

Python voor kids

Beginnen met programmeren



Johan Vereecken

Python voor kids, beginnen met programmeren

Versie 1.0.1.

Mol, september 2015

Na Scratch te zijn ontgroeid is Python voor kinderen een vrij logische keuze. Dit boek is deels gebaseerd op het boek *Snake Wrangling For Kids* van Jason R. Briggs. Origineel was ik begonnen met een vertaling van dat boek omdat mijn kinderen de Engelse taal nog niet machtig waren. Er bestond nog geen Nederlandstalig Pythonboek specifiek gericht naar de jeugd alsmede was er de indruk dat in het middelbaar onderwijs nogal weinig aandacht besteed werd aan echt programmeren. Nadien kwam de idee om het geheel met een eigen interpretatie te schrijven en open te stellen naar het publiek. Dit mede ingegeven door de tekorten aan IT'ers met open source ervaring in mijn professioneel leven. Ook is dit gestuwd door de positieve vibes van de open sourcegemeenschap en Python in 't bijzonder, maar tevens door de ontwikkeling van de goedkope Raspberry Pi waarmee via Python elektronica aangestuurd kan worden. Dit boek dient niet enkel voor zelfstudie, maar kan tevens gebruikt worden in scholen, gedurende sessies van CoderDojo, Hour of Code en andere initiatieven. Het vraagt van de lezer vooral een frisse en experimenteel ingestelde geest. Het is zeker niet uit te sluiten dat soms een zweetdruppeltje op het voorhoofd zal parelen door het nadenken.

Dank gaat uit naar Guido van Rossum voor zijn immense bijdrage tot de ontwikkeling van Python en om deze programmeertaal als open source ter beschikking te stellen. Ook wil ik Jason R. Briggs danken om me ideeën te geven voor de opmaak van dit boek. En merci aan mijn kinderen die als kritische betatesters me de commentaren niet gespaard hebben. Mijn levensgezellin dank ik om me de ruimte gegeven te hebben om dit boek op te stellen. Big thanks gaat naar Sven Claessens (CoderDojo coach in Mol) om dit boek te reviewen.

Als ik iemand zou vergeten te bedanken zijn is dit volledig te wijten aan enige premature dementie van mijn kant.

De figuren die ik niet zelf opgenomen heb komen van [openclipart.org](http://search.creativecommons.org/) en pixabay.com via de zoekrobot van <http://search.creativecommons.org/>. Het Python logo komt van de Python Software Foundation.

Licentie:



Dit werk valt onder een [Creative Commons Naamsvermelding-NietCommercieel-GelijkDelen 4.0 Internationaal-licentie](https://creativecommons.org/licenses/by-nc-sa/4.0/).

Johan, lead-coach CoderDojo Mol

Inhoud

INHOUD	2
1. PRELUDE	6
2. GAMEN OF PROGRAMMEREN	7
2.1. Enkele woorden over taal	7
2.2. De orde van de niet-giftige wurgslang	8
2.3. Je eerste Python programma	9
2.4. Je tweede Python programma... terug hetzelfde?	10
3. 6 VERMENIGVULDIGD MET 2,52 IS GELIJK AAN	11
3.1. Haakjes gebruiken en de "Orde van Operaties"	13
3.2. Niets zo wispelturing als een variabele	14
3.3. Variabelen gebruiken	17
3.4. Een stukje tekst (String)	18
3.5. Trucjes met strings	19
3.6. De tuple of het lijstje	20
Items vervangen in lijstjes	22
Items toevoegen aan het lijstje	22
Items verwijderen	23
2 lijsten zijn beter dan 1	23
3.7. Tuples en Lijsten	24
3.8. Een Map zonder papieren	25
3.9. Dingen om zelf eens te doen	27
4. SCHILDPADDEN EN ANDERE TRAGE BEESTEN	28
4.1. Dingen om zelf eens te doen	32
5. HOE EEN VRAAG STELLEN?	33
5.1. Doe dit ... of ANDERS !!!	35
5.2. Doe dit... of doe dit... of doe dit... of ANDERS!!!	35
5.3. Conditie samenvoegen	36
5.4. De grote leegte	37
5.5. Wat is het verschil?	38

6. OPNIEUW EN OPNIEUW EN OPNIEUW	41
6.1. Wanneer is een blok geen rechthoek?	43
6.2. Nu we toch lussen maken	49
6.3. Dingen om zelf eens te doen	51
7. WE GAAN WAT RECYCLEREN	52
7.1. Stukjes en brokjes	55
7.2. Modules	57
7.3. Dingen om zelf eens te doen	59
8. KLASSEN EN OBJECTEN	61
8.1. Opdelen in klassen	61
Kinderen en ouders	62
Objecten toevoegen aan klassen	63
Klasse-functies aanmaken	63
Klasse karakteristiek als functie toevoegen	64
Waarom klassen en objecten gebruiken?	65
Objecten en klassen in figuren	67
8.2. Andere eigenschappen van objecten en klassen	68
Functies erven	69
Functies die andere functies oproepen	70
8.3. Een object initialiseren	71
9. IETS OVER BESTANDEN	73
10. HIER ZIJN DE SCHILDPADDEN TERUG	74
10.1. Kleuren gebruiken	77
10.2. Het wordt donker	78
10.3. Vormen vullen	79
10.4. Dingen om zelf eens te doen	83
11. EN NU GRAFISCH	85
11.1. Snel tekenen	85
11.2. Simpel tekenen	87
11.3. Vierkantjes tekenen	88
11.4. Bogen tekenen	94
11.5. Ovalen tekenen	96

11.6.	Veelhoeken tekenen	97
11.7.	Tekeningen maken	98
11.8.	Basis animaties	99
11.9.	Actie en reactie	101
12.	POSTLUDE	104
13.	BIJLAGE A: TREFWOORDEN	105
	and	105
	as	105
	assert	106
	break	106
	class	107
	del	107
	elif	107
	else	107
	except	108
	exec	108
	finally	108
	for	108
	from	109
	global	110
	if	110
	import	111
	in	111
	is	111
	lambda	111
	not	112
	or	112
	pass	113
	print	114
	raise	114
	return	114
	try	115
	while	115

with	115
yield	115
14. BIJLAGE B: FUNCTIES	116
abs()	116
bool()	116
dir()	117
eval()	118
float()	118
int()	119
len()	119
max()	120
min()	120
open()	120
range()	121
sum()	121
15. BIJLAGE C: MODULES	123
Module random	123
Module sys	124
Module time	125
16. BIJLAGE D: OPLOSSINGEN	131
16.1. Hoofdstuk 3	131
16.2. Hoofdstuk 4	131
16.3. Hoofdstuk 5	132
16.4. Hoofdstuk 6	132
16.5. Hoofdstuk 7	133
16.6. Hoofdstuk 10	134
17. VERSIEBEHEER	136

1. Prelude

Een voorwoordje aan de ouders (van kinderen tot ongeveer 12 jaar oud)...

In dit boek wordt gebruik gemaakt van Python. Dit is een high-level computertaal uit de open sourcewereld. Je kan deze gratis downloaden (<https://www.python.org/>) en gebruiken. Python is één van de gemakkelijkste talen om te leren, te gebruiken en tegelijkertijd zeer krachtig. Ze wordt gebruikt door veel professionele programmeurs. Je dient de laatste versie van Python te installeren op je computer, want het boek gaat uit van Python v3.x. Deze computertaal is beschikbaar onder verschillende besturingssystemen, zowel onder de commerciële als die uit de open source wereld. Dit boek kan onmogelijk een beschrijving geven van de verschillende installatiemethoden voor alle besturingssystemen, maar in de Linux, Windows, Mac OSX en BSD-wereld zijn genoeg beschrijvingen te vinden om dit gemakkelijk te realiseren. Zelfs ik ben erin geslaagd om het tot een goed einde te brengen. Wil of kan je echt geen Python installeren, dan kan je een deel van de code in dit boek ook uitvoeren via <http://www.learnpython.org/>.

Het kan best zijn dat je de eerste hoofdstukken samen met je kind moet doorlopen, maar laat het na een tijdje vooral zelf verder doen. Dit is afhankelijk van kind tot kind. Je kind zal wel aangeven wanneer je niet meer nodig bent en het toetsenbord zelf opeisen...

Je kind zou minstens de basisfunctionaliteit van een teksteditor moeten kunnen gebruiken. Neen, geen tekstverwerker zoals MS Word, Libre Office Writer of Pages, want daarmee kan je niet programmeren. Maar wel een tekst editor zoals Notepad++, Leafpad, Nano, Textmate,... Deze basisfuncties zijn het aanmaken van een nieuw tekstbestand, het openen en sluiten, alsook het bewaren van tekstbestanden. Heden bevat Python bij de installatie voor alle besturingssystemen een IDE (een geïntegreerde ontwikkelomgeving), m.n. IDLE, die je in plaats van een teksteditor of ook als console kan gebruiken. De rest van de basis van Python wordt beschreven in dit boek.

2. Gamen of programmeren

Brrrrr, leren programmeren... Dat is toch niets voor mij? Gamen, ja, da's pas fijn, maar waarom zou ik nu willen programmeren. Dat is toch enkel voor nerds?

Misschien heb je net een laptop gekregen? Pa of ma heeft nog een oude computer die daar toch maar in de kast ligt. Nonkel Lieven werkt in een bedrijf waar ze net nieuwe computers gaan installeren en hij kan een oude voor jou recupereren. Of je bent hip, experimenteert graag en je koopt jezelf een experimenteerbord zoals de Raspberry Pi voor enkele tientallen euro. Whatever. Vraag je ouders of een vriend om er een open source besturingssysteem op te zetten, of doe het gewoon zelf. Het kost je geen cent, enkel wat moeite (maar niet echt veel). Dan kan je je zakgeld aan nuttigere dingen besteden.

Je speelt graag games, maar je zou eigenlijk wel eens willen weten hoe ze die maken en je wil misschien ook zelf eens een game maken. Go forward. Als je dit boek uit hebt gelezen en veel hebt gespeeld en geëxperimenteerd met wat erin staat ben je al zeer goed op weg om iets moois te kunnen maken.

Dit boek vertelt je iets over wat computers kunnen. Geen gedoe over de hardware. Nee, we gaan het hebben over wat er gebeurt waardoor computers doen waarvoor ze gemaakt zijn: het uitvoeren van programma's en opdrachten.

Zonder programma's zouden computers enkel kunnen dienen als tafeltje of dienblad. En zelfs dan zouden ze niet echt nuttig zijn, want daarvoor zijn ze te onhandig of te klein.

Misschien heb je wel een Wii of een PS of een Ipod. Ook die hebben programma's nodig om te kunnen werken. De TV, de Blu-ray-speler, de wagen van je ouders, je drone, ze hebben allemaal computerprogramma's om te kunnen functioneren. Sommige hebben simpele programma's, andere functioneren enkel met heel ingewikkelde code.

Als je kan programmeren kan je allerlei nuttige dingen doen zoals je eigen games maken, je eigen website opzetten, een robot aansturen, zelf een alarminstallatie bouwen zodat je weet wanneer broer of zus je kamer is binnengekomen. Misschien kan het je wel helpen om je huiswerk te maken...

2.1. Enkele woorden over taal

Net zoals mensen, maar ook walvissen, dolfijnen en misschien ook wel ouders (maar daar moet je het wel eerst eens met je vrienden over hebben), gebruiken computers

ook een eigen taal. En net zoals mensen kennen ze zelfs meerdere talen. Er zijn zoveel talen dat ze bijna het ganse alfabet bevatten, zo zijn er A, B, C, D maar ook BASIC, FORTRAN, ASSEMBLER, PYTHON, enz.

Er bestaan programmeertalen die genoemd zijn naar mensen (zoals PASCAL), er zijn er ook die genoemd zijn naar bekende televisiereeksen en er zijn er die rare tekens bevatten (zoals C++ en ook C#). Er zijn zelfs talen die op elkaar lijken en slechts een weinig van elkaar verschillen. De uitvinders hadden blijkbaar toch niet zoveel verbeelding. Er zijn ook talen die verdwenen zijn, toch blijven er nog steeds zeer veel talen over. Maar, we gaan het over slechts 1 taal hebben in dit boek, anders zijn we over 10 jaar nog bezig en dan kunnen we beter gewoon stoppen.

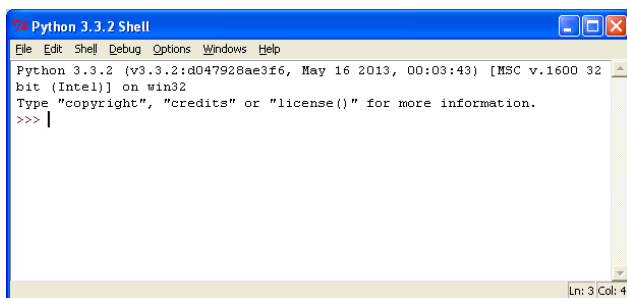
2.2. De orde van de niet-giftige wurgslang

Of beter gezegd... Python.

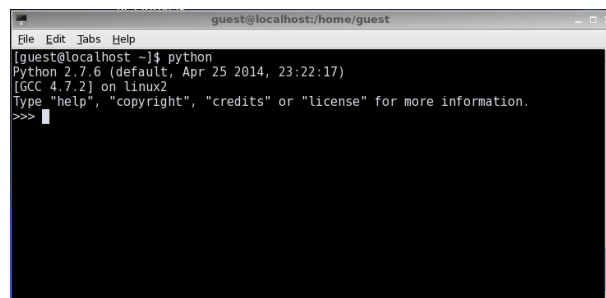
Een Python is niet enkel een slang, maar tevens een programmeertaal. Maar, de taal is niet genoemd naar een reptiel, wel is het een van de weinige talen die genoemd is naar een TV-show. Monty Python's Flying Circus was een zeer populaire komedie in de jaren '70 met absurde Britse humor (net zoals The Young Ones, Bottom,...).

Er zijn een aantal dingen betreffende Python (de computertaal, niet de komedie en ook niet de slang) die het zeer interessant maken om ermee te leren programmeren. Voor ons is het belangrijkste dat je direct kan starten en snel programma's kan maken.

Daarom is het belangrijk dat je ma of pa het voorwoord hebt laten lezen als de computer niet van jou is. Vraag hen om een terminalsessie op te starten in een Linux-distributie (bijv. Konsole) en dan **python3** in te tikken of in Windows IDLE3 te selecteren. Je bekomt dan een scherm gelijkaardig als in Figuur 1 of Figuur 2.



Figuur 1: Python-console in Windows



Figuur 2: Python-console in Linux

Als je bemerkt dat je ouders het voorwoord niet gelezen hebben... omdat je merkt dat er iets niet goed functioneert... Ga dan terug naar het begin en stop het boek onder hun neus als ze de krant proberen te lezen of op hun smartphone of tablet bezig zijn. Bekijk hen met een lief snoetje en probeer “please, please, please” tot het vervelend wordt voor hen en ze uit hun zetel komen. Als dat echt niet helpt, kan je het gewoon ook zelf doen. Eventueel een beetje googlen als het niet direct lukt of je wat extra hulp nodig hebt. Normaal zou het geen probleem mogen opleveren.

2.3. Je eerste Python programma


OK, je bent nu al zover geraakt. Je bent in de Python-console (of Python shell). Dit is een manier om Python commando's en programma's te laten lopen. Bij de opstart van de console (of na het invoeren van een commando) zie je wat we een ‘prompt’ noemen. Bij Python bestaat die uit drie opeenvolgende ‘groter dan’ tekens:

```
>>>
```

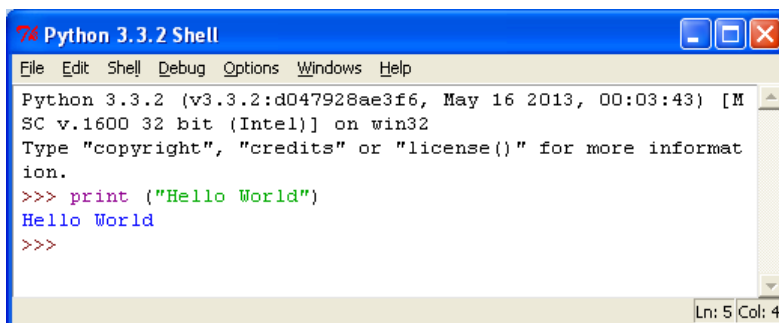
Als je genoeg Python commando's samen neemt bekom je een programma dat je buiten de console kan draaien, maar we gaan het nu nog simpel houden: we typen onze commando's direct in de console bij de prompt (>>>). We beginnen zoals alle cursussen starten met het wereldberoemde “**Hello World**” (Engels voor *Dag Wereld*). Laat ons afspreken dat alles wat in het boek in **vet** en tegelijkertijd in het lettertype `courier` gedrukt staat achter de prompt (>>>) door jou ingetypt wordt, zoals hieronder met groene letters in een zwarte kader. Wat niet in het vet lettertype staat is datgene wat de computer antwoordt. We typen het volgende na de cursor (zie ook Figuur 3) en vergeet de aanhalingstekens niet:

```
>>> print ("Hello World")
>>>
```

print betekent gewoon: druk af.

