De Programmeursleerling

Leren coderen met Python 3

Pieter Spronck

De Programmeursleerling

Leren coderen met Python 3

Pieter Spronck

Version 1.0.15

26–08–2017

De originele vorm van dit boek is LaTeX source code. Het compileren van deze LaTeX source genereert een systeem-onafhankelijke versie van een tekstboek, die geconverteerd kan worden naar andere formaten, en die kan worden geprint.

De LaTeX source van dit boek wordt (mettertijd) beschikbaar gesteld via `http://www.spronck.net/pythonbook`.

Dit boek is een vertaling door de auteur zelf van het boek *The Coder's Apprentice: Learning Programming with Python 3*,

De meest recente versie van het boek zal steeds beschikbaar zijn via `http://www.spronck.net/pythonbook`.

Voorwoord

Computer technologie verandert de wereld in een hoog tempo.

Bijna 30 jaar geleden kreeg ik mijn eerste baan als computerprogrammeur. In die tijd gebruikten alleen grote bedrijven met uitgebreide administraties computers. Of liever gezegd, één computer, want het was zeldzaam dat een bedrijf er meerdere had. Vrijwel niemand had een PC in huis, Internet bestond nog niet, niemand had een mobiele telefoon. Mensen gebruikten schrijfmachines.

In de afgelopen 30 jaar is de manier waarop mensen leven en werken drastisch gewijzigd. Dit is buitengewoon duidelijk als je kijkt naar het werk dat mensen doen. Bijvoorbeeld, toen ik een kind was werd twee keer per dag post bezorgd – vandaag de dag komt de postbode twee keer per week langs, wat betekent dat het leger van professionele postbodes gedecimeerd is. Bankkantoren sluiten omdat bankieren veel gemakkelijker online gedaan kan worden. Helpdesks worden bemand door digitale avatars of vervangen door online informatiesystemen. Grote warenhuizen gaan failliet omdat mensen inkopen online doen, wat heeft geleid tot een enorme reductie in het aantal benodigde verkopers. En hoewel dat een lichte toename in de behoefte aan transportmedewerkers heeft veroorzaakt, met zelfrijdende voertuigen in het verschiet zal de werkgelegenheid voor chauffeurs snel tot nul afnemen.

Dit zijn alle beroepen voor wat we noemen "laag opgeleiden," maar dat wil niet zeggen dat "hoog opgeleiden" veilig zijn. Ik heb programmeren gedoceerd aan professionele journalisten, die me vertelden dat computers grote delen van hun werk overnemen, zoals het schrijven van eenvoudige artikelen en het doen van onderzoek – ze wilden mijn cursus volgen omdat ze wisten dat zonder vaardigheden in het omgaan met digitale technologie, ze binnenkort op straat zouden staan. Programma's zijn al ontwikkeld om eenvoudige maar o-zo-tijdrovende activiteiten van juristen te automatiseren, namelijk het doorzoeken van achtergrondmateriaal. Computers kunnen componeren, schilderen, en beeldhouwen – waarom zou je iemand een half jaar lang aan een blok graniet laten schaven als een 3D-printer een beeldhouwwerk in een paar uur tijd produceert? Zelfs het ontwerpen en uitvoeren van wetenschappelijke experimenten wordt in diverse wetenschapsgebieden al deels door computers gedaan.

In de 30 jaar dat ik werk, heb ik de arbeidsmarkt zien veranderen van een situatie waarin het gebruik van computers zeldzaam is, tot een situatie waarin de behoefte aan menselijke arbeidskrachten enorm is afgenomen – ongeacht het beroep. En die verandering is niet ten einde.

Dat betekent niet dat er geen plaats meer is voor mensen op de arbeidsmarkt. Maar het betekent wel dat alleen mensen die iets kunnen bijdragen wat de computer niet goed alleen

kan doen, een baan kunnen krijgen. In de nabije toekomst zal de werkgelegenheid onveranderlijk verbonden zijn aan de mogelijkheden van werkzoekenden om de kracht van mensen en computers met elkaar te combineren zodat beide versterkt worden.

Het probleem is dat om de mogelijkheid te hebben computers in te zetten om de kwaliteit van je werk te verbeteren, het onvoldoende is als je een word processor of spreadsheet kunt gebruiken. Je moet daadwerkelijk in staat zijn de mogelijkheden van computers uit te buiten vanuit het perspectief van je eigen beroep. Bijvoorbeeld, een journalist die slechts in staat is een programma te draaien dat feiten van het Internet verzamelt, is niet nodig. Echter, een journalist die in staat is een dergelijk programma uit te breiden en te verbeteren, is een waardevolle kracht.

Om computers op een dergelijke creatieve manier te gebruiken, heb je vaardigheden nodig die je in staat stellen te denken en problemen op te lossen als een programmeur. Ik heb jarenlang studenten programmeren geleerd, en ik weet dat voor velen dit geen natuurlijke manier van werken is. Om de noodzakelijke vaardigheden te verkrijgen, moeten studenten een aantal intensieve cursussen op dit gebied volgen.

Als je bedenkt dat het de taak van universiteiten en hogescholen is om studenten voor te bereiden op een arbeidsmarkt waarop ze meer dan 40 jaar moeten functioneren, en als je overweegt dat in de nabije toekomst (als het niet al zover is) de kennis en kunde om de kracht van computers te incorporeren in om-het-even-welke baan een vereiste is voor iedere employee, zou je verwachten dat "programmeren" beschouwd wordt als een basisvaardigheid voor iedere student. Helaas is dat niet zo. Typische basiscursussen voor iedere student zijn "wetenschappelijk schrijven," "wetenschapsfilosofie," en "statistiek," maar de meeste opleidingen zien programmeren nog steeds als niet meer dan een optie. Dat is het niet.

Iedere opleiding die "programmeren" niet als een verplichte cursus op het programma zet, faalt in mijn visie in haar taak studenten te prepareren voor de arbeidsmarkt. Ik zou zelfs het liefste zien dat middelbaren scholen – of misschien zelfs basisscholen – de leerlingen al zouden leren programmeren, omdat programmeervaardigheden gemakkelijker te leren zijn op jonge leeftijd. De reden is dat programmeren een creatieve manier van denken vereist, die lastiger aan te leren is als je al getraind bent in de "reproductieve" manier van denken en werken die scholen aanleren.

Alle studenten, ongeacht hun richting, moeten kunnen programmeren. Niet omdat iedereen programmeur moet worden – professioneel programmeren is hoogst gespecialiseerd werk dat slechts weinigen hoeven te beheersen. Maar de kennis om programma's te kunnen bouwen geeft studenten de mogelijkheid te problemen aan te pakken als een programmeur, geeft hen inzicht in de mogelijkheden en beperkingen van computers, en geeft hen de kracht computers in te zetten in een specifiek domein op een uniek menselijke manier.

Het doel van dit boek is om iedereen die dat wil te leren programmeren in Python. Het boek is voornamelijk gericht op middelbaren scholieren, en studenten die onbekend zijn met programmeren. Het boek moet het mogelijk maken voor iedereen om basiskennis van programmeren op te doen, en zodoende voorbereid te zijn op de arbeidsmarkt van de eenentwintigste eeuw.

Pieter Spronck
2 mei 2016
Maastricht, Nederland

Pieter Spronck is een Computer Science professor bij Tilburg University.

Dankwoord

Ik dank Allen B. Downey, die het uitermate sterke Python 2 boek *Think Python: How to Think Like a Computer Scientist* heeft geschreven. Ikzelf heb Python programmeren geleerd met dit boek, en ik heb het LaTeX template van Downey's boek gebruikt om mijn eigen boek te schrijven. Downey heeft recentelijk een Python 3 versie van zijn boek uitgebracht. Als je Engels voldoende beheerst en al een beetje bekend bent met programmeren, doe je er goed aan dat boek erbij te halen – het is gratis te downloaden.

Peter Wentworth heeft een Python 3 versie van Downey's boek gemaakt. Zijn stijl van lesgeven is niet de mijne, maar ik heb een hoop informatie uit zijn boek gehaald, waarvoor dank.

Ik dank ook Guido van Rossum, die Python origineel ontworpen heeft. Ik ben gek op programmeren, maar slechts weinig programmeertalen zijn plezierig om te gebruiken. Python is zeer prettig in het gebruik, wat vooral aan Van Rossum te danken is.

Ik dank ook Ákos Kádár, Nanne van Noord, en Sander Wubben, die met mij gewerkt hebben aan een vroege versie van een Python cursus, waarop ik dit boek gebaseerd heb.

Ook dank ik de leden van Monty Python, wiens televisie en audio programma's mij Engels hebben geleerd, en aan wie Python haar naam te danken heeft. Ik heb hier en daar vertalingen van stukjes van hun teksten gebruikt in demonstraties en opgaves in dit boek.

Ik dank Myrthe Spronck, die de website bij dit boek gebouwd heeft. De website is te vinden via `http://www.spronck.net/pythonbook`.

Tenslotte dank ik iedereen die suggesties of correcties heeft ingestuurd (een lijst volgt hieronder).

Als je een suggestie of correctie hebt, stuur dan een email naar `pythonbook@spronck.net` (dat adres is natuurlijk niet bedoeld om vragen over programma's te stellen – er zijn vele plaatsen op het Internet waar programmeerhulp geboden wordt), of een boodschap achterlaten op het forum `http://www.spronck.net/forum`. Als ik een wijziging maak in het boek op basis van feedback, zal ik je naam in dit dankwoord opnemen (tenzij je aangeeft dat je dat liever niet hebt).

Bijdragen

- Larry Cali ontdekte een fout in het antwoord bij opgave 4.3, waar problemen konden optreden met waardes die Python niet precies kan opslaan. Ik heb het antwoord verbeterd en een opmerking gemaakt in hoofdstuk 3 (gecorrigeerd in versie 1.0.5).
- Isaac Kramer ontdekte een fout in opgave 9.5, die het probleem in de code onzichtbaar maakte. Ik heb de code zo aangepast dat het probleem inderdaad zichtbaar is zoals ik uitleg bij de antwoorden (gecorrigeerd in versie 1.0.6).
- Ruud van Cruchten gaf aan dat de uitleg over het geven van commentaar over meerdere regels in hoofdstuk 4 incompleet was en tot fouten kon leiden. Ik heb deze beschrijving uitgebreid (gecorrigeerd in versie 1.0.7).
- Nade Kang gaf aan dat het antwoord op Opgave 7.9 (tweede raad-spelletje) verwarrend kan zijn. Ik heb de code iets aangepast ter verduidelijking (gecorrigeerd in versie 1.0.7).

- Shiyu Zhang wees me op een paar nutteloze parameters in listing 8.16. Ik heb dit aangepast (gecorrigeerd in versie 1.0.8).
- Claudia Dai gaf een kleine fout in het antwoord op opgave 10.1 (klinkers tellen) aan (gecorrigeerd in versie 1.0.9).
- Een aantal studenten suggereerde dat het toevoegen van stroomdiagrammen aan de hoofdstukken over condities en iteraties zou helpen om meer begrip over deze concepten te krijgen. Ik heb deze suggestie opgevolgd (toegevoegd aan versie 1.0.9).
- Mauro Crociara gaf meerdere ideeën voor verbeteringen (toegevoegd aan versie 1.0.11).
- David Ytsma wees mij op een fout in het antwoord bij opgave 6.1 (cijfers geven; gecorrigeerd in versie 1.0.11).
- Chris Spinks wees mij op fouten in het antwoord code bij opgave 21.4, de uitgebreide fruitmand, en in de reguliere expressies bij opgaves 25.3 en 25.4, waar namen uit een tekst gehaald moeten worden (opgelost in versie 1.0.12).
- Dirk Remmelzwaal wees mij op een foutje in de voorbeeldcode in sectie 5.3.3, waar `pcinput` besproken wordt (opgelost in versie 1.0.12).
- Jaap van der Heide wees mij op een stukje onvertaalde tekst in hoofdstuk 4 (opgelost in versie 1.0.13).
- Patrick Vekemans wees mij op een fout in de code in subsectie 7.3.2 (opgelost in versie 1.0.13).
- Jose Perez-Carballo gaf aan dat de lijst van gereserveerde woorden die ik had opgenomen, de lijst was die voor Python 2 geldt, en die in Python 3 wijzigingen heeft ondergaan (opgelost in versie 1.0.13).
- Jos Kaats wees me op een foutje in mijn beschrijving van de aanroep van functies vanuit functies (opgelost in versie 1.0.14).
- Luis Mendo Tomas had een flink aantal opmerkingen die alle geleid hebben tot wijzigingen, onder andere wat betreft de beschrijving van default waardes voor functie parameters (versie 1.0.14).
- Sven de Windt wees me op een typo (opgelost in versie 1.0.15).
- Max Bierlee wees me op meerdere schrijffouten (opgelost in versie 1.0.15).

Inhoudsopgave

Hoofdstuk 1

Introductie

Computers zijn prachtige machines. De meeste machines (auto's, televisies, magnetrons) zijn gemaakt voor een specifiek doel dat ze effectief en efficiënt kunnen bereiken. Computers daarentegen zijn doelloze machines, die alles wat je maar wilt aangeleerd kunnen krijgen. De kunde die je in staat stelt om computers te laten doen wat je wilt, heet "programmeren."

In iedere wetenschappelijke richting en in elk beroep, moeten mensen vandaag de dag omgaan met grote hoeveelheden data. Zij die in staat zijn de kracht van computers in te zetten om gebruik te maken van die data, met andere woorden, zij die kunnen programmeren, zijn veel beter in staat hun beroep uit te oefenen dan zij die dat niet kunnen. Ik durf zelfs te stellen dat in de zeer nabije toekomst zij die niet kunnen programmeren niet meer arbeidsgeschikt zijn. Daarom zie ik het als een noodzaak dat iedereen tijdens zijn of haar opleiding leert programmeren.

Programmeren betekent niet alleen dat je weet wat programmeerregels doen; het betekent ook dat je kunt denken als een programmeur, en dat je problemen kunt analyseren vanuit het perspectief dat ze opgelost moeten worden door een computer. Deze vaardigheden kun je niet leren uit een boek. Je kunt ze alleen leren door daadwerkelijk programma's te maken.

Dit boek is geschreven om je de basis van de Python 3 computertaal te leren. Studenten leren met dit boek niet alleen de taal te gebruiken, maar doen er ook oefeningen mee.

Het boek is niet alleen bedoeld voor studenten die vanuit zichzelf al kunnen en willen programmeren. Het is ook en misschien zelfs vooral bedoeld voor hen voor wie programmeren een vreemde taak is. De teksten in dit boek zijn vaak uitgebreider dan je in andere boeken tegenkomt, en proberen problemen te voorzien die je tegen kunt komen wanneer je nieuwe concepten probeert te begrijpen.

1.1 Hoe dit boek te gebruiken

Dit boek is bedoeld als cursus. Het is niet direct bedoeld als naslagwerk voor de Python taal. Naslagwerken zijn niet nodig, een uitstekende taalreferentie voor Python kan op Internet gevonden worden (`http://docs.python.org`).

De hoofdstukken van dit boek zijn geschreven om in volgorde bestudeerd te worden. Voor een korte cursus Python, waarbij je je concentreert op "imperatief programmeren," moet je de volgende onderwerpen bestuderen: variabelen en expressies, condities en iteraties, functies, strings, lists en dictionaries, en bestandsverwerking. Met andere woorden, je kunt je beperken tot de hoofdstukken 1 tot en met 19, waarbij de hoofdstukken 9 (recursie), 14 (sets), 17 (exceptions), 18 (binaire bestanden), and 19 (bitsgewijze operatoren) beschouwd kunnen worden als optioneel materiaal, dat je kunt overslaan totdat je ze nodig hebt (waarbij ik wel aanbeveel dat je probeert recursie te begrijpen, aangezien het je helpt om sommige opgaven uit latere hoofdstukken op te lossen).

Voor een uitgebreidere cursus Python moet je in ieder geval ook object oriëntatie bestuderen, dat wil zeggen hoofdstukken 20 tot en met 23, waarbij hoofdstuk 23 (iteratoren en generatoren) beschouwd kan worden als geavanceerde stof.

De overige hoofdstukken zijn alle nuttig maar optioneel. Je kunt hiervan bestuderen wat je wilt, hoewel ik aanbeveel dat je ze op zijn minst doorbladert om te zien waar ze over gaan. Toekomstige edities van dit boek kunnen extra optioneel materiaal bevatten.

Tijdens het bestuderen van dit boek moet je een computer bij de hand houden waarop je Python hebt geïnstalleerd (hoofdstuk 2 legt uit hoe je Python kunt krijgen voor jouw computer). Het boek bevat vele kleine en grotere oefeningen, die je allemaal moet doen tijdens het bestuderen. Als je veel van de oefeningen overslaat, zul je zeker niet leren programmeren. Meer over het doen van oefeningen volgt later in dit hoofdstuk (1.9).

Veel van de code fragmenten in dit boek – in ieder geval alle antwoorden op de opgaves en alle iets langere fragmenten – hebben een bestandsnaam boven de code genoemd. Dat betekent dat die code beschikbaar is als bestand, verkrijgbaar via de website die met dit boek geassocieerd is (`http://www.spronck.net/pythonbook`). Je kunt de code downloaden en meteen in een editor laden als je dat wilt.

Let op! Het kopiëren en plakken van code vanuit een PDF bestand naar een editor werkt over het algemeen niet. Tekst in een PDF bestand is niet opgeslagen op een manier die ervoor zorgt dat spaties correct gekopieerd worden. Dus je moet ofwel code handmatig intypen, ofwel de bestanden gebruiken die ik beschikbaar heb gesteld.

1.2 Aannames en veronderstellingen

Dit boek veronderstelt dat je geen programmeervaardigheden hebt, maar dat je die wilt aanleren. Het veronderstelt ook dat je in ieder geval het vermogen hebt om abstract te denken.

Realiseer je dat om te leren programmeren je een flinke hoeveelheid tijd moet investeren. Je kunt niet volstaan met het materiaal doorlezen en hier en daar een kleine oefening doen. Je moet daadwerkelijk met de stof oefenen door ook de grotere opgaven te doen. Als je je beperkt tot de basishoofdstukken (alles tot en met het omgaan met tekstbestanden), en je hebt nog geen programmeerervaring, moet je rekenen op een tijdsinvestering van 100 tot 200 uur, afhankelijk van je aanleg. Als je alles wilt leren wat het boek aanbiedt, moet je rekenen op 200 tot 400 uur.

Dit boek leert je niet om een professioneel programmeur te zijn. Het leert je de initiële vaardigheden die iedere programmeur ook heeft aangeleerd tijdens zijn of haar opleiding.

Het boek eindigt nadat die eerste vaardigheden zijn bijgebracht. Voor de meeste mensen is dat voldoende om iedere programmeertaak die ze tegenkomen aan te pakken, en biedt voldoende basis om meer te leren wanneer dat nodig is.

Ik gebruik in het boek zoveel mogelijk de Nederlandse taal, maar ik kom er niet onderuit om zo nu en dan ook gebruik te maken van Engelstalige terminologie. Dat heeft drie redenen: (1) sommige termen zijn gewoon niet goed vertaalbaar (bijvoorbeeld "statement"), (2) sommige termen refereren aan Python taalelementen en die zijn Engelstalig (bijvoorbeeld "float," wat een gebroken getal is), en (3) voor sommige termen is het nu eenmaal zo dat hoewel er een Nederlandstalige variant is, de Engelstalige variant het meest gebruikt wordt (bijvoorbeeld "loop" in plaats van "lus," of "data" in plaats van "gegevens"). Voor dergelijke termen is het verstandig dat gewoon de Engelstalige variant gebruikt wordt, omdat die bij iedereen bekend is. Ik zal dit soort termen uitleggen de eerste keer dat ze gebruikt worden, maar later in het boek staan ze dan zonder verdere uitleg in de tekst. Ik som ze op in appendix G, die je eventueel erop na kunt slaan als je een term nogmaals wilt opzoeken.

1.3 Waarom Python?

Python wordt door velen gezien als een taal die bij uitstek geschikt is om mensen te leren programmeren. Het is een krachtige taal, die gemakkelijk te gebruiken is, en die alle mogelijkheden biedt die andere talen ook bieden. Je kunt Python programma's draaien op verschillende machines en verschillende besturingssystemen. Het is gratis verkrijgbaar. Voor beginnende programmeurs heeft het het voordeel dat het dwingt om nette code te schrijven. Python wordt ook in de praktijk veel gebruikt, soms als basis voor complete programma's, soms als uitbreiding op programma's die in een andere taal geschreven zijn.

Het belangrijkste voordeel is dat Python het mogelijk maakt om je te concentreren op "denken als een programmeur," in plaats van op alle excentrieke afwijkingen die een specifieke taal heeft. Hier volgt een voorbeeld van het verschil tussen Python en een aantal andere populaire programmeertalen: Het eerste programma dat iedereen schrijft in een computertaal is *Hello World*. Dit is een programma dat de tekst "Hello, world!" op het scherm zet. In de zeer populaire taal C++, wordt *Hello World* als volgt gecodeerd:

```
#include <iostream>
int main() {
    std::cout << "Hello, world!";
}
```

In C#, de populaire variant can C++ die uitgebracht is door Microsoft, is het:

```
using System;
namespace HelloWorld {
    class Hello {
        static void Main() {
            Console.WriteLine( "Hello, world!" );
            Console.ReadKey();
        }
    }
}
```

In Objective-C, Apple's variant op C++, is het nog erger:

```
#import <Foundation/Foundation.h>
int main ( int argc, const char * argv[] ) {
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSLog (@"Hello, world!");
    [pool drain];
    return 0;
}
```

In Java, dat voor veel studenten aan universiteiten en hogescholen de eerst-geleerde taal is, wordt het:

```
class Hello {
    public static void main( String[] args ) {
        System.out.println( "Hello, world!" );
    }
}
```

En zie nu wat *Hello World* in Python is:

```
print( "Hello, world!" )
```

Ik neem aan dat we het met elkaar eens zijn dat de Python versie van dit programma leesbaarder en begrijpelijker is dan de andere varianten – zelfs als je de taal niet kent.

1.4 Python's beperkingen als programmeertaal

Python is een universele programmeertaal. Dat betekent dat je het kunt gebruiken voor alles wat je maar met een programmeertaal zou willen of kunnen doen. Mag je dan concluderen dat als je eenmaal Python beheerst, je nooit meer een andere taal hoeft te leren?

Het antwoord is dat dat afhangt van wat je wilt bereiken. Python kun je inderdaad voor alles gebruiken, maar het is niet voor alles het meest geschikt. Bijvoorbeeld, game ontwikkelaars gebruiken meestal C++ of C#, omdat die talen heel snelle programma's opleveren, en snelheid is van groot belang voor games. Mensen die complexe statistische berekeningen moeten doen hebben ook hun eigen talen. Soms moet je programma's schrijven die moeten samenwerken met andere programma's die in een specifieke taal geschreven zijn, en moet je ook die taal gebruiken. En voor sommige programmeertaken zijn talen met een andere filosofie tot het schrijven van code het meest geschikt.

Samenvattend heeft Python op zich geen beperkingen, maar zijn voor specifieke problemen specifieke andere talen geschikter. Dat gezegd hebbende, voor de meeste mensen is kennis van Python voldoende om alles te doen wat nodig is voor hun studie of beroep. Daarbij komt dat als je eenmaal Python beheerst, je een sterke basis hebt om andere talen te leren. Daarom denk ik dat Python uitermate geschikt is om beginnelingen te leren programmeren.

1.5 Wat betekent "denken als een programmeur?"

Dit boek heeft niet alleen als doel om je Python te leren, maar ook om je te leren denken als een programmeur, omdat denken als een programmeur noodzakelijk is om te begrijpen waarvoor je computers kunt gebruiken en hoe je ze moet gebruiken. Maar wat betekent

het om te "denken als een programmeur"? Ik zal die vraag beantwoorden door hem te illustreren met een taak:

Stel je voor dat je een stapel kaarten hebt, en op iedere kaart staat een verschillend getal geschreven. Je moet deze kaarten sorteren van laag naar hoog, met het laagste getal bovenop. De meeste mensen kunnen die taak uitvoeren. Voor de meeste mensen geldt ook dat als je ze vraagt *hoe* ze de taak uitvoeren, ze je vragend aan zullen kijken en zeggen: "Eh..., ik sorteer ze van laag naar hoog... wat bedoel je met hoe ik dat doe?" Anderen zullen zeggen: "Ik zoek eerst de hoogste kaart en die leg ik onderop. Dan zoek ik de op-een-na-hoogste en die leg ik erbovenop. Enzovoorts." Dit legt min of meer uit hoe ze het doen, maar als je dan vraagt: "Maar hoe vind je de hoogste kaart?" zullen ze je ook vragend aankijken.

Het probleem is dat als je een computer moet uitleggen hoe een stapel kaarten gesorteerd moet worden, je niet kunt veronderstellen dat de computer iets kan met vage uitspraken, zelfs als die uitspraken voor mensen duidelijk zijn. Je kunt niet tegen de computer zeggen: "Zoek de hoogste kaart," want zelfs als de computer Nederlands zou begrijpen, dan zou hij nog vragen hoe hij de hoogste kaart moet vinden. Je moet heel expliciet zijn over de handelingen die moeten worden uitgevoerd. Je moet iets zeggen als: "Neem de bovenste kaart in je linkerhand. Doe dan het volgende totdat de stapel leeg is: Neem de bovenste kaart in je rechterhand. Als het getal op de kaart in je rechterhand hoger is dan het getal op de kaart in je linkerhand, dan stop je de kaart in je linkerhand in de aflegstapel en stopt de kaart in je rechterhand in je linkerhand. Anders stop je de kaart in je rechterhand in de aflegstapel. Als de stapel leeg is en je rechterhand ook leeg is, is de kaart in je linkerhand de hoogste kaart."

Natuurlijk heeft een computer geen handen en begrijpt hij ook geen Nederlands. Maar een computer begrijpt wel computertaal. Computertalen hebben een zeer precieze syntax[1] en een zeer precieze semantiek,[2] wat betekent dat een programma een ondubbelzinnige manier is om te beschrijven hoe een taak uitgevoerd moet worden. Dus om een computer een taak uit te laten voeren, moet je met behulp van een computertaal de computer stap voor stap uitleggen hoe de taak uitgevoerd moet worden. Slechts dan kan de computer de taak uitvoeren.

Omdat het vaak lastig is om alle stappen te overzien die een computer moet zetten om een taak uit te voeren, moet je de taak verdelen in kleinere subtaken, die je misschien ook weer moet opdelen in subtaken, totdat de subtaken zo klein zijn dat je gemakkelijk de stappen kunt zien die je nodig hebt om zulke subtaken uit te voeren. Je kunt dan ieder van de subtaken implementeren, en ze samenvoegen om een programma voor de grote taak te vormen.

Denken als een programmeur betekent dat je een taak kunt beschouwen vanuit het perspectief dat een computerprogramma geschreven moet worden om de taak op te lossen, dat je in staat bent een logische opdeling van een taak in subtaken te maken, en dat je kunt herkennen wanneer de subtaken klein genoeg zijn dat je ze kunt implementeren. Dit is een vaardigheid die de meeste mensen kunnen leren, maar die veel oefening vereist omdat er een denkproces voor nodig is dat anders is dan waar de meeste mensen aan gewend zijn.

Door te leren programmeren in Python, beginnend met kleine programma's die groeien in

[1] De syntax zijn de regels die beschrijven wat correct-gevormde zinnen in een taal zijn.

[2] De semantiek beschrijft de manier waarop syntactisch correct-gevormde zinnen geïnterpreteerd moeten worden.

complexiteit, breng je jezelf langzaamaan ook de denkprocessen bij die een programmeur van nature beheerst.

1.6 De kunst van het programmeren

Programmeren is een kunstvorm. Een docent programmeren is in veel opzichten vergelijkbaar met een tekenleraar.

Veel mensen krijgen tekenlessen op de middelbare school. Een tekenleraar vertelt eerst over materialen: potloden en papier, verschillende kleuren, verschillende hardheden, gummen, inkt, inktpennen, verf, etcetera. De leerlingen maken hun eerste tekeningen en schilderijen op basis van die kennis. De leraar legt technieken uit: het mengen van verf, het creëren van speciale effecten, de combinatie van technieken, het gebruiken van perspectief, etcetera. De leerlingen krijgen opdrachten als "teken een kat," en de leraar geeft een beoordeling zowel wat betreft het gebruik van materialen als in hoeverre wat door de leerlingen geproduceerd wordt daadwerkelijk lijkt op een kat.

Een docent programmeren heeft vergelijkbare taken. Eerst vertelt hij de studenten over programmeerprincipes, basis taalelementen die in iedere computertaal aanwezig zijn. Hij vertelt hoe die taalelementen gebruikt kunnen worden om eenvoudige programma's te schrijven. Daarna gaat hij dieper op de taal in, en bespreekt geavanceerde technieken die de studenten kunnen gebruiken om complexere programma's te maken, en lastige concepten op een eenvoudige manier in programma's op te nemen. Studenten krijgen opdrachten als "schrijf een programma dat een tekst alfabetiseert," en ze worden beoordeeld zowel op de mate waarin ze programmeertechnieken beheersen, als in hoeverre hun programma's in staat zijn de taak uit te voeren.

Een tekenleraar zal stellen dat een student die voor de opdracht "teken een kat" een cirkel met twee driehoekjes erboven en twee puntjes in het midden inlevert weliswaar een kat heeft getekend, maar niet de materiaaltechnieken beheerst. Een student die een prachtige tekening van een boom inlevert is misschien een meester in materiaaltechnieken, maar kan ze niet gebruiken om een taak uit te voeren. En twee studenten die precies dezelfde tekening van een kat inleveren, hebben overduidelijk plagiaat gepleegd. Dat gezegd hebbende, er is niet slechts één acceptabele tekening van een kat. Er zijn vele mogelijke tekeningen van een kat die leerlingen kunnen inleveren om aan te tonen dat ze op weg zijn artiesten te worden.

Op dezelfde wijze wil een docent programmeren die een opdracht geeft, zien dat zijn studenten de kennis die ze hebben opgedaan creatief inzetten om een versie van een programma te creëren dat aan de opdracht voldoet. Studenten die de technieken niet beheersen kunnen de taak niet oplossen, of kunnen slechts een gedeeltelijke oplossing inleveren. Studenten die de technieken wel beheersen kunnen nog steeds niet in staat zijn om wat ze hebben geleerd te combineren op nieuwe, originele manieren om een oplossing te produceren. En twee studenten die exact dezelfde oplossing inleveren, hebben die ergens vandaan gekopieerd en proberen weg te komen met plagiaat.

Programmeren is een kunstvorm, waarvoor je niet alleen technieken moet leren beheersen, maar ook moet leren die technieken op een creatieve manier in te zetten om problemen op te lossen. Het grootste verschil tussen het bouwen van programma's en het maken van schilderijen is dat je wat betreft schilderen nog kunt debatteren of een plaatje van een bulldog met puntoren volstaat als weergave van een kat, terwijl het bij programmeren

veel gemakkelijker is om programma's te diskwalificeren als oplossing voor een specifiek probleem.

Iedereen weet dat je nooit een goede artiest zult worden als je alleen materialen bestudeert. Je moet met de materialen oefenen, en je vaardigheden trainen door verschillende taken uit te voeren. Voor programmeren geldt precies hetzelfde: je kunt niet leren programmeren zonder een hoop programma's te schrijven. Programmeren vereist niet alleen kennis, maar ook vaardigheden die door toepassing ontwikkeld worden, en een soort creativiteit die je in staat stelt je vaardigheden uit te breiden en nieuwe taken aan te pakken.

Het ligt voor de hand dat er slechts weinig meester-kunstenaars zijn wiens werk in tentoonstellingen te zien is. Maar we kunnen allemaal tekeningen van een kat maken, en dat volstaat voor wat we dagelijks nodig hebben. Op dezelfde manier zijn slechts weinigen in de wieg gelegd om meester-programmeur te worden, maar dat is voor de meesten onder ons ook niet nodig zolang we maar in staat zijn om de programmeerproblemen op te lossen die we tegenkomen in studie en werk. Maar besef wel dat het niet volstaat om slechts de basistechnieken te beheersen: creativiteit is altijd nodig.

1.7 Groei van klein naar groot

Dit is niet het enig beschikbare Pythonboek, maar de meeste boeken die ik heb gezien veronderstellen behoorlijk wat achtergrondkennis en ervaring bij de studenten. Boeken voor beginners zijn zeldzaam. Toch zijn er diverse alternatieven voor dit boek, sommige zelfs gratis (hoewel bij mijn weten ze allemaal in het Engels geschreven zijn).

Een groot probleem dat ik heb met Pythonboeken voor beginners is dat ze programmeren aantrekkelijk proberen te maken door van meet af aan applicaties te maken die nuttig of leuk zijn. Een typische titel is "Leer Games Maken met Python!"

Een dergelijke aanpak is misleidend. Op de eerste plaats, als je zulke boeken doorneemt, zul je ontdekken dat er slechts eenvoudige woord- en getalspelletjes worden gebouwd, en niet de nieuwe *Halo*, *Civilization*, *Bejeweled*, of zelfs iets simpels als *Flappy Bird*. Dit is niet wat een student verwacht van een boek over games programmeren. Bovendien zul je ontdekken dat zelfs de simpele spelletjes die het boek gebruikt te lastig zijn voor nieuwelingen om te leren programmeren. Ik snap dat een dergelijk boek enthousiasme bij de studenten probeert te kweken door inherent aantrekkelijk materiaal aan te bieden. Maar zulk enthousiasme verdwijnt snel wanneer de studenten zich realiseren dat het materiaal niet is wat ze verwachtten, en op dat punt wordt het onderwerp meer een obstructie dan een stimulans.

Ongeacht hoe je tegen het onderwerp aankijkt, om te leren programmeren moet je starten met basisconcepten voordat je kunt overgaan naar aantrekkelijke en nuttige toepassingen. De wens of noodzaak om te leren programmeren moet de student motiveren, en niet de loze verwachting dat je na een paar uur studeren een leuk spel in elkaar kunt zetten. Daarom begin ik in dit boek klein, met basis programmeerkennis, en bouw gestaag op naar complexere zaken. Het boek blijft echter niet steken in de kleine dingen – als je het hele boek doorwerkt zul je aan het einde een vaardig programmeur zijn.

Door het hele boek heen zijn opgaves opgenomen, die ik aantrekkelijk heb proberen te maken voor de studenten die dit soort dingen leuk vinden. Sommige studenten hebben me verteld dat ze het plezierig vinden om eraan te werken. Ik heb echter ook veel studenten

gezien voor wie het haast een lijdensweg is, en die ernaar verlangen andere dingen te doen. Ongeacht hoe je hier tegenaan kijkt, als je wilt leren programmeren en je de opgaves maakt, zal het boek je alles leren wat je moet weten om iedere applicatie die je maar wilt te bouwen – zelfs leuke spelletjes als je dat aanspreekt.

1.8 Python 2 of Python 3?

Er bestaan verschillende versies van Python. De meest populaire versies zijn Python 2 en Python 3. Python 3 is, zoals je kunt verwachten, een update van Python 2. Python 2 programma's zijn helaas niet volledig compatibel met Python 3 (met andere woorden, je kunt Python 2 programma's over het algemeen niet draaien op een computer waarop je alleen Python 3 geïnstalleerd hebt). Omdat er nog steeds een hoop Python 2 programma's gebruikt worden, is Python 2 een taal die nog steeds onderhouden wordt.

Python 3 is gemaakt om een aantal inconsistenties en eigenaardigheden van Python 2 op te lossen. Voor studenten voor wie programmeren nieuw is, is dit een groot voordeel, omdat er minder "vreemde" taalelementen zijn die ze moeten leren als ze Python 3 verkiezen boven Python 2.

Om een voorbeeld te geven, als je 7/4 uitrekent in Python 2, krijg je als antwoord 1, en niet 1.75 wat je zou verwachten. De reden is dat zowel 7 als 4 gehele getallen zijn ("integers"), en daarom is de uitkomst van de deling ook een geheel getal. Als je 1.75 als uitkomst wilt hebben, moet je ervoor zorgen dat minstens een van de twee getallen een gebroken getal is. 7.0/4 geeft daarom als uitkomst 1.75. Dit is de manier waarop vrijwel alle computertalen omgaan met getallen. Voor studenten voor wie programmeren nieuw is, is dit contra-intuïtief. Python 3 heeft dit probleem opgelost, en doet de conversie naar gebroken getallen automatisch. Dus in Python 3 geeft 7/4 de uitkomst 1.75. Veel Python 2 programma's zijn echter geschreven onder de aanname dat een integer-deling naar beneden afrondt, wat betekent dat deze programma's niet meer correct functioneren als je ze uitvoert als Python 3 programma's. Dus zijn Python 2 en Python 3 niet compatibel.

Omdat Python 3 intuïtiever is dan Python 2, en omdat vandaag de dag de meeste Python programma's en modules op zijn minst geconverteerd zijn naar Python 3, is dit boek geschreven voor Python 3. Als je ooit terug moet naar Python 2, is dat niet moeilijk. De verschillen tussen Python 2 en Python 3 die ik ken heb ik beschreven in appendix B (dit is geen compleet overzicht). Als je alleen Python 3 wilt gebruiken, kun je die appendix laten voor wat het is. Maar omdat ik vaak de vraag "wat zijn precies de verschillen tussen Python 2 en Python 3" gesteld krijg, leek het me verstandig deze appendix op te nemen.

1.9 Oefening

De meeste hoofstukken hebben kleine opgaves op diverse plaatsen in de tekst. Die opgaves zijn bedoeld om iets uit te leggen of als een snelle test of je alles nog steeds begrijpt. Je moet die opgaves meteen doen als je ze tegenkomt. Ik geef zelden antwoorden voor die opgaves, want als je de stof snapt zijn ze erg eenvoudig, en als je ze niet kunt maken moet je de voorgaande tekst opnieuw bestuderen totdat je het wel kunt. Als het dan nog steeds niet lukt, kun je beter om hulp vragen.

Aan het einde van ieder hoofdstuk is een sectie getiteld "Opgaves," met daarin één of meerdere genummerde opgaves. Je moet al die opgaves proberen te maken, en je moet ze

zelfstandig maken (dus zonder hulp en zonder de antwoorden op te zoeken). Antwoorden voor die opgaves zijn opgenomen achterin het boek, in appendix F. Je kunt ze ook downloaden via `http://www.spronck.net/pythonbook`. Ik wil de volgende punten expliciet onder de aandacht brengen:

- Je moet aan de opgaves werken totdat je ze opgelost hebt. Het volstaat niet om een beetje te proberen en dan het antwoord op te zoeken. Een dergelijke aanpak is volstrekt zinloos. Je zult nooit leren programmeren als je niet nadenkt over oplossingsmethodieken, code schrijft, en code test. Als je een opgave niet kunt oplossen zelfs als je er lange tijd aan hebt gewerkt, doe je er beter aan hulp te vragen dan het antwoord op te zoeken. Het niet kunnen oplossen van een opgave betekent dat er iets in het materiaal zit dat je nog niet begrijpt, en het is belangrijk dat je ontdekt wat dat is, zodat je dat gebrek kunt verhelpen.
- Je moet alle opgaves maken. De enige manier om te leren programmeren is te oefenen. Je zult veel code moeten schrijven om de praktijk van het programmeren te internaliseren. De paar opgaves die ik aan het einde van ieder hoofdstuk op heb genomen zijn nog niet voldoende om dat te bereiken, maar ze zijn een begin. Als je niet de moeite neemt om al die opgaves te doen, hoef je ook niet de moeite te doen om te proberen programmeren te leren.
- Je moet de opgaves zelfstandig maken. Aan de opgaves werken in groepsverband laat één persoon leren terwijl de rest erbij zit en toekijkt. Studenten vertellen me vaak dat ze een leermethode hebben waarbij ze aan opdrachten werken in groepsverband en antwoorden bediscussiëren. Dat werkt misschien voor het analyseren van teksten en het opzetten van experimenten, maar werkt meestal niet voor coderen. Toekijken hoe iemand anders code schrijft leert je erg weinig over het schrijven van code. Je moet zelf code schrijven.
- Voor geen van de opgaves is informatie nodig die nog niet naar voren was gebracht op het moment dat de opgave in het boek verschijnt. Vaak zijn er wel betere manieren om een opgave op te lossen door gebruik te maken van Python constructies die later in het boek komen. Maar die zijn op zich niet nodig. Het doel van de opgaves is te oefenen met het materiaal dat je op dat moment kent. Ze zijn niet bedoeld om toekomstig materiaal te bestuderen. Zelfs als je bekend bent met andere manieren van oplossen, probeer je dan toch te beperken tot het materiaal dat voor de opgave staat. Als je dat eenmaal gedaan hebt, en je wilt later terugkeren bij een opgave om het op een andere manier te doen, is dat natuurlijk uitstekend.
- Als je een opgave hebt opgelost en je je oplossing goed getest hebt, kun je je oplossing vergelijken met het antwoord dat ik geef. Je zult vaak zien dat mijn antwoord anders is dan het jouwe. Dat betekent niet dat jouw antwoord fout is! Er zijn meestal veel verschillende manieren om een programmeerprobleem op te lossen. Sommige zijn "beter" in bepaalde opzichten dan andere. Vele zijn "precies even goed." Wat van belang is dat je een probleem kunt oplossen door code te schrijven, niet dat je een probleem kunt oplossen door de meest efficiënte code te schrijven. Het volstaat om het probleem op te lossen; oplossingen efficiënt maken is van veel minder belang. Bijvoorbeeld, "efficiëntie" is minder belangrijk dan "begrijpbaarheid" en "onderhoudbaarheid."

Hieronder staan alvast twee genummerde opgaves voor dit eerste hoofdstuk. Leer ervan.

Opgaves

Opgave 1.1 Ga samen zitten met een andere persoon, en doe de volgende opgave. Neem vier speelkaarten met verschillende waardes. Schud ze, en leg ze met de voorkant naar beneden op tafel. Eén van jullie moet ze sorteren, van laag naar hoog. Deze persoon mag de kaarten verplaatsen, maar mag niet kijken naar de voorkant van de kaarten. Hij of zij mag wel twee kaarten aanwijzen, die de ander dan oppakt, bekijkt, en terug legt, aangevend welke van de twee hoger is. Houd bij hoeveel van die vergelijkingen nodig zijn. Als de eerste persoon denkt dat de kaarten gesorteerd zijn, draai je ze om en controleert of het correct gedaan is.

De eerste person vervult de rol van het programma, dat instructies uitvoert zonder ze te begrijpen. De tweede persoon speelt de rol van processor, die bepaalde handelingen kan uitvoeren voor het programma, zoals, in dit geval, het vergelijken van getallen.

Als je niet in staat bleek de kaarten te sorteren, denk dan na over hoe je de taak kunt uitvoeren onder de gegeven omstandigheden, en probeer het dan nog eens. Als het je wel is gelukt, maar je had meer dan zes vergelijkingen nodig, denk dan na over hoe je het met zes kunt doen. Als je precies zes nodig had, denk dan na over of het ook kan met minder dan zes. Als je het met minder dan zes deed, bedenk dan of je aanpak inderdaad garandeert dat de kaarten gesorteerd worden.

Opgave 1.2 Nadat je de eerste opgave hebt gedaan, schrijf dan samen met je partner instructies die in gewoon Nederlands uitleggen hoe je onder de gegeven omstandigheden kaarten kunt sorteren. Haal er dan een derde persoon bij, en vraag die persoon de instructies te volgen, terwijl een van de eerste twee de rol van processor op zich neemt. Zeg dat de derde persoon de instructies zo exact mogelijk moet volgen, zonder ze te interpreteren. Deze oefening is het meest illustratief als de derde persoon geen idee heeft van wat de bedoeling van de instructies is. Als de poging tot sorteren gedaan is, controleer je of het resultaat correct is.

De tekstuele instructie is vergelijkbaar met een programma. Als de derde persoon de stappen niet kan volgen, lijkt dat op een syntax fout. Als de derde persoon de stappen kan volgen maar het resultaat is niet wat het moet zijn, is er een functionele fout gemaakt. Beide soorten fouten kom je tegen als je computers programmeert.

Opmerking: Het is best lastig dergelijke instructies te schrijven. Gelukkig is dit gemakkelijker in een computertaal, omdat syntax en semantiek van een computertaal strak gedefinieerd zijn. Het Nederlands is, net als iedere andere menselijke taal, niet geschikt om ondubbelzinnige instructies te schrijven.

Hoofdstuk 2

Python Gebruiken

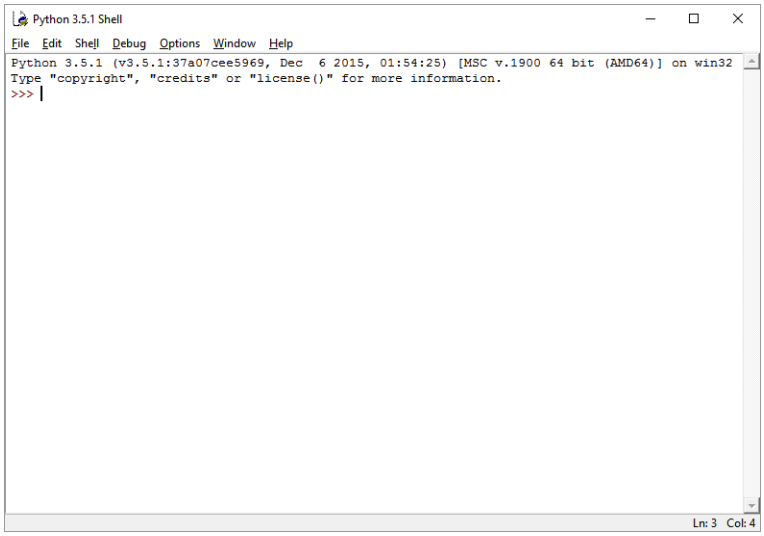
Zoals ik heb uitgelegd in de introductie, zul je Python code moeten schrijven om te kunnen leren met dit boek. Dat betekent dat je een computer nodig hebt waarop Python geïnstalleerd is, en je moet weten hoe je Python programma's kunt schrijven. Dit hoofdstuk legt uit hoe je Python aan het werk krijgt.

2.1 Python installeren

Om Python te gebruiken heb je een "Python interpreter" nodig. Python interpreters zijn gratis verkrijgbaar voor vrijwel alle computers. Via de website `http://www.python.org` kun je een Python interpreter downloaden voor jouw computer. Let erop dat je een Python 3 interpreter neemt. Installeer de interpreter door het bestand te openen dat je hebt gedownload. Nadat de installatie is afgelopen, ben je in principe klaar om Python programma's te schrijven en uit te voeren.

Het maakt niet uit wat voor operating system je gebruikt, of het nu Windows, Mac OS X, Linux, of iets anders is: voor iedere machine schrijf je dezelfde code. Je kunt zelfs een programma nemen dat je op de ene machine geschreven hebt en dat kopiëren naar een andere machine met een ander besturingssysteem, en waarschijnlijk zal het programma nog steeds werken (tenzij het programma systeem-specifieke code bevat, maar dat bespreek ik in een veel later hoofdstuk).

Er zijn Python cursussen die je gebruik laten maken van een online Python systeem, omdat ze weten dat het voor sommige mensen een obstakel is om software te installeren. Dat kan, maar het heeft drie nadelen: (1) er zijn gratis systemen die heel beperkt en dus nauwelijks bruikbaar zijn; (2) er zijn betaalde systemen die meer kunnen maar die geld kosten en vaak eigenaardigheden bevatten omdat ze nu eenmaal in een browser draaien; en (3) uiteindelijk moet je er toch toe overgaan om Python op je eigen computer te installeren, dus waarom zou je er niet meteen mee beginnen? Maar als je die nadelen voor lief wilt nemen, zou je kunnen beginnen met een online Python systeem, en pas in een later hoofdstuk overgaan op een lokaal geïnstalleerd systeem.



Afb. 2.1: De Python shell in IDLE.

2.2 Python programma's creëren

Python programma's zijn bestanden. Over het algemeen wordt voor Python bestanden de extensie `.py` gebruikt.

De meeste Python installaties (in ieder geval die voor Windows en Mac OS) installeren ook een omgeving om programma's in te schrijven, IDLE geheten. Ze zetten ook een icoon op het scherm en/of een optie in een start/programmamenu dat IDLE start. IDLE is nogal kaal, maar een geschikte omgeving om in te programmeren.

Als je IDLE start, ben je in een zogeheten "Python shell" (zie afbeelding 2.1) Dit is, als het ware, een interactief Python programma, waarin je regels code kunt typen die onmiddellijk worden uitgevoerd als je op `Enter` drukt. Bijvoorbeeld, als je **`print`**`(7/4)` typt, geeft IDLE meteen als antwoord 1.75. Dit is niet de manier waarop je wilt programmeren, maar je kunt wel op deze wijze snel het effect testen van regels Python code.

Om programma's met IDLE te creëren, kun je nieuwe Python bestanden aanmaken, of bestaande Python bestanden openen, via het "File" menu. IDLE geeft je dan een nieuw window waarin je code kunt schrijven, aanpassen, en opslaan. Je kunt zelfs code laten uitvoeren in dit window via het "Run" menu (er is ook een toets die je daarvoor kunt gebruiken, meestal `F5`). Het programma wordt dan uitgevoerd in de "Python shell," dus daar kun je dan input ingeven en output zien. Zorg ervoor dat je als bestandsnaam voor je programma's altijd iets kiest met de extensie `.py`.

Er zijn meer gebruikersvriendelijke manieren om Python programma's te schrijven. Je hebt dan een "tekst editor" nodig, bij voorkeur een editor die specifiek Python ondersteunt. Merk op dat een tekst editor niet hetzelfde is als een word processor (tekstverwerker); een tekst editor heeft geen manieren om tekst te formatteren. Als je code in een tekst editor typt, kan het zijn dat je wel formattering ziet, zoals vetgedrukte woorden en gekleurde woorden, maar dit is "syntax highlighting," een techniek die je laat zien dat je bepaalde programmeertaal elementen aan het schrijven bent.

Er zijn veel tekst editors beschikbaar die Python ondersteunen, voor verschillende besturingssystemen. Vele zijn gratis verkrijgbaar. Als je IDLE niet prettig vindt werken, kun je een alternatief vinden op Internet. Alle tekst editors hebben hun eigen voor- en nadelen, dus wat je prefereert mag je zelf uitmaken.

2.3 Python programma's uitvoeren

Wanneer je een Python programma geschreven hebt, zie je de naam van het programma in de folder waar je het hebt opgeslagen. Je kunt proberen het uit te voeren op dezelfde manier waarop je andere programma's start, bijvoorbeeld door erop dubbel te klikken. Voor veel Python programma's geldt dat als je ze op deze manier probeert te starten, je niks ziet gebeuren, of niet meer dan een flits van een zwart window, waarna er niks meer gebeurt. De reden is dat Python programma's worden uitgevoerd in een "command-line shell" van het besturingssysteem. Als je geen Linux gebruiker bent, is dat waarschijnlijk niet iets waar je mee bekend bent. Wat er gebeurt is dat Python de command-line shell opent, het programma draait, en als het programma afgelopen is, de shell weer sluit. Dat geeft je het gevoel dat er niks gebeurt, maar er is wel degelijk iets gebeurd – je zag alleen niet wat.

Voor het grootste deel van dit boek kun je je programma's gewoon draaien in de editor waarin je ze schrijft, zoals ik hierboven beschreef voor IDLE. Je kunt de command-line shell openen (dit is gewoonlijk een wat verborgen optie in de lijst van geïnstalleerde programma's) en de programma's "handmatig" draaien, maar er is zelden een reden om dat te doen.

2.4 Aanvullend materiaal

Naast dit boek zul je af en toe een Python referentie willen raadplegen. Dat gaat het gemakkelijkste via Internet. Zoek op "Python" met een korte omschrijving van wat je wilt, en je zult meteen links vinden die rechtstreeks naar de Python handboeken leiden (ze zijn te vinden op `http://docs.python.org`). Je kunt zelfs links vinden naar sites waarbij iemand precies uitlegt hoe je een programmeerprobleem oplost. Hoewel zulke sites heel prettig zijn als je Python problemen probeert op te lossen in de praktijk, leer je er weinig van. Dus ik raad je aan dit soort sites te vermijden zolang je aan het leren bent.

Wanneer je Python installeert, wordt er meestal ook een handboek geïnstalleerd in een `Doc` folder onder de Python folder. Die kun je gebruiken als je niet met Internet verbonden bent.

Als je nog een boek wilt gebruiken naast dit boek, en je hebt geen problemen met Engels, dan beveel ik het klassieke boek *Think Python: How to Think Like a Computer Scientist* van Allen B. Downey aan, gratis verkrijgbaar via `http://greenteapress.com/wp/`. Een Python 3 versie van dit boek is sinds 2016 beschikbaar. Het grootste verschil met mijn boek wat betreft inhoud is dat mijn boek meer opgaves bevat, meer gericht is op mensen voor wie programmeren volledig nieuw is, en een aantal belangrijke onderwerpen bespreekt die Downey weglaat, zoals een uitgebreide bespreking van bestandsverwerking.

Naast dit boek en soortgelijke boeken, bestaan er ook open video cursussen. Zelf geloof ik niet dat je goed kunt leren programmeren door naar een video te kijken. De enige manier om te leren programmeren is het te doen.

In dit boek heb ik ook een lijst met veelvoorkomende problemen die je kunt tegenkomen bij het schrijven en draaien van Python programma's, en mogelijke oplossingen voor deze problemen, opgenomen in appendix A).

Opgaves

Opgave 2.1 Download Python and installeer het op je computer. Start IDLE. Creëer een bestand `hello.py`, waarin je de code schrijft van het *Hello World* programma dat ik liet zien in hoofdstuk 1 – het bestaat uit één regel code, namelijk:

```
print( "Hello, world!" )
```

Draai het programma, en zie dat de tekst "Hello, world!" getoond wordt in de IDLE shell.

Opgave 2.2 In de IDLE shell kun je commando's typen op de "IDLE prompt" (>>>). Geef het commando **`print`**`(7/4)`. Je ziet dat de uitkomst 1.75 is. Geef daarna het commando `7/4` (dus zonder **`print`**). Zie nu dat het antwoord ook 1.75 is.

De reden dat je in het tweede geval ook het antwoord ziet, is dat de IDLE shell altijd de uitkomst van een commando laat zien. De uitkomst van `7/4` is 1.75, en dus laat IDLE 1.75 zien. De uitkomst van een **`print`** commando is niks, dus de shell laat niks zien – echter, het **`print`** commando zelf drukt de uitkomst af van wat er zich tussen de haakjes bevindt, en dat is het resultaat van wat je krijgt als je 7 deelt door 4, dus 1.75. Daarom zie je in beide gevallen 1.75, maar de eerste is het resultaat van het gebruik van het **`print`** commando, terwijl het tweede het resultaat is van de shell die laat zien wat de evaluatie van een berekening is.

Schrijf nu een Python programma (dus in een bestand met de extensie `.py`) dat alleen het commando `7/4` bevat. Voordat je dit programma uitvoert, bedenk wat je verwacht dat er gebeurt als je het uitvoert. Zal de shell 1.75 laten zien? Of laat de shell niks zien? Of krijg je een foutmelding?

Controleer of je verwachting uitkomt.

Hoofdstuk 3

Expressies

Welkom bij het eerste echte programmeerhoofdstuk. In dit hoofdstuk bespreek ik "expressies," die je kunt beschouwen als berekeningen die je ook kunt uitvoeren met een simpele rekenmachine. Het is een klein begin, maar deze expressies ga je nodig hebben in alle volgende hoofdstukken.

3.1 Resultaten tonen

Als je een expressie schrijft in de Python shell, en hem uitvoert, wordt het resultaat van de expressie eronder getoond. Bijvoorbeeld, als je het volgende commando in de shell schrijft en op `Enter` drukt, zie je het resultaat 12.

```
5 + 7
```

Echter, zoals ik liet zien in opgave 2.2, als je een programma schrijft dat alleen het commando `5 + 7` bevat, dan zie je geen resultaat. In plaats daarvan moet je voor programma's altijd expliciet aangeven dat je resultaten wilt tonen, zelfs als het gaat om een commando dat op de laatste regel van het programma staat.

Ook al gaat dit hoofdstuk over expressies, ik moet toch eerst iets uitleggen dat geen expressie is, maar een functie, namelijk een functie die het mogelijk maakt resultaten te laten zien. De functie die dat doet is **`print`**. Ik heb deze functie al gebruikt in hoofdstukken 1 en 2.

De **`print`** functie gebruik je als volgt: je schrijft het woord **`print`**, gevolgd door een rond openingshaakje, gevolgd door hetgeen je wilt laten zien, gevolgd door een rond sluithaakje. Bijvoorbeeld (en ook dit commando heb ik meerdere malen laten zien):

```
print( "Hello, world!" )
```

Als je deze code draait (door het op te slaan in een Python bestand en het te draaien in IDLE), zie je dat de tekst "Hello, world!" in de shell wordt getoond.

Overigens is het gebruikelijk dat wanneer auteurs van teksten over programmeren een functie bij naam noemen, ze er een openings- en sluithaakje achter zetten, om aan te geven dat het een functienaam betreft. Vanaf nu zal ik dat ook doen. Bovendien gebruiken auteurs soms niet het woord "functie," maar het woord "statement" of "commando." Deze termen duiden echter meestal op alles wat Python kan uitvoeren, niet alleen functies. Bijvoorbeeld, een expressie is ook een "commando."

Je kunt meerdere dingen tegelijkertijd laten zien met een **`print`**`()` functie, door alles wat je wilt laten zien tussen de haakjes te zetten, met komma's ertussen. De **`print`**`()` functie laat dan al die items zien, met spaties ertussen. Bijvoorbeeld:

```
print( "Ik", "heb", "twee", "appels", "en", "een", "banaan" )
```

De spaties in dit commando zijn overigens overbodig.

```
print("Ik","heb","twee","appels","en","een","banaan")
```

is precies hetzelfde als het commando ervoor. Je kunt zulke spaties toevoegen om de leesbaarheid te vergroten. Je mag ook spaties zetten tussen het woord **`print`** en het openingshaakje, maar de gewoonte is om bij functies altijd het openingshaakje aan de naam van de functie "vast te plakken."

Merk op dat je met **`print`**`()` niet alleen teksten, maar ook getallen kunt laten zien. Je mag zelfs teksten en getallen door elkaar gebruiken.

```
print( "Ik", "heb", 2, "appels", "en", 1, "banaan" )
```

Opgave Laat een paar teksten zien met behulp van een Python programma. Let erop dat je alle teksten tussen aanhalingstekens moet plaatsen; je mag naar keuze dubbele of enkele aanhalingstekens gebruiken.

3.2 Data types

Voordat ik toekom aan expressies, is er nog een onderwerp dat aan bod moet komen, en dat is data types. Er zijn drie verschillende data types die je op dit moment moet kennen, en dat zijn strings, integers, en floats.

3.2.1 Strings

Een string is een tekst, die bestaat uit nul of meer tekens, omsloten door aanhalingstekens. Je mag dubbele of enkele aanhalingstekens gebruiken. Wat je kiest maakt niet uit, bijvoorbeeld: `"appel"` is equivalent met `'appel'`. Echter, als je tekst een enkel aanhalingsteken bevat, moet je hem om problemen te voorkomen omsluiten met dubbele aanhalingstekens; dus `"mango's"` is een correcte string, terwijl `'mango's'` niet correct is. Hetzelfde geldt natuurlijk voor een dubbel aanhalingsteken in een string, die dan omsloten moet worden door enkele aanhalingstekens.

Maar wat moet je doen als een string zowel dubbele als enkele aanhalingstekens bevat? Je kunt dat oplossen door in de string een "backslash" (\) op te nemen voor ieder dubbel of

enkel aanhalingsteken dat in de string staat. Dit vertelt Python dat dat aanhalingsteken behandeld moet worden als een teken in de string, en niet als een afsluiting van de string. Dus `'mango\'s'` is een correcte string, wat je kunt zien als je hem probeert te printen:

```
print( 'mango\'s' )
```

Maar wat moet je dan doen als je een echte backslash wilt opnemen in de string, en die backslash moet dan ook nog eens staan voor een dubbel of enkel aanhalingsteken? Dat kun je oplossen door voor de backslash een extra backslash op te nemen, wat er een teken in de string van maakt in plaats van een aanduiding dat wat erachter komt letterlijk genomen moet worden. Bijvoorbeeld, controleer wat de volgende code doet als je hem uitvoert (je kunt hem intypen in de Python shell):

```
print( 'mango\\\'s' )
```

Als je dit verwarrend vindt kun je het voorlopig vergeten; ik ga het hierover in een later hoofdstuk nog uitgebreid hebben. Voor dit moment is het voldoende om te weten dat een string een tekst is die omsloten is door dubbele of enkele aanhalingstekens. Een string kan iedere lengte hebben, inclusief nul tekens lang.

Let erop dat je alleen “rechte” aanhalingstekens gebruikt in Python programma’s, en niet “ronde.” Tekstverwerkers hebben de neiging om rechte aanhalingstekens automatisch te vervangen door ronde, maar Python herkent zulke ronde aanhalingstekens niet. Tekst editors doen dat niet, maar als je om wat voor reden dan ook Python code kopieert naar een tekstverwerker, dan zou het best kunt gebeuren dat je aanhalingstekens gewijzigd worden. Kijk daarvoor uit.

3.2.2 Integers

Integers zijn gehele getallen, die positief of negatief (of nul) kunnen zijn. Er is een zekere grens aan hoe groot integers kunnen worden, die afhankelijk is van je computer en besturingssysteem. Voor de meeste toepassingen maakt dit niet uit, en kom je nooit aan die grenzen toe. Python is niet als rekenmachines met een 10-cijfer display.

Er zijn meerdere manieren mogelijk om een specifieke integer-waarde te schrijven. `1` is hetzelfde als `+1` (er zijn nog meer manieren om `1` te schrijven, maar die volgen in een later hoofdstuk). Dus zowel `print( 1 )` als `print( +1 )` geven hetzelfde resultaat. Voor strings is dat natuurlijk anders: de string `"1"` is niet hetzelfde als de string `"+1"`.

Als je integers in Python gebruikt, mag je ze niet schrijven met scheiders tussen de veelvouden van 1000 om ze leesbaarder te maken. Je moet het getal 1 miljard dus schrijven als `1000000000` en niet als `1,000,000,000` (de Engelse conventie) of `1.000.000.000`.

Bestudeer de volgende code en bedenk wat er gebeurt als je hem uitvoert. Kopieer hem daarna naar de Python shell en voer hem uit.

```
print( 1,000,000,000 )
```

Opgave Als je voorspelling van wat de code doet niet correct is, probeer dan te bedenken waarom de code deze uitkomst geeft.

3.2.3 Floats

Floats, wat kort is voor "floating-point getallen" ("gebroken getallen"), zijn getallen met decimalen. Bijvoorbeeld, `3.14159265` is een float. Merk op dat je een punt in plaats van een komma moet gebruiken als decimaal-scheider. Veel landen (inclusief Nederland) gebruiken een komma als decimaal-scheider, maar Python houdt zich aan de conventie van Engelstalige landen en gebruikt daarom de punt.

Als je een integer hebt die je wilt gebruiken als float, kun je dat doen door er `.0` achter te zetten. Bijvoorbeeld, `13` is een integer, maar `13.0` is een float. Ze geven nog steeds dezelfde waarde weer, en als je ze in code met elkaar vergelijkt (dat bespreek ik in hoofdstuk 6), dan zal Python stellen dat ze hetzelfde zijn.

Er zijn bepaalde begrenzingen aan de grootte van de floats, en aan de precisie. Het is onwaarschijnlijk dat je ooit de maximale groottes bereikt, aangezien Python wetenschappelijke notatie voor grote getallen gebruikt. Maar als je Python gebruikt voor zeer exacte berekeningen, kun je wel in de problemen komen met precisie. Voor de meeste toepassingen gebeurt dat niet, maar als je een natuurkundige bent wiens berekeningen grote getallen omvatten op quantumniveau, moet je je wel bewust zijn van deze beperkingen.

Door de manier waarop Python floats opslaat, kunnen bepaalde getallen niet precies vastgelegd worden. Bijvoorbeeld, het statement `print( (431 / 100) * 100 )` geeft als antwoord 430.99999999999994, en niet 431 zoals je zou verwachten. Als je weet dat de uitkomst van een berekening waarin floats zitten een integer moet zijn, dan doe je er goed aan om de uitkomst af te ronden naar het dichtstbijzijnde gehele getal. Dat kun je doen met behulp van de `round()` functie, die ik bespreek in hoofdstuk 5.

3.3 Expressies

Dan kan nu eindelijk het onderwerp van dit hoofdstuk onder de loep genomen worden, namelijk "expressies." Een expressie is een combinatie van één of meerdere waardes (zoals strings, integers, of floats) met behulp van operatoren, die dan een nieuwe waarde oplevert. Je kunt je expressies dus voorstellen als berekeningen.

3.3.1 Eenvoudige berekeningen

Eenvoudige berekeningen worden gemaakt door twee waardes te combineren met een operator ertussenin. Een aantal voor de hand liggende operatoren zijn:

```
+   optelling
-   aftrekking
*   vermenigvuldiging
/   deling
//  integer deling
**  machtsverheffing
%   modulo
```

Hier volgen een paar voorbeelden:

```
print( 15+4 )
print( 15-4 )
```

```
print( 15*4 )
print( 15/4 )
print( 15//4 )
print( 15**4 )
print( 15%4 )
```

Ik neem aan dat je weet wat deze operatoren voorstellen, behalve misschien de integer deling en de modulo.

De integer deling (ook wel genoemd "floor division") is simpelweg een deling die naar beneden afrondt naar een geheel getal. Als er floats in de berekening zitten, is het resultaat nog steeds een float, maar naar beneden afgerond. Als de berekening alleen integers omvat, is het resultaat een integer.

De modulo operator (%) produceert de rest die overblijft na deling. Bijvoorbeeld: als ik 14 deel door 5, is de uitkomst 2.8. Dat betekent dat ik twee keer 5 kan aftrekken van 14, en dan nog steeds een positief getal overhoud, maar als ik het een derde keer aftrek wordt het resultaat negatief. Dus als ik 5 twee keer aftrek van 14, rest er een getal kleiner dan 5. Deze rest is wat de modulo operator oplevert.

In eenvoudige termen: als ik 14 koekjes heb die ik moet verdelen over 5 kinderen, kan ik ieder kind 2 koekjes geven. Ik heb dan nog 4 koekjes over, omdat ik dan meer kinderen dan koekjes heb. Dus als je 14 deelt door 5 met integer deling, geeft dat 2 (koekjes per kind), terwijl 14 modulo 5 als rest 4 (koekjes in mijn hand) geeft.

Als zijdelingse opmerking: de code hierboven bestaat uit meerdere regels. Iedere regel is één "statement," bestaande uit een commando dat Python uitvoert (in de code hierboven is dat voor iedere regel een `print()` commando). De meeste programmeertalen stellen het als een verplichting dat ieder statement eindigt met een speciaal teken, bijvoorbeeld een puntkomma (;). Python verlangt dat niet, maar dan moet ieder statement ook op zijn eigen regel staan. In principe mag je meerdere Python statements op één regel zetten, maar dan moeten er puntkomma's tussen de statements staan. In de praktijk doen Python programmeurs dat niet, omdat het code lelijk, slecht leesbaar, en slecht onderhoudbaar maakt. Dus ik stel voor dat je de Python-gewoonte volgt en ieder statement zijn eigen regel geeft.

3.3.2 Complexe berekeningen

Je mag waardes en operatoren combineren om grotere berekening te maken, net zoals je kunt met geavanceerde rekenmachines. Je mag daarbij haakjes gebruiken om de evaluatievolgorde te bepalen, en je mag die haakjes zelfs nesten. Zonder haakjes zal Python de operatoren evalueren in de volgorde die wiskundigen gebruiken, waarvoor op basisscholen vaak de zin "Meneer Van Dalen Wacht Op Antwoord" wordt gebruikt (machtsverheffen, vermenigvuldigen, delen, worteltrekken, optellen, aftrekken).[3]

Bekijk de berekening hieronder en probeer te bepalen wat de uitkomst is voordat je hem uitvoert in de Python shell.

[3] In het Engels wordt de volgorde PEMDAS genoemd, en eigenlijk geeft die beter aan wat daadwerkelijk de volgorde is: haakjes (parentheses), exponenten, multiplicatie en divisie (deling), additie (optelling) en subtractie (aftrekking). Feitelijk is de Nederlandse volgorde incorrect, omdat worteltrekken na delen wordt geplaatst, terwijl wiskundig gezien worteltrekken een vorm van machtsverheffen is, en dus voor vermenigvuldigen komt.

```
print( 5*2-3+4/2 )
```

Ik moet een paar opmerkingen maken over deze berekening.

Op de eerste plaats valt het op dat de uitkomst een float is (zelfs al zijn er geen decimalen). De reden is dat er een deling in de berekening zit, en dat betekent voor Python dat de uitkomst automatisch een float is.

Op de tweede plaats is het zo dat, zoals ik al eerder opmerkte, spaties door Python genegeerd worden. De code hierboven is dus equivalent met:

```
print( 5 * 2 - 3 + 4 / 2 )
```

Het is zelfs gelijk aan:

```
print( 5*2 - 3+4    / 2 )
```

Ik heb lange discussies moeten voeren met mensen die beweren dat de code hierboven als uitkomst 6.5 of 1.5 geeft, want het is toch *overduidelijk* dat je eerst $5 * 2$ en $3 + 4$ moet berekenen voordat je toekomt aan die aftrekking en deling. Dat is kolder. Het maakt niet uit hoeveel spaties je rondom de operatoren zet: spaties worden genegeerd. Als je echt eerst $3 + 4$ wilt laten berekenen, moet je er haakjes omheen zetten. Spaties kunnen leesbaarheid verhogen als je ze goed toepast, maar voor Python zijn ze betekenisloos.

```
print( (5*2) - (3+4)/2 )
print( ((5*2)-(3+4)) / 2 )
```

Opgave Nu is de tijd gekomen om je eerste programma te schrijven. Schrijf een programma dat het aantal seconden in een week berekent. Je moet daarvoor natuurlijk niet je rekenmachine of smartphone grijpen, daarop de berekening doen, en dan gewoon de uitkomst afdrukken. Je moet de berekening doen in Python code. Het programma is slechts één regel lang, dus je kunt het gewoon in de Python shell doen, maar ik raad je aan om echt een programmabestand te maken.

3.3.3 String expressies

Een aantal van de hierboven genoemde operatoren kunnen voor strings worden gebruikt, maar niet allemaal.

Specifiek, je kunt de plus ($+$) gebruiken om twee strings aan elkaar te "plakken," en je kunt de ster ($*$) gebruiken met een string en een integer om een string te maken die een herhaling van de originele string bevat. Zie hier:

```
print( "tot"+"ziens" )
print( 3*"hallo" )
print( "tot ziens"*3 )
```

Je kunt geen getal optellen bij een string, of twee strings met elkaar vermenigvuldigen. Zulke operatoren zijn niet gedefinieerd, en geven foutmeldingen. Geen van de andere operatoren werkt voor strings.

3.3.4 Type casting

Soms moet je een data type of waarde veranderen in een ander data type. Je kunt dat doen met "type casting" functies.

In latere hoofdstukken (5 en 8) ga ik in detail op functies in, maar voor nu is het voldoende als je weet dat een functie een naam heeft, en parameters kan hebben tussen de haakjes die achter de naam staan. De functie doet iets met die parameters en geeft een resultaat terug. Bijvoorbeeld, de functie **`print()`** drukt de parameter waardes af op het scherm, en geeft niks terug.

Type casting functies nemen de parameter waarde die tussen de haakjes is gegeven, en geven een waarde terug die (bijna) hetzelfde is als de parameter waarde, maar van een verschillend type. De drie belangrijkste type casting functies zijn:

- **`int()`** geeft de parameter waarde terug als integer (indien noodzakelijk afgerond naar beneden)
- **`float()`** geeft de parameter waarde terug als float (waarbij .0 indien noodzakelijk wordt toegevoegd)
- **`str()`** geeft de parameter waarde terug als string

Bekijk de verschillen tussen de volgende twee regels code:

```
print( 15/4 )
print( int( 15/4 ) )
```

Of de volgende twee regels:

```
print( 15+4 )
print( float( 15+4 ) )
```

Ik had al aangegeven dat je de plus-operator niet kunt gebruiken om een getal aan een string vast te plakken. Als je zoiets toch wilt doen, kun je een oplossing maken met behulp van string type casting:

```
print( "Ik heb " + str( 15 ) + " appels." )
```

3.4 Stijl

Misschien is het je opgevallen dat ik in mijn voorbeelden veel spaties gebruik. Bijvoorbeeld, bij de haakjes die achter een functienaam staan, zet ik altijd een spatie na het openingshaakje en voor het sluithaakje. In berekeningen zet ik vaak spaties rondom operatoren als dat de berekening beter leesbaar maakt. Ik zet ook vaak lege regels in mijn code, en gebruik consequent vier spaties als tabulatie waar nodig.

De meeste van deze zaken zijn gewoon "stijl." De spaties bij de haakjes en rond de operatoren zijn niet nodig. Python begrijpt de code ook prima als ze er niet staan. De volgende vier regels code zijn equivalent:

```
# Equivalente regels code
print( 2 + 3 )
print(2+3)
print( 2+3)
print       (       2       +       3       )
```

Het "vastplakken" van het openingshaakje aan de functienaam doet vrijwel iedere programmeur, maar de rest verschillen de stijlen van spaties plaatsen tussen programmeurs (bijvoorbeeld, mijn stijl waarin ik een spatie plaats voor een sluithaakje is hoogst zeldzaam). Je kunt hierin je eigen stijlvoorkeur gebruiken, je hoeft niet de mijne te volgen. Maar ik raad je wel aan om een consistente stijl te gebruiken, want dat maakt je code leesbaarder, zelfs voor programmeurs die er een andere stijl op nahouden.

Merk op dat de code hierboven een "hash mark" (#; hiervoor bestaat geen Nederlandstalig woord) heeft op de eerste regel, waarna een tekst volgt die wat details over de code uitlegt. Deze regel is een commentaarregel: als je een hash mark gebruikt in je code (behalve als die in een string staat, natuurlijk) dan is alles wat rechts van de hash mark staat op de regel commentaar, dat door Python genegeerd wordt. Je kunt commentaar gebruiken om details over je code te geven, als je denkt dat uitleg nodig is. Meer over het geven van commentaar volgt in een later hoofdstuk.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Het gebruik van de **`print`**`()` functie om zaken op het scherm te tonen
- Data types string, integer, en float
- Berekeningen
- Basale string expressies
- Type casting tussen strings, integers, en floats, middels **`str`**`()`, **`int`**`()`, en **`float`**`()`

Opgaves

Opgave 3.1 Een boek kost in de winkel €24,95, maar boekwinkels krijgen 40 procent korting bij inkoop. Het verschepen van boeken kost €3 voor het eerste boek, en 75 cent voor ieder volgende boek. Bereken hoeveel de winkel betaalt voor 60 boeken.

Opgave 3.2 Kun je de fouten in de volgende regels code identificeren? Verbeter ze.

exercise0302.py

```
print( "Een boodschap" ).
print( "Een boodschap' )
print( 'Een boodschapf"' )
```

Opgave 3.3 Als er iets fout zit in code, geeft Python meestal een foutmelding. Dit zijn vaak "syntax fouten," die aangeven dat er iets fout zit in de vorm van je code (bijvoorbeeld, voor de code hierboven werden `SyntaxError`s gerapporteerd). Er zijn ook "runtime errors," die aangeven dat je code op zich syntactisch correct lijkt, maar dat er iets fout is gegaan bij de uitvoering ervan. Een goed voorbeeld is de `ZeroDivisionError`, die aangeeft dat je probeerde te delen door nul (wat niet mag, zoals je weet). Schrijf een kort programma dat zo'n fout genereert als je het uitvoert.

Opgave 3.4 Hier is een ander illustratief voorbeeld van een runtime error. Voer het programma uit en bestudeer de fout die gemeld wordt. Kun je het probleem oplossen?

exercise0304.py

```
print( ((2*3)/4 + (5-6/7)*8 )
print( ((12*13)/14 + (15-16)/17)*18 )
```

Opgave 3.5 Je kijkt op de klok en je ziet dat het 14.00u is. Je zet een alarm dat 535 uur later af moet gaan. Hoe laat is het als het alarm afgaat? Schrijf een programma dat het antwoord afdrukt. Hint: Gebruik de modulo operator.

Hoofdstuk 4

Variabelen

Als je met code werkt, ben je vaak bezig met het ontwerpen van een procedure (of "algoritme") dat een probleem op een algemeen toepasbare manier oplost. Bijvoorbeeld, in opgave 3.1 moest je de prijs van 60 boeken uitrekenen. De code die je schreef lost het probleem alleen op voor precies 60 boeken voor een bepaalde prijs. Als je een dergelijk probleem algemener wilt oplossen, moet je variabelen gebruiken om waardes in op te slaan.

4.1 Variables en waardes

Een variabele is een plaats in het geheugen van de computer die een naam heeft gekregen, en waarin je een waarde kunt opslaan. De naam mag je zelf kiezen, en wordt over het algemeen de "variabele naam" genoemd.

Om een variabele te creëren in Python (dus om een variabele naam te kiezen) moet je een waarde "toekennen" aan gekozen naam middels het is-gelijk (=) symbool. Aan de linkerkant van het is-gelijk symbool zet je de variabele naam, en aan de rechterkant de waarde die je wilt opslaan in de variabele. Dit kan ik het beste uitleggen aan de hand van een voorbeeld:

```
x = 5
print( x )
```

In deze code gebeuren twee dingen. Ten eerste wordt er een variabele gecrëerd met de naam `x` door er een waarde in op te slaan, in dit geval de waarde 5. In het Engels heet dit een "assignment", en het is-gelijk teken wordt ook wel de "assignment operator" genoemd. Ten tweede wordt de inhoud van de variabele `x` op het scherm getoond middels **`print()`**. Merk op dat **`print( x )`** niet de letter `x` toont, maar de waarde die in de variabele `x` is opgeslagen.

Je kunt je de variabele `x` voorstellen als een doos waarop je met een dikke, zwarte viltstift een `x` hebt geschreven, zodat je hem later gemakkelijk terug kunt vinden. Je kunt iets in de doos stoppen, en je kunt in de doos kijken om te zien wat er in zit (er kan wel slechts één ding tegelijkertijd in de doos zitten). Je kunt aan de inhoud van de doos refereren door de naam te gebruiken die je op de doos hebt geschreven. De term "variabele" duidt op de

variabele naam, dus de letter x op de doos. De term "waarde" duidt op de waarde die is opgeslagen in de variabele, dus de inhoud van de doos.

Aan de rechterkant van het is-gelijk teken kun je alles plaatsen wat een waarde oplevert. Het hoeft niet een enkel getal te zijn. Het mag ook een berekening zijn, of een string, of een aanroep van een functie die een waarde oplevert (bijvoorbeeld de **int()** functie).

Opgave In het vorige hoofdstuk was er een oefenopgave die je het aantal seconden in een week liet uitrekenen. Kopieer die berekening in een programma, en ken hem toe aan een variabele. Druk dan de variabele af.

De eerste keer in je programma dat je een waarde toekent aan een specifieke variabele naam, wordt de bijbehorende variabele gecreëerd. Als je later een andere waarde aan dezelfde variabele toekent, wordt de eerste waarde "overschreven." In de metafoor van de doos: je maakt de doos leeg en stopt er iets nieuws in. Een variabele bevat altijd de laatst-verkregen waarde.

```
x = 5
print( x )
x = 7 * 9 + 13   # overschrijft de vorige waarde van x
print( x )
x = "En nu iets heel anders..."
print( x )
x = int( 15 / 4 ) - 27
print( x )
```

Als een variabele is aangemaakt (en dus een waarde heeft) kun je hem overal in je code gebruiken waar je waardes gebruikt. Je kunt bijvoorbeeld de variabele gebruiken in een berekening.

```
x = 2
y = 3
```

```
print( "x =", x )
print( "y =", y )
print( "x * y =", x * y )
print( "x + y =", x + y )
```

Je kunt de inhoud van een variabele kopiëren in een andere variabele, via de assignment operator.

```
x = 2
y = 3
print( "x =", x, "en y =", y )

# Verwissel de waardes van x en y via z
z = x
x = y
y = z
print( "x =", x, "en y =", y )
```

Als je een waarde toekent aan een variabele, mag je zelfs de variabele zelf gebruiken aan de rechterkant van de toekenning, zolang de variabele maar bestaat op het moment dat je dat doet. De rechterkant van een assignment wordt altijd geheel geëvalueerd voordat de toekenning plaatsvindt.

```
x = 2
print( x )
x = x + 3
print( x )
```

Merk op dat de variabele gecreëerd moet zijn voordat je hem kunt gebruiken. De volgende code geeft een fout, omdat `dagen_per_year` nog niet gecreëerd is voordat hij gebruikt wordt in de eerste regel:

```
print( dagen_per_jaar )
dagen_per_jaar = 365
```

4.2 Variabele namen

Tot op dit punt heb ik slechts variabelen `x`, `y`, en `z` gebruikt (en een foute `dagen_per_jaar`). Je bent echter vrij om variabele namen te kiezen die je wilt, als je je daarbij maar aan een aantal eenvoudige regels houdt:

- Een variabele naam mag slechts bestaan uit letters, cijfers, en "underscores" (_)
- A variabele naam moet beginnen met een letter of een underscore.
- A variabele naam mag geen gereserveerd woord zijn

"Gereserveerde woorden" (of "keywords") zijn:

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Je mag zowel hoofd- als kleine letters gebruiken in variabele namen, maar je moet wel beseffen dat Python "case sensitive" is, dus gevoelig voor de verschillen tussen hoofd- en kleine letters. Bijvoorbeeld, de variabele naam `wereld` is voor Python niet hetzelfde als de variabele naam `Wereld`.

4.2.1 Conventies

Programmeurs houden zich aan een flink aantal conventies wanneer ze variabele namen kiezen. De belangrijkste zijn de volgende:

- Programmeurs kiezen *nooit* een variabele naam die ook de naam is van een functie (of het nu een standaard Python functie betreft of een functie die ze zelf hebben geschreven). Als je dat doet, loop je de kans dat de functie niet langer door de code gebruikt kan worden, wat kan leiden tot uitermate vreemde fouten.
- Programmeurs proberen variabele namen zo te kiezen dat ze betekenisvol zijn in de context van het programma. Bijvoorbeeld, een variabele die het aantal seconden in een week bijhoudt, zou de naam `secs_per_week` kunnen hebben, maar niet `ik_haat_mijn_baan`. Het zou nog erger zijn om het aantal seconden in een week op te slaan in een variabele `secs_per_maand`.
- Een uitzondering op het kiezen van betekenisvolle variabele namen is het kiezen van namen voor "wegwerp variabelen," dat wil zeggen, variabelen die slechts in een klein deel van de code gebruikt worden, naderhand niet meer gebruikt worden, en die van zichzelf eigenlijk geen betekenis hebben. Programmeurs kiezen vaak één-letter namen voor dit soort variabelen, zoals `i` of `j`.
- Om verwarring tussen hoofd- en kleine letters te vermijden, gebruiken programmeurs meestal alleen kleine letters in variabele namen.
- Als een variabele naam uit meerdere woorden bestaat, zetten programmeurs underscores tussen die woorden.
- Programmeurs kiezen nooit variabele namen die beginnen met een underscore. Het gebruik van dat soort namen is voorbehouden aan de auteurs van Python.

Je moet proberen je voor je eigen code ook aan deze conventies te houden. De conventie met betrekking tot het kiezen van betekenisvolle variabele namen is vooral belangrijk, omdat het code leesbaar en onderhoudbaar maakt. Kijk bijvoorbeeld eens naar de volgende code:

```
a = 3.14159265
b = 7.5
c = 8.25
d = a * b * b * c / 3
print( d )
```

Snap je wat deze code doet? Je ziet waarschijnlijk wel dat er een benadering van π in voorkomt, maar wat stelt `d` voor?

Deze code berekent het volume van een kegel. Dat zou je waarschijnlijk niet geraden hebben, maar dat is wel wat er gebeurt. Nu vraag ik je de code zo te wijzigen dat het volume van een kegel met een hoogte van 4 meter berekend wordt. Welke wijziging maak je in de code?A Als de hoogte in de formule staat, is het waarschijnlijk `b` of `c`. Maar welk van de twee? Als je een beetje wiskunde kent, en je bekijkt de berekening van `d`, dan zie je dat `b` gekwadrateerd wordt in de berekening. Dat lijkt dus de voet van de kegel te zijn, wat zou betekenen dat `c` de hoogte is. Maar dat kun je niet zeker weten.

Bekijk nu de volgende, equivalente code:

```
pi = 3.14159265
straal = 7.5
hoogte = 8.25
volume_van_kegel = pi * straal * straal * hoogte / 3
print( volume_van_kegel )
```

Dat is een stuk leesbaarder, nietwaar? Als ik je nu vraag de code te wijzigen, verwacht ik niet dat je ook maar een moment zult twijfelen.

Code met betekenisvolle namen wordt vaak "zelf-documenterend" genoemd; je hoeft er geen commentaarregels in op te nemen om de gebruiker te laten begrijpen wat de code doet en hoe het werkt. Dat neemt niet weg dat in bovenstaande code een commentaarregel als:
`# berekening van volume van kegel met straal 7.5 en hoogte 8.25`
niet zou misstaan.

4.2.2 Oefenen met variabele namen

Opgave In de code hieronder wordt de waarde 1 toegekend aan een aantal (potentiële variabele namen. Sommige hiervan zijn correct, andere niet. Identificeer de incorrecte namen en leg uit waarom ze incorrect zijn.

```
classificatie = 1   # 1
Classificatie = 1   # 2
cl@ssificatie = 1   # 3
class1f1cat1e = 1   # 4
1classificatie = 1  # 5
_classificatie = 1  # 6
class = 1           # 7
Class = 1           # 8
```

Antwoord The derde, vijfde, en zevende zijn incorrect. De derde omdat er een at-sign (@) in zit, de vijfde omdat hij begint met een cijfer, en de zevende omdat het een gereserveerd woord is (dat gelukkig opvalt vanwege de syntax highlighting). De andere zijn weliswaar correct, maar de zesde zou volgens de conventie vermeden moeten worden omdat het begint met een underscore, en de tweede en achtste ook, omdat die een hoofdletter bevatten. De achtste is het ergst, want die lijkt op een gereserveerd woord.

4.2.3 Constanten

Veel programmeertalen geven je de mogelijkheid om "constanten" te creëren, wat waardes zijn die aan een variabele zijn toegekend, die geen andere waarde meer kan krijgen. Het is conventie dat alle letters in dit soort variabele namen hoofdletters zijn. Constanten kunnen gebruikt worden om code leesbaarder en onderhoudbaarder te maken. Bijvoorbeeld, om voor een rekening van €24,95 exclusief BTW het eindbedrag te berekenen, kun je de volgende code gebruiken:

```
totaal = 24.95
eind_totaal = int( 100 * totaal * 1.21 ) / 100
print( eind_totaal )
```

Het is echter leesbaarder om te schrijven:

```
BTW_FACTOR = 1.21
CENTEN = 100

totaal = 24.95
eind_totaal = int( CENTEN * totaal * BTW_FACTOR ) / CENTEN
print( eind_totaal )
```

Niet alleen is dit leesbaarder, het maakt het ook gemakkelijk om de code aan te passen als de BTW tarieven wijzigen. Zeker als constanten meerdere malen terugkeren in code, is het beter om ze eenmalig een waarde te geven bovenin de code, waar ze gemakkelijk gevonden en gewijzigd kunnen worden. Als het numerieke waarden betreft, zoals de BTW factor, spreekt men vaak over "magische getallen": getallen waarvan de waarde voor de code een speciale betekenis heeft, die niet duidelijk is als je alleen het getal ziet. Je bent dus beter af als je een betekenisvolle naam ziet in plaats van een getal.

Hoewel constanten erg nuttig kunnen zijn, worden ze niet door Python ondersteund (wat erg jammer is). Dat wil zeggen dat in de code hierboven `BTW_FACTOR` een reguliere variabele is die overal in de code gewijzigd kan worden. Het is echter de gewoonte om dit soort variabelen die volledig uit hoofdletters bestaan te beschouwen als constanten die niet in de code gewijzigd mogen worden nadat ze hun initiële waarde hebben gekregen. Ik raad je aan dit soort semi-constanten te gebruiken als er magische getallen in je code voorkomen.

4.3 Debuggen met variabelen

Een veelvoorkomende oorzaak van functionele fouten in programma's is dat variabelen blijken niet de waardes te bevatten waarvan je dacht dat ze ze bevatten. Een goede manier om je code te "debuggen" (dat wil zeggen, uit te vinden waar in je code fouten staan en die te verbeteren) is het printen van variabele namen op geschikte plaatsen. Bijvoorbeeld, de volgende code geeft een foutmelding als je hem uitvoert.

listing0401.py

```
nr1 = 5
nr2 = 4
nr3 = 5
```

```
print( nr3 / (nr1 % nr2) )
nr1 = nr1 + 1
print( nr3 / (nr1 % nr2) )
nr1 = nr1 + 1
print( nr3 / (nr1 % nr2) )
nr1 = nr1 + 1
print( nr3 / (nr1 % nr2) )
```

Misschien zie je wat het probleem is, maar stel dat je het niet ziet, hoe vind je dan uit wat er mis is? Je ziet dat de fout ontdekt wordt op regel 10, wat wil zeggen dat alles nog steeds werkte op regel 9. Als je een extra regel code zet tussen regel 9 en 10, die de waarde afdrukt van `nr1`, `nr2`, `nr3` and misschien ook `nr1%nr2`, dan ontdek je waarschijnlijk snel wat er misloopt. **`print()`** statements veranderen niks aan de variabelen, dus je kunt ze veilig toevoegen. Een fatsoenlijke manier om het probleem in deze code op te lossen (dus een andere manier dan gewoon de laatste regel te verwijderen) zal ik in een later hoofdstuk introduceren.

Opgave Voeg de extra regel toe aan de foute code.

4.4 Soft typing

Alle variabelen hebben een data type. In veel programmeertalen wordt het data type van een variabele gespecificeerd op het moment dat de variabele gecreëerd wordt. Bijvoorbeeld, in C++ zet je het type van een variabele voor de variabele naam op het moment dat je de variabele definieert, als volgt:

```
int secs_per_week = 7 * 24 * 60 * 60;
```

Dit wordt "hard typing" genoemd.[4] Hard typing heeft als voordeel dat als je waarde in een variabele probeert te stoppen van het verkeerde type, het programma een foutmelding kan geven. Dit vermijdt een aantal vervelende problemen die kunnen optreden tijdens het schrijven of uitvoeren van een programma.

In Python geef je niet een vast type aan een variabele, maar de variabele heeft wel een type, namelijk het type van de waarde die erin is opgeslagen. Dit betekent dat als een variabele een nieuwe waarde krijgt, het type ook kan veranderen. Dit wordt "soft typing" genoemd. (Nota bene: Ik ben persoonlijk van mening dat Python nog geschikter zou zijn om beginnelingen programmeren te leren als het hard typing in plaats van soft typing zou hebben, maar Guido van Rossum, de ontwerper van Python, is het daar niet mee eens.)

De data types die je tot nu toe gezien hebt zijn integer, float, en string. Je kunt het data type van een waarde of variabele zien met behulp van de functie **`type()`**.

```
a = 3
print( type( a ) )
a = 3.0
print( type( a ) )
a = "3.0"
print( type( a ) )
```

[4] Er bestaat geen Nederlandse benaming voor dit fenomeen.

Omdat variabelen een type hebben, past het effect van operatoren die tussen variabelen staan zich aan aan de types van de variabelen. Bijvoorbeeld, in de code hieronder wordt de optelling (+) twee keer gebruikt, en het effect verandert naar gelang de variabele types.

```
a = 1
b = 4
c = "1"
d = "4"
print( a + b )
print( c + d )
```

Omdat `a` en `b` beide getallen zijn, is de + in `a + b` de numerieke optelling. Omdat `c` en `d` beide strings zijn, is de + in `c + d` de "concatenatie" ("vastplak") operator.

Opgave Wat gebeurt er in de code hierboven als je `a + c` probeert te printen? Als je het niet weet, probeer het dan.

Opgave Wat toont de code hieronder? Denk erover na, en voer dan de code uit. Zorg dat je snapt wat er gebeurt.

```
naam = "John Cleese"
print( "naam" )
```

Opgave Wjzig de code hierboven zodat de naam van een bekend lid van Monty Python wordt getoond.

4.5 Verkorte operatoren

Middels de operatoren die ik heb uitgelegd, kun je de inhoud van variabelen zo vaak wijzigen in je code als je nodig hebt. Je kunt nieuwe waardes in bestaande variabelen stoppen. Vaak wil je dat inderdaad doen. Bijvoorbeeld, het komt in code vaak voor dat er 1 moet worden opgeteld bij een numerieke variabele (waarom dat zo vaak voorkomt zul je zien in hoofdstuk 7). Omdat dit zo vaak gebeurt, bevat Python een aantal " verkorte notaties" om de inhoud van variabelen aan te passen.

De volgende code:

```
aantal_bananen = 100
aantal_bananen = aantal_bananen + 1
print( aantal_bananen )
```

is hetzelfde als:

```
aantal_bananen = 100
aantal_bananen += 1
print( aantal_bananen )
```

Het verschil is de tweede regel. Als je iets wilt optellen bij een variabele, kun je += gebruiken als assignment operator, met de variabele aan de linkerkant en wat je erbij op wilt tellen aan de rechterkant. Dit bespaart de moeite van het twee keer typen van de variabele naam, en maakt je code over het algemeen leesbaarder (omdat programmeurs ervan uitgaan dat je de += operator gebruikt als je ergens iets bij wilt optellen).

Op vergelijkbare manier kun je -= gebruiken om iets af te trekken van een variabele, *= voor vermenigvuldiging, /= voor deling, **= voor machtsverheffing, etcetera. De meeste verkorte versies worden weinig gebruikt, behalve +=, die juist heel veel gebruikt wordt, en -=, die zo nu en dan gebruikt wordt.

Opgave Wat toont de code hieronder? Controleer of je gelijk hebt.

listing0402.py

```
aantal_bananen = 100
aantal_bananen += 12
aantal_bananen -= 13
aantal_bananen *= 19
aantal_bananen /= aantal_bananen
print( aantal_bananen )
```

4.6 Commentaar

De code die je vanaf dit punt schrijft zal vaak meer dan vijf regels lang zal zijn. Dat is voldoende aanleiding om het gebruik van commentaarregels te bediscussiëren. Commentaarregels zijn teksten in code die Python negeert tijdens de uitvoering, maar die bedoeld zijn om specifieke delen van de code uit te leggen. Commentaar is niet alleen van nut voor andere mensen die je code moeten bestuderen en/of wijzigen, maar ook voor jezelf, aangezien je soms je eigen code moet wijzigen weken of maanden nadat je de code oorspronkelijk geschreven hebt, en je niet meer precies weet wat je gedaan hebt.

Er zijn twee manieren om commentaar op te nemen in Python code. De eerste manier is door gebruik te maken van de "hash mark" (#), die aangeeft dat alles dat op dezelfde regel code rechts van de markering staat, commentaar is (mits de hash mark niet in een string staat). De tweede manier is door een stuk commentaar tekst, dat meerdere regels mag beslaan, vooraf te gaan door drie aanhalingstekens (dubbel of enkel), en dezelfde markering aan het einde van de tekst te zetten. In dit geval moet je de drievoudige aanhalingstekens aan het begin ook aan het begin van een regel zetten, en je kunt deze manier van commentaar geven niet gebruiken in een blok code dat inspringt. De reden is dat je feitelijk een string die uit meerdere regels bestaat in je code plaatst (meer hierover in hoofdstuk 10).

De volgende code laat voorbeelden zien van beide manieren van becommentariëren:

listing0403.py

```
# Commentaar: schrijf je code hier...
# Valt het je op dat alles rechts van de # commentaar is?
print( "Iets..." ) # dat door Python genegeerd wordt?
print( "en iets anders.." ) # Geef zo commentaar op je code!
"""Een andere manier van commentaar geven is middels
```

```
drievoudige aanhalingstekens. Dit soort commentaar mag
meerdere regels""" # beslaan.
'''dit mag ook via enkele aanhalingstekens''' # maar pas op
# met het gebruik van aanhalingstekens IN je commentaar als
# je de meerdere-regels method gebruikt.
print( "Klaar." )
```

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Wat variabelen zijn
- Het toekennen van een waarde aan een variabele
- Correcte namen voor variabelen
- Conventies met betrekking tot variabele namen
- Soft typing
- Het debuggen van code waarin variabelen onverwachte waardes hebben
- Verkorte operatoren
- Commentaar

Opgaves

Opgave 4.1 Definieer drie variabelen `var1`, `var2` en `var3`. Bereken het gemiddelde en stop het in een variabele `gemiddelde`. Toon het gemiddelde. Voeg drie commentaren toe.

Opgave 4.2 Schrijf code die de oppervlakte van een cirkel berekent, gebruik makend van variabelen `straal` en `pi` = 3.14159. Voor het geval je het vergeten bent, de formule is straal keer straal keer pi. Toon de uitkomst als volgt: "De oppervlakte van een cirkel met straal ... is ..."

Opgave 4.3 Schrijf code die een hoeveelheid centen (opgeslagen in een variabele met de naam `bedrag`) classificeert als een combinatie van grotere geldstukken. Je code gebruikt dollars (100 ct), kwartjes (25 ct), dubbeltjes (10 ct), stuivers (5 ct), en centen (1 ct). Je programma begint met het bepalen hoeveel dollarstukken er in het bedrag passen, dan hoeveel kwartjes er in het restbedrag zitten nadat de dollars eruit genomen zijn, dan de hoeveelheid dubbeltjes, dan de stuivers, en tenslotte de centen. Het resultaat is dat je het bedrag uitdrukt in het minimale aantal muntjes dat nodig is.

