Het bord voor een nieuw spel weergeven

```
40. def haalNieuwBord():
41.     # Een nieuwe datastructuur maken voor het 60x15-bord
42.     bord = []
43.     for x in range(60): # de hoofdlijst is een lijst met 60 lijsten
44.         bord.append([])
```

Bij de start van elk nieuw spel hebben we een nieuwe datastructuur voor `bord` nodig. De `bord`-datastructuur is een lijst met strings. De eerste lijst stelt de X-coördinaat voor. Aangezien het bord van het spel 60 tekens breed is, moet de eerste lijst 60 lijsten bevatten. We maken een `for`-lus die 60 lege lijsten achteraan de lijst toevoegt.

```
45.         for y in range(15): # elke lijst in de hoofdlijst heeft 15 strings van
1 teken
46.             # verschillende tekens gebruiken voor de oceaan om hem leesbaarder
te maken.
47.             if random.randint(0, 1) == 0:
48.                 bord[x].append('~')
49.             else:
50.                 bord[x].append('`')
```

Maar `bord` is meer dan alleen een lijst met 60 lege lijsten. Al die 60 lijsten stellen een X-coördinaat van het spelbord voor. Er zijn 15 rijen op het bord, dus al die 60 lijsten moeten elk 15 tekens bevatten. Regel 45 is een andere `for`-lus die 15 strings van één teken lang gaat toevoegen om de oceaan voor te stellen.

De 'oceaan' is gewoon een aantal willekeurig gekozen `'~'` - en`'`'`-tekens. Als de retourwaarde van `random.randint()` `0` is, voegen we de string `'~'` toe. Anders gebruiken we de string `'`'`. Hierdoor krijgt de oceaan een beetje een stormachtig uiterlijk.

Onthoud dat de variabele `bord` een lijst is met 60 lijsten, die elk 15 strings bevatten. Dus als je naar de string wilt wijzen op coördinaat 26,12, geef je dat aan met `bord[26][12]`, niet met `bord[12][26]`. De X-coördinaat komt eerst, en dan pas de Y-coördinaat.

```
51.     return bord
```

Ten slotte retourneert de functie de waarde in de variabele `bord`.

De willekeurige schatkisten maken

```
 53. def haalWillekeurigeSchatkisten(aantalKisten):
 54.     # een lijst maken met datastructuren voor een schatkist (lijsten met
dubbele items van x,y-coördinaten)
 55.     kisten = []
 56.     for i in range(aantalKisten):
 57.         kisten.append([random.randint(0, 59), random.randint(0, 14)])
 58.     return kisten
```

Het spel bepaalt ook willekeurig waar de verborgen schatkisten worden geplaatst. De schatkisten worden voorgesteld door een lijst met lijsten van twee integers. Deze twee integers zijn de X- en Y-coördinaten van één schatkist.

Bijvoorbeeld: als de datastructuur voor schatkisten `[[2, 2], [2, 4], [10, 0]]` zou zijn, betekent dat dat er drie schatkisten zijn, één op 2,2, de tweede op 2,4 en de derde op 10,0.

De parameter `aantalKisten` vertelt de functie hoeveel schatkisten er moeten worden gemaakt. De `for`-lus op regel 56 itereert '`aantalKisten`' zo vaak en bij elke iteratie voegt regel 57 een lijst met twee willekeurige integers toe. De X-coördinaat kan alles zijn van 0 t/m 59, en de Y-coördinaat kan alles zijn van 0 t/m 14. De expressie `[random.randint(0, 59), random.randint(0, 14)]` die wordt doorgegeven aan de methode append, resulteert in een lijstwaarde als `[2, 2]` of `[2, 4]` of `[10, 0]`. Deze lijstwaarde wordt toegevoegd achteraan in de variabele `schatkisten`.

Vaststellen of een zet geldig is

```
 60. def isGeldigeZet(x, y):
 61.     # retourneert True als de coördinaten op het bord voorkomen, anders False.
 62.     return x >= 0 and x <= 59 and y >= 0 and y <= 14
```

Als de spelers X- en Y-coördinaten typen om aan te geven waar ze een sonar willen laten zakken, typen ze misschien geen geldige coördinaten. Het X-coördinaat moet een getal zijn van 0 t/m 59 en het Y-coördinaat moet een getal zijn van 0 t/m 14.