Na het invoeren van deze tekst op het einde van de lijn druk je op de -toets. En hopelijk zie je direct onder je invoer iets verschijnen zoals dit:

```
Hello World
>>>
```



Figuur 3: Eerste programma

Daarna zie je onmiddellijk terug de prompt verschijnen. Dit betekent dat de Python-console klaar is voor nieuwe commando's.

Proficiat !! Je hebt net je eerste Python programma geschreven. Was het nu zo moeilijk?

Neen, hé. Het commando `print` is een functie die alles naar de console schrijft wat tussen de haakjes staat. We zullen het commando nog veel gebruiken.

2.4. Je tweede Python programma... terug hetzelfde?

Python programma's zouden niet erg bruikbaar zijn als je elke keer opnieuw de programma's zou moeten invoeren als je ze wil laten lopen. Of als je een programma voor iemand zou maken en die persoon zou het elke keer opnieuw moeten intypen.

Je kent wel de tekstverwerker die je ouders gebruiken of die je misschien op school gebruikt. Maar daar kunnen we echt niks mee aanvangen om te programmeren. Die tekstverwerker maakt echt een rommelboeltje van ons programma, zodat de computer echt niet meer weet wat ie moet doen. Daarom gebruiken we een editor (zoals Notepad, Notepad++, Leafpad,...). Als je niet weet welke editor op je computer staat, vraag dan even na bij ma of pa en onthoud het voor de volgende keer, zodat je hen niet terug moet lastig vallen. Anders beginnen ze misschien te morren.

Open de teksteditor en typ het commando exact zoals je het in de console hebt ingegeven:

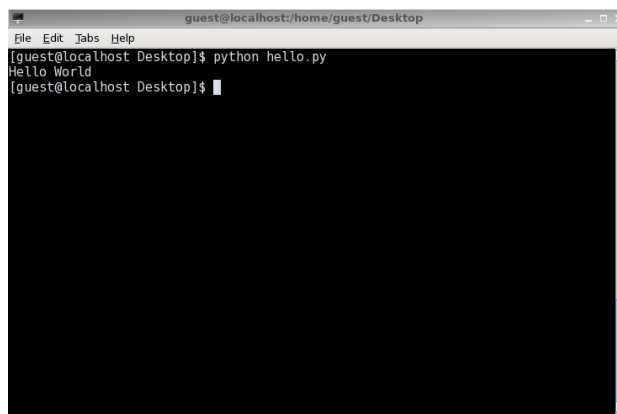
```
print ("Hello World")
```

Klik met de cursor op het Bestand (of File) menu en bewaar (Save) het. Wanneer je om een bestandsnaam gevraagd wordt (Filename) noem het dan "Hello.py" en bewaar het in je homedirectory (of thuisfolder). Misschien moet je eventjes zoeken naar deze folder of directory.

In Linux open je terug de Terminal (zoals bijv. Konsole) en typ:

```
python hello.py
```

Je ziet terug Hello World verschijnen in het venster net zoals eerder in de console:



```

guest@localhost:/home/guest/Desktop
File Edit Tabs Help
guest@localhost Desktop]$ python hello.py
Hello World
guest@localhost Desktop]$

```

Figuur 4: Console met programma dat tekstbestand uitvoert


Zoals je ziet hebben de makers van Python eraan gedacht om je heel veel typwerk te besparen. Doe je vader maar eens denken aan de tijd toen ie nog op zijn C64 of zijn TI994/A of zijn TRS80 werkte en de cassette recorder niet goed functioneerde. Toen moest ie alles opnieuw intypen. Lach hem maar eens goed uit, maar sta wel klaar om hard weg te kunnen lopen...

Dit is niet het begin van het einde, maar het einde van het begin.

Welkom in het wonderbaarlijke rijk van het Programmeren. We zijn echt heel simpel gestart met de “Hello World” applicatie. In de volgende hoofdstukken gaan we meer bruikbare dingen uitvoeren met de Python-console en verder kijken wat er bij programmeren allemaal komt kijken.

3. 6 vermenigvuldigd met 2,52 is gelijk aan

Hoeveel is 6 vermenigvuldigd met 2,52? Een rekenmachine zou nu wel handig zijn, nietwaar. Misschien ben je wel een rekenwonder en kan je dit in je hoofd uitrekenen, maar kan je datzelfde dan ook voor 12365,5432 maal 42342,764? Da’s al iets moeilijker om uit het hoofd te rekenen. Je kan hetzelfde ook uitrekenen met Python. Je start de console terug op (of IDLE3) zoals hiervóór werd beschreven. Als je de prompt ziet typ je **6*2.52**

Let wel op dat je als decimaal scheidingssymbool een punt (.) gebruikt en geen komma (,), net zoals op je rekenmachine. Bij computers gebruiken we ook het asterisk symbool (*) als maalteken en niet het x of X symbool, want anders weet de computer niet of je een letter wil gebruiken of je wil vermenigvuldigen (zo slim is een computer nu ook weer niet). Druk vervolgens op de -toets.

```

>>> 6*2.52
15.120000000000001
>>>

```

W817... Wat is er aan de hand met Python?

Rekent die nu fout? Als ik met mijn rekenmachine 6 x 2.52 bereken dan bekom ik 15,12. Waarom geeft Python me nu zo’n raar getal na de komma? Kan Python niet rekenen, maakt ie nu zo’n fouten? Wel,

eigenlijk is Python niet fout, andere programma's hebben dat probleem ook. De reden moet gezocht worden hoe de computer zelf omgaat met fractionalen (getallen met een decimaal). Het is een complex en verwarrend iets als je net begint te programmeren, maar wat je best onthoudt is dat het resultaat van een berekening met decimalen soms niet helemaal exact is wat je verwachtte. Dit is zo voor vermenigvuldiging, voor deling, maar ook voor optelling en aftrekking.

Maar nu misschien een vergelijking die iets meer bruikbaar is. Stel dat je €5 per week zakgeld krijgt en elke week ook €7 bijverdient door je oma te helpen, hoeveel verdien je dan op een jaar tijd als je alles in je spaarvarken stopt?

Als we dit zouden uitschrijven op papier, dan zouden we iets bekomen zoals:

$$(5 + 7) \times 52$$

Wat dus 5 + 7 euro is per week en dat is gelijk aan €12. Dit vermenigvuldigen we dan met 52 weken (zoveel zijn er in 1 jaar). Dit rekt iets gemakkelijker:

$$12 \times 52 = 624 \rightarrow \text{dus we bekomen } €624 \text{ per jaar.}$$

In Python met de console bekomen we ook

```
>>> (5+7)*52
624
>>> 12*52
624
>>>
```

En als we nu eens voor €8 per week aan snoepjes zouden kopen? Hoeveel hebben we dan nog over op het einde van het jaar? We zouden ook dit terug op papier op meerdere manieren kunnen uitrekenen, maar tevens door de computer in de Python-console:

```
>>> (5+7-8)*52
208
>>>
```

We zouden dus nog €208 overhouden.

Dit is niet echt bruikbaar, want we kunnen dit ook met een rekenmachine uitrekenen. We komen hier later nog op terug en tonen je hoe we er veel beter gebruik kunnen van maken.

In de Python-console kunnen we optellen, vermenigvuldigen, aftrekken en delen. Dat allemaal met nog een heleboel andere wiskundige bewerkingen waar we nu nog niet mee gaan werken. Eerst gaan we werken met de wiskundige basissymbolen in Python (feitelijk noemen we ze 'operatoren') en deze zijn:

+	Optelling
-	Aftrekking
*	Vermenigvuldiging
/	Deling

Tabel 1: De wiskundige basisoperatoren

De reden waarom we een schuine streep (/) gebruiken als deling is omdat het nogal moeilijk is om een lijn als deling te tekenen. Ook zijn ze blijkbaar op het toetsenbord vergeten het gewone deelteken (÷) te zetten zoals je op school moet gebruiken voor breuken. Als je bijvoorbeeld zou willen weten hoeveel snoepjes er per doosje kunnen zitten als je 1000 snoepjes in 25 doosjes moet stoppen. Dan moet je 1000 delen door 25 en dan zou je de volgende deling schrijven:

$$\frac{1000}{25}$$

Of je zou kunnen werken met een staartdeling, maar je zou zelfs kunnen schrijven:

$$1000 \div 25$$

En dan moet je gaan zoeken naar het ÷ symbool op je toetsenbord. Moeilijk te vinden hé. Ik heb het nog niet gevonden op het toetsenbord. In Python schrijf je gewoon:

```
>>>1000/25
>>>
```

Dat is toch veel gemakkelijker zou ik denken, maar ik ben dan ook slechts een boek, natuurlijk.

3.1. Haakjes gebruiken en de “Orde van Operaties”

Een operatie is het gebruik maken van een operator (één van de symbolen die je terugvindt in de bovenstaande Tabel 1). Er zijn veel meer operatoren dan enkel deze uit die korte lijst (optelling, aftrekking, vermenigvuldiging en deling). Je moet wel weten dat vermenigvuldiging en deling vóór de optelling en de aftrekking komen bij de berekeningen. Met een moeilijke uitdrukking zeggen grote mensen soms ook dat ze een hogere orde hebben. Dat betekent dat je eerst vermenigvuldigingen en delingen uitrekent en daarna pas de optelling en de aftrekking. We zullen dit aantonen met enkele voorbeeldjes (want het is heel belangrijk om dat te onthouden, ook bij de wiskunde op school).

Hieronder zijn alle operatoren optellingen (+) en alles wordt achtereenvolgens opgeteld, je mag dit in je Python-console typen en ook wat experimenteren natuurlijk:

```
>>> print (10 + 25 + 15)
50
>>>
```

Op gelijkaardige wijze: indien er enkel optellingen en aftrekkingen zijn als operatoren, dan bekijkt Python ze in de volgorde waarin ze getypt zijn:

```
>>> print (10 + 25 - 15)
20
>>>
```

Maar in de volgende opdracht zit een vermenigvuldigingsoperator waarbij het product van de twee getallen 10 en 20 eerst berekend wordt en pas daarna bij het getal 5 wordt

opgeteld. Want zoals hoger gezien is de vermenigvuldiging van een hogere orde dan de optelling.

```
>>> print (5 + 10 * 20)
205
>>>
```

Maar wat gebeurt er nu als we haakjes plaatsen? Haakjes hebben voorrang op de operatoren en we zien het verschil hieronder:

```
>>> print ((5 + 10) * 20)
300
>>>
```

Dit resultaat (300) is echt wel verschillend van het vorige resultaat (205). Net zoals in de wiskundeles berekent ook Python eerst wat binnen de haakjes staat en daarna wat erbuiten staat. Hier zeggen we dat we eerst 5 en 10 optellen, wat 15 oplevert en daarna gaan we dit getal 15 vermenigvuldigen met 20. En daaruit komt 300.

We kunnen het nog wat moeilijker maken door haakjes te plaatsen binnen haakjes. Je moet dan wel opletten wanneer je haakjes opent en wanneer je ze terug sluit. Om je een voorbeeld te geven kijk je maar even hieronder en experimenteer zelf ook maar eens.

```
>>> print (((5 + 10) * 20) / 10)
30
>>>
```

Hier gaat Python eerst kijken naar de binnenste haakjes (5 + 10) wat 15 is, en dan naar diegene daarbuiten (15 * 20) wat 300 oplevert. Tenslotte wordt door Python gekeken naar de andere operator, nl. de deling door 10. Hieruit bekomen we dan 30. Als we de haakjes zouden weglaten, dan wordt iets anders bekomen:

```
>>> print 5 + 10 * 20 / 10
25
>>>
```

Of ook zonder het `print` commando:

```
>>> 5+10*20/10
25
>>>
```

En dit is toch wel verschillend van het commando met de haakjes.

Dus denk eraan: vermenigvuldiging en deling komen vóór optelling en aftrekking.

3.2. Niets zo wispelturing als een variabele

Een variabele is een programmeerterm die een plaats beschrijft waar je dingen tijdelijk bewaart. Deze dingen kunnen getallen of tekst maar ook lijstjes met getallen en tekst en een heleboel andere zaken zijn. Eigenlijk nog teveel om nu reeds op te noemen. Als je hoort over variabelen denk dan aan iets dat lijkt op een brievenbus. Je kan zaken zoals brieven of pakjes

in de brievenbus stoppen, net zoals je zaken (getallen, tekst, lijstjes, enzovoort) in een variabele stopt. De meeste programmeertalen werken op deze manier met variabelen, maar niet allemaal.

Bij Python werken variabelen een beetje anders. Ze zijn niet echt een brievenbus met zaken



erin, maar een variabele is een beetje zoals een post-it briefje dat op de buitenkant van de brievenbus kleef. Je kan dat kleefbriefje eraf nemen en op iets anders kleven en als het kleefbriefje groot genoeg is zelfs op meer dan één ding kleven. We maken een variabele door deze een naam te geven, dan het “gelijk aan”-teken (=) te gebruiken en daarna tegen Python te vertellen welke waarde we aan die naam willen geven. Dit lijkt allemaal moeilijk, maar een voorbeeld zal het wel duidelijker maken:

```
>>> joepie = 200
>>>
```

We hebben net een variabele gemaakt die **joepie** noemt en die we de waarde 200 gegeven hebben. Zo vertellen we Python dat het dit getal moet onthouden omdat we het nadien nog willen gebruiken. Als we zouden vergeten zijn (een boek kan moeilijk onthouden, nietwaar) waar de variabele naar verwijst, dan typen we in de console gewoon **print** gevolgd door de naam van de variabele die tussen haakjes staat. Bijvoorbeeld:

```
>>> joepie = 200
>>> print (joepie)
200
>>>
```

We kunnen Python ook vertellen dat deze variabele naar iets anders moet verwijzen. Zo kan de variabele ook 150 zijn en dan vragen we in de tweede regel om aan te tonen dat dit ook zo is met het **print** commando:

```
>>> joepie = 150
>>> print (joepie)
150
>>>
```

We kunnen ook meer dan één naam aan hetzelfde geven:

```
>>> joepie = 150
>>> hoera = joepie
>>> print (hoera)
150
>>>
```

In deze code vertellen we Python dat we de naam (of het kleefbriefje) **hoera** willen toekennen aan hetzelfde waarnaar **joepie** verwijst, namelijk 150.

Zoals reeds gezegd kan een variabele ook een tekst zijn. We kunnen **joepie** bijvoorbeeld ook de waarde "Hier is Python" geven, maar dan moet je wel enkele (') of dubbele (") aanhalingstekens gebruiken. Je mag zelf kiezen:

```
>>> joepie = "Hier is Python"
>>> print (joepie)
Hier is Python
>>>
```

Je hoeft niet verlegen te zijn om verder te experimenteren. Doe maar verder. En als je iets fout doet, dan kan je eruit leren. Of misschien ontdek je zo wel iets nieuw.

Wel, ik moet toegeven dat **joepie** of **hoera** niet echt nuttige namen zijn voor een variabele. Het zegt niet waarvoor we het gebruiken. Een brievenbus is makkelijk, deze wordt gebruikt voor de post, maar een variabele kan voor meerdere dingen gebruikt worden en kan verwijzen naar een heleboel verschillende dingen. Dus we willen dat onze variabele een nuttige naam heeft zodat we nadien nog weten waarvoor deze daarna gebruikt wordt. Stel dat je de Python-console opgestart hebt, je dan **joepie = 150** intypte en 2 jaar zou gaan gamen, skaten, snorkelen of basketten. Wanneer je nadien terug zou komen aan je computer, zou je dan nog weten waarvoor dat getal 150 diende?

Ik denk niet dat ik het nog zou weten.