Opgave 4.4 Kun je een manier bedenken om de inhoud van twee numerieke variabelen om te wisselen zonder daarbij gebruik te maken van een derde hulp-variabele? Basiscode hiervoor is gegeven in het code blok hieronder. Probeer de inhoud van `a` en `b` te verwisselen zonder een derde variabele erbij te halen. Om je te helpen, is de eerste stap al gegeven. Je hoeft slechts twee regels code toe te voegen.

exercise0404.py

```
a = 17
b = 23
print( "a =", a, "en b =", b )
a += b
# Voeg hier twee regels toe om a en b om te wisselen
print( "a =", a, "en b =", b )
```

Hoofdstuk 5

Eenvoudige Functies

Ik heb in de voorgaande hoofdstukken al gesproken over een aantal basis "functies," zoals `print()` en `int()`. In dit hoofdstuk worden deze functies in meer detail besproken, en zal ik een aantal nieuwe functies introduceren die nuttig gaan zijn in de volgende hoofdstukken. In hoofdstuk 8 ga ik bespreken hoe je je eigen functies kunt maken.

5.1 Elementen van een functie

A functie is een blok herbruikbare code dat een bepaalde actie uitvoert. Om een functie aan het werk te zetten, roep je hem aan (Engels: "call"), met de parameters als de functie nodig heeft. Je hoeft niet te weten hoe de functie precies werkt. Je hoeft slechts drie dingen te weten:

- De naam van de functie
- De parameters die de functie nodig heeft (als die er zijn)
- De waarde die de functie teruggeeft (als er zo'n waarde is)

Ik ga deze elementen één voor één bespreken.

5.1.1 Functie naam

Iedere functie heeft een naam. Net als variabele namen, mogen functie namen alleen bestaan uit letters, cijfers, en underscores, en mogen ze niet starten met een cijfer. Vrijwel alle standaard Python functies bestaan alleen uit kleine letters. Gewoonlijk is de naam van een functie een korte beschrijving van wat de functie doet.

Als je in een tekst refereert aan een functie, is het de gewoonte dat je achter de naam van de functie een openings- en sluithaakje zet, omdat functies in code altijd die haakjes hebben (zelfs al staat er niks tussen de haakjes).

5.1.2 Parameters

Sommige functies worden aangeroepen met parameters ("argumenten"), die meestal verplicht zijn. De parameters worden geplaatst tussen de haakjes die achter de functienaam staan. Als er meerdere parameters zijn, moet je er komma's tussen zetten.

De parameters zijn de waardes die de programmeur aan de functie geeft om te verwerken. Bijvoorbeeld, de functie **int**() wordt aangeroepen met één parameter, namelijk de waarde waarvan de functie probeert een integer representatie te maken. De **print**() functie mag worden aangeroepen met een willekeurig aantal parameters (zelfs nul), die de functie op het scherm zal tonen, waarna de functie naar een nieuwe regel op het scherm zal gaan.

Over het algemeen is het zo dat een functie de waardes van de parameters niet kan wijzigen. Bekijk bijvoorbeeld de volgende code:

```
x = 1.56
print( int( x ) )
print( x )
```

Als je deze code uitvoert, zie je dat **int**() niet de waarde van x heeft aangepast; de functie heeft alleen aan **print**() doorgegeven wat de integer representatie van de waarde van x is. De reden dat dit zo is, is dat over het algemeen alleen de waarde van parameters wordt doorgegeven (Engels: "passed by value"). Dit betekent dat de functie geen toegang heeft tot de variabelen die als parameters gebruikt worden, maar dat de functie kopieën krijgt van de waardes die in de parameters staan. Ik zeg "over het algemeen" omdat dit niet geldt voor alle data types, maar het geldt in ieder geval voor de data types die tot op dit moment bediscussieerd zijn. Pas in hoofdstuk 12 ga ik spreken over data types die wel door functies gewijzigd kunnen worden, en op dat moment zal ik dat heel duidelijk maken.

Als een functie meerdere parameters krijgt, maakt de volgorde uit. Bijvoorbeeld, de standaard functie **pow**() krijgt twee parameters, en rekent de waarde uit van de eerste die wordt verheven tot de macht weergegeven door de tweede.

```
basis = 2
exponent = 3
print( pow( basis, exponent ) )
```

De namen die aan de variabelen zijn gegeven doen niet ter zake, de eerste wordt verheven tot de macht die de tweede is. Dus de volgende code geeft een ander antwoord dan de vorige, omdat de variabelen in een andere (nogal verwarrende) volgorde aan de functie worden doorgegeven.

```
basis = 2
exponent = 3
print( pow( exponent, basis ) ) # verwarrende variabele namen
```

Wat gebeurt er als je een functie aanroept met parameter waardes waarmee de functie niet kan werken? Bijvoorbeeld, wat gebeurt er als ik **int**() aanroep met een string die geen integer bevat, of **pow**() met strings in plaats van getallen? Dat leidt meestal tot "runtime errors" (fouten tijdens de uitvoering van code). Bijvoorbeeld, beide regels in de volgende code leiden tot runtime errors.

```
x = pow( 3, "2" )
y = int( "twee-en-een-half" )
```

5.1.3 Retour waarde

Een functie heeft vaak een retour waarde. Als een functie een waarde retourneert, kun je die in je code gebruiken. Bijvoorbeeld, de **int()** functie retourneert een integer representatie van de parameter die is meegegeven. Je kunt deze retour waarde in een variabele stoppen middels een assignment, of je kunt de waarde op een andere manier gebruiken, bijvoorbeeld deze onmiddellijk printen. Je kunt er zelfs voor kiezen niks met de waarde te doen, maar in dat geval had het waarschijnlijk weinig zin om de functie aan te roepen.

```
x = 2.1
y = '3'
z = int( x )
print( z )
print( int( y ) )
```

Zoals je hierboven kunt zien, kun je zelfs functie aanroepen als parameter meegeven aan een functie, bijvoorbeeld, in de laatste regel van de code hierboven krijgt de **print()** functie als waarde een aanroep van de **int()** functie mee. De aanroep van **int()** vindt dan plaats voordat de **print()** wordt afgehandeld, dus de return waarde van **int()** is een parameter voor **print()**.

Niet alle functies retourneren een waarde. Bijvoorbeeld, **print()** geeft geen waarde terug. Als je niet uitkijkt, kan dit tot vreemd gedrag van je code leiden. Voer maar eens de volgende code uit:

```
print( print( "Hello, world!" ) )
```

Je ziet dat deze code twee regels print. De eerste bevat de tekst "Hello, world!" en de tweede het woord "None." Wat betekent dat woord "None"? Om dat te begrijpen, moet je uitpluizen hoe Python deze regel code verwerkt.

Wanneer Python deze regel code bekijkt, ziet het eerst **print(** <iets> **)**. Omdat <iets> een argument is, moet dat eerst geëvalueerd worden. <iets> is **print(** <nog_iets> **)**. Omdat <nog_iets> een argument is, moet Python dat eerst evalueren. <nog_iets> is de string "Hello, world!". Die hoeft niet verder geëvalueerd te worden, dus **print()** wordt uitgevoerd met als argument "Hello, world!", en Python "vangt" de retour waarde van deze uitvoering omdat die nodig is als <iets>.

En daar is het probleem: **print()** heeft geen retourwaarde, dus er is niks wat Python kan substitueren voor <iets>. Voor dit soort situaties heeft Python een speciale waarde die **None** heet. Dus het eerste **print()** commando krijgt als argument **None** mee, en dat leidt ertoe dat Python "None" op het scherm afdrukt.

None is een speciale waarde die aangeeft "geen waarde." Als je **None** probeert af te drukken, drukt Python het woord "None" af, maar dat is niet een string met de waarde "None". Het woord geeft slechts aan dat er niks af te drukken was. **None** is niet hetzelfde als een lege

string (""). Een lege string heeft nog steeds een waarde, namelijk een string met lengte nul. **`None`** is geen string, geen float, geen integer, niks. Dus wees voorzichting met het aanroepen van een functie als parameter; als de functie geen retour waarde heeft, kunnen er vreemde dingen gebeuren.

5.1.4 Een functie is een zwarte doos

In de wereld van programmeurs betekent een "zwarte doos" iets waar je wat in kunt stoppen en wat uit kunt halen, maar waarvan je niet kunt zien hoe het van binnen werkt. Je mag een functie beschouwen als een zwarte doos: het is niet van belang te weten hoe de functie werkt of hoe de code in de functie eruit ziet. Slechts de naam, de parameters, en de retour waarde moet je kennen om de functie te kunnen gebruiken. Het zou kunnen dat de functie intern variabelen aanmaakt en berekeningen doet, maar die hebben geen effect op de rest van de code.

...Tenminste, als de functie netjes geïmplementeerd is. Een functie die geen effect heeft op de rest van de code heet een "pure functie." Alle functies die ik hier bespreek zijn "pure functies." Maar het feit dat er een aparte naam is voor functies die de rest van de code niet aantasten, geeft al aan dat er ook functies bestaan die niet "puur" zijn. Dit is het duidelijkst bij functies waar de gebruiker parameters aan mee geeft, waarbij de inhoud van de variabelen die als parameter worden meegegeven gewijzigd wordt. Dat kan niet voor iedere soort variabele. En als het gebeurt kan dat best acceptabel zijn, als het de bedoeling is en goed gedocumenteerd is. Zulke functies heten "modifiers." Ik bespreek ze in een later hoofdstuk.

Vooralsnog mag je aannemen dat iedere functie die je gebruikt, geen effect heeft op de rest van de code. Het is veilig om functies aan te roepen.

5.2 Basis functies

Ik introduceer nu een aantal basis functies die je in je programma's kunt gebruiken.

5.2.1 Type casting

Ik heb al gesproken over type casting functies in hoofdstuk 4, maar nu ik meer details van functies heb gegeven, kan ik de beschrijving completeren.

- **`float`**`()` heeft één parameter en retourneert een floating-point representatie van de waarde van de parameter. Als de waarde een integer is, krijg je dezelfde waarde terug als float. Als de parameter een float is, krijg je dezelfde waarde terug. Als de parameter een string bevat die je als integer of float zou kunnen interpreteren, dan krijg je die float terug als waarde. Anders krijg je een runtime error.
- **`int`**`()` heeft één parameter en retourneert een integer representatie van de waarde van de parameter. Als de waarde een integer is, krijg je dezelfde waarde terug. Als de waarde een float is, krijg je een integer terug die de waarde van de float naar beneden heeft afgerond. Als de parameter een string bevat die je als integer zou kunnen interpreteren, dan krijg je die integer terug als waarde. Anders krijg je een runtime error.

- **str**() heeft één parameter en retourneert een string representatie van de waarde van de parameter.

Opgave Wat denk je dat de volgende code doet? Als je het niet weet, test dan de code.

```
print( 10 * int( "100,000,000" ) )
```

Opgave De code hierboven geeft een runtime error. Los het probleem op door precies twee tekens te verwijderen.

5.2.2 Berekeningen

Een paar basis Python functies helpen met berekeningen.

- **abs**() krijgt een numerieke parameter waarde. Als de waarde positief is, wordt hij weer geretourneerd. Als de waarde negatief is, wordt de waarde vermenigvuldigd met -1 geretourneerd.
- **max**() krijgt twee of meer numerieke parameters en retourneert de grootste.
- **min**() krijgt twee of meer numerieke parameters en retourneert de kleinste.
- **pow**() krijgt twee numerieke parameters en retourneert de eerste verheven tot de macht weergeven door de tweede. Optioneel mag je een derde parameter meegeven, die een integer moet zijn. Als je dat doet, krijg je de waarde modulo de derde parameter terug.
- **round**() krijgt een numerieke parameter die wiskundig wordt afgerond. Optioneel mag je als tweede parameter een integer meegeven die aangeeft hoeveel cijfers achter de komma behouden moeten worden. Als de tweede parameter niet wordt meegegeven, wordt afgerond op gehele getallen.

Opgave Bekijk de code hieronder en bedenk wat er op het scherm getoond wordt. Voer daarna de code uit en controleer of je gelijk hebt.

listing0501.py

```
x = -2
y = 3
z = 1.27

print( abs( x ) )
print( max( x, y, z ) )
print( min( x, y, z ) )
print( pow( x, y ) )
print( round( z, 1 ) )
```

5.2.3 len()

len() is a basis functie die één parameter krijgt, en die de lengte van die parameter teruggeeft. Op dit moment is het enig zinvolle data type dat je mee kunt geven aan **len**()

een string, waarvan je dan de lengte krijgt. In latere hoofdstukken volgen meer data types waarvan je de lengte kunt bepalen.

Opgave Wat print de code hieronder? Controleer of je vermoeden klopt.

```
print( len( 'man' ) )
print( len( 'mango' ) )
print( len( "" ) )  # "" is een lege string
```

Opgave En wat denk je van de code hieronder? Denk goed na voor je een antwoord geeft.

```
print( len( 'mango\'s' ) )
```

5.2.4 `input()`

In veel programma's wil je dat de gebruiker van het programma data verstrekt. Je kunt de gebruiker vragen een string in te typen met behulp van de functie **`input()`**. De functie krijgt één parameter mee, namelijk een string. Deze string is de zogeheten "prompt." Als **`input()`** wordt aangeroepen, wordt de prompt op het scherm gezet en mag de gebruiker een tekst ingeven. De gebruiker mag ingeven wat hij of zij wil, inclusief niks. De gebruiker sluit het ingeven af met een druk op de `Enter` toets. De retour waarde van de functie is hetgeen de gebruiker heeft ingegeven, exclusief de `Enter`.

Het hangt van de omgeving waar je je programma in draait hoe de gebruiker de ingave precies doet. Soms wordt er een rechthoek op het scherm getoond met de prompt ervoor waarin de gebruiker mag typen. Als je een Python programma draait op de command-line, moet de gebruiker ook de ingave op de command-line doen. Sommige editors tonen een popup-window waarin de gebruiker moet typen.

Hier is een voorbeeld:

```
tekst = input( "Geef een tekst in: " )
print( "Je hebt het volgende ingetypt:", tekst )
```

Realiseer je dat **`input()`** altijd een string teruggeeft. Bekijk de volgende code:

```
nummer = input( "Geef een getal: " )
print( "Je getal in het kwadraat is", nummer * nummer )
```

Het maakt niet uit wat je ingeeft, deze code geeft een runtime error. Omdat **`input()`** een string teruggeeft, wordt op de tweede regel een poging gedaan twee strings met elkaar te vermenigvuldigen, en dat kan niet. Het maakt niet uit of je string een getal bevat: een string is een string. Je kunt het probleem oplossen middels type casting, bijvoorbeeld:

```
nummer = input( "Geef een getal: " )
nummer = float( nummer )
print( "Je getal in het kwadraat is", nummer * nummer )
```

Voor deze code geldt dat als de gebruiker een getal ingeeft, de code doet wat hij moet doen. Maar als de gebruiker iets anders ingeeft, dat niet in een getal omgezet kan worden, krijg je toch weer een runtime error. Ook dat probleem kan opgelost worden, maar ik heb nog niet de zaken uitgelegd die je nodig hebt om dit probleem aan te pakken, en het zal nog tot hoofdstuk 17 duren voordat ik eraan toekom. Tot dat moment geef ik iets later in dit hoofdstuk een methode die je kunt gebruiken om je programma om een getal te laten vragen zonder dat het "crasht" als de gebruiker een wijsneus probeert te zijn en iets anders ingeeft.

Opgave Schrijf code die de gebruiker twee getallen laat ingeven, en die dan toont wat de uitkomst is als je ze optelt en als je ze vermenigvuldigt. Je code mag een runtime error geven als de gebruiker iets ingeeft wat geen getal is.

5.2.5 `print()`

De functie **`print`**`()` krijgt nul of meer parameters mee, toont ze op het scherm (als het er meerdere zijn, met spaties ertussen), en gaat daarna naar de volgende regel. Dus als je twee **`print`** statements gebruikt, komt de output van de tweede onder die van de eerste te staan.

Als **`print`**`()` wordt aangeroepen zonder parameters, gaat de functie alleen naar de volgende regel. Zo kun je lege regels op het scherm zetten.

Je mag als parameters meegeven wat je wilt, en **`print`**`()` zal zijn best doen het op het scherm weer te geven. Vooralsnog zul je echter alleen basis data types printen.

`print``()` kan twee speciale parameters meekrijgen, die `sep` en `end` heten.

`sep` geeft aan wat er getoond moet worden tussen iedere twee parameters, en is als default een spatie. Je kunt die spatie wijzigen in iets anders via `sep`, inclusief in een lege string.

`end` geeft aan wat **`print`**`()` moet tonen nadat alle parameters zijn getoond, en is als default een "nieuwe regel." Door `end` te wijzigen kun je ervoor zorgen dat **`print`**`()` iets anders doet dan naar een nieuwe regel gaan als hij klaar is met het tonen van parameters.

On `sep` en `end` te gebruiken, moet je speciale parameters opnemen, namelijk parameters `sep=<string>` en/of `end=<string>` (merk op: als in een beschrijving van code je iets ziet dat tussen < en > staat, betekent dat meestal dat je dat niet letterlijk moet typen, maar moet vervangen door van wat de term tussen de < en > aangeeft, dus `<string>` betekent dat je daar een string moet plaatsen). Bijvoorbeeld:

```
print( "X", "X", "X", sep="x" )
print( "X", end="" )
print( "Y", end="" )
print( "Z" )
```

Als je deze code uitvoert, zie je twee regels. De eerste bevat "XxXxX," aangezien er is aangegeven dat er drie keer een hoofdletter "X" afgedrukt moet worden met tussen iedere twee als separator een kleine letter "x." De tweede regel bevat "XYZ", omdat weliswaar dit drie verschillende aanroepen van **`print`**`()` betreft, maar na ieder van de eerste twee er niet naar een volgende regel wordt gegaan.

5.2.6 format()

format() is een nogal complexe functie die op een speciale manier gebruikt moet worden. De functie staat je toe een geformatteerde string te bouwen, dus een string waarin bepaalde waardes op een specifiek geformatteerde manier zijn opgenomen. Om een voorbeeld te geven, stel je voor dat ik de uitkomst van de berekening van een float wil tonen:

```
print( 7/11 )
```

Nu stel ik dat je de uitkomst moet tonen met 3 decimalen. Dat zou je kunnen doen met de **round()** functie, dus iets als:

```
print( round( 7/11, 3 ) )
```

Dit werkt, maar misschien heb ik extra eisen. Misschien stel ik dat je 10 posities ruimte moet reserveren voor deze uitkomst, en dat je binnen die 10 posities de uitkomst links moet aanlijnen. Dat lijkt lastig te realiseren, maar middels de **format()** functie is het vrij eenvoudig om waardes te formatteren op allerlei manieren. De volgende toepassing van **format()** realiseert het afronden op drie decimalen:

```
print( "{:.3f}".format( 7/11 ) )
```

format() "werkt" op een string. Tot op dit moment heb ik alleen functies gebruikt die aangestuurd worden via parameters. Echter, er zijn functies die alleen werken met een specifiek data type, en die op zo'n manier gedefinieerd zijn dat een variabele (of waarde) van dat data type vóór de functie naam moet staan, met een punt tussen de variabele (of waarde) en de naam van de functie. De reden hiervoor is een techniek die "object oriëntatie" heet, en die ik zal bediscussiëren in hoofdstukken 20 tot en met 23. Je hoeft nu alleen te weten dat zulke functie "methodes" genoemd worden, en dat je, om ze aan te roepen, een variabele (of waarde) van het juiste type voor de aanroep van de methode moet zetten, met een punt ertussen. Die variabele (of waarde) zelf is ook beschikbaar voor de methode, net als de parameters.

De **format()** methode (laten we de correcte benaming gebruiken, het is geen functie maar een methode) wordt als volgt aangeroepen: <string>**.format()**. Hij retourneert een nieuwe string, die een geformatteerde versie is van de string waarvoor de methode is aangeroepen. **format()** kan een willekeurig aantal parameters meekrijgen, die in de geformatteerde string ingebracht kunnen worden op specifieke plaatsen.

De plaatsen waar **format()** de parameter waardes in de string plaatst worden in de string aangegeven middels accolades ({ en }). Als je alleen {} gebruikt om de parameters aan te duiden, worden ze van links naar rechts afgehandeld. Bijvoorbeeld:

```
print( "De eerste drie getallen zijn {}, {} en {}.".format(
    "een", "twee", "drie" ) )
```

Als je ze in een andere volgorde wilt afhandelen, kun je de volgorde bepalen door een getal tussen de acolades te zetten. De eerste parameter is nummer 0, de tweede nummer 1, de derde nummer 2, etcetera (als je het vreemd vindt om nummering te beginnen bij nul, weet dan dat dat gebruikelijk is in programmeertalen, en dat je het nog vaker zult tegenkomen in dit boek). Bijvoorbeeld:

```
print( "Achterwaarts zijn ze {2}, {1} en {0}.".format(
    "een", "twee", "drie" ) )
```

format() kan variabelen van ieder type verwerken, zolang ze maar een fatsoenlijke string representatie hebben. Bijvoorbeeld, **format()** kan getallen verwerken, en zelfs verschillende soorten data types mixen:

```
print( "De eerste drie getallen zijn {}, {} en {}.".format(
    1, "twee", 3.0 ) )
```

Als je de parameters op een specifieke manier wil formatteren, zijn daar mogelijkheden voor, als je een dubbele-punt (:) tussen de accolades zet, na het volgorde-nummer als je dat gebruikt, met rechts van de dubbele-punt formatteringsinstructies. Ik zal een aantal formatteringsinstructies opnoemen.

Ik begin met instructies voor string parameters. Als je een bepaalde hoeveelheid tekens wilt reserveren voor een string, dan kun je dat aangeven met een integer rechts van de dubbele-punt. Dit wordt de "precisie" genoemd. De volgende code gebruikt een precisie van 7.

```
print( "De eerste drie getallen zijn {:7}, {:7} en {:7}.".format(
    "een", "twee", "drie" ) )
```

Als de precisie te kort is voor de lengte van de string, neemt **format()** gewoon meer ruimte voor de string. Je kunt de precisie dus niet gebruiken om een string voortijdig af te breken.

```
print( "De eerste drie getallen zijn {:3}, {:3} en {:3}.".format(
    "een", "twee", "drie" ) )
```

Als je precisie gebruikt, kun je de parameter links aanlijnen, centreren, of rechts aanlijnen in de ruimte die je hebt gereserveerd. Dat doe je door een "alignment" teken te plaatsen tussen de dubbele punt en de precisie. Deze "alignment" tekens zijn < voor links aanlijnen, ^ voor centreren, en > voor rechts aanlijnen.

```
print( "De eerste drie getallen zijn {:<7}, {:^7} en {:>7}.".
    format( "een", "twee", "drie" ) )
```

Ik ga nu over op formatteringsinstructies voor getallen. Als je een getal wilt laten interpreteren als een integer, moet je de kleine letter "d" plaatsen rechts van de dubbele punt ("d" staat hierbij voor "decimaal"). Wil je dat het getal wordt geïnterpreteerd als een float, dan moet je een "f" plaatsen rechts van de dubbele-punt. **format()** maakt de juiste conversie voor je als dat kan. Je kunt echter niet van een float een integer maken, want dat veroorzaakt een runtime error.

```
print( "{} gedeeld door {} is {}".format( 1, 2, 1/2 ) )
print( "{:d} gedeeld door {:d} is {:f}".format( 1, 2, 1/2 ) )
print( "{:f} gedeeld door {:f} is {:f}".format( 1, 2, 1/2 ) )
```

Net als bij strings, kun je voor getallen precisie en aanlijning gebruiken. Dit doe je op dezelfde manier. En net als bij strings geldt dat als de precisie niet voldoende groot is, de functie gewoon de ruimte neemt die nodig is. Let erop dat een eventueel min-teken en een eventuele punt in een float ook plaats nodig hebben.

```
print( "{:5d} gedeeld door {:5d} is {:5f}".format( 1, 2, 1/2 ) )
print( "{:<5f} gedeeld door {:^5f} is {:>5f}".format( 1,2,1/2 ))
```

Tenslotte, en misschien het meest nuttig, kun je aangeven met hoeveel decimalen een float getoond moet worden, door een punt en een getal te plaatsen links van de letter "f." De `format()` methode zal het getal afronden tot het juiste aantal decimalen. Merk op dat je ook mag aangeven dat je nul decimalen wilt zien door `.0` te gebruiken, wat ervoor zal zorgen dat een float wordt getoond als een integer.

```
print( "{:.2f} gedeeld door {:.2f} is {:.2f}".format( 1,2,1/2 ))
```

De combinatie van precisie, aanlijning, en decimalen staat je toe om redelijk uitziende tabellen te tonen.

listing0502.py

```
s = "{:>5d} keer {:>5.2f} is {:>5.2f}"
print( s.format( 1, 3.75, 1 * 3.75 ) )
print( s.format( 2, 3.75, 2 * 3.75 ) )
print( s.format( 3, 3.75, 3 * 3.75 ) )
print( s.format( 4, 3.75, 4 * 3.75 ) )
print( s.format( 5, 3.75, 5 * 3.75 ) )
```

5.3 Modules

Python biedt basis functies, waarvan ik er een aantal hierboven besproken heb. Naast die basis functies biedt Python ook een groot aantal zogeheten "modules," waarin zich vele nuttige functies bevinden. Om de functies van een module te gebruiken in een programma, moet je de juiste module importeren door boven in je programma **import** `<modulenaam>` op te nemen. Je kunt dan alle functies die in de betreffende module staan in je programma gebruiken, maar je moet de functie-aanroepen vooraf laten gaan door de naam van de module en een punt. Bijvoorbeeld, om de functie `sqrt()` uit de `math` module (die de wortel van een getal trekt) te gebruiken, roep je `math.sqrt()` aan nadat je `math` geïmporteerd hebt.

Als alternatief kun je ook specifieke functies vanuit een module importeren, via:
from `<module>` **import** `<functie1>, <functie2>, <functie3>, ...`
Het voordeel van een dergelijke manier van functies importeren is dat je in je code niet de naam van de module voor de functie-aanroep hoeft te zetten.

Bijvoorbeeld:

```
import math

print( math.sqrt( 4 ) )
```

is equivalent aan:

```
from math import sqrt

print( sqrt( 4 ) )
```

Als je een functie onder een andere naam in je programma wilt gebruiken, kun je dat doen middels het gereserveerde woord **as**. Dit kan zinvol zijn als je meerdere modules gebruikt waarin toevallig functies voorkomen die dezelfde naam hebben.

```
from math import sqrt as squareroot

print( squareroot( 4 ) )
```

Ik bespreek nu een aantal functies uit twee veelgebruikte standaard modules, en een aantal functies die in een module staan die ik voor dit boek gebouwd heb (in hoofdstuk 8 leg ik uit hoe je je eigen modules kunt maken). Er zijn veel meer standaard modules naast de modules die ik hieronder noem, en sommige ervan komen later nog aan de orde. Andere zul je zelf moeten opzoeken als je ze nodig hebt. Je mag er echter van uitgaan dat voor ieder min-of-meer-algemeen probleem dat je wilt oplossen, er iemand is geweest die er een module voor ontwikkeld heeft die het eenvoudig of zelfs triviaal maakt om het probleem op te lossen. Dus in de praktijk geldt: ga niet meteen zelf coderen, maar zoek eerst even uit of je niet gebruik kunt maken van de moeite die iemand anders gedaan heeft.

5.3.1 `math`

De `math` module bevat een aantal nuttige wiskundige functies. Deze functies zijn meestal zeer efficiënt, en retourneren meestal een float. Ik noem hier een klein aantal van de functies; als je er meer wilt kennen kun je ze opzoeken in de Python referentie):

- `exp()` krijgt één numerieke parameter en retourneert e tot de macht van die parameter. Als je niet weet wat e is: e is een speciaal getal met veel interessante eigenschappen, en wordt veel gebruikt in natuurkunde, wiskunde, en statistiek.
- `log()` krijgt één numerieke parameter en retourneert het natuurlijk logaritme van die parameter. Het natuurlijk logaritme is de waarde die de parameter als uitkomst heeft als je e verheft tot deze waarde. Net als e heeft het natuurlijk logaritme toepassingen in natuurkunde, wiskunde, en statistiek.
- `log10()` krijgt één numerieke parameter en retourneert het logaritme met 10 als basis van de parameter.
- `sqrt()` krijgt één numerieke parameter en retourneert de vierkantswortel van die parameter.

Bijvoorbeeld:

listing0503.py

```
from math import exp, log

print( "De waarde van e is bij benadering", exp( 1 ) )
```

```
e_sqr = exp( 2 )
print( "e kwadraat is", e_sqr, "wat betekent" )
print( "dat log(", e_sqr, ") gelijk is aan", log( e_sqr ) )
```

5.3.2 random

De random module bevat functies die pseudo-toevalsgetallen genereren. Ik zeg "pseudo-toevalsgetallen" en niet "toevalsgetallen," aangezien het onmogelijk is voor digitale computers om echt toevalsgetallen te genereren. Maar voor alle toepassingen mag je ervan uitgaan dat deze module toevalsgetallen genereert.

- random() krijgt geen parameters, en retourneert een toevalsgetal als een float binnen het bereik $[0, 1)$, dat wil zeggen een bereik tussen nul en 1, waarbij 0.0 wel meedoet maar 1.0 niet.
- randint() krijgt twee parameters, beide integers, waarbij de eerste kleiner dan of gelijk aan de tweede moet zijn. Het retourneert een toevalsgetal dat een integer is dat ligt binnen het bereik dat begrensd wordt door deze twee parameters, inclusief beide parameters. Bijvoorbeeld, randint(2,5) retourneert 2, 3, 4, of 5, elk met een gelijke kans.
- seed() initialiseert de toevalsgetal generator van Python. Als je een lijst van toevalsgetallen wilt hebben die iedere keer hetzelfde is voor je programma, kun je dat voor elkaar krijgen door aan het begin van je programma seed() aan te roepen met een vast getal, bijvoorbeeld 0. Dit kan nuttig zijn bij het testen van je programma. Als je de generator weer echt toevallige getallen wilt laten genereren op een later punt in je programma, kun je seed() nogmaals aanroepen zonder parameter.

For example:

listing0504.py

```
from random import random, randint, seed

seed()
print( "Een toevalsgetal tussen 1 en 10 is", randint( 1, 10 ) )
print( "Een ander is", randint( 1, 10 ) )
seed( 0 )
print( "3 toevalsgetallen zijn:", random(), random(), random() )
seed( 0 )
print( "Dezelfde 3 zijn:", random(), random(), random() )
```

5.3.3 pcinput

pcinput is een module die ik voor dit boek geschreven heb. Je vindt hem in appendix C, en je kunt hem gemakkelijk zelf maken (of eenvoudigweg downloaden via http://www.spronck.net/pythonbook). De module bevat vier handige functies, die de gebruiker op een veilige manier om specifieke input vragen. De functies zijn de volgende:

- `getInteger()` krijgt één string parameter, de prompt, en vraagt de gebruiker via die prompt om een integer in te geven. Als de gebruiker iets ingeeft wat geen integer is, wordt gevraagd de input opnieuw in te geven. De functie eindigt pas als de gebruiker een correcte integer heeft ingegeven, en de functie retourneert dan de ingegeven waarde als een integer.
- `getFloat()` krijgt één string parameter, de prompt, en vraagt de gebruiker via die prompt om een float in te geven. Als de gebruiker iets ingeeft wat geen float is, wordt gevraagd de input opnieuw in te geven. De functie eindigt pas als de gebruiker een correcte float heeft ingegeven, en de functie retourneert dan de ingegeven waarde als een float.
- `getString()` krijgt één string parameter, de prompt, en vraagt de gebruiker via die prompt om een string in te geven. Alles wat de gebruiker ingeeft wordt als correct beschouwd. De functie retourneert de ingegeven waarde, waarbij spaties voor en na de ingegeven tekst verwijderd zijn.
- `getLetter()` krijgt één string parameter, de prompt, en vraagt de gebruiker via die prompt om één letter in te geven. Alleen letters van het alfabet zijn acceptabel. Pas als de gebruiker precies één letter heeft ingegeven eindigt de functie, en de letter wordt dan als een hoofdletter geretourneerd.

Deze functies helpen je dus om code te schrijven die de gebruiker vraagt om input met een specifiek data type te verstrekken, omdat ze garanderen dat het programma inderdaad iets binnenkrijgt dat van het gevraagde data type is. De code geeft geen runtime error als de gebruiker iets anders ingeeft. De functies zijn niet erg netjes, omdat ze foutmeldingen geven in het Nederlands als iets fouts ingegeven wordt. Dat betekent dat je deze functies niet moet gebruiken als je een Engelstalig programma schrijft (daar heb ik een andere versie van de module voor), maar om Python te leren zijn deze functies afdoende.

Opgave Creëer of download de `pcinput` module, zorg dat hij staat in de folder waar je je programma's schrijft, en maak dan een Python programma met onderstaande code. Voer het programma uit en test het door iets in te geven wat geen integer is.

```
from pcinput import getInteger

num1 = getInteger( "Geef een geheel getal: " )
num2 = getInteger( "Geef een ander geheel getal: " )

print( num1, "+", num2, "=", num1 + num2 )
```

Opgave Vraag de gebruiker om een string in te geven. Gebruik dan die string als prompt om de gebruiker te vragen een float in te geven.

Merk op: Ik leg niet uit hoe `pcinput` werkt, omdat ik er concepten voor gebruik die pas in hoofdstuk 17 aan bod komen. Je zult later leren hoe je zelf dit soort functies kunt maken. Je hoeft je vooralsnog niet druk te maken over hoe ze werken, je hoeft ze alleen maar te gebruiken. Dat is de houding die je tegenover de meeste standaardfuncties moet hebben: zolang je maar weet wat ze doen, welke parameters ze nodig hebben, en wat ze retourneren, heeft het geen zin na te denken over hoe ze werken.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Wat functies zijn
- Functie namen
- Functie parameters
- Functie retourwaardes
- Details van type casting functies **`float`**`()`, **`int`**`()`, en **`str`**`()`
- Basis berekeningen met **`abs`**`()`, **`max`**`()`, **`min`**`()`, **`pow`**`()`, en **`round`**`()`
- **`len`**`()`
- **`input`**`()`
- Details van the **`print`**`()` functie
- String formattering met **`format`**`()`
- Wat modules zijn
- De `math` functies `exp()`, `log()`, `log10()`, en `sqrt()`
- De `random` functies `random()`, `randint()`, en `seed()`
- De `pcinput` functies `getInteger()`, `getFloat()`, `getString()`, en `getLetter()`

Opgaves

Opgave 5.1 Vraag de gebruiker om een string, en druk de lengte van de string af. Gebruik de **`input`**`()` functie en niet de `getString()` functie, aangezien de `getString()` functie spaties verwijdert die voor en na de ingegeven tekst staan.

Opgave 5.2 De stelling van Pythagoras zegt dat bij een rechthoekige driehoek het kwadraat van de schuine zijde gelijk is aan de som van de kwadraten van de twee andere zijden (ofwel $a^2 + b^2 = c^2$). Schrijf een programma dat de gebruiker om de lengte van de twee rechte zijden vraagt, en bereken dan de lengte van de schuine zijde. Toon hem op een netjes geformatteerde manier. Je hoeft geen rekening te houden met het feit dat de gebruiker ook negatieve waardes of nul zou kunnen ingeven.

Opgave 5.3 Vraag de gebruiker om drie getallen, en toon dan de grootste, de kleinste, en hun gemiddelde afgerond op twee decimalen.

Opgave 5.4 Bereken de waarde van e tot de machten -1, 0, 1, 2, en 3, en toon de resultaten met vijf decimalen op een netjes geformatteerde manier.

Opgave 5.5 Stel dat je een geheel toevalgetal tussen 1 en 10 wilt hebben (inclusief 1 en 10), maar je hebt alleen de `random()` functie beschikbaar. Hoe doe je dat?

Hoofdstuk 6

Condities

In een programma zijn er vaak regels code die je alleen wilt uitvoeren onder bepaalde omstandigheden. Om dat te regelen, bieden alle programmeertalen zogeheten "conditionele statements" of "condities." In dit hoofdstuk leg ik uit hoe condities werken in Python

6.1 Boolean expressies

Een conditioneel statement, vaak een "if"-statement genoemd, bestaat uit een test en één of meerdere acties. De test is een zogeheten "boolean expressie." De acties worden alleen uitgevoerd als de test evalueert als zijnde "waar." Bijvoorbeeld, een app op een smartphone kan een waarschuwing geven als de batterij minder dan 5% vol is. Dat betekent dat de app test of een zekere variabele `batterij_energie` kleiner is dan 5, dus of de vergelijking `batterij_energie < 5` als zijnde "waar" geëvalueerd wordt. Als de variabele momenteel de waarde 17 bevat, evalueert de test `batterij_energie < 5` als zijnde "onwaar."

6.1.1 Booleans

In Python wordt "waar" weergegeven door de waarde **`True`**, en "onwaar" door de waarde **`False`**.

`True` en **`False`** zijn zogeheten "boolean waardes," die door Python zijn gedefinieerd. **`True`** en **`False`** zijn zelfs de *enige* booleans, en alles wat niet **`False`**, is automatisch **`True`**.

Als je je afvraagt welke data type **`True`** en **`False`** hebben: ze zijn van het type **`bool`**. In Python kan echter elke waarde worden geïnterpreteerd als boolean, ongeacht het data type. Dus als je een test doet of iets **`True`** of **`False`** is, en je test dat van een waarde die niet van het data type **`bool`** is, dan wordt hetgeen je test toch als ofwel **`True`** ofwel **`False`** beschouwd.

The volgende waardes worden beschouwd als zijnde **`False`**:

- De speciale waarde **`False`**
- De speciale waarde **`None`** (die ik heb besproken in hoofdstuk 5)
- Iedere numerieke waarde die nul is, bijvoorbeeld 0 en 0.0

- Iedere lege serie, bijvoorbeeld een lege string ("")
- Iedere lege "afbeelding," bijvoorbeeld een lege "dictionary" (dictionaries zijn het onderwerp van hoofdstuk 13)
- Iedere functie of methode die één van de bovenstaande waardes retourneert (inclusief functies die niks retourneren)

Iedere andere waarde wordt beschouwd als zijnde **`True`**.

Een expressie die evalueert als **`True`** of **`False`** heet een "boolean expressie."

6.1.2 Vergelijkingen

De meestgebruikte boolean expressies zijn vergelijkingen. Een vergelijking bestaat uit twee waardes met een vergelijkingsoperator ertussen. Vergelijkingsoperatoren zijn:

```
<    kleiner dan
<=   kleiner dan of gelijk aan
==   gelijk aan
>=   groter dan of gelijk aan
>    groter dan
!=   niet gelijk aan
```

Een veelgemaakte fout is om twee waardes te vergelijken met een enkele =. De enkele = is de assignment operator. Meestal (maar niet altijd) produceert Python een syntax of runtime error als je de = probeert te gebruiken om twee waardes te vergelijken.

Je kunt vergelijksoperatoren gebruiken zowel tussen getallen als tussen strings. Vergelijkingen tussen strings zijn alfabetische vergelijkingen, waarbij je wel moet bedenken dat hoofdletters altijd beschouwd worden als kleiner dan kleine letters (en cijfers kleiner dan alle letters). Ik ga daar dieper op in in hoofdstuk 10.

Hier volgen een paar voorbeelden van vergelijkingen:

listing0601.py

```
print( "1.", 2 < 5 )
print( "2.", 2 <= 5 )
print( "3.", 3 > 3 )
print( "4.", 3 >= 3 )
print( "5.", 3 == 3.0 )
print( "6.", 3 == "3" )
print( "7.", "syntax" == "syntax" )
print( "8.", "syntax" == "semantiek" )
print( "9.", "syntax" == " syntax" )
print( "10.", "Python" != "rotzooi" )
print( "11.", "Python" > "Perl" )
print( "12.", "banaan" < "mango" )
print( "13.", "banaan" < "Mango" )
```

Zorg dat je deze vergelijkingen uitvoert, en dat je snapt waarom ze de uitkomst geven die ze geven!

Opgave Begrijp je waarom `3 < 13` **`True`** oplevert, maar `"3" < "13"` **`False`** oplevert? Indien niet, denk er dan goed over na!

Je kunt de uitkomst van een boolean expressie aan een variabele toekennen als je wilt:

```
groter = 5 > 2
print( groter )
groter = 5 < 2
print( groter )
print( type( groter ) )
```

Opgave Schrijf code die test of $1/2$ groter dan, gelijk aan, of kleiner is dan 0.5. Doe dat ook voor $1/3$ en 0.33. Doe het dan ook voor $(1/3) * 3$ en 1.

Vergelijkingen tussen data types die niet vergeleken kunnen worden, leiden meestal tot runtime errors.

```
# Deze code geeft een runtime error.
print( 3 < "3" )
```

6.1.3 `in` operator

Python heeft een speciale operator die de "lidmaatschap test operator" heet, en die vanwege die onverkwikkelijke mondvol meestal de "in operator" wordt genoemd aangezien hij gecodeerd wordt als **`in`**. De **`in`** operator test of een waarde voorkomt in een collectie, als de waarde links van de **`in`** staat, en de collectie rechts van de **`in`**.

Er zijn verschillende soorten collecties in Python, maar de enige die ik tot op dit moment bediscussiëerd heb is de string. Een string is een collectie van tekens. Je kunt testen of een specifiek teken, of een groepje tekens, onderdeel is van een string middels de **`in`** operator. De tegenhanger van de **`in`** operator is de **`not in`** operator, die **`True`** oplevert als **`in`** **`False`** oplevert, en vice versa. Bijvoorbeeld:

```
print( "y" in "Python" )
print( "x" in "Python" )
print( "p" in "Python" )
print( "th" in "Python" )
print( "to" in "Python" )
print( "y" not in "Python" )
```

Zorg er weer voor dat je deze evaluaties begrijpt!

Opgave Schrijf code die test van ieder van de klinkers (`"a"`, `"e"`, `"i"`, `"o"`, `"u"`) of ze voorkomen in je naam. Je mag hoofdletters negeren.

6.1.4 Logische operatoren

Boolean expressies kunnen gecombineerd worden middels logische operatoren. Er zijn drie logische operatoren: **`and`**, **`or`**, en **`not`**.

and en **or** plaats je tussen twee boolean expressies.

Als **and** tussen twee boolean expressies staat, is het resultaat **True** als beide expressies **True** zijn; anders is het resultaat **False**.

Als **or** tussen twee boolean expressies staat, is het resultaat **True** als één of beide expressies **True** zijn; het resultaat is alleen **False** als beide **False** zijn.

not kun je voor een boolean expressie plaatsen om hem om te keren van **True** naar **False** en vice versa.

Bijvoorbeeld:

```
t = True
f = False
print( t and t )
print( t and f )
print( f and t )
print( f and f )
print( t or t )
print( t or f )
print( f or t )
print( f or f )
print( not t )
print( not f )
```

Kijk uit met het gebruik van logische operatoren, want een combinatie van **and**s en **or**s kan leiden tot onverwachte resultaten. Gebruik haakjes om te zorgen dat ze in de gewenste volgorde geëvalueerd worden. Bijvoorbeeld, in plaats van `a and b or c` te schrijven, moet je `(a and b) or c` of `a and (b or c)` schrijven (afhankelijk van de gewenste volgorde), zodat het duidelijk is welke evaluatie je wilt uitvoeren. Zelfs als je weet in welke volgorde Python de evaluatie doet zonder haakjes, hoeft dat niet te gelden voor iemand anders die je code leest.

Opgave Geef voor de code hieronder waardes voor `a`, `b`, and `c`, die ertoe leiden dat de twee expressies verschillende uitkomsten hebben.

listing0602.py

```
a = # True of False?
b = # True of False?
c = # True of False?

print( (a and b) or c )
print( a and (b or c) )
```

Als je logische expressie maakt met alleen **and**s, of alleen **or**s, hoef je geen haakjes te gebruiken, want dan is er slechts één mogelijke evaluatie van de expressie.

Boolean expressies worden van links naar rechts geëvalueerd, en Python stopt de evaluatie op het moment dat de uitkomst van de evaluatie bekend is. Neem bijvoorbeeld de volgende code:

```
x = 1
y = 0
print( (x == 0) or (y == 0) or (x / y == 1) )
```

Als je deelt door nul, geeft Python een runtime error, dus de evaluatie van `x / y == 1` geeft een error als `y` nul is. En als je de code bestudeert, zie je dat `y` inderdaad nul is. Maar de code geeft geen foutmelding. Python evalueert de boolean expressie van links naar rechts, en ziet op een gegeven moment `... or (y == 0) or ...` `y == 0` evalueert als **`True`**. Omdat een expressie die via **`or`**s gecombineerd is **`True`** is als één van de componenten **`True`** is, kan Python na evaluatie van `(y == 0)` concluderen dat deze hele expressie **`True`** is. Het is dus niet nodig dat Python `x / y == 1` evalueert, en Python doet dat dan ook niet. Het is wel van belang dat `y == 0` links van `x / y == 1` staat, zodat Python `y == 0` eerst test.

Merk op dat hoewel je heel ingewikkelde boolean expressies kunt bouwen via logische operatoren, ik je aanraad dat je je expressies zo eenvoudig mogelijk houdt. Eenvoudige expressies houden code leesbaar.

6.2 Conditionele statements

Zoals ik aan het begin van dit hoofdstuk aangaf, bestaan conditionele statements, die ook wel "condities" of "**`if`** statements" worden genoemd (omdat ze gedefinieerd worden met behulp van het gereserveerde woord **`if`**), uit een test en één of meer acties, waarbij de acties alleen worden uitgevoerd als de test **`True`** oplevert.

Hier is een voorbeeld:

```
x = 5
if x == 5:
    print( "x is 5" )
```

De syntax van een **`if`** statement is als volgt:

```
if <boolean expressie>:
    <acties>
```

Let op de dubbele punt (:) die achter de boolean expressie staat, en het feit dat `<acties>` inspringt.

6.2.1 Blokken code

In de syntactische beschrijving van de **`if`** statement, zie je dat `<acties>` "inspringt," dus één tabulatie naar rechts is geplaatst (in het Engels heet dit "indent"). Dit is opzettelijk zo gedaan en noodzakelijk. Python beschouwt statements die elkaar opvolgen en die hetzelfde niveau van inspringing hebben als één blok code. Het blok code dat onder het **`if`** statement staat is de lijst van acties die worden uitgevoerd als de boolean expressie **`True`** is. Bijvoorbeeld:

listing0603.py

```
x = 7
if x < 10:
    print( "Deze regel wordt alleen uitgevoerd als x < 10." )
    print( "En dat geldt ook voor deze regel." )
print( "Deze regel wordt echter altijd uitgevoerd." )
```

Opgave Wijzig de waarde van `x` en test hoe dat de resultaten beïnvloedt.

Dus alle regels code die onder de **`if`** staan en inspringen, horen tot het blok code dat wordt uitgevoerd als de boolean expressie behorende bij de **`if`** evalueert als **`True`**. Het blok code wordt daarentegen overgeslagen als de boolean expressie evalueert als **`False`**. Statements die volgen na de **`if`** en die niet inspringen (althans, niet zo diep als het blok code onder de **`if`**) worden uitgevoerd ongeacht het resultaat van de evaluatie van de boolean expressie.

Je hoeft je niet te beperken tot slechts één **`if`** statement. Je kunt er zoveel hebben als je wilt.

listing0604.py

```
x = 5
if x == 5:
    print( "x is 5" )
if x > 4:
    print( "x is groter dan 4" )
if  x >= 5:
    print( "x is groter dan of gelijk aan 5" )
if x < 6:
    print( "x is kleiner dan 6" )
if x <= 5:
    print( "x is kleiner dan of gelijk aan 5" )
if x != 6 :
    print( "x is niet 6" )
```

Opgave Wijzig weer de waarde van `x` en test hoe dat de resultaten beïnvloedt.

6.2.2 Inspringen

In Python is correct inspringen van het grootste belang! Zonder correcte inspringing kan Python niet zien welke regels code een blok vormen, en kan daarom niet je code uitvoeren zoals je bedoelt.[5]

[5] In veel programmeertalen (of eigenlijk in vrijwel alle programmeertalen) worden blokken code door de compiler/interpreter herkend doordat ze beginnen en eindigen met een speciaal symbool of gereserveerd woord. Bijvoorbeeld, in talen als Java en C++ worden blokken code omsloten door accolades, terwijl in talen als Pascal en Modula ze beginnen met het woord `begin` en eindigen met het woord `end`. Dat betekent dat in vrijwel alle talen correcte inspringing niet nodig is. Je ziet toch dat goede programmeurs correct inspringen, ongeacht de taal die ze gebruiken. Dat maakt het namelijk gemakkelijk te zien welke delen van de code bij elkaar horen, bijvoorbeeld, wat er hoort bij een `if` statement. Bij Python is het inspringen een verplichting. Voor ervaren programmeurs die voor het eerst Python leren komt dat wat vreemd over, maar ze beseffen al snel dat het ze niks uitmaakt – ze lieten hun code toch al netjes inspringen. Python maakt het echter ook voor beginnende programmeurs een noodzakelijkheid om netjes in te springen, wat betekent dat ze gedwongen zijn om nette code te schrijven. En dat is alleen maar nuttig voor iedereen.

Je kunt inspringen door middel van de `Tab` toets, of je kunt het doen middels spaties. De meeste editors doen aan "auto-indenting," dat wil zeggen, ze springen automatisch in als de code daartoe aanleiding geeft. Bijvoorbeeld, als je een editor hebt die Python ondersteund, en je schrijft een **`if`** statement, dan zal de editor de daarop volgende regel meteen laten inspringen (als dat niet gebeurt, heb je waarschijnlijk een syntax fout gemaakt, bijvoorbeeld de dubbele punt vergeten). Ook wordt het niveau van inspringing aangehouden voor iedere volgende regel, tenzij je middels de `Backspace` toets inspringing verwijdert.

In Python programma's is inspringing normaal vier spaties, dus een druk op de `Tab` toets moet vier posities inspringen. Zolang je in een specifieke editor werkt, kun je ofwel de `Tab` toets gebruiken, ofwel zelf de spatiebalk vier keer indrukken, om één niveau van inspringing op te schuiven. Je kunt echter in de problemen komen als je tussentijds wisselt van editor, die wellicht andere instellingen voor tabulaties gebruikt. Zelfs als je code in die andere editor er goed uitziet, kan Python toch tijdens uitvoering melden dat er "indentation conflicts" zijn (dat wil zeggen, dat tabulaties en spaties door elkaar gehutseld zijn). Dat kan gezien worden als een syntax fout. De meeste editors bieden de mogelijkheid via een optie om tabulaties automatisch te vervangen door spaties, wat vermijdt dat zulke problemen optreden. Dus als je een tekst editor gebruikt om Python code te schrijven, controleer dan of die optie bestaat, en laat hem dan automatisch tabulaties vervangen door vier spaties.

Opgave De volgende code bevat inspring-fouten. Verbeter de code.

listing0605.py

```
# Deze code bevat tabulatie-fouten!
x = 3
y = 4
if x == 3 and y == 4:
    print( "x is 3" )
   print( "y is 4" )
if x > 2 and y < 5:
print( "x > 2" )
print( "y < 5" )
if x < 4 and y > 3:
    print( "x < 4" )
        print( "y > 3" )
```

6.2.3 Twee-weg beslissingen

Het komt regelmatig voor dat een beslissing twee kanten uit kan gaan, dat wil zeggen, onder bepaalde omstandigheden wil je een bepaald iets doen, en als die omstandigheden niet optreden wil je iets anders doen. Python staat dit toe door aan een **`if`** statement een **`else`** tak toe te voegen.

```
x = 4
if x > 2:
    print( "x is groter dan 2" )
else:
    print( "x is kleiner dan of gelijk aan 2" )
```

De syntax is:

```
if <boolean expressie>:
    <acties>
else:
    <acties>
```

Merk op dat er een dubbele punt achter de **else** staat, net zoals achter de `<boolean expressie>` bij de **if**.

Het is van belang dat het woord **else** uitgelijnd is met het woord **if** waar het bij hoort. Als je ze niet correct uitlijnt, krijg je ofwel een tabulatie fout, ofwel je code doet niet wat je zou willen dat hij doet.

Een consequentie van het toevoegen van een **else** tak aan een **if** statement is dat altijd precies één van de twee blokken `<acties>` wordt uitgevoerd. Als de boolean expressie **True** is, wordt het blok code onder de **if** uitgevoerd, en wordt het blok code onder de **else** overgeslagen. Als de boolean expressie **False** is, wordt het blok code onder de **if** overgeslagen, maar wordt het blok code onder de **else** uitgevoerd.

Opgave Je kunt testen of een integer even of oneven is met de modulo operator. Namelijk als `x%2` nul is, dan is `x` even, en anders is `x` oneven. Schrijf code die vraagt om een integer en dan rapporteert of de integer even of oneven is (je kunt de `getInteger()` functie van `pcinput` gebruiken om om een integer te vragen).

Opmerking met betrekking tot inspringing: het is niet absoluut noodzakelijk om het blok code onder de **else** aan te lijnen met het blok code onder de **if**, zolang de inspringing maar consistent is binnen het blok code. Ervaren programmeurs gebruiken echter consistente inspringing door de hele code, wat het gemakkelijk maakt om te zien hoe een **if**-**else** statement werkt. Bijvoorbeeld, in de code hieronder is de inspringing voor de **else** tak kleiner dan voor de **if** tak. Syntactisch is dat correct, maar het maakt de code slechter leesbaar.

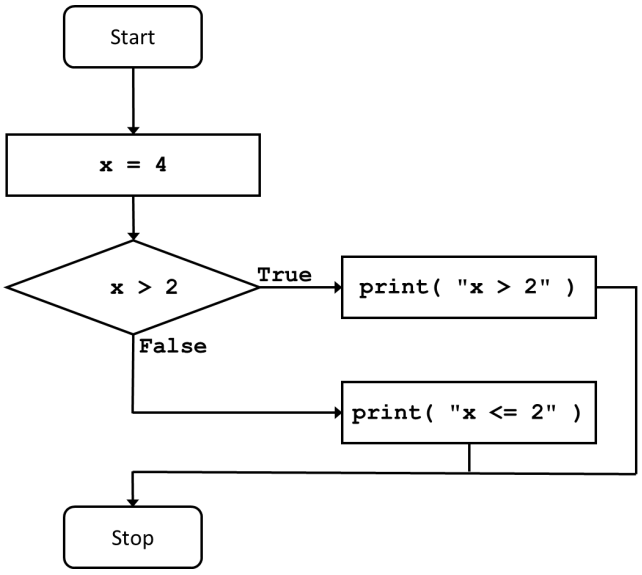
```
# Syntactisch correct maar lelijke inspringing.
x = 1
if x > 2:
    print( "x is groter dan 2" )
else:
  print( "x is kleiner dan of gelijk aan 2" )
```

6.2.4 Stroomdiagrammen

In de tijd dat het beroep "programmeur" nog vrij nieuw was, gebruikten programmeurs vaak een techniek die "stroomdiagram" genoemd wordt om algoritmes te beschrijven. Vandaag de dag worden stroomdiagrammen nog slechts zelden gebruikt. Studenten hebben mij echter aangegeven dat stroomdiagrammen helpen om begrip te krijgen van de exacte werking van conditionele expressies (en van iteraties, die in het volgende hoofdstuk besproken worden).

Een stroomdiagram is een schematische weergave van een algoritme middels blokken met pijlen ertussen. Er zijn drie soorten blokken (althans, ik heb voldoende aan drie soorten blokken voor dit boek). Rechthoekige blokken bevatten statements die uitgevoerd worden. Ruitvormige blokken bevatten een conditie die geëvalueerd wordt als **`True`** of **`False`**. Rechthoekige blokken met ronde hoeken geven ofwel de start (met de tekst "Start") ofwel het einde (met de tekst "Stop") van het algoritme aan.

Om een stroomdiagram te interpreteren, begin je bij het "Start" blok, en volgt de pijlen, waarbij je ieder statement dat je tegenkomt uitvoert. Als je een ruitvormig blok tegenkomt, evalueer je de conditie die erin staat, en dan volg je ofwel de pijl die met **`True`** gemarkeerd is als de conditie **`True`** is, ofwel de pijl die met **`False`** gemarkeerd is als de conditie **`False`** is. Wanneer je het "Stop" blok tegenkomt, ben je klaar.



Afb. 6.1: Stroomdiagram dat een twee-weg beslissing weergeeft.

Bijvoorbeeld, de code die hierboven (in 6.2.3) staat, waarbij een getal vergeleken wordt met 2 en waarbij wordt afgedrukt of het getal groter is dan 2, of kleiner dan of gelijk aan 2 is, is equivalent met het stroomdiagram dat getoond wordt in afbeelding 6.1.

6.2.5 Meer-weg beslissingen

Het komt voor dat je een situatie hebt waarbij je één van meerdere blokken code wilt uitvoeren, maar niet meer dan één. Dit soort meer-weg beslissingen kun je implementeren met een extra toevoeging aan een **if** statement, namelijk in de vorm van één of meer **elif** takken (**elif** staat voor "else if").

listing0606.py

```
leeftijd = 21
if leeftijd < 12:
    print( "Je bent een kind!" )
elif leeftijd < 18:
    print( "Je bent een teenager!" )
elif leeftijd < 30:
    print( "Je bent nog jong!" )
elif leeftijd < 50:
    print( "Beginnen grijze haren te komen?" )
else:
    print( "Wegen de jaren zwaar?" )
```

Deze code is equivalent aan het algoritme dat is weergeven in afbeelding 6.2.

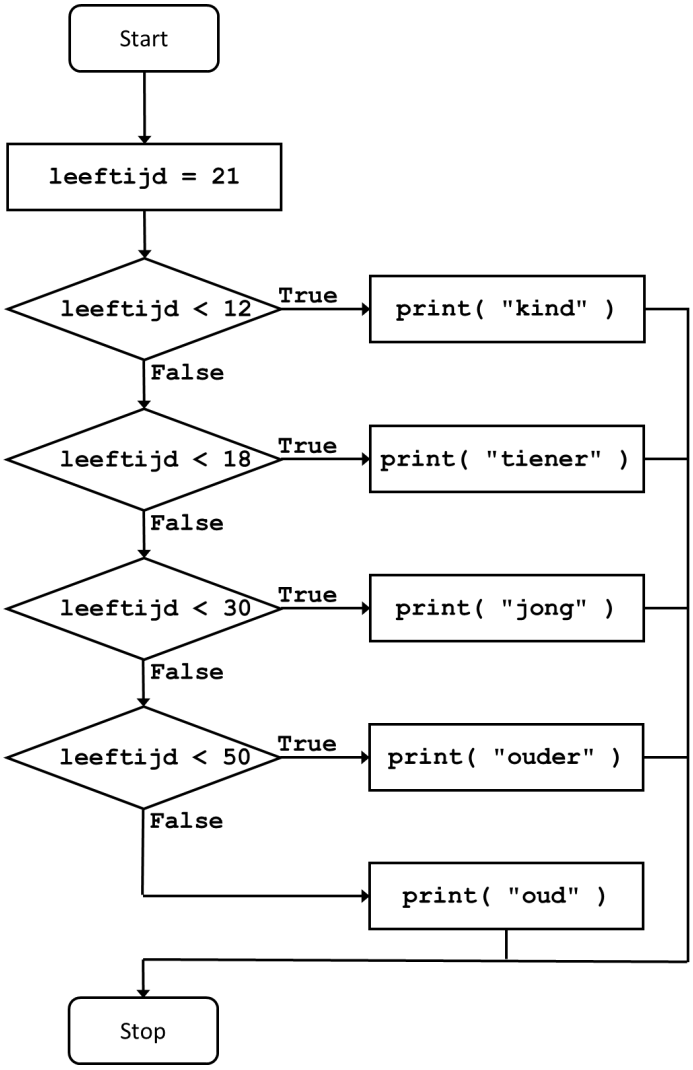
Opgave Verander in de code hierboven de waarde van de variabele `leeftijd` en bestudeer de resultaten.

De syntax is:

```
if <boolean expressie>:
    <acties>
elif <boolean expressie>:
    <acties>
else:
    <acties>
```

De syntax hierboven toont slechts één **elif**, maar je mag er meerdere hebben. De verschillende tests in een **if**-**elif**-**else** constructie worden in volgorde uitgevoerd. De eerste boolean expressie die geëvalueerd wordt als **True** laat het bijbehorende blok code uitvoeren. Geen andere blokken code in de hele constructie worden daarna nog uitgevoerd, en er worden daarna ook geen boolean expressies in de constructie meer getest.

Kortom, eerst wordt de boolean expressie bij de **if** geëvalueerd. Als die **True** is, wordt het blok code onder de **if** uitgevoerd. Anders wordt de boolean expressie bij de eerste **elif** geëvalueerd. Als die **True** blijkt te zijn, wordt de code behorende bij deze **elif** uitgevoerd. Zo niet, dan wordt de boolean expressie bij de tweede **elif** geëvalueerd. Etcetera. Slechts als alle boolean expressies in de constructie **False** blijken te zijn, wordt het blok code bij de **else** uitgevoerd.



Afb. 6.2: Stroomdiagram dat een meer-weg beslissing weergeeft.

Voor het voorbeeld hierboven betekent dit dat bij de eerste **`elif`** de test `leeftijd < 18` volstaat, en niet hoeft te worden aangevuld met een test **`and`** `leeftijd >= 12`, want als `leeftijd` kleiner geweest zou zijn dan 12, dan zou de boolean expressie voor de **`if`** al **`True`** zijn geweest, en zou de boolean expressie bij de eerste **`elif`** niet eens door Python zijn gezien.

Het toevoegen van **`else`** is altijd optioneel. In de meeste gevallen waarin ik zelf meerdere **`elif`**s gebruik, zet ik wel een **`else`**, al was het maar voor het afvangen van fouten.

Opgave Schrijf een programma dat een variabele `gewicht` heeft. Als `gewicht` groter is dan 20 (kilo), print je: "Er moet een toeslag van $25 betaald worden voor baggage die te zwaar is." Als `gewicht` kleiner is dan 20, print je: "Goede reis!" Als `gewicht` precies 20 is, print je: "Poeh! Dat gewicht is precies goed!" Wijzig de waarde van `gewicht` een paar keer om de code te testen.

6.2.6 Geneste condities

Gegeven de syntactische regels voor **if-elif-else** constructies en inspringing, is het mogelijk om **if** statements op te nemen in de code blokken behorende bij andere **if** statements. Zo'n geneste **if** wordt alleen uitgevoerd als de boolean expressie behorende bij het code blok waarin de geneste **if** staat **True** is.

listing0607.py

```
x = 41
if x%7 == 0:
    # --- Hier begint een genest blok code  ---
    if x%11 == 0:
        print( x, "is deelbaar door 7 en 11." )
    else:
        print( x, "is deelbaar door 7, maar niet door 11." )
    # --- Hier eindigt een genest blok code ---
elif x%11 == 0:
    print( x, "is deelbaar door 11, maar niet door 7." )
else:
    print( x, "is niet deelbaar door 7 of 11." )
```

Deze code is equivalent aan het algoritme dat wordt weergegeven in afbeelding 6.3.

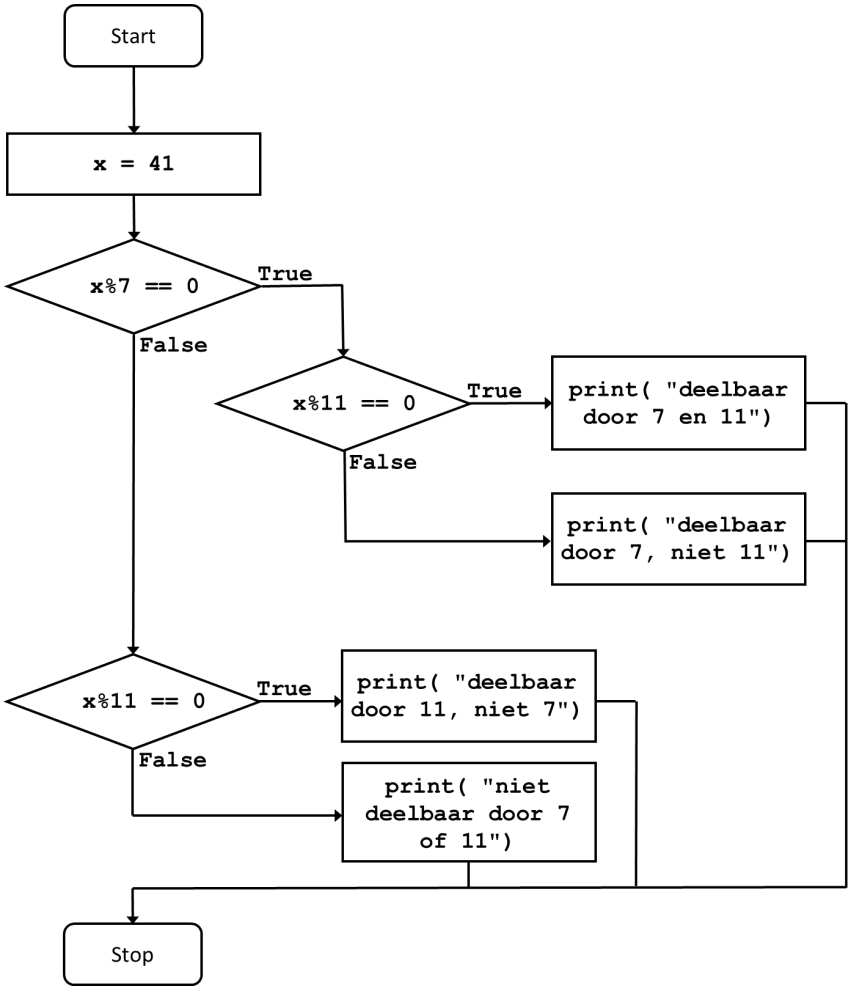
Opgave Wijzig de waarde van x en controleer de resultaten.

In het voorbeeld hierboven wil ik alleen maar het nesten van **if** statements demonstreren. Er zijn leesbaardere manieren om te doen wat deze code doet. Het nesten van **if** statements kan vaak vermeden worden door **elif**s te gebruiken. Bijvoorbeeld, de code hieronder doet hetzelfde als de "leeftijd" code hierboven, maar nu met geneste **if**s in plaats van **elif**s.

listing0608.py

```
leeftijd = 21
if leeftijd < 12:
    print( "Je bent een kind!" )
else:
    if leeftijd < 18:
        print( "Je bent een teenager!" )
    else:
        if leeftijd < 30:
            print( "Je bent nog jong!" )
        else:
            if leeftijd < 50:
                print( "Beginnen grijze haren te komen?" )
            else:
                print( "Wegen de jaren zwaar?" )
```

Ik neem aan dat je het ermee eens bent dat de versie met **elif**s prettiger leest.



Afb. 6.3: Stroomdiagram dat een geneste conditie weergeeft.

6.3 Vroegtijdig afbreken

Soms wil je een programma vroegtijdig beëindigen onder bepaalde condities. Bijvoorbeeld, je programma vraagt de gebruiker om een waarde, en voert met die waarde een aantal berekeningen uit. Als de gebruiker een waarde invoert die niet in de berekeningen gebruikt kan worden, wil je het programma meteen beëindigen. Dat kun je als volgt coderen:

```
from pcinput import getInteger

num = getInteger( "Geef een positief geheel getal: " )
if num < 0:
    print( "Je had een positief geheel getal moeten geven!" )
else:
    print( "Ik handel je getal", num, "af" )
    print( "Nog meer code" )
    print( "Honderden regels code" )
```

Het is irritant dat een groot deel van het programma al één inspringing diep staat, terwijl je er de voorkeur aan zou hebben gegeven als het programma gestopt was na de foutmelding, en de rest van het programma zonder inspringing geschreven zou kunnen worden. Je kunt dat regelen met behulp van de functie `exit()` die in de module `sys` staat. De code is dan:

listing0609.py

```
from pcinput import getInteger
from sys import exit

num = getInteger( "Geef een positief geheel getal: " )
if num < 0:
    print( "Je had een positief geheel getal moeten geven!" )
    exit()

print( "Ik handel je getal", num, "af" )
print( "Nog meer code" )
print( "Honderden regels code" )
```

Als je deze code uitvoert en een negatief getal ingeeft, kan het gebeuren dat je ziet dat Python een `SystemExit` melding genereert, die eruit ziet als een grote, lelijke fout. Dit is afhankelijk van de editor die je gebruikt (IDLE geeft deze melding niet). Het is geen fout, ook al ziet het er zo uit. Deze melding zegt alleen dat je het programma geforceerd beëindigd hebt, maar dat is precies wat je wilde doen. Je mag dit beschouwen als een nette manier van afbreken.

In principe moet je meldingen van Python over je programma niet negeren, maar deze is een uitzondering. Je mag je programma op deze manier afbreken. In hoofdstuk 8 zal ik een andere manier van programma afbreken bespreken, die ervoor zorgt dat je deze melding niet krijgt. Dat kun je tegen die tijd gebruiken (als de melding je echt stoort), maar vooralsnog moet je hem maar accepteren.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Wat boolean expressies zijn
- Boolean waardes **`True`** en **`False`**
- Vergelijkingen met <, <=, ==, >, >=, en !=
- De **`in`** operator
- Logische operatoren **`and`**, **`or`**, en **`not`**
- Conditionele statements met **`if`**, **`elif`**, en **`else`**
- Blokken code
- Inspringing
- Geneste condities
- `exit()`

Opgaves

Opgave 6.1 Cijfers voor proefwerken en tentamens vallen tussen nul en 10 (inclusief nul en 10), en worden altijd afgerond op halve punten. De Amerikaanse stijl van beoordelen gebruikt letters. Ter vergelijking, de cijfers 8.5 tot en met 10 zijn in Amerika “A,” 7.5 en 8 zijn “B," 6.5 en 7 zijn “C,” 5.5 en 6 zijn “D,” en 5 en lager is “F.” Schrijf code die deze vertaling van cijfers naar letters maakt, waarbij de gebruiker gevraagd wordt om het cijfer in te geven. Als de gebruiker een cijfer buiten het gegeven bereik ingeeft, moet je een foutmelding geven. Je hoeft niet te eisen dat de gebruiker alleen getallen ingeeft die geheel zijn of eindigen in .5, maar je mag dat wel doen, aangezien je op zich voldoende kennis hebt om dit op te lossen. In dat geval, geef een zinvolle foutmelding als de gebruiker iets incorrects ingeeft.

Opgave 6.2 Snap je welke redeneerfout gemaakt is in de volgende code?

exercise0602.py

```
score = 98.0
if score >= 60.0:
    oordeel = 'D'
elif score >= 70.0:
    oordeel = 'C'
elif score >= 80.0:
    oordeel = 'B'
elif score >= 90.0:
    oordeel = 'A'
else:
    oordeel = 'F'
print( oordeel )
```

Opgave 6.3 Vraag de gebruiker om een string. Druk af hoeveel *verschillende* klinkers er in de string zitten. De hoofdletter versie van een klinker wordt beschouwd als gelijk aan de kleine-letter versie. Probeer de uitvoer een beetje netjes te maken (bijvoorbeeld, het is lelijk om te zeggen: “Er zitten 1 verschillende klinkers in de string”). Voorbeeld: als de gebruiker als string “De Heilige Handgranaat van Antioch” ingeeft, zegt het programma dat er 4 verschillende klinkers in de string zitten.

Opgave 6.4 Je kunt kwadratische vergelijkingen oplossen met de wortelformule. Kwadratische vergelijkingen hebben de vorm $Ax^2 + Bx + C = 0$. Dit soort vergelijkingen heeft nul, één of twee oplossingen. De eerste oplossing is $(-B + sqrt(B^2 - 4AC))/(2A)$. De tweede oplossing is $(-B - sqrt(B^2 - 4AC))/(2A)$. Er zijn geen oplossingen als de waarde onder de wortel negatief is. Er is één oplossing als de waarde onder de wortel nul is. Schrijf een programma dat de gebruiker vraagt om waardes voor A, B, en C, en dan rapporteert hoeveel oplossingen er zijn, en welke oplossingen dat zijn. Let erop dat je ook afhandelt wat er gebeurt als A nul is, of als A en B allebei nul zijn.

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

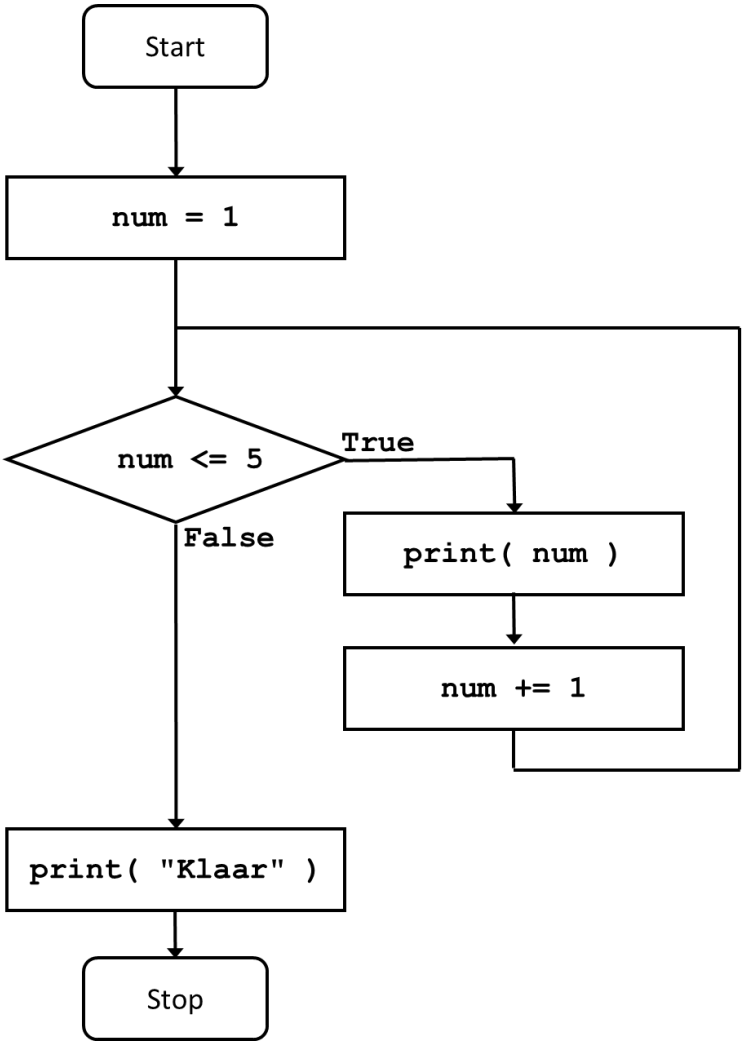
Ik neem een eenvoudig voorbeeld: het printen van de nummers 1 tot en met 5. Met een while loop kun je dat als volgt doen:

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

Pythin 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.

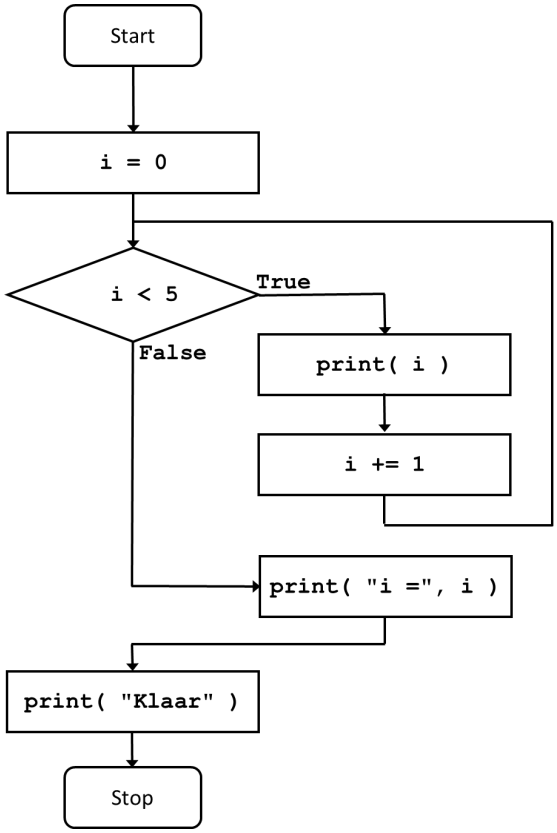
7.3.1 `else`

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.

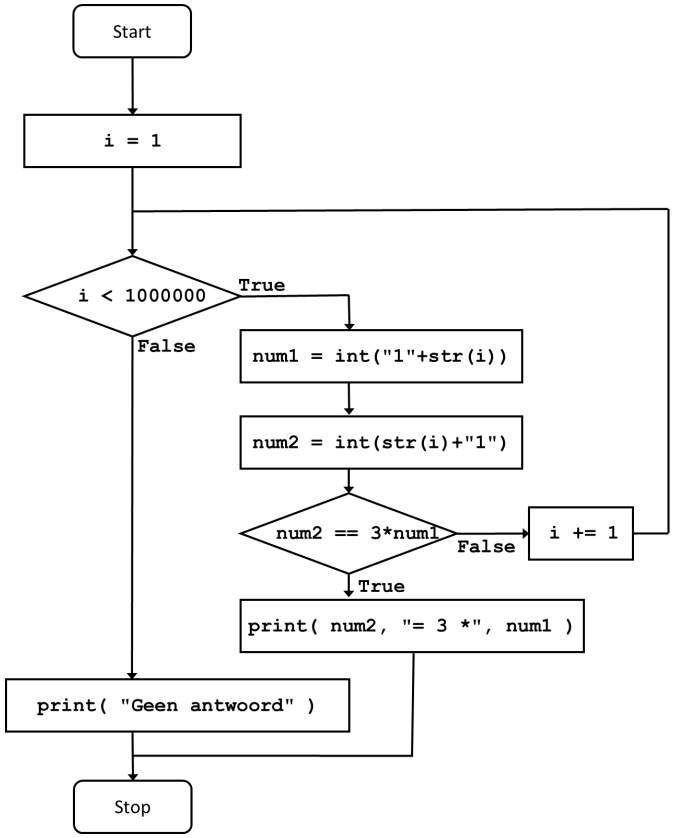
In dit voorbeeld zie je een goed gebruik van **break**. Omdat ik geen idee heb van hoe groot het nummer is dat ik zoek, controleer ik er gewoon een heleboel. Ik laat `i` oplopen tot 1000000. Ik weet natuurlijk niet of ik het antwoord vind voordat `i` 1000000 heeft bereikt, maar ik moet ergens een grens 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.

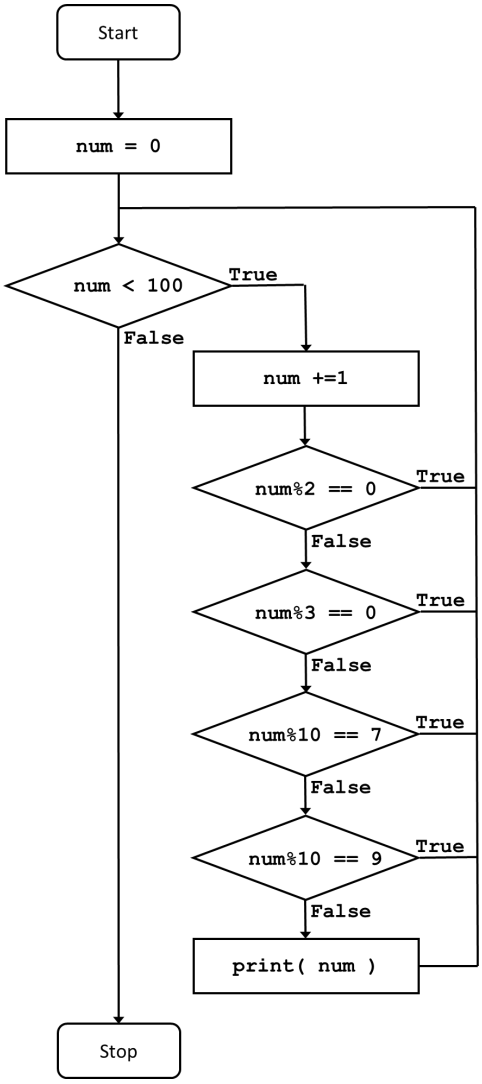
Net als **break** statements, kun je **continue** statements alleen gebruiken in loops.

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 `continues`.

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 gebruik 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.

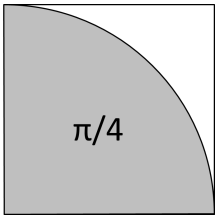
Opgave 7.5 De Fibonacci reeks is een serie getallen die start met 1, gevolgd door nogmaals 1. Ieder volgende getal is de som van de twee voorgaande getallen. De reeks start dus met 1, 1, 2, 3, 5, 8, 13, 21,... Schrijf een programma dat de Fibonacci reeks afdrukt totdat de getallen groter dan 1000 zijn.

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.

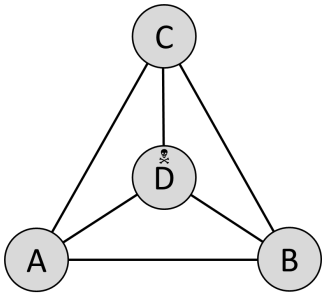
De waarschijnlijkheid dat een Driehoekkruiper op de eerste dag onmiddellijk naar D gaat is 1/3, en dan leeft hij dus maar één dag. In principe kan een Driehoekkruiper iedere willekeurige leeftijd bereiken, hoe hoog ook, door in cirkels te bewegen van A naar B naar C en terug naar A (of tegen de klok 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.



Hoofdstuk 8

Functies

In hoofdstuk 5 beschreef ik hoe je eenvoudige functies kunt gebruiken, en hoe je functies kunt importeren uit modules. Dit hoofdstuk beschrijft hoe je je eigen functies en modules kunt creëren. Als je niet meer weet wat hoofdstuk 5 over functies zei, doe je er goed aan om dat hoofdstuk nogmaals door te nemen.

8.1 Het nut van functies

Waarom zou je functies willen creëren? Er zijn een hoop goede redenen:

- Je hebt een bepaalde functionaliteit voor je code nodig die je onafhankelijk van de rest van je programma wilt ontwikkelen. Als je een dergelijke functionaliteit in een functie stopt, betekent dat dat je nadat de functie gebouwd en getest is, je de functionaliteit kunt gebruiken zonder er verder over na te denken.
- Er is een bepaalde functionaliteit die je op meerdere plekken in je programma nodig hebt, en je kunt beter die functionaliteit in een functie stoppen die op meerdere plekken aangeroepen wordt, dan dat je de code naar al die plekken kopieert.
- Je hebt een functionaliteit in je code nodig die je kunt aansturen middels parameters. Als je dergelijke functionaliteit in een functie stopt, worden de parameters duidelijker en wordt de code gemakkelijker te lezen en te onderhouden.
- Je programma is te lang om de inhoud goed te blijven overzien. Door de code op te splitsen in functies blijf je veel langer grip houden op de inhoud en werking.
- Je hebt een probleem dat je te lastig vindt om in één keer op te lossen. Je besluit het probleem op te splitsen in meerdere sub-problemen die je wel aankunt. Functies zijn een natuurlijke manier om een dergelijke aanpak vorm te geven.
- Een programma dat diep geneste condities of loops bevat, wordt veel leesbaarder als dieper geneste gedeeltes in een functie geplaatst worden.
- Het hergebruik van code wordt goed gefaciliteerd door delen van code in functies te plaatsen.
- Code die beschikbaar gesteld moet worden aan andere programmeurs, kan verspreid worden door gedocumenteerde functies in modules te plaatsen.

Over het algemeen gesproken, kunnen de voordelen van functies worden samengevat als een middel om de volgende zaken te bewerkstelligen:

- *Encapsulatie*: Het "inpakken" van een nuttig stuk code op zo'n manier dat het gebruikt kan worden zonder kennis van de specifieke werking van de code
- *Generalisatie*: Het geschikt maken van een stuk code voor diverse situaties door gebruik te maken van parameters
- *Beheersbaarheid*: Het verdelen van een complex programma in gemakkelijk-te-bevatten delen
- *Onderhoudbaarheid*: Het gebruik maken van betekenisvolle functienamen en logische opdelingen om een programma beter leesbaar en begrijpbaar te maken
- *Herbruikbaarheid*: Het faciliteren van de overdraagbaarheid van code tussen programma's
- *Recursie*: Het beschikbaar maken van een techniek die "recursie" heet, wat het onderwerp is van hoofdstuk 9.

8.2 Het creëren van functies

Hoofdstuk 5 beschrijft hoe iedere functie een naam heeft, nul of meer parameters, en mogelijk een retourwaarde. Als je je eigen functies maakt, moet je al deze elementen definiëren. Je gebruikt de volgende syntax:

```
def <functie_naam>( <parameter_lijst> ):
    <acties>
```

Functienamen moeten voldoen aan dezelfde eisen als variabele namen, dat wil zeggen, alleen letters, cijfers, en underscores, en ze mogen niet beginnen met een cijfer. De parameter lijst bestaat uit nul of meer variabele namen, met komma's ertussen. Het blok code onder de functie definitie moet inspringen.

Let erop dat Python je functie definitie moet hebben "gezien" voordat je de functie kunt aanroepen. Het is daarom de gewoonte dat functie definities bovenin een programma staan, meteen onder de **`import`** statements.

8.2.1 Hoe Python omgaat met functies

Om functies te kunnen creëren, moet je begrijpen hoe Python met functies omgaat. Bekijk het kleine Python programma hieronder. Dit programma definieert één functie, die `totZiens()` heet. Deze functie heeft geen parameters. Het blok code behorende bij de functie heeft alleen een regel die de tekst "Tot ziens!" print.

De rest van het programma is geen deel van de functie. Het deel van de code van een programma dat niet bij een functie hoort, noemt men meestal het "hoofdprogramma" (Engels: "main program"). Het hoofdprogramma print de regel "Hallo!," en roept daarna de functie `totZiens()` aan.

```
def totZiens():
    print( "Tot ziens!" )

print( "Hallo!" )
totZiens()
```

Als je dit programma uitvoert, zie je dat het eerst de regel "Hallo!" afdrukt, en daarna "Tot ziens!" Dit gebeurt zo ondanks het feit dat Python de code van boven naar beneden uitvoert; Python komt `print( "Tot ziens!" )` tegen voordat `print( "Hallo!" )` wordt aangetroffen. De reden dat Python toch eerst de tekst "Hallo!" afdrukt is dat Python de code van een functie alleen uitvoert als de functie wordt aangeroepen. Een functie die Python tegenkomt voordat de functie wordt aangeroepen wordt alleen geregistreerd – Python onthoudt dat die functie bestaat, zodat hij kan worden uitgevoerd als het hoofdprogramma (of een andere functie) de functie aanroept.

8.2.2 Parameters en argumenten

Bekijk de code hieronder. De code definieert een functie `hallo()` met één parameter, `naam` geheten. De functie gebruikt `naam` in het blok code. Er is geen expliciete assignment die `naam` een waarde geeft. `naam` bestaat als variabele naam omdat het een parameter is van de functie.

Als een functie wordt aangeroepen, moet je een waarde geven aan iedere (verplichte) parameter die voor de functie gedefinieerd is. Zo'n waarde wordt een "argument" genoemd. Dus, om de functie `hallo()` aan te roepen, moet een argument waarde voor de parameter `naam` gespecificeerd worden. Je plaatst een argument tussen de haakjes van de functie aanroep. Merk op dat het niet nodig is dat je in het hoofdprogramma weet dat de parameter `naam` wordt genoemd in de functie. Hoe hij heet is niet belangrijk. Het enige wat je moet weten is dat er een parameter is die een waarde nodig heeft, en liefst ook de beperkingen die de functie oplegt aan de waarde (bijvoorbeeld het type parameter dat de schrijver van de functie wilt dat je meegeeft).

```
def hallo( naam ):
    print( "Hallo, {}!".format( naam ) )

hallo( "Groucho" )
hallo( "Chico" )
hallo( "Harpo" )
hallo( "Zeppo" )
```

Parameters van een functie zijn niet meer en niet minder dan variabelen die je kunt gebruiken in de functie, en die hun (initiële) waarde krijgen van buiten de functie (namelijk via de functie aanroep). De parameters zijn "lokaal" voor de functie, dat wil zeggen, ze kunnen niet benaderd worden door code die niet in het blok code van de functie staat, noch kunnen ze waardes van variabelen buiten de functie beïnvloeden. Meer hierover volgt later in dit hoofdstuk.

Functies kunnen meerdere parameters hebben. Bijvoorbeeld, de volgende functie krijgt twee parameters en vermenigvuldigt hun waardes, waarna het resultaat wordt afgedrukt:

listing0801.py

```
def vermenigvuldig( x, y ):
    resultaat = x * y
    print( resultaat )

vermenigvuldig( 2020, 5278238 )
vermenigvuldig( 2, 3 )
```

8.2.3 Parameter types

In veel programmeertalen moet je bij de definitie van een functie aangeven wat de types van de parameters zijn. Dit zorgt ervoor dat de compiler/interpreter kan controleren of de functie met de juiste soort argumenten wordt aangeroepen. Bij Python geef je geen data types aan. Dit betekent dat, bijvoorbeeld, de `vermenigvuldig()` functie hierboven kan worden aangeroepen met string argumenten. Als je dat doet, veroorzaakt dat een runtime error (want je kunt geen strings met elkaar vermenigvuldigen).

Als je een "veilige" functie wilt schrijven, kun je het type argumenten dat de functie meekrijgt testen met de **`isinstance()`** functie. **`isinstance()`** krijgt een waarde of variabele mee als eerste argument, en een data type als tweede argument. De functie geeft **`True`** als de waarde of variabele van het genoemde type is, en anders **`False`**. Bijvoorbeeld:

listing0802.py

```
a = "Hallo"
if isinstance( a, int ):
    print( "integer" )
elif isinstance( a, float ):
    print( "float" )
elif isinstance( a, str ):
    print( "string" )
else:
    print( "anders" )
```

Als je de parameters zo test, moet je natuurlijk wel besluiten wat je doet als de functie met verkeerde argumenten is aangeroepen. De standaard manier om dit af te handelen is door een "exception" te genereren. Dat bespreek ik in hoofdstuk 17. Vooralsnog mag je aannemen dat functies die je zelf schrijft worden aangeroepen met argumenten van het correcte type. Zolang je alleen zelf de functies gebruikt, kun je dat altijd garanderen.

8.2.4 Default parameter waardes

Je kunt default waardes geven aan sommige parameters. Je doet dit door bij het specificeren van de parameter een assignment operator (=) en de gewenste waarde op te nemen, alsof het een reguliere assignment is. Als je de functie aanroept, mag je een waarde voor alle parameters opgeven, of slechts voor een aantal. De waardes worden aan de parameters gegeven van links naar rechts. Als je minder waardes opgeeft dan er parameters zijn, maar er default waardes gegeven zijn aan de parameters waarvoor je niks opgeeft, krijg

je geen runtime fouten. Als je een functie definieert met default waardes voor een aantal parameters, maar niet voor alle parameters, dan is het de gewoonte om de parameters waarvoor je een default waarde opgeeft rechts te plaatsen van de parameters waarvoor je geen default waarde hebt.

Als je de default waarde van een specifieke parameter wilt vervangen, en je kent de naam van de parameter maar je weet niet waar hij staat in de patameter-orde, of je wilt simpelweg de default waardes van de overige parameters intact laten, dan kun je in de aanroep van de functie deze *specifieke* parameter overschrijven door een assignment aan de parameter op te nemen tussen de haakjes, dus `<functie>( <parameternaam>=<waarde> )`.

De volgende code geeft voorbeelden van deze mogelijkheden:

listing0802a.py

```
def vermenigvuldig_xyz( x, y=1, z=7 ):
    print( x * y * z )

vermenigvuldig_xyz( 2, 2, 2 ) # x=2, y=2, z=2
vermenigvuldig_xyz( 2, 5 )    # x=2, y=5, z=7
vermenigvuldig_xyz( 2, z=5 )  # x=2, y=1, z=5
```

8.2.5 `return`

Parameters worden gebruikt om informatie van buiten de functie naar de functie toe te communiceren. Vaak wil je ook informatie vanuit de functie naar het programma buiten de functie toe communiceren. Daartoe dient het commando **`return`**.

Als Python **`return`** tegenkomt in een functie, beëindigt dat de functie. Python gaat dan verder met de code vlak na de plek waar de functie werd aangeroepen. Je mag achter het woord **`return`** nul, één, of meerdere waardes of variabelen opnemen. Deze waardes worden dan gecommuniceerd naar het programma buiten de functie. Als je de waardes wilt gebruiken, moet je zorgen dat ze in een variabele terecht komen. Dat krijg je voor elkaar door de functie-aanroep te doen als een assignment naar een variable.

Bijvoorbeeld, stel dat een functie wordt aangeroepen als `<variabele> = <functie>()`. De functie maakt een berekening, die wordt opgeslagen in een `<resultaat_variabele>`. Het commando **`return`** `<resultaat_variabele>` beëindigt de functie, waarna de waarde die in `<resultaat_variabele>` staat "uit de functie komt." Omdat `<functie>()` was aangeroepen via een assignment, komt de waarde van `<resultaat_variabele>` uiteindelijk terecht in `<variabele>`.

Ik snap dat dit wat ingewikkeld klinkt, maar het wordt waarschijnlijk duidelijk als je het volgende voorbeeld bestudeert:

listing0803.py

```
from math import sqrt

def pythagoras( a, b ):
    return sqrt( a*a + b*b )
```

```
c = pythagoras( 3, 4 )
print( c )
```

De functie `pythagoras()` berekent de wortel van de som van de kwadraten van de parameters. De uitkomst wordt via een **`return`** statement geretourneerd. Het hoofdprogramma "vangt" de waarde door het toekennen van de waarde aan de variabele `c`. Daarna wordt de inhoud van `c` geprint.

Merk op dat het **`return`** statement in bovenstaande voorbeeld een complete berekening meekrijgt. Die berekening wordt binnen de functie uitgevoerd, en slechts de waarde die de uitkomst is van die berekening wordt geretourneerd naar het hoofdprogramma.

Stel je nu voor dat je deze berekening alleen wilt uitvoeren met positieve getallen (wat niet gek zou zijn, aangezien de functie duidelijk bedoeld is om de lengte van de schuine zijde van een rechthoekige driehoek te berekenen, en wie heeft er nu ooit gehoord van een driehoek met zijden die nul of minder lang zijn). Bestudeer de volgende code:

listing0804.py

```
from math import sqrt

def pythagoras( a, b ):
    if a <= 0 or b <= 0:
        return
    return sqrt( a*a + b*b )

print( pythagoras( 3, 4 ) )
print( pythagoras( -3, 4 ) )
```

Op het eerste gezicht is er niks mis met deze code: aangezien er niks te berekenen is voor negatieve getallen, retourneert het geen waarde als er een negatief argument wordt verstrekt. Echter, als je het programma uitvoert zie je dat het de speciale waarde **`None`** afdrukt. Ik heb deze speciale waarde besproken in hoofdstuk 5. Het hoofdprogramma verwacht dat de functie `pythagoras()` een getal teruggeeft dat afgedrukt kan worden, dus `pythagoras()` voldoet niet aan de verwachting aangezien het niets retourneert in bepaalde omstandigheden. Je moet er altijd heel duidelijk over zijn welk data type je functie retourneert, en ervoor zorgen dat een retourwaarde van dat type ook altijd terugkomt, ongeacht de omstandigheden.

De volgende code is overigens equivalent met bovenstaande code (en bevat dus dezelfde fout):

listing0805.py

```
from math import sqrt

def pythagoras( a, b ):
    if a > 0 and b > 0:
        return sqrt( a*a + b*b )

print( pythagoras( 3, 4 ) )
print( pythagoras( -3, 4 ) )
```

In deze code is niet expliciet zichtbaar dat er een **return** is zonder waarde erachter, maar hij is er wel. Als Python de laatste regel van een functie uitvoert, dan volgt daarna impliciet een **return**, en zal Python dus uit de functie retourneren zonder waarde.

Wellicht vraag je je af wat je moet retourneren in omstandigheden waarvoor je geen goede retourwaarde hebt. Dat hangt af van de toepassing. Bijvoorbeeld, voor de functie `pythagoras()` zou je kunnen besluiten dat je -1 retourneert als er iets niet in orde is met de argumenten. Zolang je dat maar communiceert naar de gebruiker van de functie, kan de gebruiker in het hoofdprogramma dit soort uitzonderlijke omstandigheden naar wens afhandelen. Bijvoorbeeld:

listing0806.py

```
from math import sqrt
from pcinput import getInteger

def pythagoras( a, b ):
    if a <= 0 or b <= 0:
        return -1
    return sqrt( a*a + b*b )

num1 = getInteger( "Geef zijde 1: " )
num2 = getInteger( "Geef zijde 2: " )
num3 = pythagoras( num1, num2 )
if num3 < 0:
    print( "De getallen kunnen niet worden gebruikt." )
else:
    print( "De lengte van de diagonaal is", num3 )
```

Merk op dat iedere regel code die in een functie volgt na een **return** op het zelfde niveau van inspringing altijd genegeerd zal worden. Bijvoorbeeld, in de functie:

```
from math import sqrt

def pythagoras( a, b ):
    if a <= 0 or b <= 0:
        return -1
        print( "Deze regel wordt nooit uitgevoerd" )
    return sqrt( a*a + b*b )
```

geeft de regel onder **return** `-1` duidelijk aan hoe nutteloos hij is.

8.2.6 Het verschil tussen `return` en `print`

In het verleden heb ik diverse malen geconstateerd dat veel studenten moeite hebben met het verschil tusen een functie die een waarde retourneert, en een functie die een waarde print. Vergelijk de volgende twee stukken code:

```
def print3():
   print( 3 )
print3()
```

en:

```
def return3():
   return 3
print( return3() )
```

Zowel de functie `print3()` als de functie `return3()` worden aangeroepen in hun respectievelijke hoofdprogramma's, en beide resulteren in het printen van de waarde 3. Het verschil is dat bij `print3()` dit printen gebeurt in de functie en de functie niks retourneert, terwijl bij de functie `return3()` de waarde 3 wordt geretourneerd, en geprint in het hoofdprogramma. Voor de gebruiker lijkt er geen verschil te zijn: beide programma's printen 3. Voor een programmeur zijn beide functies echter compleet verschillend.

De functie `print3()` kan voor slechts één doel gebruikt worden, namelijk het tonen van het getal 3. De functie `return3()` kan echter gebruikt worden waar je ook maar het getal 3 nodig hebt: om het te tonen, om het te gebruiken in een berekening, of om het aan een variabele toe te kennen. Bijvoorbeeld, de volgende code verheft 2 tot de macht 3 en print de uitkomst:

```
def return3():
   return 3
x = 2 ** return3()
print( x )
```

De code hieronder, echter, geeft een runtime error:

```
def print3():
    print( 3 )
x = 2 ** print3() # Runtime error!
print( x )
```

De reden is dat hoewel `print3()` het getal 3 op het scherm toont (je ziet het boven de runtime error als je de code uitvoert), het niet de waarde 3 produceert op een manier dat een berekening er gebruik van kan maken. De functie `print3()` geeft de speciale waarde **`None`**, en die kan niet in een berekening gebruikt worden.