De functie `isGeldigeZet()` maakt gebruik van een eenvoudige expressie met `and`-operators om ervoor te zorgen dat elk deel van de voorwaarde `True` is. Als ook maar één deel `False` is, is dus de hele voorwaarde `False`. Deze functie retourneert een Boole-waarde.

Een zet doen op het bord

```
 64. def doeZet(bord, kisten, x, y):
 65.     # de datastructuur van het bord veranderen met een sonarteken. Schatkisten
verwijderen
 66.     # van de schatkistenlijsten zodra ze worden gevonden. False retourneren
als dit een ongeldige zet is.
 67.     # Anders de string retourneren van het resultaat van deze zet.
 68.     if not isGeldigeZet(x, y):
 69.         return False
```

Elke keer dat er een sonar wordt neergelaten, wordt het bord op die positie bijgewerkt met een cijfer dat aangeeft op welke afstand de dichtstbijzijnde schatkist ligt. Dus als de speler een zet doet door een X- en Y-coördinaat op te geven, verandert het bord op basis van de posities van de schatkisten.

De functie `doeZet()` krijgt vier parameters mee: de spelborddatastructuur, de schatkistdatastructuur en de X- en Y-coördinaten. Regel 69 retourneert `False` als de meegegeven X- en Y-coördinaten niet voorkomen op het spelbord. Als `isGeldigeZet()` `False` retourneert, is de retourwaarde van `doeZet()` zelf ook `False`.

Anders retourneert `doeZet()` een stringwaarde die aangeeft wat er gebeurde als reactie op de zet:

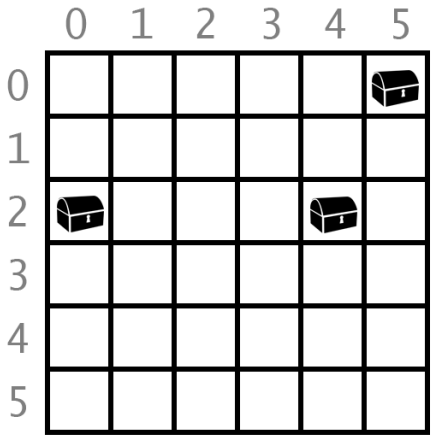
- Als de coördinaten direct op de schatkist stuiten, retourneert `doeZet()` `'Je hebt een gezonken schatkist gevonden!'`.
- Als de coördinaten binnen een afstand van 9 of minder liggen, retourneert `doeZet()` `'Schatkist op een afstand van %s van de sonar.'` (waarbij `%s` wordt vervangen door de afstand in een integer).
- Anders retourneert, `doeZet()` de string `'Sonar heeft niets gevonden. Alle schatkisten buiten bereik.'`.

```
 71.     kleinsteAfstand = 100 # alle kisten zijn dichterbij dan 100.
 72.     for cx, cy in kisten:
 73.         if abs(cx - x) > abs(cy - y):
 74.             afstand = abs(cx - x)
 75.         else:
 76.             afstand = abs(cy - y)
 77.
 78.         if afstand < kleinsteAfstand: # we willen de dichtstbijzijnde
schatkist.
 79.             kleinsteAfstand = afstand
```

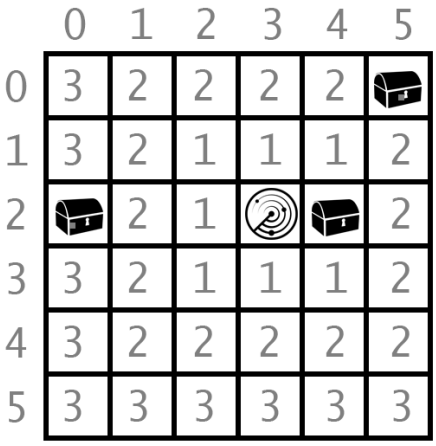
Nu je de coördinaten weet van de plek waar de speler de sonar wil laten zakken en een lijst hebt van de XY-coördinaten voor de schatkisten, heb je een algoritme nodig om te kunnen bepalen welke schatkist het dichtstbij ligt.