Ik heb dit getal daarstraks uitgevonden en ik heb eigenlijk geen idee wat **joepie = 150** betekent, behalve dat het een naam is van een variabele die verwijst naar het getal 150. Dus na twee jaar zou ook jij het echt niet meer weten.

Maar, wat als we de variabele als volgt zouden noemen: **geld_van_verjaardag**.

```
>>> geld_van_verjaardag = 150
>>>
```

We mogen dit doen omdat de namen van variabelen opgemaakt mogen worden uit letters, cijfers en onderstrepingen (_), maar ze mogen nooit starten met een cijfer en ze mogen geen spaties bevatten. Er zijn nog wel wat regeltjes, maar dat is meer iets voor als je al goed kan programmeren. Als je na 2 jaar terugkomt, dan weet je nog dat **geld_van_verjaardag** je iets zegt. Je kan in de Python-console het volgende intypen:

```
>>> print (geld_van_verjaardag)
150
>>>
```

En dan weet je onmiddellijk dat we praten over de €150 van je verjaardag. Het is niet altijd heel belangrijk om nuttige namen voor variabelen te gebruiken, maar meestal wel. Je kan bijna alles gebruiken van een enkele letter (bijvoorbeeld 'a') tot lange zinnen (maar zonder spaties). Soms, als je iets snel in elkaar wil steken, is een simpele en korte naam voor een

variabele even goed te gebruiken. Het hangt ervan af: als je later je programma nog eens wil bekijken of veranderen wil je graag weten waaraan je dacht toen je deze naam gebruikte.

`dit_is_ook_een_variabele_maar_iets_minder_gemakkelijk_bruikbaar`

3.3. Variabelen gebruiken

Nu we weten hoe een variabele te maken wordt het tijd dat we leren hoe ze gebruikt moeten worden. Herinner je jezelf nog de berekeningen die we voorheen gemaakt hebben? Jawel, die berekening waar je €5 zakgeld per week krijgt, en je van je oma voor de klusjes €7 verdient maar dat je goed je tanden moet poetsen voor die wekelijkse €8 aan snoepjes.



```
>>> print (5+7-8)*52
208
>>>
```

Als we nu de drie eerste getallen in variabelen zouden veranderen? Probeer eens het volgende in te geven in de Python-console:


```
>>> zakgeld = 5
>>> klusjes_oma = 7
>>> snoepjes = 8
>>>
```

We hebben zonet de variabelen `zakgeld`, `klusjes_oma` en `snoepjes` gemaakt. We kunnen de berekening dan opnieuw maken met deze variabelen:

```
>>> print ((zakgeld + klusjes_oma - snoepjes) * 52)
208
>>>
```

wat net hetzelfde antwoord oplevert. Maar als je nu per week €3 meer zou krijgen omdat je meer klusjes opknapt bij je oma? Dan moet je deze variabele veranderen naar 10. Daarna kan je enkele keren op de -toets drukken tot je de berekening terug hebt en daarna op de -toets drukken.

```
>>> klusjes_oma = 10
>>> print ((zakgeld + klusjes_oma - snoepjes) * 52)
364
>>>
```

Dat is nogal wat minder typen dan wanneer je alles opnieuw zou moeten invoeren om te zien dat je nu op een jaar meer verdiend hebt. Je kan ook de andere variabelen eens veranderen om te zien welk effect dat heeft. Ga daarna met de -toets zoeken naar de berekening om ze terug uit te voeren. Probeer eens te berekenen hoeveel je overhoudt als je per week de helft meer geld aan snoepjes zou geven, dus €12 in plaats van €8 (herinner je dat je intussen €10 voor je klusjes verdient):

```
>>> snoepjes = 12
>>> print ((zakgeld + klusjes_oma - snoepjes) * 52)
156
```

```
>>>
```

Je houdt dus slechts €156 over op het einde van het jaar. Al deze berekeningen zijn dus bruikbaar, maar nog niet zoals het in het echt is.

Maar voor nu is het genoeg dat je weet dat variabelen gebruikt worden om dingen (getallen, tekst, lijsten, enz) op te slaan. Denk aan de post-it kleefbriefjes!

3.4. Een stukje tekst (String)

Als je het jezelf goed herinnert heb ik gemeld dat variabelen voor allerlei zaken gebruikt kunnen worden, niet enkel cijfers. Bij het programmeren noemen we een stuk tekst meestal een “string”. Neen, niet het ondergoed dat je mama misschien heeft. Eerder hebben we reeds een string gebruikt bij de variabelen: “**Hier is Python**”, weet je nog?

Al wat je hoeft te weten over een string is dat het een hoop letters of cijfers en eventueel andere symbolen (bijvoorbeeld: -, _, ?, €, ...) zijn die op een betekenisvolle manier zijn samengebracht. Alle letters, cijfers en symbolen die gebruikt zijn in dit boek zouden een string kunnen voorstellen. Jouw naam zou een string kunnen zijn, maar ook de straat waarin je woont of het dorp. Maar tevens de naam van je beste vriend of vriendin is een string. Het eerste Python programma dat we in hoofdstuk 1 maakten gebruikte een string: “**Hello World**”, weet je nog?

Een string in Python wordt gemaakt door rond de tekst aanhalingstekens te gebruiken. We kunnen onze niet meer gebruikte variabele `joepie` gebruiken om er een string in te stoppen.

```
>>> joepie = "Dit is een string"
>>>
```


En we kunnen zien wat er in deze variabele zit door hem af te drukken op het scherm:

```
>>> print (joepie)
Dit is een string
>>>
```

We moeten geen dubbele aanhalingstekens (") gebruiken, enkele aanhalingstekens (') mogen ook, zoals reeds gezegd:

```
>>> joepie = 'En dit is een andere string'
>>> print (joepie)
En dit is een andere string
>>>
```

Let op: als je meer dan één regel tekst wil gebruiken in je string met enkele of dubbele aanhalingstekens en je let niet op dan krijg je een foutboodschap in de Python-console.

Probeer eens de volgende lijn in te tikken en druk dan de -toets. Dan krijg je een

foutboodschap die lijkt op wat hieronder staat. Het kan een beetje anders zijn als je een andere versie van Python gebruikt.

```
>>> joepie = "Dit zijn twee
SyntaxError: EOL while scanning string literal
>>>
```

Over fouten (of in 't Engels: *Errors*) zullen we later nog spreken. Wat je nu moet weten is dat als je meer dan één regel tekst wil, je dan 3 enkele aanhalingstekens (""") moet gebruiken aan het begin van de tekst en op het einde ervan:

```
>>> joepie = '''Dit zijn twee
lijnen tekst in een string.'''
>>>
```

Als je dit dan op het scherm print:

```
>>> print (joepie)
Dit zijn twee
lijnen tekst in een string.
>>>
```

Als je binnen een string aanhalingstekens wil gebruiken dan moet je een zogenaamde escape sequence gebruiken. Voor een enkel aanhalingsteken is dat \' en voor een dubbel aanhalingsteken \".

```
>>> print ("Zo\'n mooie dag")
Zo\'n mooie dag
>>> print ("\"Zo\'n mooie dag\"", zegt hij.)
"Zo\'n mooie dag", zegt hij.
>>>
```

3.5. Trucjes met strings

Ik heb een heel interessante vraag voor jou: hoeveel is $10 * 8$? Het antwoord weet je natuurlijk al lang: 80. OK, zo'n interessante vraag was het nu ook weer niet. Maar hoeveel is $10 * 'y'$ (10 keer de letter y, dikwijls ook het karakter y genoemd door programmeurs omdat ze niet enkel met letters werken)? Dit lijkt misschien onzin, maar hier is het antwoord dat Python je zal geven:

```
>>> print (10 * 'y')
yyyyyyyyyy
>>>
```

En dit werkt ook met meer dan één karakter-strings:

```
>>> print (15 * 'xyz')
xyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyzxyz
>>>
```

Een ander trucje dat je kan doen met een string is het inbedden (in het Engels *embed*) van een waarde in een string. Dit kan je gebruiken als je in eenzelfde string steeds een andere

waarde nodig hebt. Je kan dit doen door het %-symbool te gebruiken die de plaats markeert (een plaatshouder) waar je de waarde in de string wil stoppen. Ik denk dat het gemakkelijker is om het met een voorbeeldje uit te leggen:

```
>>> tekst = 'Ik ben %s jaar.'
>>> print (tekst % 12)
Ik ben 12 jaar.
>>>
```

In de eerste regel wordt de variabele `tekst` gemaakt met een string die enkele woorden en de plaatshouder (`%s`) bevat. De `%s` is een merkteken dat tegen de Python-console zegt “verander me in iets anders”. Op de regel die daarna volgt waarbij we het `print`-commando geven gebruiken we ook het `%` symbool om tegen Python te vertellen dat we daar het merkteken met het getal 12 willen hebben.

We kunnen zo’n string ook meerdere keren gebruiken en er verschillende waarden aan geven:

```
>>> tekst = 'Dag %s, ben je klaar om te programmeren?'
>>> naam1 = 'Niel'
>>> naam2 = 'Pepijn'
>>> print (tekst % naam1)
Dag Niel, ben je klaar om te programmeren?
>>> print (tekst % naam2)
Dag Pepijn, ben je klaar om te programmeren?
>>>
```

In het bovenstaande voorbeeldje hebben we 3 variabelen gemaakt (`tekst`, `naam1` en `naam2`). De eerste bevat de string met het merkteken. Op het einde kunnen we de variabele `tekst` printen met terug de `%` operator zodat de variabelen `naam1` en `naam2` gebruikt worden. Je mag gerust meer dan één plaatshouder gebruiken en dat doe je op de volgende manier:

```
>>> tekst = 'Dag %s en %s, zijn jullie klaar om te programmeren?'
>>> print (tekst % (naam1, naam2))
Dag Niel en Pepijn, zijn jullie klaar om te programmeren?
>>>
```

Als je meer dan één merkteken gebruikt, dan moet je de vervangende variabelen tussen haakjes zetten, zodus (`naam1`, `naam2`) is de juiste manier om de twee waarden van de variabelen door te geven. Een lijstje met waarden (die met komma’s gescheiden zijn) omringd door ronde haakjes wordt ook een tuple genoemd. Een tuple is een soort lijstje en daar gaan we het verder nog over hebben.

3.6. De tuple of het lijstje

We gaan boodschappen doen en we willen gebak maken. Wat hebben we nodig uit de winkel? Eieren, bloem, boter en melk. En omdat we een appelcake willen maken hebben we

ook appels nodig. We gaan dus een boodschappenlijstje maken en al deze ingrediënten in een variabele stoppen. Daarom maken we een string:

```
>>> lijstje = 'eieren, bloem, boter, melk, appelsen'
>>> print (lijstje)
eieren, bloem, boter, melk, appelsen
>>>
```

Een andere manier is om een lijst (bijvoorbeeld `lijstje`) te maken, wat iets speciaal is in Python en waarbij we vierkante haakjes gebruiken `[en]`.

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> print (lijstje)
['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>>
```

Je moet dan wel wat meer intikken, maar het is ook beter bruikbaar, let maar even op. We zouden bijvoorbeeld het 4^e item uit de lijst kunnen nemen door zijn positie te gebruiken (we noemen dit met een moeilijker benaming de indexpositie of index) tussen vierkante haakjes:

```
>>> print (lijstje[3])
melk
>>>
```

Raar hé, ik zeg het 4^e item uit de lijst en ik schrijf hierboven `lijstje[3]`.

Merk op dat indexen in de meeste computertalen starten te tellen vanaf nul. Dus ook hier is de indexpositie 0 en staat het eerste item (eieren) uit de lijst op de plaats nul, het tweede item staat op plaats 1, het derde staat op plaats 2 en wat wij zoeken nl. het vierde item (melk) staat op indexpositie 3. Dit is wel een beetje verwarrend want de meeste mensen starten te tellen vanaf 1, maar programmeurs niet, die vinden nul ook belangrijk. En eigenlijk zou iedereen dat belangrijk moeten vinden.

We kunnen ook alle items van bijvoorbeeld het 2^e tot en met het 4^e item in de lijst tonen door een dubbele punt te gebruiken tussen de vierkante haakjes:

```
>>> print (lijstje[1:4])
['bloem', 'boter', 'melk']
>>>
```

Door `[1:4]` te gebruiken zeggen we in feite dat we geïnteresseerd zijn in de items van indexpositie 2 tot (maar niet meegeteld) indexpositie 4. Zoals net gezegd starten we te tellen bij 0. Het 2^e item heeft dan indexpositie 1 en het 4^e item heeft indexpositie 3. Dus `[1:4]` telt vanaf indexpositie 1 tot indexpositie 4 (en niet: tot en met indexpositie 4), wat wel belangrijk is om te onthouden. Ook dit is een regeltje bij programmeurs (om niet teveel in verwarring te komen): men telt tot een indexpositie, niet tot en met een indexpositie. Experimenteer er nu maar even op los. Je kan ook het lijstje zelf veranderen.

Lijsten kunnen gebruikt worden om allerlei soorten items op te slaan. Ze kunnen nummers stockeren:

```
>>> mijnlijst = [5, 10, 20, 68, 14]
>>>
```

En strings:

```
>>> mijnlijst = ['aaa', 'bbbbbb', 'cccc', 'dddddddddd']
>>>
```

Maar ook een menging van getallen en strings:

```
>>> mijnlijst = [5, 10, 'aaa', 'bbbbbb']
>>> print (mijnlijst)
[5, 10, 'aaa', 'bbbbbb']
>>>
```

En zelfs lijsten van lijsten:

```
>>> lijst1 = ['x', 'y', 'z']
>>> lijst2 = [1, 2, 3]
>>> mijnlijst = [lijst1, lijst2]
>>> print (mijnlijst)
[['x', 'y', 'z'], [1, 2, 3]]
>>>
```

In het bovenstaande voorbeeld wordt een variabele genaamd `lijst1` gemaakt met 3 letters, de variabele `lijst2` met 3 cijfers en de lijst `mijnlijst` wordt gemaakt met `lijst1` en `lijst2`. Dit kan allemaal nogal verwarrend overkomen, zeker als je een lijst van een lijst van een lijst van een... gaat maken, maar gelukkig is er meestal geen nood om dingen heel complex te maken in Python. Maar het is wel handig om te weten dat je allerlei soorten items in een Python-lijstje kan stoppen, en niet alleen het boodschappenlijstje.

Items vervangen in lijstjes

We kunnen items in een lijst vervangen door de waarde in te geven op dezelfde wijze als we doen met een normale variabele. Als we het originele lijstje bekijken `['eieren', 'bloem', 'boter', 'melk', 'appelen']` waar we appelcake wilden maken, maar we willen eigenlijk geen appelcake maken maar abrikozencake. Dan moeten we dit item veranderen op indexpositie 4.

