Hoofdstuk 7

Iteraties

Computers raken niet verveeld. Als een computer een taak honderdduizenden malen moet herhalen, protesteert hij niet. Mensen daarentegen houden niet van teveel herhaling. Daarom moeten herhalende taken aan een computer worden overgelaten. Alle programmeertalen ondersteunen herhalingen. De klasse programmeerconstructies die herhalingen mogelijk maken heten "iteraties." Een veelgebruikte term is "loops" (Engels, spreek uit: "loeps" – dit woord kun je netjes vertalen als "lussen," maar dat zeggen programmeurs nooit).

Dit hoofdstuk legt uit wat je moet weten over loops in Python. Als programmeren helemaal nieuw voor je is, zul je wellicht het gevoel krijgen dat loops een lastig onderwerp zijn. Als dat zo is, neem dan de tijd voor dit hoofdstuk, en werk eraan totdat je zeker weet dat je alles snapt. Loops zijn zo'n basaal programmeerconcept dat je ze goed moet begrijpen. Elk hoofdstuk dat hierna komt maakt gebruik van loops.

7.1 `while` loop

Stel dat je de gebruiker moet vragen om vijf getallen, ze moet optellen, en dan het totaal moet laten zien. Met het materiaal dat tot nu toe behandeld is, zou je dat als volgt coderen:

```
from pcinput import getInteger

num1 = getInteger( "Nummer 1: " )
num2 = getInteger( "Nummer 2: " )
num3 = getInteger( "Nummer 3: " )
num4 = getInteger( "Nummer 4: " )
num5 = getInteger( "Nummer 5: " )

print( "Totaal is", num1 + num2 + num3 + num4 + num5 )
```

Maar wat als je om 500 nummers moet vragen? Ga je die regel code waar om een getal wordt gevraagd 500 keer kopiëren? Dat moet toch op een gemakkelijkere manier kunnen?

Natuurlijk kan dat gemakkelijker. Je kunt hiervoor een loop gebruiken.

De eerste loop die ik bespreek is de **while** loop. Een **while** statement lijkt heel veel op een **if** statement. De syntax is:

```
while <boolean expressie>:
    <acties>
```

Net als bij een **if** statement, test een **while** statement een boolean expression, en als de expressie **True** oplevert, wordt het blok code onder de **while** uitgevoerd. Echter, in tegenstelling tot een **if** statement, gaat Python, wanneer het blok code uitgevoerd is, terug naar de boolean expressie naast de **while**, en test die opnieuw. Als de expressie nog steeds **True** oplevert, dan wordt het blok code nogmaals uitgevoerd. En als het is uitgevoerd, keert Python wederom terug naar de boolean expressie. Dit wordt steeds herhaald, totdat de boolean expressie **False** oplevert. Pas op dat moment slaat Python het blok code onder de **while** over en gaat eronder verder.

Merk op dat als de boolean expressie meteen **False** oplevert de eerste keer dat Python hem ziet, dan wordt het blok code onder de **while** onmiddellijk overgeslagen, net zoals gebeurt bij een **if** statement.

7.1.1 `while` loop, eerste voorbeeld

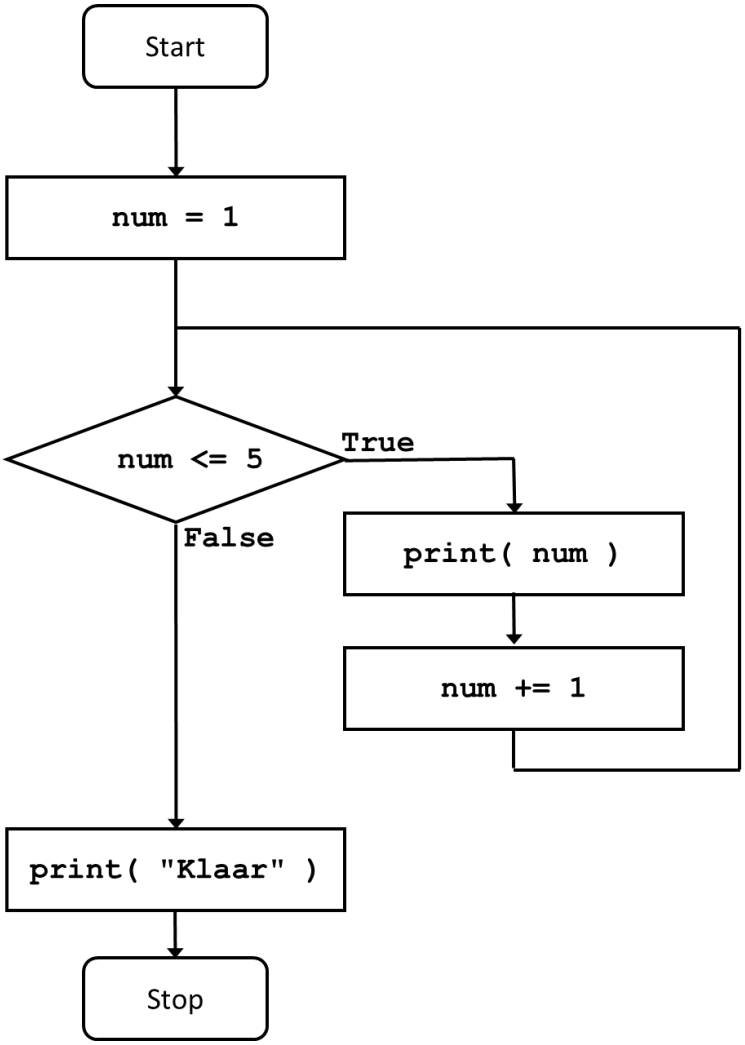
Ik neem een eenvoudig voorbeeld: het printen van de nummers 1 tot en met 5. Met een while loop kun je dat als volgt doen:

listing0701.py

```
num = 1
while num <= 5:
    print( num )
    num += 1
print( "Klaar" )
```

Deze code wordt ook weergegeven in het stroomdiagram in afbeelding 7.1.

Het is van essentieel belang dat je deze code snapt, dus ik neem hem stap voor stap door.



Afb. 7.1: Stroomdiagram dat een `while` loop weergeeft.

De eerste regel initialiseert een variabele `num`. Dit is de variabele die de code gaat afdrukken, dus hij wordt geïnitialiseerd op 1, omdat 1 de eerste waarde is die afgedrukt moet worden.

Dan start de **`while`** loop. De boolean expressie zegt `num <= 5`. Omdat `num` 1 is, en 1 kleiner is dan 5, levert de expressie **`True`**. Dus wordt het blok code onder de **`while`** uitgevoerd.

De eerste regel van het blok code print de waarde van `num`, dus 1. De tweede regel telt 1 op bij de waarde van `num`, zodat `num` nu 2 is. Python gaat daarna terug naar de boolean expressie. De laatste regel van het programma, het printen van het woord "Klaar," wordt dus (nog) niet uitgevoerd omdat hij niet in het code blok van de loop zit, en de loop nog niet afgelopen is.

Omdat `num` 2 is, evalueert de boolean expressie nog steeds als **`True`**. Het blok code wordt dus nogmaals uitgevoerd. 2 wordt afgedrukt, `num` krijgt waarde 3, en de code keert terug bij de boolean expressie.

Omdat `num` 3 is, evalueert de boolean expressie nog steeds als **`True`**. Het blok code wordt dus nogmaals uitgevoerd. 3 wordt afgedrukt, `num` krijgt waarde 4, en de code keert terug bij de boolean expressie.

Omdat `num` 4 is, evalueert de boolean expressie nog steeds als **`True`**. Het blok code wordt dus nogmaals uitgevoerd. 4 wordt afgedrukt, `num` krijgt waarde 5, en de code keert terug bij de boolean expressie.

Omdat `num` 5 is, evalueert de boolean expressie nog steeds als **`True`**. Het blok code wordt dus nogmaals uitgevoerd. 5 wordt afgedrukt, `num` krijgt waarde 6, en de code keert terug bij de boolean expressie.

Omdat `num` 6 is, evalueert de boolean expressie als **`False`** (aangezien 6 niet kleiner dan of gelijk aan 5 is). Het blok code wordt overgeslagen, en Python gaat verder met de eerste regel na het blok code. Dat is de laatste regel van het programma. Het woord "Klaar" wordt afgedrukt, en het programma eindigt.

Opgave Wijzig de code hierboven zodat de getallen 1, 3, 5, 7, en 9 afgedrukt worden.

7.1.2 `while` loop, tweede voorbeeld

Als je het eerste voorbeeld begrijpt, begrijp je waarschijnlijk ook hoe je de gebruiker kunt vragen om vijf getallen, en dan de som van de vijf te berekenen:

listing0702.py

```
from pcinput import getInteger

totaal = 0
teller = 0
while teller < 5:
    totaal += getInteger( "Geef een nummer: " )
    teller += 1

print( "Totaal is", totaal )
```

Bestudeer bovenstaande code. Er worden twee variabelen gebruikt. `totaal` wordt gebruikt om de vijf getallen in op te tellen. `totaal` begint op nul, omdat bij de start van het programma nog geen getallen ingegeven zijn, dus het totaal is nul. `teller` wordt gebruikt om te tellen hoe vaak de loop doorlopen is. Omdat de loop vijf keer uitgevoerd moet worden, start `teller` op nul en de boolean expressie voor de loop test of `teller` kleiner is dan 5. Dus teller moet iedere keer dat de loop doorlopen wordt met 1 verhoogd worden, zodat na vijf keer doorlopen de boolean expressie **`False`** is .