Een algoritme om te bepalen welke schatkist het dichtstbij ligt

De `X`- en `Y`-parameters zijn integers (bijvoorbeeld `3` en `2`), en samen geven ze de positie op het bord aan die de speler wil proberen. De variabele `kisten` zal een waarde krijgen zoals `[[5, 0], [0, 2], [4, 2]]`. Die waarde staat voor de locaties van drie schatkisten. Een illustratie hiervan zie je in Afbeelding 13-3. De afstanden vormen 'kringen' rondom de sonar die zich op positie 3,2 bevindt, zoals in Afbeelding 13-4.



Afbeelding 13-3: De schatkisten die de waarden `[[5, 0], [0, 2], [4, 2]]` voorstellen.



Afbeelding 13-4: Het bord gemarkeerd met afstanden vanaf de positie 3,2.

Maar hoe vertaal je dit in code voor het spel? Je hebt een manier nodig om de afstanden in die vierkante kring voor te stellen in een expressie. Nou, de afstand tussen de XY-coördinaten van een schatkist en een sonar zal altijd de grotere waarde zijn van twee waarden. Die twee waarden zijn de absolute waarde van het verschil tussen de twee X-coördinaten en de absolute waarde van het verschil tussen de twee Y-coördinaten.

Dat betekent dat je de X-coördinaat van de sonar en de X-coördinaat van de schatkist van elkaar moet aftrekken en de absolute waarde van dit getal moet nemen. Je doet hetzelfde voor de Y-coördinaat van de sonar en de Y-coördinaat van de schatkist. De waarde die het *grootst* is van deze twee, is de afstand.

Kijk maar: stel dat de X- en Y-coördinaten van de sonar 3 en 2 zijn, zoals in Afbeelding 13-4. De X- en Y-coördinaten van de eerste schatkist (de eerste in de lijst `[[5, 0], [0, 2], [4, 2]]`) zijn `5` en `0`.

1. Bij de X-coördinaten is `3 - 5 -2`, en de absolute waarde van `-2` is `2`.
2. Bij de Y-coördinaten is `2 - 1 1`, en de absolute waarde van `1` is `1`.
3. Als we de twee absolute waarden `2` en `1` vergelijken, is `2` de grotere waarde, dus `2` moet de afstand zijn tussen de sonar en de schatkist op positie 5,1.

We kunnen het bord bekijken in Afbeelding 13-4 en dan zien we dat dit algoritme klopt, omdat de schatkist met positie 5,1, in de kring met het cijfer 2 ligt. Laten we snel even kijken of het ook klopt voor de andere twee kisten.

We zoeken de afstand tussen de sonar op positie 3,2 en de schatkist op positie 0,2:

1. `abs(3 - 0)` is `3`.
2. `abs(2 - 2)` is `0`.
3. `3` is groter dan `0`, dus de afstand tussen de sonar op 3,2 en de schatkist op 0,2 is `3`.

We zoeken de afstand tussen de sonar op positie 3,2 en de laatste schatkist op positie 4,2:

1. `abs(3 - 4)` is `1`.
2. `abs(2 - 2)` is `0`.
3. `1` is groter dan `0`, dus de afstand is `1`.

Als we kijken naar Afbeelding 13-4 zien we dat alle drie de afstanden kloppen. Dit algoritme schijnt te werken. De afstanden van de sonars tot de drie gezonken schatkisten zijn `2`, `3` en`1`. Elke keer dat de speler raadt, wil je de afstand weten van de sonar tot de dichtstbijzijnde van de drie schatkisten. Om dit voor elkaar te krijgen, gebruiken we een variabele genaamd `kleinsteAfstand`. We bekijken de code weer even:

```
 71.     kleinsteAfstand = 100 # alle kisten zijn dichterbij dan 100.
 72.     for cx, cy in kisten:
 73.         if abs(cx - x) > abs(cy - y):
 74.             afstand = abs(cx - x)
 75.         else:
 76.             afstand = abs(cy - y)
 77.
 78.         if afstand < kleinsteAfstand: # we willen de dichtstbijzijnde
schatkist.
 79.             kleinsteAfstand = afstand
```

Op regel 72 gebruiken we het trucje voor meervoudige toewijzing in een `for`-lus. De toewijzingsstatement `ham, kaas = [5, 10]` wijst `5` toe aan `ham` en `10` aan `kaas`.