```
>>> lijstje[4] = 'abrikozen'
>>> print (lijstje)
['eieren', 'bloem', 'boter', 'melk', 'abrikozen']
>>>
```

Items toevoegen aan het lijstje


We kunnen items aan een lijst toevoegen door een methode te gebruiken die **append** (Engels voor *achteraan toevoegen*) genaamd is. We hebben net een nieuwe term geleerd: methode. Een methode is een actie of een commando dat Python vertelt dat we iets willen

doen. Over methode zullen we het later nog hebben, nu willen we enkel iets toevoegen aan ons lijstje voor de cake. We willen onze abrikozencake toch een beetje lekkerder maken met chocolade. We kunnen volgende doen:

```
>>> lijstje.append('chocolade')
>>> print (lijstje)
['eieren', 'bloem', 'boter', 'melk', 'abrikozen', 'chocolade']
>>>
```

Aan onze lijst `lijstje` voegen we dus via de methode `.append` (bemerkt het puntje voor het woord) een item toe.

Items verwijderen

We kunnen items uit onze lijst verwijderen door het commando `del` (wat een afkorting is van delete, zoals de -toets op je toetsenbord). Mama heeft nog boter gevonden in de kast en dus moeten we dat niet kopen om onze cake te maken. We kunnen boter dus schrappen uit onze lijst. Boter heeft indexpositie 2 in het lijstje (denk eraan dat we starten te tellen vanaf nul):

```
>>> del lijstje[2]
>>> print (lijstje)
['eieren', 'bloem', 'melk', 'abrikozen', 'chocolade']
>>>
```

Je ziet dat het lijstje aangepast is. Maar dat ook de indexposities van alle items die na boter kwamen er eentje vooruit geschoven zijn. Zo heeft chocolade, nu we boter gewist hebben, index 4 gekregen in plaats van indexpositie 5.

2 lijsten zijn beter dan 1

We kunnen lijsten samenvoegen door ze op te tellen, net zoals we 2 getallen optellen:

```
>>> lijst1 = [1, 2, 3]
>>> lijst2 = [4, 5, 6]
>>> print (lijst1 + lijst2)
[1, 2, 3, 4, 5, 6]
>>>
```

Maar we kunnen ook de twee lijsten optellen en de som aan een andere variabele toewijzen:

```
>>> lijst1 = [1, 2, 3]
>>> lijst2 = [4, 5, 6]
>>> lijst3 = lijst1 + lijst2
>>> print (lijst3)
[1, 2, 3, 4, 5, 6]
>>>
```

Je kan ook een lijst vermenigvuldigen, net zoals we een string vermenigvuldigen:

```
>>> lijst1 = [1, 2]
```



```
>>> print (lijst1 * 10)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>>
```

Dit had je waarschijnlijk niet verwacht. In het bovenstaande voorbeeld betekent `lijst1` vermenigvuldigen met 10 feitelijk dat we `lijst1` tien keer willen herhalen. Je begrijpt waarschijnlijk wel dat bij lijsten de operatoren delen en aftrekken niet echt zin hebben. Python weet dat ook en zal je een foutboodschap geven die lijkt op het volgende als je het toch probeert:

```
>>> lijst1 / 5
Traceback (most recent call last):
  File "<pyshell#103>", line 1, in <module>
    lijst1 / 5
TypeError: unsupported operand type(s) for /: 'list' and 'int'
>>>
```

En ook bij het maken van een verschil:

```
>>> lijst1 - 5
Traceback (most recent call last):
  File "<pyshell#104>", line 1, in <module>
    lijst1 - 5
TypeError: unsupported operand type(s) for -: 'list' and 'int'
>>>
```

3.7. Tuples en Lijsten

Een tuple (daarover hebben we het eerder al gehad) is een beetje zoals een lijst, maar in plaats van vierkante haakjes `[en]` gebruiken we ronde haakjes `(en)`. Je kan een tuple gebruiken op een gelijkaardige manier als een lijst:

```
>>> tup = (1, 2, 3)
>>> print (tup[0])
1
>>>
```

Het grote verschil met lijsten is dat een tuple niet kan veranderen eenmaal je deze gemaakt hebt. Je kan dus geen waarde veranderen zoals we eerder deden bij de lijsten. Als je dat probeert zoals bij lijsten dan krijg je een error boodschap:

```
>>> tup[1] = 3
Traceback (most recent call last):
  File "<pyshell#113>", line 1, in <module>
    tup[1] = 3
TypeError: 'tuple' object does not support item assignment
>>>
```

Maar dit betekent niet dat je de variabele zelf, die de tuple bevat, niet zou kunnen veranderen. Je kan bijvoorbeeld als die variabele eerst een tuple was, deze variabele veranderen in een lijst met totaal andere waarden:

```
>>> wat_is_het_nu = (1, 2, 3)
```

```
>>> print (wat_is_het_nu)
(1, 2, 3)
>>> wat_is_het_nu = ['xyz', 'joehoe']
>>> print (wat_is_het_nu)
['xyz', 'joehoe']
>>>
```

Eerst hebben we de variabele `wat_is_het_nu` gemaakt die wijst naar een tuple met 3 getallen. Dan hebben we deze variabele veranderd in een lijst met de strings `'xyz'` en `'joehoe'`.

Dit kan een beetje verwarrend lijken, maar bekijk het eens op een andere manier: stel dat je 2 kastjes hebt om dingen verborgen te houden voor je ouders. Elke kast heeft een naambordje. Je stopt iets in het eerste kastje, sluit het met de sleutel af en gooit de sleutel weg. Dan neem je het naambordje van dit kastje eraf. Je neemt vervolgens het tweede kastje en je stopt daar iets anders in, maar deze keer hou je de sleutel goed bij. Een tuple is een beetje zoals het eerste kastje, je kan niet meer veranderen wat erin zit. Maar je kan het naambordje dat je eraf genomen hebt wel hangen aan een ander kastje. Het tweede kastje is zoals een lijst, je kan er nog steeds dingen instoppen, maar ook uitnemen.

3.8. Een Map zonder papieren

Naast lijstjes en tuples heeft Python nog een derde manier om zaken te ordenen, een map. Het verschil met lijsten en tuples is dat elk item in een map een *sleutel* (in het Engels *key*, een soort referentie) heeft en een bijhorende waarde. Dat klinkt allemaal moeilijk, maar een voorbeeldje zal veel verduidelijken. We kunnen eerst een lijst maken met vriendjes die een hobby uitoefenen en die we aan de variabele `hobby` toekennen:

```
>>> hobby = ['Ruben, Basketbal', 'Marie, Volleybal', 'Ward, Voetbal', 'Heleen, Ballet', 'Lennert, Tennis', 'Senne, BMX', 'Wout, Basketbal']
>>>
```

Als je nu zou willen weten welke de hobby is van Lennert, dan kan je nu nog gemakkelijk in het lijstje zoeken. Maar wat als we er 100 vrienden met hun hobby zouden instoppen of 200? Dan wordt het al heel wat moeilijker om iets terug te vinden. We kunnen deze informatie ook in een map stoppen en de naam als sleutel gebruiken met de hobby als waarde:

```
>>> hobby = {'Ruben' : 'Basketbal', 'Marie' : 'Volleybal', 'Ward' : 'Voetbal', 'Heleen' : 'Ballet', 'Lennert' : 'Tennis', 'Senne' : 'BMX', 'Wout' : 'Basketbal'}
>>>
```

Zie je het verschil reeds met het lijstje? We gebruiken de dubbele punt (:) om elke sleutel van zijn waarde te scheiden. Daarnaast is elke sleutel en elke waarde omgeven door enkele

aanhalingstekens ('). Voor mappen gebruiken we nog een ander soort haakjes, namelijk de accolades { en }, dus geen vierkante [] zoals bij lijstjes of ronde () haakjes bij tuples.

Het bovenstaande resulteert in een map zoals we zien in Tabel 2:

sleutel	waarde
Ruben	Basketbal
Marie	Volleybal
Ward	Voetbal
Heleen	Ballet
Lennert	Tennis
Senne	BMX
Wout	Basketbal

Tabel 2: Map met sleutel en waarde

Indien we willen weten welke de hobby is van Heleen, dan gebruiken we haar naam in de map als key:

```
>>> print(hobby['Heleen'])
Ballet
>>>
```

Let op, we moeten vierkante haakjes gebruiken voor onze sleutel.

We kunnen ook een waarde in onze map verwijderen door de sleutel te deleten met **del**.

Als we Ward willen verwijderen doen we dit als volgt:

```
>>> del hobby['Ward']
>>> print(hobby)
{'Lennert': 'Tennis', 'Marie': 'Volleybal', 'Senne': 'BMX', 'Wout':
'Basketbal', 'Ruben': 'Basketbal', 'Heleen': 'Ballet'}
>>>
```

Als we nadien onze map afdrukken, zien we dat Ward verwijderd is.

We kunnen ook een waarde in de map wijzigen door zijn sleutel te gebruiken. Als je vriend Lennert een andere hobby gekozen heeft:

```
>>> hobby['Lennert'] = 'Atletiek'
>>> print(hobby)
{'Lennert': 'Atletiek', 'Marie': 'Volleybal', 'Senne': 'BMX',
'Wout': 'Basketbal', 'Ruben': 'Basketbal', 'Heleen': 'Ballet'}
>>>
```

Je ziet dat bij Lennert de hobby Tennis vervangen is door Atletiek.

Hopelijk is het duidelijk dat mappen een beetje werken zoals lijstjes en tuples, maar je kan ze niet samenvoegen, want dan krijg je een foutboodschap.

```
>>> hobby = {'Ruben' : 'Basketbal', 'Marie' : 'Volleybal', 'Ward' :
'Voetbal', 'Heleen' : 'Ballet', 'Lennert' : 'Tennis', 'Senne' :
'BMX', 'Wout' : 'Basketbal'}
>>> games = {'Lucas': 'Clash of Clans', 'Siebe': 'Minecraft',
'Thor': 'Supertuxkart'}
```

```
>>> hobby + games
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    hobby + games
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
>>>
```

3.9. Dingen om zelf eens te doen

In dit hoofdstuk hebben we gezien hoe we simpele wiskundige berekeningen moeten maken met de Python-console. We hebben ook gezien hoe haakjes de uitkomst van berekeningen kunnen wijzigen, door de orde van operatoren te veranderen. We hebben geleerd om Python te vertellen hoe waarden moeten onthouden worden voor later gebruik (met variabelen). Daarnaast hebben we ook bekeken hoe Python strings gebruikt om tekst op te slaan, maar we hebben ook geleerd om tuples en lijsten te gebruiken om meer dan 1 item op te sommen.

En nu wat experimentjes:

Experiment 3.1:

Je hebt 5 dozen met 25 snoepjes elk en 7 zakken met 8 chocoladerepen. Hoeveel snoepjes en chocoladerepen heb je in totaal? (Noot: Je kan dit in 1 berekening in de Python-console maken.)

Experiment 3.2:

Maak eens een lijstje met jouw favoriet speelgoed en noem het **speelgoed**. Maak daarnaast ook een lijst van je favoriete dieren en noem dit **dieren**. Dan voeg je deze 2 lijsten samen en je noemt dit resultaat **favorieten**. Op het einde print je deze variabele **favorieten**.

Experiment 3.3:

Maak variabelen voor de voornamen van je 2 beste vrienden. Maak nu een string en gebruik plaatshouders om jouw naam toe te voegen en af te drukken.

4. Schildpadden en andere trage beesten

Er zijn nogal wat gelijkenissen tussen schildpadden in de echte wereld en een Python schildpad. In de echte wereld is een schildpad een soms groen dier dat nogal traag rondkruipt en zijn huisje op de rug draagt. In de Python-wereld is de schildpad gewoon een zwart pijltje dat traag op het scherm beweegt.

De Python schildpad laat een spoor na en dan zou je het eigenlijk moeten vergelijken met een slak. Maar een slak is niet zo aantrekkelijk als een schildpad, natuurlijk. Denk dan gewoon dat de schildpad een pen op de buik draagt en zo het spoor dat ze volgt op het scherm tekent.

Vroeger bestond er een simpele programmeertaal Logo (ze bestaat in feite nog steeds, maar is niet meer zo populair). Met die computertaal werd een robotschildpadje op het computerscherm aangestuurd. Maar de schildpad is nu veranderd in een klein pijltje.

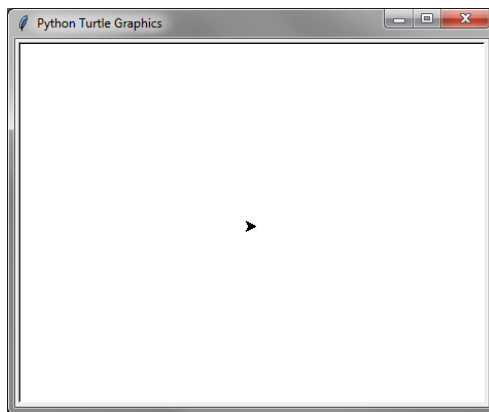
De Schildpad-module van Python (we komen later terug op modules, maar denk nu gewoon over een module als iets dat we in een programma kunnen gebruiken) is een beetje zoals de Logo-programmeertaal. Logo was nogal beperkt, waar Python veel meer mogelijkheden biedt. De module zelf is een goede manier om te leren hoe computers tekeningen maken op je scherm.

We gaan er gewoon mee beginnen en zien hoe het allemaal werkt. De eerste stap is Python vertellen dat we de schildpad willen gebruiken door de module te importeren.

```
>>> import turtle
>>>
```

Daarna hebben we een scherm of een doek (zoals bij een schilder) nodig om op te tekenen (zoals een schildersdoek) en in ons geval is het een leeg scherm:

```
>>> t = turtle.Pen()
>>>
```

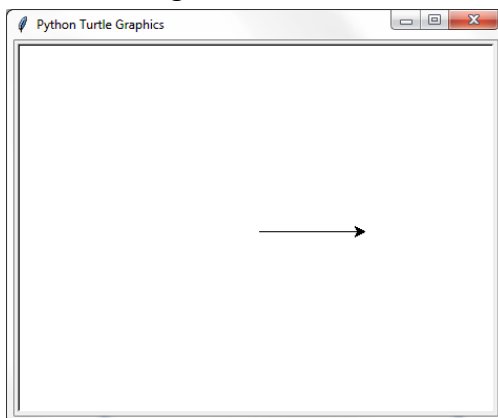


Figuur 5: Het pijltje dat de schildpad voorstelt

In dit stukje code roepen we een speciale functie `Pen()` op van de module `turtle`, die automatisch een venster opent waarin we kunnen tekenen. Een functie is een stukje herbruikbare code (we zullen later op functies terugkomen) dat iets nuttig kan uitvoeren. In ons geval geeft de `Pen()` functie ons een pijltje (een object) dat de schildpad voorstelt. We duiden het object aan met de variabele `t` (eigenlijk geven we het venster de naam `t`).

Ja, dat pijltje is echt onze Python schildpad, en neen, het lijkt niet echt op een schildpad.

Je kan instructies sturen naar de schildpad, door functies van het gemaakte object te gebruiken (door `turtle.Pen()` op te roepen). Omdat we dat object toegekend hebben aan de variabele `t`, gebruiken we `t` om instructies te sturen. Eén schildpad-instructie is voorwaarts (Forward). Dit vertelt de schildpad om voorwaarts te bewegen, afhankelijk van de richting die ze nu uitkijkt. Laat ons de schildpad nu eens 100 pixels (puntjes op je scherm, daarover straks meer) voorwaarts bewegen:



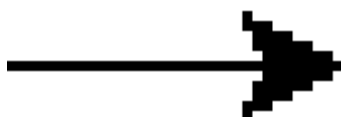
Figuur 6: De schildpad tekent een lijn

```
>>> t.forward(100)
>>>
```

Dan bekom je iets zoals in Figuur 6. De schildpad denkt dat ze 100 passen gezet heeft. Wij zien dat ze 100 pixels bewogen heeft naar rechts.

Ik heb het de hele tijd al over pixels, maar wat is een pixel nu?

Een pixel is niks anders dan een puntje (in het Engels *dot*) op het scherm. Als je heel goed naar je computerscherm kijkt, dan zie je heel kleine vierkante puntjes. Hoe recenter je computerscherm, hoe moeilijker het te zien is. Misschien moet je wel een vergrootglas gebruiken om ze te kunnen zien of met een programmaatje het beeld vergroten (bekijk ook eens Figuur 7). Alle programma's op je computer, maar ook op je gameconsole en je tablet gebruiken heel veel puntjes in allerlei kleuren om iets op je scherm te zetten zoals tekst, maar ook bijvoorbeeld een raceauto, of een figuurtje uit Minecraft of de gebouwen uit Clash of Clans.

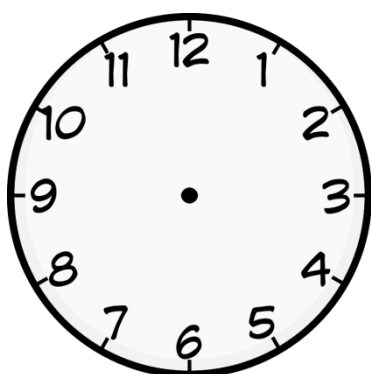


Figuur 7: Pixels in de pijl van de schildpad

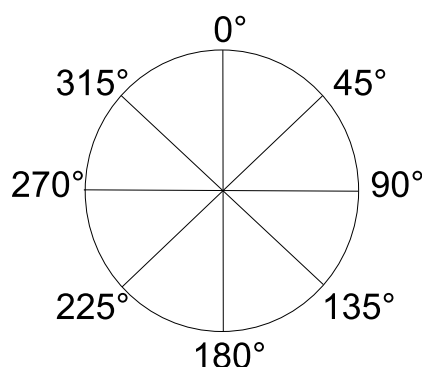
We zullen het verder nog wel hebben over deze pixels in hoofdstuk 10.

We willen de schildpad ook wel eens laten draaien naar links (in het Engels: *left*) of rechts (*right*). Maar daarvoor moet ik je eerst wat uitleggen over hoe de schildpad (en ook een computerprogramma) daarover denkt.

Misschien heb je op school nog niet geleerd over graden ($^{\circ}$), maar een uurwerk ken je zeker wel al. Die heeft 12 onderverdelingen als we enkel de uuraanduidingen tellen. Een cirkel (zoals op het uurwerk) telt 360° , wat je kan zien als je bijvoorbeeld van 12 uur doordraait naar 12 uur. Dan zie je ook dat als je een wijzer van 12 uur naar 3 uur draait, je een hoek van 90° gedraaid bent (kijk ook naar Figuur 9). Als je een wijzer van 12 uur naar 6 uur draait, dan ben je 180° gedraaid. Hoeveel graden zou je draaien als je van 12 uur naar 9 uur gaat? Denk eens goed na. Juist, 270° .



Figuur 8: Onderverdelingen op een uurwerk of cirkel



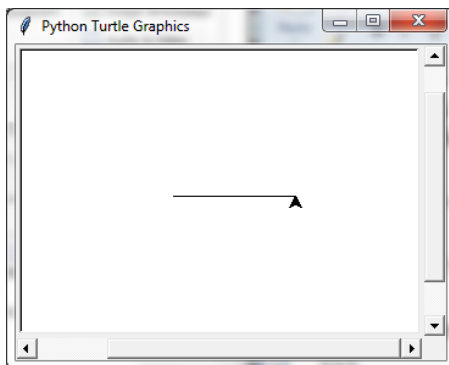
Figuur 9: Graden op een cirkel

We laten de schildpad nu 90° naar links draaien.