Je vraagt je misschien af waar ik `teller` liet starten bij 0 en de boolean expressie liet testen of `teller < 5`. Waarom begon ik niet bij `teller = 1` en heb ik niet `teller <= 5` getest? De reden is gewoonte: programmeurs zijn gewend om indices bij 0 te laten beginnen en als ze tellen, te tellen "tot maar niet inclusief." Als je verder gaat met programmeren zul je ontdekken dat de meeste code deze gewoonte naleeft. De meeste standaard programmeerconstructies die indices gebruiken, doen het ook zo. Mijn advies is daarom dat je hieraan gewend raakt, want dat zal code gemakkelijker leesbaar maken.

Opmerking: De variabele `teller` is wat programmeurs een "wegwerp variabele" noemen. Het enige doel van deze variabele is te tellen hoe vaak de loop doorlopen is. De variabele heeft geen echte betekenis voor de loop, in de loop, of nadat de loop afgelopen is. Programmeurs kiezen vaak één-letter namen voor dit soort variabelen, meestal `i` of `j` (ik neem aan dat `i` ooit begonnen is als afkorting voor "integer," en `j` gekozen is omdat het na `i` komt). In het voorbeeld heb ik voor de duidelijkheid de naam `teller` gekozen, maar een `i` was acceptabel geweest.

Opgave Wijzig de code hierboven zodat niet alleen het totaal maar ook het gemiddelde wordt afgedrukt.

Opgave De eerste voorbeeldcode in dit hoofdstuk vroeg de gebruiker om vijf getallen in te geven. In dat voorbeeld werd steeds de prompt "Geef nummer `x`: " gebruikt, waarbij `x` een cijfer is. Kun je de code hierboven, waarin een loop gebruikt wordt, zo veranderen dat ook steeds een veranderende prompt wordt gebruikt voor de getallen?

7.1.3 De `while` loop onder controle van de gebruiker

Stel je voor dat je in het tweede voorbeeld de gebruiker niet wilt beperken tot slechts vijf getallen. Je wil de gebruiker zoveel getallen laten ingeven als hij wil, inclusief helemaal geen getallen. Dat betekent dat je niet weet hoe vaak de loop doorlopen moet worden. In plaats daarvan bepaalt de gebruiker hoe vaak de loop doorlopen wordt. Je moet dus de gebruiker de mogelijkheid geven om aan te geven dat hij klaar is met getallen ingeven.

De code hieronder laat zien hoe je een **`while`** loop kunt opzetten die de gebruiker zoveel getallen laat ingeven als gewenst. De gebruiker stopt met het ingeven van getallen door een nul in te geven. Het totaal wordt afgedrukt wanneer de loop beëindigd is.

listing0703.py

```
from pcinput import getInteger

num = -1
totaal = 0
while num != 0:
    num = getInteger( "Geef een nummer: " )
    totaal += num
print( "Totaal is", totaal )
```

Deze code functioneert, maar er zitten minimaal twee lelijke dingen in. Op de eerste plaats wordt `num` geïnitialiseerd op -1. De -1 zelf is betekenisloos, ik moest gewoon een waarde hebben die ervoor zou zorgen dat de loop minstens één keer uitgevoerd zou worden. Ten tweede stopt de loop niet meteen als de gebruiker nul ingeeft; in plaats daarvan wordt de ingave van de gebruiker opgeteld bij `totaal`. Omdat die ingave nul is, wijzigt `totaal` niet, maar als ik had gesteld dat de gebruiker de loop eindigt door bijvoorbeeld een negatief getal in te geven, dan zou `totaal` daarna de verkeerde waarde bevatten.

Vanwege deze lelijke kanten aan de code, prefereren sommige programmeurs om het als volgt te schrijven:

listing0704.py

```
from pcinput import getInteger

num = getInteger( "Geef een nummer: " )
totaal = 0
while num != 0:
    totaal += num
    num = getInteger( "Geef een nummer: " )
print( "Totaal is", totaal )
```

Dit lost de twee hierboven genoemde lelijke zaken op, maar introduceert iets anders lelijks, namelijk de herhaling van de `getInteger()` functie. Hoe je dat kunt oplossen volgt later in dit hoofdstuk. Op dit moment hoef je alleen te snappen hoe de **`while`** loop werkt.

Opgave Maak een loop die de gebruiker een aantal getallen laat ingeven, totdat hij nul ingeeft, en dan het totaal en het gemiddelde afdrukt. Vergeet niet te testen met nul getallen ingeven, en met een aantal keer hetzelfde getal ingeven.

7.1.4 Eindeloze loops

De code hieronder begint bij nummer 1, en telt nummers op, totdat het een nummer tegenkomt dat, als het gekwadrateerd wordt, deelbaar is door 1000. De loop bevat een fout. Kijk of je de fout weet te vinden (zonder de code uit te voeren!).

```
nummer = 1
totaal = 0
while (nummer * nummer) % 1000 != 0:
    totaal += nummer
print( "Totaal is", totaal )
```

De titel boven deze paragraaf gaf het antwoord natuurlijk al weg: deze code bevat een loop die nooit eindigt. Als je hem uitvoert, lijkt het alsof het programma "hangt," dus wel draait maar niks doet. Maar het doet niet niks, het is zelfs hoogst actief, maar het is bezig een nooit-eindigende optelling uit te voeren. `nummer` begint bij 1, maar wordt in de loop niet gewijzigd. Daarom wordt de boolean expressie nooit **`False`**. Dit is een "eindeloze loop." Dit soort loops zijn het grootste gevaar als je **`while`** loops bouwt.

Als je deze code uitvoert in IDLE, kun je de uitvoering afbreken door `Ctrl-C` te drukken. Andere editors hebben menu opties waarmee je de uitvoering van code kunt onderbreken. In gebruikersonvriendelijke omgevingen moet je soms het proces waarin Python draait via een systeem commando of systeem programma afbreken.

Omdat iedere programmeur zo nu en dan per ongeluk een eindeloze loop schrijft, is het een goed idee als je, wanneer je begint met het schrijven van een **`while`** loop, onmiddellijk een regel code toevoegt die een wijziging maakt in hetgeen getest wordt in de boolean expressie, zodat je dat niet vergeet. Bijvoorbeeld, als je schrijft **`while`** `i < 10`, dan schrijf je er meteen de regel `i += 1` onder, en daarna begin je de rest van de code tussen deze twee regels toe te voegen.

Opgave Verbeter de code hierboven zodat het geen eindeloze loop meer is.

7.1.5 `while` loop oefeningen

Je moet nu oefenen met een paar eenvoudige **`while`** loops.

Opgave Schrijf code die aftelt. Er wordt begonnen met een specifiek nummer, bijvoorbeeld 10. Dan telt de code af naar nul, waarbij ieder nummer geprint wordt (10, 9, 8, ...). Nul wordt niet geprint, maar in plaats daarvan drukt het programma de tekst "Start!" af.

Opgave De faculteit van een positief geheel getal is gelijk aan dat getal, vermenigvuldigd met alle positieve gehele getallen die kleiner zijn (exclusief nul). Je schrijft de faculteit als het getal met een uitroepteken erachter. Bijvoorbeeld, de faculteit van 5 is $5! = 5 * 4 * 3 * 2 * 1 = 120$. Schrijf code die de faculteit van een getal berekent. Test het programma niet met getallen die hoog zijn, want de faculteit stijgt exponentieel (het is meer dan genoeg om tot 10! te testen). Hint: Om dit met een **`while`** loop te doen, moet je twee variabelen gebruiken: één die aan het einde van de loop het antwoord bevat, en één die de huidige factor bevat. In de loop vermenigvuldig je het antwoord met de factor, alvorens 1 van de factor af te trekken. Initialiseer deze variabelen met de juiste waardes voor de loop.

7.2 `for` loop

Een alternatieve manier om loops te implementeren is via de **`for`** loop. **`for`** loops zijn gemakkelijker en veiliger te gebruiken dan **`while`** loops, maar kunnen niet voor alle iteratie problemen gebruikt worden. **`while`** loops bieden een generieke oplossing voor loops. Met andere woorden, alles wat een **`for`** loop kan doen, kan een **`while`** loop ook doen, maar niet andersom.

De syntax voor de **`for`** loop is:

```
for <variabele> in <collectie>:
    <acties>
```

A **`for`** loop krijgt een collectie van items, en verwerkt deze items één voor één in de volgorde waarin ze worden aangeboden. Iedere cyclus door de loop verwerkt één item door het in de variabele te stoppen die naast de **`for`** staat. Die variabele kan dan gebruikt worden in het blok code van de loop. De variabele hoeft niet te bestaan voordat de **`for`** loop bezocht wordt. Als de variabele al bestaat, wordt hij overschreven. Als hij nog niet bestaat, wordt hij aangemaakt. Het is overigens een echte variabele, in de zin dat hij nog steeds bestaat als de loop afgelopen is. Na afloop van de loop bevat hij het laatste item dat verwerkt is.

Op dit punt vraag je je wellicht af wat een "collectie" is. Er zijn verschillende soorten collecties in Python, en ik zal er een paar introduceren in dit hoofdstuk. In latere hoofdstukken volgen er meer.

7.2.1 `for` loop met strings

De enige collectie die in de voorgaande hoofdstukken besproken is, is de string. Een string is een collectie van tekens, bijvoorbeeld, de string `"banaan"` is een collectie van de tekens `"b"`, `"a"`, `"n"`, `"a"`, `"a"`, en `"n"`, in die specifieke volgorde. De volgende code verwerkt ieder van de tekens van deze collectie:

```
for letter in "banaan":
    print( letter )
print( "Klaar" )
```

Hoewel deze code vrij triviaal is, zal ik hem voor de duidelijkheid stap voor stap bespreken (ik heb geen stroomdiagram gemaakt, want dat is lastig voor **for** loops).