Aangezien `kisten` een lijst is waar elk item in de lijst zelf weer een lijst is met twee integers, wordt de eerste van deze integers toegewezen aan `cx` en de tweede integer aan `cy`. Dus als `kisten` de waarden `[[5, 0], [0, 2], [4, 2]]` bevat, krijgt `cx` de waarde `5` en `cy` de waarde `0` bij de eerste iteratie door de lus.

Regel 73 bepaalt welke waarde hoger is: de absolute waarde van het verschil tussen de X-coördinaten, of de absolute waarde van het verschil tussen de Y-coördinaten. `abs(cx - x) > abs(cy - y)` zegt hetzelfde, maar wel wat korter! De regels 73-76 wijzen de grotere waarde toe aan de variabele `afstand`.

Dus bij elke iteratie door de `for`-lus, bevat de variabele `afstand` de afstand tussen de schatkist en de sonar. Maar wat je zoekt, is de kleinste afstand naar alle schatkisten. En hier speelt de variabele `kleinsteAfstand` zijn rol. Zodra de variabele `afstand` kleiner is dan `kleinsteAfstand`, wordt de waarde in `afstand` de nieuwe waarde van `kleinsteAfstand`.

Aan het begin van de lus geven we `kleinsteAfstand` dan de onmogelijk hoge waarde van `100` zodat ten minste één van de schatkisten een afstand zal opleveren die in de `kleinsteAfstand` kan worden gestopt. Zodra de `for`-lus klaar is, weet je dat `kleinsteAfstand` de kleinste afstand bevat tussen de sonar en alle schatkisten in het spel.

De methode `remove()` voor lijsten

De methode `remove()` voor lijsten verwijdert het eerste voorkomen van een waarde die overeenkomt met het doorgegeven argument. Typ bijvoorbeeld het volgende in de interactieve shell:

```
>>> x = [42, 5, 10, 42, 15, 42]
>>> x.remove(10)
>>> x
[42, 5, 42, 15, 42]
```

De waarde `10` is verwijderd van de lijst `x`. De methode `remove()` verwijdert alleen het eerste voorkomen van de waarde die je doorgeeft. Als de waarde nog een keer in de lijst voorkomt, wordt hij niet verwijderd. Typ bijvoorbeeld het volgende in de interactieve shell.

```
>>> x = [42, 5, 10, 42, 15, 42]
>>> x.remove(42)
>>> x
[5, 10, 42, 15, 42]
```

Zoals je ziet is alleen de eerste waarde `42` verwijderd, maar de tweede en derde keer dat de waarde voorkomt, blijft de waarde staan. De methode `remove()` levert een foutbericht op als je een waarde wilt verwijderen die niet in de lijst voorkomt:

```
>>> x = [5, 42]
>>> x.remove(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

```
81.     if kleinsteAfstand == 0:
82.         # xy is de locatie van een schatkist!
83.         kisten.remove([x, y])
84.         return 'Je hebt een gezonken schatkist gevonden!'
```

De enige keer dat `kleinsteAfstand` gelijk is aan `0` is wanneer de XY-coördinaten van de sonar hetzelfde zijn als de XY-coördinaten van de schatkist. Dit betekent dat de speler de locatie van de schatkist goed heeft geraden. De lijst met twee integers van deze schatkist wordt dan verwijderd uit de datastructuur `kisten` met behulp van de methode `remove()` voor lijsten. Daarna retourneert de functie de string `'Je hebt een gezonken schatkist gevonden!'`.

```
85.     else:
86.         if kleinsteAfstand < 10:
87.             bord[x][y] = str(kleinsteAfstand)
88.             return 'Schatkist op een afstand van %s van de sonar.' %
(kleinsteAfstand)
89.         else:
90.             bord[x][y] = 'O'
```

```
 91.             return 'Sonar heeft niets gevonden. Alle schatkisten buiten
bereik.'
```

Het else-blok dat op regel 86 begint, wordt uitgevoerd als `kleinsteAfstand` niet `0` was, wat betekent dat de speler de precieze locatie van de schatkist niet heeft geraden. Als de afstand van de sonar minder was dan 10, zorgt regel 87 ervoor dat het bord wordt gemarkeerd met de stringversie van `kleinsteAfstand`. Zo niet, wordt het bord gemarkeerd met een `'0'`.

De zet van de speler halen