```
>>> t.left(90)
>>>
```

Een andere manier om je voor te stellen hoe hoeken en graden werken gaat als volgt. Je staat voor je uit te kijken. Als je je rechterarm naar rechts laat wijzen, dan is de hoek tussen naar waar je kijkt en naar waar je wijst 90° (naar rechts). Hetzelfde kan je doen met je linkerarm, maar dan is het een hoek van 90° naar links. Als je nu je beide armen opzij strekt, dan is de hoek tussen je beide armen 180° .

Wat heeft onze Python schildpad nu gedaan? Ze keek naar voor, stak haar linkerpootje uit naar links en draaide in die richting. Nu ze gedraaid is kijkt ze natuurlijk terug vooruit, zoals je kan zien in Figuur 10.

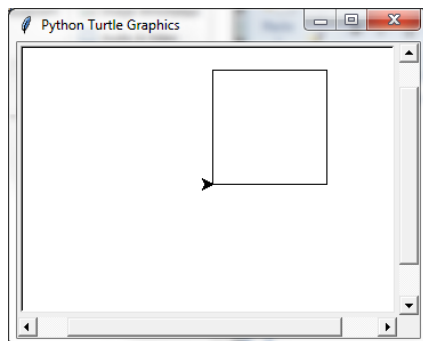


Figuur 10: De schildpad is 90° naar links gedraaid

Als we nu eens dezelfde commando's meerdere keren uitvoeren, wat zouden we dan bekomen? Laat ons eens proberen:

```
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>>
```

We bekomen dan een mooi vierkant en de schildpad staat terug op de plaats waar ze in het begin stond:



Figuur 11: Een vierkant getekend door de schildpad

We kunnen alles wissen door het volgende commando:

```
>>> t.clear()
>>>
```


Zo zijn er nog wel wat basisfuncties die je eens moet uitproberen met de schildpad. Om de schildpad rechts te laten draaien, gebruik je de functie `right()` met het aantal graden; om alles terug te zetten in de originele positie, gebruik je de functie `reset()`. Je kan de schildpad ook omdraaien, door de functie `backward()` te gebruiken en het aantal pixels op te geven. Je kan de schildpad ook zeggen dat ze de pen op haar buik omhoog moet houden zodat ze niet tekent, maar bijvoorbeeld wel zich kan verplaatsen. Dit doe je met de functie `up()`. Om de pen terug op het doek te zetten, gebruik je `down()`. Deze functies gebruik je als volgt.

```
>>> t.reset()
>>> t.right(90)
>>> t.backward(150)
>>> t.up()
>>> t.down()
>>>
```

4.1. Dingen om zelf eens te doen

We hebben net gezien hoe we onze Python schildpad lijntjes en figuren kunnen laten tekenen. De schildpad gebruikt graden om te draaien, en daarvoor moet je soms eens denken aan een uurwerk.

We gaan terug wat experimenteren. Maar sluit eerst de Python-console en start hem daarna terug op.

Experiment 4.1:

Maak een doek met de `Pen()` functie van de schildpad. Teken dan een rechthoek.

Experiment 4.2:

Stel het scherm in de oorspronkelijke staat en teken een driehoek.

5. Hoe een vraag stellen?

Als een programmeur een vraag stelt bij het programmeren dan betekent het dat ie ofwel één ding wil doen ofwel een ander ding. En wat er moet gebeuren is afhankelijk van het antwoord op die vraag. Dit noemen we een “if-statement”, dit is een uitspraak waarbij we beginnen met het woord *Als* (*if* is Engels voor *als*). Een *statement* is Engels voor een *opdracht*.

Het is misschien gemakkelijker uit te leggen met een voorbeeld (we gaan nog niets intypen op de computer):

Hoe oud ben jij? Als je ouder bent dan 18, ben je te oud.

Dit schrijven we in Python met het if-statement:

```
if leeftijd > 18:
    print("Je bent te oud")
```

Hier zien we heel wat nieuwe dingen.

Het if-statement bestaat uit het woordje if en dan komt wat we noemen een voorwaarde (in het Engels *condition*, maar dadelijk meer daarover). Op het einde van deze regel komt een dubbele-punt en daarmee willen we zeggen “doe dan wat volgt”. De volgende regels na de regel met “if” moeten in een blok staan, dat betekent dat deze allemaal 4 spaties naar rechts opgeschoven zijn (zoals met de TAB-toets). De commando’s in dit blok worden volledig uitgevoerd als het antwoord op de vraag “ja” is (of zoals we bij het programmeren zeggen: *True*, wat in het Nederlands *Waar* of *Juist* betekent).

Soms zeggen we tegen het if-statement ook het if-then-statement. *If* betekent *als*, en *then* betekent *dan*. Je leert ook wat Engels bij door te programmeren.

Een voorwaarde of conditie is een verklaring bij het programmeren die *Waar* (*True*) of *Onwaar* (*False*, is *Vals* in het Engels) kan zijn.

Om deze condities te maken hebben we bepaalde symbolen (of zoals we reeds eerder zagen, operatoren) zoals:

operator	betekenis
==	is gelijk aan
!=	is niet gelijk aan
>	is groter dan
<	is kleiner dan
>=	is groter dan of gelijk aan

<code><=</code>	is kleiner dan of gelijk aan
--------------------	------------------------------

Tabel 3: Enkele operatoren bij het if-statement

Een voorbeeldje: als je 12 jaar oud bent zal de conditie `jouw_leeftijd == 12` de waarde `True` geven, maar als je geen 12 jaar oud bent, zal ze de waarde `False` geven.

Wel belangrijk om te onthouden: Verwissel niet de twee “is gelijk aan”-symbolen (`==`) uit een conditie met het “is gelijk aan”-symbool (`=`) waarbij je een waarde toekent, zoals we gezien hebben bij variabelen. Als je een “=”-symbool gebruikt bij een conditie krijg je een foutmelding, je kan dus enkel het “==”-symbool gebruiken.


Als we nu eens de variabele `leeftijd` zetten op jouw leeftijd, en als je 12 jaar oud bent, dan zou de conditie...

```
leeftijd > 10
```

... terug de waarde `True` opleveren, want je bent te oud. Maar als je 8 jaar oud zou zijn, dan zou de waarde die je krijgt `False` zijn. Stel dat je 10 jaar oud zou zijn, dan zou de waarde die je terug krijgt ook `False` zijn, want de conditie kijkt naar “is groter dan” (`>`) 10, en niet naar “is groter dan of gelijk aan” (`>=`) 10.

Het zal wat duidelijker worden als we enkele voorbeeldjes geven in de Python-console die we nu terug kunnen openen:

```
>>> leeftijd = 12
>>> if leeftijd > 12:
        print ('Zo oud ben ik')
>>>
```

Wat gebeurt er als je bovenstaande ingetypt hebt (en twee keer op de -toets gedrukt)? Niks. Gewoon niks, niemandal.

Weet je ook waarom? Omdat de waarde van de variabele `leeftijd` niet groter is dan 12 zal het `print` commando in het blok dat op de conditie volgt niet uitgevoerd worden. Als je het bovenstaande voorbeeld nu aanpast (let goed op dat ene kleine detail):

```
>>> leeftijd = 12
>>> if leeftijd >= 12:
        print ('Zo oud ben ik')

Zo oud ben ik
>>>
```

Nu krijg je wel je leeftijd. Als je het onderstaande voorbeeld uitprobeert, bekom je dan hetzelfde? Probeer eens:

```
>>> leeftijd = 12
>>> if leeftijd == 12:
```

```
print ('Zo oud ben ik')

Zo oud ben ik
>>>
```

Natuurlijk bekom je hetzelfde. Goed gedaan.

5.1. Doe dit ... of ANDERS !!!

Oei, wie is er boos, wie is er aan het kijven? Niemand natuurlijk.



Anders is in het Engels *else*.

We kunnen het if-statement of de if-opdracht nog verder uitbreiden zodat het ook iets doet wanneer een conditie onwaar is. Zo kan bijvoorbeeld het woord 'Juist' gedrukt worden in de console als jouw leeftijd 12 jaar is, en het de waarde 'Fout' print indien dit niet zo is.

Daarvoor gebruiken we het if-then-else-statement, wat niks anders is dan zeggen: **als** iets waar is, doe **dan** dit, **anders** doe dat:

```
>>> leeftijd = 12
>>> if leeftijd == 12:
    print ('Juist')
else:
    print ('Fout')

Juist
>>>
```

Let hier op: Het **else**-gedeelte mag niet in het blok staan dat volgt als de conditie waar is. Je komt uit het blok door op de -toets te drukken (dat is meestal diegene boven de -toets). Het moet op dezelfde hoogte komen als het **if**-gedeelte van het statement. Ook na **else** moet een dubbele-punt komen wat ook hier betekent "doe dan wat volgt". Dan wordt het blok (met 4 spaties ervoor) uitgevoerd. Als we de waarde van **leeftijd** nu veranderen naar een ander getal dan krijgen we een ander antwoord:

```
>>> leeftijd = 4
>>> if leeftijd == 12:
    print ('Juist')
else:
    print ('Fout')

Fout
>>>
```

5.2. Doe dit... of doe dit... of doe dit... of ANDERS!!!

Wie is er nu alweer kwaad? Oef, gelukkig ook nu niemand.

We kunnen het if-then-else-statement nog verder uitbreiden. Mensenlief, stopt dat uitbreiden dan nooit? Wel, dat hangt enkel van jou af. We kunnen nog verder uitbreiden

door gebruik te maken van `elif` (en dat is een verkorting voor else-if, “anders als”). Het is alweer het gemakkelijkst om dit uit te leggen met een voorbeeld. We kunnen eens bekijken of jouw leeftijd 9 of 10 of 11 jaar is of nog meer:

```
1. >>> leeftijd = 11
2. >>> if leeftijd == 9:
3.     print ('Je bent 9 jaar oud')
4. elif leeftijd == 10:
5.     print ('Je bent 10 jaar oud')
6. elif leeftijd == 11:
7.     print ('Je bent 11 jaar oud')
8. elif leeftijd == 12:
9.     print ('Je bent 12 jaar oud')
10. else:
11.     print ('Je hebt zeker een andere leeftijd?')
12.
13. Je bent 11 jaar oud
14. >>>
```

Om gemakkelijker naar de bovenstaande code te kijken zijn er nummertjes voor elke regel geplaatst. Die staan niet in de Python-console.

Regel 2 kijkt of de waarde van de variabele `leeftijd` gelijk is aan 9. Dit is onwaar, dus wordt direct naar regel 4 gesprongen. Daar wordt gekeken of `leeftijd` gelijk is aan 10. Dit is alweer onwaar. Dan wordt direct naar regel 6 gesprongen waar de variabele gelijk is aan 11. Dit is waar, dus gaat Python verder naar regel 7 en voert het `print`-commando uit. Heb je gezien dat er in dit programma 5 blokken (je mag ze ook groepen noemen) zijn? Het zijn regels 3, 5, 7, 9 en 11. De regels 8 tot en met 11 worden niet meer bekeken wanneer de leeftijd 11 jaar is.

5.3. Condities samenvoegen

Je kan ook condities samenvoegen of combineren door een aantal *trefwoorden* (in het Engels *keywords*) te gebruiken zoals `and` (*en* in het Nederlands) en `or` (*of* in het Nederlands). We kunnen het voorbeeld van daarnet wat korter maken door `or` te gebruiken en de condities samen te voegen (kijk ook nog even terug naar hoofdstuk 3.5):

```
1. >>> if leeftijd == 9 or leeftijd == 10 or leeftijd == 11 or
   leeftijd == 12:
2.     print ('Je bent %s' % leeftijd)
3. else:
4.     print('Je hebt zeker een andere leeftijd')
```

Wanneer één van de condities in regel 1 True is (dit betekent dat de leeftijd 9, 10, 11 of 12 is) dan wordt het blok van regel 2 uitgevoerd. Als het False is, wat regel 3 aangeeft, dan wordt regel 4 uitgevoerd door Python.

We kunnen het voorbeeld nog wat korter maken door enkele van de de operatoren die we in Tabel 3 gezien hebben te gebruiken met `and`:

```
1. >>> if leeftijd >= 9 and leeftijd <= 12:
2.     print ('Je bent %s' % leeftijd)
```

```
3. else:
4.     print( 'Je hebt zeker een andere leeftijd')
```

Hier zie je dat als allebei de condities in regel 1 waar zijn, het blok in regel 2 uitgevoerd wordt (als de leeftijd groter dan of gelijk aan 9 is en de leeftijd is kleiner dan of gelijk aan 12). Experimenteer maar eens: je kan de variabele `leeftijd` ook eens veranderen.

5.4. De grote leegte

Er is nog een speciale waarde die aan een variabele kan gegeven worden. We hebben het er niet over gehad in het vorige hoofdstuk. Het is wel een rare waarde, want het is **niets**, zelfs niks.

Net zoals getallen, tekst, lijsten, enz waarden zijn die kunnen gegeven worden aan variabelen, is **niets** ook een soort waarde die eraan gegeven kan worden. In Python heet een lege waarde (zoals **niets**) gewoon **None** (in andere programmeertalen heet dit soms “Null”) en je kan dit gewoon zoals andere waarden gebruiken:

```
>>> waarde = None
>>> print (waarde)
None
>>>
```

None is een manier om een variabele terug te zetten naar de niet-gebruikte toestand, zoals ze was voor je er een andere waarde aan gaf. Het is ook een manier om een variabele te maken zonder er een waarde aan te geven vooraleer de variabele gebruikt wordt.

Alweer is het gemakkelijker om het met een voorbeeld uit te leggen:

Stel, je speelt in een volleybalploeg en je moet geld inzamelen voor nieuwe spelerskledij. Als je alle geld inzamelt zou je willen wachten tot iedereen het geld afgegeven had, alvorens beginnen te tellen. Als we dat in programmeer-taal zouden zeggen, dan willen we een variabele voor elke speler van het team die geld inzamelt en zetten we alle variabelen op **None**. In Python wordt dit dan:

```
>>> speler1 = None
>>> speler2 = None
>>> speler3 = None
>>>
```

We kunnen een if-statement gebruiken om deze variabelen te controleren, zodat we zien of alle spelers van het team reeds geld afgegeven hebben dat ze ingezameld hadden:

```
>>> if speler1 == None or speler2 == None or speler3 == None:
    print ('Wacht nog even tot alle spelers terug zijn')
else:
    print ('Je hebt €%s' % (speler1 + speler2 + speler3))
```

Het if-statement kijkt of één van de variabelen de waarde **None** heeft en print de eerste boodschap als dit waar is. Als elke variabele een echte waarde heeft, dan wordt de tweede boodschap met het opgehaalde bedrag gedrukt op het scherm. Als je bovenstaande uitprobeert, dan bekom je