Als de **for** loop wordt aangetroffen, neemt Python de collectie (de string `"banaan"` in dit geval) en maakt er een zogeheten "iterabele" van. Wat dat precies is komt pas in hoofdstuk 23 aan de orde, maar vooralsnog mag je aannemen dat het een lijst is van alle tekens in de string, in de volgorde dat ze in de string staan. Python neemt dan de eerste van deze tekens, en stopt hem in de variabele `letter`. Daarna voert Python het blok code onder de **for** uit.

Het blok code bevat maar één regel, namelijk het printen van `letter`. Het programma print dus de letter "b," en keert dan terug bij de **for**. Python neemt dan het volgende teken, namelijk de eerste "a," en voert wederom het blok code uit, maar nu met `"a"` als waarde in `letter`. Dit proces wordt herhaald voor ieder van de tekens. Nadat alle tekens verwerkt zijn, eindigt de **for** loop. Python voert dan nog de laatste regel van het programma uit, het printen van het woord "Klaar."

Om volledig helder te zijn: In de **for** loop hoef je niet ergens expliciet een variabele te verhogen of zo, of zelf iets te schrijven dat het volgende teken van de string pakt. De **for** loop handelt dat automatisch af: iedere keer dat de loop terugkeert bij de regel met de **for**, wordt het volgende item uit de collectie gepakt.

7.2.2 `for` loop met een variabele collectie

In de code hierboven wordt de string `"banaan"` gebruikt als collectie, maar er had ook een variabele gebruikt kunnen worden die een string bevat. Bijvoorbeeld, de volgende code is gelijk aan de vorige:

```
fruit = "banaan"
for letter in fruit:
    print( letter )
print( "Klaar" )
```

Je kunt je afvragen of dat niet gevaarlijk is. Wat gebeurt er als de programmeur iets in het blok code van de loop schrijft dat de inhoud van de variabele `fruit` wijzigt? Je kunt het effect hiervan testen met de volgende code:

listing0705.py

```
fruit = "banaan"
for letter in fruit:
    print( letter )
    if letter == "n":
        fruit = "mango"
print( "Klaar" )
```

Als je deze code uitvoert, merk je dat het wijzigen van de inhoud van de variabele `fruit` in de loop geen effect heeft op het proces. De serie tekens die de loop verwerkt wordt slechts eenmalig bepaald, namelijk de eerste keer dat Python de loop betreedt. Het wijzigen van `fruit` in `"mango"` gedurende het proces waarbij de loop de tekens van `"banaan"` verwerkt, laat de loop niet ophouden met het verwerken `"banaan"`. Het is een prachtige eigenschap van **`for`** loops dat ze gegarandeerd eindigen. **`for`** loops kunnen niet eindeloos zijn![6]

Wat bovenstaande code ook illustreert is dat het mogelijk is om een conditie in het blok code van een loop te stoppen. Wellicht verbaast je dat niet, en het hoeft je ook niet te verbazen, want de syntactische definitie van loops legt geen beperkingen op aan wat de acties zijn die een loop mag uitvoeren. Dus zulke acties mogen condities zijn. Het mogen zelfs loops zijn (meer daarover volgt later in dit hoofdstuk).

7.2.3 `for` loop met een getallenreeks

Python heeft een functie **`range()`** die het mogelijk maakt een serie opeenvolgende getallen te genereren. **`range()`** wordt vaak gebruikt in combinatie met een **`for`** loop, bijvoorbeeld om een loop te maken die een specifiek aantal malen wordt uitgevoerd. De eenvoudigste manier om **`range()`** aan te roepen is met één parameter, namelijk een integer. De functie genereert dan een opeenvolgende lijst van integers, beginnend bij nul, tot aan maar niet inclusief de parameter.

```
for x in range( 10 ):
    print( x )
```

`range()` kan meer dan één parameter meekrijgen. Als je er twee meegeeft, dan is de eerste het startgetal (de default is nul), en de tweede het eindgetal. Het eerste getal zit in de gegenereerde serie, het tweede niet. Als je drie getallen meegeeft, zijn de eerste twee als direct hiervoor aangegeven, en is de derde een stapgrootte, dat wil zeggen, de afstand tussen de gegenereerde getallen. Default stapgrootte is 1. Als je wilt aftellen dan is dat mogelijk: je geeft dan een negatieve stapgrootte. Je moet dan wel ervoor zorgen dat het startgetal groter is dan het eindgetal.

```
for x in range( 1, 11, 2 ):
    print( x )
```

Opgave Wijzig in bovenstaande code de drie parameters een paar keer, om het effect van deze wijzigingen te bestuderen. Ga door totdat je de **`range()`** functie begrijpt.

Opgave Gebruik een **`for`** loop en een **`range()`** functie om veelvouden van 3 af te drukken, beginnend bij 21, aftellend tot 3, in slechts twee regels code.

7.2.4 `for` loop met een handmatige collectie

Als je een serie items hebt in een collectie die je "handmatig" hebt samengesteld, kun je die middels een **`for`** loop verwerken als je de items tussen haakjes zet. Een serie items tussen

[6] Helaas zal ik deze uitspraak moeten aanpassen in hoofdstuk 23, maar je hebt erg geavanceerde kennis van Python nodig om een eindeloze `for` loop te maken – je mag er op dit moment van uitgaan (en in de praktijk zal dit ook altijd zo zijn) dat `for` loops gegarandeerd eindigen.

haakjes is een "tuple." Tuples worden in hoofdstuk 11 uitgebreid besproken.

```
for x in ( 10, 100, 1000, 10000 ):
    print( x )
```

Of:

```
for x in ( "appel", "peer", "druif", "banaan", "mango", "kers" ):
    print( x )
```

Een collectie die je zo samenstelt mag zelfs uit verschillende data types bestaan.

7.2.5 `for` loop oefeningen

Om grip te krijgen op het gebruik van **`for`** loops, zijn hier een paar oefeningen:

Opgave Je hebt al code gecreëerd voor een **`while`** loop waarin om vijf getallen wordt gevraagd en het totaal getoond wordt. Doe dat nu met een **`for`** loop.

Opgave Je hebt ook code gecreëerd voor een **`while`** loop die aftelt tot nul, en dan "Start!" print. Doe dat nu met een **`for`** loop.

Opgave Ik ga geen oefening geven waarin je met een **`for`** loop de gebruiker vraagt om getallen totdat de gebruiker nul ingeeft. Waarom niet?

7.3 Loop controle

Er zijn drie statements die je controle geven over de wijze waarop een loop uitgevoerd wordt. Dit zijn **`else`**, **`break`**, en **`continue`**. Ze werken met zowel **`while`** als **`for`** loops.

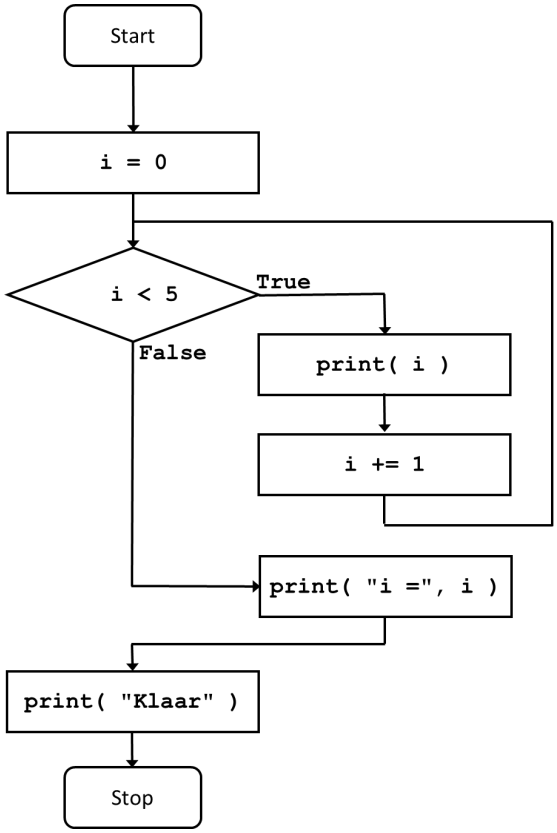
7.3.1 `else`

Net als bij een **`if`** statement, kun je aan het einde van een loop een **`else`** statement toevoegen. Het blok code bij de **`else`** wordt uitgevoerd wanneer de loop eindigt, dus wanneer de boolean expressie voor de **`while`** loop **`False`** is, of bij de **`for`** loop wanneer het laatste item verwerkt is.

Hier is een voorbeeld van **`else`** bij de **`while`** loop:

listing0706.py

```
i = 0
while i < 5:
    print( i )
    i += 1
else:
    print( "De loop eindigt, i is nu", i )
print( "Klaar" )
```



Afb. 7.2: Stroomdiagram van een `while` loop met een `else`.

Deze code is equivalent aan het stroomdiagram in afbeelding 7.2. Het stroomdiagram geeft je wellicht het idee dat het niet zinvol is een **`else`** te gebruiken, maar het kan heel nuttig zijn in combinatie met een **`break`** (die ik hierna bespreek).

Hier is een voorbeeld van **`else`** bij de **`for`** loop:

listing0707.py

```
for fruit in ( "appel", "mango", "aardbei" ):
    print( fruit )
else:
    print( "De loop eindigt, fruit is nu", fruit )
print( "Klaar" )
```

Merk op dat nadat de **`while`** loop hierboven geëindigd is, de waarde van `i` 5 is. De waarde van `fruit` na de **`for`** loop hierboven is het laatste item verwerkt is, dus `"aardbei"`.

7.3.2 break

Het **`break`** statement maakt het mogelijk een loop voortijdig af te breken. Als Python een **`break`** tegenkomt, stopt het met het verwerken van de loop, en keert niet terug bij de eerste

regel van de loop. In plaats daarvan gaat de verwerking verder met de eerste regel code na de loop.