```
 94. def zetSpelerInvoeren():
 95.     # We laten de speler zijn zet invoeren. Een lijst met dubbele items van
x,y-coördinaten retourneren.
 96.     print('Waar wil je de volgende sonar laten zakken? (0-59 0-14) (of typ
afsluiten)')
 97.     while True:
 98.         zet = input()
 99.         if zet.lower() == 'afsluiten':
100.             print('Leuk dat je meedeed!')
101.             sys.exit()
```

De functie `zetSpelerInvoeren()` verzamelt de XY-coördinaten van de volgende zet van de speler. De `while`-lus blijft de speler om zijn volgende zet vragen, totdat hij een geldige zet doet. De speler mag ook `'afsluiten'` typen als ze het spel willen afsluiten. In dat geval wordt op regel 101 de functie `sys.exit()` aangeroepen die het programma onmiddellijk beëindigt.

```
103.         zet = zet.split()
104.         if len(zet) == 2 and zet[0].isdigit() and zet[1].isdigit() and
isGeldigeZet(int(zet[0]), int(zet[1])):
105.             return [int(zet[0]), int(zet[1])]
106.         print('Typ een getal van 0 tot 59, een spatie, en dan een getal van 0
tot 14.')
```

Aangenomen dat de speler niet `'afsluiten'` heeft getypt, moet de code controleren of het een geldige zet is. Het moeten twee integers zijn met een spatie ertussen. Regel 103 roept de methode `split()` aan voor de waarde in `zet` en wijst het resultaat daarvan toe aan `zet`.

Als de speler een waarde zou hebben ingetypt als `'1 2 3'`, zou de lijst die wordt geretourneerd door middel van `split()`, `['1', '2', '3']` zijn. Dan zou de expressie `len(zet) == 2` `False` zijn en dus de hele expressie `False` zijn. Python controleert de rest van de expressie niet meer vanwege het principe van kortsluiten bij de operator AND (weet je nog, uit Hoofdstuk 10?).

Als de lengte van de lijst `2` is, zullen de twee waarden op indexpositie `zet[0]` en `zet[1]` staan. Om te checken of die waarden numerieke cijfers zijn (zoals `'2'` of `'17'`), zou je een functie kunnen gebruiken zoals `isAlleenCijfers()` uit Hoofdstuk 11. Maar Python heeft ook een ingebouwde functie die dit doet.

De methode `isdigit()` voor strings retourneert `True` als de string helemaal bestaat uit cijfertekens. Anders retourneert de methode `False`. Probeer het volgende in de interactieve shell te typen:

```
>>> '42'.isdigit()
True
```

```
>>> 'veertig'.isdigit()
False
>>> ''.isdigit()
False
>>> 'hallo'.isdigit()
False
>>> x = '10'
>>> x.isdigit()
True
```

Zowel `zet[0].isdigit()` als `zet[1].isdigit()` moeten `True` zijn om de hele voorwaarde `True` te laten zijn. Het laatste gedeelte van de voorwaarde op regel 104 roept de functie `isGeldigeZet()` aan om te controleren of de XY-coördinaten wel op het bord voorkomen.

Als de hele voorwaarde `True` is, retourneert regel 105 een lijst met als items de twee integers van de XY-coördinaten. Anders gaat de verwerking weer door de lus en wordt de speler weer gevraagd coördinaten op te geven.

De speler vragen of ze nog een keertje willen spelen

```
109. def nogmaalsSpelen():
110.     # Deze functie retourneert True als de speler nog een keer wil spelen,
anders is het False.
111.     print('Wil je nog een keer spelen? (ja of nee)')
112.     return input().lower().startswith('j')
```

De functie `nogmaalsSpelen()` hebben we in eerdere hoofdstukken al gezien.

De spelinstructies tonen aan de speler