```
Wacht nog even tot alle spelers terug zijn
```

Zelfs al zouden we 1 of 2 variabelen met een getal hebben, dan zouden we nog dezelfde boodschap krijgen:

```
>>> speler1 = 50
>>> speler2 = 80
>>> speler3 = None
>>> if speler1 == None or speler2 == None or speler3 == None:
    print ('Wacht nog even tot alle spelers terug zijn')
else:
    print ('Je hebt €%s' % (speler1 + speler2 + speler3))

Wacht nog even tot alle spelers terug zijn
>>>
```

Enkel als we alle variabelen een waarde geven zie je de boodschap uit het tweede blok:

```
>>> speler1 = 50
>>> speler2 = 80
>>> speler3 = 90
>>> if speler1 == None or speler2 == None or speler3 == None:
    print ('Wacht nog even tot alle spelers terug zijn')
else:
    print ('Je hebt €%s' % (speler1 + speler2 + speler3))

Je hebt €220
>>>
```

5.5. Wat is het verschil?

Wat is het verschil tussen 9 en '9'? Behalve de aanhalingstekens, niet veel, zou je kunnen denken. Als je jezelf nog de vorige hoofdstukken herinnert, dan weet je dat het eerste cijfer een getal is en het tweede een string. En dit doet ze meer verschillen van elkaar dan je zou denken. Eerder hebben we de waarde van een variabele (**leeftijd**) vergeleken met een getal in een if-statement:

```
>>> if leeftijd == 9:
    print ('Je bent 9 jaar oud')
>>>
```

Als je de variabele **leeftijd** gelijk aan 9 zet, dan zal het **print** commando uitgevoerd worden:

```
>>> leeftijd = 9
>>> if leeftijd == 9:
    print ('Je bent 9 jaar oud')

Je bent 9 jaar oud
```

```
>>>
```

Maar als je `leeftijd` stelt op '9' (met de aanhalingstekens), dan wordt het `print`-commando niet uitgevoerd:

```
>>> leeftijd = '9'
>>> if leeftijd == 9:
>>>     print ('Je bent 9 jaar oud')
>>>
```

Waarom wordt de code in het blok nu niet uitgevoerd? De reden is dat een string geen getal is, zelfs al lijken ze op elkaar. Kijk maar:

```
>>> leeftijd1 = 9
>>> leeftijd2 = '9'
>>> print (leeftijd1)
9
>>> print (leeftijd2)
9
>>>
```

Zie je, ze lijken exact op elkaar, maar de waarden zijn verschillend omdat de eerste een cijfer is en de tweede een string. Daarom kan `leeftijd == 9` nooit waar zijn als de waarde van de variabele een string is.

Omdat dit een beetje moeilijk lijkt, kunnen we het aan de hand van een ander voorbeeldje uitleggen. Stel dat je 9 snoepjes hebt en 9 potloden. Het aantal van beide zaken is hetzelfde, maar je kan niet zeggen dat snoepjes hetzelfde zijn als potloden. Potloden smaken echt niet lekker. Maar gelukkig hebben programmeertalen zoals Python hiervoor een magische oplossing bedacht. Dat zijn functies die strings in getallen kunnen veranderen en getallen in strings. Programmeertalen zijn jammer genoeg nog niet zo slim om potloden in snoepjes te veranderen. Die magische functie om een string in een cijfer te veranderen heet `int`. Een voorbeeldje:

```
>>> leeftijd = '9'
>>> veranderde_leeftijd = int(leeftijd)
>>>
```

De variabele `veranderde_leeftijd` heeft nu het getal 9 en geen string. Om het omgekeerde te doen, namelijk een getal in een string veranderen gebruik je de functie `str()`:

```
>>> leeftijd = 9
>>> veranderde_leeftijd = str(leeftijd)
>>>
```

De variabele `veranderde_leeftijd` bevat nu de string 9 en geen getal. Eventjes terug naar het if-statement waarbij het `print` commando niet wordt uitgevoerd. Als we eerst de variabele met de string veranderen naar een variabele met een getal dan wordt het `print`-commando wel uitgevoerd:

```
>>> leeftijd = '9'
```



```
>>> veranderde_leeftijd = int(leeftijd)
>>> if veranderde_leeftijd == 9:
    print ('Je bent 9 jaar oud')

Je bent 9 jaar oud
>>>
```

Tijd om zelf wat te experimenteren!!

Experiment 5.1:

Waarom wordt de laatste lijn ook geprint?

```
>>> leeftijd = 9
>>> if leeftijd == 8:
    print (Deze lijn niet')
else:
    print (Deze lijn wel')
print (Deze lijn ook')
>>>
```

6. Opnieuw en opnieuw en opnieuw

Vind je het ook niet vervelend als je telkens dingen moet herhalen? Opnieuw en opnieuw en opnieuw. Soms zijn er oplossingen voor, zoals in programmeertalen. Programmeurs zijn mensen die niet graag dingen opnieuw moeten herhalen. Dat vinden ook zij vervelend.

Daarom hebben ze iets uitgevonden dat een **for-loop** heet (dat is Engels voor iets zoals een *voor-lus*, maar dat is eigenlijk niet zo'n goede vertaling). Als voorbeeldje willen we dat Python 6 keer `Hier is Python` drukt op het scherm:

```
>>> print ('Hier is python')
Hier is python
>>> print ('Hier is python')
Hier is python
>>> print ('Hier is python')
Hier is python
>>> print ('Hier is python')
Hier is python
>>> print ('Hier is python')
Hier is python
>>>
```

Pfff, nogal wat werk om alles in te typen, nietwaar. Maar we kunnen ook een for-loop gebruiken:

```
>>> for x in range(0, 6):
    print ('Hier is python')

Hier is python
Hier is python
Hier is python
Hier is python
Hier is python
Hier is python
>>>
```

Opgelet: ook bij een for-loop moet je met een blok met 4 spaties werken op de regel die volgt na de dubbele-punt. Anders kan je een foutboodschap krijgen die lijkt op:

```
SyntaxError: expected an indented block

>>>
```

Een beetje uitleg bij het bovenstaande voorbeeld. De functie `range()` is een snelle en gemakkelijke manier om een lijst nummers tussen een startgetal en eindgetal te maken. Bijvoorbeeld `range(10, 15)` zou volgende lijst getallen geven: (10, 11, 12, 13, 14).

Denk er in ons voorbeeld aan dat (zoals reeds gezegd) we bij het programmeren beginnen te tellen vanaf nul en dat het laatste getal in een lijst niet meegeteld wordt. Dus in het geval van onze for-loop vertelt Python bij de code “`for x in range(0, 6)`” dat een lijst met getallen (0, 1, 2, 3, 4, 5) gemaakt moet worden en dat elk getal in de variabele `x` gestopt moet worden waarna het blok dat volgt gedrukt moet worden tot de lijst volledig doorlopen is. Om dit wat duidelijker te maken gaan we de waarde van `x` in ons voorbeeld er eens bij printen.

```
>>> for x in range(0, 6):
    print('Hier is python %s' % x)

Hier is python 0
Hier is python 1
Hier is python 2
Hier is python 3
Hier is python 4
Hier is python 5
>>>
```

Als we dit zonder for-loop zouden moeten uitvoeren, dan zouden we alweer heel wat werk hebben om het in te voeren in onze console:

```
>>> x = 0
>>> print('Hier is python %s' % x)
Hier is python 0
>>> x = 1
>>> print('Hier is python %s' % x)
Hier is python 1
>>> x = 2
>>> print('Hier is python %s' % x)
Hier is python 2
>>> x = 3
>>> print('Hier is python %s' % x)
Hier is python 3
>>> x = 4
>>> print('Hier is python %s' % x)
Hier is python 4
>>> x = 5
>>> print('Hier is python %s' % x)
Hier is python 5
>>>
```

De for-loop heeft ons 10 lijnen code uitgespaard: 12 lijnen programmeren in het laatste voorbeeld min 2 lijnen in het voorbeeld met de for-loop. Dit is zeer belangrijk want een goede programmeur is luier dan een leeuw die net gegeten heeft als het 40°C is in de savanne. Goede programmeurs vinden het niet fijn het om dingen meer dan eens te moeten doen, zodus is de for-loop een zeer goede manier om herhaaldelijk opdrachten te geven in een programeertaal.

Weet je nog dat we eerder een boodschappenlijstje gemaakt hebben om cake te maken? Daar hebben we een lijst gebruikt. We moeten ons niet beperken bij de for-loop tot de functie `range()`, want je kan ook een lijst gebruiken:

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> for x in lijstje:
    print (x)

eieren
bloem
boter
melk
appelen
>>>
```

Bovenstaande code is een manier om te zeggen dat voor elk item uit het boodschappenlijstje de waarde in de variabele `x` gestopt moet worden en dat dan de inhoud van de variabele moet afgedrukt worden op het scherm. Als we de lus niet gebruiken, moeten we het volgende moeten intypen:



```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> print (lijstje[0])
eieren
>>> print (lijstje[1])
bloem
>>> print (lijstje[2])
boter
>>> print (lijstje[3])
melk
>>> print (lijstje[4])
appelen
```

Gelukkig willen we goede programmeurs zijn...

6.1. Wanneer is een blok geen rechthoek?

Als het een blok code is. Maar wat is een blok code dan? Een blok code is een aantal programma-opdrachten die je samen wil zetten. Bijvoorbeeld bij de for-loop in het voorbeeld hiervoor, zou je misschien meer willen afdrukken dan enkel de items uit het boodschappenlijstje. Je zou op voorhand reeds de prijzen opgezocht kunnen hebben en die erbij willen zetten. Stel dat er een functie zou bestaan die `prijs()` heet. Deze functie bestaat natuurlijk niet maar het is enkel maar om een gemakkelijk voorbeeldje te maken. Wat nu volgt moet je niet in de Python-console ingeven, want omdat die functie niet bestaat zal je een foutmelding krijgen. We zouden ons programma dan als volgt kunnen schrijven (het staat niet in een zwart kadertje, dus ik herhaal: niet ingeven):

```
>>> for x in lijstje:
    prijs (x)
    print (x)
```

We hebben dus 2 opdrachten in het Python-blok: **prijs (x)** en **print (x)**. Let op want in Python zijn lege ruimtes zoals tabulatie (als je op deze -toets drukt) of spaties (als je op de lange spatietoets, deze -toets, dus) drukt heel belangrijk. Alle code die op dezelfde plaats staat (dus evenveel lege ruimte telt) behoort tot hetzelfde blok code. Een voorbeeldje maakt het je wat duidelijker:

```
print (x)
    Dit is blok1
    Dit is blok1
        Dit is blok2
        Dit is blok2
        Dit is blok2
        Dit is blok2
    Dit is terug blok1
    Dit is terug blok1
        Maar dit is blok3
        Maar dit is blok3
        Maar dit is blok3
            En dit is blok4
            En dit is blok4
```

Het is belangrijk altijd dezelfde hoeveelheid lege ruimte te nemen. Standaard neemt Python (als je in IDLE werkt) 4 spaties per blok. Als je niet steeds dezelfde hoeveelheid lege ruimte voor je programmaregel neemt zoals in volgend voorbeeldje:

```
for x in lijstje:
    prijs (x)
    print (x)
```

De tweede regel (**prijs (x)**) begint met 4 spaties. De derde regel (**print (x)**) begint met 6 spaties. In een boek zijn het aantal spaties moeilijk te tellen. Daarom zal ik even een ander karakter nemen in plaats van een spatie (zodat je het gemakkelijker kan zien), maar gebruik dit niet in Python, want Python kent enkel spaties vóór functies. Ik neem het karakter “.” als teken voor een spatie:

```
for x in lijstje:
    ...prijs (x)
    .....print (x)
```

Dit zou een foutmelding geven, nl. **IndentationError: unexpected indent**.

Als je eenmaal start met 4 spaties, dan moet je er ook mee verder gaan. Als je een blok in een blok wil plaatsen zoals hoger dan heb je 8 spaties (2 keer 4 spaties) nodig om de regels van dat tweede blok te beginnen. Terug naar ons voorbeeld van eerder:


```
.....Dit is blok1
```

```
····Dit is blok1
```

En het tweede blok dat zich “binnen in” het eerste bevindt:

```
····Dit is blok1
····Dit is blok1
········Dit is blok2
········Dit is blok2
```


Waarom zouden we een blok “binnen in” een ander blok willen stoppen, zoals we net gezien hebben? Normaal doen we dat als het tweede blok afhankelijk is van het eerste. Een beetje zoals een for-loop. Als de regel met de for-loop het eerste blok is, dan staan de opdrachten die we alsmaar willen uitvoeren in het tweede blok. Zodus, de uitvoering van deze opdrachten is afhankelijk van wat in het eerste blok gebeurt. Dat lijkt moeilijk uitgelegd, maar zo meteen wordt het wel duidelijker.

Als je in de Python-console werkt en je start te coderen in een blok, dan gaat Python automatisch verder in dat blok tot je op de -toets drukt als je op een lege lijn staat.

We gaan nu eens met een echt voorbeeld werken zodat je ziet hoe het werkt, maar open eerst de Python-console (denk eraan om 4 spaties te houden voor je blokken als het niet automatisch gebeurt door bijvoorbeeld IDLE3):

```
>>> lijstje = ['x', 'y', 'z']
>>> for a in lijstje:
    print (a)
    print (a)

x
x
y
y
z
z
>>>
```

Nadat je de tweede keer **print (a)** ingegeven hebt, druk je bij de lege regel op de -toets. Daarmee vertel je de console dat je dit blok wil stoppen. Het voorbeeld print elke letter uit de lijst tweemaal.

In het volgende voorbeeld zal een fout optreden, weet je ook waarom?

```
>>> lijstje = ['x', 'y', 'z']
>>> for a in lijstje:
    print (a)
    print (a)

SyntaxError: unexpected indent
```

```
>>>
```

De tweede regel die `print (a)` bevat heeft 6 spaties en geen 4. Python wil steeds hetzelfde aantal spaties hebben of een veelvoud.

Te onhouden:

Als je voor een blok code start met 4 spaties, moet je verdergaan met 4 spaties. Als je een blok code met 2 spaties start, dan ga je verder met 2 spaties. De meeste programmeurs gebruiken 4 spaties in Python en dat staat meestal ook zo ingesteld.

We gaan een iets moeilijker voorbeeld geven:

```
>>> lijstje = ['x', 'y', 'z']
>>> for a in lijstje:
    print (a)
    for b in lijstje:
        print (b)
```

Welke zijn de verschillende blokken en wat gaan ze doen?

Zoals je kan zien zijn er twee blokken in het bovenstaande voorbeeldje. Het eerste blok is het deel van de eerste for-loop en blok 2 is de regel van de tweede for-loop:

```
>>> lijstje = ['x', 'y', 'z']
>>> for a in lijstje:
    print (a)                #
    for b in lijstje:        # deze 3 lijnen zijn het 1e blok
        print (b)           #
```

```
>>> lijstje = ['x', 'y', 'z']
>>> for a in lijstje:
    print (a)
    for b in lijstje:
        print (b)           # deze lijn is het 2e blok
```

Heb je een idee je wat dit beetje code gaat uitvoeren?

Het gaat de 3 letters van `lijstje` afdrukken op het scherm, maar weet je ook hoeveel keer? Als we naar elke regel kijken dan kunnen we het aantal misschien wel reeds raden. We weten dat de opdracht in de eerste lus (de eerste for-loop) doorheen elk item uit het lijstje gaat en daarna de opdrachten in het 1^e blok uitvoert. Er zal dus een letter afgedrukt worden en dan start de tweede lus. Ook deze lus gaat door alle letters uit het lijstje en voert dan de opdracht in het 2^e blok uit. Wat we dus gaan zien als we de code starten, is dat eerst 'x' afgedrukt wordt door de eerste lus, gevolgd door 'x', 'y' en 'z' uit de 2^e lus. Daarna volgt 'y' uit de eerste lus alweer gevolgd door 'x', 'y' en 'z' uit de 2^e lus. Geef dit programma maar eens in de Python-console in:

```
>>> lijstje = ['x', 'y', 'z']
>>> for a in lijstje:
    print (a)
    for b in lijstje:
```

```

print (b)

x
x
y
z
y
x
y
z
z
x
y
z
>>>

```

Letters afdrukken is niet echt zinvol, denk je ook niet? Als we nu eens terug gaan naar een voorbeeld dat we eerder zagen in dit boek? Weet je nog dat we een berekening gemaakt hadden hoeveel geld je nog zou overhebben na een jaar (52 weken) als je elke week €5 zakgeld kreeg, je klusjes voor oma opknapte en daarvoor €7 kreeg, maar je ook om €8 snoepjes kocht?