Om te zien wat het nut daarvan is, volgt hier een interessante oefening. Ik zoek een getal dat start met een 1, en als je die 1 verplaatst naar het einde van het getal, dan is het resultaat een getal dat drie keer zo groot is als het originele getal. Bijvoorbeeld, als ik het getal 1867 neem, en ik verplaats de 1 van de start naar het einde, dan krijg ik 8671. Als 8671 drie keer 1867 zou zijn, dan is dat het antwoord dat ik zoek. Maar dat is niet zo, dus 1867 is niet correct. De code om dit op te lossen is vrij kort, en geeft het laagste getal dat het probleem oplost:

listing0708.py

```
i = 1
while i <= 1000000:
    num1 = int( "1" + str( i ) )
    num2 = int( str( i ) + "1" )
    if num2 == 3 * num1:
        print( num2, "is drie keer", num1 )
        break
    i += 1
else:
    print( "Geen antwoord gevonden" )
```

Deze code is weergegeven in het stroomdiagram in afbeelding 7.3.

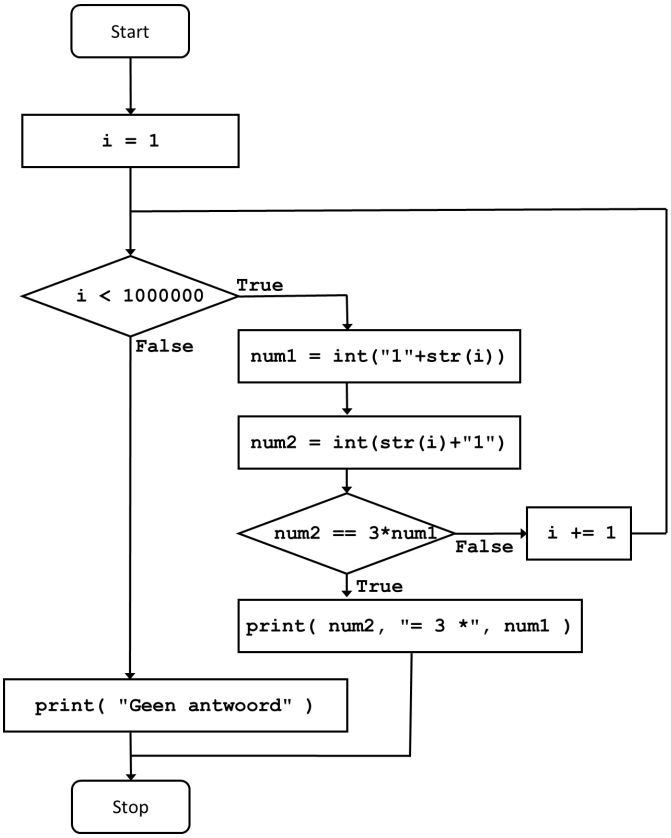
In dit voorbeeld zie je een goed gebruik van **break**. Omdat ik geen idee heb van hoe groot het nummer is dat ik zoek, controleer ik er gewoon een heleboel. Ik laat `i` oplopen tot 1000000. Ik weet natuurlijk niet of ik het antwoord vind voordat `i` 1000000 heeft bereikt, maar ik moet ergens een grens stellen, want misschien bestaat er wel helemaal geen getal dat aan de eis voldoet, en ik wil geen eindeloze loop bouwen. Maar gedurende het testen van getallen kan ik een onverwacht een antwoord vinden, en als dat gebeurt, dan "break" ik uit de loop, want het heeft dan geen zin meer om nog meer getallen te testen.

Het punt is dat ik een maximum van 1000000 heb gekozen niet omdat ik weet dat ik een miljoen getallen moet doorzoeken. Ik heb geen idee hoe vaak ik door de loop moet. Ik weet alleen dat als ik een antwoord vind, ik klaar ben en de loop verder kan afbreken. Dat is precies de bedoeling van de **break**.

Als ik een beetje mijn best doe kan ik er best voor zorgen dat de boolean expressie de vergelijking voor me doet, via iets als `while i < 1000000 and num1 != 3 * num2`. Dit wordt wat ingewikkeld, en ik moet ook `num1` en `num2` waardes geven voordat de loop start. Het is in principe altijd mogelijk om het gebruik van **break** te vermijden. Maar een **break** kan code beter leesbaar maken, zoals het doet in dit geval.

Een **break** kan niet gebruikt worden buiten een loop. **break**s zijn alleen gedefinieerd voor loops. (Ik zie vaak studenten proberen **break** statements te gebruiken in condities die niet in een loop zitten, en dan vreemd opkijken als ze een runtime error krijgen.)

Let op: als een **break** wordt uitgevoerd bij een loop die ook een **else** heeft, dan wordt de code bij de **else** *niet* uitgevoerd. Ik maak daarvan goed gebruik bij de code hierboven: de tekst die aangeeft dat geen antwoord gevonden is, wordt alleen afgedrukt als er niet met een **break** uit de loop wordt gesprongen.



Afb. 7.3: Stroomdiagram van een `while` loop met een `break`.

De volgende code controleert een cijferlijst van een student. Als alle cijfers 5.5 of hoger zijn, dan is de student geslaagd. Maar als er één of meer cijfers lager dan 5.5 zijn, dan is de student gezakt. De cijferlijst wordt verwerkt door een **`for`** loop.

listing0709.py

```
for cijfer in ( 8, 7.5, 9, 6, 6, 6, 5.5, 7, 5, 8, 7, 7.5 ):
    if cijfer < 5.5:
        print( "De student zakt!" )
        break
else:
    print( "De student slaagt!" )
```

Opgave Voer de code hierboven uit en zie dat de student zakt. Verwijder dan de 5 van de cijferlijst en zie dat de student nu slaagt. Bestudeer de code totdat je hem snapt.

7.3.3 `continue`

Als het statement **`continue`** in een blok code bij een loop wordt aangetroffen, wordt onmiddellijk het uitvoeren van de huidige cyclus in de loop beëindigd, en wordt teruggekeerd

naar de eerste regel van de loop. Voor een **while** loop betekent dat dat de boolean expressie opnieuw geëvalueerd wordt, en voor een **for** loop betekent het dat het volgende item van de collectie genomen en verwerkt wordt.

De volgende code drukt alle getallen tussen 1 en 100 af die niet door 2 of 3 gedeeld kunnen worden, en die niet eindigen op een 7 of 9.

listing0710.py

```
num = 0
while num < 100:
    num += 1
    if num%2 == 0:
        continue
    if num%3 == 0:
        continue
    if num%10 == 7:
        continue
    if num%10 == 9:
        continue
    print( num )
```

Deze code is ook weergegeven in het stroomdiagram in afbeelding 7.4.

Ik weet niet wat het nut van deze lijst is, maar in ieder geval helpt **continue** om de code te schrijven. In plaats van **continue** te gebruiken was het ook mogelijk geweest een grote boolean expressie te schrijven die bepaalt of een getal afgedrukt moet worden, maar dat zou snel onleesbaar worden. Maar, net zoals geldt voor **break** statements, **continue** statements kunnen altijd vermeden worden als je dat echt wilt. Ze helpen echter om code begrijpbaar te houden.

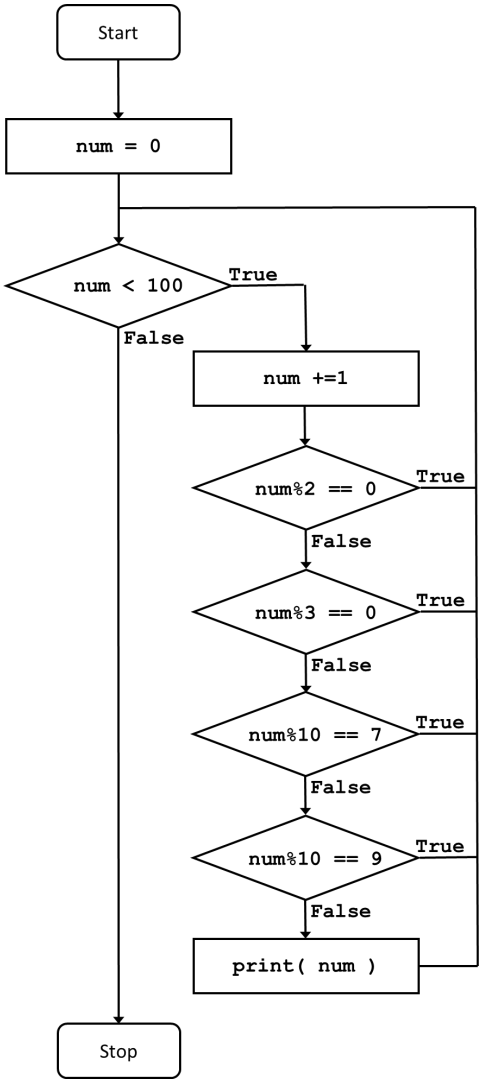
Net als **break** statements, kun je **continue** statements alleen gebruiken in loops.

Wees heel, heel voorzichtig met het gebruik van **continue** in een **while** loop. De meeste **while** loops gebruiken een getal dat het aantal cycli door de loop inperkt. Gewoonlijk wordt een dergelijk getal verhoogd als laatste regel van het blok code bij de loop. Een **continue** statement dat wordt uitgevoerd voordat die laatste regel bereikt is, keert terug bij de boolean expressie zonder dan het getal verhoogd is. Dat kan makkelijk leiden tot een eindeloze loop. Bijvoorbeeld:

```
i = 0
while i < 10:
    if i == 5:
        continue
    i += 1
```

is een eindeloze loop!