```
115. def toonUitleg():
116.     print('''Uitleg:
117. Jij bent de kapitein van de Simon, een schip dat naar schatten zoekt.
118. Je huidige missie is het vinden van drie gezonken schatkisten die zich ergens
119.  in de oceaan bevinden.
120.
121. Typ de coördinaten van het punt in de oceaan waar je een
122. sonar wilt laten zakken De sonar kan meten hoe ver weg de dichtstbijzijnde
schatkist ligt.
123. De d hieronder geeft bijvoorbeeld aan waar de sonar werd uitgezet en de 2's
124. geeft een afstand van 2 vanaf de sonar aan. De 4's geven
125. afstanden van 4 vanaf de sonar aan.
126.
127.     444444444
128.     4       4
129.     4 22222 4
130.     4 2   2 4
131.     4 2 d 2 4
132.     4 2   2 4
133.     4 22222 4
134.     4       4
135.     444444444
136. Druk op enter om door te gaan...''')
137.     input()
```

De functie `toonUitleg()` bestaat gewoon uit een paar `print()`-aanroepen die meerdere regels weergeven. De functie `input()` geeft de speler de mogelijkheid op ENTER te drukken om nog meer tekst weer te geven. Dat is nodig omdat het venster in IDLE een beperkte hoeveelheid tekst tegelijk kan weergeven.

```
139.     print('''Hier is bijvoorbeeld een schatkist (de c) op een afstand van 2
vanaf
140. de sonar (de d):
141.
142.     22222
143.     c   2
144.     2 d 2
145.     2   2
146.     22222
147.
148. Het punt waar de sonar is uitgezet wordt aangegeven met een 2.
149.
150. De schatkisten worden niet verplaatst. Sonars kunnen schatkisten opsporen
151. tot een afstand van 9. Als alle schatkisten buiten het bereik van een sonar
zijn,
152. wordt het punt gemarkeerd met een O
153.
154. Als een sonar wordt neergelaten bovenop een schatkist, heb je
155. de plaats van de schatkist gevonden, en wordt deze opgehaald. De sonar
156. blijft daar dan.
157.
158. Wanneer je een schatkist hebt opgehaald, worden alle sonars bijgewerkt om de
volgende dichtstbijzijnde
159. schatkist op te sporen.
160. Druk op enter om door te gaan...''')
161.     input()
162.     print()
```

Nadat de speler op ENTER heeft gedrukt, gaat de functie verder.

Het begin van het programma

```
165. print('S O N A R !')
166. print()
167. print('Wil je weten hoe je het moet spelen? (ja/nee)')
168. if input().lower().startswith('j'):
169.     toonUitleg()
```

De expressie `input().lower().startswith('j')` vraagt de speler of ze de instructies willen zien en retourneert `True` als de speler een string typte die begon met `'j'` of `'J'`. In dat geval wordt `toonUitleg()` aangeroepen. In het andere geval begint het spel.

```
171. while True:
172.     # spel opstellen
173.     sonars = 16
174.     hetBord = haalNieuwBord()
175.     deSchatkisten = haalWillekeurigeSchatkisten(3)
176.     tekenBord(hetBord)
177.     vorigeZetten = []
```

De `while`-lus op regel 171 is de hoofdlus van het programma. Op de regels 173-177 worden verscheidene variabelen ingesteld en deze worden beschreven in Tabel 13-1.

Tabel 13-1: Variabelen in het hoofdprogramma.

Variabele	Beschrijving
`sonars`	Het aantal sonars (en beurten) dat de speler over heeft.
`hetBord`	De bord-datastructuur die voor dit spel wordt gebruikt.
`deSchatkisten`	De lijst met schatkistdatastructuren. De functie `haalWillekeurigeSchatkisten()` retourneert een lijst met drie schatkisten op willekeurige locaties op het bord.
`vorigeZetten`	Een lijst met alle XY-zetten die de speler in dit spel heeft gedaan.

De spelvoortgang tonen aan de speler

```
179.     while sonars > 0:
180.         # Begin van de beurt:
181.
182.         # sonars/schatkiststatus laten zien
183.         if sonars > 1: extraSsonar = 's'
184.         else: extraSsonar = ''
185.         if len(deSchatkisten) > 1: extraEnSchatkist = 'en'
186.         else: extraEnSchatkist = ''
187.         print('Je hebt %s sonar%s over. Er zijn %s schatkiste%n over.' %
(sonars, extraSsonar, len(deSchatkisten), extraEnSchatkist))
```

De `while`-lus op regel 179 blijft doorgaan zo lang de speler nog sonars over heeft. Regel 187 toont een bericht aan de gebruiker over hoeveel sonars en schatkisten er nog over zijn. Maar daar hebben we een grammaticaal probleempje.

Als er twee of meer sonars over zijn, wil je printen `'2 sonars'`. Maar als er nog maar 1 sonar over is, wil je printen `'1 sonar'`. Je wilt alleen het meervoud van sonar als er meer dan 1 sonar is. Hetzelfde geldt voor `'2 schatkisten'` en `'1 schatkist'`.

Op de regels 183-186 zie je code staan achter de dubbele punt in de `if`- en `else`-statements. Dat vindt Python best. In plaats van een blok code onder de if-statement te schrijven, mag het ook op de rest van de regel. Zo wordt je code wat korter.

De twee variabelen genaamd `extraSsonar` en `extraEnSchatkist` worden ingesteld op `'s'` en `'en'` als er meer dan 1 sonar of schatkist is. Zo niet, zijn dit lege strings. Deze variabelen worden op regel 187 gebruikt.

De zet van de speler halen