Dus als je een functie wilt maken die een waarde oplevert die elders in je code gebruikt moet worden, dan moet die functie de waarde middels een **`return`** retourneren. Als je een functie wilt maken die informatie toont, dan kun je dat gewoon doen middels print statements in de functie, en hoeft de functie niets te retourneren.

8.2.7 De functie machine

Als je nog steeds moeite hebt met begrijpen hoe functies werken, denk dan als volgt:

Een functie is net een grote machine, bijvoorbeeld, een machine die pannenkoeken bakt. Er zijn input sleuven aan de bovenkant, met labels "melk," "eieren," en "meel." Dat zijn de functie parameters. Je kunt beïnvloeden wat voor pannenkoeken de machine bakt door de juiste ingrediënten in de inputs te stoppen; bijvoorbeeld, als je volkoren pannenkoeken wilt, doe je volkoren meel in de "meel" input. Natuurlijk kunnen de zaken dramatisch fout

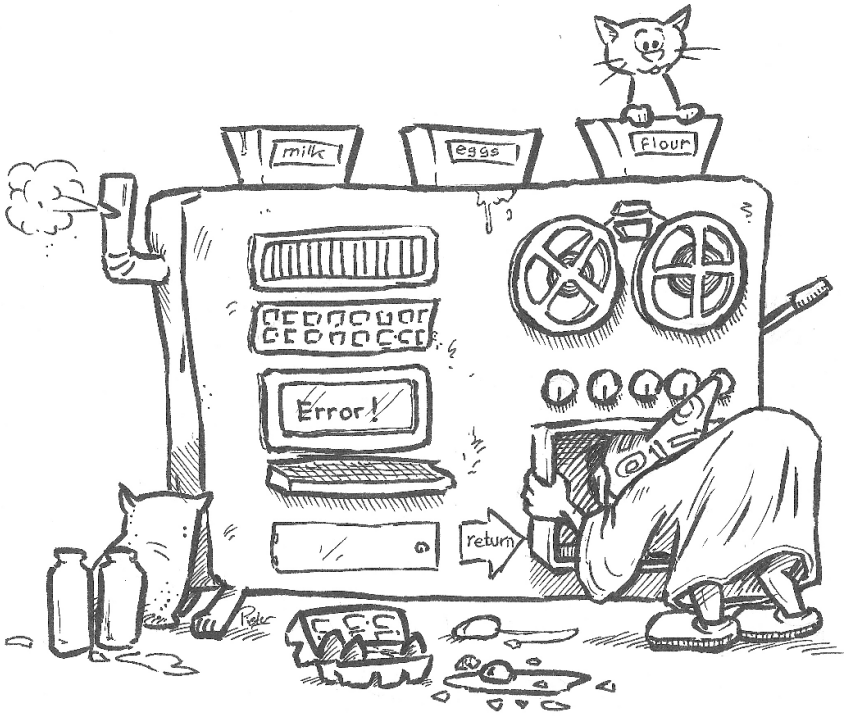
lopen als je eieren in de "melk" input doet – of als je probeert een kat in de "meel" input te stoppen.

Als de inputs gevuld zijn, begint de machine te ratelen. Je wacht geduldig naast de output sleuf die (tot niemand's verrassing) het label "return" draagt. Na enige tijd verschijnt er een pannenkoek in de uitvoer. Die pannenkoek is de retourwaarde die de machine heeft geproduceerd.

De machine heeft ook een display. Als je iets incorrects in de machine stopt, komt daar wellicht een boodschap op in de trant van "Kat zit vast in machine, herstarten alsjeblieft." Dit display is waar alles dat in de machine geprint wordt verschijnt.

Je snapt wel dat het zinloos is als, nadat je de juiste ingrediënten geladen hebt, de machine alleen maar op de display toont "Pannenkoek gereed!" Je wilt een echte pannenkoek. Daarom moet de machine de pannenkoek aanreiken via de "return," waarvandaan jij hem kunt aannemen en "assignen" aan je lunch. Een tekst alleen is onvoldoende.

Een van de aardige dingen van de pannenkoek machine is dat zelfs als jij niet weet hoe je een pannenkoek moet bakken, je toch pannenkoeken kunt krijgen als je maar de juiste ingrediënten verstrekt. Dat is ook aardig aan functies: ze kunnen complexe zaken voor je regelen, zonder dat je hoeft te weten hoe ze dat doen.



8.2.8 Meerdere retourwaardes

Je bent niet beperkt in je functies tot slechts één retourwaarde. Je kunt meerdere waardes in één keer retourneren door er komma's tussen te zetten. Als je deze waardes wilt gebruiken in je programma na aanroep van de functies, moet je ze toekennen aan meerdere variabelen. Die zet je allemaal aan de linkerkant van de assignment operator, ook met komma's ertussen. Dit kan ik het beste illustreren aan de hand van een voorbeeld:

listing0807.py

```
import datetime

def plusDagen( jaar, maand, dag, increment ):
    startdatum = datetime.datetime( jaar, maand, dag )
    einddatum = startdatum + datetime.timedelta( days=increment )
    return einddatum.year, einddatum.month, einddatum.day

y, m, d = plusDagen( 2015, 11, 13, 55 )
print( "{}/{}/{}".format( y, m, d ) )
```

De functie `plusDagen()` krijgt vier argumenten, namelijk integers die een jaar, een maand, een dag, en een aantal dagen die je wilt optellen bij de datum die wordt weergegeven door de eerste drie argumenten. Er worden drie waardes geretourneerd, namelijk een nieuw jaar, een nieuwe maand, en een nieuwe dag. De code stopt die in de variabelen `y`, `m`, en `d`.

Als je de code hierboven bestudeert, vraag je je misschien af hoe `plusDagen()` precies werkt. Zoals ik al zei, het is een voordeel van functies dat zolang ze hun werk doen en je weet wat de argumenten zijn en wat er geretourneerd wordt, dat je niet hoeft te weten hoe de functie in elkaar zit. Je kunt de functie gebruiken zonder kennis van het interne proces. Dus je mag gewoon negeren wat je in `plusDagen()` ziet staan (overigens gebruikt de code van `plusDagen()` de `datetime` module, die aan de orde komt in hoofdstuk 27).

8.2.9 Functies aanroepen vanuit functies

Functies mogen andere functies aanroepen, zolang die andere functies maar bekend zijn bij de aanroepende functie. Bijvoorbeeld, hieronder zie je hoe de functie `afstand()` de functie `pythagoras()` gebruikt om de afstand te berekenen tussen twee punten in een 2-dimensionale ruimte.

listing0808.py

```
from math import sqrt

def pythagoras( a, b ):
    if a <= 0 or b <= 0:
        return -1
    return sqrt( a*a + b*b )

def afstand( x1, y1, x2, y2 ):
    return pythagoras( abs( x1 - x2 ), abs( y1 - y2 ) )

print( afstand( 1, 1, 4, 5 ) )
```

`afstand()` kent `pythagoras()`, omdat `pythagoras()` gedefinieerd is vóór `afstand()` is aangeroepen.

Als je wilt, kun je functies *in* andere functies stoppen; met andere woorden, je kunt functies "nesten." Bijvoorbeeld, je kunt de functie `pythagoras()` in de functie `afstand()` stoppen.

Dat betekent dat `pythagoras()` kan worden aanroepen vanuit `afstand()`, maar niet op andere plekken in de code.

listing0809.py

```
from math import sqrt

def afstand( x1, y1, x2, y2 ):

    def pythagoras_binnen( a, b ):
        if a <= 0 or b <= 0:
            return -1
        return sqrt( a*a + b*b )

    return pythagoras_binnen( abs( x1 - x2 ), abs( y1 - y2 ) )

print( afstand( 1, 1, 4, 5 ) )
# print( pythagoras_binnen( 3, 4 ) )
```

Het gebruik van geneste functies is wat uitzonderlijk, maar het is mogelijk.

Merk op: Als je de hash mark verwijdert voor de laatste regel in de code hierboven, wordt er een aanroep van `pythagoras_binnen()` toegevoegd aan de code. Bij uitvoering van het programma geeft dat een runtime error, omdat `pythagoras_binnen()` alleen zichtbaar is in de functie `afstand()`.

Opgave Schrijf een functie `printx()` die alleen de letter "x" print. Schrijf daarna een functie `meerderex()` die als argument een integer krijgt en die zo vaak de letter "x" print als de integer aangeeft. Daartoe roept de functie `meerderex()` zo vaak als nodig de functie `printx()` aan.

8.2.10 Functienamen

Het is de gewoonte om namen van functies niet te laten beginnen met een underscore (zulke functienamen zijn voorbehouden aan de ontwikkelaars van Python zelf), en dat je probeert alleen kleine letters te gebruiken. Als een functienaam uit meerdere woorden bestaat, kun je ofwel underscores tussen die woorden zetten, ofwel ieder woord behalve het eerste laten starten met een hoofdletter. Verschillende programmeurs hebben hier verschillende meningen over, maar in de praktijk maakt het niet zoveel uit omdat je een functie altijd kunt herkennen aan het feit dat er haakjes achter de functienaam staan.

Bepaalde benamingen van functies zijn typerend voor bepaalde functionaliteiten.

Een functie die test of een bepaald item een bepaalde eigenschap heeft, en die dan **`True`** of **`False`** retourneert afhankelijk van het feit of de eigenschap wel of niet aanwezig is, krijgt meestal een naam die begint met het woord `is`, gevolgd door de naam van de eigenschap die start met een hoofdletter. Bijvoorbeeld, een functie die test of een getal even is, zou de naam `isEven()` krijgen.

Opgave Schrijf de functie `isEven()`.

Opgave Schrijf de functie `isOneven()`, die bepaalt of een getal oneven is, door de functie `isEven()` aan te roepen en het resultaat te inverteren.

Merk op: Als je een functie als `isEven()` wilt gebruiken in een conditionele expressie, bijvoorbeeld, als je een actie alleen wilt uitvoeren als een getal even is, hoef je niet te schrijven `if isEven( num ) == True:`. Je hoeft alleen maar te schrijven `if isEven( num ):`, want de functie retourneert al **`True`** of **`False`**. Een is-functie op zo'n manier gebruiken verhoogt de leesbaarheid van een programma.

De naam van een functie die de waarde van een attribuut opvraagt en retourneert, begint over het algemeen met het woord `get`, gevolgd door de naam van de eigenschap, beginnend met een hoofdletter. Bijvoorbeeld, een functie die van een float alleen het fractionele gedeelte (de cijfers achter de komma) retourneert, zou heten `getFractie()`.[9]

Opgave Schrijf de functie getFractie().

De tegenhanger van een "get" functie is een functie die een eigenschap een bepaalde waarde geeft. De naam van zo'n functie begint meestal met `set`, en is voor de rest gelijk aan een `get` functie. Ik kan op dit punt geen voorbeeld van een `set` functie geven, aangezien ik nog niet heb uitgelegd hoe een functie een waarde aan iets kan geven, aangezien functies de waardes van de variabelen die als argumenten zijn meegegeven niet kunnen wijzigen (tenminste, niet voor de data types die tot nu toe zijn geïntroduceerd). Dit volgt in een later hoofdstuk.

Als je je aan dergelijke functiebenamingen houdt, wordt je code beter leesbaar.

8.2.11 Functie commentaar

In alle hoofdstukken tot nu toe heb ik weinig commentaar in code geschreven. Boeken en cursussen die programmeren doceren moedigen studenten vaak aan om commentaar aan code toe te voegen. Ikzelf ben van mening dat code "zelf-documenterend" moet zijn, dat wil zeggen, dat je gemakkelijk aan code kunt zien wat het doet door het te lezen. Je kunt dat bereiken door goede namen te kiezen voor variabelen en functies, door gebruik te maken van spaties en witregels, door nette inspringing (wat gelukkig bij Python een noodzaak is), en door geen gebruik te maken van "slimme" truukjes die je code een beetje sneller maken ten koste van leesbaarheid, enkel en alleen om te laten zien hoe slim je bent.[10]

Hoewel ik commentaar zie als iets extra's dat je alleen moet gebruiken als er een noodzaak is om je code uit te leggen, is mijn opinie over commentaar bij functies anders. Het idee achter een functie is dat de gebruiker van de functie de code van de functie niet hoeft te kennen. Daarom moet hetgeen de functie doet en hoe de functie werkt worden uitgelegd middels commentaar, geschreven meteen boven de functienaam.

In commentaar bij een functie moet je drie zaken uitleggen:

- Wat de functie doet

[9] "Get" is uiteraard de Engelse term voor "nemen." Je mag hier in het Nederlands dan ook het woord "neem" voor gebruiken in plaats van "get."

[10] Een kleine anecdote wat dit betreft: tijdens mijn dagen als professioneel programmeur hoorde ik een collega eens een andere collega ophemelen door te zeggen: "Hij is zo briljant, als ik zijn code zie snap ik er niks van!" Als iemand dat over mijn code zou zeggen zou ik me diep schamen.

- Welke argumenten de functie nodig heeft of accepteert, inclusief data types
- Wat de functie retourneert, inclusief data types

Als een functie neveneffecten heeft, dus als een functie zaken buiten de functie beïnvloedt, dan moet dat ook heel duidelijk in het functiecommentaar staan. Ik heb dat niet in het lijstje hierboven genoemd, omdat een functie *geen neveneffecten zou mogen hebben.*

Voor de antwoorden bij de opgaves in dit hoofdstuk heb ik commentaar aan de functies toegevoegd op een manier die ik acceptabel acht. In volgende hoofdstukken doe ik dat niet altijd, omdat ik de functies vaak in de tekst bediscussieer, of omdat ik vind dat je de inhoud van een functie moet bestuderen. Maar ik schrijf wel altijd commentaar bij functies die ik voor andere doeleinden gebruik.

8.3 Scope en levensduur

Scope (dat vertaald kan worden als "bereik," maar dat een zo'n verwarrende vertaling dat hij niet gebruikt wordt) refereert aan "zichtbaarheid." Specifiek, waar het variabelen betreft, refereert het aan in welke delen van een programma een variabele zichtbaar is en gewijzigd kan worden. Levensduur refereert aan hoe lang een variabele in het geheugen van de computer blijft. Levensduur is gerelateerd aan scope.

8.3.1 Scope van variabelen

De scope van een variabele is het blok code waarin de variabele is gecreëerd, en alle blokken code die genest zijn binnen dat blok code op een dieper niveau van inspringing. Bijvoorbeeld:

listing0810.py

```
hallo = "Hallo!"
totziens = "Tot ziens!"

for i in range( 3 ):
    for j in range( 2 ):
        middag = "Goedemiddag"
        print( totziens )
    print( j )
    print( hallo )
    print( middag )

print( i )
print( j )
print( middag )
```

De variabelen `hallo` en `totziens` zijn gedefinieerd op het hoogste niveau van inspringing van het programma, wat betekent dat hun scope het hele programma is. De variabelen `i`, `j`, en `middag` zijn gedefinieerd in blokken code op een dieper niveau van inspringing. In de meeste programmeertalen zou dit betekenen dat hun scope beperkt is tot die diepere

niveaus, maar Python is vriendelijk op dit gebied en laat hun scope verder reiken dan het blok code waarin de variabelen gecreëerd zijn. Daarom zijn in dit programma de variabelen zichtbaar in het hele programma, na hun creatie.

Hoe werkt dit met functies?

listing0811.py

```
dubbeltje = "dubbeltje"

def geboren():
    print( "Als je voor een", dubbeltje, "geboren wordt,",
        "dan wordt je nooit een", kwartje )

kwartje = "kwartje"
geboren()
print( "dubbeltje =", dubbeltje, "en kwartje =", kwartje )
```

Zowel `dubbeltje` als `kwartje` zijn zichtbaar in de functie `geboren()`, omdat ze gedefinieerd zijn voordat de functie is aangeroepen, en omdat het blok code van de functie op een dieper niveau van inspringing zit. Maar kijk nu naar het volgende programma, dat een kleine wijziging bevat ten opzichte van het vorige:

listing0812.py

```
dubbeltje = "dubbeltje"

def geboren():
    kwartje = "stuiver"
    print( "Als je voor een", dubbeltje, "geboren wordt,",
        "dan wordt je nooit een", kwartje )

kwartje = "kwartje"
geboren()
print( "dubbeltje =", dubbeltje, "en kwartje =", kwartje )
```

Voer dit programma uit, bestudeer de code en output, en vergelijk ze met de code en output van het voorgaande programma. De variabele `kwartje` lijkt een nieuwe waarde te krijgen in de functie `geboren()`, wat ertoe leidt dat de functie nu claimt dat je na je geboorte niet minder waard kunt worden. Maar als daarna in het hoofdprogramma de waarde van `kwartje` wordt afgedrukt, zie je dat deze variabele nog steeds het woord `"kwartje"` bevat.

De reden voor het verschil is dat de variabele `kwartje` in de functie een andere is dan de variabele `kwartje` in het hoofdprogramma. Door in een functie een waarde toe te kennen aan een variabele, wordt een nieuwe, "lokale" variabele gecreëerd. En deze variabele wordt dan gebruikt voor de rest van de functie. De originele variabele `kwartje` bestaat nog steeds, maar is onzichtbaar geworden voor de functie omdat de functie een eigen variabele `kwartje` heeft gemaakt.

De levensduur van de variabele `kwartje` in de functie is de periode waarvoor het blok code van de functie wordt uitgevoerd. Zodra de functie eindigt (bijvoorbeeld omdat er een **return** in de functie staat of omdat de laatste regel van de functie is uitgevoerd), worden

de lokale variabelen van de functie vernietigd. Ze staan niet langer in het geheugen van de computer, en kunnen niet meer benaderd worden.

listing0813.py

```
appel = "appel"
banaan = "banaan"
kers = "kers"
doerian = "doerian"

def printfruit_1():
    print( appel, banaan )

def printfruit_2( appel ):
    banaan = kers
    print( appel, banaan )

def printfruit_3( appel, banaan ):
    kers = "mango"
    banaan = kers
    print( appel, banaan )

printfruit_1()
printfruit_2( kers )
printfruit_3( kers, doerian )

print( "appel =", appel )
print( "banaan =", banaan )
print( "kers =", kers )
print( "doerian =", doerian )
```

Voer deze code uit en bestudeer hem nauwlettend.

De drie functies `printfruit_1()`, `printfruit_2()`, en `printfruit_3()` printen de waardes van de variabelen `appel` en `banaan`.

In `printfruit_1()` worden de variabelen `appel` en `banaan` geprint die gedefinieerd zijn buiten de functie, aangezien de functie zelf geen variabelen met deze namen definieert.

In `printfruit_2()`, is `appel` een parameter van de functie, wat betekent dat de variabele lokaal is voor de functie en, omdat het een parameter is, zijn waarde krijgt van buiten de functie. De waarde de de parameter krijgt is de waarde van `kers`. `banaan` is een variabele die zijn waarde krijgt in de functie. Dit is daarom een nieuwe, lokale variabele `banaan`, die niks te maken heeft met de variabele `banaan` in het hoofdprogramma. Deze variabele krijgt de waarde van de variabele `kers`, die niet lokaal bekend is in de functie, dus wordt de variabele `kers` van het hoofdprogramma gebruikt. Dus krijgt de lokale variabele `banaan` de waarde “kers,” en dat is dus de waarde die geprint wordt.

In `printfruit_3()` zijn `appel` en `banaan` beide parameters, dus beide zijn lokaal voor de functie en krijgen hun initiële waarde bij de aanroep van de functie. De functie creëert een lokale variabele `kers`, die onafhankelijk is van de variabele `kers` uit het hoofdprogramma, en geeft deze variabele de waarde `"mango"`. De variabele `banaan` krijgt vervolgens

de waarde van `kers`. Omdat al deze wijzigingen gemaakt worden met lokale variabelen, hebben ze geen invloed op de waardes van de variabelen in het hoofdprogramma.

Waar het hier vooral om gaat is het feit dat, na de aanroep van functies die op allerlei manieren met de parameters spelen en die lokale variabelen aanmaken met dezelfde namen als variabelen in het hoofdprogramma, uit het afdrukken van de variabelen in het hoofdprogramma blijkt dat die allemaal nog steeds dezelfde waarde hebben als ze hadden voordat de functies werden aangeroepen. Zodra je probeert in een functie een variabele die bestaat in het hoofdprogramma een andere waarde te geven middels een assignment, wordt in plaats daarvan een nieuwe, lokale variabele gecreëerd. Een dergelijke lokale variabele is compleet onafhankelijk van het hoofdprogramma. De levensduur van zo'n lokale variabele is de periode dat de functie wordt uitgevoerd. Parameters kun je ook beschouwen als lokale variabelen.

Dit is een enorm krachtige eigenschap van functies: ze hoeven geen rekening te houden met variabelen die bestaan buiten de functie, aangezien iedere variabele die ze creëren lokaal is voor de functie.

8.3.2 Globale variabelen

Ik liet hierboven zien dat variabelen die buiten een functie zijn gecreëerd zichtbaar zijn in een functie zolang er maar niet een variabele met dezelfde naam in de functie wordt gecreëerd. Variabelen van het hoofdprogramma, die zichtbaar zijn in alle functies, worden wel "globale" variabelen genoemd, als tegenhanger van "lokale" variabelen die alleen binnen een functie zichtbaar zijn.

Het is een goede gewoonte om functies onafhankelijk te maken van het hoofdprogramma, dat wil zeggen, ze geen gebruik te laten maken van globale variabelen. Als je waardes van buiten een functie beschikbaar wilt maken voor de functie, kun je dat doen middels parameters. Een uitzondering kan gemaakt worden voor globale variabelen die als constanten voor een programma gebruikt worden (zie hoofdstuk 4). Als je een functie aan een constante laat refereren, zorg er dan wel voor dat het duidelijk is voor eenieder die de functie bekijkt dat het inderdaad een constante is, dus dat de naam van de variabele volledig uit hoofdletters bestaat.

Je vraagt je misschien af of het mogelijk is de waarde van globale variabelen te wijzigen in een functie. Dat is mogelijk, maar dan moet je in de functie expliciet duidelijk maken dat je wilt dat een wijziging van een variabele effect moet hebben op een globale variabele. Daarvoor is het gereserveerde woord **`global`** bedoeld. Het statement **`global`** `<variabele>` geeft aan dat de variabele die je noemt een variabele is die in het hoofdprogramma met de opgegeven naam gedefinieerd is. Bijvoorbeeld:

```
fruit = "appel"

def wijzigFruit():
    global fruit
    fruit = "banaan"

print( fruit )
wijzigFruit()
print( fruit )
```

Hoewel het mogelijk is de waarde van globale variabelen in functies te wijzigen, raad ik deze constructie ten zeerste af aangezien het de functie afhankelijk maakt van het hoofdprogramma (en dus is het niet langer een "pure" functie). In essentie maak je op deze manier een functie met neveneffecten, en (nu allen in koor:) *een functie mag geen neveneffecten hebben*.

Het is nimmer nodig dat een functie globale variabelen gebruikt. Als je wilt dat een functie globale variabelen wijzigt, laat de functie dan de nieuwe waarde voor de globale variabele retourneren, zodat die aan de globale variabele kan worden toegekend in het hoofdprogramma. Het hoofdprogramma mag dan beslissen of het een variabele die van het hoofdprogramma zelf is overschrijft. De enige reden dat ik het gereserveerde woord **`global`** hier noem is dat ik soms zie dat studenten er hun toevlucht toe nemen als ze onvoldoende begrip hebben van wat **`return`** doet. Het ontkennen van het bestaan van **`global`** is geen effectieve aanpak, ik geef liever toe dat het bestaat en waarschuw studenten dat ze het niet moeten gebruiken.

8.4 Grip krijgen op complexiteit

Stel dat Python geen ingebouwde functies **`max`**`()` en **`min`**`()` zou hebben, en je hebt geen kennis van wat er in de hoofdstukken hierna volgt. Ik geef je de volgende opdracht:

Schrijf een programma dat twee groepen van drie getallen krijgt (je mag het programma schrijven voor specifieke getallen, maar je voegt later toe dat de gebruiker deze getallen ingeeft). Het programma telt de waardes van de kleinste getallen van iedere groep op, en ook zo voor de waardes van de middelste getallen, en de waardes van de grootste getallen. Daarna drukt het de uitkomsten af.

Hoe los je dit op? Je kunt beginnen met iets als:

```
# Eerst krijgen de variabelen in groep 1 (num11, num12, num13) en
# groep 2 (num21, num22, num23) een waarde.

kleinste1 = 0
kleinste2 = 0
middelste1 = 0
middelste2 = 0
grootste1 = 0
grootste2 = 0

if num11 < num12:
    if num11 < num13:
        kleinste1 = num11
    else:
        kleinste1 = num13
elif num12 < num13:
    kleinste1 = num12
else:
    kleinste1 = num13

print( kleinste1 ) # Test
```

```
# Deze code zoekt de kleinste van groep 1.
# Daarna doe je hetzelfde voor de kleinste van groep 2.
# Dan moet je ook zoiets doen voor de grootste van de 2 groepen.
# Daarna moet je nog iets verzinnen voor de middelste.
# Tenslotte doe je alle optellingen en drukt de resultaten af.
```

Je kunt je voorstellen dat deze aanpak, met geneste **if** statements die zes keer herhaald worden met verschillende assignments in de takken, leidt tot een behoorlijk groot, onleesbaar, onbeheersbaar programma waarvan het lastig te zien is of het correct is of niet. Je moet het probleem op een slimmere manier aanpakken.

Stel je voor dat je een functie hebt die de kleinste van drie getallen bepaalt, een functie die de middelste van drie getallen bepaalt, en een functie die de grootste van drie getallen bepaalt. Dan is het programma triviaal geworden. Het is dan iets als:

listing0814.py

```
num11, num12, num13 = 436, 178, 992
num21, num22, num23 = 880, 543, 101

def kleinste( n1, n2, n3 ):
    return n1 # Geef iets terug

def middelste( n1, n2, n3 ):
    return n1 # Geef iets terug

def grootste( n1, n2, n3 ):
    return n1 # Geef iets terug

print( "som van kleinste =", kleinste( num11, num12, num13 ) +
    kleinste( num21, num22, num23 ) )
print( "som van middelste =", middelste( num11, num12, num13 ) +
    middelste( num21, num22, num23 ) )
print( "som van grootste =", grootste( num11, num12, num13 ) +
    grootste( num21, num22, num23 ) )
```

Opmerking: In de code hierboven heb ik een meervoudige assignment gebruikt om de variabelen num*xx* een waarde te geven, om de lengte van de code te reduceren. Hierbij kun je in één statement meerdere variabele een waarde geven door links van het is-gelijk-teken een aantal variabelen te zetten, en rechts ervan evenveel waardes. De eerste waarde gaat naar de eerste variabele, de tweede waarde naar de tweede variabele, etcetera. Dit wordt verder uitgelegd in hoofdstuk 11.

Het programma hierboven is leesbaar, begrijpbaar, en kan al getest worden. Natuurlijk, de functies `kleinste()`, `middelste()`, en `grootste()` retourneren niet de correcte waardes. Misschien weet je zelfs nog niet eens hoe je ze zou moeten implementeren. Maar je voelt waarschijnlijk wel aan dat ze geïmplementeerd kunnen worden, en je weet dat je de code voor deze functies later kunt schrijven, en in kleine stapjes.

Hoe implementeer je `kleinste()`? Zoals ik boven liet zien is het lastig om dit met een geneste **if** te doen (als je dat niet gelooft, kijk dan niet naar mijn code en probeer het zelf

te schrijven; het blijkt moeilijk te zijn om de variabelen in gedachten te houden terwijl je de geneste **if** schrijft). Kan dit misschien op een wat leesbaardere manier gedaan worden?

Is het lastig om de kleinste van twee getallen te retourneren? Nee, dat is heel gemakkelijk:

```
def kleinste_van_twee( n1, n2 ):
    if n1 < n2:
        return n1
    return n2
```

Door een dergelijke functie te nesten, kun je de een `kleinste()` functie maken die de kleinste van drie getallen bepaalt. Dat kun je ook doen voor `grootste()`. Het programma wordt dan:

listing0815.py

```
num11, num12, num13 = 436, 178, 992
num21, num22, num23 = 880, 543, 101

def kleinste_van_twee( n1, n2 ):
    if n1 < n2:
        return n1
    return n2

def grootste_van_twee( n1, n2 ):
    if n1 > n2:
        return n1
    return n2

def kleinste( n1, n2, n3 ):
    return kleinste_van_twee( kleinste_van_twee( n1, n2 ), n3 )

def middelste( n1, n2, n3 ):
    return n1 # geef iets terug

def grootste( n1, n2, n3 ):
    return grootste_van_twee( grootste_van_twee( n1, n2 ), n3 )

print( "som van kleinste =", kleinste( num11, num12, num13 ) +
    kleinste( num21, num22, num23 ) )
print( "som van middelste =", middelste( num11, num12, num13 ) +
    middelste( num21, num22, num23 ) )
print( "som van grootste =", grootste( num11, num12, num13 ) +
    grootste( num21, num22, num23 ) )
```

Dit programma werkt voor de kleinste getallen en de grootste getallen. Om het af te maken, moet nog iets bepaald worden voor de middelste. Wat is het middelste van drie getallen? Dat is het getal dat overblijft als je de kleinste en de grootste weghaalt. Kun je dit programmeren? Ik stel een functie `verwijder_twee_van_drie()` voor, die eerst de kleinste, en dan de grootste van de twee overgebleven getallen verwijdert (wat weer gelijk is aan

de eerder bepaalde grootste). Om `verwijder_twee_van_drie()` gemakkelijk te kunnen bouwen, maak ik ook `verwijder_een_van_drie()` en `verwijder_een_van_twee()`.

listing0816.py

```
num11, num12, num13 = 436, 178, 992
num21, num22, num23 = 880, 543, 101

def kleinste_van_twee( n1, n2 ):
    if n1 < n2:
        return n1
    return n2

def grootste_van_twee( n1, n2 ):
    if n1 > n2:
        return n1
    return n2

def verwijder_een_van_drie( n1, n2, n3, verwijder ):
    if n1 == verwijder:
        return n2, n3
    elif n2 == verwijder:
        return n1, n3
    return n1, n2

def verwijder_een_van_twee( n1, n2, verwijder ):
    if n1 == verwijder:
        return n2
    return n1

def verwijder_twee_van_drie( n1, n2, n3, verwijder1, verwijder2):
    num1, num2 = verwijder_een_van_drie( n1, n2, n3, verwijder1 )
    return verwijder_een_van_twee( num1, num2, verwijder2 )

def kleinste( n1, n2, n3 ):
    return kleinste_van_twee( kleinste_van_twee( n1, n2 ), n3 )

def middelste( n1, n2, n3 ):
    return verwijder_twee_van_drie( n1, n2, n3,
        kleinste( n1, n2, n3 ), grootste( n1, n2, n3 ) )

def grootste( n1, n2, n3 ):
    return grootste_van_twee( grootste_van_twee( n1, n2 ), n3 )

print( "som van kleinste =", kleinste( num11, num12, num13 ) +
    kleinste( num21, num22, num23 ) )
print( "som van middelste =", middelste( num11, num12, num13 ) +
    middelste( num21, num22, num23 ) )
print( "som van grootste =", grootste( num11, num12, num13 ) +
    grootste( num21, num22, num23 ) )
```

Het programma is nu klaar en het werkt. Het is best lang, maar alle functies zijn gemakkelijk te begrijpen, en het is ook niet moeilijk om te zien hoe het programma als geheel werkt. Het is nog steeds een stuk korter dan de allereerste poging, met zes geneste **`if`** statements, en het is een stuk leesbaarder.

Er zijn natuurlijk andere manieren om het programma aan te pakken. Met een beetje inventiviteit kom je waarschijnlijk op slimmere manieren om de kleinste, middelste, en grootste te bepalen (ik ben nog niet echt tevreden over mijn aanpak van de middelste). Maar het programma werkt en is begrijpbaar, en dat is het belangrijkste.

Je kunt de aanpak die ik gekozen heb bekritiseren. Bijvoorbeeld, de berekening van de kleinste van de drie wordt twee keer uitgevoerd: de eerste keer om de kleinste te bepalen, en de tweede keer om de middelste te bepalen. Dat geldt ook voor de grootste. Kan dat geoptimaliseerd worden, zodat deze bepalingen slechts één keer plaatsvinden? Natuurlijk kan dat, bijvoorbeeld door twee extra variabelen op te nemen die de kleinste en grootste bijhouden. Maar waarom zou ik dat doen? Dat maakt het programma niet leesbaarder, en hoewel het het programma een beetje sneller maakt, spreken we daarbij over nanoseconden. Voor een programma als dit is snelheid niet belangrijk en volledig ondergeschikt aan leesbaarheid. Laat me nogmaals benadrukken dat het oplossen van een probleem op de eerste plaats komt, onmiddellijk gevolgd door het oplossen van het probleem op een leesbare en beheersbare manier. Efficiëntie komt pas veel later.

Wat je hiervan moet leren is dat als een programma bestaat uit een serie problemen die je moeilijk op kunt lossen, je het moet proberen op te splitsen in sub-problemen en subdoelen, die je onafhankelijk van elkaar kunt aanpakken. Je kunt voor ieder van die subproblemen alvast een functie introduceren als je het programma opzet, en voorlopig vul je die dan met iets simpels, bijvoorbeeld het retourneren van een vaste waarde. Je kunt dan in ieder geval je programma al testen. Later kun je dan beginnen met het één-voor-één invullen van ieder van die functies.

8.5 Modules

Het maken van een module is eenvoudig. Je maakt gewoon een Python bestand, met de extensie `.py`, en plaatst er de functies in die je in de module wilt hebben. Je kunt dit Python bestand dan importeren in een ander Python programma (je gebruikt gewoon de naam van het bestand zonder de extensie `.py`; het bestand moet in dezelfde folder als het programma staan, of in de folder waar Python altijd zoekt naar modules), en je kunt de functies gebruiken op dezelfde manier als je functies uit de reguliere Python modules gebruikt, dat wil zeggen, je kunt ofwel specifieke functies uit de module importeren, ofwel de hele module importeren en de functies gebruiken via de `<module>.<functie>()` syntax.

8.5.1 `main()`

Als je de code van andermans Python programma's bekijkt, zeker programma's die functies bevatten die je mogelijk zou willen importeren, zie je vaak een constructie als de volgende:

```
def main():
    # code...
if __name__ == '__main__':
    main()
```

De functie `main()` bevat feitelijk het hoofdprogramma, dat andere functies kan aanroepen.

Je hoeft dit niet precies te begrijpen, maar wat hier aan de hand is is het volgende: het Python bestand bevat code die als programma kan draaien, of de functies die het bevat kunnen geïmporteerd worden in andere programma's. De bovenstaande constructie zorgt ervoor dat de code in `main()` (dus het hoofdprogramma) alleen wordt uitgevoerd als het programma als een separaat programma is gestart, en niet als een module in een ander programma geladen is. Als in plaats daarvan het programma als module is geladen, kunnen alleen de functies in het programma worden geïmporteerd, en wordt de code voor `main()` genegeerd.

Een Python bestand dat een dergelijke constructie bevat en dat voornamelijk als module wordt gebruikt, heeft vaak code in `main()` die de functies test. Dat kan nuttig zijn tijdens de ontwikkeling van de module.

Deze constructie heeft echter nog een tweede toepassing. Omdat `main()` een functie is, hoef je als je het programma tussentijds wilt verlaten, niet de `exit()` functie uit de `sys` module te gebruiken. In plaats daarvan kun je gewoon **`return`** doen in de `main()` functie. Dit vermijdt de lelijke foutboodschap die sommige editors geven bij gebruik van `exit()`.

8.6 Anonieme functies

Het concept "anonieme functies" mag je als optioneel materiaal beschouwen: anonieme functies worden niet vaak gebruikt en zijn nooit nodig. Maar om het verhaal over functies compleet te krijgen, bediscussieer ik ze hier.

Python staat het toe om functies te creëren die geen naam hebben. De functie kan aan een variabele toegekend worden, en je kunt die variabele dan gebruiken alsof het een functie is. Je gebruikt hiervoor de volgende syntax:

```
lambda <parameters>: <actie>
```

`lambda` is een gereserveerd woord. `<parameters>` is a serie parameter namen, van elkaar gescheiden middels komma's als er meer dan één is. `<actie>` is één commando. De anonieme functie heeft geen **`return`**, maar de waarde van `<actie>` wordt gebruikt als retourwaarde.

Bijvoorbeeld, de volgende code creëert een anonieme functie die het kwadraat van de parameter berekent. De functie wordt toegekend aan de variabele `f`. `f` kan dan worden gebruikt als functie om de kwadraten van getallen te berekenen.

```
f = lambda x: x*x
print( f(12) )
```

Deze code is exact gelijk aan de volgende code:

```
def f( x ):
    return x*x
print( f(12) )
```

Dus anonieme functies zijn niet anders dan reguliere functies, maar beperkter aangezien ze slechts kunnen bestaan uit een enkele regel code. Waarom worden ze dan door Python

ondersteund? Er is heel wat gedebatteerd onder de mensen die Python ontwikkelen over de vraag of **`lambda`** in de taal moet blijven. Het is ooit opgenomen in Python omdat het ook zit in andere talen, specifieke in functionele programmeertalen als Lisp en Haskell, die bestaan bij gratie van het concept van anonieme functies. Maar **`lambda`** in Python is lang niet zo krachtig als **`lambda`** in die andere talen, en, zoals ik liet zien, eigenlijk overbodig. De voornaamste reden dat het nog steeds in Python zit is als restant van het verleden en het belang dat de voorstanders van het gereserveerde woord eraan hechten.

Af en toe hebben anonieme functies nut, omdat ze een programma iets beter leesbaar kunnen maken. Ik toon hiervan een voorbeeld in hoofdstuk 12.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Het nut van functies
- Functies creëren
- Parameters en argumenten
- Waardes uit functies retourneren middels **`return`**
- Functie benamingen
- Commentaar in functies
- Variabele scope en levensduur
- Lokale en globale variabelen
- Het gebruik van functies om grip te krijgen op complexiteit
- Modules
- Gebruik van een `main()` functie
- Anonieme functies

Opgaves

In deze opgaves schrijf je functies. Je moet natuurlijk niet alleen de functies schrijven, maar ook code om de functies te testen. Om te oefenen, moet je ook je functies becommentariëren als hierboven uitgelegd.

Opgave 8.1 Maak een functie die als parameter een getal krijgt, en die dan de tafel van vermenigvuldiging voor 1 tot en met 10 van dat getal afdrukt. Bijvoorbeeld, als de parameter 12 is, dan drukt het programma als eerste regel "1 * 12 = 12" af, en als laatste regel "10 * 12 = 120."

Opgave 8.2 Schrijf een functie die twee strings krijgt als parameters. De functie retourneert het aantal tekens dat de strings gemeen hebben. Ieder teken wordt slechts eenmalig geteld, bijvoorbeeld, de strings `"een"` en `"peer"` hebben slechts één teken gemeen (de letter "e"). Je mag hoofdletters zien als verschillend van kleine letters. Merk op: de

functie *retourneert* het aantal tekens dat de strings gemeen hebben, en moet het niet in de functie printen. Om de functie te testen kun je wel de retourwaarde printen in het hoofdprogramma.

Opgave 8.3 De Grerory-Leibnitz reeks benadert de waarde van π door de berekening van $4 * (1/1 - 1/3 + 1/5 - 1/7 + 1/9...)$. Schrijf een functie die π benadert via deze reeks. De functie krijgt één parameter, namelijk een integer die aangeeft hoeveel van de termen tussen de haakjes in de reeks berekend moeten worden.

Opgave 8.4 In hoofdstuk 6 werd je gevraagd de wortelformule te implementeren om kwadratische vergelijkingen op te lossen. Een kwadratische vergelijking wordt beschreven door drie numerieke waardes, gewoonlijk A, B, en C genoemd. De vergelijking heeft nul, één, of twee oplossingen, afhankelijk van de discriminant (het deel van de vergelijking onder de wortel). Schrijf een functie die een kwadratische vergelijking kan oplossen. Als parameters krijgt het A, B, en C. Het retourneert drie waardes. De eerste is een integer die het aantal oplossingen aangeeft. De tweede is de eerste oplossing. De derde is de tweede oplossing. Als een oplossing niet bestaat, kun je een nul retourneren voor de corresponderende retourwaarde.

Opgave 8.5 In hoofdstuk 7 legde ik de loop-en-een-half uit. De uiteindelijke code voor het voorbeeld dat ik gebruikte had nog steeds iets lelijks, namelijk dat als `x` kleiner dan 0 of groter dan 1000 was, dat de code nog steeds vroeg om `y` terwijl het al bekend was dat het een andere waarde voor `x` zou moeten krijgen. Ik gaf ook aan dat he dat het gemakkelijkste kon oplossen via functies. Creëer een functie die je aan onderstaande code toevoegt en in onderstaande code aanroept, zodat het probleem wordt opgelost. Verwijder ook de `exit()` door introductie van een `main()` functie. Hint: Maak een variant van `getInteger()` die garandeert dat de integer tussen 0 en 1000 ligt.

exercise0805.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( "De nummers moeten tussen 0 en 1000 liggen" )
        continue
    if x%y == 0 or y%x == 0:
        print( "Fout: de nummers mogen geen delers zijn" )
        exit()
    print( "Vermenigvuldiging van", x, "met", y, "geeft", x * y )

print( "Tot ziens!" )
```

Opgave 8.6 In de statistiek wordt de binomiaalcoëfficiënt "`n` boven `k`," waarbij `n` groter dan of gelijk aan `k` moet zijn, berekend als $n!/(k! * (n-k)!)$, waarbij $n!$ de faculteit van `n` is. Zoals ik in hoofdstuk 7 heb uitgelegd: de faculteit van een positief geheel getal is 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, 5 faculteit is $5! = 5 * 4 * 3 * 2 * 1 = 120$. Als je alle opgaves tot nu toe gemaakt hebt, heb je hier code voor geschreven. Schrijf een functie die de binomiaalcoëfficiënt voor de twee parameters die de functie krijgt, en die de waarde retourneert. Schrijf de code op zo'n manier dat hij als module in een ander programma kan worden opgenomen (dus test de functie via een `main()` functie zoals hierboven is uitgelegd).

Opgave 8.7 Wat is er fout aan de volgende code? Los het probleem op!

exercise0807.py

```
# Wat is er fout?
def oppervlakte_van_driehoek( basis, hoogte ):
    opp = 0.5 * basis * hoogte
    print( "Een driehoek met", basis, "en hoogte",
        hoogte, "heeft oppervlakte", opp )

print( oppervlakte_van_driehoek( 4.5, 1.0 ) )
```

Dit soort code wordt vaak geschreven door studenten die de details van functies nog niet goed begrijpen.

Hoofdstuk 9

Recursie

Recursie is een speciale techniek die je kunt gebruiken nu je geleerd hebt om functies te maken. Recursie kan bepaalde problemen op een elegante en krachtige manier oplossen, maar studenten vinden het vaak een lastig onderwerp. Daarom heb ik er een apart hoofdstuk aan gewijd. Als je bij bestudering van dit hoofdstuk denkt dat het te ingewikkeld voor je is, voel je dan vrij om het voorlopig over te slaan. Je kunt naderhand bij dit hoofdstuk terugkomen. De volgende hoofdstukken zijn weer een stuk gemakkelijker.

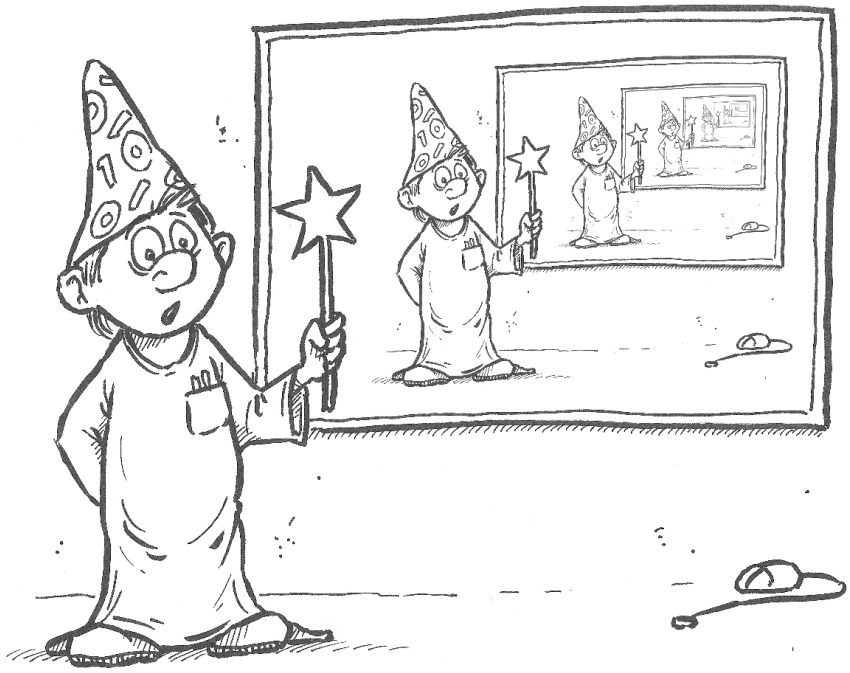
9.1 Wat is recursie?

Recursie is een techniek waarbij een functie zichzelf aanroept. Iets algemener gesteld, refereert het aan een situatie waarin een functie andere functies aanroept op zo'n manier dat de uitvoering van de eerste functie nog steeds bezig is wanneer deze functie zelf nogmaals wordt aangeroepen (bijvoorbeeld, functie `a()` roept functie `b()` aan, die dan functie `a()` weer aanroept).

Dit klinkt misschien vreemd als je het voor het eerst aantreft, maar er is niks op tegen dat een functie andere functies aanroept, en een functie mag iedere functie aanroepen die gedefinieerd is op het moment dat de functie wordt aangeroepen. Omdat een functie gedefinieerd moet zijn voordat hij wordt aangeroepen, kan hij dus zichzelf aanroepen.

"Maar," hoor ik iemand al protesteren: "als een functie zichzelf aanroept, dan roept hij zichzelf nogmaals aan, en nogmaals, en nogmaals... Betekent dat dan niet dat je een eindeloos proces krijgt, net als bij een eindeloze loop?" Het antwoord is dat er inderdaad het gevaar bestaat dat een recursieve functie, die slordig in elkaar is gestoken, een eindeloze serie aanroepen veroorzaakt. Maar recursieve functies moeten ontworpen worden op een manier dat dat niet gebeurt.

Er zijn veel problemen waarvoor recursie een elegante oplossing biedt. Daarom is het belangrijk dat je je bewust bent van de mogelijkheden van recursie, en dat je weet hoe je het kunt toepassen... en dat je weet wat de beperkingen zijn.



9.2 Recursieve definities

Een voorbeeld van een recursieve definitie is de definitie van de faculteit, die ik al heb geïntroduceerd in twee eerdere hoofdstukken. In die hoofdstukken gaf ik de volgende definitie van de faculteit: de faculteit van een positief geheel getal is dat getal, vermenigvuldigd met alle positieve gehele getallen die kleiner zijn (exclusief nul).

Wiskundigen prefereren een recursieve definitie: De faculteit $n!$ van een positief getal n wordt als volgt berekend: $1! = 1$, en $n! = n * (n-1)!$ voor $n > 1$.

Deze definitie is recursief omdat het refereert aan de faculteit van $n-1$ om de faculteit van n te definiëren. Dit leidt niet tot eindeloze recursie, omdat op enig moment n gelijk zal zijn aan 1, en de faculteit van 1 is apart gedefinieerd.

Je kunt de faculteit als een recursieve functie als volgt implementeren:

listing0901.py

```
def faculteit( n ):
    if n <= 1:
        return 1
    return n * faculteit( n-1 )

print( faculteit( 5 ) )
```

Zie hoe deze functie exact de recursieve definitie van de faculteit volgt: als `n` gelijk is aan 1, retourneert de functie 1, en anders retourneert het `n` keer de faculteit van `n-1`. (Merk op dat ik `n <= 1` schreef in plaats van `n == 1` om te voorkomen dat er problemen ontstaan als de gebruiker de functie aanroept met, bijvoorbeeld, een negatieve `n`.)

Voor het geval je het problematisch vindt om te begrijpen wat deze functie doet, beschrijf ik hieronder de aanroepen die de functie doet als hij wordt aangeroepen met 5 als argument. Ik laat aanroepen inspringen als een "hoger-niveau aanroep" nog steeds actief is als de aanroep wordt gemaakt. Een "return" die één inspringing dieper gaat dan een "aanroep," wordt gegeven in die aanroep, en retourneert de gegeven waarde.

```
aanroep faculteit( 5 )
    aanroep faculteit( 4 )
        aanroep faculteit( 3 )
            aanroep faculteit( 2 )
                aanroep faculteit( 1 )
                    return 1
                return 2 * 1
            return 3 * 2
        return 4 * 6
    return 5 * 24
print( 120 )
```

9.2.1 Wanneer gebruik je recursie

Als je doorhebt hoe de recursieve implementatie van de faculteit werkt, ziet het er wellicht aantrekkelijk uit. Het is eenvoudig, elegant, en is eigenlijk best wel "cool." Echter, de iteratieve implementatie van de faculteit is zeer te prefereren boven de recursieve.

De reden blijkt uit de beschrijving van de aanroepen hierboven. Je ziet dat voordat de aanroep `faculteit( 1 )` wordt gemaakt, er al vier aanroepen van `faculteit()` in het geheugen van de computer staan. Als je de faculteit van 100 zou willen berekenen, komen er niet minder dan 100 aanroepen van faculteit in het geheugen te staan alvorens er waardes geretourneerd gaan worden. Dit is geen goed idee, en Python zou gebrek aan (stack) geheugen kunnen krijgen, of heel, heel erg traag worden.

Daartegenover staat dat een iteratieve implementatie van de faculteit slechts twee variabelen in het geheugen hoeft te houden. Dat is snel en geeft geen gevaar dat de computer vastloopt. Je moet alleen recursieve implementaties bouwen als:

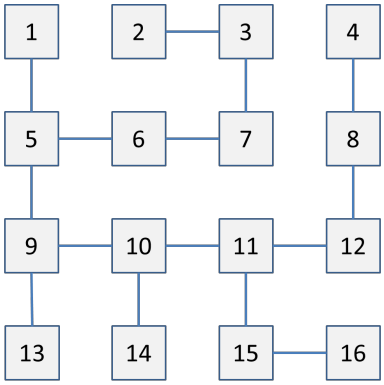
- recursie de meest natuurlijke manier is om de oplossing te implementeren; en
- het recursieve proces gegarandeerd niet te diep gaat.

Iedere recursieve functie kan ook als een iteratief proces gebouwd worden. Zo nu en dan kom je echter een probleem tegen waarvoor de recursieve oplossing veel eleganter, leesbaarder, en onderhoudbaarder is dan de iteratieve variant. In dat geval moet je overwegen een recursieve oplossing te implementeren.

9.2.2 Doolhof doorzoeken

Op dit punt in het boek is het lastig een goede demonstratie van recursie te geven, aangezien het specifieke data structuren[11] nodig heeft om de kracht te tonen. Om toch

[11] Een data structuur is een manier om een bepaalde soort data in het geheugen van de computer op te slaan. Een integer, bijvoorbeeld, is een simpele data structuur die precies één geheel getal in het geheugen vasthoudt. Er bestaan complexere data structuren, die bijvoorbeeld een rij getallen vasthouden. Deze komen in latere hoofdstukken aan bod.



Afb. 9.1: Het doolhof van `pcmaze`.

iets niet-triviaals te tonen, heb ik een module genaamd `pcmaze` gecreëerd. Je vindt deze module in appendix D, en je moet hem ofwel zelf maken, ofwel downloaden van dezelfde site waar je `pcinput.py` hebt gevonden, om de code van deze paragraaf te kunnen uitvoeren.

`pcmaze` implementeert een eenvoudig doolhof, dat een aantal genummerde cellen met elkaar verbindt. De ingang van het doolhof kun je vinden door aanroep van de functie `entrance()`.[12] De uitgang van het doolhof wordt gegeven door de functie `exit()` (niet te verwarren met de `exit()` functie uit de `sys` module). De module heeft ook een functie `connected()` die twee numerieke argumenten krijgt: deze retourneert **`True`** als er een directe verbinding bestaat tussen de cellen met die nummers, en anders **`False`**. De ingang is gegarandeerd de laagst-genummerde cel, en de uitgang is gegarandeerd de hoogst-genummerde cel.

Het doel is code te schrijven die een pad uitzet van de ingang naar de uitgang (als een dergelijk pad bestaat). Het doolhof is gevisualiseerd in afbeelding 9.1. De ingang is 1, de uitgang is 16.

Hoe vind je nu je weg door zo'n doolhof (zonder dat je de layout kent)? Recursief kan dat als volgt: Je definieert een functie `leidt_naar_uitgang()` die een pad naar de uitgang retourneert als de cel die als parameter wordt meegegeven op het pad ligt dat naar de uitgang leidt. Als die functie een pad retourneert, dan weet je dat de huidige cel op dat pad ligt. Als je hem dus aanroept met cel 1, krijg je een pad dat leidt van de ingang naar de uitgang (als dat pad er is).

Maar hoe weet de functie of een cel ligt op het pad dat leidt naar de uitgang? Als de huidige cel de uitgang zelf is, dan weet je dat, jazeker, deze cel op dat pad ligt. Zo niet, dan ligt de cel op het pad dat leidt naar de uitgang als het een verbinding heeft met een cel die ligt op het pad dat leidt naar de uitgang. Dit is een recursieve definitie.

Je moet voorzichtig zijn met zo'n recursieve definitie dat je niet vastloopt op een circulair pad in het doolhof. Dat betekent dat als de functie "beweegt" van cel A naar cel B, de functie niet meer terug mag komen bij A. Als dat gegarandeerd is, moet de functie werken. De eenvoudige implementatie die ik hier ga geven zou niet werken als er circulaire paden

[12] Ik heb Engelstalige functiebenamingen gekozen om ervoor te zorgen dat de code compatibel is met de Engelstalige versie van dit boek.

in het doolhof zouden zijn, maar die zijn er gelukkig niet. Het probleem is niet onoplosbaar als ze er wel zijn, maar om dat netjes op te lossen heb je een data structuur nodig die pas in een toekomstig hoofdstuk aan bod komt.

In pseudo-code ziet de recursieve functie `leidt_naar_uitgang()` er ongeveer als volgt uit:

```
functie leidt_naar_uitgang( huidigecel ):
    if (huidigecel is de uitgang):
        return (pad dat alleen de uitgang bevat)
    for (iedere verbondencel die nog niet onderzocht is):
        pad = leidt_naar_uitgang( verbondencel )
        if (pad is niet leeg):
            voeg huidigecel toe aan pad
            return pad
    return (leeg pad)
```

Ik geef nu een implementatie van deze recursieve oplossing. In de implementatie meteen hieronder geef ik niet het pad terug, maar ik retourneer gewoon **`True`** of **`False`** om aan te geven of het pad gevonden is, en ik print het pad in de functie zelf (ik zal later in dit hoofdstuk een complete implementatie van de pseudo-code geven).

listing0902.py

```
from pcmaze import entrance, exit, connected

def leidt_naar_uitgang( komtvan, cel ):
    if cel == exit():
        return True
    for i in range( entrance(), exit()+1 ):
        if i == komtvan:
            continue
        if not connected( cel, i ):
            continue
        if leidt_naar_uitgang( cel, i ):
            print( cel, "->", i )
            return True
    return False

if leidt_naar_uitgang( 0, entrance() ):
    print( "Pad gevonden!" )
else:
    print( "Pad niet gevonden" )
```

Ik bespreek deze recursieve functie nu in detail.

De functie krijgt twee parameters. De eerste is de cel waar het pad vandaan komt. De tweede is de cel die gecontroleerd wordt om te zien of hij naar de uitgang leidt. De eerste parameter is alleen nodig omdat je niet mag terugkeren op het pad.

De functie controleert eerst of de uitgang bereikt is. Zo ja, dan retourneert de functie **`True`**.

Als de uitgang niet bereikt is, controleert de functie alle cellen van het doolhof om te zien of ze verbonden zijn met de huidige cel.

De cel waarvandaan de aanroep gekomen is wordt uitgesloten. Ook worden alle cellen uitgesloten waarmee geen verbinding is. Maar alle overige cellen worden gecontroleerd. De cel zelf hoeft niet uitgesloten te worden, aangezien in de definitie van het doolhof een cel nooit verbonden is met zichzelf.

Als een cel blijkt naar de uitgang te leiden (doordat de recursieve aanroep **`True`** geeft), dan drukt de functie af dat de beweging "huidige cel naar gecontroleerde cel" deel is van het pad dat naar de uitgang leidt. Vervolgens retourneert de functie **`True`**.

Anders, nadat alle verbonden cellen zijn gecontroleerd en nog steeds geen pad is gevonden, retourneert de functie **`False`**.

Deze aanpak zal het volledige pad van ingang naar uitgang afdrukken, in omgekeerde volgorde.

Om duidelijk te maken wat er precies gebeurt in de functie, heb ik hem wat uitgebreid. De implementatie hieronder drukt iedere verbinding die gecontroleerd wordt af. Ik heb ook een parameter `diepte` toegevoegd, die bijhoudt hoe diep de recursie gaat. Ik vertaal die diepte in inspringingen.

listing0903.py

```
from pcmaze import entrance, exit, connected

def leidt_naar_uitgang( komtvan, cel, diepte ):
    inspringing = diepte * 4 * " "
    if cel == exit():
        return True
    for i in range( entrance(), exit()+1 ):
        if i == komtvan:
            continue
        if not connected( cel, i ):
            continue
        print( inspringing + "Controleer", cel, "->", i )
        if leidt_naar_uitgang( cel, i, diepte + 1 ):
            print( inspringing + "Pad gevonden:", cel, "->", i )
            return True
    return False

if leidt_naar_uitgang( 0, entrance(), 0 ):
    print( "Pad gevonden!" )
else:
    print( "Pad niet gevonden" )
```

9.2.3 Retourwaardes van recursieve functies

Net als reguliere functies, kunnen recursieve functie informatie communiceren aan de rest van het programma middels retourwaardes.

Een van de minder-fraaie zaken van de doolhof-oplossende functie hierboven is dat het pad geprint wordt in plaats van geretourneerd – wat er ook toe leidt dat het pad in omgekeerde volgorde wordt afgedrukt. Het zou beter zijn als de functie aanroepen hun deel

van het pad aan een hoger liggende aanroep zouden retourneren, zodat uiteindelijk het pad als geheel zou worden geretourneerd naar de eerste aanroep. Dit is wat de pseudo-code beoogde. Een goede manier om een pad te retourneren is in de vorm van een list, maar dat is het onderwerp van hoofdstuk 12. In plaats daarvan doe ik het als een string.

Het werkt als volgt: een aanroep die de uitgang vindt, retourneert het nummer van de uitgang als string. Een aanroep die een deel van het pad geretourneerd krijgt, retourneert dat pad zelf ook weer, maar voegt de huidige cel toe aan het pad. Een aanroep die een lege string terugkrijgt, retourneert zelf ook een lege string.

Dit betekent dat in de code hierboven, iedere **`return True`** vervangen moet worden door een commando dat een string retourneert dat een (deel van het) pad retourneert, en iedere **`return False`** vervangen moet worden door het retourneren van een lege string. De code wordt dan:

listing0904.py

```
from pcmaze import entrance, exit, connected

def leidt_naar_uitgang( komtvan, cel ):
    if cel == exit():
        return "{}".format( exit() )
    for i in range( entrance(), exit()+1 ):
        if i == komtvan:
            continue
        if not connected( cel, i ):
            continue
        check = leidt_naar_uitgang( cel, i )
        if check != "":
            return "{} -> {}".format( cel, check )
    return ""

check = leidt_naar_uitgang( 0, entrance() )
if check != "":
    print( "Pad gevonden!", check )
else:
    print( "Pad niet gevonden" )
```

Als je recursie wilt begrijpen, moet je deze code goed bestuderen. De code is een typische weergave van het gebruik van retourwaardes in recursieve functies. Studenten wiens begrip van recursie wankel is en die een opdracht krijgen waar informatie van een dieper niveau van recursie naar een hoger niveau van recursie gecommuniceerd moet worden, grijpen vaak naar een globale variabele om dit te doen. Je ziet dat dat niet nodig is.

Feitelijk is er geen echt verschil tussen een recursieve functie aanroep en een reguliere functie aanroep, behalve dat je bij recursieve aanroepen ervoor moet zorgen dat ze wel op een bepaald moment eindigen. Recursie ziet er alleen vreemd uit de eerste keer dat je het tegenkomt.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Recursieve functies
- Wanneer je recursieve functies gebruikt (en wanneer niet)
- Het doel van recursieve functies
- Retourwaardes van recursieve functies

Opgaves

Opgave 9.1 Een recursieve definitie van het *n*de Fibonacci getal `fib(n)` zegt dat `fib(n)` gelijk is aan `fib(n-1) + fib(n-2)`. `fib(1)` en `fib(2)` zijn beide 1. Schrijf een recursieve functie die je kunt aanroepen met een integer argument `n` die het *n*de Fibonacci getal retourneert.

Opgave 9.2 Om meer inzicht te krijgen in hoe recursie werkt: pas de Fibonacci functie aan door er een diepte-parameter aan toe te voegen, die start met nul en die met 1 verhoogd wordt iedere keer dat een diepere versie van de functie wordt aangeroepen. Bij binnenkomst van de functie druk je het nummer af waarmee de functie wordt aangeroepen, en als er een waarde wordt geretourneerd, druk je ook die waarde af. Gebruik de diepte parameter om de afgedrukte waardes in te springen. Bestudeer de output.

Opgave 9.3 Denk je dat het een goed idee is om de Fibonacci reeks recursief te implementeren? Waarom, of waarom niet?

Opgave 9.4 De grootste gemene deler is het grootste gehele getal dat twee andere getallen kan delen zonder dat er een rest overblijft. Bijvoorbeeld, de grootste gemene deler van 14 en 21 is 7, omdat 7 het grootste getal is waardoor je 14 en 21 beide kunt delen. Euclides' algoritme om de grootste gemene deler te bepalen zegt dat als je de grootste kunt delen door de kleinste, dan is het de kleinste. Anders is het de uitkomst van het bepalen van de grootste gemene deler van de kleinste en de rest die overblijft als je de grootste deelt door de kleinste. Dit is een recursieve definitie. Implementeer Euclides' algoritme in een recursieve functie. Hint: testen of twee getallen door elkaar gedeeld kunnen worden, en het berekenen van een rest, kunnen beide gedaan worden met de modulo operator. Deze code kan *heel* erg kort zijn.

Opgave 9.5 In de code hieronder heb ik een recursieve implementatie gedaan van het vragen aan de gebruiker om een string, waarin alleen kleine letters mogen worden gebruikt. Als de gebruiker iets ingeeft waar een incorrect teken in zit, zorgt een recursieve aanroep naar de functie ervoor dat een nieuwe input gevraagd wordt. Dit lijkt erop dat de loop-en-een-half vermeden wordt. Hoewel het altijd een slechte zaak is om de diepte van een recursieve aanroep afhankelijk te maken van de gebruiker, is deze implementatie niet alleen slecht, maar zelfs behoorlijk fout. Zie je wat er fout aan is, en hoe dat veroorzaakt

wordt? Hint: Het is niet de expressie `letter < 'a' or letter > 'z'`; die vergelijkingen zijn in orde.

Ik wil met klem benadrukken dat het idee hieronder slecht is. Je moet geen recursie gebruiken om doorsnee problemen af te handelen die net zo goed met iteraties gedaan kunnen worden. Recursie is bedoeld voor uitzonderlijke situaties. Zie dit niet als een voorbeeld van recursie, maar zie het als een voorbeeld van hoe recursie *niet* gebruikt moet worden. De voornaamste reden dat ik de code hier opneem is dat ik soms zie dat studenten dit soort code schrijven, en ik wil expliciet maken dat dat een slecht idee is.

exercise0905.py

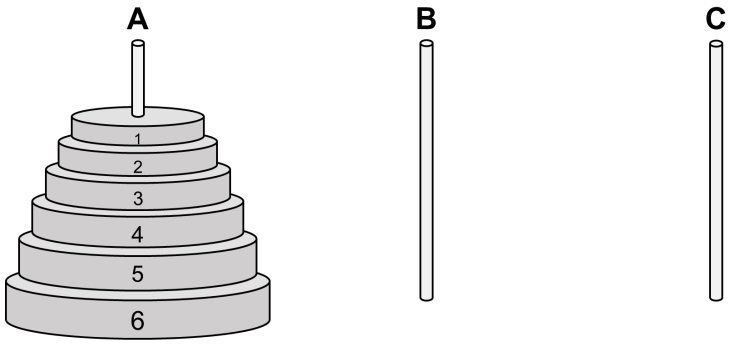
```
def vraag_input( prompt ):
    waarde = input( prompt )
    for letter in waarde:
        if letter < 'a' or letter > 'z':
            print( letter, "is niet toegestaan!")
            waarde = vraag_input( prompt ) # DOE DIT NIET!
    return waarde

s = vraag_input( "Geef een string van kleine letters: " )
print( "Je gaf in:", s )
```

Opgave 9.6 De Torens van Hanoi is een puzzel die gebruik maakt van drie palen, die A, B, en C genoemd worden. Paal A bevat een stapel schijven van verschillende grootte; de schijven zijn genummerd volgens hun grootte. De kleinste schijf is 1, de volgende 2, de volgende 3, etcetera. De grootste schijf is N. Typische waardes voor N zijn 4 en 5, hoewel in de klassieke puzzel N de waarde 64 schijnt te hebben. De schijven zijn op paal A gestapeld volgens hun grootte, met de grootste schijf onderaan en de kleinste bovenaan. Je moet nu alle schijven verplaatsen van paal A naar paal C, waarbij je vier regels in acht moet nemen: (1) je mag mag maar één schijf per keer verplaatsen; (2) je mag alleen maar schijven verplaatsen tussen de palen; (3) je mag alleen een schijf verplaatsen die bovenop een stapel ligt, en je kunt hem alleen verplaatsen naar de top van een andere stapel; en (4) je mag nooit een schijf plaatsen op een schijf die kleiner is. Schrijf een programma dat deze puzzel oplost voor een willekeurige waarde van N (om te testen moet je N niet groter dan 10 kiezen). Laat je programma de oplossing printen als een recept, met regels als "Schijf 1 van A naar C." Druk aan het einde van het recept het aantal benodigde stappen af.

Voor een recursieve oplossing, bedenk het volgende: Stel dat je de puzzel moet oplossen waarbij de grootste schijf 10 is. Dit is niet moeilijk als je hem op kunt lossen voor grootte 9. Je gebruikt dan de procedure voor grootte 9 om de bovenste 9 schijven te verplaatsen van paal A naar paal B, je verplaatst vervolgens schijf 10 van paal A naar paal C, en tenslotte gebruik je de procedure voor grootte 9 om de overgebleven 9 schijven te verplaatsen van paal B naar paal C. Maar hoe los je de puzzel op met de grootste schijf 9? Dat is niet moeilijk als je hem kunt oplossen voor grootte 8... Je kunt je voorstellen waar dit heengaat. Je kunt de complexiteit van het probleem stap-voor-stap reduceren, totdat je stelt "het is gemakkelijk om het probleem op te lossen voor grootte 2 als je het kunt oplossen voor grootte 1." Oplossen voor grootte 1 is triviaal: je verplaatst gewoon de schijf naar de paal waar hij heen moet. Dit is een recursieve definitie van de oplossing:

Om de puzzle op te lossen voor grootte N waarbij je de stapel moet verplaatsen van paal X

naar paal Y met paal Z als tijdelijke paal, dan los je het eerst op voor grootte $N - 1$ waarbij je de stapel verplaatst van paal X naar paal Z met paal Y als tijdelijke paal, dan verplaats je de schijf met grootte N van paal X naar paal Y, en tenslotte los je het probleem op voor grootte $N - 1$ waarbij je van paal Z naar paal Y gaat met paal X als tijdelijke paal.



Hoofdstuk 10

Strings

Tot nu toe gebruikten de meeste voorbeelden en opgaves getallen. Je hebt je misschien afgevraagd of programmeren vooral bedoeld is voor het manipuleren van getallen. In het dagelijks leven is het veel gebruikelijker om met tekstuele informatie om te gaan.

De reden dat de omgang met teksten was uitgesteld tot nu, is dat in programmeertalen het veel gemakkelijker is om met getallen om te gaan dan met teksten. Maar in dit hoofdstuk leg ik uit hoe je teksten kunt manipuleren met programma's.

In programmeeromgevingen worden teksten weergegeven door strings. Dit hoofdstuk geeft details over strings, en over functies die beschikbaar zijn om strings aan te pakken.

10.1 Herhaling

In hoofdstuk 3 heb ik strings kort geïntroduceerd. Ik heb uitgelegd dat een string een tekst is, die omsloten is door enkele of dubbele aanhalingstekens, en dat een string iedere lengte mag hebben, inclusief nul tekens lang. Het hoofdstuk legde ook uit dat je twee strings aan elkaar kunt plakken met behulp van de $+$, en dat je een string zichzelf kunt laten herhalen door middel van de $*$. Bijvoorbeeld:

```
s1 = "appel"
s2 = 'banaan'
print( s1 )
print( s2 )
print( s1 + s2 )
print( 3 * s1 )
print( s2 * 3 )
print( 2 * s1 + 2 * s2 )
```

Hoofdstuk 5 introduceerde de **format**() functie die strings op allerlei manieren kan formatteren. Ik gaf ook aan dat je de lengte van een string kunt bepalen met de **len**() functie.

String vergelijkingen heb ik uitgelegd in hoofdstuk 6; ik noemde specifiek het feit dat bij vergelijkingen tussen strings de alfabetische regels worden aangehouden, waarbij

hoofdletters altijd eerder in het alfabet staan dan kleine letters. Ik zal hier in dit hoofdstuk meer over zeggen. In hoofdstuk 6 gaf ik ook aan dat de **in** operator gebruikt kan worden om te testen of tekens of substrings voorkomen in een string.

Hoofdstuk 7 legde uit hoe je met een **for** loop alle tekens in een string kunt doorlopen.

```
s1 = "mango"
s2 = "banaan"
for letter in s1:
    if letter in s2:
        print( s1, "en", s2, "bevatten beide de letter", letter )
```

10.2 Strings over meerdere regels

In Python kunnen strings meerdere regels beslaan. Dat kan nuttig zijn wanneer je een erg lange string in je programma hebt, of als je de output van de string op een specifieke manier wilt formatteren. Dit kan op twee manieren bereikt worden:

- Met enkele of dubbele aanhalingstekens, en een indicatie dat de string doorloopt op de volgende regel door een backslash aan het einde van de regel te zetten.
- Met drievoudige enkele of dubbele aanhalingstekens.

Ik demonstreer eerst hoe het werkt met enkele of dubbele aanhalingstekens om de string:

listing1001.py

```
langestring = "Ik heb mijn buik vol van te worden behandeld \
als een schaap. Wat is de zin van op vakantie gaan als je \
gewoon maar een toerist bent die wordt rondgereden in een bus \
omringd door zweterige uilskuikens uit Den Haag en Rotterdam \
met hun honkbalpetjes en trainingspakken en radio's en \
De Telegraaf, klagend over de koffie - 'Oh, ze zetten geen \
lekker bakje hier, toch, niet zoals thuis' - en dan stoppen \
bij Mallorcaanse eettentjes waar ze friet en kroket verkopen \
en Heineken en calamaris met salade en ze zitten in hun \
katoenen overgooiers Nivea zonnebrand te spuiten over hun \
pafferige rauwe opgezwollen uitpuilende blubberbuiken 'omdat \
ze op de eerste dag wat teveel zon hebben gehad.'"
print( langestring )
```

Als je deze code uitvoert, zie je dat Python de string als een geheel interpreteert. De backslash (\) die aangeeft dat de string verder gaat op de volgende regel is algemener bruikbaar dan alleen voor strings: je kunt hem achter ieder Python statement zetten om aan te geven dat het statement verder gaat op de volgende regel. Dat kan nuttig zijn bij bijvoorbeeld lange berekeningen.

De aanbevolen manier om strings over meerdere regels te spreiden in Python is echter het gebruik van drievoudige aanhalingstekens. Ik heb in een eerder hoofdstuk aangegeven dat je die gebruikt om lang commentaar in de code op te nemen, maar feitelijk komt het erop

neer dat je dan een lange string midden in je programma zet. Zo'n string doet niks, tenzij je hem aan een variabele toekent. Hier is een voorbeeld van een string met drievoudige dubbele aanhalingstekens:

listing1002.py

```
langestring = """En je wordt onderworpen aan een eindeloze stroom
Hotel Miramars en Bellevues en Continentaals met hun moderne luxe
internationale studio's en Heineken van de tap en zwembaden vol
vette Duitse zakenlui die denken dat ze acrobaten zijn en
pyramides vormen en de kinderen bang maken en voordringen in de
rij en als je niet aan tafel zit om klokslag zeven dan mis je je
kop Knorr Romige Tomatensoep, het eerste item op het menu van de
Internationale Cuisine, en iedere donderdagavond is er een
amateuristisch theater in de bar, met een kleine verwijfde
Spanjool met smalle heupjes en een opblazen taart met haar haar
platgeboterd en een enorm achterwerk die Flamenco voor
Buitenlanders presenteren."""
print( langestring )
```

Het opvallende verschil tussen deze twee voorbeelden is dat in het eerste voorbeeld de string beschouwd werd als een lange, doorlopende serie tekens, terwijl in het tweede voorbeeld de verschillende regels op meerdere regels geprint wordt. De reden dat dat gebeurt in het tweede voorbeeld is dat er een onzichtbaar teken staat aan het einde van iedere regel, dat aangeeft dat Python naar een nieuwe regel moet gaan voordat verder geprint wordt. Dit is een zogeheten "newline" teken, en je kunt het expliciet in een string opnemen, door gebruik te maken van de code `"\n"`. Deze code moet je niet lezen als twee tekens, de backslash en de `"n"`, maar als een enkel "newline" teken. Je kunt met dit teken ervoor zorgen dat de output over meerdere regels geprint wordt. Dat kan zelfs als je de backslash aan het einde van een regel zet om aan te geven dat de string over meerdere regels verspreid is, als in het eerste voorbeeld. Bijvoorbeeld:

listing1003.py

```
langestring = "En een paar nasale secretaresses uit Middelburg\n\
met flubberende witte benen en buikloop die flirten met harige\n\
krombenige Spaanse obers genaamd Manuel en eens per week is er\n\
een excursie naar de plaatselijke Romeinse bouwval waar je\n\
cola kunt kopen en gesmolten ijsjes en natuurlijk Heineken en\n\
op een avond bezoek je het zogenaamde typische restaurant met\n\
rustieke uitstraling en plaatselijke atmosfeer en je zit naast\n\
een groepje toeristen uit Amstelveen die maar blijven zingen\n\
'Torremolinos, torremolinos' en klagen over het eten - 'Het is\n\
erg vettig hier, vind je niet?' - en je wordt in een hoek\n\
gedreven door een dronken groentenboer uit Hilversum met zijn\n\
instant fototoestel en plastic sandalen en het Algemeen\n\
Dagblad van afgelopen dinsdag en hij zaagt maar door en door\n\
en door over hoe meneer Jansen dit land moet besturen en\n\
hoeveel talen Frits Bolkestein wel niet spreekt en dan kotst\n\
hij zijn avondeten uit over de cocktails."
print( langestring )
```

Dit betekent dat als je niet die automatische "newlines" wilt hebben in een string die meerdere regels beslaat, je de eerste aanpak moet gebruiken, met de backslash aan het einde van iedere regel. Als je wel meerdere regels wilt, dan is de tweede aanpak waarschijnlijk het beste leesbaar.

10.3 Speciale tekens

`"\n"` is een "speciaal teken" (in het Engels heet dit een "escape sequence"). Speciale tekens in Python worden over het algemeen geschreven als een backslash gevolgd door een code. De code kan één of meerdere tekens beslaan. Python interpreteert zulke speciale tekens, als ze in een string staan, niet letterlijk.

Naast het "newline" teken `"\n"`, heb ik in hoofdstuk 3 ook de speciale tekens `"\'"` en `"\""` geïntroduceerd, die je kunt gebruiken om een enkel respectievelijk dubbel aanhalingsteken in een string op te nemen, ongeacht het type aanhalingstekens dat je om de string heen hebt gezet. Ik heb ook genoemd dat je `"\\"` kunt gebruiken om een "echte" backslash in de string op te nemen.

Naast deze zijn er nog diverse andere speciale tekens. De meeste zijn behoorlijk archaïsch en worden niet meer gebruikt op moderne computers, dus die kun je negeren. De twee die ik nog wil noemen zijn `"\t"` die een tabulatie (inspringing) in de string representeert, en `"\x`*nn*`"` waarbij *nn* staat voor twee hexadecimale cijfers, die het hexadecimale getal *nn* representeren. Bijvoorbeeld, `"\x20"` is het teken dat gerepresenteerd wordt door het hexadecimale getal 20, dat hetzelfde is als het decimale getal 32, wat een spatie is (dit leg ik later in dit hoofdstuk verder uit).

Voor het geval je nooit hebt geleerd hoe je moet tellen met hexadecimale getallen: Hexadecimale getallen gebruiken 16 verschillende cijfers, namelijk 0 tot en met 9 en A tot en met F. Een directe vertaling van hexadecimale cijfers naar decimale getallen stelt dat A gelijk is aan 10, B aan 11, etcetera. In decimale getallen wordt de waarde van een getal dat uit meerdere cijfers bestaat berekend door de cijfers te vermenigvuldigen met oplopende machten van 10, van rechts naar links; bijvoorbeeld, het getal 1426 is $6 + 2 * 10 + 4 * 100 + 1 * 1000$. Voor hexadecimale getallen doe je hetzelfde, maar vermenigvuldig je de cijfers met oplopende machten van 16; bijvoorbeeld, het hexadecimale getal $4AF2$ is $2 + 15 * 16 + 10 * 256 + 4 * 4096$. Programmeurs gebruiken graag hexadecimale getallen, omdat computers als kleinste rekeneenheid de "byte" gebruiken, en een byte kan 256 verschillende waardes bevatten; met andere woorden, een byte kan iedere waarde bevatten die je kunt uitdrukken met precies twee hexadecimale cijfers.

Waarom het nuttig kan zijn te weten hoe je hexadecimaal moet tellen en waarom je tekens in een string hexadecimaal zou willen representeren volgt later in dit boek.

10.4 Tekens in een string

Ik heb al meerdere keren laten zien dat een string een verzameling is van tekens in een specifieke volgorde. Je kunt de individuele tekens van een string middels indices benaderen.

10.4.1 String indices

Ieder teken in een string heeft een positie, en die positie kun je weergeven door het index nummer van de positie. De indices beginnen bij 0 en lopen op tot aan de lengte van de string. Hieronder zie je het woord "python" op de eerste regel, met op de tweede en derde regel indices voor ieder teken in deze string:

```
 p  y  t  h  o  n
 0  1  2  3  4  5
-6 -5 -4 -3 -2 -1
```

Zoals je kunt zien, kun je positieve indices gebruiken die beginnen met 0 bij de eerste letter van de string, en die oplopen tot het einde van de string. Je kunt ook negatieve indices gebruiken, die starten met -1 bij de laatste letter van de string, en die aflopen totdat de eerste letter van de string bereikt is.

De lengte van een string `s` kun je berekenen met **`len`**`(s)`; de laatste letter van de string heeft dus index **`len`**`(s)-1`. Met negatieve indices heeft de eerste letter van de string de index `-`**`len`**`(s)`.

Als een string is opgeslagen in een variabele, dan kun je de individuele letters van de string benaderen via de variabele naam en de index van de gevraagde letter tussen vierkante haken ([]) rechts ernaast.

```
fruit = "aardbei"
print( fruit[4] )
print( fruit[2] )
print( fruit[1] )
print( fruit[-4] )
print( fruit[-2] )
print( fruit[-5] )
print( fruit[-1] )
print( fruit[5] )
```

Je mag ook variabelen als indices gebruiken, en zelfs berekeningen of functie-aanroepen. Je moet er echter altijd voor zorgen dat berekeningen leiden tot integers, want floats kunnen niet als indices gebruikt worden. Hieronder staan een paar voorbeelden, waarvan de meeste zo ingewikkeld zijn dat ik geen reden zie om ze op deze manier in een programma te zetten. Maar ze laten zien wat de mogelijkheden zijn.

```
from math import sqrt

fruit = "aalbes"
x = 3

print( fruit[3-2] )
print( fruit[int( sqrt( 4 ) )] )
print( fruit[2**2] )
print( fruit[int( (x-len( fruit ))/3 )] )
print( fruit[-len( fruit )])
print( fruit[-x] )
```

In principe mag je een index ook gebruiken bij een string die niet in een variabele staat, bijvoorbeeld, `"aalbes"[3]` is de letter `"b"`. Het mag duidelijk zijn dat niemand dat ooit doet.

Naast enkele indices om letters in een string te benaderen, kun je ook substrings van een string benaderen door twee getallen tussen vierkante haken te zetten met een dubbele punt (:) ertussen. De eerste van deze getallen is de index waar de substring start, de tweede waar de substring eindigt. De substring is exclusief de letter die hoort bij de tweede index. Door het linkergetal weg te laten geef je aan dat de substring begint bij de start van de string (dus bij index 0). Door het rechtergetal weg te laten geef je aan dat de substring eindigt met het laatste teken van de string (inclusief dit laatste teken).

Als je probeert een teken van een string te benaderen met een index die buiten de string valt, krijg je een runtime error ("index out of bounds"). Als je een substring probeert te benaderen geldt die beperking niet; het is toegestaan om getallen te gebruiken die buiten het bereik van de string vallen.

```
fruit = "aalbes"
print( fruit[:] )
print( fruit[0:] )
print( fruit[:6] )
print( fruit[:100] )
print( fruit[:len( fruit )] )
print( fruit[1:-1] )
print( fruit[2], fruit[1:6] )
```

10.4.2 Strings doorlopen

Ik heb eerder uitgelegd hoe je de tekens van een string kunt doorlopen middels een **for** loop:

```
fruit = 'appel'
for teken in fruit:
    print( teken, '- ', end='' )
```

Nu je indices begrijpt, realiseer je je waarschijnlijk wel dat je die ook kunt gebruiken om een string te doorlopen:

listing1004.py

```
fruit = 'appel'

for i in range( 0, len( fruit ) ):
    print( fruit[i], "- ", end="" )
print()

i = 0
while i < len( fruit ):
    print( fruit[i], "- ", end="" )
    i += 1
```

Als je voldoende hebt aan toegang krijgen tot de individuele tekens in de string, is de eerste methode, waarbij de constructie **for** <teken> **in** <string> wordt gebruikt, verreweg het meest elegant en leesbaar. Maar soms moet je een probleem oplossen waarbij een andere methode nodig is.

Opgave Schrijf een programma dat van een string de indices print van alle klinkers (a, e, i, o, en u). Dit kan met een **for** loop of een **while** loop, maar de **while** lijkt iets geschikter.

Opgave Schrijf een programma dat twee strings gebruikt. Voor ieder teken in de eerste string dat in de tweede string precies hetzelfde teken heeft met precies dezelfde index, druk je het teken en de index af. Pas op voor een "index out of bounds" runtime error. Test met de strings `"The Holy Grail"` en `"Life of Brian"`.

Opgave Schrijf een functie die een string als argument krijgt, en die dan een nieuwe string retourneert die hetzelfde is als het argument, maar waarbij ieder teken dat geen letter is vervangen is door een spatie (bijvoorbeeld, de uitdrukking `"ph@t l00t"` wordt gewijzigd in `"ph t l t"`). Om zo'n functie te schrijven begin je met een lege string, en doorloopt de tekens van het argument één voor één. Als je een acceptabel teken tegenkomt, voeg je het toe aan de nieuwe string. Anders voeg je een spatie toe aan de nieuwe string. Je kunt testen of een teken acceptabel is met eenvoudige vergelijkingen, bijvoorbeeld, alle kleine letters kun je herkennen omdat ze de test `ch >= 'a' and ch <= 'z' True` maken.

10.4.3 Substrings met stappen

Substrings kunnen behalve een index voor begin en einde een derde argument krijgen, namelijk stapgrootte. Dit argument werkt equivalent aan het derde argument voor de **range**() functie. De syntax voor substrings is <string>[<begin>:<einde>:<stap>]. Indien niet opgegeven, is de stapgrootte 1.

Een veelgebruikte toepassing van de stapgrootte is het gebruik van een negatieve waarde om de string te inverteren.

```
fruit = "banaan"
print( fruit[::2] )
print( fruit[1::2] )
print( fruit[::-1] )
print( fruit[::-2] )
```

Het inverteren van een string via `[::-1]` is conceptueel gelijk aan het doorlopen van de string vanaf het laatste teken tot het eerst met achterwaartse stappen van grootte 1.

```
fruit = "banaan"
print( fruit[::-1] )
for i in range( 5, -1, -1 ):
    print( fruit[i] )
```

10.5 Strings zijn onveranderbaar

Een kerneigenschap van strings is dat ze onveranderbaar (Engels: "immutable") zijn. Dit betekent dat strings niet kunnen wijzigen. Bijvoorbeeld, je kunt niet een teken in een string wijzigen door er een nieuwe waarde aan toe te kennen. Ter demonstratie: de volgende code leidt tot een runtime error als je hem probeert uit te voeren:

```
fruit = "aaldbei"
fruit[2] = "r"  # Runtime error!
print( fruit )
```

Als je een wijziging wilt maken in een string, moet je een nieuwe string maken die de wijziging omvat; je kunt daarna de nieuwe string toekennen aan de bestaande variabele als je wilt. Bijvoorbeeld:

```
fruit = "aaldbei"
fruit = fruit[:2] + "r" + fruit[3:]
print( fruit )
```

De reden waarom strings onveranderbaar zijn, is te technisch om hier te bespreken. Onthoud alleen dat als je een string wilt wijzigen, je geen nieuwe waarde kunt toekennen aan een individueel teken uit de string. In plaats daarvan moet je de variabele die de string bevat geheel overschrijven.

10.6 `string` methodes

Er is een aantal methodes beschikbaar die ontworpen zijn om strings te bewerken. Al deze methodes worden toegepast op een string om een operatie uit te voeren. Omdat strings onveranderbaar zijn, zullen deze methodes nooit de string waarop ze werken wijzigen, maar ze retourneren in plaats daarvan een gewijzigde versie van de string.

Net als de **`format`**`()` methode die in hoofdstuk 5 besproken is, worden al de string methodes aangeroepen via de syntax `<string>.<methode>()`, met andere woorden, je specificeert de string waarop de methode moet werken, gevolgd door een punt, gevolgd door de methode. Je zult dit vanaf nu vaker tegenkomen, en de reden dat methodes op deze manier aangeroepen moeten worden volgt in latere hoofdstukken (20 en verder).

De meeste string methodes zijn geen deel van een module, en je kunt ze aanroepen zonder iets te moeten importeren. Er is een `string` module die bepaalde nuttige constanten en methodes bevat die je in je programma's kunt gebruiken, maar de methodes die ik hier noem kun je gebruiken zonder de `string` module te importeren.

10.6.1 `strip()`

`strip()` verwijdert spaties aan het begin en einde van een string, inclusief eventuele "newline" tekens en andere tekens die als spaties gezien kunnen worden. Als je iets anders dan spaties wilt verwijderen, kun je als parameter een string meegeven die bestaat uit alle te verwijderen tekens.

```
s = "    En nu iets heel anders \n    "
print( "["+s+"]" )
s = s.strip()
print( "["+s+"]" )
```

10.6.2 `upper()` en `lower()`

`upper()` creëert een versie van een string met alle letters als hoofdletters. `lower()` werkt op dezelfde manier, maar maakt van alle letters kleine letters. Geen van beide methodes heeft parameters.

```
s = "The Meaning of Life"
print( s )
print( s.upper() )
print( s.lower() )
```

10.6.3 `find()`

`find()` kun je gebruiken om in een string te zoeken naar de start-index van een bepaalde substring. Als parameter krijgt de methode de gezochte substring. Optioneel kan een tweede, numerieke parameter meegegeven worden die aangeeft bij welke index gestart moet worden met zoeken. Een optionele derde, numerieke parameter is de index waarbij het zoeken moet stoppen. Je krijgt de laagste index waarbij de substring gevonden wordt terug, of -1 als de substring niet voorkomt.

```
s = "Humpty Dumpty zat op de muur"
print( s.find( "zat" ) )
print( s.find( "t" ) )
print( s.find( "t", 12 ) )
print( s.find( "q" ) )
```

10.6.4 `replace()`

`replace()` vervangt alle instanties van een substring in een string door een andere substring. Als parameters krijgt het de substring die gezocht wordt, en de substring die als vervanging dient. Optioneel kan een derde, numerieke parameter meegegeven worden die aangeeft hoe vaak een vervanging moet plaatsvinden.

Ik wil hier nogmaals benadrukken dat strings onveranderbaar zijn, dus de `replace()` functie maakt niet echt vervangingen in de string; hij retourneert een nieuwe string die een kopie is van de originele string, waarbij de vervangingen zijn gemaakt.

```
s = 'Humpty Dumpty zat op de muur'
print( s.replace( 'zat op', 'viel van' ) )
```

10.6.5 split()

split() splitst een string op in woorden, gebaseerd op een gegeven teken of substring die als separator beschouwd wordt. De separator is een parameter, en als die niet is opgegeven, is de separator de spatie, wat inhoudt dat je een string inderdaad opsplitst in de afzonderlijke woorden (waarbij interpunctie die aan woorden vastzit beschouwd wordt als een onderdeel van de desbetreffende woorden). Als de separator meerdere keren naast elkaar staat, dan worden de extra separatoren genegeerd (dat wil zeggen dat met spaties als separator, het niet uitmaakt of er tussen twee woorden één spatie staat, of meerdere).

Het resultaat van deze opsplitsing is een "lijst" van woorden. Lijsten komen aan bod in hoofdstuk 12, dus ik ga er nu weinig over zeggen. Ik zeg alleen dat als je de afzonderlijke woorden in de lijst wilt benaderen, je de constructie **for** <woord> **in** <lijst> kunt gebruiken.

```
s = 'Humpty Dumpty      zat   op de muur   '
lijst = s.split()
for woord in lijst:
    print( woord )
```

Een nuttige toepassing van het opsplitsen van een string is dat je het kunt gebruiken om sommige basale bestandsformaten te decoderen. Bijvoorbeeld, een CSV (Comma-Separated Value) bestand is een eenvoudig formaat, waarbij iedere regel van het bestand bestaat uit waardes die van elkaar gescheiden zijn door komma's. De waardes kun je uit een regel halen middels de split() methode.[13]

```
csv = "2016,september,28,Data Processing,Tilburg University"
waardes = csv.split( ',' )
for waarde in waardes:
    print( waarde )
```

10.6.6 join()

join() is de tegenhanger van split(). join() plakt een lijst van woorden aaneen tot een string, waarbij de woorden in de string van elkaar gescheiden zijn middels een specifieke separator. Dit klinkt wellicht alsof dit een methode van lijsten zou moeten zijn, maar om historische redenen is het gedefinieerd als een string methode. Omdat alle string methodes worden aangeroepen als <string>.<methode>(), moet er een string staan voor de aanroep van join(). Die string is de separator die je wilt gebruiken, terwijl de parameter die je meegeeft de lijst is waarvan je de woorden aan elkaar wilt plakken. De retourwaarde is, als altijd, de resulterende string.

```
s = "Humpty;Dumpty;zat;op;de;muur"
lijst = s.split( ';' )
s = " ".join( lijst )
print( s )
```

[13] In werkelijkheid is het vaak iets ingewikkelder omdat er komma's kunnen staan in de waardes die zijn opgeslagen in het CSV bestand, dus het is afhankelijk van de inhoud van het bestand of de genoemde aanpak werkt. Ik ga meer zeggen over CSV bestanden in hoofdstuk 26.

10.6.7 Oefening

Opgave In de string `"Barbara had een bar, waar ze rabarbar verkocht, en die daarom de rabarbarbarbarabar werd genoemd."` is het woord `"rabarber"` verkeerd gespeld. Gebruik `replace()` om alle voorkomende gevallen van deze fout te verbeteren.

Opgave Neem de string `"Niemand verwacht de Spaanse Inquisitie!# In feite, zij die de Spaanse Inquisitie wel verwachten..."` en toon hem tot aan, maar niet inclusief, de hash mark (#). Gebruik `find()` om de index van de hash mark te bepalen.

Opgave Schrijf een programma dat een "schone" versie van alle woorden in de string print. Alle tekens die geen letter zijn, worden niet beschouwd als deel van een woord, maar als separator. Alle letters moeten in kleine letters worden omgezet. Bijvoorbeeld, de string `"Ik heb zo'n honger."` produceert vijf woorden, namelijk `"ik"`, `"heb"`, `"zo"`, `"n"`, en `"honger"`. Je kunt de functie die je eerder hebt geschreven voor het schoonmaken van strings hier gebruiken.

10.7 Codering van tekens

Alle computersystemen gebruiken een manier om tekens te coderen. De basis codering die door (vrijwel) ieder systeem ondersteund wordt is de standaard ASCII code. Dit is een 7-bits code, die 128 verschillende tekens kan weergeven. Een aantal van deze tekens (met name die met de laagste nummers) zijn controle tekens die een speciale functie hebben. De meeste hiervan zijn alleen nuttig voor ouderwetse computersystemen, maar de tabulatie, "newline," en "backspace" tekens zitten er ook tussen. Als je alleen de tekens gebruikt die op een toetsenbord met US configuratie voorkomen, beperk je je tot standaard ASCII tekens.

Vandaag de dag gebruiken veel systemen Unicode. Unicode ondersteunt veel meer tekens. Er zijn verschillende manieren om tekens op te slaan als Unicode tekens. De meest bekende is UTF-8, die één byte gebruikt voor ieder van de ASCII tekens, maar meerdere bytes voor alle andere tekens (een byte is een groep van 8 bits, waarbij iedere bit een 1 of een 0 bevat). Andere Unicode coderingen gebruiken meerdere bytes voor alle tekens. Python ondersteunt UTF-8, wat betekent dat het ook reguliere ASCII coderingen ondersteunt.

10.7.1 ASCII

Hieronder zie je de ASCII tabel. De enige tekens die ik heb weggelaten zijn de speciale tekens. Die hebben nummers nul tot en met 31, en 127. 32 is de spatie. Ik geef ook de hexadecimale code voor ieder teken weer naast de decimale code. Die worden in een later hoofdstuk relevant.

Zoals je kunt zien heeft ieder teken een getal. Om in een Python programma te achterhalen wat het nummer is van een teken, kun je de **`ord()`** functie gebruiken. **`ord`**`("A")`, bijvoorbeeld, retourneert het nummer van `"A"`, dat 65 is, zoals je kunt zien. De tegenhanger van **`ord()`** is de **`chr()`** functie. **`chr()`** krijgt een nummer als argument, en retourneert het teken dat hoort bij dat nummer. Bijvoorbeeld, **`chr`**`(65)` is de letter `"A"`.

```
DC HX      DC HX      DC HX      DC HX      DC HX      DC HX
32 20      48 30 0    64 40 @    80 50 P    96 60 `   112 70 p
33 21 !    49 31 1    65 41 A    81 51 Q    97 61 a   113 71 q
34 22 "    50 32 2    66 42 B    82 52 R    98 62 b   114 72 r
35 23 #    51 33 3    67 43 C    83 53 S    99 63 c   115 73 s
36 24 $    52 34 4    68 44 D    84 54 T   100 64 d   116 74 t
37 25 %    53 35 5    69 45 E    85 55 U   101 65 e   117 75 u
38 26 &    54 36 6    70 46 F    86 56 V   102 66 f   118 76 v
39 27 '    55 37 7    71 47 G    87 57 W   103 67 g   119 77 w
40 28 (    56 38 8    72 48 H    88 58 X   104 68 h   120 78 x
41 29 )    57 39 9    73 49 I    89 59 Y   105 69 i   121 79 y
42 2A *    58 3A :    74 4A J    90 5A Z   106 6A j   122 7A z
43 2B +    59 3B ;    75 4B K    91 5B [   107 6B k   123 7B {
44 2C ,    60 3C <    76 4C L    92 5C \   108 6C l   124 7C |
45 2D -    61 3D =    77 4D M    93 5D ]   109 6D m   125 7D }
46 2E .    62 3E >    78 4E N    94 5E ^   110 6E n   126 7E ~
47 2F /    63 3F ?    79 4F O    95 5F _   111 6F o
```

Een vergelijking tussen strings waarin alleen deze tekens gebruikt worden, gebruikt de nummers van de teken om te bepalen welke van de strings “kleiner” is. Bijvoorbeeld, de string `"mango"` is groter dan de string `"mangaan"`, omdat het eerste verschil tussen de strings het vierde teken is, wat `"o"` is voor `"mango"` en `"a"` voor `"mangaan"`. Omdat het nummer voor `"o"` groter is dan het nummer voor `"a"`, wordt de string `"mango"` beschouwd als groter dan de string `"mangaan"`. Dit is feitelijk een alfabetische vergelijking. Als er tekens in de string voorkomen die geen letters zijn, kun je in de ASCII tabel nakijken welke als kleiner beschouwd worden. Zie hoe alle cijfers kleiner zijn dan letters.