Opgave Schrijf een programma dat een reeks getallen doorloopt via een **for** loop. Als er een nul wordt aangetroffen in de lijst getallen, dan moet het programma onmiddellijk eindigen, en alleen het woord "Klaar" afdrukken (gebruik een **break** om dit te implementeren). Negatieve getallen moeten overgeslagen worden (gebruik een **continue** om



Afb. 7.4: Stroomdiagram van een `while` loop met meerdere `continue`s.

dit te implementeren; ik weet dat het ook kan met een conditie, maar ik wil dat je oefent met **`continue`**). Als er geen nul in de reeks getallen staat, moet het programma de som van alle positieve getallen afdrukken (doe dit met een **`else`**). Druk altijd "Klaar" af als het programma eindigt. Test het programma met de reeks `( 12, 4, 3, 33, -2, -5, 7, 0, 22, 4 )`. Met deze getallen moet het programma alleen "Klaar" afdrukken. Als je de nul verwijdert, moet het programma 85 afdrukken (en "Klaar").

7.4 Geneste loops

Je kunt een loop in een andere loop stoppen.

Dat is een simpele uitspraak, maar ook een van de lastigste concepten voor veel studenten om te bevatten.

Ik zal eerst een voorbeeld geven van een dubbel-geneste loop, dus een loop die een andere loop bevat. In dat geval spreken programmeurs vaak over een "buitenste loop" en "binnenste loop." De binnenste loop is een deel van het blok code van de buitenste loop.

listing0711.py

```
for i in range( 3 ):
    print( "De buitenste loop begint met i =", i )
    for j in range( 3 ):
        print( "    De binnenste loop begint met j =", j )
        print( "    (i,j) = ({},{})".format( i, j ) )
        print( "    De binnenste loop eindigt met j =", j )
    print( "De buitenste loop eindigt met i =", i )
```

Bestudeer deze code en output totdat je hem volledig begrijpt!

De code geeft eerst aan variabele `i` de waarde 0, en geeft dan aan variabele `j` de waardes 0, 1, en 2. Dan geeft het aan `i` de waarde 1, en laat `j` weer de waardes 0, 1, en 2 aannemen. Tenslotte geeft het `i` de waarde 2, en laat `j` weer 0, 1, en 2 aannemen. Dus deze code verwerkt alle paren `(i,j)` waarbij `i` en `j` 0, 1, of 2 zijn.

Merk op dat de waardes voor variabelen in de buitenste loop ook beschikbaar zijn voor de binnenste loop. `i` bestaat zowel in de buitenste als in de binnenste loop.

Stel je voor dat je alle paren `(i,j)` wilt afdrukken waarbij `i` en `j` de waardes 0 tot en met 3 kunnen aannemen, maar `j` altijd groter moet zijn dan `i`. Dit gaat als volgt:

```
for i in range( 4 ):
    for j in range( i+1, 4 ):
        print( "({},{})".format( i, j ) )
```

Zie je hoe ik slim de waarde van `i` gebruik om het bereik van `j` te bepalen?

Opgave Schrijf een programma dat alle paren `(i,j)` afdrukt, waarbij `i` en `j` de waardes 0 tot en met 3 kunnen aannemen, maar ze nooit dezelfde waarde mogen hebben.

Je kunt natuurlijk ook **`while`** loops nesten, of **`for`** loops en **`while`** loops door elkaar heen gebruiken.

Merk op dat als je een **`break`** of **`continue`** gebruikt in een binnenste loop, dat alleen effect heeft op de binnenste loop. Er is bijvoorbeeld geen commando dat je in de binnenste loop kunt gebruiken dat dan zowel de binnenste als de buitenste loop meteen afbreekt.[7]

Als je dubbel-geneste loops begrijpt, zal het waarschijnlijk geen verrassing zijn te horen dat je ook drievoudig-geneste loops kunt gebruiken, of viervoudig-geneste loops, of zelfs dieper. In de praktijk zie je echter zelden een loop die dieper genest is dan drievoudig.

```
for i in range( 3 ):
    for j in range( 3 ):
        for k in range( 3 ):
            print( "({},{},{})".format( i, j, k ) )
```

[7] Tenzij je ze in een functie gebruikt, dan kun je wel in één keer de functie en dus de loops afbreken. Maar dat volgt in hoofdstuk 8.

7.5 De loop-en-een-half

Stel dat je de gebruiker om paren getallen vraagt in een loop. Voor ieder paar getallen dat de gebruiker ingeeft wil je laten zien wat hun vermenigvuldiging is. Je wilt de gebruiker het programma laten stoppen als hij een nul ingeeft, ongeacht voor welk getal. Vanwege een onbekende reden mogen de twee getallen geen delers van elkaar zijn; als ze dat wel zijn is dat een fout en wordt het programma onmiddellijk gestopt met een foutboodschap. Tenslotte is het een eis dat de getallen in het bereik 0 tot en met 1000 liggen; als de gebruiker een getal ingeeft dat niet in dat bereik zit wordt dat echter niet beschouwd als een fout; je wilt gewoon dat de gebruiker dan een nieuw getal ingeeft. Hoe programmeer je zoiets? Hier is een eerste poging:

listing0712.py

```
from pcinput import getInteger

x = 3
y = 7

while (x != 0) and (y != 0) and (x%y != 0) and (y%x != 0):
    x = getInteger( "Geef nummer 1: " )
    y = getInteger( "Geef nummer 2: " )
    if (x > 1000) or (y > 1000) or (x < 0) or (y < 0):
        print( "Nummers moeten tussen 0 en 1000 zijn" )
        continue
    print( x, "keer", y, "is", x * y )

if x == 0 or y == 0:
    print( "Klaar!" )
else:
    print( "Fout: de nummers mogen geen delers zijn" )
```

Opgave Bestudeer deze code en maak een lijstje van alles wat je er slecht aan vindt. Als je dat gedaan hebt, lees dan verder en vergelijk je bevindingen met het lijstje hieronder. Als je zaken hebt aangetroffen die slecht zijn en die niet op de lijst staan, kun je me emailen.

Er zijn veel zaken die ik slecht vind aan deze code. Hier is mijn lijst:

- Om ervoor te zorgen dat de loop op zijn minst één keer wordt uitgevoerd, moeten `x` en `y` geïnitialiseerd worden. Waarom met 3 en 7? Dat is willekeurig, maar ik moest twee getallen nemen die geen delers van elkaar zijn. Anders zou de loop niet zijn gestart. Over het algemeen is het niet netjes om variabelen startwaardes te geven die lijken een betekenis te hebben, terwijl ze er alleen maar zijn om de variabelen te laten bestaan terwijl de waardes betekenisloos zijn. Dat wil je het liefst vermijden.
- Als je iets ingeeft dat de loop zou moeten beëindigen (bijvoorbeeld nul voor `x`), dan wordt alsnog de vermenigvuldiging uitgevoerd voordat de loop eindigt. Dat was niet de bedoeling.
- Als je 0 ingeeft voor `x`, dan wordt alsnog om `y` gevraagd, hoewel het niet uitmaakt wat voor waarde voor `y` wordt ingegeven.

- De boolean expressie naast de **while** is nogal complex. In deze code is hij nog wel leesbaar, maar meer eisen aan de getallen maken het behoorlijk onbegrijpelijk.
- De foutboodschap voor de delers wordt niet gegeven bij de test waar besloten wordt het programma af te breken (dat wil zeggen, de boolean expressie bij de **while**).

De oplossing voor deze zaken die door sommige programmeurs geprefereerd wordt, is om x en y te initialiseren met waardes die de gebruiker ingeeft. Dit lost de willekeurige initialisatie op, en lost ook het probleem op dat je de vermenigvuldiging uitvoert als dat niet meer hoeft. Je moet dan wel het vragen om input verplaatsen naar het einde van de loop. Als er dan een **continue** voorkomt in de loop, moet je vlak voor die continue ook om de inputs vragen, anders krijg je een eindeloze loop. De code wordt dan iets als:

listing0713.py

```
from pcinput import getInteger

x = getInteger( "Geef nummer 1: " )
y = getInteger( "Geef nummer 2: " )

while (x != 0) and (y != 0) and (x%y != 0) and (y%x != 0):
    if (x > 1000) or (y > 1000) or (x < 0) or (y < 0):
        print( "Nummers moeten tussen 0 en 1000 zijn" )
        x = getInteger( "Geef nummer 1: " )
        y = getInteger( "Geef nummer 2: " )
        continue
    print( x, "keer", y, "is", x * y )
    x = getInteger( "Geef nummer 1: " )
    y = getInteger( "Geef nummer 2: " )

if x == 0 or y == 0:
    print( "Klaar!" )
else:
    print( "Fout: de nummers mogen geen delers zijn" )
```

De code vermijdt twee van de drie genoemde problemen, maar voegt een nieuwe toe, die het mijns inziens alleen maar erger maakt. De lijst van problemen wordt:

- Het vragen van input komt nu drie keer voor in de code, in plaats van slechts één keer.
- Als je 0 ingeeft voor x, vraagt de code nog steeds om een waarde voor y.
- De boolean expressie bij de **while** is nogal complex.
- De foutmelding voor de delers staat niet bij de plek waar besloten wordt de loop te verlaten.