```
189.         x, y = zetSpelerInvoeren()
190.         vorigeZetten.append([x, y]) # we moeten alle zetten bijhouden zodat de
sonars kunnen worden bijgewerkt.
191.         zetResultaat = doeZet(hetBord, deSchatkisten, x, y)
192.         if zetResultaat == False:
193.             continue
```

Op regel 189 wordt meervoudige toewijzing gebruikt, aangezien `zetSpelerInvoeren()` een lijst met twee items retourneert. Het eerste item in de geretourneerde lijst wordt toegewezen aan de variabele `x`. Het tweede item wordt toegewezen aan de variabele `y`.

Vervolgens worden ze toegevoegd achteraan in de lijst `vorigeZetten`. Dit betekent dat `vorigeZetten` een lijst is van de XY-coördinaten van elke zet die de speler in dit spel doet. Deze lijst wordt verderop in het programma gebruikt, op regel 198.

De variabelen `x`, `y`, `hetBord` en `deSchatkisten` worden allemaal meegegeven aan de functie `doeZet()`. Deze functie maakt de nodige aanpassingen op het bord voor het plaatsen van een sonar op het bord.

Als `doeZet()` de waarde `False` retourneert, was er een probleem met de `x`- en `y`-waarden die je doorgaf. De statement `continue` stuurt de verwerking terug naar het begin van de `while`-lus op regel 179 om de speler nogmaals om XY-coördinaten te vragen.

Een gezonken schatkist vinden

```
195.         else:
196.             if zetResultaat == 'Je hebt een gezonken schatkist gevonden!':
197.                 # alle sonars op de kaart moeten worden bijgewerkt.
198.                 for x, y in vorigeZetten:
199.                     doeZet(hetBord, deSchatkisten, x, y)
200.             tekenBord(hetBord)
201.             print(zetResultaat)
```

Als `doeZet()` de waarde `False` niet heeft geretourneerd, heeft de functie een string van de resultaten van die zet geretourneerd. Als deze string gelijk is aan `'Je hebt een gezonken schatkist gevonden!'`, moeten alle sonars op het bord worden bijgewerkt om de *volgende* dichtstbijzijnde schatkisten aan te geven. De XY-coördinaten van alle sonars staan in `vorigeZetten`. Door te itereren door `vorigeZetten` op regel 198, kun je al deze XY-coördinaten nog eens doorgeven aan de functie `doeZet()` om de waarden op het bord opnieuw te tekenen.

Omdat het programma hier niet iets nieuws tekent, realiseert de speler zich niet dat het programma alle eerdere zetten opnieuw heeft gedaan. Het lijkt gewoon of het bord zichzelf bijwerkt.

Controleren of de speler heeft gewonnen

```
203.         if len(deSchatkisten) == 0:
204.             print('Je hebt alle gezonken schatkisten gevonden! Gefeliciteerd,
kapitein!')
205.             break
```

Vergeet niet dat de functie `doeZet()` de lijst `deSchatkisten` die je hebt meegegeven, wijzigt. Omdat `deSchatkisten` een lijst is, blijven alle wijzigingen die erin worden aangebracht, behouden nadat de verwerking terugkeert van de functie. `doeZet()` verwijdert items uit `deSchatkisten` wanneer er schatkisten worden gevonden, dus uiteindelijk (als de speler goed blijft raden) zullen alle schatkisten uit de lijst zijn verwijderd. Vergeet niet dat we met 'schatkist' de lijsten met twee items bedoelen van de XY-coördinaten in de lijst `deSchatkisten`.

Wanneer alle schatkisten zijn gevonden op het bord en zijn verwijderd uit de lijst `deSchatkisten`, heeft de lijst `deSchatkisten` een lengte van `0`. Wanneer dat gebeurt, feliciteer je de speler, en wordt een `break`-statement uitgevoerd waarmee we uit deze `while`-lus breken. De verwerking gaat dan verder met regel 209, de eerste regel na het while-blok.

Controleren of de speler heeft verloren