```
print( ord( 'A' ) )
print( ord( 'a' ) )
print( chr( 65 ) )
print( chr( 97 ) )
print( "mango" > "mangaan" )
```

Je kunt deze nummers en tekens die ermee geassocieerd zijn gebruiken in allerhande nuttige berekeningen. Bijvoorbeeld, als je wilt weten welke de twaalfde letter na de `"g"` is, kun je dat als volgt berekenen:

```
print( "De twaalfde letter na g is", chr( ord( "g" )+12 ) )
```

Om een uitgebreider voorbeeld van wat je kunt doen met codes voor tekens te laten zien, is hier een programma dat de ASCII tabel genereert als matrix:

listing1005.py

```
print( ' ', end='' )
for i in range(16):
    if i < 10:
        print( ' '+chr( ord( '0' )+i ), end='' )
    else:
        print( ' '+chr( ord( 'A' )+i-10 ), end='' )
print()
```

```
for i in range( 2, 8 ):
    print( i, end='' )
    for j in range( 16 ):
        c = i*16+j
        print( ' '+chr( c ), end='' )
    print()
```

Ik prefereer het als je de functies **ord()** en **chr()** gebruikt in een programma waar je de codes van tekens moet gebruiken. Als je wilt refereren aan de code voor de letter `"A"`, schrijf dan niet 65, maar schrijf in plaats daarvan **ord**`("A")`. 65 heeft alleen betekenis voor mensen die ASCII codes van buiten kennen, en je programma zou betekenisvol moeten zijn voor iedereen. Daarbovenop komt nog dat, hoewel ASCII een zeer wijd verbreide standaard is, er nog steeds computers zijn die andere manieren van het coderen van tekens gebruiken, dus de code voor `"A"` is niet noodzakelijkerwijs 65 (inderdaad, IBM, ik heb het over jou).

10.7.2 UTF-8

Python ondersteunt Unicode, specifiek de meeste gebruikelijke versie van Unicode, namelijk UTF-8. Dit betekent dat je allerhande "vreemde" tekens kunt gebruiken. Ik legde uit bij de beschrijving van functie en variabele namen dat je "letters" in die namen kunt gebruiken. Je nam toen waarschijnlijk aan dat ik `"A"` tot en met `"Z"` en `"a"` tot en met `"z"` bedoelde. Het grappige is dat het afhankelijk is van je computersysteem wat daadwerkelijk beschouwd wordt als letter. Bijvoorbeeld, als in je computer staat ingesteld dat de taal die je gebruikt Duits is, dan kun je letters gebruiken met umlauts. Ook het Nederlands heeft speciale letters. Ik raad je echter ten zeerste af dit soort speciale letters te gebruiken in namen voor functies en variabelen. Niet alleen zijn ze lastig te typen, maar ze maken ook je programma minder goed overdraagbaar naar andere systemen.

In UTF-8 kun je in strings de reguliere tekens opnemen precies als je zou verwachten. Je kunt ook speciale letters opnemen, maar die zien er meestal anders uit dan je zou verwachten. Omdat Python UTF-8 ondersteunt, moet je voorzichtig zijn als je teksten kopieert van, bijvoorbeeld, een tekstverwerker. Tekstverwerkers hebben de irritante gewoonte om tekens te veranderen in andere tekens, bijvoorbeeld "rechte" aanhalingstekens in "kromme," of een min-teken in een "dash." Als je zulke tekens kopieert in je programma, zal Python de tekens accepteren, maar zal ze dan niet beschouwen als, bijvoorbeeld, string begrenzingen.

Als je Unicode tekens in een string wilt opnemen, kun je dat doen met Unicode codes. Je moet dan het UTF-8 nummer van het teken kennen dat je wilt tonen. Je kunt dan de code `\u`*xxxx* opnemen, waarbij *xxxx* een hexadecimaal getal van vier hexadecimale cijfers is, om het corresponderende teken in de string te zetten. Bijvoorbeeld, de onderstaande code toont het Griekse alfabet:[14]

```
alpha = "\u0391"
for i in range( 25 ):
    print( chr( ord( alpha )+i ), end=" " )
```

[14] Er staat een vreemd teken in deze reeks Griekse letters, namelijk tussen de Rho aen de Sigma, dat gelijk is aan `\u03A2`, dat klaarblijkelijk geen legaal Unicode teken is.

Over het algemeen hoef je je niet druk te maken over teken coderingen. Ik raad je aan je te beperken tot ASCII waar mogelijk. Als je met Unicode tekens moet werken, gaan de zaken meestal automatisch goed, omdat Python Unicode ondersteunt. Af en toe zie ik vertalingsproblemen als van Unicode naar ASCII moet worden gegaan, wat meestal te maken heeft met bestandsverwerking. Het zal een tijdje duren voordat je dat soort problemen krijgt, en ik zal er meer over zeggen in hoofdstuk 16 en verder.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Strings
- Strings over meerdere regels
- Positive and negative indices voor strings
- Substrings
- Onveranderlijkheid van strings
- String methodes `strip()`, `upper()`, `lower()`, `find()`, `replace()`, `split()`, en `join()`
- Speciale tekens
- ASCII en UTF-8 coderingen

Opgaves

Opgave 10.1 Tel hoeveel er van iedere klinker (a, e, i, o, u) staan in een tekst string, en druk die teller af voor iedere klinker met een enkele geformatteerde string. Bedenk dat klinkers zowel hoofd- als kleine letters kunnen zijn.

Opgave 10.2 Hieronder staat een tekst met een aantal tekens tussen vierkante haken. Doorloop de tekst en druk alle tekens af die tussen vierkante haken staan.

exercise1002.py

```
tekst = """En ze stu[re]n [i]ngekleurde prentbriefkaarten van
plekken waarvan ze zich niet reali[s]eren dat ze er nooit
geweest zijn [a]an 'Iedereen op nummer 22, weer is prachti[g],
onz[e] kamer is aa[n]gekruisd. Missen jullie. E[t]en[ ]i[s]
vettig, maar we hebben een geweldig leuk restaurantje gevonden
in de achterstraatjes waar ze Heine[ke]n hebben en kaas en
uien chips en iemand die "Een beetje verliefd" speel[t] op een
a[c]cordeon' en je zit vier dagen vast op Schip[h]ol voor je
vijfdaagse vliegvakantie met niks anders te eten dan
uitgedroogde voorverpakte boterhammen..."""
```

Opgave 10.3 Druk een regel af met alle hoofdletters `"A"` tot en met `"Z"`. Druk eronder een regel af die of 13 letters afstand in het alfabet liggen ten opzichte van de letters erboven. Bijvoorbeeld, onder de `"A"` druk je de `"N"` af, onder de `"B"` druk je de `"O"` af, etcetera. Beschouw het alfabet als circulair, dat wil zeggen, na de `"Z"`, gaat het weer terug naar de `"A"`. Dit kan natuurlijk met twee print-commando's, maar probeer het te doen met loops en gebruik te maken van **`ord`**`()` en **`chr`**`()`.

Opgave 10.4 Tel in de tekst hieronder hoe vaak het woord "knap" voorkomt (via een programma, natuurlijk). Zowel hoofd- als kleine letters mogen worden gebruikt, en je moet wel bedenken dat het woord "knap" een zelfstanding woord moet zijn, en niet een deel van een ander woord. Hint: Als je netjes alle oefeningen hebt gedaan tot nu toe, heb je al een functie gebouwd die schone woorden uit een tekst haalt.

exercise1004.py

```
tekst = """Kapper Knap, de knappe kapper, knipt en kapt heel
knap, maar de knecht van kapper Knap, de knappe kapper,
knipt en kapt nog knapper dan kapper Knap, de knappe kapper."""
```

Opgave 10.5 Schrijf een programma dat een string neemt en er een nieuwe string van maakt die precies dezelfde tekens bevat als de originele string, maar in volgorde van hun ASCII codes. Bijvoorbeeld, de string `"Hello, world!"` geeft als resultaat de string `" !,Hdellloorw"`. Dit kan vrij gemakkelijk gedaan worden met "list" functies, maar die komen pas aan bod in hoofdstuk 12, dus probeer het nu te doen met string manipulatie.

Opgave 10.6 Typische autocorrectie maakt de volgende wijzigingen: (1) als een woord begint met twee hoofdletters gevolgd door een kleine letter, dan wordt de tweede hoofdletter gewijzigd in een kleine letter; (2) als een zin een woord bevat dat onmiddellijk gevolgd wordt door hetzelfde woord, dan wordt het tweede woord verwijderd; (3) als een zin begint met een kleine letter, dan wordt die gewijzigd in een hoofdletter; (4) als een woord volledig uit hoofdletters bestaat, behalve de eerste letter die een kleine letter is, dan wordt de kleine letter een hoofdletter en alle hoofdletters kleine letters; en (5) als de zin de naam van een dag bevat die niet met een hoofdletter begint, dan wordt de eerste letter in een hoofdletter veranderd. Schrijf een programma dat een zin neemt en die volgens deze autocorrectie regels aanpast. Je kunt het met de string hieronder testen.

exercise1006.py

```
zin = "en zo gebeurde het dat dat onze toevallige ontmoeting \
met de EErwaarde aRTHUR BElling een ommekeer betekende in ons \
leven, en vanaf dat moment gingen we iedere zondag naar \
de kerk van Sint sIMPEL bij Roombroodje MEt Jam."
```

Hoofdstuk 11

Tuples

Een tuple is een groep van één of meer waardes die als een geheel behandeld worden. Dit hoofdstuk legt uit hoe je tuples kunt herkennen en gebruiken.[15]

11.1 Gebruik van tuples

Een tuple bestaat uit een aantal waardes die van elkaar gescheiden zijn met komma's. Meestal worden tuples geschreven met haakjes eromheen, maar de haakjes zijn niet noodzakelijk (behalve in omstandigheden waar anders verwarring zou ontstaan). Bijvoorbeeld:

```
t1 = ("appel", "mango")
print( type( t1 ) )
t2 = "banaan", "kers"
print( type( t2 ) )
```

Je kunt in een tuple verschillende data types mixen.

```
t1 = ("appel", 3, 1.4)
t2 = ("appel", 3, 1.4, ("banaan", 5))
```

Je kunt de **len()** functie gebruiken om te bepalen hoeveel elementen een tuple heeft.

```
t1 = ("appel", "mango")
t2 = ("appel", 3, 1.4)
t3 = ("appel", 3, 1.4, ("banaan", 5))
print( len( t1 ) )
print( len( t2 ) )
print( len( t3 ) )
```

[15] In het Nederlands kun je "tuple" vertalen als "tupel." Je spreekt de twee woorden op dezelfde manier uit. Ik gebruik de Engelse schrijfwijze omdat tuple een data type is.

Merk op dat in dit voorbeeld de lengte van `t3` 4 is, en niet 5. Het laatste element van `t3` is de tuple `("banaan", 5)`, wat telt als één element.

Je kunt een **`for`** loop gebruiken om de elementen van een tuple te doorlopen.

```
t1 = ("appel", 3, 1.4, ("banaan", 5))
for element in t1:
    print( element )
```

Je kunt de **`max`**`()` en **`min`**`()` functies gebruiken om het maximum respectievelijk het minimum te bepalen van een tuple die bestaat uit getallen. Je kunt de elementen van een tuple met numerieke elementen bij elkaar optellen middels de **`sum`**`()` functie.

```
t1 = (327, 419, 101, 667, 925, 225)
print( max( t1 ) )
print( min( t1 ) )
print( sum( t1 ) )
```

Je kunt testen of een element onderdeel van een tuple is met behulp van de **`in`** operator.

```
t1 = ("appel", "banaan", "kers")
print( "banaan" in t1 )
print( "mango" in t1 )
```

11.1.1 Tuple assignments

Je kunt een tuple creëren door een assignment van een aantal waardes gescheiden door komma's aan een variabele. Haakjes zijn optioneel. Wat als je een tuple wilt creëren met slechts één element?

```
t1 = ("appel")
print( type( t1 ) )
```

Als je deze code uitvoert, zie je dat `t1` de class `str` heeft, dat wil zeggen, `t1` is een string. Dat er haakjes omheen staan bij de assignment maakt het niet tot een tuple. Python heeft een truukje om een tuple met slechts één element te maken, namelijk door een komma te plaatsen achter het ene element. Dit is niet erg intuïtief en ik zou het zelfs wat zwak willen noemen, maar historisch was dit de oplossing die een vroege versie van Python bevatte, en kennelijk heeft niemand iets beters weten te verzinnen.

```
t1 = ("appel",)
print( type( t1 ) )
print( len( t1 ) )
```

Python staat toe een tuple van variabelen links van de assignment operator te plaatsen. Dit is een uitzondering op de regel dat slechts één variabele links van de assignment operator staat. De waardes aan de rechterkant worden één voor één naar de linkerkant gekopieerd, van links naar rechts.

```
t1, t2 = "appel", "banaan"
print( t1 )
print( t2 )
```

Je kunt haakjes om de waardes aan de rechterkant zetten, en je kunt ook haakjes rond de variabelen aan de linkerkant zetten; dat maakt geen verschil.

Als je meer variabelen aan de linkerkant zet dan waardes aan de rechterkant, krijg je een runtime error. Hetzelfde geldt voor minder (met als uitzondering dat je precies één variabele plaatst). Je kunt wel tuples aan de rechterkant plaatsen door haakjes te gebruiken.

```
t1, t2 = ("apple", "banaan"), "kers"
print( t1 )
print( t2 )
```

11.1.2 Tuple indices

Net als bij strings, kun je individuele elementen van een tuple benaderen via indices. Waar bij strings de individuele elementen tekens zijn, zijn het bij tuples waardes. Bijvoorbeeld:

```
t1 = ("appel", "banaan", "kers", "doerian")
print( t1[2] )
```

Je kunt zelfs sub-tuples maken, met dezelfde regels als je hebt voor substrings (als je die niet meer weet, lees dan nog eens hoofdstuk 10). Een sub-tuple is ook weer een tuple. Bijvoorbeeld:

```
t1 = ("appel", "banaan", "kers", "doerian", "mango")
print( t1[1:4] )
```

Omdat tuples indices hebben, kun je als alternatief voor een **for** loop om de elementen van de tuple te doorlopen, gebruik maken van de indices.

listing1101.py

```
t1 = ("appel", "banaan", "kers", "doerian", "mango")
i = 0
while i < len( t1 ):
    print( t1[i] )
    i += 1
```

Opgave Schrijf een **for** loop die alle elementen van een tuple toont, en die ook hun indices toont.

11.1.3 Tuple vergelijkingen

Je kunt twee tuples met elkaar vergelijken met behulp van de reguliere vergelijkingsoperatoren. Deze operatoren vergelijken eerst de eerste elementen van beide tuples. Als ze

verschillend zijn, dan geeft de vergelijking van die twee elementen op basis van de regels die voor hun data types geldt, de gevraagde uitkomst. Als ze gelijk zijn, dan worden de volgende elementen van beide tuples met elkaar vergeleken, etcetera.

listing1102.py

```
t1 = ( "appel", "banaan" )
t2 = ( "appel", "banaan" )
t3 = ( "appel", "kers" )
t4 = ( "appel", "banaan", "kers" )
print( t1 == t2 )
print( t1 < t3 )
print( t1 > t4 )
print( t3 > t4 )
```

11.1.4 Tuple retourwaardes

In hoofdstuk 8 legde ik uit dat functies meerdere waardes kunnen retourneren. Als je zoiets programmeert, dan komt het er in feite op neer dat de functie een tuple retourneert. Om zulke retourwaardes af te handelen, doe je wat hierboven beschreven is voor "tuple assignments.".

11.2 Tuples zijn onveranderbaar

Net als strings, zijn tuples onveranderbaar. Dat wil zeggen dat je geen nieuwe waarde kunt toekennen aan een element van een tuple. Het voorbeeld hieronder veroorzaakt een runtime error als je het uitvoert.

```
t1 = ("appel", "banaan", "kers", "doerian")
t1[0] = "mango" # Runtime error
```

11.3 Toepassingen van tuples

Tuples worden niet vaak in Python code gebruikt (behalve als retourwaardes van functies). Een logische applicatie van tuples zou zijn de afhandeling van waardes die altijd voorkomen in kleine verzamelingen. Echter, object oriëntatie (hoofdstuk 20 en verder) biedt veel technieken en hulpmiddelen om met zulke kleine verzamelingen om te gaan, wat betekent dat programmeurs meestal voor object oriëntatie kiezen als ze dat soort zaken onder handen hebben.

Hier volgt toch een voorbeeld van het gebruik van tuples in een applicatie. Stel je voor dat je een applicatie moet schrijven die geometrische figuren in een 2-dimensionale ruimte bewerkt. Een concept dat je daarbij nodig hebt is een punt: een locatie in de 2D ruimte die geïdentificeerd wordt via twee coördinaten. In plaats van functies te schrijven die steeds een aparte X en een aparte Y coördinaat meekrijgen, kun je specificeren dat coördinaten worden meegegeven als tuples.

listing1103.py

```
from math import sqrt

# Retourneert de afstand tussen twee punten in 2-dimensionale
# ruimte. The punten zijn de parameters van de functie.
# Ieder punt is een tuple met twee numerieke waardes.
def afstand( p1, p2 ):
    return sqrt( (p1[0] - p2[0])**2 + (p1[1] - p2[1])**2 )

punt1 = (1,2)
punt2 = (5,5)
print( "Afstand tussen", punt1, "en", punt2, "is",
    afstand( punt1, punt2 ) )
```

Een voordeel van het gebruik van tuples op deze manier is dat het relatief eenvoudig is om een functie te schrijven die kan omgaan met coördinaten in willekeurige ruimtes.

listing1104.py

```
from math import sqrt

# afstand tussen twee punten in N-dimensionale ruimte.
# De punten hebben dezelfde dimensie, ze zijn allebei tuples
# met numerieke waardes, met dezelfde lengte.
def afstand( p1, p2 ):
    totaal = 0
    for i in range( len( p1 ) ):
        totaal += (p1[i] - p2[i])**2
    return sqrt( totaal )

# 1-dimensionale ruimte
punt1 = (1,)
punt2 = (5,)
print( "1D: afstand tussen", punt1, "en", punt2, "is",
    afstand( punt1, punt2 ) )

# 2-dimensionale ruimte
punt1 = (1,2)
punt2 = (5,5)
print( "2D: afstand tussen", punt1, "en", punt2, "is",
    afstand( punt1, punt2 ) )

# 3-dimensionale ruimte
punt1 = (1,2,4)
punt2 = (5,5,8)
print( "3D: afstand tussen", punt1, "en", punt2, "is",
    afstand( punt1, punt2 ) )
```

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Tuples
- Tuple assignments
- Tuple indices
- Onveranderbaarheid van tuples
- Toepassingen van tuples

Opgaves

Opgave 11.1 Een complex getal is een getal van de vorm `a + b`i, waarbij `a` en `b` constanten zijn, en i een speciale waarde, die gedefinieerd is als de wortel uit -1. Natuurlijk kun je niet echt de wortel uit -1 berekenen, dat zou een runtime error geven; in complexe berekeningen laat je altijd de i staan. Bijvoorbeeld, het complexe getal `3 + 2`i kan niet verder gesimplificeerd worden. Het optellen van complexe getallen `a + b`i en `c + d`i is gedefinieerd als `(a + c) + (b + d)`i. Representeer een complex getal als een tuple met twee numerieke waardes, en creëer een functie die de optelling van twee complexe getallen implementeert.[16]

Opgave 11.2 Vermenigvuldiging van twee complexe getallen `a + b`i en `c + d`i is gedefinieerd als `(a*c - b*d) + (a*d + b*c)`i. Schrijf een functie die de vermenigvuldiging van twee complexe getallen implementeert.

Opgave 11.3 Stel je een nieuw data type voor. Dit nieuwe data type noem ik de `inttuple`. Een `inttuple` is gedefinieerd als zijnde ofwel een integer, ofwel een tuple die `inttuples` als waardes bevat. Je ziet een voorbeeld van een `inttuple` hieronder. Schrijf een functie die alle integer waardes die in een `inttuple` zijn opgeslagen afdrukt. Hint: Omdat de `inttuple` recursief is gedefinieerd, is een recursieve functie waarschijnlijk de beste aanpak. Als je hoofdstuk 9 hebt overgeslagen, kun je waarschijnlijk deze opdracht beter ook overslaan. Als je hem wel maakt, dan kun je de **`isinstance()`** functie (uitgelegd in hoofdstuk 8) gebruiken om te bepalen of een element een integer of een tuple is. Als je alles correct doet, zal de functie als hij werkt op de `inttuple` die hieronder gegeven is, de getallen 1 tot en met 20 in volgorde afdrukken.

exercise1103.py

```
inttuple = ( 1, 2, ( 3, 4 ), 5, ( ( 6, 7, 8, ( 9, 10 ), 11 ), 12,
    13 ), ( ( 14, 15, 16 ), ( 17, 18, 19, 20 ) ) )
```

[16] Python heeft een apart data type `complex` dat complexe getallen representeert, dus er is eigenlijk geen noodzaak om complexe getallen als tuples te beschrijven, maar met als doel te oefenen met tuples werkt de opgave best.

Hoofdstuk 12

Lists

Een "list" (Engelse woord voor "lijst") is een geordende verzameling van data items, net zoals een tuple. Het verschil tussen lists en tuples is dat lists veranderbaar zijn. Dit maakt ze tot een zeer flexibele data structuur, waar je op veel manieren gebruik van kunt maken.[17]

12.1 Basis van lists

Een list is een verzameling (of "collectie") elementen.

De elementen van een list zijn *geordend*. Omdat ze geordend zijn, kun je ieder element van een list benaderen via een index, net zoals je de tekens in een string kunt benaderen. De indices beginnen bij nul, net als bij strings.

In Python kun je lists herkennen aan het feit dat de elementen van een list tussen vierkante haken ([]) staan. Je kunt het aantal elementen in een list achterhalen door middel van de **`len`**`()` functie. Via een **`for`** loop kun je de elementen van de list doorlopen. Je mag data types in een list mixen. Je kunt de functies **`max`**`()`, **`min`**`()` en **`sum`**`()` gebruiken op een list. Je

[17] Ik gebruik de Engelse benaming "list" (en het meervoud "lists") in plaats van het Nederlandse "lijst" omdat `list` een data type is.

kunt testen of een element voorkomt in een list via de **in** operator (of het niet voorkomen van een element via de **not in** operator).

listing1201.py

```
fruitlist = ["appel", "banaan", "kers", 27, 3.14]
print( len( fruitlist ) )
for element in fruitlist:
    print( element )
print( fruitlist[2] )

numlist = [314, 315, 642, 246, 129, 999]
print( max( numlist ) )
print( min( numlist ) )
print( sum( numlist ) )
print( 100 in numlist )
print( 999 in numlist )
```

Opgave Schrijf een **while** loop die elementen van een list afdrukt.

Afgezien van de vierkante haken, lijken lists veel op tuples. Maar er is een groot verschil...

12.2 Lists zijn veranderbaar

Omdat lists veranderbaar (Engels: "mutable") zijn, mag je de inhoud van een list wijzigen.

Om een element van de list te overschrijven, kun je er een nieuwe waarde aan toekennen middels een assignment.

```
fruitlist = ["appel", "banaan", "kers", "doerian", "mango"]
print( fruitlist )
fruitlist[2] = "aardbei"
print( fruitlist )
```

Je kunt ook een sub-list overschrijven door een nieuw list toe te kennen aan de sub-list. De sub-list en de nieuwe list hoeven niet dezelfde lengte te hebben.

```
fruitlist = ["appel", "banaan", "kers", "doerian", "mango"]
print( fruitlist )
fruitlist[1:3] = ["framboos", "aardbei", "aalbes"]
print( fruitlist )
```

Je kunt nieuwe elementen aan een list toevoegen door ze toe te kennen aan een lege sub-list.

```
fruitlist = ["appel", "banaan", "kers", "doerian", "mango"]
print( fruitlist )
fruitlist[1:1] = ["framboos", "aardbei", "aalbes"]
print( fruitlist )
```

Je kunt elementen van een list verwijderen door een lege list aan de te verwijderen sub-list toe te kennen.

```
fruitlist = ["appel", "banaan", "kers", "doerian", "mango"]
print( fruitlist )
fruitlist[1:3] = []
print( fruitlist )
```

Door sub-lists en assignments te gebruiken, kun je lists aanpassen op iedere manier die je wilt. Het is echter eenvoudiger om lists aan te passen via methodes. Er zijn allerlei handige methodes beschikbaar, die ik hieronder bespreek.

Opgave Neem een list die alleen woorden bevat (bijvoorbeeld één van de `fruitlists` hierboven). Verander ieder woord in de list in een woord dat alleen uit hoofdletters bestaat. Op dit punt in het boek doe je dat met behulp van een **`while`** loop die een variabele `i` laat starten bij 0 en laat oplopen tot **`len`**`(<list>)-1`. Gebruik `i` als index voor de list.

12.3 Lists en operatoren

Lists ondersteunen het gebruik van de operatoren $+$ en $*$. Deze operatoren werken op eenzelfde manier als ze werken bij strings.

Je kunt twee lists bij elkaar optellen middels de $+$ operator, en het resultaat is een list die de elementen bevat van beide opgetelde lists. Je moet het resultaat aan een variabele toekennen om het op te slaan.

Je kunt een list vermenigvuldigen met een integer om een list te creëren die de elementen van de originele list bevat, net zo vaak herhaald als de integer aangeeft. Dit kan een snelle manier zijn om een list te creëren waarvan alle elementen hetzelfde zijn.

```
fruitlist = ["appel", "banaan"] + ["kers", "doerian"]
print( fruitlist )
numlist = 10 * [0]
print( numlist )
```

Merk op: Met de $+$ kun je twee lists bij elkaar tellen, maar je kunt niet een element aan de list toevoegen, tenzij je dat nieuwe element in een list met slechts één element opneemt door er vierkante haken omheen te zetten. Als je iets dat geen list is probeert op te tellen bij een list, dan zal Python proberen dat iets te interpreteren als een list – en als dat kan (wat mogelijk is voor bijvoorbeeld een string, die kan worden geïnterpreteerd als een list met letters), dan zal het de optelling uitvoeren maar is het resultaat waarschijnlijk niet wat je verwacht. Bijvoorbeeld, de code hieronder probeert "kers" toe te voegen aan een list, maar hoewel geen van de twee manieren die gebruikt worden een fout veroorzaakt, doet alleen de tweede wat bedoeld is.

listing1202.py

```
fruitlist = ["appel", "banaan"]
fruitlist += "kers"
print( fruitlist )
```

```
fruitlist = ["appel", "banaan"]
fruitlist += ["kers"]
print( fruitlist )
```

12.4 List methodes

Python heeft een groot aantal methodes die lists wijzigen of informatie uit lists halen. Je hoeft geen module te importeren om ze te gebruiken. Omdat het methodes betreft, gebruik je de syntax `<list>.<methode>()`.

Belangrijk! Lists zijn veranderbaar en deze methodes veranderen vaak de list waarop ze werken! Dit is niet wat je gewend bent met string methodes, waarbij de methodes een nieuwe string maken en die retourneren, zonder de originele string aan te passen. De meeste list methodes hebben daarentegen een onomkeerbaar effect op de list waarop ze werken. Meestal hebben ze ook geen retourwaarde, en die heb je ook niet nodig, omdat het doel van de methodes is de list te wijzigen.

12.4.1 `append()`

`append()` plakt een nieuw element aan het einde van een list. Je roept de methode aan met het nieuwe element als argument.

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
print( fruitlist )
fruitlist.append( "mango" )
print( fruitlist )
```

Zoals hierboven is beschreven, kun je in plaats van de `append()` methode een element aan een list toevoegen via de + operator, en het resultaat weer toekennen aan de variabele die de list bevat. De `append()` methode is echter leesbaarder. `<list>.append(<element>)` is equivalent met `<list>[len(<list>):] = [<element>]`, of `<list> += [<element>]`.

12.4.2 `extend()`

`extend()` maakt een list langer door alle elementen van een tweede list aan het einde te van de eerste list toe te voegen. Je roept de methode aan met de list met de nieuwe elementen als argument.

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
print( fruitlist )
fruitlist.extend( ["framboos", "aardbei", "aalbes"] )
print( fruitlist )
```

Net als bij de `append()` methode kun je in plaats van de `extend()` methode de + operator gebruiken en het resultaat toekennen aan de originele list. En net als bij de `append()` methode, is het gebruik van de `extend()` methode te prefereren vanwege de leesbaarheid. `<list>.extend(<addlist>)` is equivalent met `<list>[len(<list>):] = <addlist>`.

12.4.3 insert()

insert() geeft de mogelijkheid een element aan een list toe te voegen op een specifieke positie. Je roept de methode aan met twee argumenten, waarvan de eerste de index is van de positie waar het nieuwe element moet komen, en de tweede het nieuwe element zelf. Als je een element toe wilt voegen aan het begin van de list, kun je index 0 gebruiken.

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
print( fruitlist )
fruitlist.insert( 2, "mango" )
print( fruitlist )
```

<list>.insert(<i>,<element>) is equivalent met <list>[<i>:<i>] = [<element>].

12.4.4 remove()

remove() laat je een element van de list verwijderen. Het element dat je wilt verwijderen geef je mee als argument. Als dit element meerdere keren voorkomt in de list, wordt de eerste instantie (die met de laagste index) verwijderd. Als je een element probeert te verwijderen dat niet voorkomt op de list, geeft dat een runtime error.

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
print( fruitlist )
fruitlist.remove( "banaan" )
print( fruitlist )
```

12.4.5 pop()

Net als remove(), verwijdert pop() een element van de list, maar doet dat via een index. Er is één numeriek argument, dat optioneel is, dat de index is van het te verwijderen element. Als geen argument wordt meegegeven, wordt het laatste element van de list verwijdered. Als een index wordt meegegeven die buiten de list valt, volgt een runtime error.

Een groot verschil tussen remove() en pop() is dat pop() een retourwaarde heeft, namelijk het element dat verwijderd is. Dit zorgt ervoor dat je via pop() snel alle elementen van een list kunt verwerken terwijl je de list leegmaakt.

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
print( fruitlist )
print( fruitlist.pop() )
print( fruitlist )
print( fruitlist.pop( 0 ) )
print( fruitlist )
```

12.4.6 del

del is noch een methode noch een functie, maar omdat het vaak in één adem genoemd wordt met remove() en pop(), zet ik het hier neer. **del** is een gereserveerd woord dat een

list element verwijderd, of zelfs een sub-list, via de index. Het lijkt op wat `pop()` doet, maar heeft geen retourwaarde. `pop()` kan ook niet gebruikt worden op sub-lists. **`del`** gebruik je via de syntax **`del`** `<list>[<index>]` of **`del`** `<list>[<index1>:<index2>]`.

```
fruitlist = ["appel", "banaan", "kers", "banaan", "doerian"]
del fruitlist[3]
print( fruitlist )
```

12.4.7 `index()`

`index()` retourneert de index van de eerste instantie in een list van het element dat als argument aan de methode is meegegeven. Een runtime error volgt als het element niet voorkomt op de list.

```
fruitlist = ["appel", "banaan", "kers", "banaan", "doerian"]
print( fruitlist.index( "banaan" ) )
```

12.4.8 `count()`

`count()` retourneert een integer die aangeeft hoe vaak het element dat als argument is meegegeven voorkomt in de list.

```
fruitlist = ["appel", "banaan", "kers", "banaan", "doerian"]
print( fruitlist.count( "banaan" ) )
```

12.4.9 `sort()`

`sort()` sorteert de elementen van de list, van laag naar hoog. Als de elementen strings zijn, betreft het een alfabetische sortering. Als de elementen getallen zijn, betreft het een numerieke sortering. Als de elementen gemixt zijn, volgt een runtime error, tenzij bepaalde extra argumenten zijn meegegeven.

listing1203.py

```
fruitlist = ["appel", "aardbei", "banaan", "framboos",
    "kers", "banaan", "doerian", "mango"]
fruitlist.sort()
print( fruitlist )

numlist = [314, 315, 642, 246, 129, 999]
numlist.sort()
print( numlist )
```

Om van hoog naar laag te sorteren, kun je een argument `reverse=<boolean>` meegeven.

```
fruitlist = ["appel", "aardbei", "banaan", "framboos",
    "kers", "banaan", "doerian", "aalbes"]
fruitlist.sort( reverse=True )
print( fruitlist )
```

Een ander argument dat je `sort()` kunt meegeven is een "key" (Engels voor sleutel). Je geeft dit argument mee volgens de syntax `<list>.sort( key=<key> )`, waarbij `<key>` een functie is die één element meekrijgt (namelijk het element dat gesorteerd moet worden) en die een waarde retourneert die als sorteringssleutel gebruikt moet worden. Een typische toepassing van het key argument is als je een list van strings wilt sorteren, waarbij je de strings "case-insensitive" (dat wil zeggen zonder verschil te maken tussen hoofd- en kleine letters) wilt sorteren. Dus als key wil je de waarde van het element volledig als kleine letters gebruiken. Dat kun je doen door als key functie **`str`**`.lower()` mee te geven. Je roept dan de `sort()` methode aan als in het volgende voorbeeld:

listing1204.py

```
fruitlist = ["appel", "Aardbei", "banaan", "framboos",
    "KERS", "banaana", "doerian", "mango"]
fruitlist.sort()
print( fruitlist )
fruitlist.sort( key=str.lower ) # case-insensitive sort
print( fruitlist )
```

Merk op dat je bij het key argument geen haakjes achter de functie naam zet. Dit is namelijk geen functie-aanroep, het is een argument dat Python vertelt welke functie gebruikt moet worden om de sorteringssleutel te genereren. Je kunt ook je eigen functies gebruiken als key. Bijvoorbeeld, in de volgende code wordt `numlist` gesorteerd met de cijfers van de drie-cijferige getallen in omgekeerde volgorde:

listing1205.py

```
def keercijfersom( item ):
    return (item%10)*100 + (int(item/10)%10)*10 + int(item/100)

numlist = [314, 315, 642, 246, 129, 999]
numlist.sort( key=keercijfersom )
print( numlist )
```

Hier is een ander voorbeeld, waarbij een list van strings gesorteerd wordt op string lengte als eerste (korte strings vóór lange strings), en alleen bij gelijke lengte op alfabetische volgorde:

listing1206.py

```
def len_alfabetisch( element ):
    return len( element ), element

fruitlist = ["appel", "aardbei", "banaan", "framboos",
    "kers", "banaan", "doerian", "mango"]
```

```
fruitlist.sort( key=len_alfabetisch )
print( fruitlist )
```

Merk op dat de `len_alfabetisch()` functie een tuple retourneert. Zoals in hoofdstuk 11 werd uitgelegd, als je twee tuples vergelijkt worden eerst de eerste elementen van de tuples vergeleken, en alleen als die gelijk zijn worden de tweede elementen vergeleken.

Op dit punt kan ik een typisch voorbeeld geven van het gebruik van een "anonieme functie," die ik introduceerde in hoofdstuk 8 (als je die niet hebt overgeslagen). Als je een anonieme functie gebruikt om de key voor een `sort()` methode te specificeren, in plaats van als een separate functie elders in het programma, houd je de code compact en leesbaar.

listing1207.py

```
fruitlist = ["appel", "aardbei", "banaan", "framboos",
    "kers", "banaan", "doerian", "mango"]
fruitlist.sort( key=lambda x: (len(x),x) )
print( fruitlist )
```

12.4.10 `reverse()`

`reverse()` zet de elementen van de list in omgekeerde volgorde.

```
fruitlist = ["appel", "aardbei", "banaan", "framboos",
    "kers", "banaan", "doerian", "aalbes"]
fruitlist.reverse()
print( fruitlist )
```

12.4.11 Oefening

Opgave Schrijf een programma dat de gebruiker vraagt wat data in te geven, bijvoorbeeld de namen van zijn of haar vrienden. De gebruiker geeft aan te stoppen met data ingeven als alleen `Enter` wordt ingedrukt. Het programma toont dan alle ingegeven data items, alfabetisch gesorteerd. Print ieder item apart, op een eigen regel.

Opgave Sorteer een list van nummers op absolute waarde. Hint: Gebruik de **`abs`**`()` functie als key.

Opgave Tel hoe vaak iedere letter (case-insensitive) voorkomt in een string. Je mag alle tekens negeren die geen letter zijn. Je zou dit natuurlijk kunnen doen met 26 variabelen, maar het is veel beter om een list te gebruiken. Zet alle 26 elementen van de list bij de start op 0. Wanneer de letters geteld zijn, druk je alle resulterende tellingen af. Als index voor de list kun je **`ord`**`(letter) - `**`ord`**`("a")` gebruiken, waarbij "letter" een kleine letter is (de functie **`ord`**`()` is uitgelegd in hoofdstuk 10).

12.5 Alias

Als je een variabele die een list omvat toekent aan een andere variabele middels de assignment operator (=), denk je misschien dat je een kopie hebt gemaakt van de list. Maar dat is niet wat er gebeurt. Je maakt op deze manier een *alias* voor de list, dat wil zeggen, een nieuwe variabele die refereert aan precies dezelfde list. Je kunt de nieuwe variabele gebruiken als een list, maar iedere wijziging die je maakt in de list waaraan de variabele refereert, is ook te zien is in de list waaraan de originele variabele refereert, en vice versa. Het zijn niet twee verschillende lists.

listing1208.py

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
fruitlist2 = fruitlist
print( fruitlist )
print( fruitlist2 )
fruitlist2[2] = "mango"
print( fruitlist )
print( fruitlist2 )
```

Iedere variabele in Python heeft een identificatie nummer. Dat nummer kun je zien met de **id()** functie. Het ID nummer geeft aan welke geheugenadres door de variabele gebruikt wordt. Voor een alias van een list is de ID (logischerwijs) hetzelfde als voor de originele list.

listing1209.py

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
fruitlist2 = fruitlist
print( id( fruitlist ) )
print( id( fruitlist2 ) )
```

Als je een kopie van een list wilt creëren, kun je dat doen met een klein truukje. In plaats van `<nieuwlist> = <oudlist>` te gebruiken, gebruik je `<nieuwlist> = <oudlist>[:]`.

listing1210.py

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
fruitlist2 = fruitlist
fruitlist3 = fruitlist[:]

print( id( fruitlist ) )
print( id( fruitlist2 ) )
print( id( fruitlist3 ) )

fruitlist[2] = "mango"
print( fruitlist )
print( fruitlist2 )
print( fruitlist3 )
```

12.5.1 `is`

Het gereserveerde woord **`is`** is geïntroduceerd om de identiteiten van twee variabelen met elkaar te vergelijken.

listing1211.py

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
fruitlist2 = fruitlist
fruitlist3 = fruitlist[:]

print( fruitlist is fruitlist2 )
print( fruitlist is fruitlist3 )
print( fruitlist2 is fruitlist3 )
```

Zoals je kunt zien, weet **`is`** te bepalen dat `fruitlist` en `fruitlist2` een alias van elkaar zijn, maar dat `fruitlist3` niet dezelfde list is. Als je ze echter vergelijkt met de == operator, zijn de resultaten anders dan als je ze vergelijkt met **`is`**:

listing1212.py

```
fruitlist = ["appel", "banaan", "kers", "doerian"]
fruitlist2 = fruitlist
fruitlist3 = fruitlist[:]

print( fruitlist == fruitlist2 )
print( fruitlist == fruitlist3 )
print( fruitlist2 == fruitlist3 )
```

De == operator vergelijkt de inhoud van de lists, dus er wordt **`True`** geretourneerd voor alle vergelijkingen. Voor data types waarvoor de == niet speciaal gedefinieerd is, wordt een identiteitsvergelijking uitgevoerd. Maar voor lists is de == operator gedefinieerd als een vergelijking van list inhoud. Ik ga dieper op dit onderwerp in als ik het ga hebben over "operator overloading" in hoofdstuk 21. .

12.5.2 Ondiepe versus diepe kopieën

Als er items op een list voorkomen die zelf lists zijn (of andere veranderbare data structuren, die in de komende hoofdstukken aan bod komen), kan het kopiëren van een list middels `<nieuwlist> = <oudlist>[:]` problemen geven. De reden is dat een dergelijke kopie een "ondiepe kopie" is. Dat betekent dat de kopie-operatie ieder element van de list via een reguliere assignment operator kopieert, wat betekent dat items in de nieuwe list die zelf een list zijn een alias worden van van de items in de originele list.

listing1213.py

```
numlist = [ 1, 2, [3, 4] ]
copylist = numlist[:]

numlist[0] = 5
numlist[2][0] = 6
```

```
print( numlist )
print( copylist )
```

In de code hierboven zie je dat de assignment `numlist[0] = 5` alleen een wijziging maakt in `numlist`, aangezien `copylist` een (ondiepe) kopie is van `numlist`. Maar de assignment `numlist[2][0] = 6` maakt een wijziging in beide lists, omdat de sub-list `[3, 4]` in `copylist` opgeslagen is als een alias.

Om een "diepe kopie" van een list te maken (dat wil zeggen, een kopie die ook echte kopieën bevat van alle veranderbare substructuren in de list, en ook echte kopieën van alle veranderbare substructuren in veranderbare substructuren in de list, etcetera), kun je de module `copy` gebruiken. De `deepcopy()` functie van de `copy` module maakt diepe kopieën van iedere willekeurige veranderbare data structuur. Je geeft aan de `deepcopy()` functie de te kopiëren data structuur als argument mee, en je krijgt als retourwaarde een diepe kopie van deze data structuur.

listing1214.py

```
from copy import deepcopy

numlist = [ 1, 2, [3, 4] ]
copylist = deepcopy( numlist )

numlist[0] = 5
numlist[2][0] = 6
print( numlist )
print( copylist )
```

De `copy` module bevat ook een functie `copy()` die ondiepe kopieën maakt. Je vraagt je misschien af waarom die functie bestaat; je kunt immers ook ondiepe kopieën maken met `<nieuwlist> = <oudlist>[:]`. Het antwoord is dat de `copy` module niet alleen werkt voor lists, maar voor alle veranderbare data structuren, en voor niet iedere data structuur is er een truukje als dat voor lists beschikbaar.

12.5.3 Lists als argumenten

Als je een list aan een functie meegeeft als argument, dan is dit een zogeheten "pass by reference" ("doorgifte via een referentie" – waarbij je een referentie kunt beschouwen als een alias). De parameter die de list bevat, en die een locale variabele voor de functie is, bevat een alias voor de list die je hebt meegegeven. Dat betekent dat de functie de inhoud van de list kan wijzigen.

Dit is een belangrijk punt, dus ik herhaal het: als je een veranderbare data structuur meegeeft als argument aan een functie, dan krijgt de functie een alias voor de data structuur, en dus kan de inhoud van de data structuur gewijzigd worden door de functie.

Je moet dus weten of een functie waaraan je een list meegeeft de list wel of niet zal wijzigen. Als je niet wilt dat de functie de originele list wijzigt, en je weet niet of de functie dat zal doen, dan moet je een diepe kopie van de list meegeven aan de functie.

listing1215.py

```
def wijziglist( x ):
    if len( x ) > 0:
        x[0] = "FRUIT!"

fruitlist = ["appel", "banaan", "kers", "doerian"]
wijziglist( fruitlist )
print( fruitlist )
```

De reden dat een list als alias wordt meegegeven, en niet als een diepe kopie, is dat technisch gezien ieder argument dat een functie meekrijgt in de computer moet worden opgeslagen in een specifiek stuk geheugen dat een deel is van de processor. Dit is de "stack," en dit stack geheugen is niet erg omvangrijk. Omdat lists enorm groot kunnen zijn, zou een programma dat lists op de stack zet allerlei runtime errors kunnen veroorzaken. In Python, net als in de meeste andere programmeertalen, worden alleen de basis data types (zoals integers, floats, en strings) als waarde aan een functie meegegeven, en alle andere data structuren als alias.

12.6 Geneste lists

Elementen van een list kunnen zelf ook lists zijn (die ook weer lists kunnen bevatten, etcetera). Op deze manier kun je een matrix in je programma creëren. Bijvoorbeeld, je kunt een boter-kaas-eieren bord als volgt bouwen (een liggend streepje is een lege cel):

```
bord = [ ["-", "-", "-"], ["-", "-", "-"], ["-", "-", "-"] ]
```

De eerste rij van het bord is `bord[0]`, de tweede rij is `bord[1]`, en de derde rij is `bord[2]`. Als je de eerste cel van de eerste rij wilt benaderen, is dat `bord[0][0]`, de tweede cel is `bord[0][1]` en de derde cel is `bord[0][2]`. Bijvoorbeeld, om een "X" in het midden van het bord te plaatsen, en een "O" in de linkerbovenhoek, gebruik je de code hieronder. Deze code toont ook het bord op een nette manier (met labels op de rijen en kolommen).

listing1216.py

```
def toon_bord( b ):
    print( "  1 2 3" )
    for rij in range( 3 ):
        print( rij+1, end=" ")
        for kol in range( 3 ):
            print( b[rij][kol], end=" " )
        print()

bord = [ ["-", "-", "-"], ["-", "-", "-"], ["-", "-", "-"] ]
bord[1][1] = "X"
bord[0][2] = "O"
toon_bord( bord )
```

12.7 List casting

Je kunt een collectie van elementen tot een list maken door de type casting functie **list()** te gebruiken. De code hieronder maakt van een tuple een list.

```
t1 = ( "appel", "banaan", "kers" )
print( t1 )
print( type( t1 ) )
fruitlist = list( t1 )
print( fruitlist )
print( type( fruitlist ) )
```

Dit is soms nodig, specifiek wanneer je een "iterator" hebt en je wilt de elementen van de iterator als een list gebruiken. Een iterator is een functie die een collectie van elementen één voor één creëert (meer over iteratoren volgt in hoofdstuk 23). Een voorbeeld van een iterator dat ik al genoemd heb is de **range()** functie. **range()** genereert een collectie getallen. Als je deze getallen wilt gebruiken in de vorm van een list, moet je list casting doen.

```
numlist = range( 1, 11 )
print( numlist )
numlist = list( range( 1, 11 ) )
print( numlist )
```

Je kunt een string ook casten als list. Dan krijg je een list waarin ieder teken in de string een element is.

12.8 List comprehensions

Een "list comprehension" (helaas is er geen geschikte Nederlandse vertaling voor deze term) is een techniek waarbij je op een compacte manier lists creëert. Ze zijn typisch voor Python, en je vindt ze zelden in andere programmeertalen. Je hebt ze niet echt nodig, omdat je via functies exact hetzelfde effect kunt bereiken. Ze worden echter vaak gebruikt in voorbeelden (vooral door mensen die willen laten zien hoe goed ze Python beheersen door met een kort statement een uitgebreid effect te ressorteren), dus het leek me verstandig om ze toch aan de orde te stellen. Als je ze nooit gaat gebruiken in je eigen code, is dat prima want list comprehensions zijn volledig optioneel. Maar je moet ze kunnen herkennen in andermans code.

Een list comprehension bestaat uit een expressie tussen vierkante haken, bestaande uit een **for** statement, gevolgd door nul of meer **for** en/of **if** statements. Het resultaat is een list die de elementen bevat die het resultaat zijn van de evaluatie van de combinatie van de **for** en **if** statements. Dit klinkt ingewikkeld (en dat is het eigenlijk ook wel), maar een voorbeeld zal veel duidelijk maken:

Stel je voor dat ik een list wil creëren die de kwadraten van de getallen 1 tot en met 25 bevat. Ik kan dat met een functie als volgt doen:

```
def kwadraatlist():
    k = []
```

```
    for i in range( 1, 26 ):
        k.append( i*i )
    return k

sl = kwadraatlist()
print( sl )
```

In Python kan ik hetzelfde effect bereiken met één regel code, namelijk als volgt:

```
sl = [ x*x for x in range( 1, 26 ) ]
print( sl )
```

Ik neem aan dat deze code min of meer voor zichzelf spreekt. Stel je nu voor dat je deze list wilt creëren, maar dat je om een of andere reden geen kwadraten wilt opnemen van getallen die op een 5 eindigen. Als ik dat in bovenstaande functie wil doen, heb ik minstens twee regels code meer nodig, maar met een list comprehension kan het nog steeds in één en hetzelfde statement:

```
sl = [ x*x for x in range( 1, 26 ) if x%10 != 5]
print( sl )
```

De resultaten kunnen behoorlijk complex zijn. Bijvoorbeeld, de volgende list comprehension creëert een list van tuples van drie integers tussen 1 en 4, waarbij de drie integers alledrie verschillend zijn:

listing1217.py

```
triolist = [ (x,y,z) for x in range( 1, 5 )
    for y in range( 1, 5 ) for z in range( 1, 5 )
    if x != y if x != z if y != z]
print( triolist )
```

Als je list comprehensions maar lastig vindt, onthoudt dan dat er absoluut geen reden is om ze te gebruiken behalve om je code compact te houden, en dat het belangrijker is je code leesbaar en begrijpbaar te houden.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Lists
- Veranderbaarheid van lists
- Gebruik van $+$ en $*$ met lists
- List methodes `append()`, `extend()`, `insert()`, `remove()`, `pop()`, `index()`, `count()`, `sort()`, en `reverse()`
- **`del`** met lists

- Aliases
- Het gereserveerde woord **`is`**
- List kopieën
- Diepe kopieën van lists via `deepcopy()`
- Lists als argumenten
- Nesten van lists
- List casting
- List comprehensions

Opgaves

Opgave 12.1 Een magische bol (in Amerika is dit de "magic 8-ball") geeft een toevalsantwoord op een vraag. De code hieronder bevat een list van mogelijke antwoorden. Maak een magische-bol programma dat een toevalsantwoord geeft op iedere willekeurige vraag.

exercise1201.py

```
antwoord = [ "Dat is zeker", "Het is zeker zo", "Zonder twijfel",
"Ja, zeker", "Je kunt erop vertrouwen", "Zoals ik het zie, ja",
"Waarschijnlijk", "Ziet er goed uit", "Ja", "Lijkt van wel",
"Vaag, probeer het nog eens", "Vraag later nog eens", "Kan ik \
beter niet zeggen", "Kan ik nu niet voorspellen", "Concentreer \
je en vraag nog eens", "Reken er maar niet op", "Ik zeg van \
niet", "Mijn bronnen zeggen van niet", "Lijkt er niet op",
"Zeer twijfelachtig" ]
```

Opgave 12.2 Een speelkaart heeft een kleur (`"Harten"`, `"Schoppen"`, `"Klaveren"`, `"Ruiten"`) en een waarde (`2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `"Boer"`, `"Vrouw"`, `"Heer"`, `"Aas"`). Maak een list met alle mogelijke speelkaarten, wat ook wel een stok wordt genoemd. Schrijf dan een functie die de stok schudt, en dus een toevallige volgorde van de kaarten veroorzaakt.

Opgave 12.3 Een "first-in-first-out" (FIFO) structuur, ook wel "queue" geheten, is een list waarbij steeds nieuwe elementen aan het einde worden toegevoegd, terwijl elementen vanaf het begin van de list verwijderd en verwerkt worden. Schrijf een programma dat een queue verwerkt. Het programma bestaat uit een loop. In de loop wordt de gebruiker om input gevraagd. Als de gebruiker alleen op de `Enter` toets drukt, eindigt het programma. Als de gebruiker iets anders intoetst, behalve als het een enkel vraagteken (?) is, voegt het programma hetgeen de gebruiker heeft ingevoerd als nieuw element aan het einde van de queue toe. Als de gebruik een enkel vraagteken ingeeft, "popt" het programma het eerste element van de queue en toont het. Houd er rekening mee dat de gebruiker een vraagteken kan ingeven terwijl de queue leeg is.

Opgave 12.4 Tel hoe vaak iedere letter voorkomt in een string (zonder verschil te maken tussen hoofd- en kleine letters). Je mag ieder teken dat geen letter is negeren. Print de letters met het aantal malen dat ze voorkomen, waarbij je de letters sorteert van veel voorkomend naar weinig voorkomend.

Opgave 12.5 De zeef van Eratosthenes is a methode om alle priemgetallen te vinden tussen 1 en een gegeven getal, gebruik makend van een list. Dit werkt als volgt. Je begint met een list te maken die bestaat uit de getallen 1 tot en met een zeker "hoogste getal." Zet de waarde van 1 op nul, omdat 1 geen priemgetal is. Verwerk nu de list in een loop. Zoek naar het eerste nummer dat niet op 0 staat, wat nummer 2 is. Dat betekent dat 2 een priemgetal is, maar alle veelvouden van 2 zijn dat niet. Dus zet alle veelvouden van 2 op 0. Zoek dan naar het volgende nummer dat geen nul is, en dat is 3. Zet alle veelvouden van 3 op nul. Zoek dan naar het volgende nummer dat geen nul is, en dat is 5. Zet alle veelvouden van 5 op nul. Verwerk zo de hele list. Als je klaar bent, zijn alleen nog de getallen over die priemgetallen zijn. Gebruik deze methode om alle priemgetallen tussen 1 en 100 te bepalen.

Opgave 12.6 Schrijf een boter-kaas-eieren programma dat twee mensen het spel samen laat spelen. Om de beurt vraagt het programma iedere speler om de rij en de kolom waar ze een teken willen plaatsen. Zorg ervoor dat het programma alleen een rij/kolom combinatie toestaat die binnen het bord valt en leeg is. Als een speler heeft gewonnen, eindigt het spel. Als het bord vol is, eindigt het spel ook, met een gelijkspel.

Dit is een redelijk lang programma om te schrijven (60 regels code of zo). Gebruik maken van functies helpt. Ik raad je aan een functie `toon_bord()` te schrijven die het bord als parameter krijgt en die het laat zien. Maak ook een functie `neemRijKolom()` die de gebruiker om een rij/kolom combinatie vraagt en die controleert of het een legale invoer betreft. Maak ook een functie `winnaar()` die controleert of het bord een winnaar heeft. Houd bij wie aan de beurt is middels een variabele `speler` in het hoofdprogramma, die je kunt meegeven aan een functie als argument als de functie dit moet weten. Ikzelf bouw ook altijd een functie `opponent()` die de speler als argument krijgt en de andere speler teruggeeft; een dergelijke functie kan gemakkelijk gebruikt worden om van speler te wisselen na een zet.

Het hoofdprogramma zal er ongeveer als volgt uitzien (in pseudo-code):

```
toon bord
while True:
    vraag om de rij
    vraag om de kolom
    if de rij/kolom combinatie al bezet is:
        geef een foutboodschap
        continue
    plaats een markering voor de speler op de rij/kolom
    toon bord
    if er is een winnaar:
        feliciteer winnaar
        break
    if bord is vol:
        zeg dat het gelijkspel is
        break
    wissel spelers
```

Opgave 12.7 Maak een programma dat een vereenvoudigde versie van het spel "Zeeslagje" speelt. De computer creëert (in het geheugen) een matrix van 3 rijen hoog en 4 kolommen breed. De rijen zijn genummerd 1, 2, en 3, en de kolommen hebben de letters A, B, C, en D. De computer verstopt in drie van de cellen een "oorlogsschip." Ieder schip is precies één cel groot. De schepen mogen elkaar noch horizontaal, noch verticaal raken. Laat het programma de schepen per toeval plaatsen, dus niet volgens een vastgestelde configuratie.

De computer vraagt de speler te "schieten" op cellen in de matrix. De speler doet dat door een kolom letter en rij cijfer in te geven (bijvoorbeeld, `"D3"`). Als de cel waarop de speler schiet niks bevat, zegt de computer "Mis!" Als de cel een schip bevat, zegt de computer "Raak!' en verwijdert het schip (dus als de speler nog eens zou schieten op dezelfde cel dan is het automatisch een mis). Als de speler erin geslaagd is alle drie de schepen tot zinken te brengen, laat de computer zien hoeveel schoten er nodig waren, en het programma eindigt.

Om te helpen bij het debuggen van het spel, laat je de computer bij de start de matrix tonen waarbij je kunt zien welke cellen een schip bevatten.

Hint: Als je dit een lastige oefening vindt, start dan met een bord waarbij je de schepen al vooraf geplaatst hebt. Als de rest van de code werkt (en dit is niet erg moeilijk na de vorige opgave), voeg dan een functie toe waarbij de schepen per toeval geplaatst worden, zonder dat je controleert of ze elkaar raken. Als dat eenmaal werkt, voeg je code toe die ervoor zorgt dat de schepen elkaar niet kunnen raken.

Opgave 12.8 Het "subset som" probleem stelt de vraag of een bepaalde verzameling van integers een deelverzameling van integers bevat die, als ze worden opgeteld, nul als antwoord geven. Bijvoorbeeld, als de verzameling is opgeslagen als een list, dan is het antwoord bij de list `[1, 4, -3, -5, 7]` "ja," aangezien $1 + 4 - 5 = 0$. Echter, voor de list `[1, 4, -3, 7]` is het antwoord "nee," aangezien er geen deelverzameling van de integers is die optellen tot nul. Schrijf een programma dat het subset som probleem oplost voor een list met integers. Als er een oplossing is, druk die af; als er geen oplossing is, geef dat dan aan.

Hint: Dit probleem pak je het beste aan met recursie. Als je hoofdstuk 9 hebt overgeslagen, kun je het beste ook deze opgave overslaan.

Hoofdstuk 13

Dictionaries

Strings, tuples en lists zijn geordende data structuren, wat inhoudt dat ze via indices benaderd kunnen worden. Maar niet alle data verzamelingn hebben een natuurlijke manier van numeriek geordend zijn, en deze kunnen dus niet (gemakkelijk) geïndiceerd worden. Python biedt "dictionaries" als een manier om ongeordende data te structureren.

13.1 Dictionary basis

Een "dictionary" (letterlijk: "woordenboek," maar die vergelijking loopt spaak in Python) is een ongeordende data structuur die een verzameling elementen bevat. Om een element te vinden, moet je de "key" ("sleutel") van het element kennen.

In de grond is een dictionary een verzameling van "keys" met geassocieerde waardes. Ieder onveranderbaar data type mag gebruikt worden als key. Een veelgebruikt data type dat als key wordt ingezet is de string.

Dictionaries creëer middels accolades `{}`, vergelijkbaar met hoe je lists creëert met vierkante haken. Een lege dictionary maak je door een assignment aan een variabele te doen met `{}`. Je kunt een dictionary met inhoud creëren door ieder element dat je erin wilt hebben tussen de accolades te zetten, met als syntax `<key>:<value>`, en komma's tussen de elementen.

Hieronder bouw ik een dictionary fruitmand, met drie elementen, namelijk de key `"appel"` met waarde 3, de key `"banaan"` met waarde 5, en de key `"kers"` met waarde 50.

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
```

Om een waarde te vinden die hoort bij een specifieke sleutel, gebruik je dezelfde syntax als voor een list, behalve dat waar je bij een list de index schrijft, je bij een dictionary de gezochte key schrijft.

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
print( fruitmand["banaan"] )
```

Je kunt via een **for** loop een dictionary doorlopen. De variabele in de **for** loop krijgt de waardes van de keys.

listing1301.py

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
for key in fruitmand:
    print( "{}:{}".format( key, fruitmand[key] ))
```

Als je een dictionary element probeert te benaderen met een key die niet voorkomt in de dictionary, krijg je een runtime error. Maar als je een nieuw element wilt toevoegen, kun je dat eenvoudigweg doen door een waarde toe te kennen aan een dictionary element met de nieuwe key. Bijvoorbeeld, om een `"mango"` toe te voegen aan de `fruitmand`, doe je het volgende:

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
print( fruitmand )
fruitmand["mango"] = 1
print( fruitmand )
```

Op dezelfde wijze kun je een bestaand dictionary element overschrijven.

Om een element te verwijderen uit een dictionary, gebruik je het gereserveerde woord **del**, net zoals je het gebruikt met lists.

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
print( fruitmand )
del fruitmand["banaan"]
print( fruitmand )
```

Je kunt het aantal elementen in de dictionary bepalen met de **len**() functie.

Snap je overigens in welke volgorde de dictionary de elementen aanbiedt als je de inhoud van de dictionary print? Denk er even over na.

Het antwoord is: de volgorde is willekeurig. Ik zei dat bij het begin van het hoofdstuk: dictionaries zijn ongeordend. Ik kan je niet eens zeggen wat voor volgorde je op je scherm ziet als een de code hierboven uitvoert, want de volgorde kan verschillen tussen computers, besturingssystemen, en versies van Python. Er is een zekere structuur in de ordening, maar niet een die je zou kunnen (of willen) voorspellen. Door voldoende nieuwe elementen toe te voegen, kan de volgorde zelfs plotseling drastisch wijzigen.

Omdat dictionaries ongeordend zijn, zijn vele concepten die gelden voor lists, niet van toepassing op dictionaries. Bijvoorbeeld, je kunt geen “subdictionary” maken door een key-bereik te definiëren, je kunt kunt een dictionary niet “sorteren” of “inverteren.” Dictionaries zijn daarom wat beperkt, maar ze kunnen nuttig zijn.

13.2 Dictionary methodes

Hieronder staan de dictionary methodes die het meest gebruikt worden.

13.2.1 copy()

Net als met lists, kun je een variabele die een dictionary bevat via de assignment operator aan een andere variabele koppelen, en daarmee maak je dan een alias voor de dictionary (als je je dat niet herinnert, moet je er hoofdstuk 12 nog eens op naslaan). Het is niet mogelijk dezelfde "truuk" die bij lists bestaat te gebruiken om een kopie te maken. In plaats daarvan hebben dictionaries een methode `copy()` die een kopie van de dictionary maakt en retourneert.

listing1302.py

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
fruitmandalias = fruitmand
fruitmandcopy = fruitmand.copy()

print( id( fruitmand ) )
print( id( fruitmandalias ) )
print( id( fruitmandcopy ) )
```

Merk op dat een dergelijke kopie een ondiepe kopie is (zie hoofdstuk 12 als je je het verschil tussen ondiepe en diepe kopieën niet herinnert). Als je een diepe kopie wilt maken, moet je de `deepcopy()` functie van de `copy` module gebruiken.

13.2.2 keys(), values(), and items()

De methode `keys()` levert een iterator die alle keys van de dictionary genereert. De methode `values()` levert een iterator die alle waardes van een dictionary genereert. De methode `items()` levert een iterator die 2-tuples genereert die alle keys en waardes van de dictionary bevatten.

Ik zeg specifiek dat deze methodes iterators retourneren en niet lists. Als je wat ze retourneren in lists wil veranderen, moet je list casting gebruiken (zie hoofdstuk 12).

```
fruitmand = { "appel":3, "banaan":5, "kers":50 }
print( list( fruitmand.keys() ) )
print( list( fruitmand.values() ) )
print( list( fruitmand.items() ) )
```

Op dit punt vraag je je wellicht af wanneer je een iterator kunt gebruiken. Je gebruikt iterators met name in **for** loops (maar je kunt ze ook gebruiken als argumenten voor de functies **max**(), **min**() en **sum**()).

```
fruitmand = {"appel":3,"banaan":5,"kers":50,"druif":0,"mango":2}
for key in fruitmand.keys():
    print( "{}:{}".format( key, fruitmand[key] ) )
print( sum( fruitmand.values() ) )
```

Omdat deze code een onvoorspelbare volgorde voor de keys doorloopt, wil je ze meestal sorteren voordat je ze in de loop verwerkt. Omdat `keys()` geen list maar een iterator levert,

kun je ze niet direct sorteren, maar moet je het resultaat van `keys()` eerst met een list casting in een list omzetten. Daarna kun je de list sorteren.

listing1303.py

```
fruitmand = {"appel":3,"banaan":5,"kers":50,"druif":0,"mango":2}
keylist = list( fruitmand.keys() )
keylist.sort()
for key in keylist:
    print( "{}:{}".format( key, fruitmand[key] ) )
```

Je kunt niet direct de `sort()` methode toepassen op de list casting, met andere woorden, `keylist = list( fruitmand.key() ).sort()` werkt niet. Je moet eerst de list creëren, en dan pas sorteren. Je kunt ook niet schrijven `for key in keylist.sort()`, omdat de `sort()` methode geen retourwaarde heeft.

Als je je afvraagt waarom Python iterators boven lists prefereert: het antwoord daarop is dat iterators meer generiek bruikbaar zijn en minder geheugen gebruiken. Het zijn "luie" methodes, omdat ze alleen een item produceren als erom gevraagd wordt.

13.2.3 `get()`

De `get()` methode kun je gebruiken om een waarde uit de dictionary te halen zelfs als je niet weet of de key die je zoekt wel in de dictionary zit. Je roept de `get()` methode aan met de key die je zoekt als argument, en het geeft de corresponderende waarde terug als de key bestaat, of de speciale waarde **`None`** als de key niet bestaat in de dictionary. Als je in plaats van **`None`** een andere waarde terug wilt krijgen als de key niet bestaat, dan kun je die waarde meegeven als het tweede, optionele argument.

listing1304.py

```
fruitmand = {"appel":3,"banaan":5,"kers":50,"druif":0,"mango":2}

appel = fruitmand.get( "appel" )
if appel:
    print( "appel is in de mand" )
else:
    print( "geen appels in de mand")

aardbei = fruitmand.get( "aardbei" )
if aardbei:
    print( "aardbei is in de mand" )
else:
    print( "geen aardbei in de mand")

banaan = fruitmand.get( "banaan", 0 )
print( "aantal bananen in de mand:", banaan )

aardbei = fruitmand.get( "aardbei", 0 )
print( "aantal aarbeien in de mand:", aardbei )
```

Voer de code hierboven uit en bestudeer de uitkomst, omdat wat de code demonstreert over de `get()` methode erg nuttig is. Stel je voor dat een collectie van items hebt met corresponderende hoeveelheden, bijvoorbeeld, de inhoud van een fruitmand waarbij de keys de namen van fruit zijn, en de waardes de hoeveelheden. Als je in de `fruitmand` dictionary zoekt met de `get()` methode en als tweede argument een nul, kun je naar een willekeurig stuk fruit zoeken in de mand zonder dat je eerst moet controleren of het bestaat in de mand, want als je naar een fruitnaam vraagt die er niet als key in voorkomt, krijg je nul terug, en dat is precies wat je wilt zien.

13.2.4 Oefening

Opgave De code hieronder bevat een list met woorden. Bouw een dictionary die al deze woorden als key heeft, en als waarde hoe vaak het woord voorkomt in de list. Print daarna de woorden met hun hoeveelheden.

listing1305.py

```
woordlist = ["appel","doerian","banaan","doerian","appel","kers",
    "kers","mango","appel","appel","kers","doerian","banaan",
    "appel","appel","appel","appel","banaan","appel"]
```

Opgave De code hieronder bevat een string met woorden gescheiden door komma's. Bouw een dictionary die al deze woorden als key heeft, en als waarde hoe vaak het woord voorkomt in de list. Print daarna de woorden met hun hoeveelheden.

listing1306.py

```
tekst = "appel,doerian,banaan,doerian,appel,kers,kers,mango," + \
    "appel,appel,kers,doerian,banaan,appel,appel,appel," + \
    "appel,banaan,appel"
```

Opgave De code hieronder bevat een kleine dictionary die vertalingen bevat van Engelse woorden naar Nederlandse. Schrijf een programma dat met behulp van deze dictionary een woord-voor-woord vertaling maakt van de tekst eronder. Een woord dat niet in de dictionary voorkomt, vertaal je niet. De dictionary gebruikt alleen kleine letters, en je kunt de hele tekst naar kleine letters omzetten voordat je de vertaling maakt. Het is aardig als je interpunctie in stand houdt in de vertaling, maar je mag die ook weglaten (het is best veel werk om de interpunctie intact te houden en het heeft niks van doen met dictionaries, dus het is niet belangrijk – het is ook veel gemakkelijker om dit te doen als je eenmaal geleerd hebt om te gaan met reguliere expressies, in hoofdstuk 25). Ik besef heel goed dat de vertaling uitermate slecht is, maar daar gaat het niet om.

listing1307.py

```
engels_nederlands = { "last":"laatst", "week":"week", "the":"de",
    "royal":"koninklijk", "festival":"feest", "hall":"hal",
    "saw":"zaag", "first":"eerst", "performance":"optreden",
    "of":"van", "a":"een", "new":"nieuw", "symphony":"symphonie",
```

```
    "by":"bij", "one":"een", "world":"wereld", "leading":
    "leidend", "modern":"modern", "composer":"componist",
    "composers":"componisten", "two":"twee", "shed":"schuur",
    "sheds":"schuren" }

zin = "Last week The Royal Festival Hall saw the first \
performance of a new symphony by one of the world's leading \
modern composers, Arthur \"Two-Sheds\" Jackson."
```

13.3 Keys

Zoals ik aangaf kan ieder onveranderbaar data type een dictionary key zijn. Dat betekent dat je strings, integers, en floats kunt gebruiken als keys. Je herinnert je wellicht dat tuples ook onveranderbaar zijn, wat betekent dat ook tuples als keys gebruikt kunnen worden. Dat is soms nuttig.

Een eenvoudig voorbeeld van het nut van tuples als keys is een dictionary waarbij je informatie wilt opslaan die geassocieerd is met punten in een 2-dimensionale ruimte (ik besprak dit in hoofdstuk 11). Er is geen goede manier waarmee je een 2-dimensionaal punt kunt opslaan als een enkel getal of string. Het is niet onmogelijk (je kunt bijvoorbeeld het getallenpaar omzetten naar hun string-representatie en ze met een komma ertussen tot één string maken) maar het wordt al snel verwarrend (bijvoorbeeld, de strings `"2,3"`, `"2, 3"`, `"+2,+3"`, en `"02,03"` zouden alle dezelfde tuple representeren, terwijl het verschillende keys zijn).

13.4 Opslaan van complexe waardes

Tot op dit moment heb ik alleen gesproken over het opslaan bij een key in een dictionary van een enkele waarde van een enkel data type. Het is echter ook mogelijk om complexe waardes op te slaan in een dictionary. De waardes kunnen willekeurige Python objecten zijn. Bijvoorbeeld, je kunt bij iedere key een list opslaan. Hieronder staat een dictionary waarbij ik studenten opsla die een cursus volgen. De cursus wordt geïdentificeerd door het cursusnummer. De studenten worden geïdentificeerd door hun studentnummers.

listing1308.py

```
curses = {
    '880254':['u123456', 'u383213', 'u234178'],
    '822177':['u123456', 'u223416', 'u234178'],
    '822164':['u123456', 'u223416', 'u383213', 'u234178']}

for c in curses:
    print( c )
    for s in curses[c]:
        print( s, end=" " )
    print()
```

Stel je voor dat ik niet alleen de studentnummers per cursus wil opslaan, maar ook de cursusnaam, de cursus ECTS (ECTS is een standaard voor studiepunten), en voor iedere student een eindcijfer. Je kunt dat (bijvoorbeeld) doen door als waarde bij een cursusnummer een dictionary op te nemen, met drie keys: "naam", "ects", en "studenten". De waarde bij "naam" is de cursusnaam als een string, de waarde voor "ects" is een integer, en de waarde voor "studenten" is een andere dictionary, die studentnummers als keys gebruikt en eindcijfers als waardes.

listing1309.py

```
curses = {
    '880254':
    { "naam":"Onderzoeksvaardigheden Data Processing", "ects":3,
        "studenten":{'u123456':8, 'u383213':7.5, 'u234178':6} },
    '822177':
    { "naam":"Logica", "ects":6,
        "studenten":{'u123456':5, 'u223416':7, 'u234178':9} },
    '822164':
    { "naam":"Computer Games", "ects":6,
        "studenten":{'u123456':7.5, 'u223416':9 } } }

for c in curses:
    print( "{}: {} ({})".format( c, curses[c]["naam"],
        curses[c]["ects"] ) )
    for s in curses[c]["studenten"]:
        print( "{}: {}".format( s, curses[c]["studenten"][s] ) )
    print()
```

Data structuren kunnen nog complexer worden dan dit als je dat wilt. Echter, als je inderdaad overweegt om Python programma's te maken voor dit soort data structuren, doe je er goed aan om eerst object oriëntatie te bestuderen (hoofdstuk 20 en verder) en waarschijnlijk een aparte cursus over databases te volgen.

13.5 Snelheid

Lists en dictionaries zijn de twee meest-gebruikte data structuren in Python. Het is vaak duidelijk welk van de twee je moet gebruiken bij een probleem, maar het kan nuttig zijn om iets te weten over hoe Python deze data structuren verwerkt voor het geval je een keus hebt.

Stel je voor dat je een groot aantal getallen uit een bestand moet lezen. De getallen zijn allemaal verschillend en kunnen willekeurig groot zijn. Je moet later getallen van een andere lijst vergelijken met de getallen die je uit het bestand hebt gelezen.

Moet je een list of een dictionary gebruiken om de getallen die je inleest op te slaan? Omdat het alleen maar getallen zijn en geen extra data, lijkt een list de beste optie. Er is echter een probleem dat optreedt als je hier een list gebruikt. Bekijk de volgende code, die een list creëert met 10000 getallen, en daarna bekijkt of 10000 andere getallen in de list voorkomen (wat geldt voor geen van de getallen).

listing1310.py

```
from datetime import datetime

numlist = []
for i in range( 10000 ):
    numlist.append( i )

start = datetime.now()
teller = 0
for i in range( 10000, 20000 ):
    if i in numlist:
        teller += 1
eind = datetime.now()

print( "{}.{} seconden om {} nummers te vinden".format(
    (eind-start).seconds, (eind-start).microseconds, teller ) )
```

Hier is code die hetzelfde doet, maar de getallen inleest in een dictionary, waarbij simpelweg voor ieder ingelezen getal een waarde van 1 in de dictionary wordt opslagen.

listing1311.py

```
from datetime import datetime

numdict = {}
for i in range( 10000 ):
    numdict[i] = 1

start = datetime.now()
teller = 0
for i in range( 10000, 20000 ):
    if i in numdict:
        teller += 1
eind = datetime.now()

print( "{}.{} seconden om {} nummers te vinden".format(
    (eind-start).seconds, (eind-start).microseconds, teller ) )
```

Als je deze code uitvoert, zul je zien dat voor de dictionary het antwoord vrijwel onmiddellijk volgt, maar dat het voor een list wat langer duurt. De reden is dat ik met behulp van de **in** operator controleer of een getal voorkomt in de list of de dictionary. Voor een list betekent dat dat Python de hele list sequentieel doorzoekt, totdat het het getal vindt of het einde van de list bereikt wordt. Dit houdt in dat Python 10000 keer 10000 getallen controleert (omdat het geen enkel getal kan vinden), ofwel 100 miljoen getallen.

Voor een dictionary is het zoeken van een key veel sneller. Python kan erg snel vaststellen of een key wel of niet in een dictionary zit.[18] Meestal is het controleren van een handjevol getallen genoeg. Daarom is de dictionary code veel, veel sneller.

[18]Python slaat de keys voor een dictionary op in een zogeheten "hash tabel." Ik leg de details daarvan niet uit, maar weet dat dit het mogelijk maakt om keys heel snel op te zoeken, ten koste van wat geheugengebruik.

Je denkt misschien dat een paar seconden zoektijd voor de list nog steeds erg weinig is, maar de zoektijd neemt kwadratisch toe met de hoeveelheid data. Afhankelijk van het soort en de omvang van het probleem, kan een dictionary zeer te prefereren zijn boven een list.

Lists nemen wel minder geheugen in beslag dan dictionaries, en als je een list direct kunt benaderen via een index, kunnen lists zeker sneller zijn dan dictionaries. Bijvoorbeeld, in bovenstaande probleem, als ik zou weten dat de list gesorteerd is dan kan ik getallen op een veel slimmere manier vinden dan via de **`in`** operator (ongeveer 14 getallen controleren volstaat) – in dat geval is het gebruik van een list misschien sneller dan het gebruik van een dictionary.

Onthoud hiervan dat een list snel is als je de elementen via hun index kunt benaderen, terwijl een dictionary een betere keuze is als de enige manier om iets te zoeken is de elementen te scannen. De **`in`** operator lijkt gemakkelijk en is leesbaar, maar als je hem gebruikt om iets te zoeken in een lange list, dan ben je verkeerd bezig.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Dictionaries
- Dictionary keys en waardes
- Dictionary methodes `copy()`, `keys()`, `values()`, `items()`, en `get()`
- Complexe dictionaries
- Snelheid verschillen tussen lists en dictionaries

Excercises

Opgave 13.1 Schrijf een programma dat een tekst neemt (bijvoorbeeld de tekst hieronder), de tekst splitst in woorden (waarbij alles dat geen letter is beschouwd wordt als een woord-scheider), en een dictionary bouwt die voor ieder woord (case-insensitive) opslaat hoe vaak het woord voorkomt in de tekst. Print dan alle woorden met hun hoeveelheden in alfabetische volgorde.

exercise1301.py

```
tekst = """Kapper Knap, de knappe kapper, knipt en kapt heel
knap, maar de knecht van kapper Knap, de knappe kapper, knipt
en kapt nog knapper dan kapper Knap, de knappe kapper."""
```

Opgave 13.2 De code hieronder bevat een list van films. Voor iedere film is er ook een list met scores. Verander deze code zo dat het alle data opslaat in één dictionary, en gebruik dan de dictionary om de gemiddelde score voor iedere film af te drukken, afgerond op één decimaal.

exercise1302.py

```
films = ["Monty Python and the Holy Grail",
         "Monty Python's Life of Brian",
         "Monty Python's Meaning of Life",
         "And Now For Something Completely Different"]

grail_scores = [ 9, 10, 9.5, 8.5, 3, 7.5,8 ]
brian_scores = [ 10, 10, 0, 9, 1, 8, 7.5, 8, 6, 9 ]
life_scores = [ 7, 6, 5 ]
different_scores = [ 6, 5, 6, 6 ]
```

Opgave 13.3 Een bibliotheek heeft boeken. Ieder boek heeft een schrijver, die je kunt identificeren door achter- en voornaam. Boeken hebben een titel. Boeken hebben ook een locatienummer dat aangeeft waar ze staan in de bibliotheek. De bibliothecaris wil kunnen vinden waar een boek staat als hij de schrijver en titel kent, en wil ook alle boeken kunnen afdrukken die van een bepaalde schrijver zijn. Welke data structuur kun je gebruiken om de boeken in op te slaan?

Hoofdstuk 14

Sets

Sets zijn ongeordende data structuren die alleen unieke elementen kunnen bevatten. Slechts weinig programmeertalen ondersteunen het gebruik van sets, maar Python doet het wel. Sets worden niet vaak gebruikt, maar kunnen soms een leuke oplossing geven voor een probleem.

14.1 Basis van sets

Een set is een ongeordende collectie van elementen. Je kunt geen specifieke elementen van een set benaderen via een index of een key. De enige manier om elementen van een set te benaderen is via een **`for`** loop, of door met de **`in`** operator te testen of een element in de set bestaat.

Je moet bij sets denken aan wiskundige verzamelingen. In de wiskunde is een verzameling een collectie van elementen die alle uniek zijn, en ieder element zit ofwel in de verzameling, ofwel niet in de verzameling. Er zijn bepaalde operatoren die toestaan verzamelingen te combineren. Ik gebruik vanaf dit punt het woord "set" in plaats van "verzameling," omdat **`set`** een data type is.

Python gebruikt dictionaries om sets te implementeren; specifiek implementeert het de elementen van een set als keys van een dictionary. Dat betekent dat alleen onveranderbare data types in een set kunnen worden opgeslagen. Sets zelf zijn echter wel veranderbaar.

Omdat Python dictionaries gebruikt om sets te implementeren, denk je misschien dat je een lege set kunt creëren door `{}` toe te kennen aan een variabele. Dat creëert echter een lege dictionary, en niet een lege set. In plaats daarvan creëer je een lege set door de retourwaarde van de functie **`set()`** (zonder argumenten) toe te kennen aan een variabele.

Om een set te creëren waarin al een paar elementen zitten, kun je wel die elementen tussen accolades toekennen aan een variabele. Als alternatief kun je de **`set()`** functie aanroepen met een list met de elementen als argument.

```
fruitset = { "appel", "banaan", "kers" }
print( fruitset )
```

Als je een set wilt creëren bestaande uit de verschillende letters van een string, dan kun je **set()** aanroepen met de string als argument (dubbele letters worden automatisch genegeerd).

```
helloset = set( "hello world" )
print( helloset )
```

Je kunt een **for** loop gebruiken om een set te doorlopen. De variabele van de **for** loop krijgt toegang tot alle element van de set. Er is geen manier om te bepalen in welke volgorde je de elementen te zien krijgt. Je kunt ze niet sorteren zolang ze in de set zitten. Je kunt echter wel met een list casting de set omzetten in een list, en die list dan sorteren.

listing1401.py

```
fruitset = { "appel", "banaan", "kers", "doerian", "mango" }
for element in fruitset:
    print( element )
print()

fruitlist = list( fruitset )
fruitlist.sort()
for element in fruitlist:
    print( element )
```

Met behulp van de **len()** functie kun je het aantal elementen in de set vaststellen.

14.2 Set methodes

Om de inhoud van een set te manipuleren, zijn de volgende methodes beschikbaar. Dit is geen complete lijst van set methodes, maar het zijn de meest gebruikte.

14.2.1 `add()` en `update()`

Om een nieuw element aan een set toe te voegen gebruik je de `add()` methode, met het nieuwe element als argument. Als je meerdere nieuwe elementen wilt toevoegen, kun je de `update()` methode gebruiken, met een list die de nieuwe elementen bevat als argument. Je kunt ook een tuple als argument gebruiken, of zelfs een string. Als je een string gebruikt, wordt ieder teken van de string gezien als een element.

Omdat een set alleen unieke elementen kan bevatten, worden duplicaten genegeerd.

listing1402.py

```
fruitset = { "appel", "banaan", "kers", "doerian", "mango" }
print( fruitset )

fruitset.add( "appel" )
fruitset.add( "aalbes" )
print( fruitset )
```

```
fruitset.update( ["appel","appel","appel","aardbei",
    "aardbei","appel","mango"] )
print( fruitset )
```

14.2.2 `remove()`, `discard()`, en `clear()`

Om elementen uit een set te verwijderen, gebruik je de `remove()` of `discard()` methodes. Beide krijgen het te verwijderen element als argument. Het verschil is dat `remove()` een runtime error geeft als het element niet bestaat in de set, terwijl `discard()` niet-bestaande elementen gewoon negeert.

```
fruitset = { "appel", "banaan", "kers", "doerian", "mango" }
print( fruitset )

fruitset.remove( "appel" )
print( fruitset )
```

`clear()` verwijdert alle elementen van de set in één keer.

14.2.3 `pop()`

De `pop()` methode verwijdert een element uit de set en retourneert het. Je kunt niet zelf aangeven welk element je wilt verwijderen, en je kunt ook niet voorspellen welk element het is, omdat sets ongeordend zijn.

```
fruitset = { "appel", "banaan", "kers", "doerian", "mango" }
while len( fruitset ) > 0:
    print( fruitset.pop() )
```

14.2.4 `copy()`

Net als bij lists en dictionaries geldt voor sets dat als je een bestaande set toekent aan een andere variabele, je een alias creëert. Als je een kopie wilt creëren van de set, kun je de `copy()` methode gebruiken.

14.2.5 `union()`

De vereniging (Engels: "union") van twee sets is een set die alle elementen van beide verenigde sets bevat. Je kunt de `union()` methode voor de ene set, met als argument de andere set, gebruiken om de vereniging van beide geretourneerd te krijgen. Dit verandert niets aan de twee gebruikte sets zelf. Je kunt als alternatief de speciale operator | ("pipeline") gebruiken om de vereniging te creëren. Merk op: je zou misschien denken dat je de + operator kunt gebruiken om twee sets te verenigen, maar + is niet voor sets gedefinieerd, en $*$ is evenmin gedefinieerd.

listing1403.py

```
fruit1 = { "appel", "banaan", "kers" }
fruit2 = { "banaan", "kers", "doerian" }
fruitunion = fruit1.union( fruit2 )
print( fruitunion )

fruitunion = fruit1 | fruit2
print( fruitunion )
```

14.2.6 `intersection()`

De doorsnede (Engels: "intersection") van twee sets is een set die alleen de elementen bevat die beide samenstellende sets gemeen hebben. Je kunt de `intersection()` methode gebruiken voor de ene set, met als argument de andere set, om de doorsnede van beide geretourneerd te krijgen. Dit verandert de samenstellende sets niet. Als alternatief kun je de speciale operator & ("ampersand") gebruiken om de doorsnede te creëren.

listing1404.py

```
fruit1 = { "appel", "banaan", "kers" }
fruit2 = { "banaan", "kers", "doerian" }
fruitintersection = fruit1.intersection( fruit2 )
print( fruitintersection )

fruitintersection = fruit1 & fruit2
print( fruitintersection )
```

14.2.7 `difference()`

Het verschil (Engels: "difference") van twee sets is een set die de elementen van de eerste set bevat, met daaruit verwijderd de elementen die de eerste set gemeen heeft met de tweede set. Je kunt het verschil creëren door de `difference()` methode aan te roepen voor de ene set, met als argument de set waarvan je de elementen uit de eerste set wilt verwijderen. Dit verandert de samenstellende sets niet. Als alternatief kun je de speciale operator - ("minus") gebruiken om het verschil te creëren.

listing1405.py

```
fruit1 = { "appel", "banaan", "kers" }
fruit2 = { "banaan", "kers", "doerian" }
fruitdifference = fruit1.difference( fruit2 )
print( fruitdifference )

fruitdifference = fruit1 - fruit2
print( fruitdifference )

fruitdifference = fruit2 - fruit1
print( fruitdifference )
```

14.2.8 `isdisjoint()`, `issubset()`, en `issuperset()`

The methodes `isdisjoint()`, `issubset()`, en `issuperset()` worden alle aangeroepen als methodes voor een set, met een tweede set als argument. Ze retourneren alle **`True`** of **`False`**. `isdisjoint()` retourneert **`True`** als de twee sets geen elementen gemeen hebben. `issubset()` retourneert **`True`** als alle elementen van de eerste set ook voorkomen in de set die als argument dient. `issuperset()` retourneert **`True`** als alle elementen van de set die als argument dient ook voorkomen in de eerste set.

listing1406.py

```
fruit1 = { "appel", "banaan", "kers" }
fruit2 = { "banaan", "kers" }

print( fruit1.isdisjoint( fruit2 ) )
print( fruit1.issubset( fruit2 ) )
print( fruit2.issubset( fruit1 ) )
print( fruit1.issubset( fruit1 ) )
print( fruit1.issuperset( fruit2 ) )
print( fruit2.issuperset( fruit1 ) )
print( fruit1.issuperset( fruit1 ) )
```

Een set is zowel een subset als een superset van zichzelf.

14.2.9 Oefening

Opgave Er is ook een methode `symmetric_difference()` die een set retourneert die alle elementen bevat van de vereniging van twee sets, behalve de elementen die zich bevinden in de doorsnede van de twee sets. Bijvoorbeeld, als set 1 de elementen A, B, en C bevat, en set 2 de elementen B, C, en D, dan bevat de "symmetric difference" van sets 1 en 2 alleen de elementen A en D. Kun je de `symmetric_difference()` methode implementeren via de combinatie van een aantal van bovenstaande methodes?

Opgave In hoofdstuk 7 was een oefening die je vroeg om alle letters te vinden die twee woorden gemeen hebben, waarbij iedere letter slechts eenmalig gerapporteerd wordt. Dit kun je zeer efficiënt doen met sets. Schrijf de code hiervoor.

14.3 Frozensets

Python kent als variant op het set type de **`frozenset`**. Je creëert een **`frozenset`** via de **`frozenset`**`()` functie. De elementen van een **`frozenset`** kunnen niet veranderd worden. Je creëert dus een **`frozenset`** onmiddellijk als je de **`frozenset`**`()` functie aanroept, want zodra de **`frozenset`** bestaat kun je geen elementen meer toevoegen of weghalen. Met andere woorden, **`frozenset`**s zijn onveranderbaar.

Alle reguliere set methodes werken ook op **`frozenset`**s, behalve de methodes die proberen de set te veranderen. Als je een dergelijke methode probeert aan te roepen voor een **`frozenset`** krijg je een syntax error.

```
fruit1 = frozenset( ["appel", "banaan", "kers"] )
fruit2 = frozenset( ["banaan", "kers", "doerian"] )

print( fruit1.union( fruit2 ) )
```

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Sets
- Methodes `add()`, `update()`, `remove()`, `discard()`, `clear()`, `pop()`, `copy()`, `union()`, `intersection()`, `difference()`, `isdisjoint()`, `issubset()`, en `issuperset()`
- Frozensets

Opgaves

Opgave 14.1 Een beroemd syllogisme zegt: *Alle mensen zijn sterfelijk. Socrates is een mens. Daarom is Socrates sterfelijk.* In de code hieronder zie je een aantal sets. De eerste is een set van alle dingen (ik weet dat er een paar ontbreken, maar beschouw hem maar als compleet voor deze opgave). De tweede is een set van alle mensen (aannemende dat de eerste set inderdaad compleet is). De derde bevat alles wat sterfelijk is (wederom, aannemende dat...). Maak gebruik van set operatoren en methodes om te laten zien dat inderdaad geldt dat (a) alle mensen sterfelijk zijn, (b) Socrates een mens is, en (c) Socrates sterfelijk is. Laat ook zien dat (d) er sterfelijke dingen zijn die geen mens zijn, en (e) er dingen zijn die niet sterfelijk zijn.

exercise1401.py

```
alles = { "Socrates", "Plato", "Eratosthenes", "Zeus", "Hera",
    "Athene", "Acropolis", "Kat", "Hond" }
mensen = { "Socrates", "Plato", "Eratosthenes" }
sterfelijken = { "Socrates", "Plato", "Eratosthenes", "Kat",
    "Hond" }
```

Opgave 14.2 Schrijf een programma dat drie sets van getallen tussen de 1 en 1000 produceert; de eerste set bestaat uit alle getallen die deelbaar zijn door 3, de tweede uit alle getallen die deelbaar zijn door 7, en de derde uit alle getallen die deelbaar zijn door 11. Dit is het gemakkelijkste te doen met list comprehensions, maar dat is niet noodzakelijk. Produceer vervolgens sets van alle getallen tussen die 1 en 1000 die (a) zowel door 3, 7, en 11 deelbaar zijn, (b) die deelbaar zijn door 3 en 7, maar niet door 11, en (c) die noch deelbaar zijn door 3, noch door 7, en noch door 11. De kortste oplossing heeft slechts één regel code nodig voor ieder van deze zes sets.

Hoofdstuk 15

Besturingssysteem

Tot nu toe heb ik Python programma's behandeld als functionaliteiten die los staan van alles wat zich buiten het programma bevindt. Python programma's draaien echter op een computer, en zo nu en dan moet een programma omgaan met de eigenschappen van een specifieke computer. Vanaf hoofdstuk 16 gaat dit een belangrijke rol spelen, omdat ik dan het omgaan met bestanden ga behandelen. Om met de eigenschappen van een specifieke computer te kunnen omgaan, introduceert Python een aantal functies in de `os` module, waar `os` de afkorting is voor "operating system" (besturingssysteem). Dit hoofdstuk bespreekt de belangrijkste functies uit de `os` module.

15.1 Computers en besturingssystemen

Een computer bestaat uit hardware, terwijl programma's bestaan uit software. Software gebruikt de faciliteiten die de hardware biedt. Toen computers voor het eerst geïntroduceerd werden, benaderden programmeurs de hardware direct vanuit de programma's (bijvoorbeeld, om een pixel zichtbaar te maken op het scherm, zette een programmeur een waarde in een specifiek geheugenadres dat rechtstreeks aan het scherm gekoppeld was – een praktijk die bekend stond onder de naam "poking"). Vandaag de dag is de hardware zo complex en divers dat dit niet langer een geschikte aanpak is. Laar staan het feit dat als je een programma wilt schrijven dat op meerdere computers moet kunnen draaien, je het je niet kunt veroorloven om hardware direct aan te spreken aangezien hardware verschilt tussen computers. Daarom benaderen programma's de hardware functionaliteiten via een "besturingssysteem."

Een besturingssysteem kan gezien worden als een laag tussen programma's en hardware, die de programma's allerhande hoog-niveau functies biedt om de hardware aan het werk te zetten. Typische besturingssystemen die vandaag op thuiscomputers gebruikt worden zijn Microsoft's "Windows," Apple's "Mac OS," en het open-source besturingssysteem "Linux" (hoewel er nog veel andere zijn). Ieder van deze systemen bestaat in meerdere varianten, van elkaar onderscheiden door nummers of "builds," en soms (bij Linux) door een bedrijfsnaam. Hoe dan ook, alle bieden functionaliteiten om hardware aan te sturen.

Een probleem is dat hoewel alle besturingssystemen functionaliteiten aanbieden, deze functionaliteiten niet op een consistente manier benoemd zijn, en verschillen in parame-

ter specificatie. Dit betekent dat als je een Python programma wilt schrijven dat de hardware benadert door direct met het besturingssysteem te communiceren, je programma niet kan draaien op andere besturingssystemen. Dit is waar de `os` module een oplossing biedt. De `os` module bevat functies die je kunt gebruiken om hardware te benaderen, ongeacht het besturingssysteem. De `os` module heeft daarom een verschillende implementatie voor verschillende besturingssystemen, maar je programma hoeft dat niet te weten, omdat de functies altijd dezelfde naam en dezelfde parameters hebben.

Dat betekent niet dat je helemaal niks hoeft te weten van de details van een besturingssysteem. Bijvoorbeeld, als je een bestand benadert, moet je bij Windows soms de letter toevoegen die een disk drive identificeert, die niet bestaat bij Mac OS. Een ander verschil is dat bestandstoegang en bestandsbeveiliging veel meer flexibiliteit heeft bij Linux dan bij Windows of Mac OS, waardoor je bij Linux een ander soort foutmeldingen kunt verwachten. Er zijn ook functies die weliswaar bestaan voor alle besturingssystemen, maar die slechts bij sommige een effect hebben. Desondanks is de `os` module een acceptabel compromis tussen portabiliteit en besturingssysteem-afhankelijkheid.

15.2 Command prompt

Als je werkt met een muisbestuurde "user interface" (UI), standaard bij Windows en Mac OS, en ook vaak gebruikt bij Linux, ben je in werkelijkheid aan het communiceren met een visuele representatie van het systeem, of meer specifiek: het bestandssysteem. Programma's en documenten zijn weergegeven middels "pictogrammen" (Engels: "icons"), die een naam hebben. Ze zijn gegroepeerd in "folders," die in werkelijkheid "directories" zijn van het bestandssysteem. Je kunt nieuwe folders creëren, documenten verwijderen, programma's hernoemen, beveiligingen wijzigen, etcetera. Al deze acties kun je ook uitvoeren door direct commando's te typen, in een omgeving die vaak een "command prompt" of "command shell" wordt genoemd.

Linux gebruikers zijn veelal bekend met de command shell, maar Windows en Mac gebruikers zijn zich meestal niet bewust van het bestaan ervan. Zowel Windows als Mac hebben een programma dat je toestaat om te werken in de command shell. Bij Windows heet het de "command prompt" en is het te vinden bij de "accessories" of "system tools" (of de Nederlandse varianten daarvan). Bij Mac heet het de "Terminal." Als je dat programma start, zie je zwart schermpje met een knipperende prompt. Hier kun je commando's typen die het besturingssysteem dan uitvoert.

De commando's die je kunt geven zijn afhankelijk van je systeem. Dit boek is niet bedoeld om je te leren hoe je het systeem moet gebruiken, maar ik wil in ieder geval aangeven dat je Python programma's kunt uitvoeren in de command shell door het volgende commando te geven:

```
python <programmanaam>.py
```

Als Python op je systeem gevonden kan worden, en het programma staat in de huidige directory (dat wil zeggen, de plaats in het bestandssysteem van de computer waar je je op dat moment bevindt), of als je het complete pad naar het programma hebt opgegeven, dan wordt het programma uitgevoerd. Dit kan handig zijn als je een programma hebt geschreven dat bestanden verwerkt, en je wilt een groot aantal bestanden verwerken als een "batch." Wederom, dat gaat te diep voor dit boek, maar je kunt op een punt in je carrière belanden waar dit extreem handig kan zijn.

De commando's die je kunt geven zijn zaken als "wijzig de huidige directory," "maak een nieuwe directory," "verwijder een lege directory," "produceer een lijst van alle bestanden in de directory," "verwijder een bestand," etcetera. Wederom, het is afhankelijk van het besturingssysteem wat je geacht wordt te doen om deze zaken te bewerkstelligen.

Opgave Zoek op je systeem op waar je de command shell kunt starten en voer het programma uit. Op Windows, typ "dir" om de bestanden in de huidige directory te zien. Op Mac en Linux doe je dit meestal met "ls." Nadat je dit geprobeerd hebt, kun je de command shell weer sluiten.

15.3 Bestandssysteem

Het bestandssysteem (Engels: "file system") van een computer kun je zien als een boomstructuur waarin directories en bestanden georganiseerd zijn.

Er is een "root"[19] directory, die het eerste toegangspunt is voor alle andere directories. De root directory wordt geïdentificeerd door een slash (/) of backslash (\), afhankelijk van het besturingssysteem. Bij Windows is het de backslash, bij Mac OS en Linux is het de voorwaartse slash. Windows ondersteunt tegenwoordig ook de voorwaartse slash voor dit doel. Ik beveel aan dat je de voorwaartse slash gebruikt indien mogelijk, omdat in strings de backslash gebruikt wordt om speciale symbolen aan te geven. Dus als je in een string een de backslash gebruikt om een directory scheider aan te geven, moet je een dubbele backslash gebruiken. Dit is wat verwarrend, en daarom beveel ik gebruik van de voorwaartse slash aan.

"Onder" de root-directory vind je meerdere andere directories, ieder met een naam, en meestal ook meerdere bestanden, die ook ieder weer een naam hebben. Onder iedere directory kun je weer andere directories en bestanden vinden.

Ieder besturingssysteem legt bepaalde beperkingen op aan de namen van directories en bestanden, hoewel over het algemeen de meeste tekens gebruikt mogen worden. Het is de gewoonte dat reguliere bestanden een extensie hebben, die aan het einde van de bestandsnaam wordt geplaatst, en die van de bestandsnaam gescheiden is middels een punt. De extensie geeft aan wat voor soort bestand het betreft, bijvoorbeeld, een uitvoerbaar programma (`.exe`), een plat tekstbestand (`.txt`), of een Python programma (`.py`). Het is ook de gewoonte dat directories geen extensie hebben. Maar dit is geen verplichting, en je kunt zeker bestanden aantreffen zonder, en directories met extensie. Merk op dat in een visuele omgeving de extensies van bestanden vaak verborgen zijn, maar ze zijn er wel – je ziet ze alleen niet.

Om een bestand uniek te identificeren, moet je het exacte "pad" vanaf de root naar het bestand kennen, de directories volgend. Het pad van een bestandsnaam ziet er uit als `/<directory>/<directory>/.../<bestandsnaam>`. Bij Windows kan er nog de letter van een disk drive voor het pad staan, waardoor het `<drive>:/<directory>/<directory>/.../<bestandsnaam>` wordt. Bijvoorbeeld, als je bij Windows op de "C" drive onder de root een directory "Python34" hebt, waaronder je een directory "Lib" hebt, waarin je een bestand "os.py" kunt vinden, dan is het pad voor dat bestand `C:/Python34/Lib/os.py`. Bij Windows wordt er geen verschil gemaakt tussen

[19] De "root" is de wortel van de boomstructuur, maar ik heb werkelijk nooit iemand dit "wortel" horen noemen.

hoofd- en kleine letters, dus voor dit pad hadden ook alleen kleine letters gebruikt mogen worden. Dat geldt echter niet voor alle besturingssystemen.