De "truc" die je kunt gebruiken om deze problemen te verhelpen is de besturing van de loop puur te doen middels **continue**s en **break**s (een misschien soms een exit() als er een fout optreedt, maar in het volgende hoofdstuk wordt daar een "nettere" oplossing voor geboden). Dus je voert de loop "altijd" uit, maar je besluit om de loop te verlaten of opnieuw in te gaan als bepaalde omstandigheden optreden die je pas *in* de loop controleert

(en niet vooraf). Om een loop "altijd" uit te voeren kun je **while True** gebruiken (dit betekent: de test die je uitvoert om te besluiten of de loop uitgevoerd moet worden, geeft altijd **True**).

listing0714.py

```
from pcinput import getInteger
from sys import exit

while True:
    x = getInteger( "Geef nummer 1: " )
    if x == 0:
        break
    y = getInteger( "Geef nummer 2: " )
    if y == 0:
        break
    if (x < 0 or x > 1000) or (y < 0 or y > 1000):
        print( "Nummers moeten tussen 0 en 1000 zijn" )
        continue
    if x%y == 0 or y%x == 0:
        print( "Fout: de nummers mogen geen delers zijn" )
        exit()
    print( x, "keer", y, "is", x * y )

print( "Klaar!" )
```

Deze code lost vrijwel alle problemen op. Het vraagt slechts eenmalig om `x` en `y`. Er is geen willekeurige initialisatie van `x` en `y`. De loop stopt onmiddellijk als een nul wordt ingegeven. En de foutmelding staat meteen daar waar de fout geconstateerd wordt. Er is geen complexe boolean expressie nodig met een hoop **and**s en **or**s. De code is een beetje langer dan de eerste versie, maar lengte maakt niet uit, en de code is een stuk leesbaarder.

Het enige probleem dat nog over is, is dat als de gebruiker een waarde ingeeft buiten het bereik 0 tot en met 1000 voor `x`, hij nog steeds een waarde voor `y` moet ingeven alvorens het programma zegt dat de getallen opnieuw ingegeven moeten worden. Dat kun je het beste oplossen met functies, wat besproken wordt in het volgende hoofdstuk (je kunt het eventueel nu al oplossen met een geneste loop, maar ik zou me er niet druk om maken).

Een loop als deze, die **while True** gebruikt, wordt soms een "loop-en-een-half" genoemd. Het is een heel gebruikelijke aanpak voor het schrijven van een loop waarbij je niet precies weet wanneer de loop moet eindigen.

Opgave De gebruiker geeft een positief geheel getal. Je gebruikt daarvoor de `getInteger()` functie van `pcinput`. Deze functie staat het echter ook toe om negatieve getallen in te geven. Als de gebruiker een negatief getal ingeeft, wil je melden dat dat niet mag, en hem opnieuw een getal laten ingeven. Dit blijf je doen totdat daadwerkelijk een positief getal is ingegeven. Zodra een positief getal is ingegeven, druk je dat af en stopt het programma. Een dergelijk probleem wordt typisch aangepakt met een loop-en-een-half, omdat je geen idee hebt van hoe vaak een gebruiker een negatief getal ingeeft totdat hij wijs wordt. Schrijf zo'n loop-en-een-half. Je heb precies één **break** nodig, en hoogstens één **continue**. Druk het positieve getal dat de gebruiker heeft ingegeven af *na* de loop. De

reden om het erna te doen is dat de loop alleen bedoeld is om de input onder controle te krijgen, en niet voor het verwerken van de correcte ingave.

Ik heb vaak geconstateerd dat studenten het gebruik van **`while True`** maar verwarrend vinden. Ze zien het vaak in voorbeeldcode, maar ze snappen niet echt het nut ervan. En vervolgens gaan ze **`while True`** opnemen in code van henzelf als ze niet precies weten wat ze moeten doen. Als je problemen hebt de loop-en-een-half te begrijpen, bestudeer dan deze paragraaf opnieuw, totdat je het wel begrijpt.

7.6 Slim gebruik van loops

Ter afronding van dit hoofdstuk, bediscussieer ik een aantal strategieën voor het ontwerpen van loops, en het ontwerpen van algoritmes in het algemeen.

7.6.1 Wanneer gebruik je een loop

Als je vijf zes-zijdige dobbelstenen rolt, hoe groot is dan de waarschijnlijkheid dat je vijf zessen rolt? het antwoord is $1/(6^5)$, maar stel dat je dat niet weet, en je wilt een simulatie gebruiken om de waarschijnlijkheid te schatten. Je kunt het rollen van een dobbelstenen imiteren via `randint()`, dus daarmee kun je ook het rollen van vijf dobbelstenen imiteren. Je kunt testen of ze alle zes zijn. Dat kun je een heleboel keren doen, en aan het eind van je programma deel je het aantal keren dat er vijf zessen vielen door het totaal aantal pogingen. Als ik dit probleem voorleg aan studenten (in een iets ingewikkeldere vorm zodat het antwoord niet gemakkelijk berekend kan worden), dan krijg ik vaak code die er als volgt uitziet (je moet `TESTS` een grotere waarde geven voor een betere schatting, maar ik wilde dat deze code niet teveel tijd vergt om uit te voeren):

listing0715.py

```
from random import randint

TESTS = 10000
succes = 0
for i in range( TESTS ):
    d1 = randint( 1, 6 )
    d2 = randint( 1, 6 )
    d3 = randint( 1, 6 )
    d4 = randint( 1, 6 )
    d5 = randint( 1, 6 )
    if d1 == 6 and d2 == 6 and d3 == 6 and d4 == 6 and d5 == 6:
        succes += 1
print( "Waarschijnlijkheid van vijf zessen is", succes / TESTS )
```

Als ik dit soort code zie, vraag ik: “Wat had je gedaan als ik gevraagd had 100 dobbelstenen te rollen? Had je 100 variabelen gemaakt en die regel code die ze rolt 100 keer gekopieerd?” Als je een stukje code hebt dat een paar keer herhaald wordt met slechts een kleine wijziging erin (of wanneer je aan het kopiëren en plakken bent in je eigen code), dan moet je beginnen te denken aan loops. Je kunt vijf dobbelstenen als volgt rollen:

```
from random import randint

for i in range( 5 ):
    d = randint( 1, 6 )
```

"Maar," hoor ik sommigen al protesteren: "ik moet de waarde van al die vijf dobbelstenen hebben om te testen of het vijf zessen zijn! Iedere keer dat deze code door de loop gaat, gooi je de vorige dobbelsteenwaarde weg!" Dat is waar, maar de regel die test of ze allemaal een zes zijn is ook erg lelijk. Kan dit geheel niet gestroomlijnd worden? Kun je geen conclusie trekken als je één dobbelsteen rolt?

Als je hier een beetje over nadenkt, kom je wellicht op het volgende idee: zodra je een dobbelsteen rolt die géén zes is, heb je al gefaald in je poging om vijf zessen te rollen, en kun je gelijk doorgaan met de volgende poging. Er zijn diverse manieren om dit te implementeren, maar hier is een hele korte die gebruik maakt van een **break** en een **else**:

listing0716.py

```
from random import randint

TESTS = 10000
succes = 0
for i in range( TESTS ):
    for j in range( 5 ):
        if randint( 1, 6 ) != 6:
            break
    else:
        succes += 1
print( "Waarschijnlijkheid van vijf zessen is", succes / TESTS )
```

Je denkt wellicht dat het moeilijk is om zoiets te verzinnen, maar het is echt niet de enige manier. Je kunt, bijvoorbeeld, de waardes van de dobbelstenen in de loop optellen en dan testen of het totaal 30 is na de loop. Of je kunt tellen hoeveel dobbelstenen een zes zijn en dan testen of dat vijf is na de loop. Of je kunt voor de loop een boolean variabele op **True** zetten, en die in de loop op **False** zetten zodra een dobbelsteen niet op zes valt; dan kun je die boolean testen na de loop.

Het punt is dat een willekeurig lange herhaling van stukken code vrijwel altijd vervangen kan worden door een loop.

7.6.2 Data items één voor één verwerken

Het is gebruikelijk dat loops een serie data items verwerken. Iedere cyclus door de loop verwerkt dan één van die items. Vaak moet je iets onthouden over de items die je verwerkt hebt, en daarvoor heb je extra variabelen nodig. Je moet slim nadenken over welke variabelen je nodig hebt.

Neem het volgende voorbeeld: ik geef je tien getallen en vraag je een programma te schrijven dat bepaalt welke de grootste is, welke de kleinste, en hoeveel er deelbaar zijn door 3.

Je kunt dan zeggen: "Het is gemakkelijk te bepalen wat de grootste en de kleinste zijn: daar gebruik ik gewoon de **`max`**`()` en **`min`**`()` functies voor (besproken in hoofdstuk 5). Deelbaar door 3 is misschien wat moeilijker, daar moet ik over denken." Maar **`max`**`()` en **`min`**`()` maken het noodzakelijk dat alle tien de getallen tegelijkertijd in het geheugen worden gehouden. Dat is nog te doen voor tien getallen, maar wat als ik om honderd had gevraagd? Of een miljoen?

Omdat je een serie getallen moet verwerken, moet je denken over een loop, en specifiek over een loop waarin iedere cyclus door de loop één van die getallen beschikbaar is (maar waarbij ze allemaal voorbij zullen komen voordat de loop eindigt). Je moet nu denken over variabelen waarmee je iets kunt onthouden in de loop, die ervoor zorgen dat je aan het einde van de loop kunt bepalen welke de grootste was, welke de kleinste, en hoeveel deelbaar zijn door 3. Welke variabelen heb je nodig?

Het antwoord, dat gemakkelijk te verzinnen is voor iemand die wat ervaring heeft met programmeren, is dat je iedere cyclus door de loop moet onthouden welk getal het grootste is *tot nu toe*, welk het kleinste is *tot nu toe*, en hoeveel er deelbaar zijn door 3 *tot nu toe*. Dat betekent dat iedere keer dat je door de loop gaat het nieuw aangeboden getal vergelijkt met de variabelen die de grootste en de kleinste bijhouden, en je de inhoud van de variabelen vervangt als dat nodig is. Je test ook of het nieuw aangeboden getal deelbaar is door 3, en zo ja, dan verhoog je de variabele die bijhoudt hoeveel er deelbaar zijn door 3 met 1.