```
207.         sonars -= 1
```

Regel 207 is de laatste regel van de `while`-lus die op regel 179 begon. Verminder de variabele `sonar` met 1 omdat de speler een sonar heeft gebruikt. Als de speler de schatkisten blijft missen, zal het aantal in `sonars` uiteindelijk `0` worden. Na deze regel springt de verwerking terug naar regel 179 om de voorwaarden van de `while`-statement opnieuw te testen (`sonars > 0`).

Als `sonars` `0` is, is de voorwaarde `False` en gaat de verwerking het while-blok uit op regel 209. Maar tot die tijd blijft de voorwaarde `True` en kan de speler blijven raden.

```
209.     if sonars == 0:
210.         print('We hebben geen sonars meer! Nu moeten we terug naar de haven')
211.         print('terwijl er nog schatkisten liggen! Game over.')
212.         print('    De overige schatkisten lagen hier:')
213.         for x, y in deSchatkisten:
214.             print('    %s, %s' % (x, y))
```

Regel 209 is de eerste regel naar de `while`-lus. Wanneer de verwerking hier is aangekomen, is het spel over. Als `sonars` `0` is, weet je dat de speler geen sonars meer over had voordat hij alle schatkisten had gevonden, dus hij heeft verloren.

De regels 210-212 vertellen de speler dat hij heeft verloren. De `for`-lus op regel 213 itereert door de schatkisten die nog in `deSchatkisten` staan en tonen hun locatie aan de speler.

De functie sys.exit()

```
216.     if not nogmaalsSpelen():
217.         sys.exit()
```

Ongeacht of de speler heeft gewonnen of verloren wordt `nogmaalsSpelen()` opnieuw aangeroepen zodat de speler kan aangeven of ze nog eens willen spelen. Zo niet, is het resultaat van de functie `nogmaalsSpelen()` de waarde `False`. De operator `not` op regel 216 draait dit om in `True`, zodat de voorwaarde van de `if`-statement `True` wordt en de functie `sys.exit()` wordt uitgevoerd. Deze functie beëindigt het programma.

Als de speler wel nog een keer wil spelen, gaat de verwerking naar het begin van de `while`-lus op regel 171 en wordt een nieuw spel gestart.

Samenvatting

Weet je nog hoe ons Boter-kaas-en-eierenspel de vakjes op het bord nummerde van 1 tot 9? Zo'n soort coördinatenstelsel was wel oké voor een bord met minder dan tien vakjes. Maar het Sonar-bord heeft 900 vakjes! Het Cartesische coördinatenstelsel waarover we hebben geleerd in het laatste hoofdstuk maakt al deze vakjes beheersbaar, vooral omdat ons spel ook nog de afstand tussen twee punten op het bord moest berekenen.

Locaties in spellen waarin een Cartesisch coördinatenstelsel wordt gebruikt, kunnen worden opgeslagen in een lijst met lijsten, zodat de eerste index de X-coördinaat is en de tweede index de Y-coördinaat. Als we dan naar de coördinaten willen verwijzen, doen we dat met `bord[x][y]`.

Deze datastructuren (zoals gebruikt voor de oceaan en de locaties van de schatkisten) maken het mogelijk om ingewikkelde zaken als data voor te stellen en in je programma's ben je dan voornamelijk bezig met het veranderen van deze datastructuren.

In het volgende hoofdstuk gaan we letters als nummers weergeven met behulp van hun ASCII-nummers. (Een computer kent een letter eigenlijk alleen als een getal. De letter ë is bijvoorbeeld ASCII-nummer 137 en dat kun je typen op je toetsenblok als je ALT ingedrukt houdt.) Door tekst weer te geven als getallen, kunnen we met letters 'rekenen' zodat we geheime berichten kunnen coderen of decoderen.