Als je in een bestandssysteem werkt (en je werkt eigenlijk altijd in een bestandssysteem, al realiseer je je dat misschien niet), dan is er een "huidige directory," die geïdentificeerd wordt met een punt (`.`). Als je een bestand in de huidige directory wilt benaderen, hoef je niet het hele pad te kennen; dan is het genoeg als je alleen de bestandsnaam kent. Eén directory "hoger" dan de huidige directory wordt geïdentificeerd door twee punten (`..`). Boven de root zit de root zelf.

Tenslotte is het belangrijk te weten dat de meeste besturingssystemen een manier hebben waarmee je bestanden kunt benaderen die niet in de huidige directory staan, maar waarvan je het pad niet kent. Bij Windows kun je bijvoorbeeld een `PATH` omgevingsvariabele zetten die alle directories bevat die Windows doorzoekt als een uitvoerbaar bestand niet in de huidige directory gevonden kan worden. Hoe je zo een omgevingsvariabele een waarde moet geven voert echter te ver voor dit boek.

15.4 `os` functies

De `os` module ondersteunt veel functies waarmee je het bestandssysteem kunt beïnvloeden. Ik noem er slechts een paar, omdat veel functies een beetje gevaarlijk zijn om te gebruiken (je kunt bijvoorbeeld gemakkelijk per ongeluk bestanden verwijderen die je had willen houden), en je hebt er ook niet zoveel nodig. Als je echt geïnteresseerd bent om het bestandssysteem te manipuleren, kun je zelf de vele functies van de `os` module bestuderen via de Python handleiding.

15.4.1 `getcwd()`

`getcwd()` ("cwd" staat voor "current working directory") retourneert de huidige directory als een string.

```
from os import getcwd
print( getcwd() )
```

15.4.2 `chdir()`

`chdir()` wijzigt de huidige directory. De nieuwe directory geef je mee als string argument.

```
from os import getcwd, chdir

home = getcwd()
print( home )
chdir( ".." )
print( getcwd() )
chdir( home )
print( getcwd() )
```

15.4.3 `listdir()`

`listdir()` retourneert een list die alle bestanden en directories bevat in de directory die als argument is meegegeven. De namen verschijnen in willekeurige volgorde. Ze bevatten niet het volledige pad.

```
from os import listdir

flist = listdir( "." )
for naam in flist:
print( naam )
```

15.4.4 `system()`

`system()` krijgt een string argument dat beschouwd wordt als systeemcommando, dat door Python wordt uitgevoerd in de command shell. Je kunt de functie te gebruiken om alles te doen wat door het besturingssysteem ondersteund wordt, inclusief het opstarten van andere programma's. Er zijn echter betere manieren om andere programma's te starten (zoek maar naar functies waarvan de naam begint met "exec").

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Besturingssystemen
- Command shell
- Bestandssystemen
- Functies `getcwd()`, `chdir()`, `listdir()`, en `system()` uit de `os` module

Opgaves

Opgave 15.1 Schrijf een programma dat alle bestanden en directories in de huidige directory toont, inclusief het volledige pad.

Hoofdstuk 16

Tekstbestanden

Eén van de belangrijkste toepassingen van Python voor dataverwerking is het lezen, wijzigen, en schrijven van tekstbestanden. Data wordt vaak opgeslagen in tekstbestanden, omdat dit soort bestanden gemakkelijk tussen verschillende programma's overgedragen kan worden. Er zijn meerdere standaard bestandsformaten voor tekstbestanden, zoals "comma-separated values" (CSV) bestanden. Python ondersteunt een aantal van die formaten via modules, waarvan ik sommige later zal bespreken. Dit hoofdstuk focust op het openen, lezen, schrijven, en sluiten van willekeurige bestanden, ongeacht het formaat.

16.1 Platte tekstbestanden

"Tekstbestanden" of "platte tekstbestanden" zijn bestanden waarin alle tekens bedoeld zijn om te interpreteren als reguliere tekens, zoals je ze op een toetsenbord intypt. Bijvoorbeeld, Python programma's zijn platte tekstbestanden, net als HTML bestanden. Tekstverwerkingsbestanden (zoals je bijvoorbeeld creëert met Word) zijn echter geen platte tekstbestanden, en plaatjes zijn dat ook niet. Als je wilt weten of een bestand een tekstbestand is, dan kun je het proberen te openen in een teksteditor (bijvoorbeeld IDLE). Als je dat doet en alleen leesbare tekst ziet, betreft het waarschijnlijk een tekstbestand. Anders wordt het een "binair bestand" genoemd (binaire bestanden worden besproken in hoofdstuk 18).

Tekstbestanden bestaan uit regels tekst. Aan het einde van iedere regel staat een "newline" symbool, in Python voorgesteld als het teken `"\n"`. Verschillende besturingssystemen hebben verschillende manieren om dit teken op te slaan: sommige Windows programma's slaan het op als "carriage return plus line feed" (`"\r\n"`), terwijl het bij Linux altijd wordt opgeslagen als een enkele `"\n"`. Zolang je in Python een bestand benadert als een regulier tekstbestand, zal Python de tekens die het leest converteren naar de standaard `"\n"`, en vice versa wanneer het tekens wegschrijft. Dus je hoeft je niet druk te maken om dit soort verschillen (behalve wanneer je bestanden kopieert naar andere besturingssystemen).

16.1.1 Handles en pointers

Als je met een bestand werkt in een programma, moet je dat bestand openen. Het openen van een bestand creëert een zogeheten "handle" of "file handle." Je kunt een handle zien

als een variabele die toegang biedt tot een bestand. Een handle bevat een "pointer," die een specifieke plaats in het bestand aanduidt. Die pointer wordt gebruikt als je leest uit het bestand of schrijft naar het bestand. Bijvoorbeeld, als je uit een bestand leest, begint het lezen daar waar de pointer wijst, en gedurende het lezen wordt de pointer verplaatst in het bestand.

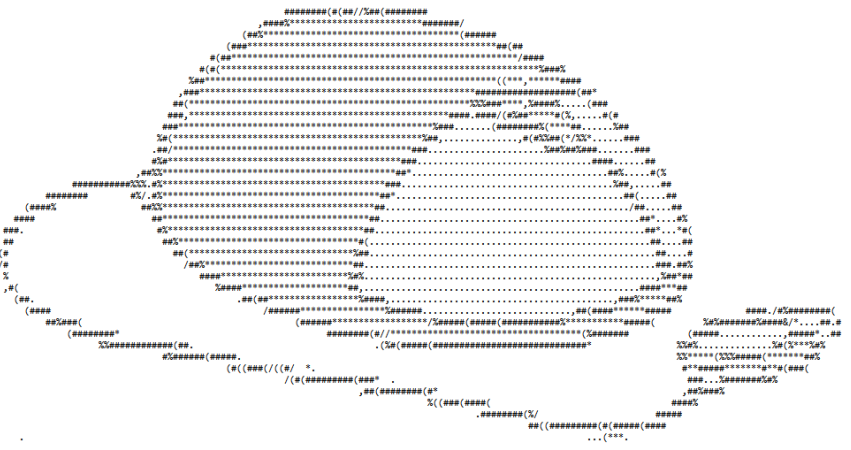
Wanneer je een bestand opent, wordt de pointer op een specifiek punt in het bestand geplaatst, afhankelijk van de manier waarop het bestand geopend is. Als je het bestand opent alleen om eruit te lezen, dan staat de pointer aan het begin. Hetzelfde geldt als je het bestand opent voor zowel lezen als schrijven. Als je een bestand opent voor "appending" (dat wil zeggen, om nieuwe data toe te voegen aan het einde van het bestand), dan staat de pointer aan het einde. Tenslotte, als je het bestand opent voor alleen schrijven, dan wordt het bestand volledig leeg gemaakt en wordt de pointer geplaatst aan het begin van het, nu lege, bestand. Om een nieuw bestand te creëren (dat wil zeggen, een bestand met een naam die nog niet bestaat), open je het bestand voor alleen schrijven.

Na het openen van het bestand is de handle het enige toegangspunt tot het bestand. Alle acties die je uitvoert op het bestand, voer je uit met behulp van methodes van de handle.

Merk op dat er een restrictie bestaat binnen het besturingssysteem op het aantal bestanden dat tegelijkertijd geopend mag worden (hoewel deze restrictie over het algemeen erg hoog ligt). Het is daarom een goed idee om bestanden te sluiten als je ze niet langer nodig hebt.

16.1.2 Verplaatsing van de pointer

De pointer, die aangeeft waar je in het bestand bezig bent, wordt automatisch verplaatst. Bijvoorbeeld, als je 10 tekens uit het bestand wilt lezen, dan geeft de pointer aan welk de eerste van die 10 tekens is. Terwijl je leest, verplaatst de pointer zich 10 tekens, zodat de nieuwe pointer positie 10 tekens verder in het bestand is dan voordat de leesactie plaatsvond. Als je met tekstbestanden werkt, is deze automatische verplaatsing van de pointer precies wat je wilt. Er zijn methodes waarmee je de pointer handmatig kunt verplaatsen, maar zulke methodes worden over het algemeen alleen gebruikt bij binaire bestanden. In dit hoofdstuk zal ik daarom deze methodes niet bespreken, maar ik breng ze aan de orde in hoofdstuk 18.



16.1.3 Buffers

Als je wijzigingen maakt in bestanden, dan worden die vaak niet onmiddellijk in de betreffende bestanden opgeslagen. In plaats daarvan slaat het besturingssysteem de wijzigingen op in een buffer in het geheugen, en maakt de wijzigingen pas definitief in het bestand als dat nodig is (een praktijk die "flushing" wordt genoemd). Je kunt het definitief maken van de wijzigingen forceren door het bestand te sluiten. Bestanden worden ook automatisch gesloten als je je programma op een normale manier beëindigt – maar het is niet zo netjes om het sluiten niet expliciet in het programma te doen.

Als je programma crasht (bijvoorbeeld vanwege een runtime error), dan kan het gebeuren dat je wijzigingen niet allemaal zijn opgeslagen, en je bestanden daarom niet up-to-date zijn tot het moment dat de crash plaatsvond. Dus als je probeert een programma te debuggen kun je er niet van uitgaan dat de inhoud van de bestanden zo is als het programma ze gemaakt zou moeten te hebben.

16.1.4 Bestandsverwerking

De meeste programma's die bestandsverwerking doen, volgen een proces dat, in een loop, de inhoud van een bestand leest, die inhoud op een of andere manier verwerkt, en tenslotte de bewerkte inhoud naar een ander bestand schrijft. Bijvoorbeeld, een programma kan regels uit een tekstbestand lezen, de woorden in iedere regel sorteren, en dan de gesorteerde woorden naar een ander bestand schrijven. Dit verschilt nauwelijks met een programma dat de gebruiker in een loop vraagt om een regel tekst, de woorden in de regel sorteert, en die woorden dan op het scherm toont met de **`print`**`()` functie. Ik heb gezien dat studenten het relatief gemakkelijk vinden om de versie van het programma te schrijven waarbij de gebruiker met het programma communiceert, maar het erg moeilijk vinden om een soortgelijk programma te schrijven dat informatie uit een bestand leest.

Ik ben er nooit achter gekomen waarom zoveel studenten bestandsverwerking zo lastig vinden, maar ik vermoed dat ze het gevoel hebben de controle over hun programma te verliezen als ze met bestanden werken. Als input handmatig verstrekt wordt, en output getoond wordt op het scherm, dan weet je min of meer welke regels code Python aan het uitvoeren is, en je kunt testen wat je wilt. Maar als je met bestanden werkt, dan moet je die bestanden van te voren klaarmaken, en dan het programma uitvoeren en wachten tot het klaar is voordat je de output bestanden kunt controleren.

Je hebt misschien het gevoel weinig controle over een bestandsverwerkend programma te hebben, maar tijdens het bouwen van het programma kun je altijd **`print`**`()` statements in je code opnemen om inzicht te krijgen in wat het programma doet. Bijvoorbeeld, wanneer je programma een regel uit een tekstbestand leest, kun je die regel printen, en als je programma een regel schrijft, kun je die ook printen. Op die manier is het inzicht dat je krijgt in de werkwijze van je programma nauwelijks anders dan wanneer je handmatige invoer aan je programma verstrekt.

16.2 Lezen van tekstbestanden

Om een tekstbestand te lezen, moet je het bestand eerst openen, dan de inhoud lezen, en tenslotte het bestand sluiten.

16.2.1 Openen van een bestand met `open()`

Om een bestand te openen, gebruik je de **`open`**`()` functie.

De **`open`**`()` functie krijgt twee argumenten, waarvan de tweede optioneel is. Het eerste argument is de naam van het bestand. Als het bestand niet in de huidige directory staat, moet je het complete pad naar het bestand opgeven zodat Python het kan vinden. Het tweede argument is de "modus." Deze geeft aan hoe Python het bestand moet behandelen. De default modus (die gebruikt wordt als er geen argument wordt opgegeven) is dat het bestand wordt geopend als tekstbestand, en er alleen uit het bestand gelezen mag worden. Ik zal later bediscussiëren hoe je andere modi opgeeft.

De **`open`**`()` functie retourneert een handle, die je gebruikt voor alle andere functionaliteiten.

In plaats van `<handle> = `**`open`**`( <bestandsnaam> )`, schrijven Python programmeurs vaak **`open`**`( <bestandsnaam> )` **`as`** `<handle>`. De tweede manier is equivalent met de eerste. Ikzelf prefereer de eerste manier, omdat deze het dichtst ligt bij de manieren waarop andere programmeertalen het openen van bestanden afhandelen. De tweede methode heeft echter een klein voordeel, dat ik zal bespreken wanneer ik het heb over het sluiten van bestanden.

16.2.2 Lezen uit een bestand met `read()`

De eenvoudigste manier om de inhoud van een bestand te lezen is via de `read()` methode, zonder argumenten, via de handle. Dit levert een string die de complete inhoud van het bestand bevat. `read()` kan een argument krijgen, dat ik zal bespreken in hoofdstuk 18, dat gaat over binaire bestanden.

Lezen uit een bestand verplaatst de pointer naar het teken dat volgt meteen na het laatste teken dat is gelezen. Dat betekent dat als je de `read()` methode aanroept zonder argumenten, de pointer verplaatst wordt naar het einde van het bestand (meteen achter het laatste teken in het bestand). Als je dan `read()` een tweede keer aanroept, valt er niks meer te lezen, omdat er niks staat na de pointer. `read()` retourneert dan een lege string.

16.2.3 Sluiten van een bestand met `close()`

Om een bestand te sluiten gebruik je de `close()` methode met de handle. Daarna is de handle niet meer gerelateerd aan het bestand. Ieder bestand dat je opent, moet je op enig moment weer sluiten in je programma.

Een compleet programma dat een bestand opent, de inhoud leest, de inhoud toont, en het bestand weer sluit, is dus het volgende:

listing1601.py

```
fp = open( "pc_rose.txt" )
print( fp.read() )
fp.close()
```

Als alles wat je met het bestand wilt doen, gedaan kan worden in een enkel blok code, dan kun je dat blok als volgt schrijven:

```
with open( <bestandsnaam> ) as <handle>:
    <acties>
```

Deze syntactische constructie heeft als voordeel dat het bestand automatisch gesloten wordt als het blok `<acties>` eindigt, dus je hoeft niet expliciet de `close()` methode aan te roepen. Deze constructie is typisch Python; je ziet hem niet in veel andere programmeertalen.

16.2.4 Tonen van de inhoud van een bestand

Nu ik de eerste paar functies en methodes voor bestandsmanipulatie heb geïntroduceerd, kan ik code schrijven die de inhoud van een bestand leest.

listing1602.py

```
with open( "pc_rose.txt" ) as fp:
    buffer = fp.read()
print( buffer )
```

Deze code verkeert in de veronderstelling dat een bestand met de naam "pc_rose.txt" beschikbaar is in dezelfde directory als waar het programma staat. Appendix E legt uit hoe je dit bestand kunt krijgen (als je het nog niet hebt). Als het bestand niet bestaat, krijg je een runtime error. Hoe je met zulke fouten om moet gaan wordt in het volgende hoofdstuk uitgelegd.

Opgave Wijzig in de code hierboven de bestandsnaam "pc_rose.txt" in een naam die niet bestaat. Bestudeer de fout die je krijgt als je het programma uitvoert.

Opgave Als je het bestand "pc_woodchuck.txt" hebt, wijzig dan de bestandsnaam in de code hierboven in die naam. Voer het programma uit en bestudeer de uitvoer.

16.2.5 Regels lezen met `readline()`

Om een tekstbestand regel voor regel te lezen, kun je de `readline()` methode gebruiken. Deze methode leest tekens uit het bestand, beginnend bij de pointer, tot aan en inclusief het volgende "newline" teken. Deze tekens worden als een string geretourneerd. Als je aan het einde van het bestand bent en je probeert een nieuwe regel te lezen, krijg je een lege string terug.

listing1603.py

```
fp = open( "pc_rose.txt" )
while True:
    buffer = fp.readline()
    if buffer == "":
        break
    print( buffer )
fp.close()
```

Als je de code hierboven uitvoert, zul je zien dat er een lege regel wordt getoond tussen ieder van de regels die uit het bestand gelezen is. Waar komen die extra regels vandaan? Denk hierover na.

De extra regels zijn er omdat de `readline()` methode een string retourneert met de tekens die gelezen zijn, inclusief het newline teken. Dus als de buffer wordt afgedrukt, wordt ook het newline teken afgedrukt. En omdat de **`print()`** functie zelf ook naar een nieuwe regel gaat nadat hij is uitgevoerd, krijg je een lege regel te zien na iedere tekstregel die wordt afgedrukt.

Opgave Schrijf een programma dat regels leest uit "pc_rose.txt," en alleen die regels toont die het woord `"naam"` bevatten.

16.2.6 Regels lezen met `readlines()`

Vergelijkbaar met de `readline()` methode is de `readlines()` methode. `readlines()` leest alle regels in een tekstbestand, en retourneert ze als een list van strings. De strings bevatten de newline tekens.

listing1604.py

```
fp = open( "pc_rose.txt" )
buffer = fp.readlines()
for line in buffer:
    print( line, end="" )
fp.close()
```

Als je de code hierboven uitvoert, zie je niet de lege regels tussen de tekstregels. Dat komt omdat ik aan de aanroep van de **`print()`** functie het `end=""` argument heb toegevoegd, zodat **`print()`** niet zelf naar een nieuwe regel gaat na het afdrukken.

16.2.7 Wanneer gebruik je welke lees-methode?

De `read()` en de `readlines()` methodes lezen beide een bestand als geheel in. Dat is geen probleem voor relatief kleine bestanden, maar voor grote bestanden kan het gebeuren dat je onvoldoende geheugen beschikbaar hebt om de inhoud van de bestanden op een efficiënte manier vast te houden. In dit soort omstandigheden (of als je niet weet hoe groot een te lezen bestand is) moet je het tekstbestand regel voor regel lezen middels de `readline()` methode.

Het is vaak een goed idee om, gedurende het ontwikkelen van code, alleen de eerste paar regels van een tekstbestand te verwerken. Je beperkt dan de tijd die je programma nodig heeft om het bestand te verwerken, en je beperkt de hoeveelheid uitvoer die je moet bestuderen, wat debuggen vergemakkelijkt. Bijvoorbeeld, de code hieronder verwerkt slechts de eerste 5 regels van een bestand.

listing1605.py

```
fp = open( "pc_jabberwocky.txt" )
teller = 0
while teller < 5:
```

```
    buffer = fp.readline()
    if buffer == "":
        break
    print( buffer, end="" )
    teller += 1
fp.close()
```

Als het programma klaar en foutvrij gemaakt is, kan ik de regeltellingen verwijderen en de loop wijzigen in **while True**, zodat het hele bestand verwerkt wordt.

Opgave Pas de code hierboven aan zodat je telt hoe vaak het woord "wauwelwok" (ongeacht gebruik van hoofd- of kleine letters) voorkomt in de eerste 5 regels. Druk alleen de telling af. Als het werkt, verwijder dan de regeltelling zodat je het programma uitvoert voor de tekst als geheel.

16.3 Schrijven in tekstbestanden

Het schrijven in een tekstbestand lijkt veel op het lezen uit een tekstbestand. Je opent het bestand, schrijft erin, en sluit het weer.

16.3.1 Openen voor schrijven

Om een bestand te openen voor schrijven, en voor schrijven alleen, geef je de waarde `"w"` mee als tweede argument aan de **open**`()` functie. Als het bestand nog niet bestaat, wordt het gecreëerd. Bestaat het wel, dan wordt het leeggemaakt.

Ik herhaal: **als je een bestand opent voor schrijven en het bestand bestaat al, dan wordt de inhoud van het bestand zonder pardon weggegooid!** Je zult geen waarschuwing krijgen die zegt "weet je het zeker?" Het bestand wordt gewoon leeggemaakt. Dus je moet heel erg voorzichtig zijn met het openen van een bestand voor schrijven. Gewoonlijk vraag ik studenten om hun programma's zo te schrijven dat eerst gecontroleerd wordt of een bestand bestaat alvorens het voor schrijven wordt geopend, en als het bestaat, een foutmelding te geven. Functies om te controleren of een bestand bestaat volgen later in dit hoofdstuk.

16.3.2 Schrijven met `write()`

Om iets te schrijven naar een tekstbestand, gebruik je de `write()` methode met als argument een string die je wilt schrijven naar het bestand. De code hieronder vraagt je om een paar strings in te geven, en schrijft ze dan naar een bestand. Het programma stopt met het vragen om strings als je een lege string ingeeft. Aan het einde opent het programma het geschreven bestand, leest de inhoud, en toont die op het scherm. Voer deze code uit, geef op zijn minst twee strings in, en zie wat er gebeurt.

listing1606.py

```
fp = open( "pc_writetest.tmp", "w" )
while True:
```

```
    tekst = input( "Geef een regel tekst: " )
    if tekst == "":
        break
    fp.write( tekst )
fp.close()

fp = open( "pc_writetest.tmp" )
buffer = fp.read()
fp.close()

print( buffer )
```

Als je hebt gedaan wat ik vroeg, zie je dat alle tekst die je hebt ingegeven in het bestand staat, maar dat alles op één lange regel staat. Er staan geen newline tekens in het bestand. De reden is dat je newline tekens expliciet moet schrijven als je ze in het bestand wilt hebben. Als je input van het toetsenbord leest via de **`input()`** functie, stop je weliswaar met input verstrekken door op `Enter` te drukken, maar dat heeft dan niet als resultaat dat er een newline teken in de ingegeven string staat. Dus je moet dat newline teken zelf toevoegen als je die nieuwe regels wilt zien.

Opgave Pas de code hierboven aan zodat er een newline teken in het bestand komt te staan na iedere ingegeven regel.

16.3.3 Schrijven met `writelines()`

Je kunt een list van strings in één keer naar een bestand schrijven, via de `writelines()` methode die de list als argument krijgt. Als je newline tekens tussen de strings wilt, moet je die expliciet opnemen aan het einde van iedere string in de list. `writelines()` is de tegenhanger van `readlines()`; als je de list die `readlines()` retourneert als argument voor `writelines()` gebruikt, zal de inhoud van het uitvoerbestand exact gelijk zijn aan de inhoud van het invoerbestand.

Er is geen `writeline()` methode. `writeline()` zou precies hetzelfde zijn als `write()`, dus hij is overbodig.

16.3.4 Oefening

Opgave Schrijf een programma dat de inhoud van “pc_rose.txt” leest, en exact dezelfde inhoud schrijft in een bestand “pc_writetest.tmp.” Open dan het bestand “pc_writetest.tmp” en toon de inhoud. Je kunt dit programma gemakkelijk bouwen door wat van de hierboven gegeven code bij elkaar te plakken.

Opgave Schrijf een programma dat de inhoud van “pc_rose.txt” leest, iedere regel achterstevoren zet, en dan de geïnverteerde regels wegschrijft naar het bestand “pc_writetest.tmp.” Open daarna “pc_writetest.tmp” en toon de inhoud.

16.4 Toevoegen aan tekstbestanden

"Toevoegen"(Engels: "appending") houdt in dat er geschreven wordt aan het einde van een bestaand bestand. Als je een bestand opent om toe te voegen, wordt de inhoud niet verwijderd, maar wordt de pointer geplaatst aan het einde van het bestand, waar je dan nieuwe data mag wegschrijven. Om een bestand in deze modus te openen, gebruik je `"a"` als het tweede argument bij het openen van het bestand.

De code hieronder toont eerst de inhoud van "pc_writetest.tmp" (die nu zou moeten bestaan). Daarna wordt de gebruiker gevraagd om regels in te geven die aan het bestand worden toegevoegd. Tenslotte wordt de nieuwe inhoud van het bestand getoond. Ik heb dit programma iets beter gestructureerd dan ik hiervoor steeds deed, door gebruik van een constante voor de bestandsnaam en middels een functie voor het tonen van de bestandsinhoud.

listing1607.py

```
NAAM = "pc_writetest.tmp"

def tooninhoud( bestandsnaam ):
    fp = open( bestandsnaam )
    print( fp.read() )
    fp.close()

tooninhoud( NAAM )

fp = open( NAAM, "a" )
while True:
    tekst = input( "Geef een regel tekst: " )
    if tekst == "":
        break
    fp.write( tekst+"\n" )
fp.close()

tooninhoud( NAAM )
```

16.5 `os.path` methodes

Je weet nu alles wat je moet weten om tekstbestanden in Python te manipuleren. Er zijn nog een aantal handige functies beschikbaar die het werken met bestanden vergemakkelijken. Deze vind je in de `os.path` module. Zoals gewoonlijk geef ik ze hier niet allemaal, maar ik noem wel de functies die je het meest zult gebruiken.

In deze functies refereert de term "pad" (Engels: "path") aan een bestandsnaam of directory naam, compleet met het volledige pad vanaf de "root." Zelfs als het pad niet volledig genoemd is, is het impliciet opgenomen aangezien ieder bestand op een specifieke plaats in het bestandssysteem te vinden is.

16.5.1 exists()

De functie `exists()` krijgt een pad als argument, en retourneert **`True`** als het pad bestaat, en anders **`False`**.

```
from os.path import exists

if exists( "pc_rose.txt" ):
    print( "pc_rose.txt bestaat" )
else:
    print( "pc_rose.txt bestaat niet" )

if exists( "pc_tulip.txt" ):
    print( "pc_tulip.txt bestaat" )
else:
    print( "pc_tulip.txt bestaat niet" )
```

16.5.2 isfile()

`isfile()` test of het pad dat als argument gegeven is een bestand is. Als het dat is, retourneert de functie **`True`**. Anders retourneert het **`False`**. Als het pad in het geheel niet bestaat, retourneert de functie ook **`False`**.

```
from os.path import isfile

if isfile( "pc_rose.txt" ):
    print( "pc_rose.txt is een bestand" )
else:
    print( "pc_rose.txt is geen bestand" )
```

16.5.3 isdir()

`isfile()` test of het pad dat als argument gegeven is een directory (folder) is. Als het dat is, retourneert de functie **`True`**. Anders retourneert het **`False`**. Als het pad in het geheel niet bestaat, retourneert de functie ook **`False`**.

```
from os.path import isdir

if isdir( "pc_rose.txt" ):
    print( "pc_rose.txt is een directory" )
else:
    print( "pc_rose.txt is geen directory" )
```

16.5.4 join()

`join()` krijgt één of meerdere delen van een pad mee als argument, en plakt die op een redelijk slimme manier aan elkaar om een geschikte naam voor een pad te vormen, die het

retourneert. Dit betekent dat het "slashes" verwijdert of toevoegt waar nodig. join() is vooral handig in combinatie met listdir() (uitgelegd in hoofdstuk 15, en als voorbeeld gebruikt hieronder).

De reden dat join() handig is in combinatie met listdir(), is dat listdir() een list van bestandsnamen teruggeeft, waarbij de directory namen niet zijn opgenomen. Als je een list van bestandsnamen vraagt, wil je ze meestal op een of ander moment openen. Maar als je een bestand probeert te openen dat niet in de huidige directory staat, dan moet je het complete pad kennen. Als je listdir() uitvoert, geef je de directory mee als argument, dus je weet waar de bestanden zich bevinden. Om een compleet pad te bouwen, moet je dus die directory naam aan de naam van ieder bestand toevoegen. In plaats van zelf te beslissen waar je "slashes" moet zetten (en wat voor slashes het moeten zijn), kun je de samenstelling van het pad overlaten aan de join() functie.

De code hieronder zoekt alle bestanden in de huidige directory, en toont ze inclusief het complete pad. De code laat zien hoe je een padnaam bouwt middels join().

```
from os import listdir, getcwd
from os.path import join

bestandslist = listdir( "." )
for naam in bestandslist:
    pad = join( getcwd(), naam )
    print( pad )
```

16.5.5 basename()

basename() haalt de bestandsnaam uit een pad, en retourneert die.

```
from os.path import basename

print( basename( "/System/Home/readme.txt" ) )
```

16.5.6 dirname()

dirname() haalt de directory naam uit een pad, en retourneert die.

```
from os.path import dirname

print( dirname( "/System/Home/readme.txt" ) )
```

16.5.7 getsize()

getsize() krijgt een pad als argument, en retourneert de grootte van het betreffende bestand als een integer (die het aantal bytes weergeeft). Als het pad geen bestand is, krijg je een runtime error.

```
from os.path import getsize

num = getsize( "pc_rose.txt" )
print( num )
```

Opgave Schrijf een programma dat de groottes van alle bestanden in de huidige directory bij elkaar optelt, en het resultaat toont.

16.6 Encoding

Tekstfiles hebben een zogeheten "encoding" (letterlijk betekent dit "versleuteling," maar die term wordt nooit gebruikt). Dit is een systeem dat voorschrijft hoe de tekens in een bestand geïnterpreteerd moeten worden. Encoding kan verschillend zijn tussen besturingssystemen. Je kunt de standaard manier van encoding voor een besturingssysteem zien door de functie `sys.getfilesystemencoding()` aan te roepen.

```
from sys import getfilesystemencoding

print( getfilesystemencoding() )
```

Als je een tekstbestand leest dat een andere manier van encoding gebruikt dan je bestandssysteem prefereert, kun je een `UnicodeDecodeError` krijgen. Of je deze fout krijgt voor een bepaald bestand, hangt af van je besturingssysteem. Een vervelende consequentie daarvan is, dat als je code hebt geschreven die een bestand leest en die fatsoenlijk werkt, en je die code naar een ander besturingssysteem brengt, een bestand dat voorheen gelezen kon worden plotseling de code laat stuklopen.[20]

Een eenvoudige manier om dit probleem te omzeilen is een extra parameter toevoegen aan het openen van een bestand, die aangeeft welk encoding mechanisme je wilt gebruiken om het bestand te lezen. Je doet dit via een parameter `encoding=<encodingnaam>`, waarbij `<encodingnaam>` een string is die verschillende waardes kan hebben. Een paar typische waardes voor deze parameter zijn:

- `ascii`: 7-bits encoding, tekens met waardes in het bereik 00-7F
- `latin-1`: 8-bits encoding, tekens met waardes in het bereik 00-FF
- `mbcs`: 2-byte encoding, die momenteel vervangen wordt door UTF-8
- `utf-8`: encoding met een variabel aantal bytes

[20] Ik moet hier een opmerking maken over een fenomeen dat wat bizar kan overkomen als je het voor het eerst tegenkomt: Je kunt deze fout krijgen als je bestand tekens bevat met een encoding die niet door je besturingssysteem ondersteund wordt, zelfs als die tekens zich bevinden in regels in het bestand die je niet eens probeert in te lezen! Bijvoorbeeld, stel dat je zo'n speciaal teken hebt op regel 10 in het bestand, en je probeert alleen de eerste 5 regels van het bestand te lezen voordat je het bestand weer sluit – je programma kan dan nog steeds de genoemde fout geven! Ik vermoed dat dit gerelateerd is aan het "bufferen" van data: als je Python vraagt een klein stukje data uit een bestand te lezen, dan leest Python toch grotere delen van het bestand, zodat het sneller is als je later meer van het bestand gaat lezen. Dus door slim te zijn, kan Python je met problemen opzadelen die je niet aan zag komen. Het is goed om je bewust te zijn van dit soort eigenschappen van Python.

Gewoonlijk worden tekstbestanden gecreëerd met `ascii` of `latin-1` encoding. Omdat `ascii` een onderdeel is van `latin-1`, kun je bij het openen van een tekstbestand altijd `latin-1` encoding gebruiken. Het is mogelijk dat als er tekens in het bestand staan die vallen buiten de `ascii` range, je andere tekens ziet dan waarmee het bestand oorspronkelijk gebouwd is – dat is afhankelijk van het encoding mechanisme van je bestandssysteem. Maar een `UnicodeDecodeError` zul je niet krijgen. Dus als je de inhoud van een bestand probeert te lezen en je krijgt een `UnicodeDecodeError`, kun je proberen het te openen via **`open`**`( <bestand>, encoding="latin-1" )`. Over het algemeen lost dit je probleem op.

Merk op dat `utf-8` een veel groter bereik aan tekens ondersteunt dan `latin-1`, maar als je een tekstbestand dat gemaakt is met `latin-1` encoding probeert te gebruiken met een bestandssysteem dat gebaseerd is op `utf-8` encoding, kun je toch een `UnicodeDecodeError` krijgen. Dat komt doordat `utf-8` geen tekens kent met waardes in het (hexadecimale) bereik 80-FF.

Als je wilt zien welke speciale tekens door jouw bestandssysteem ondersteund worden met waardes in het bereik 80-FF, kun je de code hieronder uitvoeren. De numerieke waarde van een teken in de tabel kun je afleiden door $16 * rij + kolom$ te berekenen, waarbij `rij` en `kolom` de hexadecimale nummering zijn van de rij en kolom. Met deze code laat ik geen tekens zien in het bereik 80-9F, omdat die meestal niet ingevuld zijn.

listing1608.py

```
for i in range(16):
    if i < 10:
        print( ' '+chr( ord( '0' )+i ), end='' )
    else:
        print( ' '+chr( ord( 'A' )+i-10 ), end='' )
print()
for i in range( 10, 16 ):
    print( chr( ord( 'A' )+i-10 ), end='' )
    for j in range( 16 ):
        c = i*16+j
        print( ' '+chr( c ), end='' )
    print()
```

Ik geef meer details over UTF-8 encoding in hoofdstuk 19, maar voor het manipuleren van tekstbestanden heb je voldoende aan de bovenstaande informatie.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Tekstbestanden
- Pointers
- Openen en sluiten van bestanden met **`open`**`()` en `close()`
- Lezen met `read()`, `readline()`, en `readlines()`
- Schrijven met `write()` en `writelines()`

- Toevoegen aan bestanden
- `os.path` methodes `exists()`, `isfile()`, `isdir()`, `join()`, `basename()`, `dirname()`, en `getsize()`
- Omgaan met tekstbestanden met verschillende encoding mechanismes

Opgaves

Opgave 16.1 Schrijf een programma dat het bestand "pc_woodchuck.txt" leest, en het splitst in individuele woorden (waarbij alles wat geen letter is, beschouwd wordt als een woord-scheider). Daarna bouwt het (zonder onderscheid te maken tussen hoofd- en kleine letters) een dictionary dat voor ieder woord bijhoudt hoe vaak het voorkomt in de tekst. Toon vervolgens alle woorden, met hun hoeveelheden, in alfabetische volgorde.

Opgave 16.2 Doe hetzelfde als de vorige opgave, maar verwerk nu de tekst regel voor regel. Dit is iets dat je zou moeten doen als je een erg lange tekst onder handen hebt.

Opgave 16.3 Schrijf een programma dat de inhoud van "pc_woodchuck.txt" regel voor regel inleest. Creëer een bestand "pc_woodchuck.tmp" in de huidige directory, met dezelfde inhoud als "pc_woodchuck.txt," maar waarbij alle klinkers verwijderd zijn. Toon op het scherm hoeveel tekens je hebt gelezen en hoeveel tekens je hebt geschreven.

Opgave 16.4 Schrijf een programma dat bepaalt welke woorden van minimaal drie letters voorkomen in ieder van de drie bestanden "pc_woodchuck.txt," "pc_jabberwocky.txt" en "pc_rose.txt." Maak geen onderscheid tussen hoofd- en kleine letters, en, zoals gewoonlijk, is ieder teken dat geen letter is een woord-scheider. Als je programma correct is, vind je twee woorden van drie letters die aan de eis voldoen.

Opgave 16.5 Schrijf een programma dat voor ieder van de drie bestanden die je in de vorige opgave hebt gebruikt, telt hoe vaak ieder van de 26 letters van het alfabet (zonder onderscheid tussen hoofd- en kleine letters) voorkomt. Bereken voor ieder van de bestanden de fractie `<aantal keer dat de letter voorkomt in het bestand>/<totaal aantal letters in het bestand>`. Schrijf een bestand (met een willekeurige naam, als je het maar veilig kunt overschrijven) met extensie `.csv`, dat 26 regels bevat, waarbij iedere regel als volgt geformatteerd is: `"<letter>",<fractie voor eerste bestand>,<fractie voor tweede bestand>,<fractie voor derde bestand>`. De eerste regel heeft letter `a`, de tweede letter `b`, etcetera. De fracties worden opgeslagen met 5 decimalen. Toon tenslotte de inhoud van het bestand. Aangezien het bestand een CSV bestand is, kun je het ook openen in een spreadsheet programma.

Een snelle controle om te zien of je uitkomst correct is, is dat alle fracties tussen nul en 1 moeten liggen, en dat de fracties voor "e" het hoogste is voor alle bestanden, terwijl ook de fracties voor "a" en "n" behoorlijk hoog zijn. Als je langere bestanden gebruikt die allemaal in dezelfde taal geschreven zijn, zullen de fracties meestal veel dichter bij elkaar liggen.

Hoofdstuk 17

Exceptions

Soms treden runtime errors op niet omdat je een programmeerfout hebt gemaakt, maar omdat er een probleem optreedt dat je niet kon voorzien toen je het programma schreef. Dit is buitengewoon relevant als je met bestanden werkt: bijvoorbeeld, als je een bestand verwerkt dat op een USB-stick staat, en de gebruiker verwijdert de USB-stick tijdens de verwerking, krijg je uiteraard een fout die je niet echt zou kunnen voorzien in je code. Iedere runtime error genereert in de code een zogenaamde "exception" ("uitzondering") die je kunt "afvangen." Het afvangen van een exception betekent dat je in je programma code opneemt die ervoor zorgt dat de opgetreden fout zoveel mogelijk netjes wordt afgehandeld, in plaats van je programma abrupt af te breken.

17.1 Errors en exceptions

Als je een Python programma start, controleert Python eerst of alle statements in je programma voldoen aan de syntax-eisen die Python stelt. Als dat niet het geval is, genereert Python een zogeheten "syntax error" en wordt het programma niet uitgevoerd. Als Python geen syntax errors tegenkomt, wordt het programma uitgevoerd, maar er kunnen dan nog steeds statements gevonden worden die fouten genereren als ze worden uitgevoerd. Zulke statements veroorzaken een "runtime error." Je hebt runtime errors regelmatig gezien tijdens het ontwikkelen en testen van programma's (ontken het maar niet).

Over het algemeen zul je proberen runtime errors op te lossen door je code uit te breiden of te wijzigen. Bijvoorbeeld, het volgende programma geeft een runtime error als je nul ingeeft:

```
from pcinput import getInteger

num = getInteger( "Geef een getal: " )
print( "3 gedeeld door {} is {}".format( num, 3/num ) )
print( "Tot ziens!" )
```

Python vertelt je wat voor soort error het is, namelijk een `ZeroDivisionError`. Om die op te lossen, kun je het programma wijzigen:

```
from pcinput import getInteger

num = getInteger( "Geef een getal: " )
if num == 0:
    print( "Je kunt niet delen door nul" )
else:
    print( "3 gedeeld door {} is {}".format( num, 3/num ) )
print( "Tot ziens!" )
```

`ZeroDivisionError` is de naam van een "exception" die Python genereert.[21] Als je een exception niet afhandelt in je programma, breekt Python de uitvoering van het programma af en smijt een foutmelding op het scherm. Dit houdt in dat je ervoor zou kunnen zorgen dat het programma gewoon verder doorloopt, als je er maar voor zorgt dat de exception afgehandeld wordt.

In de code hierboven zou je ervoor moeten zorgen dat er nimmer een exception wordt gegenereerd omdat door nul gedeeld wordt – aangezien je kunt voorzien dat dat zou kunnen gebeuren. Het gebeurt echter wel eens dat je zult moeten accepteren dat exceptions kunnen optreden omdat je gewoonweg niet alle omstandigheden waarin je programma wordt uitgevoerd kunt voorzien. Dat is specifiek het geval als je programma afhangt van zaken die je niet onder controle kunt hebben, zoals bij het werken met bestanden en gebruikershandelingen.

17.2 Afhandelen van exceptions

Om exceptions expliciet in je programma af te handelen, gebruik je de **`try ... except`** constructie. Er zijn verschillende manieren om deze constructie toe te passen.

17.2.1 `try` ... `except`

De meest basale vorm van de **`try ... except`** constructie heeft de volgende syntax:

```
try:
    <acties>
except:
    <exception afhandeling>
```

Als de `<acties>` tussen **`try:`** en **`except:`** worden uitgevoerd en er een exception wordt gegenereert, springt Python onmiddellijk naar de `<exception afhandeling>` en voert die uit, waarna het programma vervolgt met de regels code onder de `<exception afhandeling>`. Als er geen exceptions optreden gedurende de uitvoering van `<acties>`, wordt de `<exception afhandeling>` overgeslagen.

Gebruik makend van exception afhandeling, kan de code aan het begin van dit hoofdstuk als volgt geschreven worden om runtime errors te vermijden:

[21] In het Engels heet dit "raising an exception." `raise` is een gereserveerd woord in Python, dat iets later in dit hoofdstuk besproken wordt.

listing1701.py

```
from pcinput import getInteger

num = getInteger( "Geef een getal: " )
try:
    print( "3 gedeeld door {} is {}".format( num, 3/num ) )
except:
    print( "Je kunt niet delen door nul" )
print( "Tot ziens!" )
```

Meerdere statements mogen binnen een enkele **try ... except** gebruikt worden. Bijvoorbeeld, de volgende code genereert een exception als de gebruiker een nul ingeeft of als de gebruiker een 3 ingeeft. Beide uitzonderingen worden via dezelfde **try ... except** constructie afgehandeld.

listing1702.py

```
from pcinput import getInteger

num = getInteger( "Geen een getal: " )
try:
    print( "3 gedeeld door {} is {}".format( num, 3/num ) )
    print( "3 gedeeld door {}-3 is {}".format( num, 3/(num-3) ) )
except:
    print( "Je kunt niet delen door nul" )
print( "Tot ziens!" )
```

Dit is een beetje lelijk, niet alleen omdat deze fouten vermeden hadden kunnen worden in plaats van ze af te handelen via exceptions, maar ook omdat wanneer een exception optreedt het onduidelijk is welk van de twee statements deze veroorzaakt heeft (hoewel je in dit geval kunt zien dat als je 3 ingeeft, de eerste van de twee statements onder de **try** correct is uitgevoerd). Maar dit is slechts een demonstratie, en er zijn zeker situaties te bedenken waarin je zegt: "het maakt mij niet uit waar er een exception optreedt in deze regels code, maar als er iets gebeurt, wil ik in ieder geval *dit* doen."

17.2.2 Afhandeling van specifieke exceptions

Bekijk de code hieronder. Er kunnen minstens twee exceptions optreden als deze code wordt uitgevoerd. Welke?

```
print( 3 / int( input( "Geef een getal: " ) ) )
```

De twee exceptions die in deze code kunnen optreden zijn de `ZeroDivisionError` als je een nul ingeeft, en de `ValueError` als je iets ingeeft dat geen integer is. Probeer het als je dat niet al gedaan hebt.

Je kunt beide exceptions afhandelen met een enkele **try ... except** constructie, maar je kunt ze ook van elkaar onderscheiden door meerdere **except**s te gebruiken. Iedere **except** kan gevolgd worden door één van de specifieke exceptions, en de code die bij die **except** hoort wordt alleen uitgevoerd als die specifieke exception wordt gegenereerd.

listing1703.py

```
try:
    print( 3 / int( input( "Geef een getal: " ) ) )
except ZeroDivisionError:
    print( "Je kunt niet delen door nul" )
except ValueError:
    print( "Je gaf geen getal" )
print( "Tot ziens!" )
```

Als je "alle overige exceptions" wilt afhandelen, kun je een **except** zonder specifieke exception aan het einde toevoegen. Slechts één van de **except**s zal worden uitgevoerd, namelijk de eerste die wordt aangetroffen die van toepassing is. Dit werkt dus ongeveer als een **if ... elif ... elif ... else** constructie.

listing1704.py

```
try:
    print( 3 / int( input( "Geef een getal: " ) ) )
except ZeroDivisionError:
    print( "Je kunt niet delen door nul" )
except ValueError:
    print( "Je gaf geen getal" )
except:
    print( "Iets onverwachts ging fout" )
print( "Tot ziens!" )
```

Hier zijn een aantal specifieke exceptions die vaak optreden:

- `ZeroDivisionError`: Delen door nul
- `IndexError`: Het benaderen van een list of tuple met een index die niet binnen het legale bereik valt
- `KeyError`: Het benaderen van een dictionary met een key die onbekend is
- `IOError`: Iedere fout die kan optreden als je een bestand benadert (deze exception is een alias voor `OSError`)
- `FileNotFoundError`: Het proberen te openen van een niet-bestaand bestand om eruit te lezen
- `ValueError`: Het optreden van een fout bij het "casten" van een waarde naar een andere waarde
- `TypeError`: Het gebruiken bij een operatie van een waarde met een data type dat niet ondersteund wordt door de operatie

Opgave De code hieronder kan verschillende exceptions genereren. Deze worden nu afgehandeld door een enkele **try ... except** constructie. Breid deze code uit met de expliciete afhandeling van alle exceptions die van toepassing zijn (er zijn er minimaal drie). Ik wil hierbij benadrukken dat ik liever zie dat je exceptions vermijdt dan dat je ze afhandelt, maar in dit geval wil ik je laten oefenen met het afhandelen van exceptions.

listing1705.py

```
fruitlist = ["appel", "banaan", "kers"]
try:
    num = input( "Geef een getal: " )
    if "." in num:
        num = float( num )
    else:
        num = int( num )
    print( fruitlist[num] )
except:
    print( "Er ging iets fout" )
```

17.2.3 Toevoegen van een `else`

Aan het einde van een **`try ... except`** constructie kun je een **`else`** toevoegen. De acties die bij de **`else`** staan worden alleen uitgevoerd als er geen exception optreedt. Bijvoorbeeld, in de code hieronder wordt de berekende waarde voor `num` alleen getoond als er geen exception wordt gegenereerd.

listing1706.py

```
try:
    num = 3 / int( input( "Geef een getal: " ) )
except ZeroDivisionError:
    print( "Je kunt niet delen door nul" )
except ValueError:
    print( "Je gaf geen getal" )
except:
    print( "Iets onverwachts ging fout" )
else:
    print( num )
print( "Tot ziens!" )
```

Zelf geef ik er de voorkeur aan geen **`else`** te gebruiken bij een exception, omdat het voor mij aanvoelt alsof de code onder de **`except`**s code is die alleen moet worden uitgevoerd in uitzonderlijke omstandigheden. Maar het staat je vrij deze constructie te gebruiken als je er het nut van inziet.

17.2.4 Toevoegen van een `finally`

Je kunt nog een extra tak toevoegen aan een **`try`** constructie, namelijk **`finally`**. Bij **`finally`** kun je een serie statements opnemen die worden uitgevoerd ongeacht de wijze waarop de **`try`** constructie wordt verlaten. Als alles normaal verloopt, worden de statements bij de **`finally`** uitgevoerd, maar ook als je een runtime error krijgt, worden ze uitgevoerd. Je kunt bijvoorbeeld **`finally`** gebruiken om er zeker van te zijn dat een bestand dat je geopend hebt, wordt gesloten.

listing1707.py

```
try:
    fp = open( "pc_rose.txt" )
    print( "Bestand geopend" )
    print( fp.read() )
finally:
    fp.close()
    print( "Bestand gesloten" )
```

17.2.5 Informatie over een exception

Je kunt extra informatie krijgen over een exception door een **as** toe te voegen aan een **except**, middels de syntax **except** <exception> **as** <variabele>. De variabele bevat dan een exception "object," dat meer informatie bevat over de exception. Helaas is er geen standaard manier waarop je de informatie eruit kunt krijgen: het is afhankelijk van de specifieke exception wat de variabele bevat.

De variabele bevat altijd een tuple met argumenten, die tijdens het genereren van de exception bepaald zijn. Je kunt deze argumenten inspecteren middels <variabele>.args. Een ValueError krijgt een tuple met slechts één waarde, namelijk een string.

listing1708.py

```
try:
    print( int( "GeenInteger" ) )
except ValueError as ex:
    print( ex.args )
```

Als je de code hieronder uitvoert, zie je dat een IOError een tuple met twee waardes heeft: een integer en een string. De integer is kan erg informatief zijn, aangezien hij aangeeft wat er fout ging (als je de codes snapt).

listing1709.py

```
try:
    fp = open( "GeenBestand" )
    fp.close()
except IOError as ex:
    print( ex.args )
```

17.2.6 Advies voor het gebruik van exception afhandeling

Als je een exception opvangt in je code, handel hem dan ook netjes af, en negeer hem niet. Gebruik zeker geen generieke **try ... except** constructie waarna je niks doet met een exception. Als je denkt dat je een bepaalde exception veilig kunt negeren, vang dan alleen die specifieke exception af, en zet commentaar in je programma dat aangeeft waarom je meent dat je de exception kunt negeren. In principe moeten alle exceptions in je programma ofwel afgehandeld worden, ofwel het programma laten crashen.

17.3 Exceptions bij bestandsmanipulatie

Ieder probleem dat optreedt bij het benaderen van een bestand, of het nu het niet kunnen vinden van het bestand betreft, of de onmogelijkheid om het bestand te lezen of the schrijven, of het doen van een poging om een systeembestand of een directory te openen, leidt tot een `IOError` exception. Omdat het niet ongebruikelijk is dat zulke problemen optreden, en ze regelmatig niet door de programmeur voorzien kunnen worden, is het een goed idee om `IOError` exceptions in je programma af te vangen waar mogelijk.

Omdat er zoveel dingen fout kunnen gaan bij bestanden, kan de tuple `args` die ik hierboven besprak gebruikt worden om meer informatie over het probleem te krijgen. Bijvoorbeeld, als je programma aan de gebruiker vraagt een bestandsnaam op te geven, en als je dan een `IOError` krijgt als je het bestand probeert te openen, dan zou het error nummer (het eerste element van de tuple) kunnen aangeven dat het bestand niet bestaat (2). Een geschikte afhandeling zou dan zijn dat je de gebruiker om een nieuwe bestandsnaam vraagt.

De error nummers zijn gedefinieerd in de `errno` module, die je in je programma kunt importeren. De module geeft een aantal constanten die je kunt opnemen in je code in plaats van getallen, en het is de gewoonte om dat ook zo te doen. De meest voorkomende error nummers zijn:

- `errno.ENOENT`: Dit bestand of deze directory bestaat niet. Dit krijg je als je een bestand benadert dat niet bestaat.
- `errno.EACCESS`: Toestemming geweigerd. Je kunt deze melding in verschillende omstandigheden krijgen, zoals wanneer je probeert te lezen uit een gesloten bestand, of als je in een bestand probeert te schrijven dat voor alleen lezen bedoeld is, of als je een directory probeert te openen alsof het een bestand is.
- `errno.ENOSPC`: Onvoldoende ruimte. Je krijgt deze fout als je een bestand probeert te schrijven en er geen ruimte voor het bestand beschikbaar is, bijvoorbeeld als je probeert te schrijven naar een USB-stick die vol is.

Er is een grote lijst met dit soort error nummers, die je kunt vinden in Python handleidingen. Je snapt ze wellicht niet allemaal, en het is dan ook zo dat een hoop ervan nogal archaïsch zijn en niet meer kunnen voorkomen op moderne computers. Het beste kun je proberen om alle `IOErrors` af te vangen, en als je er een tegenkomt tijdens het ontwikkelen van een programma, de argumenten af te drukken zodat je het error nummer kent, en ook de foutboodschap. Je kunt dan opzoeken wat de fout precies inhoudt, en hem in je programma afvangen als dat op een zinvolle manier kan.

Maar net als met andere exceptions, kun je ook exceptions bij bestandsmanipulatie beter vermijden dan afvangen. Er is geen enkele reden dat je ooit een “bestand bestaat niet” fout zou mogen krijgen, omdat je kunt testen of een bestand bestaat met de `exists()` en `isfile()` functies uit de `os.path` module.

listing1710.py

```
import errno

try:
    fp = open( "GeenBestand" )
    fp.close()
```

```
except IOError as ex:
    if ex.args[0] == errno.ENOENT:
        print( "Bestand niet gevonden!" )
    else:
        print( ex.args[0], ex.args[1] )
```

`FileNotFoundError` is een "subclass" (zie hoofdstuk 22) van `IOError`. Dit betekent dat het afvangen van `FileNotFoundError` equivalent is met het afvangen van `IOError` **`as`** `ex` en testen of `ex.args[0]` de waarde `errno.ENOENT` bevat.

17.4 Genereren van exceptions

Je mag zelf in je code ook exceptions genereren. Daarvoor is het gereserveerde woord **`raise`** bestemd. Je laat het volgen door één van de bekende exceptions (je mag eventueel ook je eigen exceptions definiëren, maar daarvoor moet je de hoofdstukken 20 tot en met 22 bestuderen). Je mag een exception die je genereert willekeurige argumenten meegeven, en die zijn dan weer via het `args` attribuut te benaderen.

Je vraagt je misschien af waarom je exceptions zou willen genereren. Het antwoord is dat als je een module programmeert, en er een fout kan optreden in een van de functies in de module (bijvoorbeeld omdat het hoofdprogramma de verkeerde parameters meegeeft), het eigenlijk niet de bedoeling is dat je foutmeldingen print. Het is veel netter om een exception te genereren, en het hoofdprogramma deze exception af te laten handelen. Hier is een voorbeeld:

listing1711.py

```
def getStringLenMax10( prompt ):
    s = input( prompt )
    if len( s ) > 10:
        raise ValueError( "Lengte groter dan 10", len( s ) )
    return s

print( getStringLenMax10( "Gebruik 10 tekens of minder: " ) )
```

Als je deze code uitvoert, zie je dat als je een string invoert die meer dan 10 tekens bevat, een `ValueError` exception wordt gegenereerd. De exception krijgt twee argumenten, die je als een tuple getoond ziet worden als Python de exception op het scherm gooit. Je kunt deze exception in de code afhandelen op dezelfde manier als je exceptions afhandelt die door Python worden geproduceerd.

Het gereserveerde woord **`raise`** heeft een tweede functie: als je in de afhandeling van een **`except`** zit, dan kun je, in plaats van de exception meteen af te handelen, de exception doorgeven naar het "hoger gelegen niveau" van het programma. Je doet dat door het command **`raise`** te geven, zonder parameters, en de exception wordt dan als "nog niet afgehandeld" beschouwd. Dit kan nuttig zijn als je een beetje extra afhandeling wilt doen voordat het programma "crasht" op grond van de exception, of de exception elders wordt opgevangen. Bijvoorbeeld:

listing1712.py

```
fp = open( "pc_rose.txt ")
try:
    buf = fp.read()
    print( buf )
except IOError:
    fp.close()
    raise
fp.close()
```

Deze code loopt waarschijnlijk goed, maar als er een `IOError` optreedt tijdens het lezen van het bestand, dan wordt het bestand netjes gesloten alvorens de exception wordt doorgegeven.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Exception afhandeling
- **`except`**, **`else`**, en **`finally`**
- `ZeroDivisionError`, `IndexError`, `KeyError`, `IOError`, `ValueError`, `TypeError`, en `FileNotFoundError`
- Informatie uit exceptions halen
- Exceptions bij bestandsmanipulatie
- **`raise`**

Opgaves

Opgave 17.1 Welke exceptions kan de code hieronder veroorzaken? Breid de code uit zodat alle exceptions op een redelijke manier worden afgehandeld.

exercise1701.py

```
numlist = [ 100, 101, 0, "103", 104 ]

i1 = int( input( "Geef een index: " ) )
print( "100 /", numlist[i1], "=", 100 / numlist[i1] )
```

Hoofdstuk 18

Binaire Bestanden

"Binaire bestanden" is de term die gebruikt wordt om te refereren aan alle bestanden die geen tekstbestanden zijn. Uitvoerbare programma's zijn binaire bestanden, evenals plaatjes, films, tekstverwerker documenten, en vele andere bestandstypes. Het is niet gebruikelijk om binaire bestanden af te handelen met Python code (voor veel soorten binaire bestanden zijn gerichte programma's geschreven om ze te bewerken – denk bijvoorbeeld aan een programma als GIMP of Photoshop om plaatjes te bewerken), maar het is wel mogelijk. Dit hoofdstuk legt uit hoe je met binaire bestanden omgaat.

18.1 Openen en sluiten van binaire bestanden

Het afhandelen van binaire bestanden lijkt veel op het afhandelen van tekstbestanden. Je moet het bestand openen met **`open`**`()` om de inhoud te kunnen benaderen, je sluit het af met `close()` als je klaar bent, en je kunt lezen en schrijven met respectievelijk `read()` en `write()`.

Als je een binair bestand opent, moet je aan Python aangeven dat je dit bestand wilt behandelen in "binaire modus." Je doet dit door de letter `"b"` aan het modus argument toe te voegen. Bijvoorbeeld, om een bestand te openen in "binair lezen" modus, gebruik je `"rb"` als modus. Je kunt ook een binair bestand openen voor zowel lezen als schrijven; dat geef je aan met de modus `"r+"`, dus lezen en schrijven in binaire modus is `"r+b"` (hoewel je eventueel ook een tekstbestand kunt openen in lezen plus schrijven modus, via `"r+"`, gaf ik dat niet aan in hoofdstuk 16, omdat dat zelden zinvol is). Net als bij tekstbestanden, kun je een binair bestand openen in "alleen schrijven" modus met `"wb"`; het bestand wordt dan leeggemaakt. `"w+b"` opent een bestand voor zowel lezen als schrijven, maar maakt in tegenstelling tot `"r+b"` het bestand ook leeg om mee te beginnen.

Als je een bestand opent voor zowel lezen als schrijven, en de pointer staat niet aan het einde van het bestand als je begint met schrijven, dan ben je data aan het *over*schrijven.

Je kunt ieder bestand openen in binaire modus, ook al is het een tekstbestand. Als je echter een tekstbestand opent in binaire modus, dan behandel je het bestand ook als een binair bestand, wat inhoudt dat Python geen automatische conversie van "newline" tekens doet.

Het sluiten van een binair bestand verschilt niet van het sluiten van een tekstbestand.

```
fp = open( "pc_rose.txt", "rb" )
fp.close()
```

De code hierboven geeft geen uitvoer – als hij wel uitvoer geeft, dan betreft het een runtime error, die hoogstwaarschijnlijk inhoudt dat "pc_rose.txt" niet beschikbaar is.

18.2 Lezen uit een binair bestand

Een binair bestand kent geen "regels." De enige manier om informatie uit een binair bestand te krijgen is de `read()` methode gebruiken. Als je `read()` zonder argument gebruikt, wordt het hele bestand gelezen (beginnend bij de pointer). Als je de methode echter een integer meegeeft als argument, dan geeft die integer het aantal bytes aan dat gelezen wordt (beginnend bij de pointer, en niet verder lezend dan het einde van het bestand).

Voor het geval de term nieuw is: een "byte" is een 8-bits teken, dat wil zeggen, een getal tussen nul en 255, opgeslagen in de kleinste geheugeneenheid die door computers ondersteund wordt. De reguliere tekens die je op het toetsenbord vindt worden alle opgeslagen in een enkele byte, en de tekens in een string zijn ook ieder één byte, maar beperkt tot een kleinere reeks getallen.

18.2.1 Byte strings

En hier treden we dan de obscure krochten van de Python taal binnen. Als je leest uit een binair bestand, dan retourneert de `read()` methode niet een reguliere string zoals je gewend bent bij het lezen uit tekstbestanden – het retourneert een "byte string." Er zijn subtiele verschillen tussen strings en byte strings. Om die te demonstreren, moet ik eerst vertellen dat je kunt aangeven dat een string een byte string is door een letter `b` voor de string te plaatsen. Dus `"Hello, world"`! is een string, terwijl `b"Hello, world"`! een byte string is.

```
hw1 = "Hello, world!"
hw2 = b"Hello, world!"

print( hw1 )
print( hw2 )
```

Het verschil tussen een string en een byte string is dat een byte string tekens kan bevatten die een string niet kan bevatten. Als je even terugdenkt aan de ASCII tabel, herinner je je wellicht dat ieder teken een getal met zich geassocieerd heeft. Je zag, bijvoorbeeld, dat "A" nummer 65 heeft, en de spatie nummer 32. De spatie was het laagste teken van de ASCII tabel dat ik liet zien, en je vraagt je je dus wellicht af wat er aan de hand is met de nummers 0 tot en met 31. Het antwoord is: dat zijn controle-tekens, en geen legale tekens die je in een string kunt zetten (op een paar uitzonderingen na). Je kunt proberen ze in een string op te nemen als een speciaal teken: als je `\x`, gevolgd door een hexadecimale code van twee hexadecimale cijfers, opneemt, representeert dat het teken met dat hexadecimale getal als nummer. Bijvoorbeeld, de hexadecimale code voor een spatie is `\x20`. Met andere woorden: `"Hello, world!"` is hetzelfde als `"Hello,\x20world!"` (ik heb dit allemaal eerder besproken in hoofdstuk 10).

```
hw1 = "Hello,\x20world!"
print( hw1 )
```

Maar wat gebeurt er als je op deze manier een illegaal teken in een string probeert op te nemen? Dan wordt zo'n teken gewoon genegeerd:

```
print( "Hello,\x00\x01\x02world!" )
```

Het probleem is dat zulke tekens kunnen voorkomen in binaire bestanden, dus je moet ze uit binaire bestanden kunnen lezen. Omdat byte strings deze tekens wel kunnen bevatten, resulteert het lezen uit binaire bestanden in byte strings.

```
print( b"Hello,\x00\x01\x02world!" )
```

Je kunt de tekens in een byte string benaderen via indices, net zoals je kunt met reguliere strings. Het verschil is dat als je een teken via een index uit een string haalt, je een letter krijgt, terwijl het halen van een teken via een index uit een binaire string een nummer oplevert. De nummers zijn codes voor de letters, die je ook krijgt als je de **ord**() functie op zo'n letter loslaat.

listing1801.py

```
hw1 = "Hello, world!"
hw2 = b"Hello, world!"

for c in hw1:
    print( c, end=" " )
print()
```

```
for c in hw1:
    print( ord( c ), end=" " )
print()
for c in hw2:
    print( c, end=" " )
```

Omdat bytes getallen zijn tussen 0 en 255, kan het voorkomen dat je een getal naar een byte string met lengte 1 wilt omzetten, of een list van getallen naar een byte string met een grotere lengte. Je kunt dat doen door een **bytes** cast op een list van zulke getallen. Let op: als je een enkel teken om wilt zetten van een getal naar een byte string via deze methode, dan moet je dat teken ook in een list zetten, al heeft die list dan maar één element. Vergeet dus niet om vierkante haken om dat ene getal te zetten, want anders is het resultaat niet wat je zou verwachten.

listing1802.py

```
bs = bytes( [72,101,108,108,111,44,32,119,111,114,108,100,33] )
print( bs )
bch = bytes( [72] )
print( bch )
fout = bytes( 72 )
print( fout )
```

Kun je een byte string omzetten naar een reguliere string. Je zou misschien denken dat een string cast het doet, maar dat werkt helaas niet:

```
hw1 = b"Hello, world!"
hw2 = str( hw1 )
print( hw2 )
```

De reden dat het niet werkt, is dat als een string gegeven is als byte string, er een codering gebruikt wordt om de string op te slaan, volgens de Unicode standaard (die ik bediscussieerde in hoofdstuk 16). Je moet de byte string "decoderen" volgens een zeker coderingsschema, meestal `"utf-8"`, omdat die het meest gebruikte Unicode formaat vastlegt. Decoderen doe je middels de `decode()` methode, met het coderingsschema als argument. Je kunt op een soortgelijke manier een string omzetten naar een byte string middels de `encode()` methode.

listing1803.py

```
hw1 = b"Hello, world!"
hw2 = hw1.decode( "utf-8" )
print( hw2 )
hw3 = hw2.encode( "utf-8" )
print( hw3 )
```

Over het algemeen is er geen reden om tekstbestanden in binaire modus te lezen, tenminste niet als je de tekst wilt benaderen, en dus hoef je je bij tekstbestanden je meestal geen zorgen te maken over codering en decodering. Een uitzondering moet gemaakt worden voor

tekstbestanden die Unicode tekens bevatten. Een dergelijk bestand kun je niet behandelen als tekstbestand, en moet je dus openen in binaire modus.

18.2.2 Demonstratie binair lezen

Om te demonstreren hoe binair lezen werkt, open ik het bestand "pc_rose.txt" in binaire modus en lees tien keer tien bytes.

listing1804.py

```
fp = open( "pc_rose.txt", "rb" )
for i in range( 10 ):
    buffer = fp.read( 10 )
    print( buffer )
fp.close()
```

Als je deze code uitvoert, zie je de tien byte strings getoond. Het valt je wellicht op dat controle tekens ook getoond worden, zoals \r en \n. De \r zou je niet zien als je het bestand opent als tekstbestand, omdat Python dit teken, samen met het opvolgende teken \n, omzet naar een enkele \n. Bovendien zou je in een reguliere string ook het teken \n niet zien, omdat het een "newline" teken is dat Python zegt dat naar de volgende regel gegaan moet worden.

Als je in plaats van een tekstbestand een echt binair bestand opent en leest, zul je waarschijnlijk weinig betekenis kunnen ontdekken in de tekens die je ziet.

18.3 Schrijven in een binair bestand

Je kunt in een binair bestand schrijven middels de `write()` methode. Het verschil met schrijven naar een tekstbestand is dat je een byte string als argument moet meegeven in plaats van een reguliere string. De volgende code creëert een binair bestand en schrijft er een tekst in.

listing1805.py

```
from os.path import getsize

NAAM = "pc_binarytest.tmp"

fp = open( NAAM, "wb" )
fp.write( b"And now for something completely different...\x0A\
\x00\x00\x00\x00\xD4\xE8\xE5\xA0\xD3\xF0\xE1\xEE\xE9\xF3\xE8\xA0\
\xC9\xEE\xF1\xF5\xE9\xF3\xE9\xF4\xE9\xEF\xEE\x00\x00\x00" )
fp.close()

print( getsize( NAAM ), "bytes geschreven" )
```

Voer deze code uit om het binaire bestand te bouwen. De code hieronder opent het in tekst modus (dat kun je doen omdat Python nergens aan kan zien dat het een binair bestand is), leest de inhoud, en toont die. Je ziet dan wat leesbare tekst en een serie onleesbare tekens.

listing1806.py

```
NAAM = "pc_binarytest.tmp"

fp = open( NAAM, encoding="latin-1" )
while True:
    buffer = fp.readline()
    if buffer == "":
        break
    print( buffer )
fp.close()
```

Opgave Wijzig bovenstaande code zodat je het bestand opent in binaire modus en de inhoud toont.

18.4 Positioneren van de pointer

Het bestand "pc_binarytest.tmp" bevat een aantal geheime woorden, die je niet kunt herkennen als je de inhoud van het bestand toont. Ik gebruik ze als illustratie bij het positioneren van de pointer.

De pointer heeft in een bestand aan waar je begint met lezen of schrijven. Je kunt de pointer verplaatsen middels de `seek()` methode. `seek()` krijgt twee integer argumenten, waarvan de tweede optioneel is. Het eerste argument is de relatieve byte positie. De tweede geeft de positie aan ten opzichte waarvan het eerste argument relatief is.

Het tweede argument is 0, 1, of 2. 0 betekent "relatief ten opzichte van het begin van het bestand." 1 betekent "relatief ten opzichte van de huidige positie van de pointer." 2 betekent "relatief ten opzichte van het einde van het bestand." Als je geen tweede argument opgeeft, wordt aangenomen dat het 0 is. In de `os` module zijn er constanten voor dit argument opgenomen: `os.SEEK_SET` is 0, `os.SEEK_CUR` is 1, en `os.SEEK_END` is 2.

Het eerste argument geeft aan hoeveel bytes je verwijderd moet zijn van de positie aangegeven door het tweede argument. Als het tweede argument 0 is, moet dit een positief getal zijn; als het 2 is, moet het een negatief getal zijn; als het 1 is, mag het negatief of positief zijn, afhankelijk van of je de pointer meer naar het begin of meer naar het einde wilt bewegen. Bijvoorbeeld, `fp.seek(5)` is gelijk aan `fp.seek(5,0)`, en beweegt de pointer naar een positie 5 bytes verwijderd vanaf het begin van het bestand, op de zesde byte (de eerste byte die gelezen zal gaan worden als je de `read()` methode aanroept).

Als je wilt weten waar de pointer gepositioneerd is, kun je de `tell()` methode gebruiken. Zowel `seek()` als `tell()` werken ook voor tekstbestanden, maar zijn dan niet erg nuttig.

De geheime boodschap begint op positie 50, en is 23 tekens lang. De codering is zo gemaakt dat als je 128 aftrekt van de byte waardes, je de getallen krijgt die je met de **`ord`**`()` functie in de juiste letters kunt omzetten. Dus zo krijg je de boodschap te lezen:

listing1807.py

```
fp = open( "pc_binarytest.tmp", "rb" )
print( "1. Huidige positie van de pointer is", fp.tell() )
fp.seek( 50 )
```

```
print( "2. Huidige positie van de pointer is", fp.tell() )
buffer = fp.read( 23 )
print( "3. Huidige positie van de pointer is", fp.tell() )
fp.close()

print( buffer )
s = ""
for c in buffer:
    s += chr( c-128 )
print( "De geheime boodschap is:", s )
```

De `seek()` methode is vooral nuttig als je een bestand opent in "lezen plus schrijven" modus (`"r+b"`). Je kunt ermee door het bestand bewegen en lezen wat je moet lezen, en (over)schrijven waar dat nodig is.

Opgave Open "pc_binarytest.tmp" in binaire "lezen plus schrijven" modus, en overschrijf de gecodeerde boodschap met de vertaling. Als je het bestand weer gesloten hebt, open het dan in tekst modus, lees de inhoud, en toon die. Als je het correct hebt gedaan, zie je twee leesbare regels. Als je het bestand per ongeluk kapot maakt, kun je het altijd opnieuw creëren.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Binaire bestanden voor lezen, schrijven, en lezen plus schrijven
- Binaire `read()` en `write()`
- Byte strings
- Conversie tussen strings en byte strings via `encode()` en `decode()`
- `seek()` en `tell()` methodes

Opgaves

Opgave 18.1 Creëer een eenvoudig encryptie programma. Open een bestand in binaire modus, en lees het byte voor byte in. Tel 128 op bij iedere byte die een waarde kleiner dan 128 heeft, en trek 128 af van iedere byte die een waarde groter dan 128 heeft. Overschrijf de byte met de nieuwe waarde. Test het programma op een kopie van een tekstbestand (zorg ervoor dat het een kopie is, want de file wordt onherroepelijk gewijzigd). Test de inhoud van het aangepaste bestand: dat moet een rommeltje zijn geworden. Maar als je het programma een tweede keer draait, zou je het originele bestand terug moeten krijgen. Zo niet, dan zit er een fout in je programma. Ben je niet blij dat je met een kopie gewerkt hebt?

Opgave 18.2 De veertien meest gebruikte letters in de Engelse taal zijn: `etaoinshrdlcum` (in het Nederlands zijn het de letters `etanirodslgvhk`). Schrijf een tekst compressie programma dat op dit feit gebaseerd is. Het programma slaat de genoemde letters op in halve bytes. Een halve byte kan de getallen 0 tot en met 15 bevatten. Als je alleen de getallen 1 tot en met 15 gebruikt, kan ieder van deze getallen één van de genoemde letters representeren, en kun je de 15 gebruiken voor de spatie. Je kunt dan dus twee van deze letters (of spatie) opslaan in één byte (de waarde van die byte zou dan 16 keer de waarde van de eerste letter plus de waarde van de tweede letter zijn). Als je in de tekst een letter of teken aantreft dat niet bij deze vijftien hoort, dan geef je dat aan door halve byte met waarde 0 op te slaan, gevolgd door de hele byte die het niet-geëncrypte teken bevat. In deze opzet is het mogelijk dat de hele byte verdeeld wordt over halve bytes van twee opeenvolgende bytes, namelijk de tweede helft van de ene byte, en de eerst helft van de tweede byte. Je mag zelf kiezen of je het programma schrijft met de Nederlandse of de Engelse letters (het verschil in het programma is maar één regel), maar in de rest van deze beschrijving en in mijn oplossing gebruik ik de Engelse letters.

Hint: een eenvoudige manier om dit probleem aan te pakken is het bouwen van een list van "half-bytes." Voor de tekens die het meest voorkomen, gebruik je de index-waarde van de string `"etaoinshrdlcum "` plus 1 (dat geeft een waarde tussen 1 en 15; merk op dat het laatste teken van de string de spatie is). Voor de andere tekens sla je drie half-bytes op, namelijk nul, gevolgde door de waarde van de byte gedeeld door 16 (naar beneden afgerond), gevolgd door de waarde van de byte modulo 16. Als de half-byte-list klaar is, maak je er een byte list van door steeds paren half-bytes te nemen, en de eerste met 16 te vermenigvuldigen en de tweede erbij optellen. Die list kun je dan naar een byte string omzetten via een **`bytes`**`()` cast.

Om dit te testen: de string `"Hello, world!"`, die 13 tekens lang is, wordt, als je de hierboven beschreven procedure volgt (die van de `e` een 1 maakt, van de `t` een 2, etcetera), een byte string met 11 tekens: `b'\x04\x81\xbb@,\xf0wI\xba\x02\x10'`.

Om deze vertaling van `"Hello, world!"` naar de gegeven byte string iets beter uit te leggen (zie afbeelding 18.1): Je herinnert je wellicht dat een hexadecimale representatie van een byte bestaat uit twee hexadecimale cijfers, dat wil zeggen, ieder cijfer past in een half-byte. Met die informatie kun je begrijpen hoe de vertaling gedaan is. De eerste byte van de vertaling is `\x04`, dus de eerste half-byte is nul. Dat betekent dat het eerste teken van de originele string letterlijk is opgeslagen, dus bestaat uit de tweede half-byte van `\x04`, en de eerste half-byte van de volgende byte, die `\x81` is. Dat is de byte `\x48`. Als je de hexadecimale code 48 opzoekt in de ASCII tabel (die je vindt in hoofdstuk 10), zie je dat het de letter `H` representeert. De volgende half-byte is de tweede half-byte van `\x81`, dus 1. Omdat dit geen nul is, is het een van de veel voorkomende tekens, namelijk de eerste, de letter `e`. Zo zie je dus hoe `"Hello, world!"` gecomprimeerd wordt als de gegeven byte string. In de byte string zie je een paar tekens die niet als hun hexadecimale code zijn weergegeven; als je wilt weten wat hun hexadecimale code is, kun je die opzoeken in de ASCII tabel.

half bytes	0	4	8	1	B	B	4	0	2	C	F	0	7	7	4	9	B	A	0	2	1	0
letters	H		e		l	l	o		,			w			o	r	l	d	!			
bytes	\x04		\x81		\xBB		@		,		\xF0		w		I		\xBA		\x02		\x10	

Afb. 18.1: Compressie voorbeeld.

Ondanks de lange beschrijving, kan dit programma geschreven worden in minder dan 30 regels code, inclusief commentaar, lege regels, en tests.

Opgave 18.3 Als vervolg op de vorige opgave, schrijf je nu een decompressie programma voor de geproduceerde strings.

Hint: Je doet gewoon het omgekeerde van wat je in de vorige opgave deed: bouw de half-byte-list opnieuw. Die list kun je dan gemakkelijk vertalen naar de originele string.

Opgave 18.4 Hoewel dit hoofdstuk over binaire bestanden gaat, werden in de vorige twee opgaves geen binaire bestanden gebruikt. Er valt gewoon niet veel te oefenen met binaire bestanden: de problemen bij dit soort bestanden betreffen het behandelen van bytes, en dat is wat de vorige twee opgaves deden. Maar om te completeren wat deze twee opgaves begonnen, kun je nu een programma schrijven dat tekstbestanden comprimeert en decomprimeert.

Schrijf een programma dat vraagt om een invoerbestand, dat moet bestaan, en een uitvoerbestand, dat niet mag bestaan. Daarna vraagt het programma of je wilt comprimeren of decomprimeren. Als je ervoor kiest om te comprimeren, wordt het invoerbestand gecomprimeerd volgens de bovengenoemde methode, en als uitvoerbestand weggeschreven. Als je ervoor kiest om te decomprimeren, wordt het invoerbestand gedecomprimeerd onder de aanname dat het eerder gecomprimeerd is middels de bovengenoemde methode, en als uitvoerbestand weggeschreven. Dus je zou het originele tekstbestand weer terug moeten kunnen krijgen door eerst te comprimeren en dat te decomprimeren.

Je doet er goed aan eerst het hele bestand in het geheugen te lezen voordat je gaat (de)comprimeren, zodat je niet in de problemen komt als de byte string in een halve byte eindigt in plaats van in een hele byte na compressie. Je doet er ook goed aan zowel het invoerbestand als het uitvoerbestand als binaire bestanden te behandelen.

Hoofdstuk 19

Bitsgewijze Operatoren

Hoofdstuk 18 besprak het omgaan met binaire bestanden. Wanneer je binaire bestanden gebruikt, ben je niet langer bezig met tekens en getallen, maar je werkt met bytes. Om informatie op het niveau van bytes te verwerken, biedt Python een aantal zogeheten "bitsgewijze operatoren." Je hebt deze operatoren niet vaak nodig, maar als je binaire bestanden gaat bewerken kunnen ze van pas komen.

19.1 Bits en bytes

Een bit is de kleinste data-eenheid die een computer kan manipuleren. Een enkele bit kan slechts twee verschillende waardes aannemen, namelijk 1 en nul.

Hoewel "prehistorische" computers inderdaad geprogrammeerd werden door direct enen en nullen te manipuleren, werden al snel computers geïntroduceerd die groepjes bits behandelden. De kleinste eenheid wat dat betreft is de "byte," die bestaat uit 8 bits. Vandaag de dag worden computertalen nog steeds afgesteld op het behandelen van bytes, hoewel de meeste computers tegenwoordig grotere hoeveelheden bytes tegelijkertijd manipuleren (het meest gebruikelijk zijn computers die 32-bits of 64-bits data-eenheden gebruiken).

19.1.1 Binair tellen

Een byte bestaat uit 8 bits, die je kunt tonen als een serie enen en nullen, bijvoorbeeld `11010010`. Op deze manier kan een byte een getal in binaire code representeren. Als een byte een positief geheel getal voorstelt, kun je dat getal berekenen door de meest rechtse bit met 1 te vermenigvuldigen, de bit ernaast met 2, de bit daarnaast met 4, etcetera, en al dit waardes bij elkaar op te tellen. Bijvoorbeeld, het binaire getal `11010010` is `1*128+1*64+0*32+1*16+0*8+0*4+1*2+0*1`, wat 210 is. Merk op dat dit equivalent is met het berekenen van de waardes van decimale getallen, waarbij het rechtse cijfer vermenigvuldigd wordt met 1, het cijfer ernaast met 10, het cijfer daarnaast met 100, etcetera, en dan alle uitkomsten worden opgeteld. Het is ook equivalent met het hexadecimaal tellen, dat ik besprak in hoofdstuk 10.

Als de bits genummerd worden, is het de gewoonte om de meest rechtse bit van een binair getal nummer nul te geven, en de nummering te laten toenemen naar links; dus de rechterbit is nummer 0, de bit ernaast nummer 1, de bit daarnaast nummer 2, etcetera. De reden is dat de rechterbit de waarde 2^0 representeert (wat gelijk is aan 1), de bit ernaast de waarde 2^1, de bit daarnaast 2^2, etcetera.

Byte	1	1	0	1	0	0	1	0	
Nummer van de bit	7	6	5	4	3	2	1	0	
Gerepresenteerde waarde	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
Byte waarde	2^7	$+\,2^6$	$+\,0$	$+\,2^4$	$+\,0$	$+\,0$	$+\,2^1$	$+\,0$	$= 210$

Opgave Schrijf een programma dat van een binaire string, die bestaat uit 8 enen en nullen, berekent welk decimaal getal erdoor gerepresenteerd wordt. De mooiste oplossing gebruikt een loop, een vermenigvuldigingsfactor, en een totaal. Het totaal start bij 0. De vermenigvuldigingsfactor begint bij 1, en wordt met 2 vermenigvuldigd iedere keer dat de loop doorlopen wordt. De loop doorloopt de string van rechts naar links (of de omgekeerde string van links naar rechts). Als een teken gevonden wordt dat een "1" is, telt het de vermenigvuldigingsfactor op bij het totaal. Dit geeft uiteindelijk het gevraagde getal.

Het laagste getal dat een byte kan representeren is 00000000, wat gelijk is aan nul. Het hoogste is 11111111, wat gelijk is aan 255. Er zijn dus 256 verschillende waardes die door een byte gerepresenteerd kunnen worden.

19.1.2 Codering van tekens

De meest basale manier om tekens te coderen is via de ASCII tabel, die ik toonde in hoofdstuk 10. In die tabel staan ook voor alle tekens de hexadecimale codes. Het is je wellicht opgevallen dat de codes liepen van (hexadecimaal) 20 tot en met 7E. De codes onder de 20 worden gebruikt voor speciale controle tekens (zoals de "newline"). Code 7F wordt gewoonlijk gebruikt om de `Del` toets te representeren. Andere codes worden niet gebruikt, wat betekent dat alle ASCII tekens middels 7 bits kunnen worden weergeven, ofwel de 8-bit sequenties 00000000 tot en met 01111111.

Hoewel computers de byte gebruiken als de basis manier om data vast te leggen, gebruikt ASCII slechts 128 van de 256 tekens die in een byte zouden kunnen worden opgeslagen. De 128 byte codes die niet door ASCII worden gebruikt, hebben alle een 1 als de linkerbit. Het ligt voor de hand dat er teken coderingen zijn die een teken toekennen aan alle 256 waardes die een byte kan bevatten. Een typische manier van zo'n codering is `latin-1`, die ik in hoofdstuk 16 noemde. Helaas gebruiken niet alle coderingen dezelfde tekens voor byte waardes 128 tot en met 255. Maar alle manieren van codering die vandaag de dag in gebruik zijn hebben wel dezelfde basis ASCII tekens in de byte waardes 0 tot en met 127.

Python is gebaseerd op de Unicode codering. Meer specifiek gebruikt het UTF-8 als coderingsmechanisme (die ik eerder noemde in hoofdstukken 10 en 16). UTF-8 werkt als volgt:

- Een byte die een 0 heeft als meest linker bit is een ASCII teken.
- Een byte die een 1 heeft als meest linker bit is de start van een sequentie van meerdere bytes die samen één teken representeren. De sequentie bestaat uit een "leidende byte" (de meest linker byte) en één of meer "volgende bytes."

- Bij een multi-byte sequentie leeft de "leidende byte," van links naar rechts, een aantal bits met waarde 1, gevolgd door een bit met waarde 0, gevolgd door een aantal resterende bits. De totale lengte van de multi-byte sequentie is het aantal bits met waarde 1 links van de meest linkse 0 in de leidende byte. Bijvoorbeeld, als de leidende byte de waarde $1110xxxx$ heeft (waarbij iedere x een byte is), dan is de hele multi-byte sequentie drie bytes in lengte. Dit is inclusief de leidende byte. De minimum lengte van de sequentie is twee bytes, en de maximum lengte is zes bytes (in dat laatste geval is de leading byte $1111110x$).
- Iedere "volgende byte" heeft 10 als de meeste linkse twee bits.
- In de praktijk is UTF-8 codering beperkt to een maximum van 4-byte sequenties, en sommige van de 4-byte sequenties zijn uitgesloten.

Dit alles betekent dat UTF-8 een zeer groot aantal verschillende tekens kan representeren. De wijze van coderen betekent ook dat sommige bit-patronen geen UTF-8 tekens representeren. Hoewel ieder bitpatroon een legale string in `latin-1` codering is, is het mogelijk om bitpatronen te construeren die geen legale UTF-8 coderingen zijn. Hierdoor kunnen die vervelende `UnicodeDecodeErrors` ontstaan bij het lezen van tekstbestanden.

19.1.3 Coderen van getallen

De manier waarop getallen als bitpatronen zijn opgeslagen is wat merkwaardig, maar over het algemeen hoef je je daar niet druk om te maken. Je moet weten dat positieve gehele getallen altijd gecodeerd zijn als multi-byte patronen, die een 0 als meest linkerbit hebben. De rest van het patroon is zoals je zou verwachten, en zoals ik hierboven heb uitgelegd.

Negatieve getallen zijn echter op een andere manier gecodeerd. Ze gebruiken het zogenaamde "twee-complement systeem." Om een negatief geheel getal te coderen, neem je eerst de absolute waarde van dat getal (dat wil zeggen, de positieve versie). Van dat getal neem je het bitpatroon, en "flipt" alle bits, met andere woorden, je maakt een 1 van iedere 0, en een 0 van iedere 1. Tenslotte tel je numeriek 1 op bij het resulterende bitpatroon. Daardoor heeft een bitpatroon dat een negatief getal representeert altijd een 1 als de meest linkerbit.

Bijvoorbeeld, om -1 te coderen neem je eerst het bitpatroon van 1, dus ...00000001. Je flipt alle bits, wat je ...11111110 geeft. Tenslotte tel je 1 op bij het resultaat, wat ...11111111 oplevert. Dus -1 wordt gecodeerd als een sequentie van alleen maar enen.

Wat betreft gebroken getallen: die worden opgeslagen in de wetenschappelijke notatie, waarbij een deel van het multi-byte patroon als exponent gebruikt wordt.

De reden dat ik dit alles uitleg, is om aan te geven dat als je met bitpatronen in Python gaat spelen, en je die patronen als getallen behandelt, je het beste kunt werken met alleen positieve integers, omdat de bitpatronen van die getallen gemakkelijk geïnterpreteerd kunnen worden.

19.2 Manipulatie van bits

Python biedt een aantal operatoren die de manipulatie van data items op het niveau van bits toestaan. Dit zijn de volgende:

```
<<      shift links
>>      shift rechts
&       bitsgewijze and
|       bitsgewijze or
~       bitsgewijze not
^       bitsgewijze exclusieve or
```

Ze worden als volgt gebruikt.

19.2.1 Shift

Als je een data item hebt, kun je de << en >> operatoren gebruiken om de bits naar links of naar rechts te verschuiven. `x<<y` verschuift de bits van `x` `y` posities naar links, waarbij het bitpatroon aan de rechterkant met 0-bits wordt aangevuld. `x>>y` verschuift de bits van `x` `y` posities naar rechts, waarbij aan de linkerkant de meest links bit iedere keer gekopieerd wordt, en de bits aan de rechterkant verwijderd worden. `x` en `y` moeten beide getallen zijn.

Bijvoorbeeld, het uitroepteken (`!`) heeft decimale code 33, die je binair schrijft als 00100001. Als je dit patroon één positie naar links verschuift, krijg je 01000010, wat decimaal 66 is, en wat de code is voor de hoofdletter B. Je kunt de verschuiving ongedaan maken door het bitpatroon voor B één positie naar rechts te verschuiven.

```
code = "!"
print( chr(ord(code)<<1) )
code = "B"
print( chr(ord(code)>>1) )
```

Het valt je misschien op dat het verschuiven van een bitpatroon van een getal met één positie naar links neerkomt op het verdubbelen van het getal, terwijl het verschuiven van één positie naar rechts neerkomt op het halveren van het getal (waarbij naar beneden wordt afgerond). En inderdaad is het zo dat het verdubbelen van een getal gelijk is aan het plaatsen van een 0 aan de rechterkant van het bitpatroon, terwijl het halveren gelijk is aan het verwijderen van de rechterbit.

```
print( "345 vervierVoudigd levert", 345<<2 )
print( "345 gedeeld door 8 levert", 345>>3 )
```

19.2.2 Bitsgewijze `and`

De bitsgewijze `and` operator (`&`) neemt twee bitpatronen, en produceert een nieuw bitpatroon dat bestaat uit alleen maar nullen, behalve op de posities waar beide originele bitpatronen een 1 hebben, die in het resultaat dan ook een 1 zijn. Bijvoorbeeld, als de originele patronen het getal 11 (00001011) en het getal 6 (00000110) zijn, dan geeft de bitsgewijze `and` operator het patroon 00000010, wat het getal 2 representeert.

```
print( 11 & 6 )
```

Opgave De bitsgewijze `and` is een gemakkelijke manier om een positief getal modulo een macht van 2 te nemen. Bijvoorbeeld, als je een getal modulo 16 wilt nemen, is dat gelijk aan het toepassen van een bitsgewijze `and` met 15, wat 00001111 is. Controleer dat 345 modulo 32 gelijk is aan `345 & 31`.

19.2.3 Bitsgewijze `or`

De bitsgewijze `or` operator (`|`) neemt twee bitpatronen, en produceert een nieuw bitpatroon dat bestaat uit alleen maar enen, behalve op de posities waar beide originele bitpatronen een 0 hebben, die dan ook in het resultaat een 0 zijn. Bijvoorbeeld, als de originele patronen het getal 11 (00001011) en het getal 6 (00000110) zijn, dan geeft de bitsgewijze `or` operator het patroon 00001111, wat het getal 15 representeert.

```
print( 11 | 6 )
```

Opgave Om de waarde van een enkele bit in een patroon op 1 te zetten (dit wordt vaak genoemd het "zetten van een bit") kun je de bitsgewijze `or` toepassen met een patroon dat bestaat uit alleen maar nullen, met uitzondering van een enkele 1 op de plek waar je de bit wilt zetten. Een gemakkelijke manier om een bitpatroon te maken met alleen de juiste bit gezet, is te starten met het getal 1, en dan de shift-links operator te gebruiken om dit ene bit zover naar links te schuiven als nodig is. Neem een getal en zet de bit met index 7 (dus de achtste bit vanaf rechts geteld) op 1.

19.2.4 Bitsgewijze `not`

De bitsgewijze `not` operator (`~`) wordt voor een bitpatroon geplaatst, en produceert een nieuw bitpatroon dat alle bits van het originele patroon "geflipt" heeft (dus iedere 0 wordt een 1, en iedere 1 wordt een 0). Bijvoorbeeld, als het originele patroon het getal 11 (00001011) is, dan produceert de bitsgewijze `not` het patroon 11110100, wat het getal -12 is. Als je je afvraagt waarom het -12 is en niet -11: dat komt door het twee-complement systeem dat ik hierboven heb uitgelegd. Maak je er maar niet druk over.

```
print( ~11 )
```

Opgave Om een enkele bit in een patroon op 0 te zetten, kun je de bitsgewijze `and` gebruiken met een patroon dat bestaat uit alleen maar enen, met uitzondering van een 0 op de plek waar je de bit op 0 wilt zetten. Een gemakkelijke manier om een dergelijk bitpatroon te maken, is te beginnen met het getal 1, de shift-links operator te gebruiken om die 1 te verschuiven naar de positie die je op 0 wilt zetten, en dan het patroon te inverteren middels de bitsgewijze `not` operator. Neem een getal en zet de bit met index 3 (de vierde bit van rechts) op 0.

19.2.5 Bitsgewijze `xor`

De bitsgewijze `exclusieve or`, of "xor," operator (`^`) neemt twee bitpatronen, en produceert een nieuw bitpatroon dat een 0 heeft op alle posities waarbij de originele patronen

dezelfde waarde hebben, en een 1 op alle posities waar de originele bitpatronen verschillend zijn. Bijvoorbeeld, als de originele patronen het getal 11 (00001011) en het getal 6 (00000110) zijn, dan geeft de bitsgewijze `xor` operator het patroon 00001101, wat het getal 13 is.

```
print( 11 ^ 6 )
```

Opgave De bitsgewijze `xor` operator geeft een gemakkelijke manier om getallen te versleutelen. Neem een bitpatroon en noem dat het "masker." Pas dat masker toe op een getal via de `xor`. Dat geeft een nieuw getal, dat het versleutelde getal is. Iemand die het masker niet kent, kan het originele getal niet herleiden. Maar iemand die het masker wel kent, kan het originele getal eenvoudigweg terugkrijgen door het masker opnieuw toe te passen op het versleutelde getal. Probeer dit.

19.2.6 Afhandelingsvolgorde van bitsgewijze operatoren

Waarschuwing: de afhandelingsvolgorde van bitsgewijze operatoren is *niet* dat ze afgehandeld worden vóór de andere operatoren. Zorg ervoor dat je haakjes gebruikt om de operatoren op de juiste volgorde toe te passen als je bitsgewijze operatoren gebruikt in een berekening. Bijvoorbeeld, je denkt misschien dat `1<<1 + 2<<1` hetzelfde is als `1*2 + 2*2`, maar in werkelijkheid wordt het uitgevoerd als `(1<<(1+2))<<1`, ofwel `1*8*2`.

```
print( 1<<1 + 2<<1 )
print( (1<<1) + (2<<1) )
```

19.3 Het nut van bitsgewijze operaties

Alles wat je met bitsgewijze operatoren doet, kun je ook met de gebruikelijke berekeningsmethodes doen, waarbij de gebruikelijke methodes het voordeel hebben dat ze veel meer kunnen. Dus waarom zou je bitsgewijze operatoren gebruiken?

Bitsgewijze operatoren werken ontzettend snel. Veel, veel sneller dan de gebruikelijke methodes. Dus moet je ze gebruiken in berekeningen als dat kan? Het antwoord is "nee," en wel om de volgende twee redenen:

- Python is slim genoeg om te herkennen dat sommige berekeningen middels bitsgewijze operatoren kunnen worden uitgevoerd, en doet intern die conversie al voor je.
- Als je echt verlegen zit om hele snelle code, kun je beter helemaal geen Python gebruiken.

Een andere reden die ik vaker genoemd hoor voor het gebruik van bitsgewijze operatoren, is dat ze het mogelijk maken boolean waardes op te slaan in een hele kleine data ruimte. Bijvoorbeeld, als ik acht booleans moet opslaan kan ik dat doen in een tuple met die acht booleans, wat minimaal acht bytes ruimte kost, of ik kan ze alle acht opslaan in één byte middels bitsgewijze operatoren. Maar in de computers van tegenwoordig is de besparing

van ruimte niet erg belangrijk meer, dus alleen als je spreekt over enorme data verzamelingen zou je je over ruimte zorgen moeten gaan maken.

Dus wat is het nut van bitsgewijze operatoren? Als je het mij vraagt, hebben ze erg weinig nut, tenzij je programma's moet maken die "dicht tegen de machine" moeten werken. Soms zijn er data structuren die je het beste met bitsgewijze operatoren kunt afhandelen. Bitsgewijze operatoren kunnen ook zinvol zijn bij het manipuleren van binaire bestanden.

Om een voorbeeld te geven: kleuren zijn vaak gecodeerd in drie bytes, namelijk een rood, een groen, en een blauw kanaal. Het nummer van een kleur is dus een getal van drie bytes. Bitsgewijze operatoren zijn een natuurlijke manier om de drie kanelen van elkaar te scheiden uit het kleur-nummer. Hier is een functie die dat doet:

listing1901.py

```
def getRGB( color ):
    blauw = color & 255
    groen = (color >> 8) & 255
    rood = (color >> 16) & 255
    return rood, groen, blauw

r, g, b = getRGB( 223567 )
print( "rood={}, groen={}, blauw={}".format( r, g, b ) )
```

Voor iemand die met kleur-coderingen werkt, leest een dergelijke functie erg natuurlijk.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Binair tellen
- Codering van tekens en getallen
- Bitsgewijze operatoren <<, >>, &, |, ~, en ^

Opgaves

Opgave 19.1 Versleutel een string via de bitsgewijze exclusieve or (xor) en het patroon 00101010 als masker. Toon de resulterende string. Ontsleutel hem daarna en toon de ontsleutelde string, die hetzelfde moet zijn als de originele string.

Opgave 19.2 Schrijf een functie die als parameter een integer, een boolean, en een getal krijgt. De integer wordt gebruikt om booleans in op te slaan. Iedere bit in de integer representeert `True` of `False`. De bits van de integer zijn op de conventionele manier genummerd, met index 0 voor de meest rechter bit. Als de boolean die is meegegeven `True` is, zet de functie de bit die correspondeert met het getal op 1. Als de boolean die is meegegeven 0 is,

maakt de functie de bit die correspondeert met het getal 0. De functie retourneert daarna de integer.

Schrijf ook een functie die een integer en een getal als parameter krijgt, en die **`True`** retourneert als de bit die correspondeert met het getal 1 is, en anders **`False`**.

Om de functies te testen, helpt het om een extra functie te bouwen die de bits in de integer toont. Je kunt deze functie gebruik laten maken van de functie die de bit-waardes als **`True`** of **`False`** retourneert.

Hoofdstuk 20

Object Oriëntatie

De hoofdstukken tot dit punt bespraken een manier van programmeren die vaak "gestructureerd programmeren" of "imperatief programmeren" genoemd wordt. Deze aanpak betreft programma's die bestaan uit een sequentie van statements, beslissingen, en loops. Je kunt ieder programmeerprobleem oplossen middels deze aanpak. In de laatste decennia echter zijn er andere programmeer "paradigma's" ontwikkeld, die helpen bij het ontwerpen en implementeren van grote programma's. Een van de meest succesvolle paradigma's is "object oriëntatie," en de meeste moderne programmeertalen ondersteunen dit paradigma. Python is eigenlijk een object-georiënteerde taal.

Hoewel object oriëntatie een natuurlijke manier biedt om problemen en oplossingen te bezien, is het behoorlijk lastig om een object georiënteerd programma te ontwerpen. De reden dat het lastig is, is dat je het probleem dat je onderhanden hebt moet doorgronden in al zijn aspecten, voordat je begint met programmeren. Voor de meer complexe problemen kan dit behoorlijk overweldigend zijn, zeker als je weinig ervaring hebt met programmeren. Maar voor de grotere problemen moet je zowiezo veel tijd besteden aan het analyseren van je oplossing, en de object georiënteerde aanpak kan dan behulpzaam zijn bij het creëren van die oplossing. Je zult ook ontdekken dat de meeste modules object georiënteerd zijn opgezet, en dat object oriëntatie ook nuttig kan zijn bij de aanpak van kleinere problemen.