Je moet goede initiële waardes bepalen voor de drie variabelen. De variabele die deelbaar-door-3 bijhoudt is eenvoudig; die begint op nul. Het is wat lastiger voor de grootste en kleinste variabelen. Een goede oplossing is ze te vullen met het eerste getal, want op dat moment is dat eerste getal zowel de grootste als de kleinste.

Dit probleem staat hieronder als een genummerde opgave. Gebruik het beschreven algoritme om de opgave op te lossen.

7.6.3 Begin met de kleinste en bouw naar buiten op

Stel dat ik je de volgende opdracht geef: Van alle boeken op alle planken in de bibliotheek moet je het aantal woorden tellen, en tenslotte moet je het gemiddelde aantal woorden per boek rapporteren. Als je dit aan een mens vraagt, zal hij of zij waarschijnlijk denken: "Ik ga naar de bibliotheek, neem het eerste boek van de eerste plank, tel de woorden en schrijf het totaal op, dan neem ik het tweede boek en doe hetzelfde, etcetera. Als ik klaar ben met de eerste plank, doe ik hetzelfde met de tweede, totdat alle boeken op alle planken gedaan heb. Dan tel ik alle totalen op, en deel dat door het aantal boeken." Voor mensen werkt dit, maar als ik dit aan een computer wil vertellen, is dat best lastig.

Om dit probleem op te lossen, moet ik beginnen met de kleinste eenheid van verwerking. In dit geval is de kleinste eenheid een "boek." Het is niet "woord," want ik hoef niks te doen met woorden; ik moet totalen woorden per boek hebben. In pseudo-code,[8] wordt het tellen van woorden per boek iets als:

```
woordtotaal = 0
for woord in boek:
    woordtotaal += 1
```

[8] pseudo-code is geen echte code, maar het leest als code, en zou als zodanig gemakkelijk te implementeren moeten zijn in verschillende programmeertalen.

Als ik zoiets codeer, kan ik het al testen. Als ik tevreden ben dat ik woorden per boek kan tellen, kan ik naar een iets grotere eenheid gaan, en dat is de "plank." Hoe verwerk ik alle boeken op een plank? In pseudo-code is dat iets als:

```
for boek on plank:
    verwerk_boek()
```

Wat doet `verwerk_boek()`? Het telt de woorden. Ik heb al pseudo-code geschreven om woorden te tellen, en die kan ik hier invoegen. Dat wordt dan:

```
for boek in plank:
    woordtotaal = 0
    for woord in boek:
        woordtotaal += 1
```

Als ik dit test, loop ik tegen een probleem aan. Ik tel weliswaar woorden per boek, maar ik doe niks met die totalen. Ik overschrijf die gewoon. Om straks het gemiddelde te kunnen bepalen, moet ik het totaal van het aantal woorden weten over alle boeken. Dat betekent dat ik `woordtotaal` slechts één keer moet initialiseren, namelijk aan het begin.

```
woordtotaal = 0
for boek in plank:
    for woord in boek:
        woordtotaal += 1
```

Maar om het gemiddelde te kunnen uitrekenen, moet ik ook weten hoeveel boeken ik verwerkt heb. Dat kan ik doen met een teller `boektotaal`, die ook slechts eenmalig geïnitialiseerd moet worden. Aan het einde kan ik dan het gemiddelde afdrukken.

```
woordtotaal = 0
boektotaal = 0
for boek in plank:
    for woord in boek:
        woordtotaal += 1
    boektotaal += 1
print( woordtotaal / boektotaal )
```

Nu kan ik naar het hoogste niveau gaan: de bibliotheek als geheel. Ik weet hoe ik één plank moet verwerken, en nu moet ik alle planken verwerken. Natuurlijk moet de initialisatie van de totalen alleen aan het begin gedaan worden, en het afdrukken van het gemiddelde alleen aan het einde. Dat wil zeggen dat ik slechts één regel hoef toe te voegen om de pseudo-code af te maken:

```
woordtotaal = 0
boektotaal = 0
for plank in bibliotheek:
    for boek in plank:
        for woord in boek:
            woordtotaal += 1
        boektotaal += 1
print( woordtotaal / boektotaal )
```

Zoals je ziet heb ik een drievoudig-geneste loop ontworpen, werkend van binnen naar buiten. Dit is een goede aanpak als je met geneste loops aan de slag moet.

7.7 Over het ontwerpen van algoritmes

Als je tot op dit punt van het boek gestaag hebt doorgewerkt, kom je vanaf nu opgaves en problemen tegen waarbij je niet zeker bent over hoe je ze op moet lossen. Ik gaf een voorbeeld van een dergelijk probleem hierboven (het vinden van de grootste, de kleinste, en het aantal deelbaar door 3 van tien getallen), en de oplossing die ik bedacht. Een dergelijke oplossing wordt een algoritme genoemd. Maar hoe ontwerp je zo een algoritme?

Ik zie vaak dat studenten code aan het typen zijn zonder dat ze weten wat ze doen of wat ze willen doen. Ze proberen een opgave op te lossen maar weten niet hoe, en dus gaan ze maar typen. Je zult je wel realiseren dat dat geen effectieve manier is om oplossingen te maken (hoewel er op zich niks is tegen een beetje experimenteren).

Wat je in dat soort situaties moet doen is een stapje terugzetten, het toetsenbord met rust laten, en denken: "Hoe zou ik dit als mens aanpakken?" Probeer op te schrijven wat je zou doen als je het probleem met de hand zou aanpakken. Het maakt niet uit of het een saaie taak is die je nooit met de hand zou willen doen – je hebt een computer om saaie dingen voor je te doen.

Als je bedacht hebt wat je zou willen doen, bedenk dat hoe je dat in code kunt opschrijven. Want in principe is dat wat je de computer moet vertellen: de stappen die een mens zou kunnen nemen om de oplossing te bereiken. Als je geen manier kunt vinden waarmee een mens het probleem zou oplossen, dan zul je zeker niet in staat zijn een computer te vertellen hoe het probleem opgelost moet worden.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Wat loops zijn
- **`while`** loops
- **`for`** loops
- Eindeloze loops
- Loop controle via **`else`**, **`break`**, en **`continue`**
- Geneste loops
- De loop-en-een-half
- Slim zijn met loops

Opgaves

Omdat loops geweldig belangrijk zijn en studenten er vaak problemen mee hebben, geef ik hierbij een groot aantal opgaven. Ik raad je aan ze allemaal te maken. Je zult er veel van leren.

Opgave 7.1 Schrijf een programma dat de gebruiker een getal laat ingeven. Het programma geeft de tafel van vermenigvuldiging van het getal voor 1 tot en met 10. Bijvoorbeeld, als de gebruiker 12 ingeeft, dan is de eerste regel die afgedrukt wordt "`1 * 12 =` 12" en de laatste regel "`10 * 12 = 120`".

Opgave 7.2 Als je de vorige opgave met een **`while`** loop hebt gedaan, doe hem dan nogmaals met een **`for`** loop. Als je hem met een **`for`** loop hebt gedaan, doe hem dan nogmaals met een **`while`** loop. Als je hem gedaan hebt zonder loop, dan moet je je schamen.

Opgave 7.3 Schrijf een programma dat de gebruiker vraagt om 10 getallen, en dan de grootste, de kleinste, en het aantal deelbaar door 3 afdrukt. Gebruik het algoritme dat eerder in dit hoofdstuk beschreven is.

Opgave 7.4 "99 bottles of beer" is a traditioneel liedje gezongen in Amerika en Canada. Het wordt vaak gezongen op lange reizen omdat het gemakkelijk te onthouden en mee te zingen is, en lang duurt. In vertaling is de tekst: "99 flesjes met bier op de muur, 99 flesjes met bier. Open er een, drink hem meteen, 98 flesjes met bier op de muur." Deze tekst wordt herhaald, steeds met één flesje minder. Het lied is voorbij als de zangers nul bereiken. Schrijf een programma dat het hele lied afdrukt (ik raad je aan te beginnen met niet meer dan 10 flesjes). Kijk uit dat je je loop niet eindeloos maakt. Zorg er ook voor dat je het juiste meervoud voor het woord "flesje" gebruikt.

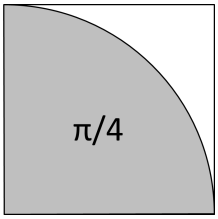
Opgave 7.5 De Fibonacci reeks is een serie getallen die start met 1, gevolgd door nogmaals 1. Ieder volgende getal is de som van de twee voorgaande getallen. De reeks start dus met 1, 1, 2, 3, 5, 8, 13, 21,... Schrijf een programma dat de Fibonacci reeks afdrukt totdat de getallen groter dan 1000 zijn.

Opgave 7.6 Schrijf een programma dat vraagt om twee woorden. Druk alle letters af die de woorden gemeen hebben. Je mag hoofletters beschouwen als verschillend van kleine letters, maar iedere letter die je rapporteert, mag slechts één keer gerapporteerd worden (bijvoorbeeld, de strings "een" en "peer" hebben slechts één letter gemeen, namelijk de letter "e"). Hint: Sla de letters die de woorden gemeen hebben op in een derde string, en als je een letter vindt die beide woorden gemeen hebben, test je of de letter al in de derde string staat alvorens je hem rapporteert.