Het was iets als:

```

>>> (5 + 7 - 8) * 52
>>>

```

Het is misschien interessanter om eens te bekijken hoe je spaargeld stijgt gedurende het jaar, dan om te weten hoeveel het is na dat jaar. Dat kunnen we met een andere for-loop uitvoeren. Maar eerst moeten we terug cijfers geven aan de variabelen:

```

>>> zakgeld = 5
>>> klusjes_oma = 7
>>> snoepjes = 8
>>>

```

Als we terug de originele berekening maken met deze variabelen, dan bekommen we:

```

>>> (zakgeld + klusjes_oma - snoepjes) * 52
208
>>>

```

Of we kunnen zien wat we na een jaar gespaard hebben met een nieuwe variabele **gespaard** en dan een lus:

```

>>> gespaard = 0
>>> for week in range (1, 53):
    gespaard = gespaard + zakgeld + klusjes_oma - snoepjes
    print ('Week %s = €%s in de spaarpot' % (week, gespaard))
>>>

```

Bij de eerste regel wordt de variabele **gespaard** op nul gezet. We noemen dit ook initialiseren, wat betekent dat we de variabele een beginwaarde geven (anders weet Python niet met welke waarde het programma voor deze variabele moet starten).

In regel 2 zetten we de lus op met de for-loop die het blok met opdrachten uit regels 3 en 4 uitvoert. Telkens de lus doorlopen wordt, zal de variabele **week** met het volgende cijfer geladen worden (startend bij 1 tot en met 52, weet je nog?).

Regel 3 is iets moeilijker. Wat we willen doen is elke week het nieuwe bedrag optellen bij het totaalbedrag dat we reeds hebben gespaard tot die week. Per week komt er dus **(zakgeld + klusjes_oma - snoepjes)** bij datgene dat we reeds gespaard hebben. De eerste week hadden we nog niks gespaard, dus was **gespaard** eerst gelijk aan nul. We geven dan aan dat **gespaard** moet worden **0 + (zakgeld + klusjes_oma - snoepjes)**. Dat wordt dan **0 + €4** en dus is de variabele **gespaard** dan gelijk aan 4. Op het einde van week 2 hadden we reeds €4 gespaard en daar komt terug **(zakgeld + klusjes_oma - snoepjes)** bij. Dus hebben we €4 + terug €4 wat dan op het einde van de tweede week €8 geeft.

Als we dit in computertermen zouden zeggen, dan zou het als volgt klinken: “verander de inhoud van de variabele **gespaard** door wat ik tot nu gespaard heb plus wat ik deze week verdiend heb.”

Eigenlijk is het “=”-symbool nogal een bazig symbool. Het zegt: “doe iets met wat aan mijn rechterkant staat en bewaar het dan voor later in de naam die aan mijn linkerkant staat.”

Regel 4 is een beetje een ingewikkelde print-opdracht, maar we hebben deze in een eerder hoofdstuk (3.5 Trucjes met strings) al gezien.

Als je het programma laat lopen bekom je iets zoals hieronder:

```
Week 1 = €4 in de spaarpot
Week 2 = €8 in de spaarpot
Week 3 = €12 in de spaarpot
Week 4 = €16 in de spaarpot
Week 5 = €20 in de spaarpot
Week 6 = €24 in de spaarpot
Week 7 = €28 in de spaarpot
Week 8 = €32 in de spaarpot
Week 9 = €36 in de spaarpot
Week 10 = €40 in de spaarpot
Week 11 = €44 in de spaarpot
Week 12 = €48 in de spaarpot
Week 13 = €52 in de spaarpot
Week 14 = €56 in de spaarpot
Week 15 = €60 in de spaarpot
Week 16 = €64 in de spaarpot
>>>
```

... en zo gaat het verder tot en met Week 52.

6.2. Nu we toch lussen maken

Een for-loop is niet de enige soort lus die je in Python kan maken. Er is ook de while-loop (while betekent in het Nederlands “zolang”). Bij een for-loop weet je precies wanneer deze zal stoppen, bij een while-loop weet je dat niet altijd op voorhand.

Een voorbeeldje maakt het iets duidelijker: Stel je een trap met 15 treden voor. Je weet op voorhand dat er 15 treden zijn, je kan deze gemakkelijk op voorhand tellen. Dat is zoals een for-loop:

```
>>> for trede in range (0, 15):
    print (trede)
```

Stel je nu een trap voor die een zeer hoge berg opgaat en dus zeer veel treden heeft. Misschien ben je al moe voordat je boven bent of misschien kom je in een sneeuwstorm terecht waardoor je moet stoppen. Dat is een while-loop:

```
>>> trede = 0
>>> while trede < 3000:
    print (trede)
    if moe == True:
        break
    elif sneeuwstorm == True:
        break
    else:
        trede = trede + 1
```

Je moet de code niet in de Python-console ingeven, want we hebben geen variabelen **moe** en **sneeuwstorm** gemaakt. Met dit voorbeeld tonen we alleen de basis aan van de while-loop. Zolang (while) de waarde van variabele **trede** minder dan 3000 is (**while trede < 3000**) wordt de code in het blok uitgevoerd. In het blok drukken we de waarde van **trede** af en kijken dan of **moe** of **sneeuwstorm** waar (True) is. Als **moe** waar is dan stopt de **break** - opdracht (**break** betekent *onderbreek*) de code die uitgevoerd wordt in de lus. Eigenlijk springen we uit de lus naar de regel die onmiddellijk volgt na het blok (maar daar staat niets meer, dus stopt het programma). Als **sneeuwstorm** waar is dan stopt ook hier de **break** – opdracht de code in de lus. Als zowel **moe** als **sneeuwstorm** onwaar is dan wordt bij de variabele **trede** 1 opgeteld en gaan we terug naar het begin van de lus waar gekeken wordt naar de voorwaarde of **trede < 3000**.

De stappen in een while-loop:

1. Bekijk of de voorwaarde nog voldoet
2. Voer de code in het blok uit
3. Herhaal

Dikwijls wordt in een while-loop meer dan 1 voorwaarde gebruikt:

```
>>> x = 15
```

```
>>> y = 30
>>> while x < 20 and y < 50:
    x = x + 1
    y = y + 1
    print(x, y)
>>>
```

In deze while-loop maken we een variabele **x** aan met de waarde 15 en een variabele **y** met de waarde 30. Er worden 2 voorwaarden gechecked in de lus: is **x < 20** en is **y < 50**. Zolang de beide voorwaarden waar (True) zijn, wordt het blok code dat erop volgt uitgevoerd, waarbij bij allebei de variabelen 1 opgeteld wordt en ze allebei geprint worden. Na het uitvoeren van het programma is de output:

```
16 31
17 32
18 33
19 34
20 35
>>>
```

Weet je ook waarom deze getallen afgedrukt worden? Probeer het eens te uit te werken. Als je je eigen oplossing wil controleren dan kan je dat hier: met **x** starten we te tellen bij 15 en met **y** bij 30. Dan tellen we 1 op bij allebei de variabelen elke keer we de lus doorlopen en dan printen we de beide variabelen af. De voorwaarden worden gecontroleerd waarbij **x < 20** moet zijn en ook **y < 50** moet zijn. Nadat de lus 5 keer doorlopen is (en dus telkens 1 optelt bij de 2 variabelen) bereikt **x** de waarde 20. Daarna is de eerste voorwaarde dat **x < 20** moet zijn niet langer meer waar en stopt Python dus met de lus.

Een andere mogelijkheid om een while-loop te gebruiken is de zogenaamde bijna-eeuwige lus. Dit is een lus die eeuwig (zolang je computer blijft werken natuurlijk) blijft draaien, of stopt omdat er iets in de code gebeurt dat de lus onderbreekt. Een voorbeeldje zonder in de console te gaan:

```
>>> while True:
    veel code hier
    veel code hier
    veel code hier
    if een_voorwaarde == True:
        break
```

De voorwaarde voor de while-loop is waar (True). Daarom zal de code **veel code hier** in het blok altijd blijven lopen (een oneindige of een eeuwige lus). Alleen als de variabele **een_voorwaarde** waar is, zal de code in de lus onderbroken worden. In het volgende hoofdstuk gaan we hier wat verder op in.

6.3. Dingen om zelf eens te doen

In het afgelopen hoofdstuk zagen we hoe we lussen gebruiken om dingen die herhaald moeten worden te programmeren. In de lus gebruikten we een blok code om de taken die herhaald moesten worden, uit te voeren.

En nu wat experimentjes:

Experiment 6.1:

Wat zal gebeuren als je de volgende code uitvoert?

```
>>> for a in range(0, 20):
    print ('Hallo %s' % a)
    if a < 9:
        break
>>>
```

Experiment 6.2:

Als je geld spaart bij de bank, krijg je intrest (vraag maar eens aan je ouders). Als je deze intrest laat staan, dan groeit je spaargeld. Intrest wordt berekend met percentages. Als de bank je 1% intrest geeft dan moet je je geld vermenigvuldigen met 0.01 (als je €100 hebt, dan doe je $€100 * 0.01$). Vergeet niet dat programmeurs, zoals rekenmachines, een puntje gebruiken in plaats van een komma. Als de bank je 2% intrest geeft, dan vermenigvuldig je met 0.02 en zo verder. Als je €200 gespaard hebt bij je bank en ze betalen je elk jaar 3% intrest, hoeveel geld zou je dan elk jaar hebben na 10 jaar gespaard te hebben? Je kan een programma maken met een for-loop. Een tip: denk eraan om de intrest bij het totaal op te tellen.

Experiment 6.3:

Wat is het resultaat van volgende (and vervangen door or)?

```
>>> x = 15
>>> y = 30
>>> while x < 20 or y < 50:
    x = x + 1
    y = y + 1
    print(x, y)
>>>
```

7. We gaan wat recycleren

Wij allemaal produceren veel afval: blikjes frisdrank, snoepzakjes, het doosje van de koekjes, de verpakking van de groenten, de plastic zak die je in de winkel krijgt, de zak waarin het brood zit, de kranten en tijdschriften,... Als al dat afval gewoon gestort zou worden dan zouden we nogal een puinhoop hebben. Daarom moeten we zoveel mogelijk recycleren, zodat we kunnen schrijven op gerecycleerd papier, zodat we terug plastic flesjes kunnen maken en we de afvalberg zo klein mogelijk houden.

Ook bij het programmeren kunnen we recycleren, en ook daar is het net zo belangrijk. Want je weet nog dat goede programmeurs een beetje lui moeten zijn. Ze typen niet graag teveel.

Er zijn verschillende manieren om in Python, maar ook in andere programmeertalen, code opnieuw te gebruiken. Eigenlijk hebben we er reeds eentje gezien, namelijk de functie `range()`. (kijk maar eens terug naar hoofdstuk 6. Opnieuw en opnieuw en opnieuw). Dus functies zijn een manier om code opnieuw te gebruiken: je schrijft de code eenmaal en je kan ze in je programma's steeds opnieuw gebruiken. Laten we eens een simpel voorbeeld van een functie opmaken:

```
>>> def mijn_functie (mijn_naam):
    print ('Dag %s' % mijn_naam)
>>>
```

De bovenstaande functie heeft de naam `mijn_functie()` en de parameter `mijn_naam`. Een parameter is een variabele die alleen te vinden is binnen in de functie waar deze bij staat. Dus het blok code dat direct start na de regel waarin `def` staat. Als je jezelf afvraagt waarvoor `def` staat, dit is een afkorting van het Engelse *define* of in het Nederlands *definieer*, wat betekent *leg vast* of *bepaal*. Je kan deze functie uitvoeren door de naam die je gegeven had op te geven en tussen de haakjes de parameter waarde (zoals net gezegd: de variabele binnen in deze functie) te plaatsen:

```
>>> mijn_functie ('Scott')
Dag Scott
>>>
```

Let hierboven op: de waarde van de parameter is een string, dus we moeten aanhalingstekens plaatsen. Als we een getal zouden zetten als parameter waarde, dan wordt deze afgedrukt:

```
>>> mijn_functie (1234)
Dag 1234
>>>
```

We kunnen ook meer dan één parameter geven aan de functie:

```
>>> def mijn_functie(naam1, naam2):
    print ('Dag %s en %s' % (naam1, naam2))
```

```
>>>
```

We kunnen deze functie op gelijkaardige wijze als de eerste keer oproepen:

```
>>> mijn_functie ('oma', 'opa')
Dag oma en opa
>>>
```

Maar we kunnen ook eerst enkele variabele aanmaken en dan de functie aanroepen met deze variabelen:

```
>>> nm1 = 'Jan'
>>> nm2 = 'Bruno'
>>> mijn_functie (nm1, nm2)
Dag Jan en Bruno
>>>
```

Tevens kunnen we waarden als output bekomen door de **return**-opdracht (Return is Engels voor *geef terug*). Als we hiervoor ons spaargeld van een week mee willen berekenen, krijgen we onderstaande:

```
>>> def gespaard(zakgeld, klusjes_oma, snoepjes):
    return zakgeld + klusjes_oma - snoepjes

>>> print (gespaard(5, 7, 8))
4
```

Bij deze bovenstaande functie zijn er 3 parameters nodig. Eerst worden de eerste twee opgeteld en daarna wordt de laatste parameter hiervan afgetrokken. Het resultaat hiervan wordt dan teruggegeven, door de **return**-opdracht. Dit resultaat wordt daarna gebruikt als de waarde van een variabele (net zoals we bij andere variabelen waarden geven):

```
>>> gespaard_geld = gespaard(100, 50, 20)
>>> print (gespaard_geld)
130
>>>
```

Let wel op: een variabele die we binnen in de functie gebruiken is niet bruikbaar wanneer de functie gestopt is:

```
>>> def testje():
    a = 5
    b = 10
    return a * b

>>> print (testje())
50
>>> print (a)
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    print (a)
NameError: name 'a' is not defined
>>> print (b)
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    print (b)
```

```
NameError: name 'b' is not defined
>>>
```

In bovenstaande voorbeeldje hebben we de functie `testje()` gemaakt die 2 variabelen `a` en `b` vermenigvuldigt en dan het resultaat teruggeeft met `return`. Als we deze functie oproepen met de opdracht `print` dan krijgen we als antwoord 50. Maar als we de waarde van de variabelen `a` of `b` uit de functie proberen af te drukken krijgen we een foutboodschap die lijkt op wat we bovenstaand krijgen `NameError: name 'a' is not defined`.

Programmeurs spreken hierbij over de *scope* (is Engels voor *bereik*) die beperkt wordt doordat de waarden van `a` of `b` niet buiten de functie bruikbaar zijn.

Om dit uit te leggen moet je jezelf even voorstellen dat een functie iets is als een klein eilandje in de oceaan. Het is veel te ver om van daar ergens naartoe te zwemmen. Soms vliegt er een vliegtuig over en dropt stukjes papier met woordjes op het eiland (dit zijn de parameters die bij de functie komen). De bewoners van het eiland voegen deze papiertjes samen tot een boodschap die ze in een fles stoppen (met een kurk erop) en die ze dan in zee gooien (dit is de waarde van `return`). Wat de eilandbewoners voor de rest op het eiland doen en met hoeveel ze die boodschap gemaakt hebben speelt geen rol voor de persoon die elders de fles vindt en de boodschap leest. Het speelt ook geen rol na hoeveel tijd die persoon de fles vindt. Dit is dus de scope, het beperken van het bereik (de vinder die alleen de boodschap kent en niks weet van wat er voor de rest op het eiland gebeurt).

MAAR: er is een klein probleempje met deze idee. Eén van de eilandbewoners heeft een grote verrekijker en kan kijken tot het vasteland. Hij kan zien wat de mensen daar allemaal doen en kan de boodschap die ze maken daaraan aanpassen:

```
>>> c = 30
>>> def test():
    a = 10
    b = 20
    return a * b * c

>>> print (test())
6000
>>>
```

Dus zelfs als de variabelen `a` en `b` niet buiten de functie gebruikt kunnen worden kan de variabele `c` die buiten de functie aangemaakt is, wel binnen de functie gebruikt worden. Denk aan de eilandbewoner met de grote verrekijker.