Omdat object oriëntatie een erg breed onderwerp is, ga ik er meerdere hoofdstukken aan besteden, waarvan dit de eerste is. Dit hoofdstuk bediscussieert de basis van object oriëntatie, en laat de meer gespecialiseerde (en krachtige!) aspecten van object oriëntatie over aan latere hoofdstukken.

20.1 De object georiënteerde wereld

Ik typ dit terwijl ik zit aan mijn keukentafel. Naast mij staat een fruitschaal. In de schaal liggen appels. Deze appels delen bepaalde eigenschappen, maar hebben ook verschillen. Ze delen hun naam, hun prijs, en hun leeftijd, maar ze hebben alle (iets) verschillende gewichten. Op de schaal liggen ook peren. Net als de appels zijn ze een soort fruit, maar ze hebben ook een hoop verschillen met appels: verschillende namen, verschillende kleuren, verschillende bomen waar ze aan groeien. Toch delen ze ook dingen met appels die alle

fruitsoorten delen, en waarin ze verschillen van, bijvoorbeeld, de tafel waar ik aan zit. Bijvoorbeeld, ik kan appels eten, en ik kan ook peren eten. Maar ik ga niet proberen een tafel te eten.

Als ik probeer mijn wereld te modelleren in een computerprogramma, moet ik objecten modelleren: objecten als appels, peren, en tafels. Sommige van die objecten hebben een hoop zaken gemeen, bijvoorbeeld, iedere appel deelt veel eigenschappen met iedere andere appel. Het klinkt daarom zinvol om een klasse "appel" te definiëren, die de eigenschappen bevat die alle appels delen, en dan voor iedere individuele appel alleen die eigenschappen in te vullen waarin ze verschillen van alle andere appels. Hetzelfde kan ik doen voor peren, die hun eigen klasse "peer" moeten krijgen. En hoewel "appels" en "peren" van elkaar verschillen, delen ze ook eigenschappen die mij het gevoel geven dat ik ze een gedeelde klasse moet geven: de klasse "fruit." Ieder object dat thuishoort in de klasse fruit heeft in ieder geval de eigenschap dat ik het kan eten. Wat betekent dat iedere appel niet alleen behoort tot de klasse "appel," maar ook tot de klasse "fruit" – net als de "peren."

Als ik er goed over nadenk: ik kan meer eten dan alleen appels en peren. Ik kan ook taart eten. En champignons. En brood. En drop. Dus misschien heb ik nog een andere klasse nodig, waartoe ook de klasse "fruit" behoort. De klasse "voedsel," misschien?

Waart dit toe leidt, is dat als ik probeer de wereld (of een deel ervan) te modelleren, ik objecten moet modelleren – en in plaats van ieder object apart te modelleren, kan ik beter klassen van objecten definiëren, zodat ik generieke uitspraken kan doen over groepen objecten. Ik kan spreken over relaties tussen klassen, en ik kan functies definiëren die op klassen werken; bijvoorbeeld, ik kan een functionaliteit "eten" definiëren die op ieder object werkt dat behoort tot de klasse "voedsel," en die tot gevolg heeft dat het object verwijderd wordt uit de wereld en dat de "voedingsstoffen" van het object toekent aan de "eter." Omdat ik objecten kan eten als ze horen tot de klasse "voedsel," kan ik "fruit" eten. En omdat ik "fruit" kan eten, kan ik objecten eten die tot de klasse "appel" behoren.

In essentie is een computerprogramma een model van een deel van de wereld. En als zodanig profiteren veel programma's van de mogelijkheid om te kunnen werken met objecten, klassen, relaties, en functionaliteiten (methodes) die werken met objecten.

20.1.1 Studenten, docenten, en cursussen

Veel programma's moeten omgaan met personen. De studentenadministratie moet omgaan met studenten, die personen zijn. De studenten volgen cursussen, die gedoceerd worden door docenten, die ook personen zijn. Ongetwijfeld slaat de administratie gegevens op over studenten en docenten, en je mag veronderstellen dat de programmeur die de studentenadministratie heeft geïmplementeerd slim genoeg was om een enkele interface te gebruiken om de gegevens van personen in te voeren.

Wat voor data wordt door alle personen gedeeld, voor zover het de studentenadministratie aangaat? Waarschijnlijk hebben alle personen een voornaam en een achternaam. Ze hebben ook een adres. Ze hebben ook een leeftijd en een geslacht. Om ze uniek te maken, geeft de studentenadministratie iedere persoon een administratienummer (ANR). Deze data elementen zijn alle "eigenschappen" of "attributen" van "personen."

Ik noemde als eigenschappen voornaam, achternaam, adres, leeftijd, geslacht, en administratienummer. Eén van deze eigenschappen is eigenlijk meer een functie dan een eigenschap. Zie je welke?

Het antwoord is "leeftijd." Leeftijd wordt berekend middels de geboortedatum en de huidige datum. Je kunt leeftijd voorstellen als eigenschap, maar het is een eigenschap die iedere keer opnieuw berekend moet worden als je hem nodig hebt. Je kunt hem niet als een vaste waarde opslaan, omdat hij morgen anders kan zijn dan vandaag, terwijl alleen de datum is veranderd. Als ik een klasse `Persoon` ontwerp die een persoon modelleert, kan ik daarom het beste "geboortedatum" een eigenschap maken, terwijl ik "leeftijd" implementeer als methode. Zoals je je wellicht herinnert is een methode een functie die behoort bij een zeker data type: als een data type `Persoon` gedefinieerd is, is `geboortedatum` een attribuut van dat data type, terwijl `leeftijd()` een methode is van het data type die de leeftijd van de persoon retourneert als integer.

Studenten en docenten zijn allebei personen. Ze delen de eigenschappen van de klasse `Persoon`. Toch zijn er verschillen. Docenten krijgen bijvoorbeeld een salaris, maar studenten niet. Studenten kunnen daarentegen cijfers krijgen voor cursussen, maar docenten niet; zij doceren de cursussen. Hieruit kan ik de volgende twee zaken afleiden:

- Hoewel studenten en docenten beide personen zijn, hebben ze duidelijke verschillen; daarom heb ik naast de klasse `Persoon` ook een klasse `Student` en een klasse `Docent` nodig, die beide worden afgeleid van de klasse `Persoon`.
- "Cursussen" lijken een inherent onderdeel te zijn van de wereld van de studentenadministratie, dus wellicht heb ik ook een klasse `Cursus` nodig.

Als ik van `Cursus` een klasse heb gemaakt, worden relaties duidelijk. Studenten hebben een relatie met meerdere cursussen, en docenten ook, maar op een andere manier. Studenten "schrijven zich in" voor cursussen. Het lijkt er dus op dat er een methode `inschrijven()` moet komen, die ervoor zorgt dat een student een relatie krijgt met een cursus. De vraag is: moet `inschrijven()` een methode zijn van `Student`, met de cursus als argument, of een methode van `Cursus`, met de student als argument? Wat denk je?

Het antwoord is: "dat hangt ervan af." Het hangt ervan af hoe je de student beziet in het licht van de studentenadministratie. Voor mij als docent voelt het natuurlijk aan om `inschrijven()` een methode te maken van een cursus, omdat ik een cursus zie als een verzameling studenten. Maar er is niks op tegen om een student te bezien als een entiteit die een verzameling cursussen bevat. Je kunt er ook voor kiezen om `inschrijven()` als een methode van zowel studenten als cursussen te zien, of een andere klasse te definiëren die de methode `inschrijven()` bevat met zowel de student als de cursus als argumenten.

Dit illustreert hoe moeilijk de object georiënteerde visie op programmaontwerp kan zijn: door de klassen te definiëren die de wereld modelleren waarmee het programma werkt, moet je keuzes maken die een grote invloed hebben op de exacte aanpak van het programma. Zwakke keuzes leiden tot implementatieproblemen. Je moet flink wat tijd besteden aan het ontwerpen van het object georiënteerde model dat de basis vormt voor het programma, en alle consequenties van je keuzes proberen te overzien. Zelfs voor doorgewinterde programmeurs is dat een moeilijke taak. Maar een solide object-georiënteerd model maakt programma's gemakkelijk te lezen, te begrijpen, en te onderhouden. Het object georiënteerde paradigma is vaak de moeite waard.

20.1.2 Klassen, objecten, en hierarchiën

In de object-georiënteerde wereld behoort iedere onderscheidbare entiteit tot een klasse (Engels: "class"). Een klasse is een generiek model voor een groep entiteiten. De klasse

beschrijft alle attributen die de entiteiten gemeen hebben, en beschrijft de methodes die de klasse aanbiedt waarmee de wereld buiten de klasse invloed op de klasse kan uitoefenen.

Op zichzelf is een klasse geen entiteit. Een entiteit die behoort tot een klasse, wordt een "object" genoemd. De terminologie is dat een object een "instantie" (soms: "instantiatie," maar dat is geen correct Nederlands) is van een bepaalde klasse. Een klasse beschrijft attributen, maar een object dat een instantie is van de klasse geeft waardes aan de attributen. En hoewel een klasse de methodes beschrijft die hij ondersteunt, kun je een methode alleen uitvoeren voor een object dat een instantie is van de klasse.

Een klasse is een data type, een object is een waarde.

Klassen bestaan in hierarchiën. Een generieke, hoog-niveau klasse kan eigenschappen en methodes beschrijven die gedeeld worden door verschillende sub-klassen. Een sub-klasse kan eigenschappen en methodes toevoegen, en kan zelfs eigenschappen en methodes wijzigen (maar kan over het algemeen niet eigenschappen of methodes verwijderen, en moet dat ook nooit doen). Iedere sub-klasse kan zelf ook weer sub-klassen hebben.

Bijvoorbeeld, de klasse `Appel` kan een sub-klasse zijn van de klasse `Fruit`, die weer een sub-klasse kan zijn van de klasse `Voedsel`. Dit betekent dat waar je in een programma een object nodig hebt dat een instantie is van de klasse `Voedsel`, je niet alleen objecten mag gebruiken die directe instanties van `Voedsel` zijn, maar ook objecten die een instantie zijn van `Fruit` of van `Appel`. Dat werkt niet andersom: als je bijvoorbeeld een functie hebt geschreven die een object van de klasse `Appel` nodig heeft, kun je die niet gebruiken met instanties van `Fruit`, of instanties van andere sub-klassen van `Fruit`. Hoewel een `Appel` `Fruit` is, is `Fruit` geen `Appel`. Je kunt geen `Appel`s met `Peer`en vergelijken.

Een klasse hiërarchie wordt geïmplementeerd met behulp van een mechanisme dat "inheritance" ("overerving") wordt genoemd, wat het onderwerp is van hoofdstuk 22.

20.1.3 Klassen en data types in Python

De meeste object-georiënteerde programmeertalen kennen een aantal basale data types, en staan je toe om klassen te definiëren, wat neerkomt op het definiëren van nieuwe data types. Dit gold ook voor Python tot en met versie 2. Sinds Python 3 is echter ieder data type een klasse.

Je kunt dit deels zien aan de manier waarop een groot aantal functionaliteiten van basale data types in Python geïmplementeerd zijn als methodes. Een methode wordt altijd aangeroepen via de syntax `<variabele>.<methode>()`, in tegenstelling tot functies, die worden aangeroepen als `<functie>( <variabele> )`. Het feit, bijvoorbeeld, dat als je een kleine-letter-versie van een string wilt creëren, je dit effectueert middels `<string>.lower()` geeft al aan dat een string een instantie van een klasse is.

Maar niet alleen strings zijn instanties van klassen: integers en floats zijn dat ook. Ze hebben zelfs methodes, maar die worden zelden expliciet aangeroepen. Methodes worden vaak wel impliciet aangeroepen, bijvoorbeeld, als je twee getallen optelt met $+$, is dat eigenlijk de aanroep van een methode. Ik zal dit bediscussiëren in hoofdstuk 21.

20.2 Object oriëntatie in Python

Nu ik de basis filosofie van object oriëntatie heb uitgelegd, kan ik beschrijven hoe object oriëntatie werkt in Python. Het begint met het creëren van nieuwe klassen middels het gereserveerde woord **class**.

20.2.1 class

Een "class" (ik zal vanaf dit punt meestal het woord "class" gebruiken in plaats van "klasse," omdat het refereert aan het gereserveerde woord dat Python gebruikt) kun je beschouwen als een nieuw data type. Wanneer een class gecreëerd is, kun je instanties van de class toekennen aan variabelen. Om eenvoudig te beginnen, zal ik een class creëren die een punt in een 2-dimensionale ruimte beschrijft. Ik noem dit de class `Punt` (class benaming kent dezelfde eisen als de benaming van variabelen, en het is de conventie dat de naam van een class begint met een hoofdletter). Het creëren van de class `Punt` in Python is ongelofelijk eenvoudig:

```
class Punt:
    pass
```

Het gereserveerde woord **pass** in de class definitie betekent "doe niks." Dit gereserveerde woord kun je overal gebruiken waar je een commando moet plaatsen, maar waar je nog niks hebt om er neer te zetten. Je mag het niet leeg laten, of alleen een commentaar regel schrijven. Maar zodra je statements toevoegt, kun je het woord **pass** verwijderen.

Om een object te creëren dat een instantie van de class is, ken ik aan een variabele de naam van de class toe, met haakjes erachter, alsof het de aanroep van een functie is (je kunt argumenten tussen de haakjes zetten, maar dat bediscussieer ik later in dit hoofdstuk).

```
class Punt:
    pass

p = Punt()
print( type( p ) )
```

Natuurlijk is een punt meer dan alleen een object. Een punt heeft een x en een y coördinaat. Omdat Python "soft types" gebruikt, kun je waardes aan attributen toekennen om ze te creëren. Dit doe je in een speciale initialisatie methode in de class.

20.2.2 __init__()

De initialisatie methode van een class heeft de naam `__init__` (twee "underscores," gevolgd door het woord `init`, gevolgd door nog twee "underscores"). Zelfs als je de methode `__init__()` niet expliciet voor een class definieert, bestaat hij toch. Je kunt de `__init__()` methode gebruiken om alles te initialiseren waarvan je wilt dat het bestaat bij het instantiëren van de class.

In het geval van `Punt`, moet de `__init__()` methode ervoor zorgen dat iedere instantie van `Punt` een `x` en een `y` coördinaat heeft. Dit implementeer je als volgt:

listing2001.py

```
class Punt:
    def __init__( self ):
        self.x = 0.0
        self.y = 0.0

p = Punt()
print( "({}, {})".format( p.x, p.y ) )
```

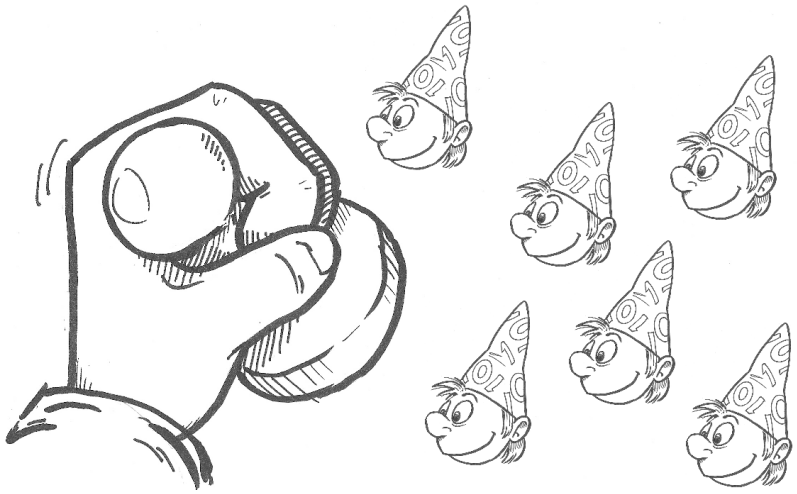
Bestudeer bovenstaande code goed. Je ziet dat de `__init__()` methode net zo gedefinieerd is als je een functie zou definiëren, maar binnen de class definitie.

`__init__()` krijgt één parameter, die `self` genoemd is. Iedere methode die je definieert krijgt altijd minstens één parameter, die gevuld wordt met een referentie aan het object waarvoor je de methode aanroept. Het is de gewoonte dat deze eerste parameter altijd `self` genoemd wordt. Dat is niet verplicht, maar iedereen doet het zo (zelfs in het Nederlands). Als je vergeet parameters op te nemen, krijg je een runtime error. Als je vergeet `self` op te nemen als eerste parameter, maar je hebt wel andere parameters gespecificeerd, dan zal Python de eerste van je parameters vullen met een referentie naar het object, en krijg je daarna waarschijnlijk alsnog een runtime error (omdat je niet had verwacht dat dat zou gebeuren).

In de `__init__()` methode voor `Punt`, krijgt de instantie van `Punt` twee attributen, die je kunt beschouwen als variabelen die een deel van het object zijn. Deze worden `x` en `y` genoemd, en omdat ze een deel zijn van het object, refereer je aan ze als `self.x` en `self.y`. Ze krijgen beide de waarde 0.0, wat betekent dat ze floats zijn.

Om aan deze attributen te refereren als het object eenmaal bestaat, gebruik je de syntax `<object>.<attribuut>`, zoals je kunt zien in de laatste regel van bovenstaande code, waar het zojuist gecreëerde object getoond wordt middels een **`print`**`()` statement.

Je vraagt je misschien af of je alleen attributen kun creëren in de `__init__()` methode. Het antwoord is: nee, je kunt attributen ook creëren in andere methodes, en zelfs buiten de definitie van de class.



```
class Punt:
    def __init__( self ):
        self.x = 0.0
        self.y = 0.0

p = Punt()
p.z = 0.0
print( "({}, {}, {})".format( p.x, p.y, p.z ) )
```

De meeste Python programmeurs (mijzelf incluis) vinden wat er gebeurt in de code hierboven bijzonder lelijk. Het is een goed gebruik om alle attributen die je aan een object wilt toekennen uitsluitend te creëren in de `__init__()` methode (hoewel je hun waardes elders kunt wijzigen), zodat je weet dat iedere instantie van de class deze attributen heeft, en geen instantie méér dan deze heeft.

Als je een versie van een class wilt hebben met extra attributen, kun je "inheritance" gebruiken om een nieuwe class te creëren op basis van de bestaande class, die deze extra attributen heeft. Ik beschrijf dat in een toekomstig hoofdstuk. Vooralsnog moet je ervoor zorg dragen dat iedere class alle benodigde attributen gedefinieerd krijgt in de `__init__()` methode.

Net als andere methodes kan de `__init__()` methode argumenten krijgen. Je kunt die argumenten gebruiken om (sommige van) de attributen een waarde te geven. Bijvoorbeeld, als ik een instantie van `Punt` onmiddellijk waardes wil geven voor de `x` en `y` coördinaten, kan ik de volgende class definitie gebruiken:

listing2002.py

```
class Punt:
    def __init__( self, x, y ):
        self.x = x
        self.y = y

p = Punt( 3.5, 5.0 )
print( "({}, {})".format( p.x, p.y ) )
```

`__init__()` heeft nu drie parameters. De eerste is nog steeds `self`, aangezien dat altijd de eerste parameter moet zijn. De tweede en derde heten respectievelijk `x` en `y`. Ik had ze ook anders mogen noemen (binnen de beperkingen die gelden voor variabele namen), maar ik koos `x` en `y` als de meest voor-de-hand-liggende benamingen. Ik ken `x` toe aan `self.x`, en `y` aan `self.y`.

Ik roep nu de instantie van een punt aan met waardes voor de `x` en `y` coördinaten als argumenten. Het eerste argument komt bij de methode binnen als de tweede parameter, en het tweede argument als de derde parameter, aangezien de eerste parameter altijd gebruikt wordt om de referentie naar het object zelf door te geven.

Als je het meegeven van zulke argumenten optioneel wilt maken, kun je de parameters default waardes geven middels een assignment in de parameter specificatie. Dat doe je als volgt:

listing2003.py

```
class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y

p1 = Punt()
print( "({}, {})".format( p1.x, p1.y ) )

p2 = Punt( 3.5, 5.0 )
print( "({}, {})".format( p2.x, p2.y ) )
```

Opgave Creëer een list van alle punten met integer coördinaten, waarbij zowel de x als de y coördinaat waardes tussen 0 en 3 kunnen aannemen.

20.2.3 `__repr__()` en `__str__()`

In de code hierboven toon ik de attributen van punten. Maar wat gebeurt er als ik het punt zelf probeer te printen?

listing2004.py

```
class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y

p = Punt( 3.5, 5.0 )
print( p )
```

Ik neem aan dat je het dan met me eens bent dat het getoonde resultaat (in feite de plaats in het computergeheugen waar het object zich bevindt) weinig informatief is. Als ik een punt print, wil ik de coördinaten zien. Python biedt een voorgedefinieerde methode waarmee ik dat kan regelen, namelijk de methode `__repr__()`. `__repr__()` moet een string retourneren, en die string wordt getoond als geprobeerd wordt het object te tonen.

listing2005.py

```
class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

p = Punt( 3.5, 5.0 )
print( p )
```

Dat ziet er een stuk beter uit.

Python kent nog een tweede methode om een string versie van een object te creëren, namelijk `__str__()`. `__str__()` is hetzelfde als `__repr__()`, maar wordt alleen gebruikt als het object geprint wordt, of als argument gebruikt wordt in de **`format`**`()` methode. Als `__str__()` niet gedefinieerd is, wordt `__repr__()` in plaats ervan gebruikt (maar niet vice versa). Als `__str__()` wel gedefinieerd is, dan kun je ervoor zorgen dat iets anders gebeurt wanneer het object getoond wordt middels de **`print`**`()` methode, en wanneer het getoond wordt op andere manieren.

Je denkt nu misschien: "welke andere manieren?" De belangrijkste "andere manier" om objecten te tonen is in de command shell, als je alleen de naam van de variabele ingeeft die het object bevat.

Het is de gewoonte dat de `__repr__()` methode een string retourneert die ieder detail van een object bevat, zodat je (indien nodig) aan de hand van deze string het object opnieuw zou kunnen instantiëren, terwijl `__str__()` een string retourneert die een netjes geformatteerde, goed leesbare versie van de meest belangrijke informatie van een object bevat. In veel gevallen kunnen deze strings hetzelfde zijn.

Veel programmeurs negeren `__repr__()` en definiëren alleen `__str__()`. Ik denk dat dat verkeerd om is: je zou altijd `__repr__()` moeten definiëren, terwijl `__str__()` optioneel is. Als je `__repr__()` gebruikt, zorg er dan voor dat je alle details van een object retourneert. Als je besluit om sommige details weg te laten, kun je beter `__str__()` gebruiken.

Opgave Breid de class `Punt` uit met een `kleur` attribuut, waarbij `kleur` een getal is tussen 0 en $2^{24} - 1$. Zorg ervoor dat de kleur in de `__init__()` methode een waarde krijgt, en in de `__repr__()` methode geretourneerd wordt.

20.3 Methodes

Ik heb al drie methodes geïntroduceerd, namelijk `__init__()`, `__repr__()`, en `__str__()`. Dit zijn voorgedefinieerde methodes die iedere class heeft. Omdat ze door de ontwikkelaars van Python gedefinieerd zijn, hebben ze excentrieke namen die beginnen en eindigen met een dubbele underscore. Er zijn nog meer van zulke methodes, die ik in latere hoofdstukken aan de orde laat komen.

Je kunt ook je eigen methodes definiëren voor een class. Zulke methodes krijgen namen die lijken op de namen van functies, en die dezelfde conventies volgen: ze beginnen met een kleine letter, en als er meerdere woorden zijn, hebben ze underscores tussen of hoofdletters voor de eerste letter van ieder tweede en volgende woord. De prefix `is` wordt gebruikt voor methodes die een **`True`**/**`False`** statement maken over een object, de prefix `get` wordt gebruikt om een waarde uit een object te krijgen, en de prefix `set` wordt gebruikt om een waarde in een object te zetten.

Bijvoorbeeld, voor een punt kan ik de methode `afstand_tot_oorsprong()` creëren, die berekent hoe ver het punt afligt van het punt (0,0).

listing2006.py

```
from math import sqrt

class Punt:
    def __init__( self, x=0.0, y=0.0 ):
```

```
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )
    def afstand_tot_oorsprong( self ):
        return sqrt( self.x*self.x + self.y*self.y )

p = Punt( 3.5, 5.0 )
print( p.afstand_tot_oorsprong() )
```

Je kunt ook methodes maken die een object wijzigen. Bijvoorbeeld, een "translatie" van een punt is een verschuiving van een punt langs een vector, dat wil zeggen, over een bepaalde afstand in horizontale en een bepaalde afstand in verticale richting. De methode `translatie()` krijgt twee argumenten (naast `self`, uiteraard), die de horizontale en verticale verschuivingen vastleggen.

listing2007.py

```
from math import sqrt

class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )
    def translatie( self, shift_x, shift_y ):
        self.x += shift_x
        self.y += shift_y

p = Punt( 3.5, 5.0 )
p.translatie( -3, 7 )
print( p )
```

Zoals je ziet heb ik geen retourwaarde voor `translatie()` gedefinieerd (die had ik niet nodig), maar de `translatie()` methode wijzigt de coördinaten van het punt, en effectueert daarbij de translatie actie.

Opgave Breid de class `Punt` uit met een methode die een punt wijzigt in het spiegelpunt ten opzichte van de oorsprong, dat wil zeggen, inverteer de tekens van de coördinaten, bijvoorbeeld: (3,4) wordt (-3,-4) en (-1,2) wordt (1,-2).

20.4 Nesten van objecten

Objecten kunnen worden opgenomen in andere objecten. Bijvoorbeeld, een rechthoek kun je definiëren als een punt dat de linkerbovenhoek aangeeft, een breedte, en een hoogte. De class `Rechthoek` wordt dan als volgt gedefinieerd:

listing2008.py

```
class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

class Rechthoek:
    def __init__( self, punt, breedte, hoogte ):
        self.punt = punt
        self.breedte = breedte
        self.hoogte = hoogte
    def __repr__( self ):
        return "[{},w={},h={}]".format( self.punt, self.breedte,
            self.hoogte )

p = Punt( 3.5, 5.0 )
r = Rechthoek( p, 4.0, 2.0 )
print( r )
```

In deze definitie bevat een `Rechthoek` object een `Punt` object.

Opgave Creëer een andere versie van de `Rechthoek` class, die vastgelegd is middels de punten in de linkerbovenhoek en de rechteronderhoek.

20.4.1 Kopieën en referenties

Hieronder staat een kopie van de code hierboven, met een paar extra regels die `Punt p` wijzigen.

listing2009.py

```
class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

class Rechthoek:
    def __init__( self, punt, breedte, hoogte ):
        self.punt = punt
        self.breedte = breedte
        self.hoogte = hoogte
    def __repr__( self ):
        return "[{},b={},h={}]".format( self.punt, self.breedte,
            self.hoogte )
```

```
p = Punt( 3.5, 5.0 )
r = Rechthoek( p, 4.0, 2.0 )
print( r )

p.x = 1.0
p.y = 1.0
print( r )
```

Je ziet dat doort het wijzigen van `p`, de `Rechthoek r` ook gewijzigd wordt. Het punt dat in de rechthoek is opgenomen, is feitelijk een aliasvan het punt dat was meegegeven als argument aan de `__init__()` methode bij het creëren van de rechthoek. Net als lists, dictionaries, en sets, worden alle objecten die instanties zijn van een zelf-gedefinieerde class, doorgegeven als referentie aan functies en methodes. Dus wordt `Rechthoek r` gecreëerd met een relatie met `Punt p`. Op deze manier kun je relaties tussen objecten representeren.

Dat is niet altijd iets wat je wilt. Het is onwaarschijnlijk dat je een `Rechthoek` object een relatie wilt laten hebben met het punt dat fungeert als de linkerbovenhoek. Hoe kun je dat oplossen? Je kunt het oplossen door een kopie van het object te creëren. Dat kan middels de `copy` module. Zoals ik eerder heb aangegeven, produceert de `copy()` functie van de `copy` module een ondiepe kopie; als je een diepe kopie wilt maken, moet je de `deepcopy()` functie gebruiken. Voor een `Punt` is dat niet nodig, omdat er geen verschil is tussen een ondiepe en een diepe kopie van instanties van deze class.

listing2010.py

```
from copy import copy

class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

class Rechthoek:
    def __init__( self, punt, breedte, hoogte ):
        self.punt = copy( punt )
        self.breedte = breedte
        self.hoogte = hoogte
    def __repr__( self ):
        return "[{},b={},h={}]".format( self.punt, self.breedte,
            self.hoogte )

p = Punt( 3.5, 5.0 )
r = Rechthoek( p, 4.0, 2.0 )
print( r )

p.x = 1.0
p.y = 1.0
print( r )
```

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Klassen en objecten
- Het gereserveerde woord **`class`**
- Creëren van objecten
- `__init__()`, `__repr__()`, en `__str__()`
- Methodes
- Nesten van objecten
- Relaties

Opgaves

Opgave 20.1 Creëer een versie van de class `Rechthoek` die "veilig" is door te garanderen dat zowel breedte als hoogte positieve waardes zijn (hoe je dat doet laat ik aan jou over). Breid de class uit met methodes die de oppervlakte en omtrek berekenen. Schrijf ook een methode die de rechteronderhoek retourneert als een `Punt`. Creëer tenslotte een methode die een twee `Rechthoek`en als parameter meekrijgt, en die het deel waar de twee rechthoeken overlappen retourneert als een rechthoek (deze laatste methode is veel moeilijker te schrijven dan de andere).

Opgave 20.2 Een student heeft een voornaam, een achternaam, een geboortedatum (bestaande uit jaar, maand, en dag, of geïmplementeerd als een `datetime` object als je de moeite neemt om de `datetime` module te bestuderen), en een administratienummer. Een cursus heeft een naam en een nummer. Studenten kunnen zich inschrijven voor cursussen. Creëer een class `Student` en een class `Cursus`. Creëer een aantal studenten en een aantal cursussen. Schrijf iedere student in voor een paar cursussen. Toon een lijst van studenten, die hun nummer, voornaam, achternaam, en leeftijd toont, en per student alle cursussen waarvoor de student is ingeschreven.

Hoofdstuk 21

Operator Overloading

"Operator overloading"[22] is een ongelofelijk krachtige techniek die object oriëntatie meedraagt, waarmee je op een natuurlijke manier nieuwe data types (classes) in een programma kunt integreren. Operator overloading is altijd gebaseerd op de definitie van een aantal speciale methodes, die de typische `__<naam>__()` structuur hebben.

21.1 Het idee achter operator overloading

Als je Python programma's schrijft voor basale data types, dan gebruik je operatoren als optellen, aftrekken, vermenigvuldigen, en delen, net zoals operatoren voor het maken van vergelijkingen en allerlei andere standaard functionaliteiten. Zulke interacties zijn niet gedefinieerd voor classes die je zelf maakt, maar Python staat het wel toe dat je definieert wat er moet gebeuren als een programmeur probeert een operator toe te passen op één van jouw classes. Dit wordt "operator overloading" genoemd.

Bijvoorbeeld, stel dat je een class definieert die een representatie is van een quaternion.[23] Je weet dat het optellen en vermenigvuldigen van quaternionen goed gedefinieerde operaties zijn. Daarom zou je wellicht willen definiëren wat er gebeurt als je twee van je quaternionen met elkaar verbindt middels de + operator. Python staat je toe dat vast te leggen. Python staat je toe wat de + operator doet voor iedere willekeurige class die je implementeert.

Is dat niet geweldig? Nu kun je een class `Student` definiëren, en ervoor zorgen dat als twee studenten met een + bij elkaar worden geteld, hun leeftijden bij elkaar worden opgeteld. Prachtig, niet?

Nee, dat is helemaal niet prachtig. Het is overduidelijk een zinloze operatie om twee studenten op te tellen. Je kunt proberen te verzinnen wat een natuurlijke interpretatie zou

[22] Ik ken hier geen Nederlandse vertaling voor. "Overloading" betekent zoveel als "iets overdekken met iets anders."

[23] Quaternionen zijn een extensie van complexe getallen. Het zijn 4-dimensionale getallen, met een reëel deel en drie imaginaire delen, die `i`, `j`, en `k` genoemd worden, met specifieke definities voor de vermenigvuldiging van ieder van deze delen. Ik ga geen details verstrekken, omdat ze niet belangrijk zijn voor dit boek (je kunt ze opzoeken als je geïnteresseerd bent). Ik wil ze slechts gebruiken als een voorbeeld van een type getallen dat niet standaard in Python is opgenomen.

zijn van het optellen van studenten, maar je zult de conclusie moeten trekken dat wat je ook verzint, het zal altijd iets onlogisch zijn. Je moet niet proberen een optelling te definiëren voor klassen waarbij de optelling niet op een natuurlijke manier bestaat. Eén van de gevaren van het toepassen van operator overloading is dat als je het op een onnadenkende manier doet, je programmacode nonsens kan lijken.

Maar operator overloading kent krachtige toepassingen. In de rest van dit hoofdstuk zal ik een aantal van die toepassingen introduceren. Er zijn er meer dan ik noem, maar ik zal de meest gebruikelijke aan de orde laten komen.

Overigens is operator overloading een typisch voorbeeld van "polymorphisme," een concept dat een functie toestaat verschillende resultaten te produceren op basis van de types van de argumenten. Polymorphisme wordt vaak genoemd als een van de krachtige eigenschappen van object oriëntatie.

21.2 Vergelijkingen

In hoofdstuk 20 beschreef ik dat objecten een alias van elkaar kunnen zijn, maar dat je ook echte kopieën van objecten kunt maken. Wat gebeurt er als ik probeer objecten met elkaar te vergelijken?

listing2101.py

```
class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

p1 = Punt( 3, 4 )
p2 = Punt( 3, 4 )
p3 = p1
print( p1 is p2 )
print( p1 is p3 )
print( p1 == p2 )
print( p1 == p3 )
```

Het gereserveerde woord **is** wordt gebruikt om identiteiten met elkaar te vergelijken. Omdat p3 een alias is van p1, retourneert `p1 is p3` **True**, terwijl `p1 is p2` **False** retourneert.

Het ==-teken zou een waarde-vergelijking moeten doen. Omdat p1 en p2 refereren aan hetzelfde punt in de 2-dimensionale ruimte, zou het fijn zijn als `p1 == p2` **True** zou retourneren (want dat is wat je zou verwachten van een waarde-vergelijking). Maar dat gebeurt niet. Vreemd is dat niet, aangezien Python niet kan weten hoe je de waarde van Punten moet vergelijken (althans niet zonder verdere specificatie). Daarom doet == de enige vergelijking die Python standaard kent, namelijk de vergelijking van identiteiten, dus dezelfde vergelijking als **is** doet. Je kunt Python echter vertellen hoe de waarde van twee punten vergeleken kan worden via de speciale methode `__eq__()`:

listing2102.py

```
class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )
    def __eq__( self, p ):
        return self.x == p.x and self.y == p.y

p1 = Punt( 3, 4 )
p2 = Punt( 3, 4 )
p3 = p1
print( p1 is p2 )
print( p1 is p3 )
print( p1 == p2 )
print( p1 == p3 )
```

De `__eq__()` methode vertelt Python in dit geval wat gedaan moet worden als twee objecten van het type `Punt` met elkaar vergeleken worden middels ==. Het retourneert **`True`** als de `x` en `y` coördinaten gelijk zijn, en anders **`False`**. In dit voorbeeld heeft er "operator overloading" plaatsgevonden voor de vergelijkingsoperator == door de `__eq__()` methode in te vullen.

Je kunt ook de andere vergelijkingsoperatoren "overloaden," dus de `!=`, `>`, `>=`, `<`, en `<=`:

- `__eq__()` voor gelijkheid (==)
- `__ne__()` voor ongelijkheid (!=)
- `__gt__()` voor groter dan (>)
- `__ge__()` voor groter dan of gelijk aan (>=)
- `__lt__()` voor kleiner dan (<)
- `__le__()` voor kleiner dan of gelijk aan (<=).

Als je `__eq__()` invult maar `__ne__()` niet, dan retourneert `__ne__()` automatisch het omgekeerde van wat `__eq__()` retourneert. Geen van de andere methodes heeft een dergelijke automatische interpretatie.

Je bent niet beperkt tot het vergelijken van objecten van dezelfde class. Bijvoorbeeld, ik kan een class `Quaternion` maken die een quaternion implementeert, die ik zou willen vergelijken met een integer of een float. Dat kan:

listing2103.py

```
class Quaternion:
    def __init__( self, a, b, c, d ):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
```

```
    def __repr__( self ):
        return "({},{}i,{}j,{}k)".format( self.a, self.b,
            self.c, self.d )
    def __eq__( self, n ):
        if isinstance( n, int ) or isinstance( n, float ):
            if self.a == n and self.b == 0 and \
                self.c == 0 and self.d == 0:
                return True
            else:
                return False
        elif isinstance( n, Quaternion ):
            if self.a == n.a and self.b == n.b and \
                self.c == n.c and self.d == n.d:
                return True
            else:
                return False
        return NotImplemented

c1 = Quaternion( 1, 2, 3, 4 )
c2 = Quaternion( 1, 2, 3, 4 )
c3 = Quaternion( 3, 0, 0, 0 )
if c1 == c2:
    print( c1, "==", c2 )
else:
    print( c1, "!=", c2 )
if c1 == c3:
    print( c1, "==", c3 )
else:
    print( c1, "!=", c3 )
if c3 == 1:
    print( c3, "==", 1 )
else:
    print( c3, "!=", 1 )
if c3 == 3:
    print( c3, "==", 3 )
else:
    print( c3, "!=", 3 )
if c3 == 3.0:
    print( c3, "==", 3.0 )
else:
    print( c3, "!=", 3.0 )
if c3 == "3":
    print( c3, "== \"3\"" )
else:
    print( c3, "!= \"3\"" )
if 3 == c3:
    print( 3, "==", c3 )
else:
    print( 3, "!=", c3 )
```

De implementatie van de `__eq__()` methode in de code hierboven test eerst of een vergelijking gemaakt wordt met een `Quaternion`, een integer, of een float. Als het één van deze data types is, dan wordt de vergelijking uitgevoerd, en het resultaat geretourneerd als **`True`** of **`False`**. Als het geen van de data types is, wordt `NotImplemented` geretourneerd. `NotImplemented` is een speciale waarde die aangeeft dat de vergelijking geen zinvolle uitkomst geeft. Hoewel de `__ne__()` methode het resultaat van de `__eq__()` methode inverteert, kan hij niet `NotImplemented` inverteren.

In de code hierboven zou je iets moeten opvallen aan de laatste vergelijking. De vergelijking `3 == c3` wordt uitgevoerd. Gewoonlijk wordt de vergelijkingsoperator uitgevoerd voor de linker-operand, wat in dit geval wil zeggen dat de vergelijking die gedefinieerd is voor de integer 3 wordt uitgevoerd, met `c3` als argument. Maar voor integers houdt de `__eq__()` methode geen rekening met `Quaternionen` (de makers van Python, die verantwoordelijk zijn voor de integer class, kunnen immers niet weten dat ik een `Quaternion` class zal bouwen), en dus zal de vergelijking `NotImplemented` retourneren. Als dat echter gebeurt, inverteert Python de operanden, zodat in dit geval de vergelijking `c3 == 3` wordt uitgevoerd. Dit leidt dan wel tot een resultaat, aangezien voor een `Quaternion` de vergelijking met een integer gedefinieerd is. Hetzelfde zal gebeuren voor de `!=` operator. En iets vergelijkbaars gebeurt voor de andere vergelijkingsoperatoren, maar als daar de operanden worden verwisseld, wordt ook een < omgewisseld met een >, en een <= met een >=, zoals je zou verwachten.

Opgave In hoofdstuk 20 werd een class `Rechthoek` gedefinieerd. Voeg aan deze class operatoren toe die de gelijkheid van rechthoeken testen (twee rechthoeken zijn gelijk als ze exact dezelfde vorm hebben, zelfs als ze niet dezelfde plek in de ruimte innemen), en voeg ook operatoren toe die testen of een rechthoek kleiner of groter dan een andere rechthoek is (op basis van de oppervlaktes). Test de nieuwe operatoren. Ik wil hierbij opmerken dat ik een beetje twijfel of dit wel acceptabele definities zijn voor vergelijkingen van rechthoeken, maar als oefening kan het ermee door.

Er is nog een speciaal soort vergelijking die ik wil bespreken, en dat is het testen of een object **`True`** of **`False`** is. Veel objecten worden als **`False`** beschouwd in speciale omstandigheden; bijvoorbeeld, een lege list wordt als **`False`** geëvalueerd. Ik heb dit kort besproken in hoofdstuk 6.

```
buffer = []
if buffer:
    print( buffer )
else:
    print( "buffer is empty" )
```

Je kunt je eigen evaluatie van een object definiëren die gebruikt wordt als het object optreedt als conditie. Dit gaat via de `__bool__()` methode.

`__bool__()` wordt aangeroepen als een object als conditie behandeld wordt. Hij retourneert **`True`** of **`False`**. Als `__bool__()` niet geïmplementeerd is, wordt in plaats ervan `__len__()` aangeroepen (die bespreek ik hieronder), en zal er **`False`** geretourneerd worden als `__len__()` nul retourneert. Als noch `__bool__()` noch `__len__()` geïmplementeerd is, zal het object altijd als **`True`** beschouwd worden in een conditie.

21.3 Berekeningen

Er zijn methodes beschikbaar die definiëren wat er gebeurt als je een instantie van een class combineert met een waarde via een reguliere berekeningsoperator. De meest belangrijke zijn:

- `__add__()` voor optellen (+)
- `__sub__()` voor aftrekken (-)
- `__mul__()` voor vermenigvuldigen (*)
- `__truediv__()` voor delen (/)
- `__floordiv__()` voor integer delen (//)
- `__mod__()` voor modulo (%)
- `__pow__()` voor machtsverheffen (**)
- `__lshift__()` voor shift-links (<<)
- `__rshift__()` voor shift-rechts (>>)
- `__and__()` voor bitsgewijze **and** (&)
- `__or__()` voor bitsgewijze **or** (|)
- `__xor__()` voor bitsgewijze `xor` (^)

Bijvoorbeeld, voor quaternionen is de optelling gedefinieerd als: $(A + Bi + Cj + Dk) + (E + Fi + Gj + Hk) = (A + E) + (B + F)i + (C + G)j + (D + H)k$. Je kunt natuurlijk ook integers en floats optellen bij quaternionen. Dit kan als volgt geïmplementeerd worden:

listing2104.py

```
class Quaternion:
    def __init__( self, a, b, c, d ):
        self.a, self.b, self.c, self.d = a, b, c, d
    def __repr__( self ):
        return "({},{}i,{}j,{}k)".format( self.a, self.b,
            self.c, self.d )
    def __add__( self, n ):
        if isinstance( n, int ) or isinstance( n, float ):
            return Quaternion( n+self.a, self.b, self.c, self.d )
        elif isinstance( n, Quaternion ):
            return Quaternion( n.a + self.a, n.b + self.b, \
                n.c + self.c, n.d + self.d )
        return NotImplemented

c1 = Quaternion( 3, 4, 5, 6 )
c2 = Quaternion( 1, 2, 3, 4 )
print( c1 + c2 )
print( c1 + 10 )
```

Als een berekeningsoperator gebruikt wordt met je nieuwe class als de rechteroperand, en de linkeroperand ondersteunt deze operator niet (dat wil zeggen: de linkeroperand

retourneert `NotImplemented`), dan controleert Python of je nieuwe class de operatie ondersteunt als rechteroperand. Je moet dat mogelijk maken via extra methodes, die dezelfde namen hebben als de bovengenoemde operatoren, maar met een `r` voor de naam, bijvoorbeeld, `__radd__()` is de optelling met het object als rechteroperand (alle andere methodes kun je op dezelfde manier creëren).

De code hierboven zal een runtime error geven als je probeert `10 + c1` uit te rekenen (probeer het maar). Je moet de `__radd__()` methode implementeren om dat op te lossen.

listing2105.py

```
class Quaternion:
    def __init__( self, a, b, c, d ):
        self.a, self.b, self.c, self.d = a, b, c, d
    def __repr__( self ):
        return "({},{}i,{}j,{}k)".format( self.a, self.b,
            self.c, self.d )
    def __add__( self, n ):
        if isinstance( n, int ) or isinstance( n, float ):
            return Quaternion( n+self.a, self.b, self.c, self.d )
        elif isinstance( n, Quaternion ):
            return Quaternion( n.a + self.a, n.b + self.b, \
                 n.c + self.c, n.d + self.d )
        return NotImplemented
    def __radd__( self, n ):
        return self.__add__( n )

c1 = Quaternion( 3, 4, 5, 6 )
print( 10 + c1 )
```

Je ziet dat ik het probleem heb opgelost door `__radd__()` te implementeren als een directe aanroep van `__add__()`. Je vraagt je misschien af waarom Python dat niet automatisch doet. De reden is van wiskundige aard: het komt inderdaad vaak voor dat + "commutatief" is, wat wil zeggen dat je de operanden mag omwisselen zonder dat dat het eindresultaat beïnvloedt, maar dat is zeker niet altijd het geval. Maar als voor je nieuwe class de optelling commutatief is, dan kun je `__radd__()` implementeren door eenvoudigweg `__add__()` aan te roepen.

Voor de verkorte operatoren +=, -=, *=, etcetera, kun je ook aparte methodes definiëren. Deze hebben dezelfde namen als de methodes hierboven, maar met een `i` voor de naam, bijvoorbeeld, `__iadd__()` implementeert de += operator (wederom kun je op dezelfde manier de namen voor de andere methodes creëren). Deze methodes moeten `self` wijzigen, en ook het resultaat (meestal `self`) retourneren. Als ze niet geïmplementeerd zijn, valt Python terug op de reguliere interpretatie, dat wil zeggen, als een statement `x += y` wordt gegeven, probeert Python `x.__iadd__(y)` uit te voeren, en als dat `NotImplemented` geeft, zal het `x = x.__add__(y)` uitvoeren. Daarom hoef je meestal niet de methodes voor de verkorte operatoren te implementeren.

Opgave Breid de `Quaternion` class uit met aftrekking. Aftrekken werkt equivalent met optellen, maar alle plussen worden vervangen door minnen. Merk op dat aftrekken niet commutatief is, dus je kunt `__rsub__()` niet als een simpele aanroep van `__sub__()`

implementeren. Het is echter niet bepaald moeilijk om `__rsub__()` te implementeren, dus doe dat ook.

21.4 Eénwaardige operatoren

Eénwaardige operatoren zijn operatoren die op een enkel object werken, dus niet in combinatie met een ander object. Een typisch voorbeeld is het min-teken (-) dat je voor een getal kunt zetten om het negatief te maken. Je kunt sommige eenwaardige operatoren overloaden, evenals een aantal basale functies die op een enkel object werken.

- `__neg__()` implementeert de negatie (-) van een object
- `__pos__()` implementeert een plus-teken (+) voor een object (dit doet meestal niks)
- `__invert__()` implementeert de bitsgewijze **`not`** (~)
- `__abs__()` implementeert de absolute waarde van een object via de **`abs`**`()` functie
- `__int__()` implementeert de (naar beneden afgeronde) integer waarde van een object middels de **`int`**`()` functie; deze moet een integer retourneren
- `__float__()` implementeert de conversie naar een float van een object middels de **`float`**`()` functie; deze moet een float retourneren
- `__round__()` implementeert afronding middels de **`round`**`()` functie. Een optioneel tweede argument kan gegeven worden dat het aantal decimalen specificeert; er moet een integer of een float geretourneerd worden
- `__bytes__()` implementeert de representatie van het object als een byte string. Hierin is de methode gelijkwaardig met de `__str__()` methode die in hoofdstuk 20 beschreven werd

listing2106.py

```
class Quaternion:
    def __init__( self, a, b, c, d ):
        self.a, self.b, self.c, self.d = a, b, c, d
    def __repr__( self ):
        return "({},{}i,{}j,{}k)".format( self.a, self.b,
            self.c, self.d )
    def __neg__( self ):
        return Quaternion( -self.a, -self.b, -self.c, -self.d )
    def __abs__( self ):
        return Quaternion( abs( self.a ), abs( self.b ),
            abs( self.c ), abs( self.d ) )
    def __bytes__( self ):
        return self.__str__().encode( "utf-8" )

c1 = Quaternion( 3, -4, 5, -6 )
print( c1 )
print( -c1 )
print( abs( c1 ) )
print( bytes( c1 ) )
```

Het lijkt op het eerste gezicht handig om in de code hierboven ook de `__int__()`, `__float__()`, en `__round__()` methodes te implementeren, om respectievelijk de functies **`int`**`()`, **`float`**`()`, en **`round`**`()` toe te passen op `self.a`, `self.b`, `self.c`, en `self.d`. Helaas werkt dat niet, omdat de betreffende functies integers of floats moeten retourneren, en niet `Quaternionen`. Ik zie geen zinvolle interpretatie van **`int`**`()`, **`float`**`()`, en **`round`**`()` voor de class `Quaternion`, anders dan ik suggereerde, dus deze methodes moeten niet geïmplementeerd worden.

21.5 Sequenties

Een speciale class is de sequentie. Ik heb al een aantal sequentie classes geïntroduceerd, namelijk tuples, lists, dictionaries, en sets. Zulke classes bevatten een serie elementen, die je kunt benaderen middels een index of een key. Je kunt zelf ook een sequentie class creëren, door een aantal methodes die informatie over elementen van de class geven te overloaden.

- `__len__()` implementeert de **`len`**`()` functie, die een integer retourneert die aangeeft hoeveel elementen het object bevat.
- `__getitem__()` implementeert het retourneren van een element met de key (of index) die als argument is meegegeven. Deze methode wordt aangeroepen als het object benaderd wordt met een waarde tussen vierkante haken, bijvoorbeeld `x[key]` met `x` het object en `key` de key of index van het gezochte element. Als `key` een index is en de index valt buiten het correcte bereik, dan moet de methode een `IndexError` genereren (zie hoofdstuk 17). Als `key` iets anders is (bijvoorbeeld een sleutel voor een dictionary) en deze refereert niet aan een bestaand element, dan moet de methode een `KeyError` genereren. Als `key` een index is, dan moeten voor een complete implementatie ook zogenaamde "slice objects" ondersteund worden).
- `__setitem__()` implementeert het toekennen van een waarde aan een element van het object dat de key of index heeft die als argument is meegegeven, met de toe te kennen waarde als tweede argument. Deze methode wordt aangeroepen als een waarde wordt toegekend middels een assignment aan het object met een waarde tussen vierkante haken, bijvoorbeeld `x[key] = value`.
- `__delitem__()` implementeert het verwijderen van een element uit het object met als key of index het gegeven arument, wanneer het gereserveerde woord **`del`** wordt gebruikt, bijvoorbeeld **`del`** `x[key]`.
- `__missing__()` wordt aangeroepen door `__getitem__()`, met de key of index als argument, als deze key of index niet verwijst naar een element dat in het object bestaat. Deze methode is vooral bedoeld voor subclasses van de Python dictionary.
- `__contains__()` krijgt een element mee (en dus niet een key of index) als argument, en retourneert **`True`** als het element in het object bestaat, en anders **`False`**. Deze methode wordt aangeroepen als het gereserveerde woord **`in`** gebruikt wordt om te testen of het element in het object aanwezig is.

Om te demonstreren hoe deze methodes werken, heb ik een sequentie geïmplementeerd die een Filippine puzzel beschrijft. Deze puzzel bestaat uit een serie vragen, die ieder beantwoordt kunnen worden met één woord. Van ieder antwoord is één letter "speciaal." Als je de speciale letters achter elkaar zet, krijg je de oplossing van de puzzel.

Ik heb ieder van de puzzelwoorden gedefinieerd als een instantie van de class `FilippineWoord`, die als attributen heeft het antwoord, de index van de speciale letter in het antwoord, en de vraag. De class `Filippine` is de complete puzzel, dat wil zeggen, een sequentie van `FilippineWoorden`. Ik heb de methodes `__len__()`, `__getitem__()`, `__setitem__()`, en `__delitem__()` geïmplementeerd (de laatste twee worden in de code hieronder niet gebruikt).

listing2107.py

```
class FilippineWoord:
    def __init__( self, woord, index, vraag ):
        self.woord = woord
        self.index = index
        self.vraag = vraag

class Filippine:
    def __init__( self, naam, woorden ):
        self.naam, self.woorden = naam, woorden
    def __len__( self ):
        return len( self.woorden )
    def __getitem__( self, n ):
        return self.woorden[n]
    def __setitem__( self, n, waarde ):
        self.woorden[n] = waarde
    def __delitem__( self, n ):
        del self.woorden[n]
    def toon( self ):
        print( self.naam )
        for i in range( len( self ) ):
            print( "{}. {}".format( i+1, self[i].vraag ),
                end = "  " )
            for j in range( len( self[i].woord ) ):
                if j == self[i].index:
                    print( "* ", end="" )
                else:
                    print( "_ ", end="" )
            print()
    def oplossing( self ):
        s = ""
        for i in range( len( self ) ):
            s += self[i].woord[self[i].index]
        return s

puzzel = Filippine(
    "De Monty Python en de Heilige Graal Filippine",
    [ FilippineWoord( "ANTHRAX", 5,
          "Sir Galahad bestormde kasteel" ),
      FilippineWoord( "BORS", 2, "Een konijn doodde Sir" ),
      FilippineWoord( "TIM", 0, "De wijze tovenaar heet" ),
      FilippineWoord( "HERBERT", 0,
          "De erfgenaam van het Moeras Kasteel is prins" ),
```

```
      FilippineWoord( "ZWALUW", 4,
          "Een kokosnoot is te zwaar voor een Europese" ),
      FilippineWoord( "MINSTREELS", 5,
          "De ridders aten Robins" ) ] )

puzzel.toon()
```

Ik heb nog twee methodes geïmplementeerd, die demonstreren hoe de hierboven genoemde methodes werken. `toon()` toont de puzzel, en gebruikt als zodanig de **`len`**`()` functie en indices om de woorden te benaderen. `oplossing()` toont de oplossing van de puzzel, en gebruikt ook **`len`**`()` en indices.

Het zou overigens mooier geweest zijn als ik de sterretjes, die de speciale letters aangeven, onder elkaar had afgedrukt. Het is echter afhankelijk van de editor die je gebruikt of er een vaste letterbreedte is, en dus is het lastig om dat voor elkaar te krijgen. Je kunt hier zelf een oplossing voor implementeren als je wilt (voor dit hoofdstuk is dat niet van belang).

Er is nog een belangrijke methode die je voor een sequentie class kunt implementeren, namelijk `__iter__()`. Ik stel een discussie van deze methode echter uit tot hoofdstuk 23.

Als je een sequentie class bouwt, kun je overwegen om ook de methode `__add__()` te implementeren, en wellicht ook een goede interpretatie van de methode `__mul__()`.

Opgave Een `Zin` is een list van woorden. Een basale `Zin` class is hieronder gegeven. Implementeer de `__len__()`, `__getitem__()`, `__setitem__()`, en `__contains__()` methodes voor deze class.

listing2108.py

```
class Zin:
    def __init__( self, words ):
        self.words = words
    def __repr__( self ):
        return " ".join( self.words )

s = Zin( [ "Er", "is", "slechts", "een", "ding", "ter",
"wereld", "erger" "dan", "beroddeld", "worden", "en",
"dat", "is", "niet", "beroddeld", "worden" ] )
print( s )
print( len( s ) )
print( s[7] )
s[7] = "prettiger"
print( "beroddeld" in s )
```

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Operator overloading

- Overloaden van vergelijkingsoperatoren middels `__eq__()`, `__ne__()`, `__gt__()`, `__ge__()`, `__lt__()`, en `__le__()`
- `NotImplemented`
- `__bool__()`
- Overloaden van operatoren voor berekeningen middels `__add__()`, `__sub__()`, `__mul__()`, `__truediv__()`, `__floordiv__()`, `__mod__()`, `__pow__()`, `__lshift__()`, `__rshift__()`, `__and__()`, `__or__()`, en `__xor__()`
- Rechtshandige versies van overload methodes voor berekeningen
- Verkorte versies van overload methodes voor berekeningen
- Overloaden van eenwaardige operatoren `__neg__()`, `__pos__()`, `__invert__()`, `__abs__()`, `__int__()`, `__float__()`, `__round__()`, en `__bytes__()`
- Overloaden van operatoren voor sequentie classes `__len__()`, `__getitem__()`, `__setitem__()`, `__delitem__()`, `__missing__()`, en `__contains__()`

Opgaves

Opgave 21.1 Een speelkaart heeft een kleur (`"Harten"`, `"Schoppen"`, `"Klaveren"`, `"Ruiten"`) en een waarde (`2`, `3`, `4`, `5`, `6`, `7`, `8`, `9`, `10`, `"Boer"`, `"Vrouw"`, `"Heer"`, `"Aas"`). Implementeer een class `Kaart`. Zorg ervoor dat twee kaarten gelijk zijn als ze een gelijke waarde hebben, en dat andere vergelijkingen de volgorde van de waardes gebruiken (dus met 2 als laagste waarde en Aas als hoogste waarde). Test de nieuwe class.

Opgave 21.2 Gebruik de class `Kaart` van de vorige opgave. Creëer ook een class `Trekstapel`. Een `Trekstapel` is een sequentie van kaarten. De kaarten vormen een stapel met de laagste index voor de bovenste kaart, en de hoogste index voor de onderste kaart. Implementeer de `__len__()` en `__getitem__()` methodes. Creëer een `voegtoe()` methode die een kaart toevoegt aan de stapel aan de onderkant, en een `trek()` methode om een kaart aan de bovenkant van de stapel te verwijderen en te retourneren. Test de class.

Opgave 21.3 Gebruik de class definities die je hiervoor hebt gecreëerd, en maak twee trekstapels. De eerste bevat de Ruiten 2, de Harten Heer, en de Klaveren 7 (in deze volgorde). De tweede bevat de Harten 4, de Harten 3, en de Schoppen 8 (in deze volgorde). Laat de twee stapels het spel "Oorlogje" spelen. Dit gaat als volgt: Trek de bovenste kaart van iedere stapel. De hoogste kaart van de twee wordt onderop de stapel waar hij vandaan kwam gelegd, en vervolgens wordt ook de andere kaart onderop die stapel gelegd. Het spel gaat door totdat slechts één stapel over is.

Hint: Met deze opzet duurt het spel 13 rondes, en de eerste stapel wint (dat moet wel, want de eerste stapel bevat een kaart die nooit door een kaart van de tweede stapel verslagen kan worden). Zie je wat een saai spel "Oorlogje" is? Waarom kinderen dit zouden willen spelen – met zelfs een volledige stok kaarten – is me een raadsel.

Merk op dat normaal het spel "Oorlogje" gespeeld wordt met speciale regels die optreden als twee kaarten met dezelfde waarde getrokken worden, maar in dit geval hebben alle

kaarten een verschillende waarde dus die situatie kan niet optreden. Je hoeft dus ook geen rekening daarmee te houden, maar als je dat toch wilt doen, mag het wel.

Opgave 21.4 Implementeer een class `Fruitmand`. De `Fruitmand` bevat stukken fruit, en voor ieder stuk fruit kan het een bepaald aantal hebben. Houd het eenvoudig: sla de stukken fruit op als een dictionary, waarbij de naam van het fruit als key wordt gebruikt en het aantal als waarde. Voor deze opgave is er geen beperking op wat de naam van een stuk fruit mag zijn, iedere string is acceptabel. Implementeer de `__add__()` methode om een stuk fruit toe te voegen aan de mand (en het zou een goed idee kunnen zijn om ook de `__iadd()__` methode te implementeren), en implementeer de `__sub__()` methode om een stuk fruit te verwijderen (en `__isub__()` wellicht ook). Implementeer de `__contains__()` methode om te controleren of een bepaalde fruitsoort in de mand zit. Implementeer ook de `__getitem__()` methode om te controleren hoeveel er van een fruitsoort in de mand zit, de `__setitem__()` methode om in één keer de hoeveelheid van een fruitsoort een waarde te geven, en de `__len__()` methode om vast te stellen hoeveel verschillende soorten fruit in de mand zitten. Merk op dat het noodzakelijk is om een key uit de dictionary te verwijderen als de bijbehorende waarde 0 is.

Hoofdstuk 22

Overerving

Overerving (Engels: “inheritance”) is een mechanisme om een nieuwe class te baseren op een bestaande class, door enkel en alleen de verschillen tussen de twee aan te geven. Dit is een bijzonder krachtig concept dat het mogelijk maakt hoogst flexibele en gemakkelijk onderhoudbare programma’s te schrijven.

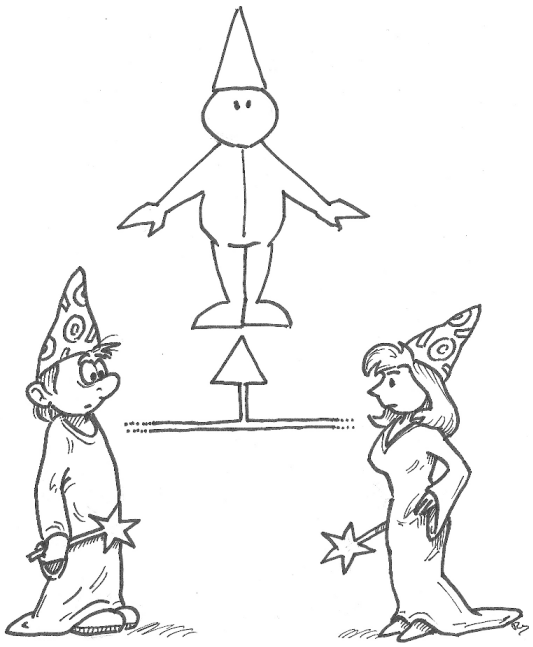
22.1 Class overerving

In hoofstuk 20 gaf ik een voorbeeld van een class `Appel` en een class `Peer`, die beide “afgeleid” waren van een class `Fruit`. Ik gaf ook een voorbeeld van een class `Student` en een class `Docent` die beide zijn afgeleid van een class `Persoon`. Je implementeert een dergelijk “hiërarchie” van classes via “overerving.”

Overerving is erg eenvoudig. Bij de definitie van een nieuwe class, zet je tussen haakjes de naam van een andere class. De nieuwe class erft dan alle attributen en methodes van de andere class, wat wil zeggen dat ze automatisch zijn opgenomen in de nieuwe class.

listing2201.py

```
class Persoon:
    def __init__( self, voornaam, achternaam, leeftijd ):
        self.voornaam = voornaam
        self.achternaam = achternaam
        self.leeftijd = leeftijd
    def __repr__( self ):
        return "{} {}".format( self.voornaam, self.achternaam )
    def minderjarig( self ):
        return self.leeftijd < 18

class Student( Persoon ):
    pass

albert = Student( "Albert", "Applebaum", 19 )
print( albert )
print( albert.minderjarig() )
```



Zoals je kunt zien, heeft de class `Student` alle attributen en methodes van de class `Persoon` geërfd. De nieuwe class (in dit geval `Student`) wordt de "subclass" genoemd, en de class waarvan de subclass wordt afgeleid (in dit geval `Persoon`) heet de "superclass" (soms "ouder-class" genoemd).

22.1.1 Uitbreiden en overschrijven

Als je een subclass wilt uitbreiden met nieuwe methodes, dan kun je gewoon die nieuwe methodes definiëren in de subclass. Als je op deze manier methodes definieert die al bestaan in de superclass, dan "overschrijven" ze de methodes van de superclass, wat wil zeggen dat bij een aanroep van de betreffende methode voor de subclass, de methode van de subclass gebruikt wordt, en niet die van de superclass.

Vaak gebeurt het dat als je een methode overschrijft, je nog steeds de methode van de superclass wilt kunnen gebruiken. Bijvoorbeeld, als de class `Student` een list bevat van de cursussen waarvoor de student is ingeschreven, dan wordt die list geïnitialiseerd in de `__init__()` methode. Dus moet ik de `__init__()` methode van `Persoon` overschrijven, waardoor de naam en leeftijd van de student niet langer geïnitialiseerd worden, tenzij ik ervoor zorg dat dat wel gebeurt. Ik zou bijvoorbeeld een kopie kunnen maken van de `__init__()` methode van `Persoon` en die kopie aanpassen voor `Student`, maar het is beter om de `__init__()` methode van `Persoon` aan te roepen vanuit de `__init__()` methode van `Student`. Op die manier zorg ik ervoor dat, mocht de `__init__()` methode van `Persoon` gewijzigd worden, ik de `__init__()` methode van `Student` niet hoef aan te passen.

Er zijn twee manieren om de methode van een andere class aan te roepen: middels een "class call" of middels de **`super`**`()` methode.

Een "class call" houdt in dat een methode wordt aangeroepen via de syntax `<classnaam>.<methode>()`. Dus om de methode `__init__()` van `Persoon` aan te roepen,

kan ik schrijven `Persoon.__init__()`. Ik ben daarbij niet beperkt tot het aanroepen van alleen superclass methodes; ik kan op deze manier methodes van iedere willekeurige class aanroepen. Omdat dit geen reguliere aanroep van een methode is, moet `self` als argument meegegeven worden. Dus, voor de code hierboven, als ik de `__init__()` methode van `Persoon` wil aanroepen vanuit de `__init__()` methode van `Student`, dan schrijf ik `Persoon.__init__( self, firstname, lastname, age )` (ik mag hier `self` gebruiken omdat iedere instantie van `Student` ook een instantie van `Persoon` is, aangezien `Student` een subclass is van `Persoon`).

Via de standaardfunctie **`super`**`()` kan ik direct aan de superclass refereren vanuit een subclass, zonder dat ik de naam van de superclass ken. Ik kan dus de `__init__()` methode van de superclass van `Student` aanroepen via **`super`**`().__init__()`. Ik hoef dan niet `self` als eerste argument mee te geven. Dus, voor de code hierboven, als ik de `__init__()` methode van `Persoon` wil aanroepen vanuit de `__init__()` methode van `Student`, dan schrijf ik **`super`**`().__init__( firstname, lastname, age )`.

Ik prefereer de tweede benadering, gebruik makend van **`super`**`()`, maar alleen op deze specifieke manier: het aanroepen van de directe superclass via overerving vanuit een enkele class. **`super`**`()` kan op diverse andere manieren gebruikt worden en heeft wat eigenaardigheden, die ik later in dit hoofdstuk zal bespreken.

In de code hieronder krijgt de class `Student` twee nieuwe attributen: een studieprogramma en een list van cursussen. De methode `__init__()` wordt overschreven om die nieuwe attributen toe te voegen, waarbij ook de `__init__()` methode van `Persoon` wordt aangeroepen. `Student` krijgt daarnaast een nieuwe methode, `inschrijven()`, om cursussen toe te voegen aan de cursus-list. Tenslotte heb ik als demonstratie de methode `minderjarig()` overschreven om studenten minderjarig te maken als ze nog geen 21 zijn (mijn excuses daarvoor).

listing2202.py

```
class Persoon:
    def __init__( self, voornaam, achternaam, leeftijd ):
        self.voornaam = voornaam
        self.achternaam = achternaam
        self.leeftijd = leeftijd
    def __repr__( self ):
        return "{} {}".format( self.voornaam, self.achternaam )
    def minderjarig( self ):
        return self.leeftijd < 18

class Student( Persoon ):
    def __init__( self, voornaam, achternaam,
        leeftijd, programma ):
        super().__init__( voornaam, achternaam, leeftijd )
        self.cursussen = []
        self.programma = programma
    def minderjarig( self ):
        return self.leeftijd < 21
    def inschrijven( self, cursus ):
        self.cursussen.append( cursus )
```

```
albert = Student( "Albert", "Applebaum", 19, "CSAI" )
print( albert )
print( albert.minderjarig() )
print( albert.programma )
albert.inschrijven( "Toepassingen van rationaliteit" )
albert.inschrijven( "Verweer tegen de zwarte kunsten" )
print( albert.cursussen )
```

22.1.2 Meervoudige overerving

Je kunt een class creëren die erft van meerdere andere classes. Dit wordt "meervoudige overerving" genoemd. Je specificeert dan alle superclasses, met komma's ertussen, tussen de haakjes van de class definitie. Die nieuwe class vormt dan een combinatie van alle superclasses.

Als een methode van een instantie van de nieuwe class wordt aangeroepen, moet Python beslissen welke methode gebruikt wordt als deze voorkomt in meerdere van de samenstellende classes. Python controleert daarvoor eerst de subclass zelf. Als de methode daar niet gevonden wordt, worden alle superclasses gecontroleerd, van links naar rechts. Zodra een specificatie van de methode gevonden wordt, wordt die uitgevoerd.

Als je de methode van een superclass wilt uitvoeren, moet je aan Python aangeven welke superclass je daarvoor wilt gebruiken. Dat gaat het beste via een "class call." Maar je kunt hier ook `super()` voor gebruiken. Dat is wel een beetje ingewikkeld. Je moet als argumenten de classes verstrekken die je wilt laten controleren, in de volgorde dat je wilt dat ze gecontroleerd worden. Het eerste argument van `super()` wordt daarbij echter overgeslagen (ik neem aan dat dat `self` zou moeten zijn).

Dit werkt dus ongeveer als volgt: Je hebt drie classes, A, B, en C. Je creëert een nieuwe class D die erft van de andere drie, via de definitie `class D( A, B, C )`. Als je in de `__init__()` methode van D de `__init__()` methodes van de drie superclasses wilt aanroepen, doe je dat via class calls `A.__init__()`, `B.__init__()`, en `C.__init__()`. Echter, als je de `__init__()` methode van slechts één ervan wilt aanroepen, en je niet precies weet van welke, maar je weet wel in welke volgorde je ze zou willen controleren (bijvoorbeeld B, C, A), dan kun je een aanroep doen via `super()` met `self` als eerste argument en de drie andere classes in de volgorde van controle (bijvoorbeeld `super( self, B, C, A ).__init__()`).

Zoals ik al zei, het is wat ingewikkeld. Meervoudige overerving is zowiezo ingewikkeld. Over het algemeen raad ik je aan om meervoudige overerving niet te gebruiken, tenzij er geen enkele andere manier is om een probleem op te lossen. Veel object georiënteerde talen ondersteunen meervoudige overerving niet eens, en de talen die dat wel doen zetten er vaak waarschuwingen bij.

Ik ga daarom geen een praktijkvoorbeeld geven van meervoudige overerving, noch zul je hieronder opgaves aantreffen die het gebruiken. Je moet het gewoon vermijden, totdat je flink veel ervaring hebt opgebouwd met Python en object georiënteerd programmeren. En als je dat eenmaal bereikt hebt, kom je waarschijnlijk gemakkelijk tot manieren om programma's te construeren die meervoudige overerving niet nodig hebben.

22.2 Interfaces

Een interface is een class die specifieke attributen en methodes vastlegt zonder een implementatie te geven van die methodes. Het idee is dat je gedwongen bent een subclass af te leiden die de methodes implementeert. Je kunt dan, zelfs als je nog niet weet welke subclasses allemaal gemaakt gaan worden, toch al functies bouwen die methodes aanroepen van de interface class, met als aanname dat deze functies worden aangeroepen met instanties van subclasses waarvoor de methodes ingevuld zijn. Om dit goed te begrijpen, kan ik beter een voorbeeld geven.

Stel dat ik een applicatie wil bouwen die werkt met voertuigen. Misschien is dit een reisplanner die berekent hoe ik van punt A naar punt B kan komen. De reisplanner heeft een kaart met daarop alle mogelijke punten en alle mogelijke verbindingen tussen naburige punten. Het heeft ook een lijst van voertuigen, waarbij sommige voertuigen gelimiteerd zijn tot bepaalde punten, en slechts beperkte andere punten verbinden (bijvoorbeeld, boten reizen alleen tussen havens). De reisplanner krijgt een start- en een eindpunt, en geeft dan een uitvoer die iets zegt als: neem de auto en reis van het startpunt naar punt X, neem dan het vliegtuig om van punt X naar punt Y te reizen, neem dan de bus om van punt Y naar punt Z te reizen, en loop tenslotte van punt Z naar het eindpunt.

Deze applicatie heeft een definitie van voertuigen nodig. Om een optimaal reisplan te maken, moet de applicatie weten welke voertuigen waar aanwezig zijn, waarheen ze kunnen reizen, en de reissnelheid (zodat je niet een plan krijgt dat zegt “loop van Amsterdam naar Moskou”). Het kan ook zinvol zijn om een werkwoord op te nemen dat beschrijft hoe je reist met een voertuig (bijvoorbeeld “rijd,” “vaar,” of “vlieg”). Je moet er goed over nadenken hoe je dergelijke voertuigen wilt implementeren. Een mogelijke aanpak is ieder voertuig een methode te geven die een beginpunt als argument krijgt en die retourneert of het voertuig op dat beginpunt aanwezig is, een methode die een eindpunt krijgt en die retourneert of dat eindpunt met dat voertuig bereikt kan worden, een methode die twee punten krijgt en die aangeeft hoeveel tijd het voertuig nodig heeft om tussen die twee punten te reizen, en een methode die het corresponderende werkwoord retourneert (ik zeg niet dat deze implementatie een goed idee is, maar het is er een die gebruikt zou kunnen worden). Dus je kunt de class `Voertuig` als volgt implementeren:

listing2203.py

```
class Voertuig:
    def __init__( self ):
        self.startpunt = []
        self.eindpunten = []
        self.werkwoord = ""
        self.naam = ""
    def __str__( self ):
        return self.naam
    def isStartpunt( self, p ):
        return NotImplemented
    def isEindpunt( self, p ):
        return NotImplemented
    def snelheid( self, p1, p2 ):
        return NotImplemented
    def reisWerkwoord( self ):
        return NotImplemented
```

Een dergelijke class heet een “interface” of “abstracte class” (in computertheorie zijn er subtiele verschillen tussen interfaces en abstracte classes, maar voor Python zijn die niet van belang). Je gebruikt deze class niet om er instanties van te creëren, wat de reden is dat alle methodes `NotImplemented` retourneren. In plaats daarvan gebruik je de class als een superclass waarvan je subclasses afleidt, waarbij de subclasses min of meer gedwongen zijn om een implementatie voor iedere van de voorgedefinieerde methodes te geven. Dit betekent dat, ongeacht het voertuig dat je als subclass creëert, de methodes van `Voertuig` altijd voor het voertuig zullen bestaan. Functies die iets doen met voertuigen mogen er dus vanuit gaan dat de betreffende methodes beschikbaar zijn.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Overerving
- Overschrijven
- Class calls
- **`super()`**
- Meervoudige overerving
- Interfaces

Opgaves

Opgave 22.1 Hieronder geef ik een class `Rechthoek` met een `x` en `y` coördinaat die de linkerbovenhoek vastleggen, en een breedte `b` en hoogte `h`. Creëer nu een class `Vierkant` die zoveel mogelijk erft van `Rechthoek`.

exercise2201.py

```
class Rechthoek:
    def __init__( self, x, y, b, h ):
        self.x = x
        self.y = y
        self.b = b
        self.h = h
    def __repr__( self ):
        return "[({},{}),b={},h={}]".format( self.x, self.y,
            self.b, self.h )
    def oppervlakte( self ):
        return self.b * self.h
    def omtrek( self ):
        return 2*(self.b + self.h)
```

Opgave 22.2 Een `Rechthoek` en een `Vierkant` zijn vormen. Er zijn, uiteraard, meerdere vormen die op verschillende manieren gedefinieerd zijn, maar die met rechthoeken en vierkanten gemeen hebben dat ze een oppervlakte en een omtrek hebben. Definieer een interface class `Vorm`, waarvan `Rechthoek` en `Vierkant` sub(sub)classes zijn. Definieer ook een class `Cirkel` die je afleidt van `Vorm`.

Opgave 22.3 Een bekend speltheoretisch probleem is het "Iterated Prisoner's Dilemma," in het Nederlands officieel halfslachtig vertaald als het "geïtereerde dilemma van de gevangene." In dit probleem spelen twee strategieën tegen elkaar over meerdere rondes. Iedere ronde kan iedere strategie kiezen tussen twee mogelijkheden (zonder daarbij te weten wat de andere strategie kiest): "Coöperatie" (C) of "Defectie" (D) (vrij vertaald: "samenwerken" of "tegenwerken"). Als beide strategieën C spelen, krijgen ze allebei 3 punten. Als ze beide D spelen, krijgen ze allebei 1 punt. Als één strategie C speelt en één strategie D, dan krijgt de strategie die C speelt 0 punten, en de strategie die D speelt 6 punten. Het doel van iedere strategie is zoveel mogelijk punten te verdienen over de gehele duur van het spel.

Hieronder is een eenvoudige versie van het "Iterated Prisoner's Dilemma" gecodeerd. Een strategie om het spel te spelen is gedefinieerd door de class `Strategie`. Het hoofdprogramma laat twee strategieën tegen elkaar spelen over 100 rondes (het is niet moeilijk om in het hoofdprogramma meer dan twee strategieën tegen elkaar te laten spelen in paren, maar dat maakt de code een stuk langer en is niet van belang voor de opgave). De class `Strategie` heeft geen implementatie voor de methode `keuze()`. Om een strategie te creëren, leid je een nieuwe class van `Strategie` af, en vul je op zijn minst de `keuze()` methode in. Optioneel mag je ook de `laatstezet()` methode invullen, en de `__init__()` methode uitbreiden.

Implementeer de volgende strategieën:

- `Random` speelt COOPERATIE of DEFECTIE per toeval
- `AltijdD` speelt altijd DEFECTIE
- `OogOmOog` start met COOPERATIE, en speelt daarna wat de tegenstander in de vorige ronde speelde (de `laatstezet()` methode krijgt altijd te zien wat de tegenstander speelde nadat een keuze gemaakt is)
- `OogOmTweeOgen` start met twee keer COOPERATIE, en speelt dan DEFECTIE als de opponent DEFECTIE speelde in de twee voorgaande rondes, en anders COOPERATIE
- `Meerderheid` start met COOPERATIE, en speelt dan wat de tegenstander speelde in de meerderheid van de voorgaande rondes

Als je nog meer strategieën wilt implementeren, mag dat natuurlijk. Test sommige strategieën uit tegen elkaar door ze in te vullen bij de toekenningen aan `strategie1` en `strategie2` (vergeet niet om ze een naam te geven tussen de haakjes).

Merk op dat de manier waarop ik de score telling heb gecodeerd (met een statement als `3 if c1 == COOPERATIE else 1`) een verkorte manier is om met Python een eenvoudige conditie te schrijven, die lijkt op list comprehension (zie hoofdstuk 12). Dat heb ik alleen gedaan om ruimte te besparen en leesbaarheid te verhogen; je mag dit natuurlijk ook doen met de 4 regels code die nodig zijn om dit in een **`if`**-statement te zetten.

exercise2203.py

```
COOPERATIE = 'C'
DEFECTIE = 'D'
RONDES = 100

class Strategie:
    def __init__( self, name="" ):
        self.name = name
        self.score = 0
    def keuze( self ):
        # Moet COOPERATIE of DEFECTIE retourneren
        return NotImplemented
    def laatstezet( self, mijnzet, opponentzet ):
        # Krijgt de laatste zet die gemaakt is, na keuze()
        pass
    def plusscore( self, n ):
        self.score += n

strategie1 = Strategie()
strategie2 = Strategie()

for i in range( RONDES ):
    c1 = strategie1.keuze()
    c2 = strategie2.keuze()
    if c1 == c2:
        strategie1.plusscore( 3 if c1 == COOPERATIE else 1 )
        strategie2.plusscore( 3 if c2 == COOPERATIE else 1 )
    else:
        strategie1.plusscore( 0 if c1 == COOPERATIE else 6 )
        strategie2.plusscore( 0 if c2 == COOPERATIE else 6 )
    strategie1.laatstezet( c1, c2 )
    strategie2.laatstezet( c2, c1 )

print( "Eind score", strategie1.name, "is", strategie1.score )
print( "Eind score", strategie2.name, "is", strategie2.score )
```

Hoofdstuk 23

Iteratoren en Generatoren

Iteratoren maken het mogelijk ervoor te zorgen dat je een zelf-gedefinieerde class kunt gebruiken in **`for ... in ...`** statements. Generatoren zijn een eenvoudige manier om iteratoren te creëren.

23.1 Iteratoren

Je hebt het **`for ... in ...`** commando regelmatig gebruikt. Het is je wellicht opgevallen dat het op allerlei verschillende manieren gebruikt wordt.

listing2301.py

```
for i in [1,2,3,4]:
    print( i, end=" " )
print()
for i in ( "pi", 3.14, 22/7 ):
    print( i, end=" ")
print()
for i in range( 3, 11, 2 ):
    print( i, end=" ")
print()
for c in "Hallo":
    print( c, end=" " )
print()
for key in { "appel":1, "banaan":3 }:
    print( key, end=" " )
```

Lists, strings, en dictionaries zijn alle "iterabelen" (een vernederlandsing van het Engelse woord "iterable"), wat betekent dat ze gebruikt mogen worden in **`for ... in ...`** statements. Vele andere objecten kunnen ook als iterabelen gebruikt worden. Je kunt ervoor zorgen dat dat ook geldt voor instanties van je eigen classes.

Een "iterator" is een object dat een nieuwe element retourneert iedere keer dat je de standaardfunctie **`next()`** aanroept met het object als argument. Als het object niks meer heeft

dat geretourneerd kan worden, genereert het een `StopIteration` exception. Als je deze exception wilt vermijden, kun je een optioneel tweede argument aan **`next()`** meegeven, dat geretourneerd wordt als de iterator niks meer heeft. Je kunt van iedere iterabele een iterator object maken middels de standaardfunctie **`iter()`**.

```
iterator = iter( ["appel", "banaan", "kers"] )
print( next( iterator, "END" ) )
print( next( iterator, "END" ) )
print( next( iterator, "END" ) )
print( next( iterator, "END" ) )
```

Je kunt iteratoren gebruiken in **`for ... in ...`** statements.

```
iterator = iter( ["appel", "banaan", "kers"] )
for fruit in iterator:
    print( fruit )
```

23.1.1 Iterabele objecten

Een object dat dient te functioneren als iterabele moet de volgende twee methodes bevatten:

- een methode `__iter__()` die de iterabele (meestal het object zelf) retourneert
- een methode `__next__()` die toegang geeft tot alle elementen die het object bevat, één voor één, en die als er geen elementen meer zijn de `StopIteration` exception genereert (in een **`for ... in ...`** loop, betekent dit dat de loop eindigt)

Je kunt alle elementen van een iterabele doorlopen met **`for ... in ...`**. Er zijn drie manieren om zulke iterabele objecten te maken. De eerste twee beginnen met de iterabele als een container die een sequentie van elementen bevat.

De eerste manier verwijdert, iedere keer als `__next__()` wordt aangeroepen, één van de elementen en retourneert het, waarna de iterabele dus één element minder bevat. Wanneer alle elementen behandeld zijn, zal het bij iedere volgende aanroep `StopIteration` genereren. Hier is een voorbeeld van zo'n iterator die de eerste tien getallen van de Fibonacci reeks retourneert.

listing2302.py

```
class Fibo:
    def __init__( self ):
        self.seq = [1,1,2,3,5,8,13,21,34,55]
    def __iter__( self ):
        return self
    def __next__( self ):
        if len( self.seq ) > 0:
            return self.seq.pop(0)
        raise StopIteration()
```

```
fseq = Fibo()
for n in fseq:
    print( n, end=" " )
```

De tweede manier houdt een index bij in de sequentie van elementen, en verhoogt die index bij iedere aanroep van `__next__()`, waarna het corresponderende element geretourneerd wordt. Wanneer de index buiten de grenzen van de sequentie komt, wordt `StopIteration` gegenereerd. Je kunt op deze manier een herbruikbare iterabele maken, als je een methode toevoegt die de index weer op nul zet.

listing2303.py

```
class Fibo:
    def __init__( self ):
        self.seq = [1,1,2,3,5,8,13,21,34,55]
        self.index = -1
    def __iter__( self ):
        return self
    def __next__( self ):
        if self.index < len( self.seq )-1:
            self.index += 1
            return self.seq[self.index]
        raise StopIteration()
    def reset( self ):
        self.index = -1

fseq = Fibo()
for n in fseq:
    print( n, end=" " )
print()
fseq.reset()
for n in fseq:
    print( n, end=" " )
```

De derde manier maakt van de iterabele niet een container met elementen, maar een object dat het volgende element berekent wanneer `__next__()` wordt aangeroepen. Een dergelijke iterabele kan eindig zijn, maar kan in principe ook een oneindige aantal elementen retourneren. Je kunt de iterabele ook opnieuw laten beginnen als je een methode toevoegt om de berekening opnieuw te initialiseren.

listing2304.py

```
class Fibo:
    def reset( self ):
        self.nr1 = 0
        self.nr2 = 1
    def __init__( self, maxnum=1000 ):
        self.maxnum = maxnum
        self.reset()
    def __iter__( self ):
        return self
```

```
    def __next__( self ):
        if self.nr2 > self.maxnum:
            raise StopIteration()
        nr3 = self.nr1 + self.nr2
        self.nr1 = self.nr2
        self.nr2 = nr3
        return self.nr1

fseq = Fibo()
for n in fseq:
    print( n, end=" " )
print()
fseq.reset()
for n in fseq:
    print( n, end=" " )
```

Waarschuwing: Wees erg voorzichtig met het bouwen van een iterabele die een oneindig aantal elementen kan retourneren. Programmeurs gaan ervan uit dat **for ... in ...** geen eindeloze loop kan veroorzaken, maar in het voorbeeld hierboven kan een eindeloze loop ontstaan als ik geen limiet aan het aantal elementen stel. Bij een dergelijke iterabele kun je het beste een verplicht maximum stellen aan het aantal elementen, wat ik in dit voorbeeld doe door de parameter `maxnum` op te nemen.

Opgave Creëer een iterator die alle kwadraten van integers tussen 1 en 10 genereert. Je mag zelf kiezen welke aanpak je volgt.

23.1.2 Gedelegeerde iteratie

In de voorbeelden hierboven werd een iterabele gecreëerd door het aanroepen van de `__iter__()` methode voor een object, dat zichzelf retourneert. Dat hoeft niet op die manier. Een iterabele mag de iteratie delegeren[24] aan een ander object, dat wordt aangemaakt door de iterabele en geretourneerd wordt als de `__iter__()` methode wordt aangeroepen.

listing2305.py

```
class FiboIterable:
    def __init__( self, seq ):
        self.seq = seq
    def __next__( self ):
        if len( self.seq ) > 0:
            return self.seq.pop(0)
        raise StopIteration()

class Fibo:
    def __init__( self, maxnum=1000 ):
        self.maxnum = maxnum
    def __iter__( self ):
```

[24] De naam "gedelegeerde iteratie" heb ik zelf bedacht. Als er een "officiële" naam voor deze werkwijze bestaat, hoor ik dat graag.

```
        nr1 = 0
        nr2 = 1
        seq = []
        while nr2 <= self.maxnum:
            nr3 = nr1 + nr2
            nr1 = nr2
            nr2 = nr3
            seq.append( nr1 )
        return FiboIterable( seq )

fseq = Fibo()
for n in fseq:
    print( n, end=" " )
print()
for n in fseq:
    print( n, end=" " )
```

Deze aanpak heeft een aantal voordelen:

- Je kunt meerdere instanties van de iterabele parallel aan elkaar uitvoeren zonder er expliciet meer dan één te creëren (omdat ze automatisch worden gecreëerd wanneer dat nodig is, dus als **`for ... in ...`** gebruikt wordt)
- Je hoeft geen methode `reset()` aan te roepen om weer van voor af aan te beginnen; iedere nieuwe aanroep van de iterabele begint weer van voor af aan
- De gedelegeerde iterabele wordt automatisch uit het geheugen verwijderd wanneer er geen elementen meer in zitten (Python geeft namelijk automatisch geheugen vrij als het gebruikt wordt door een object waaraan het programma niet langer kan refereren)

23.1.3 `zip()`

Je kunt tuples creëren die de elementen bevatten van meerdere iterabelen middels de standaardfunctie **`zip()`**. Om een eenvoudig voorbeeld te geven:

```
z = zip( [1,2,3], [4,5,6], [7,8,9] )
for x in z:
    print( x )
```

Een zip-object is een iterator, dat wil zeggen, je kunt het zip-object zelf niet printen, maar je kunt de elementen van het object doorlopen via een **`for ... in ...`** constructie. Het *i*de element van het zip-object bestaat uit de *i*de elementen van ieder van de iterabelen die als argumenten gebruikt worden. Als deze iterabelen van ongelijke lengte zijn, dan is de lengte van het zip-object gelijk aan de kortste lengte van de argumenten.

In het voorbeeld hierboven heb ik lists gebruikt als argumenten, maar je kunt iedere iterabele als argument gebruiken. Bijvoorbeeld, in de code hieronder zip ik een range, een iterator, en een list comprehension.

listing2306.py

```
class Dubbel:
    def __init__( self ):
        self.seq = [2*x for x in range( 1, 11 )]
    def __iter__( self ):
        return self
    def __next__( self ):
        return self.seq.pop(0)

seq = zip( range( 1, 11 ), Dubbel(), [3*x for x in range(1,11)] )
for x in seq:
    print( x )
```

Opgave Creëer een zip-object dat tuples met twee elementen produceert: het eerste element is een integer, lopend van 1 tot 10. Het tweede element is het kwadraat van het eerste element.

23.1.4 `reversed()`

De ingebouwde functie **`reversed()`** creëert vanuit een iterabele een iterator die de elementen van de iterator in omgekeerde volgorde verwerkt. De iterabele wordt als argument meegegeven. Niet alle iterabelen kunnen omgekeerd worden, maar de iterabelen die onderdeel zijn van standaard Python (zoals lists) kunnen het in ieder geval. Als je ervoor wilt zorgen dat een iterabele die je zelf creëert middels **`reversed()`** omgekeerd kan worden, moet je de Python documentatie bestuderen.

```
fruitlist = ["appel", "peer", "kers", "banaan"]
for fruit in reversed( fruitlist ):
    print( fruit )
```

23.1.5 `sorted()`

De ingebouwde functie **`sorted()`** creëert vanuit een iterabele een iterator die de elementen van de iterator gesorteerd verwerkt. De iterabele wordt als argument meegegeven. Er zijn daarnaast twee optionele argumenten. De eerste is `key=<key>`, waarbij `<key>` de naam is van een functie die gebruikt wordt om de key van het sorteerproces te definiëren. Dit werkt gelijk aan de `key=<key>` parameter voor de list `sort()` methode – zie hoofdstuk 12 voor meer informatie. Als geen key wordt meegegeven is de sorteervolgorde voor strings alfabetisch, en voor getallen numeriek. Voor andere data types, of gemixte data types, hangt het van de specificatie van het key argument af. Het tweede optionele argument is `reverse=<boolean>`, dat via **`True`** of **`False`** aangeeft of de sortering een omgekeerd resultaat moet geven.

```
fruitlist = ["appel", "peer", "kers", "banaan"]
for fruit in sorted( fruitlist ):
    print( fruit )
```

23.2 Generatoren

Een generator is een functie die het gedrag van een iterabel object emuleert. Over het algemeen is het korter en gemakkelijker om een generator te bouwen dan het is om een iterabele te bouwen. Diverse standaardfuncties zijn als generator geïmplementeerd, bijvoorbeeld de functie **range**().

Generatoren zijn gebaseerd op het gereserveerde woord **yield**. Als __next__() wordt aangeroepen voor een generator, wordt de functie uitgevoerd totdat **yield** wordt aangetroffen, en dan wordt de waarde geretourneerd die met de **yield** geassocieerd is. Daar aangekomen, "wacht" de functie totdat __next__() opnieuw wordt aangeroepen, waarna de functie verder gaat waar hij gebleven was totdat **yield** weer gevonden wordt. StopIteration wordt automatisch gegenereerd wanneer de functie eindigt.

Je hoeft niet expliciet __next__() en/of __iter__() te definiëren voor een generator. Een generator bestaat bij gratie van het feit dat de functie het gereserveerde woord **yield** bevat, en het geassocieerde iterabele object wordt automatisch door Python gecreëerd, inclusief correcte implementaties voor __next__() en __iter__().

listing2307.py

```
def fibo( maxnum ):
    nr1 = 0
    nr2 = 1
    while nr2 <= maxnum:
        nr3 = nr1 + nr2
        nr1 = nr2
        nr2 = nr3
        yield nr1

fseq = fibo( 1000 )
for n in fseq:
    print( n, end=" " )
print()
for n in fseq:
    print( n, end=" " )
```

23.2.1 Generator expressies

In hoofdstuk 12 introduceerde ik het concept "list comprehension." Omdat een list in een iterator veranderd kan worden, en daarom in een generator, heeft Python een gelijksoortig concept geïntroduceerd voor generatoren, en noemt dit "generator expressies." De syntax van een generator expressie is gelijk aan de syntax van een list comprehension, behalve dat er ronde in plaats van vierkante haken omheen staan.

Bijvoorbeeld, de volgende generator expressie retourneert alle kwadraten tot en met 100:

```
seq = (x*x for x in range( 11 ))
for x in seq:
    print( x, end=" " )
```

Als je de twee buitenste haakjes in de generator expressie wijzigt in vierkante haken, wordt de code uitgevoerd met `seq` als resultaat van een list comprehension. Om hierover heel duidelijk te zijn: met list comprehension wordt de hele list in één keer gegenereerd, terwijl met een generator expressie de elementen pas gegenereerd worden wanneer ze nodig zijn. Daarom is in principe een generator expressie te prefereren, aangezien die minder geheugen nodig heeft.

23.3 `itertools` module

De `itertools` module bevat een verzameling functies die geavanceerde manipulatie van iteratoren mogelijk maken. Als je dit tot het uiterste doortrekt, voorzien ze je van een soort "iterator algebra" waarmee je gespecialiseerde hulpmiddelen kunt bouwen in Python. Ik noem hier slechts een klein aantal van de basisfuncties van `itertools` die wellicht van nut kunnen zijn.

23.3.1 `chain()`

`chain()` neemt twee of meer iterabelen as argumenten en functioneert als een iterabele die de samenstellende iterabelen één voor één afwerkt.

```
from itertools import chain

seq = chain( [1,2,3], [11,12,13,14], [x*x for x in range(1,6)] )
for item in seq:
    print( item, end=" ")
```

23.3.2 `zip_longest()`

`zip_longest()` werkt net als **`zip`**`()`, maar creëert een iterabele met een lengte gelijk aan die van de langste van de samenstellende iterabelen. Je kunt een `fillvalue=` argument opgeven om aan te geven welke waarde er op de lege plekken moet komen te staan.

```
from itertools import zip_longest

seq = zip_longest( "appel", "framboos", "banaan", fillvalue=" " )
for item in seq:
    print( item )
```

23.3.3 `product()`

`product()` creëert een iterabele die alle elementen produceert van het Cartesisch product van de iterabelen die als argumenten zijn meegegeven. In iets minder wiskundige termen: als je twee iterabelen hebt, waarvan de eerste de elementen x, y, en z heeft, en de tweede de elementen a en b, produceert `product()` een iterabele met de elementen xa, xb, ya, yb, za, en zb.

```
from itertools import product

seq = product( [1,2,3], "ABC", ["appel","banaan"] )
for item in seq:
    print( item )
```

23.3.4 permutations()

permutations() krijgt een iterabele als argument, en een optioneel tweede argument dat een lengte aangeeft. Het creëert een iterabele die alle permutaties produceert die combinaties zijn van elementen van de meegegeven iterabele, met de gegeven lengte. Als geen lengte wordt meegegeven, genereert het alle permutaties van alle elementen. Als de meegegeven iterabele elementen dubbel bevat, zullen er kopieën van permutaties zijn.

```
from itertools import permutations

seq = permutations( [1,2,3], 2 )
for item in seq:
    print( item )
```

23.3.5 combinations()

combinations() krijgt een iterabele als argument, en een tweede argument dat een lengte aangeeft. Het creëert een iterabele die combinaties produceert van elementen van het eerste argument, met de gegeven lengte. De lengte is *niet* optioneel (wat logisch is, aangezien bij maximale lengte er maar één combinatie is). De elementen van de combinaties staan in de volgorde dat ze in de originele iterabele stonden. Als er elementen meerdere malen voorkomen in de originele iterabele, zullen er kopieën van combinaties zijn.

```
from itertools import combinations

seq = combinations( [1,2,3], 2 )
for item in seq:
    print( item )
```

23.3.6 combinations_with_replacement()

combinations_with_replacement() werkt net als combinations(), maar ieder element van de iterabele mag meerdere keren gebruikt worden.

```
from itertools import combinations_with_replacement

seq = combinations_with_replacement( [1,2,3], 2 )
for item in seq:
    print( item )
```

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Iteratoren
- Iterabelen
- `__iter__()`, `__next__()`, en `StopIteration`
- Verschillende manier om iterabele objecten te implementeren
- **`zip`**`()`
- Generatoren
- **`yield`**
- Generator expressies
- Van de `itertools` module de functies `chain()`, `zip_longest()`, `product()`, `permutations()`, `combinations()`, en `combinations_with_replacement()`

Opgaves

Opgave 23.1 Schrijf een programma dat de gebruiker vraagt om positieve integers in te geven. De gebruiker mag er zoveel ingeven als gewenst, en geeft aan dat de laatste integer is ingegeven door nul in te geven. Het programma toont vervolgens alle getallen tussen 1 en 100 die niet deelbaar zijn door ieder van de integers die zijn ingegeven. Toon die getallen middels een **`for ... in ...`** loop, waarbij je gebruik maakt van een iterator om de getallen te produceren.

Opgave 23.2 Schrijf een generator die faculteiten produceert. De eerste waarde die geretourneerd wordt is 1!, de tweede 2!, de derde 3!, etcetera, tot en met 10!. Bereken niet iedere keer de faculteit opnieuw, maar bewaar het laatst verkregen getal en gebruik dat om het volgende te berekenen.

Opgave 23.3 Vraag de gebruiker om een woord in te geven. Produceer alle anagrammen van dat woord. Als het woord bepaalde letters dubbel bevat, is het acceptabel als bepaalde anagrammen meerdere malen geproduceerd worden. Bijvoorbeeld, als het woord `"gen"` is, produceer je `"eng"`, `"egn"`, `"gen"`, `"gne"`, `"neg"`, en `"nge"` (volgorde maakt niet uit).