Opgave 7.7 Schrijf een programma dat π benadert door middel van toevalsgetallen, als volgt: Neem een vierkant van 1 bij 1. Als je een dart in dat vierkant gooit op een willekeurige plek, is de waarschijnlijkheid dat de afstand van de dart tot de linkeronderhoek van het vierkant kleiner dan 1 is gelijk aan $\pi/4$. Om dat in te zien, bedenk dat de oppervlakte van een cirkel met straal 1 π is, dus de oppervlakte van een kwart cirkel is $\pi/4$. Als dus een dart op een willekeurig punt in het vierkant landt, is de waarschijnlijkheid dat die dart in de kwart cirkel landt $\pi/4$. Als je N darts in het vierkant werpt, en M ervan landen in de kwart cirkel, dan benadert $4M/N$ de waarde π als N voldoende groot is.

Het programma heeft een constante die aangeeft hoeveel darts gesimuleerd moeten worden. Het toont een benadering van π door het werpen van zoveel darts te simuleren. De

afstand van een punt (x, y) tot de linkeronderhoek kun je berekenen als $sqrt(x^2 + y^2)$. Gebruik de `random()` functie uit de `random` module.



Opgave 7.8 Schrijf een programma dat een toevalsgetal neemt tussen 1 en 1000 (je kunt `randint()` daarvoor gebruiken). Het programma vraagt de gebruiker het getal te raden. Na iedere poging van de gebruiker zegt het programma "Lager" als het te raden getal lager is, "Hoger" als het te raden getal hoger is, of "Je hebt het geraden!" als het getal correct is. Het programma eindigt met afdrukken hoeveel pogingen de gebruiker nodig had om het getal te raden. Voor test-doeleinden kan het slim zijn om het te raden getal op het scherm te laten zien, totdat je zeker weet dat het programma goed werkt.

Opgave 7.9 Schrijf een programma dat het omgekeerde is van het vorige: nu neemt de gebruiker een getal in gedachten en de computer probeert het te raden. Op de pogingen van de computer moet de gebruiker antwoorden met een letter: "L" voor lager als het te raden getal lager is, "H" voor hoger als het te raden getal hoger is, en "C" voor correct (je kunt de `getLetter()` functie van `pcinput` hiervoor gebruiken). Als de computer het getal geraden heeft, drukt het af hoeveel pogingen er nodig waren. Zorg ervoor dat de computer ook herkent dat er geen mogelijk antwoord is (misschien omdat de gebruiker een vergissing heeft gemaakt, of omdat de gebruiker de computer in het ootje probeerde te nemen). Een slim programma hoeft hoogstens tien keer te raden.

Opgave 7.10 Een priemgetal is een positief geheel getal dat deelbaar is door precies twee verschillende getallen, namelijk 1 en het priemgetal zelf. Het laagste en enige even priemgetal is 2. De eerste 10 priemgetallen zijn 2, 3, 5, 7, 11, 13, 17, 19, 23, en 29. Schrijf een programma dat de gebruiker om een getal vraagt en dan afdrukt of het getal een priemgetal is. Hint: Test in een loop alle mogelijke delers van het getal. In deze loop kun je concluderen dat een getal niet priem is zodra je een deler (anders dan 1 of het getal zelf) hebt gevonden. Je kunt echter alleen na afloop van de loop constateren dat het getal priem is, want dan heb je pas alle mogelijke delers getest.

Opgave 7.11 Schrijf een programma dat een tafel van vermenigvuldiging afdrukt voor de cijfers 1 tot en met een zeker getal `num` (je mag voor de output aannemen dat `num` één cijfer is). Een tafel van vermenigvuldiging voor 1 tot en met 3 ziet er als volgt uit:

```
. |  1  2  3
-----------
1 |  1  2  3
2 |  2  4  6
3 |  3  6  9
```

Dus de cellen van de matrix bevatten het label van de kolom vermenigvuldigd met het label van de rij.

Opgave 7.12 Schrijf een programma dat alle gehele getallen tussen 1 en 100 afdrukt die geschreven kunnen worden als de som van twee kwadraten. De uitvoer is een lijst van regels van de vorm `z = x**2 + y**2`, bijvoorbeeld, `58 = 3**2 + 7**2`. Als een getal twee keer wordt afgedrukt met verschillende manieren om het te schrijven als de som van twee kwadraten, dan is dat acceptabel (dit kan gelden voor 50, 65, en 85).

Opgave 7.13 Je rolt vijf zes-zijdige dobbelstenen, één voor één. Hoe groot is de waarschijnlijkheid dat de waarde van iedere dobbelsteen gelijk is aan of groter is dan de waarde van de vorige dobbelsteen? Bijvoorbeeld, "1, 1, 4, 4, 6" is een succes, maar "1, 1, 4, 3, 6" is dat niet. Bepaal deze waarschijnlijkheid door een groot aantal pogingen te simuleren.

Opgave 7.14 A, B, C, en D zijn alle verschillende cijfers. Het getal DCBA is gelijk aan 4 keer het getal ABCD. Wat zijn de cijfers? Om ABCD en DCBA conventionele schrijfwijzes van getallen te laten zijn, mag noch A, noch D nul zijn. Gebruik een viervoudig geneste loop.

Opgave 7.15 Volgens een oude puzzel zijn vijf piraten en hun aap gestrand op een eiland. Gedurende de dag verzamelen ze kokosnoten, die ze op een grote stapel leggen. Wanneer de nacht valt gaan ze slapen.

Midden in de nacht wordt de eerste piraat wakker. Hij vertrouwt zijn makkers niet, dus hij verdeelt de stapel kokosnoten in vijf gelijke delen, neemt wat hij aanneemt dat zijn deel is, en verstopt dat. Hij houdt één kokosnoot over, en die geeft hij aan de aap. Dan valt hij weer in slaap.

Een uur later wordt de tweede piraat wakker. Hij handelt net als de eerste piraat: hij verdeelt de stapel in vijf gelijke delen, neemt wat hij denkt dat zijn deel is, en verstopt dat. Ook hij houdt een kokosnoot over die hij aan de aap geeft. Dan valt hij weer in slaap.

Hetzelfde gebeurt de andere drie piraten: één voor één worden ze wakker, verdelen de stapel, geven een kokosnoot aan de aap, verstoppen hun deel, en gaan weer slapen.

In de ochtend worden ze allemaal wakker. Ze verdelen eerlijk wat er van de stapel kokosnoten over is. Dat laat één kokosnoot over, en die gaat naar de aap.

De vraag is: wat is het kleinste aantal kokosnoten dat er op de originele stapel kan hebben gelegen?

Schrijf een Python programma dat de puzzel oplost. Het is nog beter als je het kunt oplossen voor een willekeurig aantal piraten.

Opgave 7.16 Beschouw de driehoek die hier beneden is weergegeven. Deze driehoek bevat een kolonie Driehoekkruipers, en een Verslinder van Driehoekkruipers. De Verslinder zit in punt D. Alle Driehoekkuipers worden geboren in punt A. Een Driehoekkruiper die bij punt D aankomt, wordt opgegeten.

Iedere dag kruipt iedere Driehoekkruiper over een van de lijnen in de driehoek naar een willekeurig bepaald punt, maar niet naar het punt waar hij de dag ervoor was. Deze beweging kost één dag. Bijvoorbeeld, een Driehoekkruiper die net geboren is in punt A, zal

op de eerste dag van zijn leven kruipen naar B, C, of D. Als hij naar B gaat, zal hij de volgende dag naar C of D gaan, maar niet terug naar A. Als hij naar C gaat, zal hij de volgende dag naar B of D gaan, maar niet terug naar A. Als hij naar D gaat, wordt hij opgegeten.

De waarschijnlijkheid dat een Driehoekkruiper op de eerste dag onmiddellijk naar D gaat is 1/3, en dan leeft hij dus maar één dag. In principe kan een Driehoekkruiper iedere willekeurige leeftijd bereiken, hoe hoog ook, door in cirkels te bewegen van A naar B naar C en terug naar A (of tegen de klok in, van A naar C naar B en terug naar A). Maar omdat hij iedere dag na de eerste per toeval kiest voor een volgend punt, is iedere dag na de eerste de waarschijnlijkheid dat hij wordt opgegeten 1/2.

Schrijf een programma dat een benadering berekent van de gemiddelde leeftijd waarop een Driehoekkruiper overlijdt. Doe dit door de simulatie van 100.000 Driehoekkruipers, waarbij je alle dagen dat ze leven optelt, en het totaal deelt door 100.000. De uitvoer van je programma moet gegeven zijn als een gebroken getal met twee decimalen.

Hint 1: Er zijn twee manieren waarop je dit programma kunt aanpakken: ofwel je simuleert het gedrag van één Driehoekkruiper en herhaalt dat 100.000 keer, ofwel je start met een populatie van 100.000 Driehoekkruipers in punt A, en verdeelt die over variabelen die bijhouden hoeveel Driehoekkruipers in ieder punt zitten op iedere dag, waarbij je ook bijhoudt van welk punt ze komen (overblijvende Driehoekkruipers worden aan een toevallig naburig punt toebedeeld). De eerste methode is kort en eenvoudig maar traag, de tweede is lang en complex maar snel. Je mag zelf kiezen welke methode je wilt gebruiken.

Hint 2: Begin niet met 100.000 Driehoekkruipers voor je eerste pogingen. Start met 1000 (of misschien met slechts 1), en probeer het pas met 100.000 als je weet dat je programma min of meer klaar is. Testen gaat veel sneller met minder Driehoekkruipers. 1000 Driehoekkruipers kunnen worden gesimuleerd in minder dan een seconde, dus als je programma meer tijd nodig heeft, heb je waarschijnlijk een eindeloze loop gemaakt.

Hint 3: Ik zal niet te specifiek zijn, maar het antwoord is ergens tussen de 1 en 5 dagen. Als je iets buiten dat bereik krijgt, is het zeker fout. Je kunt het juiste antwoord wiskundig bepalen voordat je de opgave maakt, maar dat kan behoorlijk lastig zijn.