Opgave 23.4 Doe de voorgaande opgave nogmaals, maar zorg er nu voor dat alle anagrammen uniek zijn, zelfs als het woord dubbele letters bevat. Bijvoorbeeld, als het woord `"aap"` is, produceer je `"aap"`, `"apa"`, en `"paa"`.

Opgave 23.5 Het "subset som" probleem stelt de vraag of een bepaalde verzameling van integers een deelverzameling van integers bevat die, als ze worden opgeteld, nul als antwoord geven. Bijvoorbeeld, als de verzameling is opgeslagen als een list, dan is het

antwoord bij de list `[1, 4, -3, -5, 7]` "ja," aangezien $1 + 4 - 5 = 0$. Echter, voor de list `[1, 4, -3, 7]` is het antwoord "nee," aangezien er geen deelverzameling van de integers is die optellen tot nul. Schrijf een programma dat het subset som probleem oplost voor een list met integers. Als er een oplossing is, druk die af; als er geen oplossing is, geef dat dan aan.

Dit is een herhaling van één van de opgaves uit hoofdstuk 12 (Lists). In dat hoofdstuk zei ik dat je deze opgave het beste recursief kunt aanpakken. Echter, door de `itertools` module te gebruiken, kun je hem nu oplossen zonder recursie (ik vermoed dat recursie nog steeds plaatsvindt in de `itertools` module, maar jij hoeft je er niet druk om te maken).

Opgave 23.6 Schrijf een programma dat alle mogelijke sub-dictionaries produceert van een dictionary, en ze opslaat in een list. Bijvoorbeeld, als de dictionary `{"a":1,"b":2}` is, dan produceert het programma `[{},{"a":1},{"b":2},{"a":1,"b":2}]` (de volgorde in de list maakt niet uit). Gebruik de `itertools` module.

Opgave 23.7 Schrijf een programma dat bepaalt hoe je acht koninginnen kunt plaatsen op een schaakbord op zo'n manier dat geen van hen de anderen aanvalt. Dit is een klassiek probleem dat lijkt weinig van doen te hebben met dit hoofdstuk, maar als je een slimme manier bedenkt om het aan te pakken met de `permutations()` functie, dan zul je zien dat dit een verrassend kort programma is.

Hoofdstuk 24

Command Line Verwerking

In hoofdstuk 15 merkte ik op dat als je grote hoeveelheden data wilt verwerken die verspreid is over meerdere bestanden, je soms Python programma's wilt schrijven die in de command shell draaien. Dit wordt command line verwerking genoemd. Dit hoofdstuk beschrijft hoe je dat moet aanpakken.

24.1 De command line

Hoofdstuk 15 legde uit hoe je de "command prompt" of "command shell" van de computer kunt benaderen. Als je dat niet meer weet, neem dat hoofdstuk nog eens door. Het hoofdstuk gaf ook aan dat je Python programma's kunt starten op de command prompt via het commando:

```
python <programmanaam>.py
```

Dit werkt zolang je systeem weet waar het Python kan vinden en het programma zich in de huidige directory bevindt. Als het programma niet in de huidige directory staat, werkt het nog steeds, als je de programmanaam maar specificeert inclusief het volledige pad.

24.1.1 Batch verwerking

Stel je voor dat je een programma hebt geschreven dat de gebruiker vraagt om een bestandsnaam en misschien een aantal extra parameters. Daarna verwerkt je programma het genoemde bestand, gebruik makend van de parameters. Ik vraag je vervolgens het programma te draaien voor alle bestanden in een bepaalde directory. Deze directory bevat meer dan tienduizend bestanden. Wat doe je?

Je kunt je programma aanpassen zodat het niet meer vraagt om een bestand, maar om de naam van een directory, en dan alle bestanden in die directory verwerkt. Daarbij vraagt het dan niet de gebruiker om parameters, maar parameters die op één of andere manier van de bestandnaam worden afgeleid, of iets dergelijks. Er moet tenslotte een manier zijn om de parameters te achterhalen, dus je kunt je programma zo schrijven dat het dat zelf doet. Dit lost het probleem op. Maar dan vraag ik je om het programma uit te voeren voor een groot aantal directories. Wat doe je?

Je kunt je programma verder aanpassen zodat het een list bevat van alle directories die je moet verwerken, en ze dan één voor één afwerken. En ieder keer dat ik je vraag een extra directory te verwerken, pas je gewoon het programma aan. Het maakt niet uit wat ik vraag, je kunt altijd het programma verder aanpassen. Hoewel je wellicht een beetje geïrriteerd raakt van al mijn wijzigingsverzoeken.

Er bestaat een alternatieve manier om dit soort problemen af te handelen, en dat is via batch verwerking. Alle command shells ondersteunen het uitvoeren van "batchbestanden," die bestaan uit een lijst van commando's die je aan het besturingssysteem geeft. Onder Windows hebben dit soort bestanden de extensie `.bat`, terwijl bij de Mac en Linux ze een willekeurige naam kunnen hebben. Je kunt echter verschillende command shells installeren die andere conventies gebruiken – je kunt zelfs de Python shell voor dit doel gebruiken en Python zelf gebruiken om batchbestanden te schrijven.

Het batchbestand bevat commando's die gebruik maken van het eerste programma dat ik hierboven beschreef, dat slechts één bestand bewerkt, en roept dat programma aan voor ieder te verwerken bestand. Het probleem is dat het programma zo geschreven was dat het gebruikersinvoer nodig heeft, en je gaat natuurlijk niet iedere keer invoer geven als het batchbestand het programma opnieuw aanroept. De oplossing is het programma zo aan te passen dat het command line argumenten kan gebruiken.

24.1.2 Command line argumenten

Op de command line kun je een Python programma starten met een aantal argumenten. Je schrijft dan:

```
python <programmanaam>.py <argument_1> <argument_2> ... <argument_n>
```

De argumenten zijn van elkaar gescheiden door middel van spaties en mogen van alles zijn. Als je een argument hebt dat zelf een spatie bevat, moet je het tussen dubbele aanhalingstekens zetten. Dat doet natuurlijk onmiddellijk de vraag rijzen wat je moet doen als het argument een dubbel aanhalingsteken bevat, en helaas is het antwoord "dat hangt af van de command shell die je gebruikt." Meestal moet je het dubbele aanhalingsteken vooraf laten gaan door een backslash, of je moet een dubbel dubbel aanhalingsteken schrijven.

Natuurlijk zal je programma de argumenten niet automatisch verwerken. Je moet het programma uitbreiden met code die de argumenten "binnenhaalt."

24.1.3 `sys.argv`

Je krijgt toegang tot de command line argumenten die aan het programma worden meegegeven via een voorgedefinieerde list, die beschikbaar is als je de `sys` module importeert. De list heet `sys.argv`. Het is een list van strings, waarbij iedere string één van de command line argumenten is.

`sys.argv` bevat altijd minimaal één argument, namelijk de complete naam van het Python bestand dat je uitvoert, inclusief het pad. Om te weten hoeveel command line argumenten er zijn, kun je, zoals gewoonlijk, de **`len`**`()` functie gebruiken.

Als je programmeert in een editor die je ook gebruikt om je programma's te testen, kun je over het algemeen geen command-line argumenten specificeren. Dus als je hiermee wilt

experimenteren, moet je je programma's daadwerkelijk testen in de command shell. Dat is een heel gedoe, zeker tijdens het ontwikkelen van een programma. Ik kan je echter vertellen hoe je je programma's zodanig kunt opzetten dat het afhandelen van command-line argumenten optioneel is.

24.2 Flexibele command line verwerking

Als ik Python programma's schrijf, geef ik de voorkeur aan werken met een editor. Er zijn een paar editors die het meegeven van command-line argumenten ondersteunen tijdens testen, maar de meeste doen dat niet. Dus ik wil mijn programma's zodanig bouwen dat ze command-line argumenten kunnen verwerken, maar ik ze toch kan testen vanuit een editor, en het testen van het programma in de echte command shell slechts éénmalig hoef te doen. Dat doe ik als volgt:

Voor iedere parameter die een waarde moet krijgen via de command line, creëer ik een globale variabele. Ik vul die globale variabelen met default waardes. In de rest van het programma gebruik ik die variabelen alsof het constanten zijn. Slechts bij de start van het hoofdprogramma controleer ik of er command-line argumenten zijn meegegeven, en als dat zo is, overschrijf ik de variabelen met de waardes die op de command line staan.

Het voordeel van deze aanpak is dat ik mijn programma kan ontwikkelen zonder command-line parameters te gebruiken. Als ik verschillende waardes voor de command-line argumenten wil testen, stop ik gewoon andere waardes in de variabelen die ik gebruik om de command-line argumenten op te slaan. Ik kan het programma zelfs zo opzetten dat ik de variabele ofwel vul met een command-line parameter als die is meegegeven, ofwel de gebruiker vraag om een waarde voor de command-line parameter als die niet is meegegeven.

Zulk soort code ziet er typisch als volgt uit:

listing2401.py

```
import sys

# 3 variabelen die de command line parameters bevatten
invoer = "input.txt"
uitvoer = "output.txt"
shift = 3

# Verwerken van command line parameters
# (werkt met 0, 1, 2, of 3 parameters)
if len( sys.argv ) > 1:
    invoer = sys.argv[1]
if len( sys.argv ) > 2:
    uitvoer = sys.argv[2]
if len( sys.argv ) > 3:
    try:
        shift = int( sys.argv[3] )
    except TypeError:
        print( sys.argv[3], "is geen getal." )
        sys.exit(1)
```

Deze code ondersteunt drie command line argumenten: de eerste twee zijn strings, en de derde is een integer. De derde wordt onmiddellijk geconverteerd van een string naar een integer (aangezien een command line argument altijd een string is), en het programma wordt afgebroken als deze conversie mislukt. Ik zou meer controles hebben kunnen inbouwen in deze demonstratie, maar ik neem aan dat dat op dit punt in je programmeercarrière geen probleem voor je is.

Merk op dat alledrie de argumenten een default waarde hebben: de eerste string heeft als default waarde `"input.txt"`, de tweede string heeft als default waarde `"output.txt"`, en de integer heeft als default waarde 3. Je hoeft niet alle argumenten op de command line mee te geven: als je er geen mee geeft, worden alle drie de default waardes gebruikt; als je er één meegeeft, wordt de eerste string overschreven met het command-line argument, en houden de andere twee hun default waarde; etcetera.

24.2.1 `sys.exit()`

In de code hierboven wordt het programma afgebroken via `sys.exit()` als een argument niet voldoet aan de gestelde eisen. `sys.exit()` heb ik geïntroduceerd in hoofdstuk 6. Ik vertelde er toen niet bij dat `sys.exit()` een numeriek argument kan krijgen, zoals je hierboven ziet. Dit argument wordt geretourneerd naar het batchbestand waarin het programma is aangeroepen, en het batchbestand kan er mogelijk op reageren. Meestal is dit argument een foutcode. Typisch wordt nul als argument gegeven als alles in principe correct verwerkt is (een programma dat "normaal" eindigt retourneert ook een nul), en anders wordt een ander getal gebruikt. Omdat sommige systemen deze getallen beperken tot de waardes 0 tot en met 255, is het de gewoonte om die restrictie ook te gebruiken voor de waardes die `sys.exit()` als argument mee kan krijgen.

24.2.2 `argparse`

Er bestaat een module die command-line verwerking ondersteunt, namelijk de standaard module `argparse`. Om eerlijk te zijn zie ik zelf het nut van zo'n module niet in, omdat command-line verwerking te simpel is om er veel tijd aan te spenderen. Maar sommige Python programma's, met name programma's die bedoeld zijn om het besturingssysteem uit te breiden, kennen een grote variëteit aan command-line argumenten, en kunnen profiteren van een dergelijke module. Als je erin geïnteresseerd bent, kun je er de documentatie op nalezen.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Command-line argumenten
- `sys.argv`
- Flexibele command-line verwerking

Opgaves

Opgave 24.1 Creëer een programma dat je kunt starten met nul of meer numerieke argumenten. Als het programma een niet-numeriek argument meekrijgt, geeft het een foutmelding. Als het alleen maar numerieke argumenten meekrijgt, telt het ze op en drukt de som af. Test het programma op de command line.

Hoofdstuk 25

Reguliere Expressies

Als je een probleem moet oplossen met tekstanalyse, data verwerking, het zoeken van patronen in grote data verzamelingen, of het filteren van data uit webpagina's, en je zoekt op het Internet naar een oplossing voor het probleem, dan zul je zien dat het eerste antwoord dat meestal op vragen hierover gegeven wordt is "Waarom gebruik je geen reguliere expressies?" of zelfs "Gebruik gewoon regex," zonder verdere uitleg. Dat zijn nogal zelfingenomen antwoorden, aangezien maar weinig mensen de term reguliere expressies kennen, en zelfs als ze hem kennen, ze reguliere expressies vaak eng en onbegrijpelijk vinden. En het is waar dat op het eerste gezicht reguliere expressies zo esoterisch en verwarrend zijn dat de meeste mensen liever besmuikt wegsluipen dan er tijd aan te besteden. Wat jammer is, daar reguliere expressies een krachtig hulpmiddel zijn dat niet mag ontbreken in de gereedschapskist van eenieder die regelmatig moet omgaan met tekstuele data.

In dit hoofdstuk leg ik uit hoe je basale reguliere expressies schrijft en gebruikt met Python. Je zult ontdekken dat ze een krachtige manier bieden om snel complexe patronen in data te identificeren, en toegang geven tot functionaliteiten die in gewoon Python lastig te implementeren zijn. Hoewel dit hoofdstuk geen compleet overzicht verschaft over reguliere expressies, zul je aan het einde ervan voldoende begrijpen om de meeste, zo niet alle problemen rondom het identificeren van patronen in teksten op te kunnen lossen, en je kunt beginnelingen toespreken met de woorden: "Je moet reguliere expressies gebruiken om je problemen op te lossen." Nu kun jij ook zelfingenomen zijn!

25.1 Reguliere expressies met Python

Reguliere expressies zijn tekst strings die een "patroon" beschrijven dat je kunt vinden in tekstuele data. Bijvoorbeeld, de reguliere expressie `a+` beschrijft een patroon dat bestaat uit een serie van één of meer keer de letter "a." In de string "aardvarken" vind je dit patroon twee keer, namelijk de "aa" aan het begin van het woord, en de enkele "a" in de tweede helft van het woord.

Een reguliere expressie bestaat altijd uit een string, die ieder teken mag bevatten. Sommige tekens zijn "meta-tekens" die een speciale betekenis hebben in reguliere expressies. Je moet voorzichtig zijn met ze te gebruiken (hoe je ze gebruikt volgt later). De meta-tekens zijn:

```
. ^ $ * + ? { } [ ] \ | ( )
```

Ik ga later in dit hoofdstuk uitleggen hoe je reguliere expressies samenstelt. Ik moet eerst bediscussiëren hoe je reguliere expressies opneemt in Python code.

25.1.1 De `re` module

Om reguliere expressies in Python te gebruiken, moet je de `re` module importeren.

Een reguliere expressie kun je beschouwen als een stuk code. Die code kun je "compileren" via de `re` module om een "patroon object" te produceren. Dat patroon object kun je dan gebruiken om te zoeken naar het patroon in data. Bijvoorbeeld, in de volgende code wordt de reguliere expressie `a+` gecompileerd om een patroon object te produceren dat wordt opgeslagen als `pAplus`. Dat patroon object wordt vervolgens gebruikt om het patroon te zoeken in de string "aardvarken." De patronen die gevonden worden, worden opgeslagen in een list, en die list wordt dan geprint.

listing2501.py

```
import re

pAplus = re.compile( r"a+" )
lAplus = pAplus.findall( "aardvarken" )
print( lAplus )
```

Opgave Verander het woord "aardvarken" in de code hierboven in iets anders, en bekijk hoe de uitvoer verandert.

Je vraagt je misschien af wat die letter "`r`" doet voor de string die de reguliere expressie bevat. Waarom schreef ik `r"a+"` in plaats van gewoon `"a+"`? Deze letter "`r`" vertelt Python dat deze string beschouwd moet worden als "ruwe data," dat wil zeggen, dat Python niet moet proberen delen van de string te converteren via de standaard Python string interpretaties. Dat is met name belangrijk als de reguliere expressie een `"\b"` bevat, wat voor reguliere expressies betekent "woord begrenzing" (dat leg ik later in dit hoofdstuk uit), maar voor Python een speciaal teken is dat "backspace" betekent. Dus je doet er goed aan die letter "r" altijd voor reguliere expressies op te nemen, om problemen te voorkomen.

Hoewel het in de praktijk niet veel voorkomt, mag je een optionele tweede parameter (een zogeheten "vlag") opnemen in de aanroep van `compile()`, die aangeeft dat het patroon op een speciale manier verwerkt moet worden. Het argument `re.I` geeft aan dat er geen onderscheid gemaakt moet worden tussen hoofd- en kleine letters, terwijl `re.S` aangeeft dat het patroon ook "newlines" moet verwerken, en `re.M` aangeeft dat het patroon de metatekens `^` en `$` moet toepassen op iedere regel tekst, en niet alleen de tekst als geheel. Je mag deze argumenten met elkaar combineren middels pipe-lines (`|`).

25.1.2 Verkorte compilatie

Het is toegestaan om de compilatie-stap over te slaan, en het zoeken van het patroon aan te roepen met een "class call" in de `re` module. In plaats van methodes aan te roepen van het patroon object, kun je de methodes direct aanroepen voor `re`, waarbij de reguliere expressie als eerste parameter wordt opgegeven. De code hierboven wordt dan:

```
import re

lAplus = re.findall( r"a+", "aardvarken" )
print( lAplus )
```

Als je deze code uitvoert, zie je dat de uitvoer precies gelijk is aan die van de eerste code. De tweede manier compileert nog steeds de reguliere expressie, maar slaat het patroon object niet op. Als het patroon slechts een paar keer in de code gebruikt wordt, dan is deze manier prima. Als het patroon echter vaak gebruikt wordt, is de eerste methode te prefereren, omdat de compilatie van de reguliere expressie (wat de meeste tijd in beslag neemt) slechts eenmalig gedaan wordt, in plaats van iedere keer.

25.1.3 Match objecten

De `findall()` methode die ik hierboven gebruikte retourneert alle instanties van het patroon in de string waarin gezocht wordt. Vaak heb je meer informatie nodig dan alleen de patronen; bijvoorbeeld, je wilt misschien weten waar precies in de string het patroon staat. De `re` module heeft methodes die resulteren in zogeheten "match objecten," wat objecten zijn die behalve het tekstuele patroon meer informatie bevatten, zoals de index waar het patroon gevonden is. Bijvoorbeeld, de `search()` methode retourneert een match object voor de eerste instantie waar het patroon in de string voorkomt.

```
import re

m = re.search( r"a+", "Kijk uit voor het aardvarken!" )
print( "{} gevonden op index {}".format( m.group(), m.start() ) )
```

Zoals je kunt zien heeft het match object diverse nuttige methodes. Deze zijn:

- `group()` die het gevonden patroon retourneert
- `start()` die de index retourneert waar het patroon start
- `end()` die de index retourneert waar het patroon geëindigd is

De `group()` methode heeft een aantal handige toepassingen die je via argumenten kunt benaderen, wat ik later zal bespreken.

De `match()` methode lijkt op de `search()` methode, maar controleert of het patroon bestaat bij de start van de string (dus beginnend bij index 0). Beide methodes retourneren **`None`** als het patroon niet gevonden is, dat door Python als **`False`** beschouwd wanneer het als conditie gebruikt wordt.

```
import re

m = re.match( r"a+", "Kijk uit voor het aardvarken!" )
if m:
    print( "{} start de string".format( m.group() ) )
else:
    print( "Het patroon staat niet aan de start van de string" )
```

25.1.4 Lists van matches

Ik liet zien dat de `findall()` methode een list bouwt van de voorkomende patronen. `findall()` wordt gecomplementeerd door de `finditer()` methode, die een list (of liever gezegd, een iterator) bouwt van match objecten voor het gezochte patroon. De beste manier om zo'n list te verwerken is via de **`for`** `m` **`in`** `...` constructie. Bijvoorbeeld:

listing2502.py

```
import re

mlist = re.finditer( r"a+",
    "Kijk uit! Een gevaarlijk aardvarken is ontsnapt!" )
for m in mlist:
    print( "{} gevonden bij index {} tot index {}.".format(
        m.group(), m.start(), m.end() ) )
```

25.2 Reguliere expressies schrijven

Nu de basis van het gebruik van reguliere expressies in Python via de `re` module is uitgelegd, kan ik eindelijk beginnen met de uitleg van het echte schrijven van reguliere expressies.

25.2.1 Reguliere expressies met vierkante haken

De eenvoudigste reguliere expressie is een string van tekens, die een patroon beschrijft dat bestaat uit precies die string van tekens. Je mag ook een verzameling tekens beschrijven middels vierkante haken `[` en `]`. Bijvoorbeeld, de reguliere expressie `[aeiou]` beschrijft een teken dat een `"a"`, `"e"`, `"i"`, `"o"`, of `"u"` is. Dit betekent dat als `[aeiou]` onderdeel is van een reguliere expressie, dat op die locatie in het patroon één van die letters moet staan (en wel precies één, en niet meerdere). Bijvoorbeeld, om de woorden `"ball"`, `"bell"`, `"bill"`, `"boll"` en `"bull"` te zoeken, kun je de reguliere expressie `b[aeiou]ll` gebruiken.

listing2503.py

```
import re

slist = re.findall( r"b[aeiou]ll", "Bill Gates en Uwe Boll \
dronken Red Bull bij het voetballen in Campbell." )
print( slist )
```

Opgave Wijzig de reguliere expressie hierboven zodat niet alleen de woorden "ball" en "bell" gevonden worden, maar ook "Bill", "Boll", en "Bull".

Je kunt tussen de vierkante haken een min-teken gebruiken tussen twee tekens om aan te geven dat niet alleen die twee tekens bedoeld worden, maar ook alle tekens die ertussen liggen. Bijvoorbeeld, de reguliere expressie `[a-dqx-z]` is equivalent met `[abcdqxyz]`. Om

alle letters van het alfabet aan te geven, zowel als hoofd- of als kleine letter, kun je `[A-Za-z]` gebruiken.

Bovendien kun je een "dakje" (`^`) meteen rechts naast de vierkante-haak-open plaatsen om aan te geven dat je het tegengestelde bedoeld van wat er tussen de vierkante haken staat. Bijvoorbeeld, `[^0-9]` geeft aan alle tekens behalve cijfers.

25.2.2 Speciale tekens

In reguliere expressies, net als in strings, wordt het backslash teken (`\`) gebruikt om aan te geven dat het volgende teken een speciale betekenis heeft. De speciale tekens die je voor strings kunt gebruiken, gelden ook voor reguliere expressies, maar reguliere expressies hebben er meer. Er zijn ook een aantal meta-tekens die op een speciale manier geïntepreteerd worden. De volgende speciale tekens zijn gedefinieerd (er zijn er meer, maar deze worden het meest gebruikt):

```
\b    Woord begrenzing (breedte nul)
\B    Geen woord begrenzing (breedte nul)
\d    Cijfer [0-9]
\D    Geen cijfer [^0-9]
\n    Nieuwe regel (newline)
\r    Return
\s    Spatie (inclusief tabulatie)
\S    Geen spatie
\t    Tabulatie
\w    Alfanumeriek teken [A-Za-z0-9_]
\W    Geen alfanumeriek teken [^A-Za-z0-9_]
\/    Voorwaartse slash
\\    Backslash
\"    Dubbel aanhalingsteken
\'    Enkel aanhalingsteken
^     Start van een string (breedte nul)
$     Einde van een string (breedte nul)
.     Ieder teken
```

Merk op dat "breedte nul" betekent dat het teken niet een echt teken in de string aanduidt, maar een positie in de string tussen twee tekens (of het begin of einde van de string). Bijvoorbeeld, de reguliere expressie `^A` representeert een string die start met de letter `"A"`.

Bovendien kun je tekens of substrings tussen haakjes zetten, wat de tekens "groepeert." Binnen een groep kun je een keuze tussen meerdere (groepjes) tekens aanduiden middels een pipe-line (`|`). Bijvoorbeeld, de reguliere expressie `(appel|banaan|peer)` is de string `"appel"` of de string `"banaan"` of de string `"peer"`.

Je moet je ervan bewust zijn dat sommige van deze speciale tekens (zeker die zonder backslash, de haakjes, en de pipe-line) niet werken zoals aangegeven als ze tussen vierkante haken staan. Bijvoorbeeld, de punt tussen vierkante haken is niet "ieder teken," maar een echte punt.

25.2.3 Herhaling

Waar reguliere expressie pas echt interessant worden is wanneer herhalingen worden gebruikt. Een aantal van de meta-tekens worden gebruikt om aan te geven dat (een deel van) een reguliere expressie meerdere keren herhaald wordt. De volgende herhalingsoperatoren worden vaak gebruikt:

```
*      Nul of meer keer
+      1 of meer keer
?      Nul of 1 keer
{p,q}  Minstens p en hoogstens q keer
{p,}   Minstens p keer
{p}    Precies p keer
```

Je zet zo'n operator achter het deel van de reguliere expressie dat herhaald moet worden. Bijvoorbeeld, `ab*c` betekent de letter `"a"`, gevolgd door nul of meer keer de letter `"b"`, gevolgd door de letter `"c"`. Deze expressie representeert de strings `"ac"`, `"abc"`, `"abbc"`, `"abbbc"`, `"abbbbc"`, etcetera.

Als je een herhalingsoperator zet achter een groepje (tussen haakjes), duidt het op de herhaling van het groepje. Bijvoorbeeld, `(ab)*c` representeert de strings `"c"`, `"abc"`, `"ababc"`, `"abababc"`, `"abababababc"`, etcetera.

Het matchen van herhalingen gebeurt "gulzig" (Engels: "greedy"). Er wordt altijd geprobeerd het patroon zo vroeg mogelijk in de tekst te matchen, en de herhaling zo breed mogelijk uit te strekken. Bekijk de volgende code:

listing2504.py

```
import re

mlist = re.finditer(r"ba+","Schaap zegt 'baaaaah' tot Ali Baba.")
for m in mlist:
    print( "{} is found at {}.".format(m.group(), m.start()))
```

Opgave Wijzig de reguliere expressie in de code hierboven zodat het iedere `"b"` die gevolgd wordt door één of meer `"a"`s vindt, waarbij de `"b"` eventueel een hoofdletter mag zijn. De uitvoer moet zijn `"baaaaa"`, `"Ba"` en `"ba"`.

Opgave Wanneer je de vorige opgave hebt opgelost, wijzig je de reguliere expressie zodat hij patronen vindt die bestaan uit een `"b"` of `"B"` gevolgd door één of meer `"a"`s, en dat één of meer keer herhaald. De uitvoer moet zijn `"baaaaa"` en `"Baba"`. Je moet hier haakjes bij gebruiken. Test de reguliere expressie ook uit op een aantal andere teksten.

Hier is er nog een, die één of meer herhalingen van de letter `"a"` zoekt:

listing2505.py

```
import re

mlist = re.finditer(r"a+","Schaap zegt 'baaaaah' tot Ali Baba.")
for m in mlist:
    print( "{} is found at {}.".format(m.group(), m.start()))
```

Als je deze code uitvoert, zie je dat het patroon vijf keer gevonden wordt: drie keer een enkele "a", een dubbele "a", en een groep van vijf "a"s. Je vraagt je misschien af waarom het proces niet ook de "a"s vindt binnen de langere patronen, bijvoorbeeld, de twee groepen van vier "a"s die te vinden zijn in de groep van vijf "a"s. De reden is dat de `finditer()` en `findall()` methodes, als ze een match gevonden hebben, verder gaan zoeken meteen na de laatst gevonden match. Gewoonlijk is dit het gedrag dat je wilt zien.

Opgave Wijzig nu de `r"a+"` in de code hierboven in `r"a*"`, zodat gezocht wordt naar nul of meer "a"s. Voordat je de code uitvoert, probeer te bedenken wat het resultaat zal zijn. Voer dan de code uit en zie of je gelijk hebt. Indien niet, snap je dan wel waarom de uitkomst zo is als hij is?

Je zult wel gemerkt hebben dat reguliere expressies snel complex worden. Het is een goed idee om commentaar erbij te schrijven zodat je ook later je eigen code nog begrijpt.

25.2.4 Oefening

Met alles wat je tot nu toe geleerd hebt, moet je de volgende opgave kunnen maken. Het is verstandig om deze op te lossen voordat je met de rest van het hoofdstuk verder gaat. De opgave bestaat uit een stuk code dat je moet afmaken door een aantal reguliere expressies in te vullen.

Opgave Als je de code hieronder uitvoert, zoekt hij naar alle reguliere expressies in `relist`, in alle strings in `slist`. Dan wordt voor iedere string de nummers afgedrukt van de reguliere expressies waarvoor matches zijn gevonden. Je doel is om de reguliere expressies in `relist` in te vullen volgens de beschrijvingen die in het commentaar ernaast staan. Merk op dat de eerste zeven reguliere expressies de string als geheel beschrijven, en die moet je dus laten starten met een dakje en eindigen met een dollar teken, wat aangeeft dat de expressie de string van het begin (^) tot het einde ($) moet representeren.

listing2506.py

```
import re

# List van strings die worden gebruikt voor testen.
slist = [ "aaabbb", "aaaaaa", "abbaba", "aaa", "bEver ottEr",
    "tango samba rumba", " hello world ", " Hello World " ]

# Reguliere expressies die moeten worden ingevuld.
relist = [
    r"",  # 1. Alleen a's gevolgd door alleen b's, inclusief ""
    r"",  # 2. Alleen a's, inclusief ""
    r"",  # 3. Alleen a's en b's, willekeurige volgorde, incl. ""
    r"",  # 4. Precies drie a's
    r"",  # 5. Noch a's noch b's, maar "" is toegestaan
    r"",  # 6. Een even aantal a's (en niks anders)
    r"",  # 7. Precies twee woorden, ongeacht spaties
    r"",  # 8. Bevat een woord dat op "ba" eindigt
    r""   # 9. Bevat een woord dat begint met een hoofdletter
]
```

```
for s in slist:
    print( s, ':', sep='', end=' ' )
    for i in range( len( relist ) ):
        m = re.search( relist[i], s )
        if m:
            print( i+1, end=' ' )
    print()
```

De correcte uitvoer is:

```
aaabbb: 1 3
aaaaaa: 1 2 3 6
abbaba: 3 8
aaa: 1 2 3 4
bEver ottEr: 7
tango samba rumba: 8
 hello world : 5 7
 Hello World : 5 7 9
```

Zorg dat je deze allemaal kunt oplossen voordat je verder gaat!

25.3 Groeperen

Zoals ik hierboven heb uitgelegd, kun je groepen maken binnen een reguliere expressie middels haakjes. De reguliere expressie `(\d{1,2})-(\d{1,2})-(\d{4})` kan bijvoorbeeld een datum beschrijven: één of twee cijfers, gevolgd door een streepje, gevolgd door één of twee cijfers, gevolgd door een streepje, gevolgd door vier cijfers (als je deze reguliere expressie niet begrijpt, bestudeer dan de eerdere delen van dit hoofdstuk totdat je hem wel begrijpt). Deze expressie bevat drie groepen: de eerste bestaand uit één of twee cijfers, de tweede bestaand uit één of twee cijfers, en de derde bestaand uit vier cijfers. De code hieronder zoekt naar dit patroon in een string.

listing2507.py

```
import re

pDatum = re.compile( r"(\d{1,2})-(\d{1,2})-(\d{4})" )
m = pDatum.search( "In antwoord op uw schrijven van 25-3-2015, \
heb ik besloten een huurmoordenaar op u af te sturen." )
if m:
    print( "Datum: {}; dag: {}; maand: {}; jaar: {}".format(
            m.group(0), m.group(1), m.group(2), m.group(3) ) )
```

Als je deze code uitvoert, zie je dat niet alleen het resultaat als geheel wordt gevonden (via de methode `group()` of `group(0)`), maar ook dat je ieder van de groepen afzonderlijk kunt benaderen, via methodes `group(1)` voor de dag, `group(2)` voor de maand, en `group(3)` voor het jaar. Je kunt ook de methode `groups()` gebruiken om een tuple te krijgen waarin alle groepen zitten.

25.3.1 `findall()` en groepen

De `findall()` methode retourneert een list van patroon objecten. In de voorbeelden die ik tot nu toe heb gegeven, was dat een list van strings. En inderdaad is een patroon object een string als er ten hoogste één groep in de reguliere expressie zit. Als er meerdere groepen zijn, is een patroon object een tuple waarin alle groepen zitten.

listing2508.py

```
import re

pDatum = re.compile( r"(\d{1,2})-(\d{1,2})-(\d{4})" )
datumlist = pDatum.findall( "In antwoord op uw schrijven van \
25-3-2015, heb ik op 27-3-2015 besloten u verder te negeren." )
for datum in datumlist:
    print( datum )
```

25.3.2 Benoemde groepen

Het is mogelijk om iedere groep een naam te geven, via de constructie `?P<naam>` (waarbij je voor "naam" de naam invult die je aan de groep wilt geven – in dit geval laat je de < en > dus in de uitdrukking staan) onmiddellijk achter het openingshaakje. Je kunt daarna aan de groepen refereren met hun naam, in plaats van een index.

listing2509.py

```
import re

pDatum = re.compile(
    r"(?P<dag>\d{1,2})-(?P<maand>\d{1,2})-(?P<jaar>\d{4})" )
m = pDatum.search( "In antwoord op uw schrijven van 25-3-2015, \
heb ik besloten een zingend telegram op u af te sturen." )
if m:
    print( "dag is {}".format( m.group('dag') ) )
    print( "maand is {}".format( m.group('maand') ) )
    print( "jaar is {}".format( m.group('jaar') ) )
```

25.3.3 Refereren binnen een reguliere expressie

Stel dat je een reguliere expressie moet schrijven die een string representeert waarin een willekeurig teken (dat geen spatie is) twee keer voorkomt. Bijvoorbeeld, de string "gasoven" zou geen match geven, maar de string "magnetron" wel (aangezien de `"n"` twee keer voorkomt). Dit kun je niet doen met de mogelijkheden van reguliere expressies die ik tot nu toe bediscussieerd heb. Je kunt het echter oplossen met groepen, en speciale referenties binnen reguliere expressies, als volgt: je gebruikt het speciale teken `\x`, waarbij `x` een cijfer is, waarmee je dan refereert aan de groep met index `x` in het patroon. Dus de gevraagde reguliere expressie is `(\S).*\1`.

Omdat op dit punt deze reguliere expressie misschien wat moeilijk te begrijpen is, zal ik hem in detail doornemen. De `\S` is een speciaal teken dat een non-spatie voorstelt. Door er

haakjes omheen te zetten, wordt het een groep, en omdat de eerste (en enige) groep is, is de index 1. De `.*` stelt een serie van nul of meer tekens voor die alles kunnen zijn (de punt is een meta-teken dat ieder teken kan voorstellen). Tenslotte is de `\1` een referentie aan de eerste groep, en stelt dat je hier exact moet hebben wat de eerste groep is. Als je je afvraagt of je niet ook moet beschrijven wat er vóór de `\S` of na de `\1` komt, dan is het antwoord dat dat niet nodig is, aangezien deze reguliere expressie niet de string als geheel representeert. Dus zolang dit maar ergens voorkomt in de string, wordt het patroon gevonden.

Test deze reguliere expressie in de code hieronder, waarbij je de string `"Monty Python's Flying Circus"` vervangt door andere strings, en het resultaat van de uitvoering bekijkt.

listing2510.py

```
import re

m = re.search( r"(\S).*\1", "Monty Python's Flying Circus" )
if m:
    print( "{} komt twee keer voor in de string".format(
        m.group(1) ) )
else:
    print( "Geen match gevonden." )
```

Opgave Kun je de reguliere expressie in de code hierboven zo wijzigen dat het een patroon beschrijft waarbij een willekeurig teken minimaal drie keer voorkomt?

Opgave Kun je de reguliere expressie wijzigen zodat het een patroon beschrijft waarbij twee tekens twee keer voorkomen? Dit is behoorlijk moeilijk en daarom optioneel, maar als je het probeert, zorg er dan voor dat je het test met op zijn minst de strings `"aaaa"`, `"aabb"`, `"abab"` and `"abba"`. Deze moeten allemaal een match geven, tenzij je ook nog eist dat de twee tekens verschillend moeten zijn, in welk geval `"aaaa"` geen match mag geven (maar dat maakt de reguliere expressie nog eens een stuk moeilijker).

25.4 Vervangen

Hoewel gewoonlijk reguliere expressies alleen gebruikt worden om naar patronen te zoeken, kun je ze ook gebruiken om substrings in een string te vervangen door andere substrings. Dit doe je met de `sub()` methode. `sub()` krijgt als argumenten het patroon dat je wilt vervangen, het patroon waarmee je wilt vervangen, en de string. `sub()` retourneert de nieuwe string (bedenk dat strings onveranderbaar zijn, dus `sub()` maakt geen wijziging in de string, zelfs niet als die in een variabele staat; als je de nieuwe string wilt gebruiken moet je hem zelf in een variabele stoppen).

De vervanging is meestal gewoon een string, maar er kunnen verwijzingen in staan naar groepen in het eerste patroon. Je moet dan een formaat gebruiken dat verschilt van het `\x` formaat dat ik hierboven beschreef. Als je wilt refereren aan groep `x` in het patroon (waarbij `x` een getal is), dan schrijf je `\g<x>` (ook hier moet je de < en > in het patroon opnemen). De reden voor dit verschil is ondubbelzinnigheid; het maakt het mogelijk om verschil te maken tussen, bijvoorbeeld, een referentie naar groep 2 gevolgd door het teken 0, en een referentie naar groep 20.

```
import re

s = re.sub( r"([iy])z(eert)", "\g<1>s\g<2>", "Of je nu \
categorizeert, rationalizeert, of analyzeert, je moet \
een s gebruiken!" )
print( s )
```

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Wat reguliere expressies zijn
- Welke meta-tekens in reguliere expressies gebruikt kunnen worden
- Hoe reguliere expressies in Python gebruikt worden, via de `re` module
- Compileren van reguliere expressies met `re.compile()`
- Wat match objecten zijn
- Zoeken naar patronen middels `match()`, `search()`, `findall()`, en `finditer()`
- Vervangen van patronen met de `sub()` methode
- Het gebruik van vierkante haken in reguliere expressies om verschillende mogelijkheden voor een teken te representeren
- Gebruik van speciale tekens in reguliere expressies, waarvan veel beginnen met een backslash
- Herhalen van subpatronen in reguliere expressies via herhalingsoperatoren
- Groeperen van subpatronen middels haakjes
- Het gebruik van groepen om resultaten te ontrafelen
- Refereren binnen een patroon
- Mensen op zelfingenomen wijze het gebruik van reguliere expressies aanbevelen

Opgaves

Opgave 25.1 Neem aan dat een woord alleen kan bestaan uit letters van het alfabet (hoofd- of kleine letters). Schrijf code die een reguliere expressie gebruikt om alle woorden van een tekst in een list te zetten.

Opgave 25.2 Gebruik een reguliere expressie en de `findall()` methode om een list te maken van alle instanties van het woord “de” in een zin. Druk het aantal instanties af. Je code mag geen onderscheid maken tussen hoofd- en kleine letters. Zorg ervoor dat je de combinatie “de” niet telt als het een onderdeel is van een ander woord (bijvoorbeeld “onderdeel,” “delicaat”, of “woede”).

Opgave 25.3 De volledige naam van een persoon bestaat uit twee woorden, die naast elkaar staan, en die alleen letters van het alfabet bevatten, die allemaal kleine letters zijn behalve de eerste, die een hoofdletter moet zijn. Tussen de woorden mogen alleen spaties staan. De woorden beginnen en eindigen bij een woordscheider. Bijvoorbeeld, volgens deze specificatie is Kardinaal Richelieu een naam, maar Charles d'Artagnan niet, noch Gilbert duPrez, Joe DiMaggio, of Unit X1138. Maak een reguliere expressie die een list maakt van alle twee-woorden combinaties in een zin die (waarschijnlijk) de namen van personen zijn onder deze aanname.

Opgave 25.4 Als vervolg op de vorige opgave, neem nu aan dat de naam van een persoon mag bestaan uit twee of meer woorden, met alle andere criteria als hierboven gegeven. Schrijf een reguliere expressie die alle namen uit een tekst haalt.

Opgave 25.5 Als in een stuk tekst een persoon spreekt, wordt dit meestal weergegeven door de gesproken tekst tussen dubbele aanhalingstekens (") te plaatsen. Schrijf een reguliere expressie die alle delen van een tekst die tussen dubbele aanhalingstekens staan eruit haalt. Hint: Gebruik groepen, en bedenk dat reguliere expressies "gulzig" zijn.

Opgave 25.6 Als je data wilt halen uit HTML pagina's, zoek je vaak naar de delen waarin je geïnteresseerd bent middels "mark-ups." Stel dat je een pagina hebt met data van personen, die een ID en een naam hebben. De ID is een negen-cijferig getal, omsloten door een code `<id>` ervoor, en een code `</id>` erna. De naam van de persoon volgt onmiddellijk na de ID, en wordt gemarkeerd door de code `<naam>` ervoor, en de code `</naam>` erna. Gebruik een reguliere expressie om alle IDs met bijbehorende namen uit de tekst hieronder te halen en te tonen. Er zijn er vijf.

exercise2506.py

```
import re

tekst = "<html><head><title>Lijst van personen met ids</title>\
</head><body>\
<p><id>123123123</id><naam>Groucho Marx</naam>\
<p><id>123123124</id><naam>Harpo Marx</naam>\
<p><id>123123125</id><naam>Chico Marx</naam>\
<randomcrap>Etaoin<id>Shrdlu</id>qwerty</naam></randomcrap>\
<nocrap><p><id>123123126</id><naam>Zeppo Marx</naam></nocrap>\
<address>Chicago</address>\
<morerandomcrap><id>999999999</id>geennaamtezien!\
</morerandomcrap>\
<p><id>123123127</id><naam>Gummo Marx</naam>\
<note>Zoek hem op via <a href=\"http://www.google.com\">\
Google.</a></note>\
</body></html>"
```

Hoofdstuk 26

Bestandsformaten

Data is meestal opgeslagen in bestanden, die geconstrueerd zijn volgens een specifiek bestandsformaat. Gestandaardiseerde bestandsformaten worden door Python vaak ondersteund door een module. In dit hoofdstuk beschrijf ik een aantal veelgebruikte bestandsformaten en de modules die ze ondersteunen.

26.1 CSV

CSV staat voor "Comma-Separated Values," en is het meest gebruikte bestandsformaat voor het importeren en exporteren van data in en vanuit spreadsheets en databases. Het formaat stelt dat iedere regel in het CSV bestand één record (een complete entiteit) bevat, waarbij de velden van het record in een specifieke volgorde opgenomen zijn, van elkaar gescheiden zijn door komma's. De eerste regel van het bestand mag een lijst van de namen van de velden bevatten.

De code hieronder laadt en toont de inhoud van een typisch CSV bestand. Appendix E legt uit hoe je dit CSV bestand kunt krijgen (als je het niet al gedownload hebt).

```
fp = open( "pc_inventory.csv" )
print( fp.read().strip() )
fp.close()
```

Helaas is het CSV formaat niet gestandaardiseerd, en verschillende applicaties gebruiken licht verschillende versies van CSV bestanden. Er is wel een standaard die vrij veel gebruikt wordt, en deze standaard is geïmplementeerd in de Python `csv` module. De module ondersteunt ook diverse "dialecten" van het CSV formaat om verschillende soorten bestanden te kunnen lezen en schrijven.

Het kan gebeuren dat je een excentriek CSV formaat moet gebruiken dat niet door de module ondersteund wordt. Je kunt dan proberen met reguliere expressies je eigen versie van het CSV formaat te ondersteunen, of aan de module je eigen dialect toevoegen. Geen van beide biedt een aantrekkelijk perspectief.

26.1.1 CSV `reader()`

De `csv` module bevat een `reader()` functie die toegang geeft tot een CSV bestand. De `reader()` functie krijgt een handle als argument, en retourneert een iterator die je de mogelijkheid geeft regels uit het bestand te halen als een list waarbij ieder veld van een record een element van de list is. Je moet het bestand open laten zolang je het wilt benaderen met `reader()`.

listing2601.py

```
from csv import reader

fp = open( "pc_inventory.csv", newline='' )
csvreader = reader( fp )
for regel in csvreader:
    print( regel )
fp.close()
```

De Python documentatie geeft de aanbeveling om, als je `reader()` gebruikt op een bestand (en dat is wat je meestal doet), een `newline=''` argument op te nemen bij het openen van het bestand (dat deed ik in de code hierboven). Dit is noodzakelijk als er tekstvelden in het CSV bestand staan die zelf newline tekens bevatten.

`reader()` zelf kan ook argumenten krijgen. De meest gebruikte zijn `delimiter=<teken>`, die aangeeft welk `<teken>` tussen twee velden staat (default is de komma), en `quotechar=<teken>`, die aangeeft welk `<teken>` gebruikt wordt om strings mee te omsluiten (default is het dubbele aanhalingsteken).

26.1.2 CSV `writer()`

Het schrijven van een CSV bestand is slechts een beetje moeilijker dan het lezen. Je creëert een handle voor een CSV bestand dat je wilt schrijven door het te openen in `"w"` modus, en je geeft de handle mee aan de `writer()` functie van de `csv` module. Het object dat geretourneerd wordt door `writer()` heeft een methode `writerow()` die je kunt aanroepen met een list met velden, die dan in het uitvoerbestand worden weggeschreven in CSV formaat.

De aanroep van `writer()` kan dezelfde argumenten krijgen als `reader()`, inclusief een `delimiter` en een `quotechar`. Daarnaast kun je ook nog een `quoting=<quotemethode>` argument opgeven, waarbij `<quotemethode>` één van de volgende waardes heeft:

- `csv.QUOTE_ALL`, die ervoor zorgt dat elk veld tussen `quotechars` geplaatst wordt
- `csv.QUOTE_MINIMAL`, die ervoor zorgt dat alleen velden door `quotechars` omsloten worden als dat absoluut noodzakelijk is (dit is de default)
- `csv.QUOTE_NONNUMERIC`, die ervoor zorgt dat alle velden die geen integers of floats zijn omsloten worden door `quotechars`
- `csv.QUOTE_NONE`, die ervoor zorgt dat geen enkel veld door `quotechars` omsloten wordt

Het omsluiten van een veld met `quotechars` is alleen nodig als de string buitengewone tekens bevat, zoals newline tekens, of tekens die ook als `delimiter` worden gebruikt.

listing2602.py

```
from csv import writer

fp = open( "pc_writetest.csv", "w", newline='' )
csvwriter = writer( fp )
csvwriter.writerow( ["FILM", "SCORE"] )
csvwriter.writerow( ["Monty Python and the Holy Grail", 8] )
csvwriter.writerow( ["Monty Python's Life of Brian", 8.5] )
csvwriter.writerow( ["Monty Python's Meaning of Life", 7] )
fp.close()
```

Opgave Nadat je de code hierboven hebt gebruikt om het bestand "pc_writetest.csv" te creëren, open je het bestand en gebruikt `reader()` om de inhoud ervan te tonen.

26.2 Pickling

Stel dat je een bepaalde data structuur in een bestand wilt opslaan, bijvoorbeeld een list van tuples. Je kunt dat doen door de tuples naar strings te converteren en die op te slaan in een bestand, met iedere tuple op een eigen regel. Als je dan later de data structuur weer wilt opbouwen, moet je het bestand inlezen, de regels uiteen rafelen, en de list van tuples weer opbouwen. Je kunt je voorstellen dat dit een behoorlijk bewerkelijke aanpak is die een lastig stuk code vereist.

Gelukkig hoef je deze code niet te schrijven. Python biedt een oplossing voor het opslaan van data structuren in bestanden, waarbij zowel de structuur als de inhoud wordt opgeslagen. Dit wordt "pickling" genoemd.[25] Je kunt de hele data structuur in één keer in een bestand wegschrijven door het bestand voor schrijven te openen in binaire modus, en dan de functie `dump()` van de module `pickle` aan te roepen met de data structuur als eerste argument, en de handle als tweede argument.

listing2603.py

```
from pickle import dump

kaaswinkel = [ ("Roquefort", 12, 15.23),
    ("White Stilton", 25, 19.02), ("Cheddar", 5, 0.67) ]

fp = open( "pc_cheese.pck", "wb" )
dump( kaaswinkel, fp )
fp.close()

print( "uitvoer gereed" )
```

[25] In het Nederlands betekent dit "conservering," maar ik denk dat niemand begrijpt waar je het over hebt als je voor Python deze techniek zo noemt. Een leukere vertaling zou "pekelen" zijn, maar aangezien pekelen met zout gebeurt terwijl "pickling" met zuur gebeurt (augurken heten bijvoorbeeld "pickles" in het Engels), is ook die vertaling te verwarrend. Net als "inmaken."

Om de inhoud van een "pickle" bestand te lezen, gebruik je de functie `load()` van de `pickle` module. `load()` krijgt de handle van het bestand, geopend in binaire lees-modus, als argument.

listing2604.py

```
from pickle import load

fp = open( "pc_cheese.pck", "rb" )
buffer = load( fp )
fp.close()

print( type( buffer ) )
print( buffer )
```

Je kunt zien dat `load()` de data structuur netjes opnieuw opbouwt.

Het werkt zelfs voor classes die je zelf definieert:

listing2605.py

```
from pickle import dump, load

class Punt:
    def __init__( self, x, y ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({},{})".format( self.x, self.y )

p = Punt( 2, 5 )
fp = open( "pc_point.pck", "wb" )
dump( p, fp )
fp.close()

fp = open( "pc_point.pck", "rb" )
q = load( fp )
fp.close()

print( type( q ) )
print( q )
```

26.3 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is een bestandsformaat dat veel gebruikt wordt in hedendaagse applicaties, vooral applicaties waarin via web services gecommuniceerd wordt. Het wordt door veel programmeertalen ondersteund (waaronder natuurlijk Javascript). JSON lijkt op pickling in de zin dat het objecten die in het geheugen van het computer bestaan opslaat in bestanden, waarbij de structuur bewaard blijft. Een verschil

tussen pickling en JSON is dat JSON objecten opslaat op een manier die mensen kunnen lezen.

De `json` module werkt, net als de `pickle` module, met een `dump()` functie die data structuren naar een bestand schrijft, en een `load()` functie die data structuren inlaadt uit een bestand. Het bestand moet een tekstbestand zijn, en geen binair bestand.

listing2606.py

```
from json import dump, load

kaaswinkel = [ ("Roquefort", 12, 15.23),
    ("White Stilton", 25, 19.02), ("Cheddar", 5, 0.67) ]

fp = open( "pc_cheese.json", "w" )
dump( kaaswinkel, fp )
fp.close()

fp = open( "pc_cheese.json", "r" )
buffer = load( fp )
fp.close()

print( type( buffer ) )
print( buffer )
```

Alternatieven voor `dump()` en `load()` zijn de functies `dumps()` en `loads()`, die geen handle als argument krijgen. In plaats daarvan krijgt `dumps()` niks in de plaats van het handle argument, maar retourneert een string die de data structuur in JSON formaat bevat, terwijl `loads()` een string krijgt als argument in plaats van een handle, en de data structuur laadt uit die string.

Deze functies kunnen een groot aantal optionele argumenten krijgen die precies bepalen hoe de data wordt opgeslagen; bijvoorbeeld je kunt een `indent=` argument aan `dump()` en `dumps()` geven die bepaalt hoe ver wordt ingesprongen, en je kunt argumenten meegeven die de data in de dump sorteren. Raadpleeg de Python documentatie als je hier meer van wilt weten.

Een zwakte van de `json` module is dat alleen standaard Python data structuren ondersteund worden. Als je eigengemaakte classes wilt opslaan, moet je ze eerst converteren naar de standaard Python structuren. De `json` module biedt speciale `JSONencoder` en `JSONdecoder` classes die daarbij kunnen helpen. Het gaat te ver om die hier te bediscussiëren.

26.4 HTML en XML

HTML en XML zijn standaardformaten die gebruikt worden om informatie op webpagina's te tonen. Ze bestaan uit leesbare tekst, waarin formatteringsinstructies zijn opgenomen. Data analisten moeten regelmatig data "schrapen" uit webpagina's. Je kunt daar reguliere expressies voor gebruiken, maar als de pagina's redelijk fatsoenlijk geformatteerd zijn, kun je de "Beautiful Soup" module gebruiken.

De Beautiful Soup module wordt in Python `bs4` genoemd (`bs3` kwam ervoor, en er kunnen meer updates volgen). De module bevat de `BeautifulSoup` class die je kunt gebruiken om HTML en XML bestanden te laden. `bs4` wordt niet standaard door Python ondersteund; je moet het apart installeren, wat een beetje gedoe geeft, tenzij je gebruik maak van het `pip` programma dat wel standaard met Python 3 wordt meegeleverd.

Een alternatief voor Beautiful Soup is de `lxml` module, maar de eerstgenoemde is populairder.

Omdat dit soort modules apart installaties nodig hebben, ga ik ze niet hier beschrijven. Ik wil alleen aangeven dat als je data uit webpagina's wilt halen (en dat moet je inderdaad waarschijnlijk doen op een bepaald moment), je beter eerst de standaard hulpmiddelen die beschikbaar zijn kunt bestuderen voordat je je werpt op het ontwerpen van excentrieke reguliere expressies.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- Lezen en schrijven van CSV bestanden met de `csv` module
- Pickling met de `pickle` module
- Lezen en schrijven van JSON bestanden met de `json` module
- De beschikbaarheid van hulpmiddelen voor web schrapen

Opgaves

Opgave 26.1 Open het bestand "pc_inventory.csv" en lees de inhoud via de CSV `reader()`. Schrijf de inhoud naar een ander CSV bestand, waarbij je een spatie gebruikt als `delimiter`, en enkele aanhalingstekens als `quotechars`. Open het bestand in text modus en toon de inhoud om die te controleren.

Opgave 26.2 Laad de inhoud van het bestand "pc_inventory.csv," en stop die in een list van lists (iedere regel uit het bestand is één van de lists in de list van lists). Sla de list van lists op in JSON formaat. Open het bestand in text modus en toon de inhoud om die te controleren.

Hoofdstuk 27

Diverse Nuttige Modules

Op dit punt in het boek heb ik min of meer alles wat je echt moet weten om een goede Python programmeur – of misschien zelfs een goede programmeur in het algemeen – te zijn aan de orde gebracht. Ik wil nu nog snel een aantal nuttige modules naar voren brengen die geen eigen hoofdstuk nodig hebben. Ik ga niet veel details geven; als je eenmaal weet wat het nut van een module is, kun je er meer informatie over vinden in de Python handleiding.

27.1 `datetime`

De `datetime` module bevat functies die je berekeningen laten uitvoeren met datum en tijd. De module bevat een aantal classes die daarbij helpen, waarvan de meest belangrijke zijn: `datetime`, `timedelta`, `date`, en `time`. `datetime` bevat attributen `year` (jaar), `month` (maand), `day` (dag), `hour` (uur), `minute` (minuut), `second` (seconde), `microsecond` (duizendste seconde), en `tzinfo` (tijdzone). `date` en `time` bevatten een deelverzameling van deze attributen. Objecten die instanties zijn van deze classes zijn onveranderbaar.

Ik beperk mij hier tot het bespreken van `datetime` en `timedelta`, maar er bestaan equivalente functies en methodes voor de andere classes.

`datetime` objecten bevatten een datum en een tijd. Sommige van de methodes in `datetime` objecten zijn:

- `now()` creëert een `datetime` object dat de huidige datum en tijd bevat. Je roept deze methode typisch aan met een class call om een waarde voor `now()` te krijgen.
- `datetime()` creëert een `datetime` object met de opgegeven argumenten. De eerste drie argumenten zijn niet optioneel, en bevatten in volgorde numerieke waardes voor jaar, maand, en dag. De andere attributen (uur, minuut, seconde, duizendste seconde, en tijdzone) zijn optioneel. Argumenten kun je ofwel in de voorgenoemde volgorde geven, of via de syntax `<argument>=<waarde>`, waarbij `<argument>` de naam van een attribuut is zoals hierboven beschreven.

```
from datetime import datetime

print( datetime.now() )
```

Als je `datetime` objecten afdrukt krijg je een specifiek formaat als uitvoer. Als je dit formaat wilt wijzigen (inclusief het afdrukken van de naam van de dag in de week) dan kun je in de `datetime` module functies vinden die dat mogelijk maken. Zie de Python handleiding voor meer informatie.

Om met `datetime` objecten te rekenen, heb je `timedelta` nodig. Een `timedelta` object specificeert het verschil tussen twee `datetime` objecten. Een `timedelta` object bevat `days` (dagen), `seconds` (seconden), en `microseconds` (duizendste seconden). Je kunt ook andere attributen in een `timedelta` object vinden, die tijdverschil op andere manieren uitdrukken, maar de enige drie die het opslaat zijn de drie die ik hier noem; andere attributen worden uit deze drie berekend.

Je kunt allerlei berekeningen uitvoeren met `timedelta` objecten, maar de meest nuttige behandelen het verschil tussen twee `datetime` objecten. Je kunt dus een `timedelta` object optellen bij een `datetime` object om een nieuw `datetime` object als uitkomst te krijgen, of je kunt twee `datetime` objecten van elkaar aftrekken om het verschil te krijgen als een `timedelta` object.

listing2701.py

```
from datetime import datetime, timedelta

ditjaar = datetime.now().year
xmasditjaar = datetime( ditjaar, 12, 25, 23, 59, 59 )
dezedag = datetime.now()
dagen = xmasditjaar - dezedag

if dagen.days < 0:
    print( "Kerst komt volgend jaar weer." )
elif dagen.days == 0:
    print( "Het is Kerst!" )
else:
    print( "Slechts", dagen.days, "dagen tot Kerst!" )
```

27.2 `collections`

De `collections` module bevat handige classes die je helpen om iterabelen te manipuleren, zoals strings, tuples, lists, dictionaries, en sets. `collections` biedt interessante functionaliteiten, waarvan de meeste enigszins excentriek zijn, wat het onwaarschijnlijk maakt dat je ze snel nodig hebt. Ik noem de twee die ik het meest gebruikt zie worden, namelijk de `Counter` class en de `deque` class.

Een `Counter` object lijkt op een dictionary, die elementen bevat in de vorm van keys, en voor ieder van de elementen een "telling" als waarde. Je creëert een `Counter` object door bij creatie het als argument een sequentie te geven waarvan je de elementen wilt tellen. Wanneer het gecreëerd hebt, kun je nuttige methodes gebruiken, zoals:

- `most_common()` krijgt als argument een integer, en retourneert een list met die elementen die de hoogste "telling" hebben, zoveel als het integer argument aangeeft.

De elementen van de list zijn 2-tuples, waarbij het eerste element van iedere tuple een van de getelde elementen is, en het tweede element de corresponderende telling. Ze zijn geordend van hoogste naar laagste telling. Als geen integer argument is meegegeven, bevat de list alle elementen.

- `update()` krijgt een iterabele als argument en telt de elementen van die nieuwe iterabele op bij de tellingen die al in het object staan.

listing2702.py

```
from collections import Counter

data = [ "appel", "banaan", "appel", "banaan", "appel", "kers" ]
c = Counter( data )
print( c )
print( c.most_common( 1 ) )

data2 = [ "mango", "kers", "kers", "kers", "kers" ]
c.update( data2 )
print( c )
print( c.most_common() )
```

Een `deque` object is een list die je moet gebruiken als een "queue," dat wil zeggen, een list waarbij elementen toegevoegd en verwijderd worden aan alleen de uiteinden van de list. Een `deque` object ondersteunt methodes die gelijk zijn aan de gebruikelijke list-methodes, zoals `append()`, `extend()`, en `pop()`, maar die bij een `deque` object alleen werken aan het "rechter-uiteinde" van de list (bij de hoogste index). Daarnaast bevat hij ook gelijksoortige methodes die werken aan het "linker-uiteinde" van de list (bij index 0), namelijk `appendleft()`, `extendleft()`, en `popleft()`. Voor de rest zijn de methodes zoals je zou verwachten. Je creëert een `deque` object met de iterabele die je in een `deque` wilt veranderen als argument.

```
from collections import deque

dq = deque( [ 1, 2, 3 ] )
dq.appendleft( 4 )
dq.extendleft( [ 5, 6 ] )
print( dq )
```

27.3 urllib

De `urllib` module geeft je de mogelijkheid om webpagina's te benaderen zoals je bestanden benadert. Er zijn twee modules die van belang zijn: `urllib.request` bevat functies om informatie op het Internet te benaderen, en `urllib.error` bevat definities van exceptions die gegenereerd kunnen worden. Je kunt `urllib` ook gebruiken om te communiceren met webpagina's; als je dat wilt doen, moet je de `urllib.parse` module bestuderen. Ik geef hier alleen een eenvoudig voorbeeld waarin de inhoud van een webpagina wordt gelezen.

listing2703.py

```
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
from sys import exit

try:
    u = urlopen( "http://www.python.org" )
except HTTPError as e:
    print( "HTTP Error", e )
    sys.exit()
except URLError as e:
    print( "URL error", e )
    sys.exit()

for i in range( 5 ):
    tekst = u.readline()
    print( tekst )

u.close()
```

Merk op dat van `urllib` alleen `urlopen` geïmporteerd hoeft te worden. Zodra je een webpagina hebt geopend, beschik je over een handle, waarop je de reguliere methodes kunt gebruiken die zijn uitgelegd in hoofdstuk 16.

27.4 `glob`

De `glob` module verschaft een functie `glob()` waarmee je een list van bestandsnamen kunt produceren, gebaseerd op een zoek-patroon dat als argument wordt meegegeven. Het zoek-patroon wordt geschreven volgens Unix conventies, waarvan de meeste ook op andere systemen gelden. Deze zijn als volgt:

- Een vraagteken (?) in een bestandsnaam duidt op een willekeurig teken
- Een ster (*) in een bestandnaam duidt op een sequentie van nul of meer willekeurige tekens
- Een reeks van tekens tussen vierkante haken ([]) duidt op een willekeurig teken uit deze reeks; een streepje tussen twee van de tekens duidt op een reeks die begint bij het linkerteken en eindigt bij het rechterteken

Bijvoorbeeld, het patroon `"A[0-9]?B.*"` zoekt alle bestanden die beginnen met de letter A, gevolgd door een cijfer, gevolgd door een willekeurig teken, gevolgd door een B, met een willkeurige extensie. Het hangt af van het besturingssysteem of dit patroon onderscheid maakt tussen hoofd- en kleine letters.

Verwar een dergelijk patroon niet met een reguliere expressie. Oppervlakkig gezien lijken ze op elkaar (zoals het feit dat een * een serie willekeurige tekens weergeeft), maar ze zijn compleet verschillend. Dit soort bestand-zoek-patronen ondersteunen alleen de speciale tekens en reeksen die hierboven staan (en die voor reguliere expressies soms een andere betekenis hebben), en je gebruikt ze alleen voor `glob` of wanneer je rechtstreeks met het besturingssysteem communiceert in de command shell.

```
from glob import glob

glist = glob( "*.pdf" )
for name in glist:
    print( name )
```

De glob module bevat ook een functie iglob(), waarvan de functionaliteit gelijk is aan de functionaliteit van glob(), maar die een iterator retourneert in plaats van een list.

Opgave Gebruik glob() om de namen van alle Python bestanden in de huidige directory te tonen.

27.5 statistics

De statistics module geeft je toegang tot diverse statistische functies. Al deze functies krijgen als argument een sequentie of iterator met getallen (integers of floats).

- mean() berekent het gemiddelde van een sequentie van getallen
- median() berekent de mediaan van een sequentie van getallen, dat wil zeggen, het "middelste" getal
- mode() berekent de modus van een sequentie van getallen, dat wil zeggen, het getal dat het meest voorkomt
- stdev() berekent de standaard deviatie van een sequentie van getallen
- variance() berekent de variantie van een sequentie van getallen

Er zitten nog meer functies in de statistics module, maar de bovengenoemde zijn het meest gebruikt. Voor geavanceerde statistische berekeningen zijn andere modules beschikbaar, maar die noem ik niet in dit boek.

Deze functies kunnen een StatisticsError genereren. Dit is relevant voor de mode() functie, omdat deze exception gegenereerd wordt als er geen unieke modus is.

listing2704.py

```
from statistics import mean, median, mode, stdev, variance, \
    StatisticsError

data = [ 4, 5, 1, 1, 2, 2, 2, 3, 3, 3 ]

print( "gemiddelde:", mean( data ) )
print( "mediaan:", median( data ) )
try:
    print( "modus:", mode( data ) )
except StatisticsError as e:
    print( e )
print( "st.dev.: {:.3f}".format( stdev( data ) ) )
print( "variantie: {:.3f}".format( variance( data ) ) )
```

Merk op dat voor een sequentie met een even aantal getallen, de mediaan het gemiddelde is van de twee "middelste" getallen. Sommige mensen prefereren een andere manier van omgaan met de mediaan bij een even aantal getallen; als je een andere aanpak wilt, bestudeer dan andere functies in de `statistics` module.

Wat betreft de modus: in de literatuur kun je verschillende definities vinden van wat de modus is. De meest gebruikte definitie is "het meest voorkomende getal," maar wat als er meerdere getallen zijn die alle het meest voorkomen? Wat als ieder getal uniek is? De versie van de modus in de `statistics` module lijkt niet de meest gebruikte te zijn.

Wat je geleerd hebt

In dit hoofdstuk is het volgende besproken:

- `datetime` module
- `collections` module
- `urllib` module
- `glob` module
- `statistics` module

Opgaves

Opgave 27.1 Gebruik de `Counter` class om de vijf meest voorkomende letters in een tekst te tonen, inclusief hun tellingen.

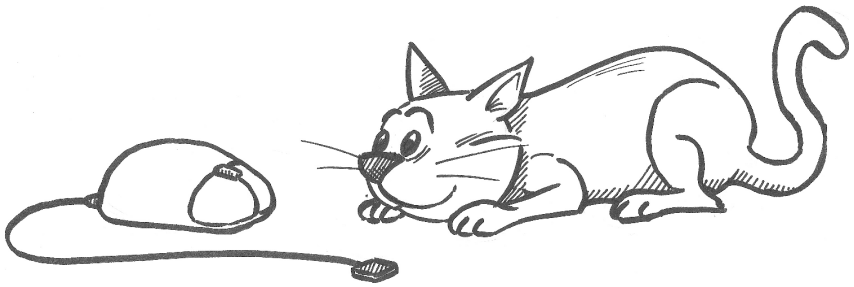
Opgave 27.2 Creëer een programma dat de gebruiker vraagt om getallen, totdat de gebruiker een nul ingeeft. Daarna toont het programma het gemiddelde, de mediaan, en de modus van de getallen. Je kunt de `statistics` module gebruiken voor het gemiddelde en de mediaan; voor de modus toon je echter alle getallen die het meest voorkomen, zelfs als dat betekent dat je meer dan één getal moet tonen. Per definitie moet een getal dat modus is minstens twee keer voorkomen; als ieder getal uniek is, is er geen modus. Hint: Je kunt de `Counter` class gebruiken om de modus te construeren.

Epiloog

Als je het hele boek hebt doorgewerkt en je bent in staat geweest om de meeste opgaves zelf te maken: Gefeliciteerd! Je bent nu een programmeur. Je hebt de basale kennis geïnternaliseerd die toepasbaar is op iedere programmeertaal, en je kunt nu de meeste programmeerproblemen oplossen die je aantreft in je carrière als student of professionele werknemer of zelfstandige. Als je een andere programmeertaal wilt leren, heb je nu een solide basis om dat snel te doen. Bovendien heb je geleerd om problemen te benaderen zoals programmeurs doen, wat een waardevolle capaciteit is waar je veel profijt van zult hebben.

Toekomstige edities van dit boek kunnen nog meer informatie en interessante problemen bevatten. Dat zal allemaal optioneel materiaal zijn, maar als je programmeren net zo leuk vindt als ik, wil je het misschien doornemen.

Als je opmerking hebt over de inhoud van de huidige versie van het boek, ontvang ik je email graag via `pythonbook@spronck.net`.



Appendix A

Problemen Oplossen

Ik krijg een ImportError als ik code probeer uit te voeren

Controleer de naam van de module die je probeert te importeren. Als het één van de standaard Python modules is, heb je waarschijnlijk een spelfout gemaakt. Of je hebt `.py` achter de naam gezet – dat moet je niet doen. Als de fout optreedt bij `pcinput` of `pcmaze`, dan heb je ofwel deze modules niet gebouwd of gedownload (zie appendix C of D om dit op te lossen), of je hebt ze geplaatst op een locatie waar Python ze niet kan vinden. Zorg dat je ze kopieert naar dezelfde plaats als waar je je Python programma's zet.

Ik krijg een FileNotFoundError: [Errno 2]

Je probeert een bestand te openen dat Python niet kan vinden. Wellicht ben je vergeten het volledige pad te vermelden in de bestandsnaam, of je denkt dat het bestand in de huidige directory staat terwijl dat niet zo is. Of misschien probeert je code één van de standaard tekstbestanden te openen die ik gebruik voor het boek, en je hebt die nog niet. Als dat het geval is, zie dan appendix E om ze te krijgen.

Ik krijg een SyntaxError maar ik heb geen idee wat ik fout heb gedaan

Als er meerdere syntax errors worden gerapporteerd, moet je proberen de fout die het eerst gerapporteerd wordt, als eerste op te lossen. Volgende fouten zijn vaak het gevolg van de eerste. Python rapporteert bij de fout de regel waarop de fout wordt aangetroffen. Controleer die regel. Bekijk ook de regel erboven: het is niet ongebruikelijk dat je een fout hebt gemaakt op een bepaalde regel, maar Python ziet de fout pas als het met de volgende regel bezig is. Syntax kleuren kunnen ook een indicatie geven waar je de fout hebt gemaakt. Gebruikelijke syntax fouten van beginnende programmeurs zijn het vergeten van een dubbele punt (`:`) na een **`if`**, **`while`**, of **`for`** statement, het maken van spelfouten in variabele namen, en fouten met tabulatie (inspringen).

Ik krijg een SyntaxError die een "Non-UTF-8 code" rapporteert

Je hebt een teken in je programma gebruikt dat niet door Python verwerkt kan worden. Bijvoorbeeld, misschien heb je je eigen naam in de code gezet (misschien zelfs alleen maar

in een commentaar-regel), en je naam wordt gespeld met een speciaal teken dat niet op het toetsenbord staat. Beperk je tot de tekens die op een US toetsenbord zitten. Het is niet zo dat je geen speciale tekens mag gebruiken, maar de regels om dat te doen worden in de latere hoofdstukken van het boek uitgelegd.

Ik krijg een vreemde fout zelfs als ik de voorbeeld code uitvoer

Zorg ervoor dat je Python 3.4 of een later versie gebruikt. Ik heb de code geschreven met Python 3.4, en heb vernomen dat sommige constructies niet goed werken in eerdere versies van Python.

Ik voer mijn programma uit maar het doet niks

Misschien staat er een eindeloze loop in je programma, dus het programma werkt wel, maar komt nooit toe bij het punt dat uitvoer geproduceerd wordt. Controleer je loops. Soms is het nuttig om een `print()` statement op te nemen bij het begin van je programma, zodat je kunt zien dat het programma daadwerkelijk is opgestart. `print()` statements in de code kunnen je ook helpen om te ontdekken waar het vastloopt.

Ik heb een functie (of class) gedefinieerd in mijn programma, maar het lijkt erop dat Python hem niet kan benaderen

Zorg dat je de aanroep van de class of functie correct gespeld hebt. Bedenk dat Python onderscheid maakt tussen hoofd- en kleine letters! Als de spelling correct is, wees er dan zeker van dat je niet een variabele hebt gecreëerd met dezelfde naam als de functie (of class). Als je dat hebt gedaan, interfereer je met de mogelijkheden van Python om je functie (of class) te benaderen.

Ik staar al uren naar mijn programma en ik kan het niet aan het werk krijgen

Soms is het goed om te pauzeren. Leg het programma weg, ga naar huis, speel een spelletje, doe fitness, neem een douche, wat je maar wilt. Neem het programma morgen weer op. Vraag het aan een willekeurige programmeur: soms lopen ze vast bij het ontwikkelen van een programma en kunnen een probleem niet oplossen, terwijl de oplossing onmiddellijk duidelijk is als ze de volgende dag op het werk arriveren. Wat kan helpen is om iemand anders bij je computer uit te nodigen en je probleem aan deze andere persoon uit te leggen. Vaak gebeurt het dat je, terwijl je het probleem uitlegt, plotseling ziet waar je een fout hebt gemaakt. Wat je echter zeker *niet* moet doen, is verder gaan met schrijven aan je programma zonder het probleem op te lossen. Daarmee maak je er alleen maar een grotere chaos van. Een veel beter idee is het programma te kopiëren en dan regels te verwijderen of code te simplificeren totdat je programma weer iets doet dat werkt. Dat geeft je tenminste een idee waar je je fout moet zoeken.

Appendix B

Verschillen met Python 2

Deze appendix bevat een overzicht van verschillen tussen Python 2 en Python 3, voor zover gerelateerd aan de inhoud van dit boek en voor zover ik ervan weet. Je kunt deze appendix negeren als je alleen Python 3 gaat gebruiken.

Operatoren

De delingsoperator (/) werkt in Python 2 anders dan in Python 3. In Python 3 wordt automatisch aangenomen dat als je een deling maakt, je floats moet gebruiken, en de deling neemt aan dat de getallen die gebruikt worden floats zijn, en levert altijd een float op als uitkomst. In Python 2 wordt aangenomen dat de uitkomst van de deling van het type is dat het "meest gedetailleerd" is van de getallen die erbij betrokken zijn, dat wil zeggen, als je twee getallen deelt en één ervan is een float, dan is de uitkomst van een deling ook een float. Maar als je twee integers deelt, is de uitkomst een integer (decimalen die mathematisch gezien er zouden moeten zijn worden weggelaten). Python 2 werkt op dit gebied zoals de meeste programmeertalen doen, maar zoals Python 3 werkt is meer intuïtief en leidt tot minder fouten.

Gereserveerde woorden

In Python 3 zijn **`print`** en **`exec`** niet langer gereserveerde woorden; het zijn nu functies. **`True`**, **`False`**, **`None`** en **`nonlocal`** zijn nu echter wel gereserveerde woorden.

Basale functies

Een klein verschil tussen Python 2 en Python 3 is dat als je de **`type`**`()` functie gebruikt, Python 3 het woord `class` toont, waar Python 2 het woord `type` toont. De reden is dat in Python 3 alle types geïmplementeerd zijn als classes.

De **`format`**`()` functie was in latere versies van Python 2 geïmplementeerd, maar bestond niet in de eerdere versies. In plaats daarvan werd een formatteerstijl ondersteund die "percentage-codes" gebruikt, zoals ook gebruikt wordt in talen zoals C++. Dit werd direct ondersteund door de **`print`**`()` functie, en bleef onderdeel van de **`print`**`()` functie zelfs

nadat **format**() toegevoegd was (om redenen van compatibiliteit). Het gevolg is dat in de meeste Python 2 programma's, zelfs de programma's die in een recente versie van Python 2 zijn geschreven, deze oudere manier van formatteren gebruikt wordt. De oudere aanpak wordt niet langer door Python 3 ondersteund. Overigens is het gebruik van haakjes bij de **print**() functie niet nodig voor Python 2, maar verplicht voor Python 3.

In Python 3 is de **range**() functie een iterator (zie hoofdstuk 23). Dit betekent dat het erg weinig geheugen gebruikt: het onthoudt alleen het laatst gegenereerde getal, de stapgrootte, en de limiet die bereikt moet worden. In Python 2 is **range**() anders geïmplementeerd: het produceert alle getallen in één keer in het geheugen. Dit betekent dat een statement als **range**(1000000000) in Python 2 zoveel geheugen nodig heeft dat je programma zou kunnen vastlopen. Dat gebeurt niet bij Python 3. In Python 2 is het daarom aan te raden om **range**() niet te gebruiken voor meer dan tienduizend getallen of zo, terwijl in Python 3 er geen restricties zijn.

In Python 2 zijn string manipulaties methodes die opgenomen zijn in de `string` module, en die je gewoonlijk aanroept door de module te importen en de methodes te gebruiken via de `string.<methode>()` syntax. Zulke aanroepen zijn niet langer nodig in Python 3.

Data structuren

In Python 2 geeft het sorteren van een list met een mengelmoes van data types een runtime error. De `sort()` functie ondersteunt ook een argument `cmp=<functie>`, dat je toestaat een functie op te geven die twee elementen met elkaar vergelijkt. Deze functie bestaat niet langer in Python 3, maar je kunt de key parameter voor hetzelfde doel gebruiken. In de Python module `functools` is een functie `cmp_to_key()` opgenomen die een oude-stijl `cmp` specificatie in een nieuwe-stijl `key` specificatie wijzigt.

In Python 2 retourneren de dictionary methodes `keys()`, `values()`, en `items()` een list in plaats van een iterator. Python 2 ondersteunt ook een methode `has_key()` die je kunt gebruiken om te controleren of een key in de dictionary zit, maar deze methode is verwijderd uit Python 3.

Python 2 ondersteunt sets niet als een basis-data structuur. Je moet de `sets` module importeren om ze te gebruiken. Je creëert bovendien een set met de `Set()` methode, en niet de **set**() functie. Om een set met elementen te creëren, is in Python 2 de enige manier om de elementen als een list mee te geven aan de `Set()` methode.

De structuur van exceptions is gewijzigd tussen Python 2 en Python 3. Specifiek is de exception `IOError` tot een alias gemaakt van de `OSError` exception, omdat het nogal lastig was om te onderscheiden welke specifieke problemen ieder van deze onderschept.

Unicode

Python 2 ondersteunt Unicode niet, maar Python 3 is erop gebaseerd. Python 3 strings zijn Unicode strings. Je merkt geen verschil zolang al je strings alleen ASCII tekens bevatten, maar Python 2 strings kunnen problemen krijgen als Unicode tekens in de strings gebracht worden. Python 3 ondersteunt ook Unicode in namen voor variabelen, functies, classes, en methodes, wat Python 2 niet toestaat. Ik raad je echter sterk af om in de namen van deze entiteiten tekens op te nemen die geen ASCII zijn.

Python 2 heeft geen byte strings. Die heeft Python 2 niet nodig, aangezien Unicode niet ondersteund wordt. Als je in Python 2 een teken wilt schrijven waarvan de numerieke waarde 0 is, gebruik je gewoon **`chr(0)`**. De `read()` en `write()` methodes voor binaire bestanden gebruiken reguliere strings in Python 2. Dit kan niet met Python 3.

In Python 2 werkt de `pickle` module op tekstbestanden. Dat kan niet langer in Python 3, omdat Python 3 Unicode ondersteunt. Pickle bestanden die met Python 2 gecreëerd zijn, kun je niet inladen in Python 3 en vice versa.

Iteratoren

Python 3 is veel meer gebaseerd op iteratoren en generatoren dan Python 2, wat een hoop voordelen heeft, vooral waar het snelheid en geheugengebruik betreft. Het gevolg is dat er een groot aantal verschillen bestaat tussen Python 2 en Python 3 op dit gebied. Ik heb ze niet allemaal onderzocht, maar hier zijn er een paar:

Iteratoren in Python 2 hebben een **`next()`** methode. Die hebben ze niet langer in Python 3, waar deze methode `__next__()` heet.

In Python 2 produceert **`zip()`** een list in plaats van een iterabele.

De `itertools` module kent ook een paar verschillen. Bijvoorbeeld, in Python 2 kent het een functie `izip()` die een iterabele produceert, maar omdat in Python 3 **`zip()`** dat zelf al doet, is `izip()` uit `itertools` verwijderd.

Modules

Naast `pickle` en `itertools` is ook `urllib` flink gewijzigd tussen Python 2 en Python 3.

De `statistics` module bestaat niet in Python 2.

Appendix C

pcinput.py

In veel opgaves in dit boek is het nuttig om een functie te hebben die gebruiker een input laat geven die voldoet aan specifieke eisen. Ik heb een module gecreëerd, `pcinput` geheten, die een aantal van die functies bevat. In veel van de opgaves en voorbeelden in dit boek veronderstel ik dat je die module beschikbaar hebt. Je kunt de module downloaden van `http://www.spronck.net/pythonbook`, of de code hieronder overnemen in een bestand "pcinput.py," ervoor zorgend dat deze is opgeslagen in dezelfde directory als waar je je eigen code schrijft.

Deze functies zijn wat lelijk omdat ze foutmeldingen geven als er iets mis is. Maar mooiere functies zouden lastiger zijn om te gebruiken (je moet dan exceptions afhandelen, en die behandel ik pas in hoofdstuk 17). Om Python te leren zijn ze uitstekend geschikt.

Ieder van de functies vraagt de gebruiker om een waarde in te geven van een bepaald type (float, integer, string, of hoofdletter), en retourneert die waarde. Je kunt de functies aanroepen met een string als argument, die dan als prompt gebruikt wordt.

pcinput.py

```
def getFloat( prompt ):
    while True:
        try:
            num = float( input( prompt ) )
        except ValueError:
            print( "Geen getal -- probeer het opnieuw" )
            continue
        return num

def getInteger( prompt ):
    while True:
        try:
            num = int( input( prompt ) )
        except ValueError:
            print( "Geen geheel getal -- probeer het opnieuw" )
            continue
        return num
```

```
def getString( prompt ):
    line = input( prompt )
    return line.strip()

def getLetter( prompt ):
    while True:
        line = input( prompt )
        line = line.strip()
        line = line.upper()
        if len( line ) != 1:
            print( "Geef precies een letter in" )
            continue
        if line < 'A' or line > 'Z':
            print( "Geef een letter van het alfabet in" )
            continue
        return line
```

Appendix D

pcmaze.py

In hoofdstuk 9 laat ik in een voorbeeld een doolhof doorzoeken. In dat voorbeeld gebruik ik een module `pcmaze`, die ik voor dit boek heb geschreven. De module bevat een doolhof, en geeft functies om eigenschappen van het doolhof op te vragen. Je kunt de module downloaden van `http://www.spronck.net/pythonbook`, of de code hieronder overnemen in een bestand "pcmaze.py," ervoor zorgend dat het in dezelfde directory staat als waar je je eigen code schrijft.

pcmaze.py

```
def connected( x, y ):
    if x > y:
        return connected( y, x )
    if (x,y) in ((1,5),(2,3),(3,7),(4,8),(5,6),(5,9),(6,7),
        (8,12),(9,10),(9,13),(10,11),(10,14),(11,12),(11,15),
        (15,16)):
        return True
    return False

def entrance():
    return 1

def exit():
    return 16
```

Appendix E

Test Tekstbestanden

In hoofdstuk 16 worden verscheidene korte tekstbestanden gebruikt om functionaliteiten te demonstreren. In het hoofdstuk 26 gebruik ik ook een CSV bestand als demonstratie. Je kunt deze bestanden downloaden via `http://www.spronck.net/pythonbook`, of je kunt ze zelf creëren. De inhoud van deze bestanden is hieronder gegeven: je kunt ze met een willekeurige tekst editor maken (zelfs met IDLE), waarbij je ze moet opslaan onder de naam die als titel gegeven is. De eerste is een kort citaat uit Shakespeare's *Romeo and Juliet* (vertaald door Jules Grandgagnage), de tweede is een variant op een bekende Engelstalige tongbreker, en de derde is een gedicht uit Lewis Carroll's *Through the Looking Glass* (vertaald door Alfred Kossman en C. Reedijk). De laatste is een CSV bestand dat ik heb gebouwd. De namen van de tekstbestanden heb ik gelijk gehouden aan die ik gebruikt heb voor de Engelstalige versie van dit boek.

pc_rose.txt

```
Het is slechts je naam die mijn vijand is;
Jij bent wie je bent, jezelf, niet een Montague.
Wat is een Montague? Is het een hand, een voet,
een arm, een gezicht, of enig ander deel
behorend aan een man? O, had je maar een andere naam!
Wat betekent een naam? Dat wat wij een roos noemen
zou met een andere naam net zo zoet geuren;
Zo zou ook Romeo, had hij een andere naam,
even volmaakt zijn als hij nu is.
Romeo, doe weg die naam,
Hij is geen deel van jou, verruil hem
voor alles wat ik ben.
```

pc_woodchuck.txt

```
Hoeveel hout kan een houthakker hakken
Als een houthakker hout kan hakken?
```

```
Hij kan hakken zoveel als hij kan hakken
En hij hakt zoveel als een houthakker kan hakken
Als een houthakker hout kan hakken.
```

pc_jabberwocky.txt

```
WAUWELWOK

't Wier bradig, en de spiramants
Bedroorden slendig in het zwiets:
Hoe klarm waren de ooiefants,
Bij 't bluifen der beriets.

Pas op de Wauwelwok, mijn kind!
Zo scherp getand, van klauw zo wreed!
Zorg dat Tsjoep-Tsjoep je nimmer vindt,
Vermijd de Barbeleet.

Hij nam zijn gnijpend zwaard ter hand:
Lang zocht hij naar den aarts-schavoest
Maar nam rust in lommers lust
Op een tumtumboomknoest.

En toen hij zat in diep gedenk,
Kwam Wauwelwok met vlammend oog,
Dwars door het bos met zwalpse zwenk,
Sluw borbelend wijl hij vloog.

Een, twee! Hup twee. En door en door
Ging kler de kling toen krissekruis.
Hij sloeg hem dood en blodd'rig rood
Bracht hij het tronie thuis.

Hebt gij versnaggeld Wauwelwok?
Kom aan mijn hart, o jokkejeugd!
O, heerlijkheid, fantabeltijd!
Hij knorkelde van vreugd.

't Wier bradig, en de spiramants
Bedroorden slendig in het zwiets:
Hoe klarm waren de ooiefants,
Bij 't bluifen der beriets.
```

pc_inventory.csv

```
ID,CATEGORIE,NAAM,VOORRAAD,PRIJS
1,Fruit,appel,1000,0.87
```

```
2,Fruit,banaan,2500,0.34
3,Fruit,kers,11225,0.07
4,Fruit,doerian,0,5.52
5,Kaas,Roquefort,46,12.23
6,Kaas,Blue Stilton,1,19.88
7,Kaas,Gouda,7,11.99
8,Fruit,aalbes,355,0.77
9,Fruit,mango,24,1.56
10,Kaas,Cheddar,333,13.15
```

Appendix F

Antwoorden

Deze appendix bevat de antwoorden bij de meeste opgaves. Al deze antwoorden zijn ook beschikbaar als bestanden, te downloaden via `http://www.spronck.net/pythonbook`.

Het is zinloos om deze antwoorden te bekijken als je niet zelf intensief hebt geprobeerd de opgaves op te lossen. Je kunt alleen programmeren leren door het te doen. Gebruik deze antwoorden alleen om ze te vergelijken met je eigen oplossingen, of als een laatste redmiddel als je geen idee hebt over hoe je een probleem moet aanpakken. Maar als je een probleem niet kunt oplossen, is het meestal beter om er een eerder deel van het boek op na te slaan om informatie op te zoeken die je niet hebt begrepen of vergeten bent.

Vaak zijn de antwoorden die ik geef slechts één van vele mogelijkheden om een opgave op te lossen. Als jij een andere manier hebt gevonden, kan dat best correct zijn, maar zorg ervoor dat je je oplossingen uitgebreid test om er zeker van te zijn dat ze correct zijn.

Bedenk dat, hoewel de antwoorden die ik geef meestal efficiënt zijn, efficiëntie niet een hoofddoel is wanneer je code schrijft. Je hoofddoel is om code te schrijven die een probleem oplost, en pas als dat gelukt is, moet je overwegen of de oplossing efficiënter gemaakt kan worden. Leesbaarheid en onderhoudbaarheid zijn veel belangrijker dan efficiëntie.

Hoofdstuk 1

Antwoord 1.1 Een recht-toe-recht-aan sortering die eerste de hoogste kaart zoekt, dan de op-één-na-hoogste, etcetera, heeft zes vergelijkingen nodig. Je kunt het bijvoorbeeld als volgt doen:

Nummer de kaarten van 1 tot 4, links naar rechts. Het nummer hoort bij de positie waar de kaart ligt, dus als je twee kaarten omwisselt, dan wissel je in feite hun nummers. Vergelijk kaarten 1 en 2 en verwissel ze als 1 hoger is dan 2. Vergelijk kaarten 2 en 3 en verwissel ze als 2 hoger is dan 3. Vergelijk kaarten 3 en 4 en verwissel ze als 3 hoger is dan 4. 4 is nu de kaart met het hoogste nummer. Vergelijk nu kaarten 1 en 2 weer, en verwissel ze als 1 hoger is dan 2. Vergelijk kaarten 2 en 3 weer, en verwissel ze als 2 hoger is dan 3. De op-één-na-hoogste kaart is nu 3. Tenslotte vergelijk je kaarten 1 en 2, en verwisselt ze als 1 hoger is dan 2. Je hebt nu de kaarten gesorteerd, waarbij je zes vergelijkingen nodig had.

Om het met minder vergelijkingen te doen, heb je een ietwat complexere procedure nodig. Begin weer met het nummeren van de kaarten, van 1 tot 4. Vergelijk kaarten 1 en 2 en verwissel ze als 1 hoger is dan 2. Vergelijk nu kaarten 3 en 4, en verwissel ze als 3 hoger is dan 4. Vergelijk dan 1 met 3. Als 1 hoger is dan 3, verwissel je zowel kaart 1 en 3, als kaart 2 en 4. De laagste kaart ligt nu op plek 1, en je weet ook dat kaart 3 lager is dan kaart 4. Vergelijk nu kaarten 2 en 3. Als 2 lager is dan 3 ben je klaar met slechts vier vergelijkingen, je hoeft niks meer te vergelijken of te verwisselen. Als 2 echter hoger is dan 3, moet je 2 en 3 omwisselen. Je moet dan nog steeds 3 en 4 vergelijken, en omwisselen als 3 hoger is dan 4. Maar dan ben je klaar, met slechts vijf vergelijkingen. Dus je kunt vier kaarten sorteren met minimaal vier en maximaal vijf vergelijkingen.

Misschien denk je nu slim te zijn en, als je 1 en 2 vergeleken (en misschien verwisseld) hebt, en 3 en 4 vergeleken (en misschien verwisseld) hebt, je 2 en 3 moet vergelijken, want als op dat moment 2 lager is dan 3, dan ben je klaar met slechts 3 vergelijkingen. Echter, als op dat moment 2 hoger blijkt te zijn dan 3, zou dat kunnen betekenen dat je toch zes vergelijkingen nodig hebt.

Als programmeren nieuw voor je is, dan zal het lastig zijn de procedure met maximaal vijf vergelijkingen te verzinnen. Raak dus niet ontmoedigd als je daar niet opkomt. Het is belangrijker dat je een taak weet op te lossen dan dat je hem op de meest efficiënte manier weet op te lossen.

Antwoord 1.2 Een goede aanpak is vier rechthoeken op een stuk papier te zetten en deze te nummeren, en dan een kaart in iedere rechthoek te plaatsen. Zeg tegen de persoon die de instructies gaat uitvoeren dat "vergelijk de kaart in rechthoek x met de kaart in rechthoek y" betekent dat hij of zij aan de processor moet vragen die kaarten op te rapen, te bekijken, en ze terug te leggen waar ze vandaan kwamen, en daarna aan te geven welke de hoogste van de twee is. Dan kun je de volgende instructies geven voor de simpele, zes-stappen procedure die ik hierboven beschrijf:

1. Vergelijk de kaart in rechthoek 1 met de kaart in rechthoek 2. Als de kaart in rechthoek 1 hoger is dan de kaart in rechthoek 2, verwissel ze dan.
2. Vergelijk de kaart in rechthoek 2 met de kaart in rechthoek 3. Als de kaart in rechthoek 2 hoger is dan de kaart in rechthoek 3, verwissel ze dan.
3. Vergelijk de kaart in rechthoek 3 met de kaart in rechthoek 4. Als de kaart in rechthoek 3 hoger is dan de kaart in rechthoek 4, verwissel ze dan.
4. Vergelijk de kaart in rechthoek 1 met de kaart in rechthoek 2. Als de kaart in rechthoek 1 hoger is dan de kaart in rechthoek 2, verwissel ze dan.
5. Vergelijk de kaart in rechthoek 2 met de kaart in rechthoek 3. Als de kaart in rechthoek 2 hoger is dan de kaart in rechthoek 3, verwissel ze dan.
6. Vergelijk de kaart in rechthoek 1 met de kaart in rechthoek 2. Als de kaart in rechthoek 1 hoger is dan de kaart in rechthoek 2, verwissel ze dan.

Het idee om genummerde rechthoeken te gebruiken om aan kaarten te refereren is vergelijkbaar met het gebruiken van variabelen in een computerprogramma. Variabelen worden uitgelegd in hoofdstuk 4. Het uitleggen van de meer efficiënte procedure die hierboven staat is een stuk lastiger, omdat je er geneste condities voor nodig hebt.

Hoofdstuk 2

Antwoord 2.1 Python draait nu op je computer. Gefeliciteerd!

Antwoord 2.2 Je ziet niets in de shell (behalve het woord `RESTART` dat je altijd ziet als je een programma draait). `7/4` is een correct Python commando, dus het programma geeft geen foutmelding. Het programma rekent `7/4` uit, maar toont geen resultaat, dus het programma laat niet 1.75 zien. De shell toont het resultaat van het draaien van het programma. Maar omdat het programma zelf geen resultaat heeft, is er ook niks dat de shell kan laten zien. Dus je ziet niets.

Hoofdstuk 3

Antwoord 3.1

answer0301.py

```
print( 60 * (0.6 * 24.95 + 0.75) + (3 - 0.75) )
```

Antwoord 3.2 Alle regels moeten `print( "Een boodschap" )` zijn (of hetzelfde maar met de string tussen enkele aanhalingstekens). De fout in de eerste regel is dat er een punt achter staat. Die punt moet weg. De fout in de tweede regel is dat er iets staat dat een string zou moeten zijn, maar dat start met een dubbel aanhalingsteken en eindigt met een enkel aanhalingsteken. Ofwel het dubbele aanhalingsteken moet een enkel aanhalingsteken worden, of andersom. De derde regel is strict genomen niet fout, maar waarschijnlijk was de bedoeling ervan dat de `f"` er niet zou staan.

Antwoord 3.3

answer0303.py

```
print( 1/0 )
```

Antwoord 3.4 Het probleem is dat er een enkel sluithaakje ontbreekt op de eerste regel code. Ik heb het haakje verwijderd dat ik origineel had geschreven na de 6, maar dat kun je niet meer weten; je kunt slechts de haakjes tellen en zien dat er een sluithaakje ontbreekt.

Het verwarrende is dat de foutboodschap zegt dat er een fout is gevonden op de tweede regel. De tweede regel is echter correct. De reden dat Python de tweede regel aanduidt als fout, is dat Python dat laatste sluithaakje niet heeft gevonden op de eerste regel, en dus verwacht dat de regel doorloopt op de tweede regel. Daarna ontdekt Python dat er iets mis is, en rapporteert de fout.

Dit kom je soms ook in je eigen code tegen: er wordt een fout gerapporteerd op een bepaalde regel, maar de echte fout is al gemaakt op een eerdere regel. Dit soort fouten betreffen vaak het ontbreken van haakjes of aanhalingstekens. Houd daar rekening mee.

Antwoord 3.5

answer0305.py

```
print( str( (14 + 535) % 24 ) + ".00" )
```

Hoofdstuk 4

Antwoord 4.1

answer0401.py

```
# Dit programma berekent het gemiddelde van 3 variabelen:
# var1, var2, en var3
var1 = 12.83
var2 = 99.99
var3 = 0.12
gemiddelde = (var1 + var2 + var3) / 3 # Berekening
print( gemiddelde ) # Ziet er wat lelijk uit, maar
# dat wordt beter na een discussie over formattering
```

Antwoord 4.2

answer0402.py

```
pi = 3.14159
straal = 12
print( "De oppervlakte van een cirkel met straal",
    straal, "is", pi * straal * straal )
```

Antwoord 4.3

answer0403.py

```
CENTEN_IN_DOLLAR = 100
CENTEN_IN_KWARTJE = 25
CENTEN_IN_DUBBELTJE = 10
CENTEN_IN_STUIVER = 5

bedrag = 1156
centen = bedrag

dollars = int( centen / CENTEN_IN_DOLLAR )
centen -= dollars * CENTEN_IN_DOLLAR
kwartjes = int( centen / CENTEN_IN_KWARTJE )
centen -= kwartjes * CENTEN_IN_KWARTJE
dubbeltjes = int( centen / CENTEN_IN_DUBBELTJE )
centen -= dubbeltjes * CENTEN_IN_DUBBELTJE
stuivers = int( centen / CENTEN_IN_STUIVER )
centen -= stuivers * CENTEN_IN_STUIVER
centen = int( centen )

print( bedrag / CENTEN_IN_DOLLAR, "bestaat uit:" )
print( "Dollars:", dollars )
print( "Kwartjes:", kwartjes )
print( "Dubbeltjes:", dubbeltjes )
print( "Stuivers:", stuivers )
```

```
print( "Centen:", centen )
```

Antwoord 4.4

answer0404.py

```
a = 17
b = 23
print( "a =", a, "en b =", b )
a += b
b = a - b
a -= b
print( "a =", a, "en b =", b )
```

Hoofdstuk 5

Antwoord 5.1

answer0501.py

```
s = input( "Geef een string: " )
print( "Je hebt", len( s ), "letters ingegeven" )
```

Antwoord 5.2

answer0502.py

```
from pcinput import getFloat
from math import sqrt

zijde1 = getFloat( "Geef de lengte van de eerste zijde: " )
zijde2 = getFloat( "Geef de lengte van de tweede zijde: " )
zijde3 = sqrt( zijde1 * zijde1 + zijde2 * zijde2 )
print( "De lengte van de diagonaal is {:.3f}.".format( zijde3 ) )
```

Antwoord 5.3

answer0503.py

```
from pcinput import getFloat

num1 = getFloat( "Geef nummer 1: " )
num2 = getFloat( "Geef nummer 2: " )
num3 = getFloat( "Geef nummer 3: " )

print( "De grootste is", max( num1, num2, num3 ) )
print( "De kleinste is", min( num1, num2, num3 ) )
print( "Het gemiddelde is", round( (num1 + num2 + num3)/3, 2 ) )
```

Antwoord 5.4

answer0504.py

```
from math import exp

s = "e tot de macht {:2d} is {:>9.5f}"
print( s.format( -1, exp( -1 ) ) )
print( s.format( 0, exp( 0 ) ) )
print( s.format( 1, exp( 1 ) ) )
print( s.format( 2, exp( 2 ) ) )
print( s.format( 3, exp( 3 ) ) )
```

Antwoord 5.5

answer0505.py

```
from random import random

print( "Een toevalsgetal tussen 1 en 10 is",
    1 + int( random() * 10 ) )
```

Hoofdstuk 6

Antwoord 6.1

answer0601.py

```
from pcinput import getFloat

grade = getFloat( "Geef een cijfer: " )
check = int( grade * 10 )
if grade < 0 or grade > 10:
    print( "Cijfers liggen tussen 0 en 10." )
elif check%5 != 0 or check != grade*10:
    print( "Cijfers zijn afgerond op halve punten." )
elif grade >= 8.5:
    print( "A" )
elif grade >= 7.5:
    print( "B" )
elif grade >= 6.5:
    print( "C" )
elif grade >= 5.5:
    print( "D" )
else:
    print( "F" )
```

Antwoord 6.2 De enig mogelijke antwoorden zijn “D” en “F,” aangezien de tests in de verkeerde volgorde zijn geplaatst. Bijvoorbeeld, als de score 85 is, dan is dat niet alleen groter dan 80.0, maar ook groter dan 60.0, dus de uitkomst zal dan “D” zijn.

Antwoord 6.3

answer0603.py

```
from pcinput import getString

s = getString( "Geef een string: " )
count = 0
if ("a" in s) or ("A" in s):
    count += 1
if ("e" in s) or ("E" in s):
    count += 1
if ("i" in s) or ("I" in s):
    count += 1
if ("o" in s) or ("O" in s):
    count += 1
if ("u" in s) or ("U" in s):
    count += 1

if count == 0:
    print( "Er zitten geen klinkers in de string." )
elif count == 1:
    print( "Er zit maar 1 verschillende klinker in de string." )
else:
    print( "Er zijn", count, "verschillende klinkers." )
```

Antwoord 6.4

answer0604.py

```
from pcinput import getFloat
from math import sqrt

a = getFloat( "A: " )
b = getFloat( "B: " )
c = getFloat( "C: " )

if a == 0:
    if b == 0:
        print( "Deze vergelijking heeft geen onbekende!" )
    else:
        print( "Er is 1 oplossing, namelijk", -c/b )
else:
    discriminant = b*b - 4*a*c
    if discriminant < 0:
        print( "Er zijn geen oplossingen" )
    elif discriminant == 0:
```

```
        print( "Er is 1 oplossing, namelijk", -b/(2*a) )
    else:
        print( "Er zijn 2 oplossingen, namelijk",
                (-b+sqrt(discriminant))/(2*a), "en",
                (-b-sqrt(discriminant))/(2*a) )
```

Hoofdstuk 7

Antwoord 7.1

answer0701.py

```
from pcinput import getInteger

num = getInteger( "Geef een nummer: " )
i = 1
while i <= 10:
    print( i, "*", num, "=", i*num )
    i += 1
```

Antwoord 7.2

answer0702.py

```
from pcinput import getInteger

num = getInteger( "Geef een nummer: " )
for i in range( 1, 11 ):
    print( i, "*", num, "=", i*num )
```

Antwoord 7.3

answer0703.py

```
from pcinput import getInteger

AANTAL = 10
grootste = 0
kleinste = 0
deelbaar3 = 0

for i in range( AANTAL ):
    num = getInteger( "Geef nummer "+str( i+1 )+": " )
    if num%3 == 0:
        deelbaar3 += 1
    if i == 0:
        kleinste = num
        grootste = num
        continue
```

```
    if num < kleinste:
        kleinste = num
    if num > grootste:
        grootste = num

print( "Kleinste is", kleinste )
print( "Grootste is", grootste )
print( "Deelbaar door 3 is", deelbaar3 )
```

Antwoord 7.4

answer0704.py

```
flessen = 10
s = "s"
while flessen != "geen":
    print( "{0} flesje{1} met bier op de muur, "\
        "{0} flesje{1} met bier.".format( flessen, s ) )
    flessen -= 1
    if flessen == 1:
        s = ""
    elif flessen == 0:
        s = "s"
        flessen = "geen"
    print( "Open er een, drink hem meteen, {} flesje{} "\
        "met bier op de muur.".format( flessen, s ) )
```

Als je een backslash (\) plaatst aan het einde van een regel code, dan wordt de volgende regel aan de regel met de backslash geplakt. Als hierdoor twee strings tegen elkaar aan komen te staan, dan worden die samen als één string beschouwd. Ik gebruik dat hier om ervoor te zorgen dat de code past binnen de breedte van de pagina. Ik vertel er meer over in hoofdstuk 10. Je hebt het zelf niet nodig: je kunt het hele `print()` statement op een lange regel schrijven.

Antwoord 7.5

answer0705.py

```
num1 = 0
num2 = 1
print( 1, end=" " )
while True:
    num3 = num1 + num2
    if num3 > 1000:
        break
    print( num3, end=" " )
    num1 = num2
    num2 = num3
```

Antwoord 7.6

answer0706.py

```
from pcinput import getString

woord1 = getString( "Geef woord 1: " )
woord2 = getString( "Geef woord 2: " )
gemeen = ""
for letter in woord1:
    if (letter in woord2) and (letter not in gemeen):
        gemeen += letter

if gemeen == "":
    print( "De woorden hebben geen tekens gemeen." )
else:
    print( "De woorden delen de volgende tekens:", gemeen )
```

Antwoord 7.7

answer0707.py

```
from random import random

DARTS = 100000
raak = 0
for i in range( DARTS ):
    x = random()
    y = random()
    if x*x + y*y < 1:
        raak += 1

print( "Een redelijke benadering van pi is", 4 * raak / DARTS )
```

Antwoord 7.8

answer0708.py

```
from random import randint
from pcinput import getInteger

antwoord = randint( 1, 1000 )
teller = 0
while True:
    poging = getInteger( "Raad een getal: " )
    if poging < 1 or poging > 1000:
        print( "Je moet een getal tussen de 1 en 1000 noemen" )
        continue
    teller += 1
    if poging < antwoord:
        print( "Hoger" )
```

```
    elif poging > antwoord:
        print( "Lager" )
    else:
        print( "Je hebt het geraden!" )
        break

if teller == 1:
    print( "Je hoefde maar 1 keer te raden. Geluksvogel." )
else:
    print( "Je moest", teller, "keer raden." )
```

Antwoord 7.9

answer0709.py

```
from pcinput import getLetter
from sys import exit

teller = 0
laagste = 0
hoogste = 1001
print( "Denk aan een getal tussen 1 en 1000." )

while True:
    poging = int( (laagste + hoogste) / 2 )
    teller += 1
    prompt = "Ik raad "+str( poging )+". Is jouw getal"+\
        " (L)ager of (H)oger, of is dit (C)orrect? "
    antwoord = getLetter( prompt )
    if antwoord == "C":
        break
    elif antwoord == "L":
        hoogste = poging
    elif antwoord =="H":
        laagste = poging
    else:
        print( "Antwoord H, L, of C." )
        continue
    if laagste >= hoogste-1:
        print( "Je moet een fout gemaakt hebben,",
            "want je hebt gezegd dat het getal hoger is dan",
            laagste, "maar ook lager dan", hoogste )
        print( "Ik stop ermee" )
        exit()

if teller == 1:
    print( "In 1 keer geraden! Ik kan gedachten lezen!" )
else:
    print( "Ik moest", teller, "keer raden." )
```

Antwoord 7.10

answer0710.py

```
from pcinput import getInteger

num = getInteger( "Geef een nummer: " )
if num < 2:
    print( num, "is niet priem" )
else:
    i = 2
    while i*i <= num:
        if num%i == 0:
            print( num, "is niet priem" )
            break
        i += 1
    else:
        print( num, "is priem" )
```

Ik heb in deze code een handigheidje gebruikt om de berekening veel sneller te laten lopen. De code test de delers tot aan de wortel uit `num` (dus het test tot het moment dat `i*i > num`). De reden is dat als er een getal is dat groter is dan de grens van deze test, dat een deler is van `num`, dan is er ook een getal dat kleiner is dan deze grens dat een deler is. Bijvoorbeeld, als `num` 21 is, ligt de wortel tussen 4 en 5. Het algoritme test daarom alleen getallen tot en met 4. Er is een deler voor 21 die groter is dan 4, namelijk 7. Maar dat betekent dat er ook een deler is die kleiner dan (of gelijk aan) 4 is, en die is er inderdaad, namelijk 3. Deze handigheid veroorzaakt een enorme versnelling van het algoritme ten opzichte van het testen van alle getallen tot `num`; bijvoorbeeld, als `num` in de buurt is van 1 miljoen, en het is priem, dan zou je ongeveer 1 miljoen getallen moeten testen om te concluderen dat het priem is als je deze grens niet zou aanhouden, terwijl mijn algoritme slechts duizend getallen test.

Als je deze handigheid gevonden hebt, geweldig! Zo niet, maar je geen zorgen: als je code werkt, gaat het je tot nu toe prima af. Het lastigste is code werkend te krijgen. Als je daarin slaagt, ben je op weg een goed programmeur te worden.

Een laatste opmerking: zorg dat je je code uitgebreid test! Het is gemakkelijk om fouten te maken met specifieke getallen. In dit geval, zorg dat je in ieder geval een negatief getal test, nul, 1 (alledrie geen priemgetallen), 2 (het eerste en enige even priemgetal), 3 (het laagste oneven priemgetal), diverse niet-priemgetallen, diverse priemgetallen, een paar grote priemgetallen, en getallen die een kwadraat zijn van een priemgetal (bijvoorbeeld 25). Nog beter is als je een loop schrijft rond je code die alle getallen test tussen, bijvoorbeeld, -10 en 100. Dan ben je echt met een intensieve test bezig.

Antwoord 7.11

answer0711.py

```
num = 9
print( ". |", end="" )
for i in range( 1, num+1 ):
    print( "{:>3}".format( i ), end="" )
```

```
print()
for i in range( 3*(num+1) ):
    print( "-", end="" )
print()
for i in range( 1, num+1 ):
    print( i, "|", end="" )
    for j in range( 1, num+1 ):
        print( "{:>3}".format( i*j ), end="" )
    print()
```

Antwoord 7.12

answer0712.py

```
for i in range( 1, 101 ):
    for j in range( 1, i ):
        for k in range( j, i ):
            if j*j + k*k == i:
                print( "{} = {}**2 + {}**2".format( i, j, k ) )
```

Een meer efficiënte versie van dit algoritme is:

answer0712a.py

```
from math import sqrt

for i in range( 1, 101 ):
    for j in range( 1, int( sqrt( i ) )+1 ):
        for k in range( j, int( sqrt( i ) )+1 ):
            if j*j + k*k == i:
                print( "{} = {}**2 + {}**2".format( i, j, k ) )
```

Deze reden dat dit een efficiënter algoritme is, en waarom het werkt, is hierboven beschreven voor de priemgetallen opgave.

Antwoord 7.13

answer0713.py

```
from random import randint

POGINGEN = 10000
DOBBELSTENEN = 5

succes = 0

for i in range( POGINGEN ):
    laatste = 0
    for j in range( DOBBELSTENEN ):
        waarde = randint( 1, 6 )
        if waarde < laatste:
```

```
            break
        laatste = waarde
    else:
        succes += 1

print( "De waarschijnlijkheid van een oplopende serie van vijf",
    "dobbelsteen worpen is {:.3f}".format( succes/POGINGEN ) )
```

Antwoord 7.14

answer0714.py

```
for A in range( 1, 10 ):
    for B in range( 10 ):
        if B == A:
            continue
        for C in range( 10 ):
            if C == A or C == B:
                continue
            for D in range( 1, 10 ):
                if D == A or D == B or D == C:
                    continue
                num1 = 1000*A + 100*B + 10*C + D
                num2 = 1000*D + 100*C + 10*B + A
                if num1 * 4 == num2:
                    print( "A={}, B={}, C={}, D={}".format(
                        A, B, C, D ) )
```

Dit programma genereert alle combinaties van A, B, C, en D die de puzzel oplossen. Er is er echter maar één. Het programma zoekt toch verder totdat alle combinaties getest zijn. Dat is gelukkig een erg snel proces. Als je het programma wilt afbreken als een oplossing gevonden is, is de beste aanpak een functie te schrijven die deze code bevat, en die functie af te breken als de oplossing gevonden is. Functies worden besproken in het volgende hoofdstuk.

Antwoord 7.15

answer0715.py

```
PIRATES = 5
kokosnoten = 0
while True:
    kokosnoten += 1
    stapel = kokosnoten
    for i in range( PIRATES ):
        if stapel % PIRATES != 1:
            break
        stapel = (PIRATES-1) * int( (stapel - 1) / PIRATES )
    if stapel % PIRATES == 1:
        break
print( kokosnoten )
```

Deze oplossing start met nul kokosnoten en blijft kokosnoten aan de stapel toevoegen totdat iedere piraat zijn deel kan nemen en 1 kokosnoot overhoudt, steeds het deel van de piraat en die ene kokosnoot van de stapel verwijderend. Om te testen of er een kokosnoot overblijft na de verdeling wordt de modulo operator gebruikt. Nadat alle piraten hun deel hebben genomen, is het probleem opgelost als er 1 kokosnoot overblijft na een eerlijke verdeling van de overgebleven stapel.

Vind je het niet verbazend dat je een probleem met zo'n lange beschrijving kunt oplossen met zo weinig code?

Antwoord 7.16 Ik geef eerst een oplossing die iedere Driehoekkruiper apart simuleert. Deze oplossing is gemakkelijk te begrijpen:

answer0716.py

```
from random import randint

NUMKRUIPERS = 100000
leeftijd = NUMKRUIPERS # Ze leven minstens een dag

for i in range( NUMKRUIPERS ):
    if randint( 0, 2 ): # Sterf niet op dag 1
        leeftijd += 1
        while randint( 0, 1 ): # Sterf niet
            leeftijd += 1

print( "{:.2f}".format( leeftijd / NUMKRUIPERS ) )
```

De tweede oplossing beschouwt de populatie als een geheel, en verdeelt hem in groepen. Op de eerste dag wordt een-derde van de groep afgehaald, en het programma telt 1 dag op bij de totale leeftijd voor ieder van de Kruipers in die groep. Op de tweede dag wordt de helft van de overgeblevenen afgehaald, en wordt 2 dagen voor ieder van hen opgeteld bij de totale leeftijd. De derde dag wordt weer de helft eraf gehaald, en bij de totale leeftijd wordt opgeteld dat zij drie dagen leefden. Dit gaat door totdat de populatie op is. Soms blijft er een enkele Kruiper over omdat de groep niet netjes verdeeld kan worden (omdat een Kruiper leeft of sterft, maar niet Schrödinger's Kat kan zijn). Zo'n Kruiper wordt per toeval aan de levenden of de stervenden toebedeeld. Je ziet dat deze code kan omgaan met enorm grote aantallen Driehoekkruipers zonder problemen.

answer0716a.py

```
from random import randint

NUMKRUIPERS = 1000000000
num = NUMKRUIPERS
dood = int( num / 3 )
if NUMKRUIPERS % 3:
    if randint( 0, NUMKRUIPERS % 3 ): # Kruiper over op dag 1
        dood += 1
leeftijd = dood # Dag-1 doden leven 1 dag
num -= dood
```

```
dag = 2
while num > 0:
    dood = int( num / 2 )
    num -= dood # Dood de helft
    if dood != num: # Er is er 1 over
        if randint( 0, 1 ): # Besluit of 1 sterft
            dood, num = num, dood # Verwissel waardes
    leeftijd += dood * dag
    dag += 1

print( "{:.2f}".format( leeftijd / NUMKRUIPERS ) )
```

Tenslotte geef ik hier een oplossing die ik niet suggereerde, maar die geweldig werkt. Hij is erg kort, ongelofelijk snel, en geeft een vrijwel exact antwoord. De oplossing berekent eenvoudigweg een benadering van $\frac{1}{3}+\frac{2}{3}(\frac{1}{2}(2+\frac{1}{2}(3+\frac{1}{2}(4+\frac{1}{2}(5+...)))))$. Het gebruikt slechts 100 termen van deze berekening, maar dat is meer dan genoeg om een zeer nauwkeurige benadering van het antwoord te krijgen. Dit is natuurlijk een berekening en geen simulatie, maar het is de wiskundige expressie die je feitelijk werd gevraagd op te lossen.

answer0716b.py

```
schatting = 1/3
rest = 2/3
for dagen in range( 2, 101 ):
    rest /= 2
    schatting += rest * dagen
print( "{:.2f}".format( schatting ) )
```

Het exacte antwoord is overigens $2\frac{1}{3}$ dagen; een benadering moet 2.33 of 2.34 geven.

Hoofdstuk 8

Antwoord 8.1

answer0801.py

```
from pcinput import getInteger

# tafel krijgt een integer als parameter. Het drukt de
# tafel van vermenigvuldiging voor deze integer af.
def tafel( n ):
    i = 1
    while i <= 10:
        print( i, "*", n, "=", i*n )
        i += 1

num = getInteger( "Geef een getal: " )
tafel( num )
```

Antwoord 8.2

answer0802.py

```
from pcinput import getString

# gemeen krijgt twee strings als parameters. Het retourneert
# het aantal tekens dat de strings gemeen hebben.
def gemeen( w1, w2 ):
    tekens = ""
    for letter in w1:
        if (letter in w2) and (letter not in tekens):
            tekens += letter
    return len( tekens )

woord1 = getString( "Geef woord 1: " )
woord2 = getString( "Geef woord 2: " )

num = gemeen( woord1, woord2 )
if num <= 0:
    print( "De woorden delen geen tekens." )
elif num == 1:
    print( "De woorden hebben 1 teken gemeen" )
else:
    print( "De woorden hebben", num, "tekens gemeen" )
```

Antwoord 8.3

answer0803.py

```
# gregoryLeibnitz benadert pi middels de Gregory-Leibnitz
# reeks. Hij krijgt 1 parameter, een integer, die aangeeft
# hoeveel termen berekend worden. De benadering wordt als
# float geretourneerd.
def gregoryLeibnitz( num ):
    appr = 0
    for i in range( num ):
        if i%2 == 0:
            appr += 1/(1 + i*2)
        else:
            appr -= 1/(1 + i*2)
    return 4*appr

print( gregoryLeibnitz( 50 ) )
```

Antwoord 8.4

answer0804.py

```
from pcinput import getFloat
from math import sqrt
```

```
# Deze functie lost een kwadratische vergelijking op.
# De parameters zijn numerieke waardes voor A, B, and C in de
# vergelijking Ax**2 + Bx + C = 0. Het retourneert drie waardes.
# De eerste is een integer 0, 1, of 2, die aangeeft hoeveel
# oplossingen er zijn. Daarna volgen de oplossingen.
# Zonder oplossingen zijn beide 0. 1 oplossing wordt geretourneerd
# als de eerste van de twee, en de tweede is 0.
def wortelformule( a, b, c ):
    if a == 0:
        if b == 0:
            return 0, 0, 0
        return 1, -c/b, 0
    discriminant = b*b - 4*a*c
    if discriminant < 0:
        return 0, 0, 0
    elif discriminant == 0:
        return 1, -b/(2*a), 0
    else:
        return 2, (-b+sqrt(discriminant))/(2*a), \
            (-b-sqrt(discriminant))/(2*a)

num, opl1, opl2 = wortelformule( getFloat( "A: " ),
    getFloat( "B: " ), getFloat( "C: " ) )
if num == 0:
    print( "Er zijn geen oplossingen" )
elif num == 1:
    print( "Er is 1 oplossing, namelijk", opl1 )
else:
    print( "Er zijn 2 oplossingen, namelijk:", opl1, "en", opl2 )
```

Antwoord 8.5

answer0805.py

```
from pcinput import getInteger

def getNummer( prompt ):
    while True:
        num = getInteger( prompt )
        if num < 0 or num > 1000:
            print( "Geef een getal tussen 1 en 1000" )
            continue
        return num

def main():
    while True:
        x = getNummer( "Geef nummer 1: " )
        if x == 0:
            break
```

```
        y = getNummer( "Geef nummer 2: " )
        if y == 0:
            break
        if x%y == 0 or y%x == 0:
            print( "Fout: de nummers mogen geen delers zijn" )
            return
        print( "Vermenigvuldiging van",x,"met",y,"geeft",x*y )
    print( "Tot ziens!" )

if __name__ == '__main__':
    main()
```

Antwoord 8.6

answer0806.py

```
# Berekent de faculteit van de paremeter n, een integer.
# Retourneert de uitkomst als integer.
def faculteit( n ):
    waarde = 1
    for i in range( 2, n+1 ):
        waarde *= i
    return waarde

# Berekent n boven k; n en k zijn integer parameters;
# Retourneert n boven k als integer (want dat is het altijd).
def binomiaalcoefficient( n, k ):
    if k > n:
        return 0
    return int( faculteit( n ) /
        (faculteit( k )*faculteit( n - k )) )

def main():
    print( faculteit( 5 ) )
    print( binomiaalcoefficient( 8, 3 ) )

if __name__ == '__main__':
    main()
```

Antwoord 8.7 De code probeert de retourwaarde van de functie te printen, maar omdat de functie geen retourwaarde heeft, wordt het woord **None** afgedrukt. Als je een functie maakt waarvan je de output wilt tonen, kun je ofwel de output in de functie printen en de functie aanroepen zonder de retourwaarde te gebruiken, ofwel je laat de functie de waarde retourneren en je print het in het hoofdprogramma. Niet allebei. Het geniet meestal de voorkeur om functies niet zelf te laten printen, want als ze een waarde retourneren maakt dat ze meer algemeen bruikbaar (bijvoorbeeld, als de gegeven functie de waarde retourneert, kan de oppervlakte van een driehoek vanaf dat moment elders in het programma in berekeningen gebruikt worden). Als je deze uitleg niet begrijpt, lees dan nogmaals hoofdstuk 5.

Hoofdstuk 9

Antwoord 9.1

answer0901.py

```
def fib( n ):
    if n <= 2:
        return 1
    return fib( n-1 ) + fib( n-2 )

print( fib( 20 ) )
```

Antwoord 9.2

answer0902.py

```
def fib( n, depth ):
    indent = 6 * depth * " "
    print( "{}fib({})".format( indent, n ) )
    if n <= 2:
        print( "{}return {}".format( indent, 1 ) )
        return 1
    value = fib( n-1, depth+1 ) + fib( n-2, depth+1 )
    print( "{}return {}".format( indent, value ) )
    return value

print( fib( 5, 0 ) )
```

Antwoord 9.3 Aangezien de reeks van Fibonacci net zo goed als iteratieve functie geïmplementeerd kan worden (dat heb je al in het vorige hoofdstuk gedaan), is het geen goed idee om er een recursieve functie van te maken. De redenen zijn dezelfde als voor de faculteit, wat in het hoofdstuk is uitgelegd. Een bijkomende reden is dat de recursieve definitie feitelijk alle termen van de sequentie meerdere keren berekent, wat je kunt zien als je naar de output van de tweede opgave kijkt, terwijl eenmalig berekenen voldoende kan zijn.

Antwoord 9.4

answer0904.py

```
def gcd( m, n ):
    if m % n == 0:
        return n
    return gcd( n, m%n )

print( gcd( 7*5*13, 2*3*7*11 ) )
```

De recursieve definitie spreekt over een kleinste en een grootste getal, maar het grappige is dat je daar geen rekening mee hoeft te houden bij het schrijven van de functie. Als je de functie aanroept met eerst de kleinste en dan de grootste, dan betekent dat alleen maar dat de functie een extra keer wordt aangeroepen. Verder geeft hij gewoon de juiste uitkomst.

Antwoord 9.5 Het probleem met deze code is dat als ik een string ingeef met twee foute tekens, dat het zichzelf twee keer recursief zal aanroepen, namelijk voor ieder fout teken een keer. Bijvoorbeeld, als de gebruiker `"route67"` ingeeft, volgt eerst een recursieve aanroep voor de `"6"`. Als de gebruiker dan een correcte string ingeeft, zou de functie moeten eindigen. Maar dat doet de functie niet. In plaats daarvan gaat de functie door met het controleren van de originele string, komt de `"7"` tegen, en roept dan zichzelf nogmaals aan voor weer een andere string.

Ik kan uitleggen hoe je dit probleem kunt oplossen door de code te wijzigen, maar de echte oplossing is dat je geen recursieve functies moet schrijven die interactie hebben met de gebruiker.

Antwoord 9.6

answer0906.py

```
GROOTTE = 4

def hanoi( p_van, p_tmp, p_naar, grootte ):
    if grootte == 1:
        print( "Schijf 1 van", p_van, "naar", p_naar )
        return 1
    stappen = hanoi( p_van, p_naar, p_tmp, grootte-1 )
    print( "Schijf", grootte, "van", p_van, "naar", p_naar )
    stappen += 1+hanoi( p_tmp, p_van, p_naar, grootte-1 )
    return stappen

stappen = hanoi( 'A', 'B', 'C', GROOTTE )
print( stappen, "stappen gedaan" )
```

Er is een iteratieve manier om deze puzzel op te lossen, namelijk de volgende. Herhaal deze twee stappen totdat de puzzel is opgelost: (1) Verplaats schijf 1 naar de volgende paal; (2) Verplaats een schijf die *niet* schijf 1 is naar een andere paal (er is altijd precies één schijf waarvoor dit kan). Het enige probleem dat overblijft is te bepalen of je schijf 1 "met de klok mee" of "tegen de klok in" moet verplaatsen. Als de grootte van de grootste schijf even is, moet je met de klok mee gaan (van A naar B, van B naar C, of van C naar A), en anders tegen de klok in (van A naar C, van C naar B, of van B naar A). Deze oplossing is snel en vermijdt recursie, wat betekent dat hij bruikbaar is voor grotere schijven dan de recursieve oplossing toestaat. Hij is echter complexer om te implementeren omdat je iedere paal moet representeren als een list (lists volgen in hoofdstuk 12), en je moet een manier bedenken om te bepalen dat de puzzel is opgelost (dat kan door simpelweg de stappen te tellen, dat wil zeggen, te tellen tot $2^N - 1$).

Hoofdstuk 10

Antwoord 10.1

answer1001.py

```
text = """En Sint Atilla hief de handgranaat ten hemel, en
```

```
zeide, "O Here, zegen deze handgranaat, zodat daarmee u uw
vijanden tot kleine stukjes kunt blazen, in uwer genade."
En de Here grinnikte. En het volk laafde zich aan lammeren,
en luiaards, en karpers, en anchovis, en orang oetans, en
cornflakes, en fruitvliegjes, en grote stu..."""

tela, tele, teli, telo, telu = 0, 0, 0, 0, 0
for c in text:
    if c.upper() == "A":
        tela += 1
    elif c.upper() == "E":
        tele += 1
    elif c.upper() == "I":
        teli += 1
    elif c.upper() == "O":
        telo += 1
    elif c.upper() == "U":
        telu += 1

print( "tels: a={}, e={}, i={}, o={}, u={}".format(
    tela, tele, teli, telo, telu ) )
```

Antwoord 10.2

answer1002.py

```
tekst = """En ze stu[re]n [i]ngekleurde prentbriefkaarten van
plekken waarvan ze zich niet reali[s]eren dat ze er nooit
geweest zijn [a]an 'Iedereen op nummer 22, weer is prachti[g],
onz[e] kamer is aa[n]gekruisd. Missen jullie. E[t]en[ ]i[s]
vettig, maar we hebben een geweldig leuk restaurantje gevonden
in de achterstraatjes waar ze Heine[ke]n hebben en kaas en
uien chips en iemand die "Een beetje verliefd" speel[t] op een
a[c]cordeon' en je zit vier dagen vast op Schip[h]ol voor je
vijfdaagse vliegvakantie met niks anders te eten dan
uitgedroogde voorverpakte boterhammen..."""

start = -1
while True:
    start = tekst.find( "[", start+1 )
    if start < 0:
        break
    eind = tekst.find( "]", start )
    if eind < 0:
        break
    print( tekst[start+1:eind], end="" )
    start = eind
print()
```

Antwoord 10.3

answer1003.py

```
ch = "A"
while ch <= "Z":
    print( ch, end=" " )
    ch = chr( ord( ch )+1 )
print()

for i in range( 26 ):
    rotr13 = (i + 13)%26
    ch = chr( ord( "A" ) + rotr13 )
    print( ch, end=" " )
```

Antwoord 10.4

answer1004.py

```
tekst = """Kapper Knap, de knappe kapper, knipt en kapt heel
knap, maar de knecht van kapper Knap, de knappe kapper,
knipt en kapt nog knapper dan kapper Knap, de knappe kapper."""

def schoon( s ):
    ns = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            ns += c
        else:
            ns += " "
    return ns

tel = 0
for woord in schoon( tekst ).split():
    if woord == "knap":
        tel += 1

print( "Aantal keren dat het woord \"knap\" voorkomt:", tel )
```

Antwoord 10.5

answer1005.py

```
tekst = "Hello, world!"
ntekst = ""
while len( tekst ) > 0:
    i = 0
    ch = tekst[i]
    j = 1
    while j < len( tekst ):
```

```
        if tekst[j] < ch:
            ch = tekst[j]
            i = j
        j += 1
    tekst = tekst[:i] + tekst[i+1:]
    ntekst += ch
print( ntekst )
```

Antwoord 10.6

answer1006.py

```
from sys import exit

zin = "en zo gebeurde het dat dat onze toevallige ontmoeting \
met de EErwaarde aRTHUR BElling een ommekeer betekende in ons \
leven, en vanaf dat moment gingen we iedere zondag naar \
de kerk van Sint sIMPEL bij Roombroodje MEt Jam."

# Test of het echt wel een zin is
if len( zin  ) <= 0:
    exit()

# Eerste letter hoofdletter
nieuwezin = zin[0].upper() + zin[1:]

woordlijst = nieuwezin.split()
laatstewoord = ""
nieuwezin = ""

for w in woordlijst:

    # Dubbele hoofdletters correctie
    if len( w ) > 2 and w[0] >= "A" and w[0] <= "Z" and \
        w[1] >= "A" and w[1] <= "Z" and w[2] >= "a" and \
        w[2] <= "z":
        w = w[0] + w[1].lower() + w[2:]

    # Dagen met een hoofdletter
    dag = w.lower()
    if dag == "zondag" or dag == "maandag" or dag == "dinsdag" \
        or dag == "woensdag" or dag == "donderdag" or \
        dag == "vrijdag" or dag == "zaterdag":
        w = dag[0].upper() + dag[1:]

    # CAPS LOCK correctie
    if w[0] >= "a" and w[0] <= "z":
        hoofdletters = True
        for c in w[1:]:
            if not (c >= "A" and c <= "Z"):
```

```
                hoofdletters = False
                break
        if hoofdletters:
            w = w[0].upper() + w[1:].lower()

    # Duplicaten verwijderen
    if w == laatstewoord:
        continue

    nieuwezin += w + " "
    laatstewoord = w

nieuwezin = nieuwezin.strip()
print( nieuwezin )
```

Hoofdstuk 11

Antwoord 11.1 De functie `toon_complex()` heb ik gemaakt om complexe getallen op een nette manier te tonen, en dat neemt de meeste code in beslag. Ik had niet gevraagd of je zo'n functie zou willen maken, en het is ook niet nodig.

answer1101.py

```
def plus_complex( c1, c2 ):
    return (c1[0] + c2[0], c1[1] + c2[1])

def toon_complex( c ):
    s = "("
    if c[1] == 0:
        return str( c[0] )
    elif c[0] != 0:
        s += str( c[0] )
        if c[1] > 0:
            s += "+"
    if c[1] != 1:
        if c[1] == -1:
            s += "-"
        else:
            s += str( c[1] )
    s += "i)"
    return s

num1 = (2,1)
num2 = (0,2)
print( toon_complex( num1 ), "+", toon_complex( num2 ), "=",
     toon_complex( plus_complex( num1, num2 ) ) )
```

Antwoord 11.2 Voor deze oplossing heb ik een minimalistische versie van `toon_complex()` gebruikt.

answer1102.py

```
def maal_complex( c1, c2 ):
    return (c1[0]*c2[0] - c2[1]*c1[1], c1[0]*c2[1] + c1[1]*c2[0])

def toon_complex( c ):
    return "({},{}i)".format( c[0], c[1] )

num1 = (2,1)
num2 = (0,2)
print( toon_complex( num1 ), "*", toon_complex( num2 ), "=",
     toon_complex( maal_complex( num1, num2 ) ) )
```

Antwoord 11.3

answer1103.py

```
inttuple = ( 1, 2, ( 3, 4 ), 5, ( ( 6, 7, 8, ( 9, 10 ), 11 ), 12,
    13 ), ( ( 14, 15, 16 ), ( 17, 18, 19, 20 ) ) )

def toon_inttuple( it ):
    for element in it:
        if isinstance( element, int ):
            print( element, end=" ")
        else:
            toon_inttuple( element )

toon_inttuple( inttuple )
```

Hoofdstuk 12

Antwoord 12.1

answer1201.py

```
from random import choice

antwoord = [ "Dat is zeker", "Het is zeker zo", "Zonder twijfel",
"Ja, zeker", "Je kunt erop vertrouwen", "Zoals ik het zie, ja",
"Waarschijnlijk", "Ziet er goed uit", "Ja", "Lijkt van wel",
"Vaag, probeer het nog eens", "Vraag later nog eens", "Kan ik \
beter niet zeggen", "Kan ik nu niet voorspellen", "Concentreer \
je en vraag nog eens", "Reken er maar niet op", "Ik zeg van \
niet", "Mijn bronnen zeggen van niet", "Lijkt er niet op",
"Zeer twijfelachtig" ]
```

```
input( "Stel je vraag aan de magische bol: " )
print( "De magische bol zegt:", choice( antwoord ) )
```

De `choice()` functie uit de `random` module selecteert een element van een list per toeval. Je kunt ook `randint()` gebruiken, waarbij je een index per toeval selecteert.

Antwoord 12.2

answer1202.py

```
from random import randint

stok = []
for value in ("Aas", "2", "3", "4", "5", "6", "7", "8",
    "10", "Boer", "Vrouw", "Heer"):
    for kleur in ("Harten", "Schoppen", "Klaveren", "Ruiten"):
        stok.append( kleur + ' ' + value )

for i in range( len( stok ) ):
    j = randint( i, len( stok )-1 )
    stok[i], stok[j] = stok[j], stok[i]

for kaart in stok:
    print( kaart )
```

Antwoord 12.3

answer1203.py

```
fifo = []
while True:
    k = input( "> " )
    if k == "":
        break
    if k != "?":
        fifo.append( k )
    elif len( fifo ) > 0:
        print( fifo.pop(0) )
    else:
        print( "Lijst is leeg" )
```

Antwoord 12.4

answer1204.py

```
tekst = """Let op, het is heel eenvoudig om je te verdedigen
tegen een man die gewapend is met een banaan. Eerst dwing je
hem de banaan te laten vallen; dan ontwapen je hem door de
banaan op te eten. Dat maakt hem onschadelijk."""
```

```
def tel_letter( x ):
    return x[0], -ord(x[1])

tellist = []
for i in range( 26 ):
    tellist.append( [0, chr(ord("a")+i)] )

for letter in tekst.lower():
    if letter >= "a" and letter <= "z":
        tellist[ord(letter)-ord("a")][0] += 1

tellist.sort( reverse=True, key=tel_letter )

for tel in tellist:
    print( "{:3}: {}".format( tel[0],tel[1] ) )
```

Antwoord 12.5 Er zijn twee manieren om dit probleem aan te pakken: ofwel je begint met een list met nummers, of je begint met een list met booleans en je beschouwt de index van een boolean als nummer. In de oplossing hier direct onder gebruik ik de methode met booleans. Ik laat de list met nul starten, omdat dat een hoop aftrekkingen scheelt, terwijl het slechts één boolean meer dan nodig in het geheugen plaatst. De methode is erg snel omdat ik met indices kan werken.

answer1205.py

```
nummers = 101 * [True]
nummers[1] = False
for i in range( 1, len( nummers ) ):
    if not nummers[i]:
        continue
    print( i, end=" " )
    j = 2
    while j*i < len( nummers ):
        nummers[j*i] = False
        j += 1
```

Hier is de alternatieve oplossingsmethode, waarbij ik de functie **range**() gebruik om de list te bouwen. Ik verwijder items van de list wanneer ik weet dat ze niet priem zijn. Omdat er erg veel list manipulaties nodig zijn, is deze methode iets trager dan de vorige, maar de snelheid neemt toe naarmate er meer en meer getallen van de list verwijderd worden.

answer1205a.py

```
MAXNUM = 100
nummers = list( range(2,MAXNUM+1) )
i = 0
while i < len( nummers ):
    j = i+1
    while j < len( nummers ):
        if nummers[j]%nummers[i]:
```

```
            j += 1
        else:
            nummers.pop(j)
    i += 1
for i in nummers:
    print( i, end=" " )
```

Antwoord 12.6

answer1206.py

```
from pcinput import getInteger

LEEG = "-"
SPELERX = "X"
SPELER0 = "0"
MAXZET = 9

def toon_bord( b ):
    print( "  1 2 3" )
    for rij in range( 3 ):
        print( rij+1, end=" ")
        for kol in range( 3 ):
            print( b[rij][kol], end=" " )
        print()

def opponent( p ):
    if p == SPELERX:
        return SPELER0
    return SPELERX

def neemRowKolom( speler, wat ):
    while True:
        num = getInteger( "Speler "+speler+", welke "+wat+
            " kies je? " )
        if num < 1 or num > 3:
            print( "Geef 1, 2, of 3" )
            continue
        return num

def winnaar( b ):
    for rij in range( 3 ):
        if b[rij][0] != LEEG and b[rij][0] == b[rij][1] \
            and b[rij][0] == b[rij][2]:
            return True
    for kol in range( 3 ):
        if b[0][kol] != LEEG and b[0][kol] == b[1][kol] \
            and b[0][kol] == b[2][kol]:
            return True
    if b[1][1] != LEEG:
```

```
        if b[1][1] == b[0][0] and b[1][1] == b[2][2]:
            return True
        if b[1][1] == b[0][2] and b[1][1] == b[2][0]:
            return True
    return False

bord = [[LEEG,LEEG,LEEG],[LEEG,LEEG,LEEG],[LEEG,LEEG,LEEG]]
speler = SPELERX

toon_bord( bord )
zet = 0
while True:
    rij = neemRowKolom( speler, "rij" )
    kol = neemRowKolom( speler, "kolom" )
    if bord[rij-1][kol-1] != LEEG:
        print( "Rij", rij, "en kolom", kol, "is niet leeg" )
        continue
    bord[rij-1][kol-1] = speler
    toon_bord( bord )
    if winnaar( bord ):
        print( "Speler", speler, "wint!" )
        break
    zet += 1
    if zet == MAXZET:
        print( "Het is gelijkspel." )
        break
    speler = opponent( speler )
```

Antwoord 12.7

answer1207.py

```
from pcinput import getString
from random import randint

LEEG = "."
SCHIP = "X"
SCHEPEN = 3
BREEDTE = 4
HOOGTE = 3

def toonBord( b ):
    print( "  ", end="" )
    for kol in range( BREEDTE ):
        print( chr( ord("A")+kol ), end=" " )
    print()
    for rij in range( HOOGTE ):
        print( rij+1, end=" ")
        for kol in range( BREEDTE ):
            print( b[rij][kol], end=" " )
```

```
        print()

def plaatsSchepen( b ):
    for i in range( SCHEPEN ):
        while True:
            x = randint( 0, BREEDTE-1 )
            y = randint( 0, HOOGTE-1 )
            if b[y][x] == SCHIP:
                continue
            if x > 0 and b[y][x-1] == SCHIP:
                continue
            if x < BREEDTE-1 and b[y][x+1] == SCHIP:
                continue
            if y > 0 and b[y-1][x] == SCHIP:
                continue
            if y < HOOGTE-1 and b[y+1][x] == SCHIP:
                continue
            break
        b[y][x] = SCHIP

def neemDoel():
    while True:
        cel = getString( "Welke cel kies je? " ).upper()
        if len( cel ) != 2:
            print( "Geef een cel in als XY,",
                "met X een letter en Y een cijfer" )
            continue
        if cel[0] not in "ABCD":
            print( "De letter moet tussen A en",
                chr( ord("A")+BREEDTE-1 ), "liggen" )
            continue
        if cel[1] not in "123":
            print( "Cijfer moet tussen 1 en", HOOGTE, "liggen" )
            continue
        return ord(cel[0])-ord("A"), ord(cel[1])-ord("1")

bord = []
for i in range( HOOGTE ):
    rij = BREEDTE * [LEEG]
    bord.append( rij )
plaatsSchepen( bord )
toonBord( bord )

raak = 0
zetten = 0
while raak < SCHEPEN:
    x, y = neemDoel()
    if bord[y][x] == SCHIP:
        print( "Raak!" )
        bord[y][x] = LEEG
```

```
        raak += 1
    else:
        print( "Mis!" )
    zetten += 1

print( "Je had", zetten, "zetten nodig om mij te verslaan." )
```

Antwoord 12.8

answer1208.py

```
# Recursieve functie die bepaalt of intlist (integer-list) een
# subset heeft die optelt tot totaal. Het retourneert de
# subset, of een lege list als er geen subset is die werkt.
def subset_telt_op_tot( intlist, totaal ):
    for num in intlist:
        if num == totaal:
            return [num]
        nlist = intlist[:]
        nlist.remove( num )
        retlist = subset_telt_op_tot( nlist, totaal-num )
        if len( retlist ) > 0:
            retlist.insert( 0, num )
            return( retlist )
    return []

numlist = [ 3, 8, -1, 4, -5, 6 ]
oplossing = subset_telt_op_tot( numlist, 0 )
if len( oplossing ) <= 0:
    print( "Geen subset telt op tot nul" )
else:
    for i in range( len( oplossing ) ):
        if oplossing[i] < 0 or i == 0:
            print( oplossing[i], end="" )
        else:
            print( "+{}".format( oplossing[i] ), end="" )
    print( "=0" )
```

De recursieve functie doorloopt alle getallen in `intlist`. Als het huidige getal gelijk is aan `totaal`, dan is een oplossing gevonden en wordt een list geretourneerd met daarin alleen het huidige nummer. Anders roept het zichzelf aan met een kopie van de list, waarbij het huidige nummer verwijderd is, met een totaal waarvan het huidige nummer is afgetrokken (bijvoorbeeld, als het huidige nummer 3 is en het totaal is 5, dan wordt 3 van de list verwijderd en wordt gecontroleerd of met de nieuwe list een totaal van 2 bereikt kan worden; immers, dan kan 5 bereikt worden door 3 op te tellen bij de deelverzameling die optelt tot 2). Als de recursieve aanroep een niet-lege list teruggeeft, dan is een oplossing gevonden, en wordt het huidige nummer aan de retourlist toegevoegd, en de retourlist geretourneerd. Anders wordt een lege list geretourneerd als alle nummers gecontroleerd zijn.

Merk op: De oplossing voor dit probleem test alle mogelijke deelverzamelingen totdat een

oplossing is gevonden of alle deelverzamelingen getest zijn. Je vraagt je misschien af of dit slimmer kan, gezien het feit dat het aantal deelverzamelingen exponentieel toeneemt met de grootte van de list. Het antwoord is dat er oplossingen kunnen zijn die een verbetering zijn op de huidige oplossing (bijvoorbeeld, als ik er rekening mee houd dat een getal meermalen op de list kan voorkomen dan kan ik sommige deelverzamelingen uitsluiten omdat ze equivalent zijn met andere), maar dat over het algemeen geldt dat voor een willekeurige list van getallen, er geen algoritme bekend is dat het probleem oplost zonder alle deelverzamelingen te testen als er geen oplossing is. Voor degenen die iets weten van complexiteitstheorie: het subset som probleem is "NP-hard."

Hoofdstuk 13

Antwoord 13.1

answer1301.py

```
tekst = """Kapper Knap, de knappe kapper, knipt en kapt heel
knap, maar de knecht van kapper Knap, de knappe kapper, knipt
en kapt nog knapper dan kapper Knap, de knappe kapper."""

def schoon( s ):
    ns = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            ns += c
        else:
            ns += " "
    return ns

wdict = {}
for woord in schoon( tekst ).split():
    wdict[woord] = wdict.get( woord, 0 ) + 1

keylist = list( wdict.keys() )
keylist.sort()
for key in keylist:
    print( "{}: {}".format( key, wdict[key] ) )
```

Antwoord 13.2

answer1302.py

```
movies = {  "Monty Python and the Holy Grail":
            [ 9, 10, 9.5, 8.5, 3, 7.5,8 ],
            "Monty Python's Life of Brian":
            [ 10, 10, 0, 9, 1, 8, 7.5, 8, 6, 9 ],
            "Monty Python's Meaning of Life":
            [ 7, 6, 5 ],
            "And Now For Something Completely Different":
```

```
            [ 6, 5, 6, 6 ] }

keylist = list( movies.keys() )
keylist.sort()
for key in keylist:
    print( "{}: {}".format( key, round(
        sum( movies[key] )/len( movies[key] ), 1 ) ) )
```

Antwoord 13.3 Veel verschillende antwoorden zijn mogelijk. Het eenvoudigste antwoord is waarschijnlijk om een dictionary te gebruiken waarvan de keys tuples zijn, die de schrijvers' achter- en voornamen bevatten. De waarde bij een key is een list, die de boeken van de corresponderende schrijver bevat. Een boek is een tuple bestaande uit een titel en een locatie. Om alle boeken van een schrijver uit te lijsten, kun je de list met boeken zoeken met de schrijver's naam. Om de locatie van een specifiek boek te vinden, zoek je op dezelfde manier de list met boeken van de schrijver, en zoekt dan in de list naar het juiste boek om de locatie vast te stellen. Je zou in plaats van een list van boeken ook een dictionary kunnen bouwen met de boek titel als key en de locatie als waarde, maar schrijvers produceren over het algemeen niet zoveel boeken dat dat nodig is. Het is natuurlijk zowiezo geen geweldig idee om namen en titels als keys te gebruiken, omdat ze lang zijn en het gemakkelijk is spelfouten te maken. Maar dit waren de enige identificerende elementen die ik noemde. Ik zal er echter niet over klagen als je gemakkelijke en minder dubbelzinnige keys introduceert in de data structuur (maar over het algemeen geldt dat je beter een database systeem kunt gebruiken om een bibliotheek in op te slaan).

Hoofdstuk 14

Antwoord 14.1

answer1401.py

```
alles = { "Socrates", "Plato", "Eratosthenes", "Zeus", "Hera",
    "Athene", "Acropolis", "Kat", "Hond" }
mensen = { "Socrates", "Plato", "Eratosthenes" }
sterfelijken = { "Socrates", "Plato", "Eratosthenes", "Kat",
    "Hond" }

print( mensen.issubset( sterfelijken ) ) # (a)
print( "Socrates" in mensen ) # (b)
print( "Socrates" in sterfelijken ) # (c)
print( len( sterfelijken.difference( mensen ) ) > 0 ) # (d)
print( len( alles.difference( sterfelijken ) ) > 0 ) # (e)
```

Antwoord 14.2

answer1402.py

```
set3 = set( [3*x for x in range( 1, int( 1001/3 ) )] )
set7 = set( [7*x for x in range( 1, int( 1001/7 ) )] )
set11 = set( [11*x for x in range( 1, int( 1001/11 ) )] )
```

```
seta = set3 & set7 & set11
setb = (set3 & set7) - set11
setc = set( range( 1, 1001 ) ) - set3 - set7 - set11
```

Je kunt de sets afdrukken om te zien dat ze correct zijn.

Hoofdstuk 15

Antwoord 15.1

answer1501.py

```
from os import listdir, getcwd

flist = listdir( "." )
for name in flist:
    print( getcwd() + "/" + name )
```

Hoofdstuk 16

Antwoord 16.1 Hopelijk herinnerde je je dat je iets soortgelijks hebt gedaan in hoofdstukken 10 en 13. De code hieronder is voor het grootste deel een kopie van de code die je eerder geschreven hebt. Het enige verschil met de code die je schreef in hoofdstuk 13 is dat de tekst niet aangeleverd is als een string, maar gelezen wordt uit een bestand.

answer1601.py

```
def schoon( s ):
    nieuws = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            nieuws += c
        else:
            nieuws += " "
    return nieuws

fp = open( "pc_woodchuck.txt" )
tekst = fp.read()
fp.close()

wdict = {}
for woord in schoon( tekst ).split():
    wdict[woord] = wdict.get( woord, 0 ) + 1

keylist = list( wdict.keys() )
keylist.sort()
```

```
for key in keylist:
    print( "{}: {}".format( key, wdict[key] ) )
```

Antwoord 16.2

answer1602.py

```
def schoon( s ):
    nieuws = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            nieuws += c
        else:
            nieuws += " "
    return nieuws

wdict = {}
fp = open( "pc_woodchuck.txt" )
while True:
    regel = fp.readline()
    if regel == "":
        break
    for woord in schoon( regel ).split():
        wdict[woord] = wdict.get( woord, 0 ) + 1
fp.close()

keylist = list( wdict.keys() )
keylist.sort()
for key in keylist:
    print( "{}: {}".format( key, wdict[key] ) )
```

Antwoord 16.3

answer1603.py

```
from os.path import join
from os import getcwd

def verwijderklinkers( regel ):
    nieuweregel = ""
    for c in regel:
        if c not in "aeiouAEIOU":
            nieuweregel += c
    return nieuweregel

inputnaam = join( getcwd(), "pc_woodchuck.txt" )
outputnaam = join( getcwd(), "pc_woodchuck.tmp" )

fpi = open( inputnaam )
```

```
fpo = open( outputnaam, "w" )

telread = 0
telwrite = 0

while True:
    regel = fpi.readline()
    if regel == "":
        break
    telread += len( regel )
    regel = verwijderklinkers( regel )
    fpo.write( regel )
    telwrite += len( regel )

fpo.close()
fpi.close()

print( "Gelezen:", telread )
print( "Geschreven:", telwrite )
```

Antwoord 16.4 De oplossing die ik hieronder geef maakt gebruik van sets. Er wordt een set gebouwd voor ieder bestand, dat alle woorden bevat uit het bestand die 3 of meer letters hebben. Je kunt de lengte van de gezochte woorden veranderen door `LEN` aan te passen. Om het programma flexibel te maken, heb ik het niet gelimiteerd tot slechts drie bestanden en sets, maar gebruik ik zoveel sets als er namen in de list van bestanden zijn. Het programma neemt de doorsnede van de sets om de oplossing te bepalen.

answer1604.py

```
from os.path import join
from os import getcwd

LEN = 3

def schoon( s ):
    nieuws = ""
    s = s.lower()
    for c in s:
        if c >= "a" and c <= "z":
            nieuws += c
        else:
            nieuws += " "
    return nieuws

lijst = ["pc_jabberwocky.txt","pc_rose.txt","pc_woodchuck.txt"]
setlist = []

for naam in lijst:
    bestandnaam = join( getcwd(), naam )
    woordset = set()
```

```
    setlist.append( woordset )
    fp = open( bestandnaam )
    while True:
        regel = fp.readline()
        if regel == "":
            break
        woordlist = schoon( regel ).split()
        for woord in woordlist:
            if len( woord ) >= LEN:
                woordset.add( woord )
    fp.close()

combinatie = setlist[0].copy()
i = 1
while i < len( setlist ):
    combinatie = combinatie & setlist[i]
    i += 1

for woord in combinatie:
    print( woord )
```

Antwoord 16.5 Wederom heb ik een oplossing gekozen die flexibel is wat betreft het aantal bestanden dat verwerkt wordt. Het programma is gemakkelijker te schrijven als je ervan uit gaat dat er slechts drie bestanden zijn, en niet meer. Als je een dergelijke oplossing gekozen hebt is dat prima (zolang het programma verder de juiste uitvoer geeft), aangezien het werken met een variabel aantal bestanden meer een oefening is voor het hoofdstuk over iteraties. Maar op dit moment zou je gewend moeten zijn aan het gebruik van iteraties, dus probeer ze toe te passen als dat leidt tot een superieure oplossing.

answer1605.py

```
from os.path import join
from os import getcwd

lijst = ["pc_jabberwocky.txt","pc_rose.txt","pc_woodchuck.txt"]
letterlist = [ len( lijst )*[0] for i in range( 26 ) ]
totaallist = len( lijst ) * [0]

# Verwerk de invoerlijst regel voor regel, maak kleine letters
# en tel de letters in letterlist, en houd totaaltellingen
# bij in totaallist.
aantal = 0
for naam in lijst:
    bestandnaam = join( getcwd(), naam )
    fp = open( bestandnaam )
    while True:
        regel = fp.readline()
        if regel == "":
            break
        regel = regel.lower()
```

```
        for c in regel:
            if c >= 'a' and c <= 'z':
                totaallist[aantal] += 1
                letterlist[ord(c)-ord("a")][aantal] += 1
    fp.close()
    aantal += 1

# Schrijf tellingen in CSV formaat.
uitvoerbestand = join( getcwd(), "pc_writetest.csv" )
fp = open( uitvoerbestand, "w" )
for i in range( len( letterlist ) ):
    s = "\"{}\"".format( chr( ord("a")+i ) )
    for j in range( len( lijst ) ):
        s += ",{:.5f}".format( letterlist[i][j]/totaallist[j] )
    fp.write( s+"\n" )
fp.close()

# Laat de inhoud van de CSV zien ter controle.
fp = open( uitvoerbestand )
print( fp.read() )
fp.close()
```

Hoofdstuk 17

Antwoord 17.1 De code kan een `ValueError` genereren als je iets ingeeft dat geen integer is, een `IndexError` als je een index ingeeft buiten het bereik `{-5,4}`, een `ZeroDivisionError` als je index 2 ingeeft, en een `TypeError` als je index 3 ingeeft. De code hieronder doet een simpele afhandeling door een foutmelding te geven, maar je kunt ook een loop bouwen om de code zodat de gebruiker om een nieuwe invoer wordt gevraagd.

answer1701.py

```
numlist = [ 100, 101, 0, "103", 104 ]

try:
    i1 = int( input( "Geef een index: " ) )
    print( "100 /", numlist[i1], "=", 100 / numlist[i1] )
except ValueError:
    print( "Je gaf geen geheel getal" )
except IndexError:
    print( "De index moet tussen -5 en 4 liggen" )
except ZeroDivisionError:
    print( "Het lijkt erop dat de list een 0 bevat" )
except TypeError:
    print( "Het lijkt erop er een niet-numeriek element is" )
except:
    print( "Iets onverwachts is gebeurd" )
    raise
```

Hoofdstuk 18

Antwoord 18.1 Ik gebruik een kopie van "pc_rose.txt" die ik "pc_rose_copy.txt" noem.

answer1801.py

```
NAAM = "pc_rose_copy.txt"

def toon_inhoud( filename ):
    fp = open( filename, "rb" )
    print( fp.read() )
    fp.close()

def encryptie( filename ):
    fp = open( filename, "r+b" )
    buffer = fp.read()
    fp.seek(0)
    for c in buffer:
        if c >= 128:
            fp.write( bytes( [c-128] ) )
        else:
            fp.write( bytes( [c+128] ) )
    fp.close()

toon_inhoud( NAAM )
encryptie( NAAM )
toon_inhoud( NAAM )
```

Antwoord 18.2

answer1802.py

```
letters = "etaoinshrdlcum "
ongecodeerd = "Hello, world!"

# Toon de string zonder codering, ter controle
print( ongecodeerd, len( ongecodeerd ) )

# Creeer een half-byte-list als basis voor codering
halfbytelist = []
for c in ongecodeerd:
    if c in letters:
        halfbytelist.append( letters.index( c )+1 )
    else:
        byte = ord( c )
        halfbytelist.extend( [0, int( byte/16 ), byte%16 ] )
if len( halfbytelist )%2 != 0:
    halfbytelist.append( 0 )

# Maak van de half-byte-list een bytelist.
bytelist = []
```

```
for i in range( 0, len( halfbytelist ), 2 ):
    bytelist.append( 16*halfbytelist[i] + halfbytelist[i+1] )

# Maak van de bytelist een bytestring en toon die ter controle.
gecodeerd = bytes( bytelist )
print( gecodeerd, len( gecodeerd ) )
```

Dit programma is 25 regels, waarvan 4 commentaar, 4 leeg, en 3 slechts voor testdoeleinden zijn. Dit kan dus met 14 regels code. Dat was niet al te erg, toch?

Antwoord 18.3

answer1803.py

```
letters = "etaoinshrdlcum "
gecodeerd = b'\x04\x81\xbb@,\xf0wI\xba\x02\x10'

# Toon de gecodeerde byte string ter controle.
print( gecodeerd, len( gecodeerd ) )

# Creeer een half-byte-list op basis van de byte string.
halfbytelist = []
for c in gecodeerd:
    halfbytelist.extend( [ int( c/16 ), c%16 ] )
if halfbytelist[-1] == 0:
    del halfbytelist[-1]

# Maak van de half-byte-string een string.
gedecodeerd = ""
while len( halfbytelist ) > 0:
    num = halfbytelist.pop(0)
    if num > 0:
        gedecodeerd += letters[num-1]
        continue
    num = 16*halfbytelist.pop(0) + halfbytelist.pop(0)
    gedecodeerd += chr( num )

# Toon de string, ter controle.
print( gedecodeerd, len( gedecodeerd ) )
```

Antwoord 18.4 Dit programa lijkt lang, maar is eenvoudig. `comprimeer()` en `decomprimeer()` werden eerder gebouwd. Ik heb alleen de invoer en uitvoer in byte strings gewijzigd. De rest bestaat vooral uit het afhandelen van mogelijke fouten die kunnen optreden bij bestandsmanipulatie. De basis-functionaliteit van een programma kun je vaak in een paar regels schrijven, terwijl foutafhandeling drie keer zo lang is.

answer1804.py

```
from pcinput import getString, getLetter
from os.path import exists, getsize
```

```
LETTERS = b"etaoinshrdlcum "

# Comprimeer byte string ongecodeerd, retourneer gecomprimeerd
def comprimeer( ongecodeerd ):
    halfbytelist = []
    for c in ongecodeerd:
        if c in LETTERS:
            halfbytelist.append( LETTERS.index( c )+1 )
        else:
            halfbytelist.extend( [0, int( c/16 ), c%16 ] )
    if len( halfbytelist )%2 != 0:
        halfbytelist.append( 0 )
    bytelist = []
    for i in range( 0, len( halfbytelist ), 2 ):
        bytelist.append( 16*halfbytelist[i] + halfbytelist[i+1] )
    return bytes( bytelist )

# Decomprimeer byte string gecodeerd, retourneer gedecomprimeerd
def decomprimeer( gecodeerd ):
    halfbytelist = []
    for c in gecodeerd:
        halfbytelist.extend( [ int( c/16 ), c%16 ] )
    if halfbytelist[-1] == 0:
        del halfbytelist[-1]
    bytelist = []
    while len( halfbytelist ) > 0:
        num = halfbytelist.pop(0)
        if num > 0:
            bytelist.append( LETTERS[num-1] )
            continue
        num = 16*halfbytelist.pop(0) + halfbytelist.pop(0)
        bytelist.append( num )
    return bytes( bytelist )

# Vraag om input bestand en lees de inhoud
while True:
    filein = getString( "Geef input bestand: " )
    if not exists( filein ):
        print( filein, "bestaat niet" )
        continue
    try:
        fp = open( filein, "rb" )
        buffer = fp.read()
        fp.close()
    except IOError as ex:
        print( filein, "kan niet verwerkt worden, kies andere" )
        print( "Error [{}]: {}".format( ex.args[0], ex.args[1] ))
        continue
    break
```

```
# Vraag om output bestand en creeer het
while True:
    fileout = getString( "Geef output bestand: " )
    if exists( fileout ):
        print( fileout, "bestaat al" )
        continue
    try:
        fp = open( fileout, "wb" )
    except IOError as ex:
        print( fileout, "kan niet aangemaakt worden,",
            "kies een andere naam" )
        print( "Error [{}]: {}".format( ex.args[0], ex.args[1] ))
        continue
    break

# Vraag of de gebruiker wil comprimeren of decomprimeren.
while True:
    dc = getLetter( "Wil je (C)omprimeren of (D)ecomprimeren? " )
    if dc != 'C' and dc != 'D':
        print( "Kies C of D" )
        continue
    break

# Comprimeer of decomprimeer de buffer.
if dc == 'C':
    buffer = comprimeer( buffer )
else:
    buffer = decomprimeer( buffer )

# Sla de verwerkte buffer op in het output bestand.
try:
    fp.write( buffer )
    fp.close()
except IOError as ex:
    print( "Het schrijven lukte niet" )
    print( "Error [{}]: {}".format( ex.args[0], ex.args[1] ))

# Rapporteer de groottes van input en output.
print( getsize( filein ), "bytes gelezen" )
print( getsize( fileout ), "bytes geschreven" )
```

Hoofdstuk 19

Antwoord 19.1

answer1901.py

```
s = "Hello, world!"
```

```
mask = (1<<5) | (1<<3) | (1<<1)    # 00101010

codering = ""
for c in s:
    codering += chr(ord(c)^mask)
print( codering )

decodering = ""
for c in codering:
    decodering += chr(ord(c)^mask)
print( decodering )
```

Antwoord 19.2

answer1902.py

```
def setBit( opslag, index, value ):
    masker = 1<<index
    if value:
        opslag |= masker
    else:
        opslag &= ~masker
    return opslag

# getBit() retourneert 0 als de bit behorende bij de index op 1
# staat, en anders retourneert het iets anders. Omdat alleen 0
# als False wordt beschouwd, kun je deze functie gebruiken om de
# waarde van een bit te testen.
def getBit( opslag, index ):
    masker = 1<<index
    return opslag & masker

def toonBits( opslag ):
    for i in range( 8 ):
        index = 7 - i
        if getBit( opslag, index ):
            print( "1", end="" )
        else:
            print( "0", end="" )
    print()

opslag = 0
opslag = setBit( opslag, 0, True )
opslag = setBit( opslag, 1, True )
opslag = setBit( opslag, 2, False )
opslag = setBit( opslag, 3, True )
opslag = setBit( opslag, 4, False )
opslag = setBit( opslag, 5, True )
toonBits( opslag )
```

```
opslag = setBit( opslag, 1, False )
toonBits( opslag )
```

Hoofdstuk 20

Antwoord 20.1

answer2001.py

```
from copy import copy

class Punt:
    def __init__( self, x=0.0, y=0.0 ):
        self.x = x
        self.y = y
    def __repr__( self ):
        return "({}, {})".format( self.x, self.y )

class Rechthoek:
    def __init__( self, punt, breedte, hoogte ):
        self.punt = copy( punt )
        self.breedte = abs( breedte )
        self.hoogte = abs( hoogte )
        if self.breedte == 0:
            self.breedte = 1
        if self.hoogte == 0:
            self.hoogte = 1
    def __repr__( self ):
        return "[{},w={},h={}]".format( self.punt,
            self.breedte, self.hoogte )
    def oppervlakte( self ):
        return self.breedte * self.hoogte
    def omtrek( self ):
        return 2*(self.breedte + self.hoogte)
    def rechtsonder( self ):
        return Punt( self.punt.x + self.breedte,
            self.punt.y + self.hoogte )
    def overlap( self, r ):
        r1, r2 = self, r
        if self.punt.x > r.punt.x or \
            (self.punt.x == r.punt.x and \
            self.punt.y > r.punt.y):
            r1, r2 = r, self
        if r1.rechtsonder().x <= r2.punt.x or \
            r1.rechtsonder().y <= r2.punt.y:
            return None
        return Rechthoek( r2.punt,
            min( r1.rechtsonder().x - r2.punt.x, r2.breedte ),
            min( r1.rechtsonder().y - r2.punt.y, r2.hoogte ) )
```

```
r1 = Rechthoek( Punt( 1, 1 ), 8, 5 )
r2 = Rechthoek( Punt( 2, 3 ), 9, 2 )

print( r1.oppervlakte() )
print( r1.omtrek() )
print( r1.rechtsonder() )
r = r1.overlap( r2 )
if r:
    print( r )
else:
    print( "De rechthoeken overlappen niet" )
```

Antwoord 20.2 Vanwege de lijst die je moet tonen, kun je het beste de methode `inschrijven()` in de student plaatsen. Voor de geboortedatum gebruik ik de `datetime` module; omdat je de leeftijd van de student moet berekenen, heb je ook de dag van vandaag nodig, waarvoor je een functie in de `datetime` module vindt.

answer2002.py

```
from datetime import date
from random import random

class Cursus:
    def __init__( self, naam, nummer ):
        self.naam = naam
        self.nummer = nummer
    def __repr__( self ):
        return "{}: {}".format( self.nummer, self.naam )

class Student:
    def __init__( self, achternaam, voornaam, geb_datum, anr):
        self.achternaam = achternaam
        self.voornaam = voornaam
        self.geboortedatum = geb_datum
        self.anr = anr
        self.cursussen = []
    def __str__( self ):
        return self.voornaam+" "+self.achternaam
    def age( self ):
        vandaag = date.today()
        jaren = vandaag.year - self.geboortedatum.year
        if vandaag.month < self.geboortedatum.month or \
            (vandaag.month == self.geboortedatum.month \
            and vandaag.day < self.geboortedatum.day):
            jaren -= 1
        return jaren
    def inschrijven( self, cursus ):
        if cursus not in self.cursussen:
            self.cursussen.append( cursus )
```

```
studenten = [
    Student( "Alikruik", "Adrie", date( 1989, 10, 3 ), 453211 ),
    Student( "Bonzo", "Beatrijs", date( 1991, 12, 29 ), 476239 ),
    Student( "Continuum", "Carola", date( 1992, 3, 7 ), 784322 ),
    Student( "Domoor", "Dirk", date( 1993, 7, 11 ), 995544 ) ]
cursussen =[
    Cursus( "Vinologie", 787656 ),
    Cursus( "Lepelbuigen voor gevorderden", 651121 ),
    Cursus( "Onderzoeksvaardigheden: Babbelen", 433231 )]

for student in studenten:
    for cursus in cursussen:
        if random() > 0.3:
            student.inschrijven( cursus )

for student in studenten:
    print( "{}: {} {} ({})".format( student.anr,
        student.voornaam, student.achternaam, student.age() ) )
    if len( student.cursussen ) == 0:
        print( "\tNo cursussen" )
    for cursus in student.cursussen:
        print( "\t{}".format( cursus ) )
```

Hoofdstuk 21

Antwoord 21.1

answer2101.py

```
KLEUR = ["Harten","Schoppen","Klaveren","Ruiten"]
RANG = ["2","3","4","5","6","7","8","9","10",
    "Boer","Vrouw","Heer","Aas"]

class Kaart:
    def __init__( self, kleur, rang ):
        self.kleur = kleur # index in KLEUR list
        self.rang = rang # index in RANG list
    def __repr__( self ):
        return "({},{})".format( self.kleur, self.rang )
    def __str__( self ):
        return "{} {}".format(KLEUR[self.kleur],RANG[self.rang])
    def __eq__( self, c ):
        if isinstance( c, Kaart ):
            return self.rang == c.rang
        return NotImplemented
    def __gt__( self, c ):
        if isinstance( c, Kaart ):
            return self.rang > c.rang
```

```
        return NotImplemented
    def __ge__( self, c ):
        if isinstance( c, Kaart ):
            return self.rang >= c.rang
        return NotImplemented

k5 = Kaart( 2, 3 )
r5 = Kaart( 3, 3 )
sh = Kaart( 1, 11 )
print( "{}, {}, {}".format( k5, r5, sh ) )
print( k5 == r5 )
print( k5 == sh )
print( k5 > sh )
print( k5 >= sh )
print( k5 < sh )
print( k5 <= sh )
```

Merk op dat je de `__ne__()`, `__lt__()`, en `__le__()` niet hoeft te implementeren, aangezien die automatisch worden gewijzigd in aanroepen van andere methodes die je implementeert.

Antwoord 21.2 Om de leesbaarheid te vergroten heb ik de methodes die niet nodig zijn uit `Kaart` verwijderd voor dit programma.

answer2102.py

```
KLEUR = ["Harten","Schoppen","Klaveren","Ruiten"]
RANG = ["2","3","4","5","6","7","8","9","10",
    "Boer","Vrouw","Heer","Aas"]

class Kaart:
    def __init__( self, kleur, rang ):
        self.kleur = kleur
        self.rang = rang
    def __str__( self ):
        return "{} {}".format(KLEUR[self.kleur],RANG[self.rang])

class Trekstapel:
    def __init__( self, stapel=[] ):
        self.stapel = stapel
    def __len__( self ):
        return len( self.stapel )
    def __getitem__( self, n ):
        return self.stapel[n]
    def voegtoe( self, c ):
        self.stapel.append( c )
    def trek( self ):
        if len( self ) <= 0:
            return None
        return self.stapel.pop(0)
    def __repr__( self ):
```

```
        sep = ""
        s = ""
        for c in self.stapel:
            s += sep + str( c )
            sep = ", "
        return s

ts1 = Trekstapel( [Kaart(0,1),Kaart(0,5),Kaart(2,4),Kaart(1,12)])
print( ts1 )
print( ts1[1] )
ts1.voegtoe( Kaart(3,12) )
print( ts1 )
print( ts1.trek() )
print( ts1 )
```

Antwoord 21.3

answer2103.py

```
KLEUR = ["Harten","Schoppen","Klaveren","Ruiten"]
RANG = ["2","3","4","5","6","7","8","9","10",
    "Boer","Vrouw","Heer","Aas"]

class Kaart:
    def __init__( self, kleur, rang ):
        self.kleur = kleur
        self.rang = rang
    def __repr__( self ):
        return "({},{})".format( self.kleur, self.rang )
    def __str__( self ):
        return "{} {}".format(KLEUR[self.kleur],RANG[self.rang])
    def __eq__( self, c ):
        if isinstance( c, Kaart ):
            return self.rang == c.rang
        return NotImplemented
    def __gt__( self, c ):
        if isinstance( c, Kaart ):
            return self.rang > c.rang
        return NotImplemented
    def __ge__( self, c ):
        if isinstance( c, Kaart ):
            return self.rang >= c.rang
        return NotImplemented

class Trekstapel:
    def __init__( self, stapel=[] ):
        self.stapel = stapel
    def __len__( self ):
        return len( self.stapel )
    def __getitem__( self, n ):
```

```
        return self.stapel[n]
    def voegtoe( self, c ):
        self.stapel.append( c )
    def trek( self ):
        if len( self ) <= 0:
            return None
        return self.stapel.pop(0)
    def __repr__( self ):
        sep = ""
        s = ""
        for c in self.stapel:
            s += sep + str( c )
            sep = ", "
        return s

ts1 = Trekstapel( [Kaart(3,0), Kaart(0,11), Kaart(2,5)] )
ts2 = Trekstapel( [Kaart(3,2), Kaart(3,1), Kaart(1,6)] )

i = 1
while len( ts1 ) > 0 and len( ts2 ) > 0:
    print( "Ronde", i )
    print( "Stapel1:", ts1 )
    print( "Stapel2:", ts2 )
    c1 = ts1.trek()
    c2 = ts2.trek()
    if c1 > c2:
        ts1.voegtoe( c1 )
        ts1.voegtoe( c2 )
    else:
        ts2.voegtoe( c2 )
        ts2.voegtoe( c1 )
    i += 1

print( "Het spel is uit" )
if len( ts1 ) > 0:
    print( "Stapel1:", ts1 )
    print( "De eerste stapel wint!" )
else:
    print( "Stapel2:", ts2 )
    print( "De tweede stapel wint!" )
```

Antwoord 21.4 Vergeet niet dat je in de `__add__()` en `__sub__()` methodes (diepe) kopieën moet creëren van de fruitmand, die je dan aanpast en retourneert. Als je in plaats daarvan de `fruitmand` zelf wijzigt in plaats van een diepe kopie, dan zou een statement als `nieuwemand = fruitmand + "appel"` leiden tot `nieuwemand` als alias voor `fruitmand`. Het is niet waarschijnlijk dat een programmeur die de class gebruikt dat wil. Degene die het hoofdprogramma schrijft moet zelf beslissen of hij of zij de fruitmand wil wijzigen, of alleen een gewijzigde versie van de fruitmand wil zien.

In de `__iadd__()` en `__isub__()` methodes daarentegen wordt je geacht om de fruitmand

te wijzigen, en `self` te retourneren. Want een statement als `fruitmand += "appel"` is duidelijk bedoeld om `fruitmand` te veranderen.

De rest van de class is erg eenvoudig te begrijpen, zolang je er maar voor zorgt dat je fruit uit de mand verwijderd met **`del`** wanneer er nul of minder stuks in de mand zitten.

answer2104.py

```
from copy import deepcopy

class Fruitmand:

    def __init__( self, stukken={} ):
        self.stukken = stukken

    def __repr__( self ):
        s = ""
        sep = "["
        for fruit in self.stukken:
            s += sep+fruit+":"+str( self.stukken[fruit] )
            sep = ", "
        s += "]"
        return s

    def __contains__( self, fruit ):
        return fruit in self.stukken

    def __add__( self, fruit ):
        fmcopy = deepcopy( self )
        fmcopy.stukken[fruit] = fmcopy.stukken.get( fruit, 0 )+1
        return fmcopy

    def __iadd__( self, fruit ):
        self.stukken[fruit] = self.stukken.get( fruit, 0 )+1
        return self

    def __sub__( self, fruit ):
        if fruit not in self.stukken:
            return self
        fmcopy = deepcopy( self )
        fmcopy.stukken[fruit] = fmcopy.stukken.get( fruit, 0 )-1
        if fmcopy.stukken[fruit] <= 0:
            del fmcopy.stukken[fruit]
        return fmcopy

    def __isub__( self, fruit ):
        self.stukken[fruit] = self.stukken.get( fruit, 0 )-1
        if self.stukken[fruit] <= 0:
            del self.stukken[fruit]
        return self
```

```
    def __len__( self ):
        return len( self.stukken )

    def __getitem__( self, fruit ):
        return self.stukken.get( fruit, 0 )

    def __setitem__( self, fruit, n ):
        if n <= 0:
            if fruit in self.stukken:
                del self.stukken[fruit]
        else:
            self.stukken[fruit] = n

fm = Fruitmand()
fm += "appel"
fm += "appel"
fm += "banaan"
fm = fm + "kers"
fm["mango"] = 20
print( len( fm ) )
print( fm )
print( "banaan" in fm )
print( "doerian" in fm )
fm -= "appel"
fm -= "banaan"
fm = fm - "kers"
fm -= "doerian"
print( fm )
print( "banaan" in fm )
fm["mango"] = 0
print( fm )
```

Hoofdstuk 22

Antwoord 22.1

answer2201.py

```
class Rechthoek:
    def __init__( self, x, y, b, h ):
        self.x, self.y, self.b, self.h = x, y, b, h
    def __repr__( self ):
        return "[({},{}),b={},h={}]".format( self.x, self.y,
          self.b, self.h )
    def oppervlakte( self ):
        return self.b * self.h
    def omtrek( self ):
        return 2*(self.b + self.h)
```

```
class Vierkant( Rechthoek ):
    def __init__( self, x, y, b ):
        super().__init__( x, y, b, b )

s = Vierkant( 1, 1, 4 )
print( s, s.oppervlakte(), s.omtrek() )
```

Antwoord 22.2

answer2202.py

```
from math import pi

class Vorm:
    def oppervlakte( self ):
        return NotImplemented
    def omtrek( self ):
        return NotImplemented

class Cirkel( Vorm ):
    def __init__( self, x, y, s ):
        self.x, self.y, self.s = x, y, s
    def __repr__( self ):
        return "[({},{}),r={}]".format( self.x, self.y, self.s )
    def oppervlakte( self ):
        return pi * self.s * self.s
    def omtrek( self ):
        return 2 * pi * self.s

class Rechthoek( Vorm ):
    def __init__( self, x, y, b, h ):
        self.x, self.y, self.b, self.h = x, y, b, h
    def __repr__( self ):
        return "[({},{}),w={},h={}]".format( self.x, self.y,
            self.b, self.h )
    def oppervlakte( self ):
        return self.b * self.h
    def omtrek( self ):
        return 2*(self.b + self.h)

class Vierkant( Rechthoek ):
    def __init__( self, x, y, b ):
        super().__init__( x, y, b, b )

v = Vierkant( 1, 1, 4 )
print( v, v.oppervlakte(), v.omtrek() )
c = Cirkel( 1, 1, 4 )
print( c, c.oppervlakte(), c.omtrek() )
```

Antwoord 22.3 `GeheugenStrategie` is gedefinieerd om `OogOmOog`, `OogOmTweeOgen`, en `Meerderheid` van af te leiden. `GeheugenStrategie` houdt alle rondes in het spel bij. Dat is overdreven, aangezien `OogOmOog` alleen de laatste zet hoeft te kennen, `OogOmTweeOgen` alleen de laatste twee zetten, en `Meerderheid` kan volstaan met een totaaltelling. Maar voor uitgebreidere strategieën kan de `GeheugenStrategie` een goed startpunt zijn.

Een interessant resultaat van de competities tussen de strategieën is dat `AltijdD` per definitie nooit een lagere score heeft dan zijn tegenstander, maar dat als je iedere strategie tegen iedere andere strategie laat spelen en dan de scores optelt, de enige strategie die slechter scoort dan `AltijdD` de `Random` strategie is. Als je wilt snappen hoe dat komt, volg dan een cursus speltheorie.

answer2203.py

```
from random import random

COOPERATIE = 'C'
DEFECTIE = 'D'
RONDES = 100

class Strategie:
    def __init__( self, naam="" ):
        self.naam = naam
        self.score = 0
    def keuze( self ):
        # Moet COOPERATIE of DEFECTIE retourneren
        return NotImplemented
    def laatstezet( self, mijnzet, tegenstanderzet ):
        # Krijgt de laatste zet die gedaan is, na keuze()
        pass
    def plusscore( self, n ):
        self.score += n

class AltijdD( Strategie ):
    def keuze( self ):
        return DEFECTIE

class Random( Strategie ):
    def keuze( self ):
        if random() >= 0.5:
            return COOPERATIE
        return DEFECTIE

class GeheugenStrategie( Strategie ):
    def __init__( self, naam="" ):
        super().__init__( naam )
        self.historie = []
    def laatstezet( self, mijnzet, tegenstanderzet ):
        self.historie.append( (mijnzet, tegenstanderzet) )

class OogOmOog( GeheugenStrategie ):
```

```
    def keuze( self ):
        if len( self.historie ) < 1:
            return COOPERATIE
        return self.historie[-1][1]

class OogOmTweeOgen( GeheugenStrategie ):
    def keuze( self ):
        if len( self.historie ) < 2:
            return COOPERATIE
        if self.historie[-1][1] == DEFECTIE and \
            self.historie[-2][1] == DEFECTIE:
            return DEFECTIE
        return COOPERATIE

class Meerderheid( GeheugenStrategie ):
    def keuze( self ):
        telD = 0
        for i in range( len( self.historie ) ):
            if self.historie[i][1] == DEFECTIE:
                telD += 1
        if telD > len( self.historie ) / 2:
            return DEFECTIE
        return COOPERATIE

strategie1 = AltijdD( "Altijd Defectie" )
strategie2 = Meerderheid( "Meerderheid" )

for i in range( RONDES ):
    c1 = strategie1.keuze()
    c2 = strategie2.keuze()
    if c1 == c2:
        strategie1.plusscore( 3 if c1 == COOPERATIE else 1 )
        strategie2.plusscore( 3 if c2 == COOPERATIE else 1 )
    else:
        strategie1.plusscore( 0 if c1 == COOPERATIE else 6 )
        strategie2.plusscore( 0 if c2 == COOPERATIE else 6 )
    strategie1.laatstezet( c1, c2 )
    strategie2.laatstezet( c2, c1 )

print( "Score van", strategie1.naam, "is", strategie1.score )
print( "Score van", strategie2.naam, "is", strategie2.score )
```

Hoofdstuk 23

Antwoord 23.1

answer2301.py

```
from pcinput import getInteger
```

```
class NietDeelbaarDoor:
    def __init__( self ):
        self.seq = list( range( 1, 101 ) )
    def __iter__( self ):
        return self
    def __next__( self ):
        if len( self.seq ) > 0:
            return self.seq.pop(0)
        raise StopIteration()
    def verwerk( self, num ):
        i = len( self.seq )-1
        while i >= 0:
            if self.seq[i]%num == 0:
                del self.seq[i]
            i -= 1
    def __len__( self ):
        return len( self.seq )

ndd = NietDeelbaarDoor()
while True:
    num = getInteger( "Geef een getal: " )
    if num < 0:
        print( "Negatieve getallen worden overgeslagen" )
        continue
    if num == 0:
        break
    ndd.verwerk( num )

if len( ndd ) <= 0:
    print( "Er zijn geen getallen meer" )
else:
    for num in ndd:
        print( num, end=" " )
```

Antwoord 23.2

answer2302.py

```
def faculteit():
    totaal = 1
    for i in range( 1, 11 ):
        totaal *= i
        yield totaal

fseq = faculteit()
for n in fseq:
    print( n, end=" " )
```

Antwoord 23.3

answer2303.py

```
from itertools import permutations

woord = input( "Geef een woord: " )
seq = permutations( woord )
for item in seq:
    print( "".join( item ) )
```

Antwoord 23.4 De enige wijziging tussen deze code en het antwoord op de vorige opgave is dat ik van de iterabele een set heb gemaakt.

answer2304.py

```
from itertools import permutations

woord = input( "Geef een woord: " )
seq = permutations( woord )
for item in set( seq ):
    print( "".join( item ) )
```

Antwoord 23.5

answer2305.py

```
from itertools import combinations

numlist = [ 3, 8, -1, 4, -5, 6 ]
oplossing = []

for i in range( 1, len( numlist )+1 ):
    seq = combinations( numlist, i )
    for item in seq:
        if sum( item ) == 0:
            oplossing = item
            break
    if len( oplossing ) > 0:
        break

if len( oplossing ) <= 0:
    print( "Er is geen subset die optelt tot nul" )
else:
    for i in range( len( oplossing ) ):
        if oplossing[i] < 0 or i == 0:
            print( oplossing[i], end="" )
        else:
            print( "+{}".format( oplossing[i] ), end="" )
    print( "=0" )
```

Merk op dat hoewel deze code alle subsets bouwt, wat een aantal is dat exponentieel toeneemt met de lengte van `numlist`, er nooit meer dan één subset in het geheugen staat omdat iteratoren worden gebruikt. Dus deze oplossing werkt uitstekend voor zelfs lange reeksen getallen (al wordt hij op een gegeven moment wel erg traag, maar dat kan niet verholpen worden omdat dit een NP-hard probleem is).

Antwoord 23.6

answer2306.py

```
from itertools import combinations

testdict = {"a":1, "b":2, "c":3, "d":4 }
resultaat = [ {} ]

keylist = list( testdict.keys() )
for lengte in range( 1, len( testdict)+1 ):
    for item in combinations( keylist, lengte ):
        subdict = {}
        for key in item:
            subdict[key] = testdict[key]
        resultaat.append( subdict )

print( resultaat )
```

Antwoord 23.7 Als je de rijen en de kolommen nummert van 0 tot 7, dan komt ieder rij nummer en ieder kolom nummer precies één keer voor in de oplossing. Neem een list met de acht kolom nummers, en beschouw de index in deze list als de erbij horende rij nummers. Alle potentiële oplossingen worden gerepresenteerd door permutaties van deze list. Je hoeft alleen maar te zoeken naar een permutatie waarbij er geen twee koninginnen op dezelfde diagonaal staan, dus waarbij het absolute verschil tussen hun rij nummers gelijk is aan het absolute verschil tussen hun kolom nummers.

answer2307.py

```
from itertools import permutations

SIZE = 8 # Bord grootte

def toon_bord( kols ):
    for i in kols :
        for j in range( len( kols ) ):
            if i == j:
                print( 'K', end=" " )
            else:
                print( '-', end=" " )
        print()

def is_oplossing( kols ):
    for rij in range( len( kols ) ):
        kol = kols[rij]
```

```
        for i in range( rij+1, len( kols ) ):
            if i - rij == abs( kols[i] - kol ):
                return False
    return True

kols = list( range( SIZE ) )

for p in permutations( kols ):
    if is_oplossing( p ):
        toon_bord( p )
        break
else:
    print( "Geen oplossing gevonden" ) # Should not happen.
```

Hoofdstuk 24

Antwoord 24.1

answer2401.py

```
import sys

totaal = 0
for i in sys.argv[1:]:
    try:
        totaal += float( sys.argv[i] )
    except TypeError:
        print( sys.argv[i], "is geen getal." )
        sys.exit(1)

print( "The arguments add up to", totaal )
```

Hoofdstuk 25

Antwoord 25.1

answer2501.py

```
import re

zin = "De prijs voor een 2-kamer appartement op  Manhattan \
start bij 1 miljoen dollars, en kan oplopen tot het tienvoudige \
op 42nd Street."

pwoord = re.compile( r"[A-Za-z]+" )
woordlist = pwoord.findall( zin )
for woord in woordlist:
    print( woord )
```

Antwoord 25.2

answer2502.py

```
import re

zin = "De diender arresteerde de doerak op de Dam."

pde = re.compile( r"\bde\b", re.I )
delist = pde.findall( zin )
print( len( delist ) )
```

Antwoord 25.3

answer2503.py

```
import re

zin = "Michael Jordan, Bill Gates, and de Dalai Lama besloten \
om samen een vliegtochtje te ondernemen, toen ze een hippie \
zagen op de landingsbaan."

pnaam = re.compile( r"\b([A-Z][a-z]*\s+[A-Z][a-z]*)\b" )
naamlist = pnaam.findall( zin )
for naam in naamlist:
    print( naam )
```

Antwoord 25.4

answer2504.py

```
import re

zin = "William Randolph Hearst trachtte alle exemplaren van \
Orson Welles' meesterwerk 'Citizen Kane', te vernietigen, omdat \
hij het bezwaarlijk vond dat de protagonist een nauwelijks-\
verhulde karikatuur van hemzelf was. Ik vraag me af of William \
Henry Gates De Derde hetzelfde zou doen."

pnaam = re.compile( r"\b([A-Z][a-z]*(\s+[A-Z][a-z]*)+)\b" )
naamlist = pnaam.finditer( zin )
for naam in naamlist:
    print( naam.group(1) )
```

Antwoord 25.5

answer2505.py

```
import re

zin = "Klant: \"Ik wens een klacht te deponeren! \
```

```
Hallo mevrouw!\"\n\
Winkelier: \"Hoe bedoelt u, mevrouw?\"\n\
Klant: \"Het spijt me, ik ben verkouden.\"\n"

pgesproken = re.compile( r"\"[^\"]*\"" )
gesprokenlist = pgesproken.findall( zin )
for tekst in gesprokenlist:
    print( tekst )
```

Antwoord 25.6

answer2506.py

```
import re

tekst = "<html><head><title>Lijst van personen met ids</title>\
</head><body>\
<p><id>123123123</id><naam>Groucho Marx</naam>\
<p><id>123123124</id><naam>Harpo Marx</naam>\
<p><id>123123125</id><naam>Chico Marx</naam>\
<randomcrap>Etaoin<id>Shrdlu</id>qwerty</naam></randomcrap>\
<nocrap><p><id>123123126</id><naam>Zeppo Marx</naam></nocrap>\
<address>Chicago</address>\
<morerandomcrap><id>999999999</id>geennaamtezien!\
</morerandomcrap>\
<p><id>123123127</id><naam>Gummo Marx</naam>\
<note>Zoek hem op via <a href=\"http://www.google.com\">\
Google.</a></note>\
</body></html>"

pidnaam = re.compile( r"<id>([^<]+)</id><naam>([^<]+)</naam>" )
mlist = pidnaam.finditer( tekst )
for m in mlist:
    print( m.group(1), m.group(2) )
```

Hoofdstuk 26

Antwoord 26.1

answer2601.py

```
from csv import reader, writer

fp = open( "pc_inventory.csv", newline='' )
fpo = open( "pc_writetest.csv", "w", newline='' )
csvreader = reader( fp )
csvwriter = writer( fpo, delimiter=' ', quotechar="'" )
for line in csvreader:
    csvwriter.writerow( line )
```

```
fp.close()
fpo.close()

fp = open( "pc_writetest.csv" )
print( fp.read() )
fp.close()
```

Als je dit correct het gedaan, zie je dubbele aanhalingstekens rondom “Blue Stilton,” die daar staan omdat er een spatie in voorkomt.

Antwoord 26.2

answer2602.py

```
from csv import reader
from json import dump

data = []

fp = open( "pc_inventory.csv", newregel='' )
csvreader = reader( fp )
for regel in csvreader:
    data.append( regel )
fp.close()

fp = open( "pc_writetest.json", "w" )
dump( data, fp )
fp.close()

fp = open( "pc_writetest.json" )
print( fp.read() )
fp.close()
```

Hoofdstuk 27

Antwoord 27.1

answer2701.py

```
from collections import Counter

z = "Je moeder was een hamster en je vader rook naar vlierbessen"
zin2 = ""
for c in z.lower():
    if c >= 'a' and c <= 'z':
        zin2 += c

clist = Counter( zin2 ).most_common( 5 )
for c in clist:
```

```
    print( "{}: {}".format( c[0], c[1] ) )
```

Antwoord 27.2

answer2702.py

```
from collections import Counter
from pcinput import getInteger
from statistics import mean, median
from sys import exit

numlist = []
while True:
    num = getInteger( "Geef een getal: " )
    if num == 0:
        break
    numlist.append( num )

if len( numlist ) <= 0:
    print( "Geen getallen ingegeven" )
    exit()

print( "Gemiddelde:", mean( numlist ) )
print( "Mediaan:", median( numlist ) )

clist = Counter( numlist ).most_common()
if clist[0][1] <= 1:
    print( "Er is geen modus" )
else:
    mlist = [str( x[0] ) for x in clist if x[1] == clist[0][1] ]
    s = ", ".join( mlist )
    print( "Modus:", s )
```

Voor de modus heb ik een redelijk slimme list comprehension geschreven, maar je kunt dit uiteraard ook in meerdere regels code schrijven.

Appendix G

Engelse Terminologie

In deze appendix geef ik een overzicht van Engeltalige termen die regelmatig voorkomen in dit boek. Dit zijn termen die in het boek voorkomen omdat er ofwel geen fatsoenlijke vertaling voor is, ofwel het onder programmeurs gebruikelijk is de Engelstalige term te gebruiken.

- *append* (spreek uit: "appent"): Toevoegen aan het einde. Je kunt bijvoorbeeld elementen toevoegen aan het einde van een list, of nieuwe data toevoegen aan het einde van een bestand. Dit wordt "appending" genoemd.
- *assignment* (spreek uit: "essajnment"): Letterlijk vertaald betekent assignment "toekenning." Een assignment is het geven van een waarde aan een variabele, middels de assignment operator (het is-gelijk teken). Een voorbeeld van een assignment is `x = 5`.
- *batch* (spreek uit: "betsj"): Een groep commando's die achter elkaar worden uitgevoerd. Dit slaat meestal op een serie Python programma's die je wilt uitvoeren zonder ze handmatig te moeten aansturen.
- *case-insensitive* (spreek uit: "kees-insenzittif"): Hoofdletters worden behandeld als kleine letters. In het Nederlands wordt dit soms "hoofdletterongevoeligheid" genoemd, maar dat is dergelijk krom taalgebruik dat ik liever de Engelse variant gebruik.
- *cast* (spreek uit: "kast")" Het omzetten van een item met een bepaald data type naar een item met een ander data type. Bijvoorbeeld, "list casting" zet een collectie om in een list.
- *class* (spreek uit: "klas"): Een klasse, dat wil zeggen, een beschrijving van de attributen en methodes die een bepaalde groep objecten gemeen hebben.
- *class call* (spreek uit: "klas kol"): De aanroep van een methode van een andere class.
- *command line* (spreek uit: "kommand lajn"): De plek waar je in de command shell commando's kunt intypen.
- *command shell* (spreek uit: "kommand sjel"): Een programma dat je toestaat om rechtstreeks met het besturingssysteem van je computer te communiceren door commando's in te typen.

- *crash* (spreek uit: "kresj"): Het plotseling eindigen van een programma doordat er een fout optreedt.
- *dictionary* (spreek uit: "diksjenerrie"): Letterlijk vertaald betekent dictionary "woordenboek." In Python is het een data type dat bestaat uit een verzameling sleutels met bijbehorende waardes, die gevonden kunnen worden als je de sleutel kent.
- *encoding* (spreek uit: "enkoding"): Letterlijk vertaald betekent encoding "versleuteling." Het verwijst ofwel naar de wijze waarop tekens in een bestand zijn opgeslagen (bijvoorbeeld als ASCII of Unicode), ofwel naar het vertalen van tekens naar een van die manieren van opslaan.
- *error* (spreek uit: "error"): Engels voor "fout."
- *exception* (spreek uit: "eksepsjun"): Engels voor "uitzondering." Een exception wordt gegenereerd door een programma tijdens de uitvoering als er iets fout gaat. Je kunt exceptions afvangen in je code, wat "exception handling" wordt genoemd.
- *float* (spreek uit: "floot"): Float is de afkorting voor "floating-point number," oftwel een "gebroken getal." Het is tevens de benaming van het data type dat gebroken getallen omvat.
- *handle* (spreek uit: "hendel"): Een handle is een variabele die toegang geeft tot een bestand.
- *inheritance* (spreek uit: "inherrittens"): Letterlijk vertaald betekent inheritance "overerving." Het is de benaming voor het mechanisme waarbij subclasses de eigenschappen en methodes krijgen van de class waarvan ze zijn afgeleid.
- *input* (spreek uit: "inpoet"): De invoer die de gebruiker aan een programma verstrekt door op het toetsenbord te typen, of die uit een bestand wordt gelezen.
- *integer* (spreek uit: "intedjer"): Een geheel getal.
- *interface* (spreek uit: "interfees"): In object georiënteerde programmeertalen: een class die gebruikt wordt om een definitie vast te leggen, maar die niet gebruikt kan worden om direct instanties van te creëren.
- *iterator* (spreek uit: "ajtereetor"): Een functie die items één voor één genereert.
- *key* (spreek uit: "kie"): Letterlijk vertaald betekent key "sleutel." Een key wordt gebruikt om een waarde in een dictionary te vinden. In het Nederlands wordt hier wel eens het woord sleutel voor gebruikt, maar in programmeertalen is het toch meer geaccepteerd om het woord key hiervoor te gebruiken.
- *keyword* (spreek uit: "kiewurt"): Een woord dat in Python gereserveerd is voor specifieke taken, en niet door de programmeur gebruikt mag worden als variabelenaam.
- *list* (spreek uit: "list"): Een list is een lijst. Ik gebruik het woord **`list`** omdat het een data type is.
- *list comprehension* (spreek uit: "list komprehensjen"): Comprehension betekent letterlijk "begrip," maar dat heeft niks van doen met de term "list comprehension." List comprehension is een techniek waarbij je een list creëert door een functionele beschrijving van de list te geven. Bijvoorbeeld, je schrijft niet `[1,4,9,16]`, maar in plaats daarvan schrijf je een Python uitdrukking die je kunt vertalen als: "alle kwadraten van gehele getallen tussen 1 en 20." Er is geen fatsoenlijke vertaling voor deze techniek beschikbaar.

- *loop* (spreek uit: "loep"): Letterlijk vertaald betekent loop "lus." Een loop is een iteratie in een programma, meestal via een **`while`** of een **`for`**.
- *newline* (spreek uit: "njoelajn"): Een "newline" is een nieuwe regel; het "newline" symbool, in Python voorgesteld als "\n", geeft als het voorkomt in een string aan dat de tekst vanaf dat punt verder moet gaan op een nieuwe regel.
- *output* (spreek uit: "outpoet"): De uitvoer van een programma die op het scherm wordt getoond, of die in een bestand wordt weggeschreven.
- *overloading* (spreek uit: "overloding"): Een deel van de term "operator overloading," die inhoudt dat je een betekenis geeft aan een operator voor nieuwe data types.
- *path* (spreek uit: "paff"): Een bestandsnaam inclusief de directorystructuur die precies aangeeft waar in het bestandssysteem het bestand zich bevindt.
- *pickling* (spreek uit: "pikling"): Het opslaan in een bestand van een hele data structuur via de "pickle" module.
- *pointer* (spreek uit: "pojnter"): Pointer betekent letterlijk "aanwijzer." Een pointer in een bestand is een variabele die refereert aan een specifieke positie in dat bestand.
- *print* (spreek uit: "print"): Letterlijk vertaald betekent print "afdrukken" (of "druk af"). In het Nederlands houdt dit meestal impliciet in "afdrukken op papier," maar in programmeertalen kan het ook betekenen "laten zien op het scherm."
- *queue* (spreek uit: "kjoe"): Een queue is een specifiek soort rij, namelijk een rij (in Python meestal geïmplementeerd als list) waarbij nieuwe elementen aan het einde van de rij worden toegevoegd, en elementen alleen verwijderd mogen worden vanaf het begin van de rij.
- *raise* (spreek uit: "rees"): "Raise an exception" betekent het actief genereren van een exception in een programma. Om een exception te genereren, gebruik je het gereserveerde woord **`raise`**.
- *root* (spreek uit: "roet"): De wortel van een boomstructuur, ofwel, de hoogstgelegen knoop in een boomstructuur, van waaruit de rest van de boom benaderd kan worden.
- *runtime error* (spreek uit: "runtajm error"): Een fout die optreedt tijdens het uitvoeren van een programma. Meestal hoort bij een runtime error een specifieke foutmelding van het programma.
- *statement* (spreek uit: "steetment"): Letterlijk vertaald betekent statement "opdracht," maar die vertaling is nauwelijks van toepassing op programma's. Een statement is één programma-regel, bijvoorbeeld een commando of een berekening.
- *tuple* (spreek uit: "tupel"): Het Nederlandse woord voor tuple is "tupel." Het is een verzameling van één of meerdere waardes die bij elkaar horen en in een vaste volgorde staan. Ik had eventueel de Nederlandse schrijfwijze kunnen aanhouden, maar omdat **`tuple`** een data type is, heb ik besloten toch maar de Engelse schrijfwijze te gebruiken.
- *underscore* (spreek uit: "underskoor"): het "laag liggende brede streepje" dat zich op de meeste toetsenborden bevindt op dezelfde toets als het min-teken.

Index