We kunnen ook de for-loop die we eerder gemaakt hadden om ons spaargeld gedurende een jaar te berekenen in een functie stoppen:

```
>>> def jaar (zakgeld, klusjes_oma, snoepjes):
    gespaard = 0
    for week in range (1, 53):
        gespaard = gespaard + zakgeld + klusjes_oma - snoepjes
```

```

    print ('Week %s = €%s in de spaarpot' % (week, gespaard))

>>> jaar (5, 7, 8)
Week 1 = €4 in de spaarpot
Week 2 = €8 in de spaarpot
Week 3 = €12 in de spaarpot
Week 4 = €16 in de spaarpot
Week 5 = €20 in de spaarpot
Week 6 = €24 in de spaarpot
Week 7 = €28 in de spaarpot
Week 8 = €32 in de spaarpot
Week 9 = €36 in de spaarpot
Week 10 = €40 in de spaarpot
Week 11 = €44 in de spaarpot
Week 12 = €48 in de spaarpot
Week 13 = €52 in de spaarpot

```

En zo gaat het verder tot de laatste week van het jaar.

Als we meer zakgeld krijgen (€10) en voor de klusjes van oma meer centen ontvangen (€8), maar ook minder zouden snoepen (€5), hoeveel zouden we dan overhouden?

```

>>> jaar (10, 8, 5)
Week 1 = €13 in de spaarpot
Week 2 = €26 in de spaarpot
Week 3 = €39 in de spaarpot
Week 4 = €52 in de spaarpot
Week 5 = €65 in de spaarpot
Week 6 = €78 in de spaarpot
Week 7 = €91 in de spaarpot
Week 8 = €104 in de spaarpot
Week 9 = €117 in de spaarpot
Week 10 = €130 in de spaarpot
Week 11 = €143 in de spaarpot
Week 12 = €156 in de spaarpot
Week 13 = €169 in de spaarpot

```

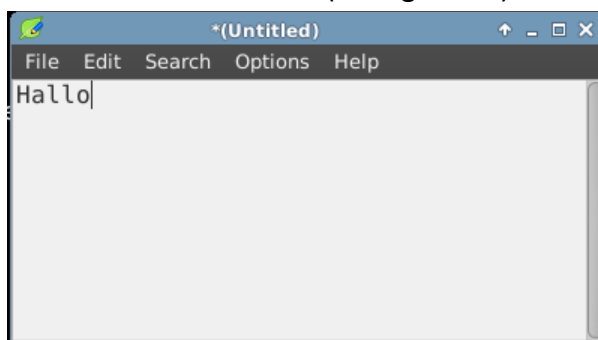
En zo verder tot en met Week 52.

Dit is toch wel beter bruikbaar dan telkens opnieuw de for-loop te moeten hertypen als je eens andere waarden wil gebruiken. We kunnen functies ook samen nemen in een groep en dat noemen we dan 'modules'. En daar wordt Python echt goed bruikbaar. Maar Modules zien we binnenkort.

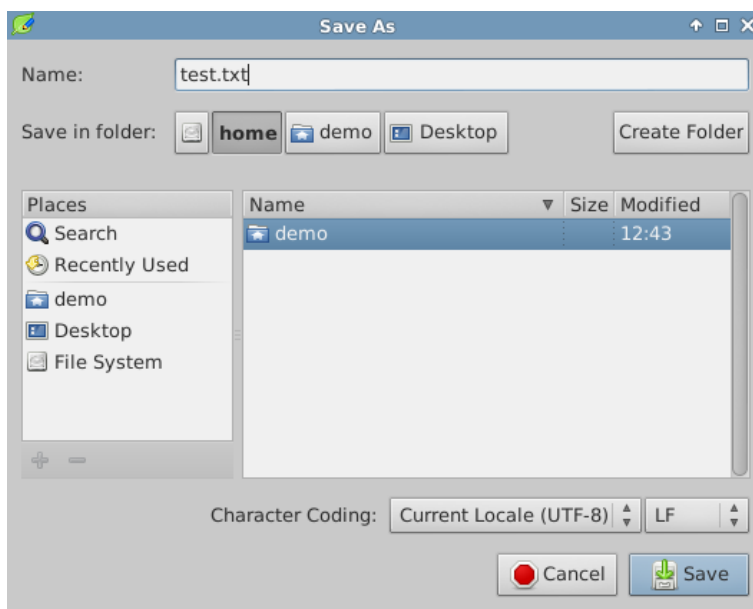
7.1. Stukjes en brokjes

Toen Python op de computer geïnstalleerd werd, zijn ook een hele hoop functies en modules meegeïnstalleerd. Sommige functies zijn standaard beschikbaar. De functie `range()`, die we reeds besproken hebben in de vorige hoofdstukken, is er zo één. Een functie die we nog niet gebruikt hebben is `open()`.

Om te zien hoe de functie **open** gebruikt wordt open je een tekst editor (Notepad++, Leafpad of een andere) en je typt er wat tekst in zoals in Figuur 12. Dan bewaar je het bestand in je “Home”-directory door op File, daarna op “Save as”, dan de “Home”-folder te klikken. Je geeft het bestand de naam “test.txt” (zie Figuur 13).



Figuur 12: Wat tekst in de tekst editor Leafpad



Figuur 13: de “Save As” dialoog van de Leafpad editor

Open daarna terug de Python-console en voer het volgende in:

```
>>> bestandje = open('test.txt')
>>> print (bestandje.read())
Hallo
>>>
```

Wat voert dit stuk code nu uit? De eerste regel roept de functie **open** op en geeft de naam door van het bestand dat je net voordien gemaakt had (**test.txt**) als parameter. De functie geeft een speciale waarde terug (dat we een object noemen) die het bestand weergeeft. Let op: het is niet het bestand zelf. Het is een beetje zoals wijzen met je wijsvinger naar het bestand: ‘hier is het!!’. Het bestandsobject wordt bewaard in de variabele **bestandje**.

De volgende regel roept een speciale functie (**read()**) op van het bestandsobject om de inhoud van het bestand in te lezen en af te drukken op het scherm van de Python-console.

Omdat de variabele `bestandje` een object bevat moeten we de functie `read` gebruiken met een puntje (.) ervoor zodat we `bestandje.read()` krijgen.

In Bijlage B: Functies op het einde van het boek staat heel wat meer informatie over de ingebouwde functies in Python.

7.2. Modules

We hebben intussen al enkele verschillende manieren gezien om code opnieuw te gebruiken: we kunnen functies zelf maken of de ingebouwde functies van Python gebruiken (zoals `int()`, `str()`, `range()`, `open()`). Daarnaast hebben we de speciale soort functies met objecten, die we oproepen met het punt-symbool (.). Verder hebben we modules die een heleboel functies en objecten kunnen groeperen. We zagen eerder al de module 'turtle' in hoofdstuk 4. Schildpadden en andere trage beesten. Een ander voorbeeld van zoiets is de module `time` (Engels voor *tijd*). Open de Python-console en voer volgende opdracht uit:

```
>>> import time
>>>
```

Met het `import` commando vertellen we Python dat we toegang willen tot een module. In het voorbeeldje hierboven willen we toegang tot de `time`-module. Dan kunnen we functies en objecten oproepen die in deze module beschikbaar zijn door het punt-symbool te gebruiken vóór de functie):

```
>>> print (time.localtime())
time.struct_time(tm_year=2015, tm_mon=1, tm_mday=8, tm_hour=22,
tm_min=0, tm_sec=20, tm_wday=3, tm_yday=8, tm_isdst=0)
>>>
```

De functie `localtime()` is een functie binnen in de module `time` die de huidige datum en tijd weergeeft. Dit is opgedeeld in aparte stukken: jaar, maand, dag, uur, minuut, seconde, weekdag, dag van het jaar, en als laatste of rekening gehouden wordt met zomertijd (1 indien wel, 0 indien niet). De aparte stukken worden in een tuple opgeslagen (zie hoofdstuk 3.7 Tuples en Lijsten). Je kan een andere functie in de module `time` gebruiken om de datum en tijd die weergegeven wordt door de functie `localtime()` om te vormen in iets dat beter begrijpbaar is:


```
>>> tijd = time.localtime()
>>> print (time.asctime(tijd))
Thu Jan  8 22:02:59 2015
>>>
```

Maar we kunnen dat ook allemaal in 1 regel code stoppen:


```
>>> print (time.asctime(time.localtime()))
Thu Jan  8 22:04:01 2015
>>>
```

Stel nu even dat je iemand die aan de computer zit wil vragen om een waarde in te geven. Dit kan je doen door een print-opdracht te gebruiken samen met de module 'sys', op de zelfde manier opgeroepen als de 'time'-module met de opdracht `import`:

```
>>> import sys
>>>
```

Binnen in de 'sys'-module bevindt zich een object `stdin` (wat staat voor standaard input). Dit object heeft een methode (of functie) genoemd `readline()`, die gebruikt wordt om de regel tekst in te lezen van iemand die op het toetsenbord iets intypt (tot die persoon op de -toets drukt. Probeer het eens uit in de Python-console:

```
>>> print (sys.stdin.readline())
ik leer Python
ik leer Python
>>>
```

Als je dan een zinnetje typt en dan op de -toets drukt, zal dat zinnetje herhaald worden. Kijk nog eens terug naar de code die we eerder gemaakt hebben met een if-statement (zie 5.3 Conditie samenvoegen):

```
>>> if leeftijd >= 9 and leeftijd <= 12:
    print ('Je bent %s' % leeftijd)
else:
    print( 'Je hebt zeker een andere leeftijd')
```

In plaats van de variabele op voorhand aan te maken, kunnen we nu ook aan iemand vragen om de waarde in te geven. Maar we moeten dan eerst de code in een functie veranderen:

```
>>> def je_leeftijd(leeftijd):
    if leeftijd >=9 and leeftijd <= 12:
        print ('Je bent %s' % leeftijd)
    else:
        print ('Je hebt zeker een andere leeftijd')
>>>
```

Dit kan opgeroepen worden door een getal als waarde voor de parameter in te vullen. We gaan eerst eens testen of dit wel werkt:

```
>>> je_leeftijd (8)
Je hebt zeker een andere leeftijd
>>> je_leeftijd (11)
Je bent 11
>>>
```

Nu we weten dat er geen problemen zijn met onze functie, kunnen we de functie veranderen zodat we vragen naar de leeftijd van diegene die aan het toetsenbord zit:

```
>>> def je_leeftijd():
    print ('Geef je leeftijd in:')
    leeftijd = int(sys.stdin.readline())
    if leeftijd >=9 and leeftijd <= 12:
        print ('Je bent %s' % leeftijd)
```

```

    else:
        print ('Je hebt zeker een andere leeftijd')

>>>

```

Omdat `readline()` weergeeft wat iemand als tekst intypt (eigenlijk een string), moeten we de functie `int()` gebruiken om de string naar een getal om te vormen. Dan pas kan dit correct werken in het if-statement.

Probeer nu zelf eens door de functie `je_leeftijd()` op te roepen zonder parameters. Typ daarna een getal als 'Geef je leeftijd in:' verschijnt:

```

>>> je_leeftijd()
Geef je leeftijd in:
9
Je bent 9
>>> je_leeftijd()
Geef je leeftijd in:
6
Je hebt zeker een andere leeftijd
>>> je_leeftijd()
Geef je leeftijd in:
h
Traceback (most recent call last):
  File "<pyshell#90>", line 1, in <module>
    je_leeftijd()
  File "<pyshell#85>", line 3, in je_leeftijd
    leeftijd = int(sys.stdin.readline())
ValueError: invalid literal for int() with base 10: 'h\n'
>>>

```

Zolang je een getal ingeeft zal Python altijd een string teruggeven. Als je een letter of iets anders dan een getal ingeeft, krijg je een foutboodschap omdat Python in dit geval niet weet wat daarmee aan te vangen.

Sys en time zijn slechts 2 van de vele modules die in Python zitten. Als je meer informatie wil over enkele andere ingebouwde modules, kijk dan eens naar Bijlage C: Modules.

7.3. Dingen om zelf eens te doen

In dit hoofdstuk hebben we geleerd hoe we moeten recyclen in Python door functies en modules te gebruiken. We hebben ook iets geleerd over het bereik van een variabele en hoe variabelen buiten functies ook erbinnen gezien kunnen worden, maar waarbij variabelen die in functies zitten erbuiten niet gezien kunnen worden. We hebben ook gezien hoe we zelf een functie maken met `def`.

En nu wat experimentjes:

Experiment 7.1:

In experiment 6.2 hebben we gezien hoe we een for-loop moesten maken om de intrest te berekenen als je €200 gespaard had gedurende 10 jaar. Die for-loop kan gemakkelijk in een functie gestopt worden. Probeer een functie te maken die een startbedrag en een percentage intrest gebruikt. Je kan de functie `bereken_intrest()` noemen zoals:

```
bereken_intrest(200, 0.03)
```

Experiment 7.2:

Neem de functie die je net gemaakt hebt en doe deze de intrest berekenen voor verschillende periodes, zoals bijvoorbeeld 7 jaar of 12 jaar. Je zou het kunnen doen zoals

```
bereken_intrest(200, 0.03, 7)
```

Experiment 7.3:

Maak eens, in plaats van een simpele functie waarbij je de waarden als parameters doorgeeft, een klein programma dat iemand vraagt om de waarden in te geven met de functie `sys.stdin.readline()`. In dit geval kunnen we de functie zonder parameters oproepen: `bereken_intrest()`

Om dit mini-programma te schrijven hebben we wel een functie nodig die nog niet besproken is: `float()`. Dit is een functie die wat lijkt op de functie `int()`, maar die strings verandert in getallen met een decimale punt. In computertaal spreekt men van *floating point number* wat neerkomt op een *getal met een decimale punt*, zoals 5.50 of 123.456, maar niet 6 of 28.

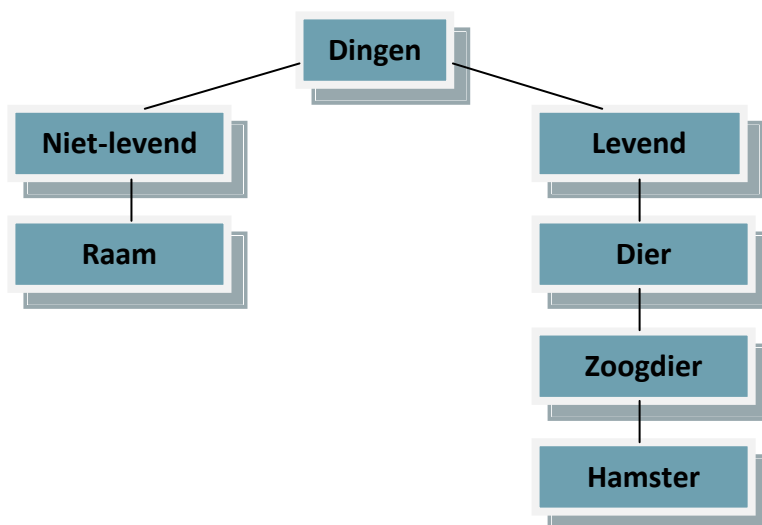
8. Klassen en objecten

Een hamster is zoals een raam. Je zal je afvragen of ik van de tafel gevallen ben en ben blijven botsen. En toch is een hamster zoals een raam: het zijn dingen en in Python noemen we zo iets objecten. Bij programmeren is het idee van objecten belangrijk, het is een manier van code in een programma te organiseren en dingen op te delen om moeilijke zaken in vele maar minder moeilijke delen op te splitsen. We hebben reeds een object gebruikt in hoofdstuk 4 toen we met de `Pen()`–functie bij onze schildpad werkten.

Om goed te begrijpen hoe objecten in Python werken, moeten we denken aan types van objecten. We hebben het reeds gehad over de hamster en het raam. Een hamster is een zoogdier en een zoogdier is een type van een dier; en onze hamster is levend. Aan de andere kant hebben we ons raam. Daar kunnen we niet zoveel over zeggen, hé. Een raam leeft niet, het is een niet-levend object. We hebben nu reeds enkele klassen gezien: zoogdier, dier, levend, niet-levend. Met klassen kunnen we dus... klasseren.

8.1. Opdelen in klassen

In Python worden objecten gedefinieerd door klassen. Bij klassen kunnen we denken aan een manier om objecten in groepen op te delen. Hieronder zie je een opdeling van de klassen voor onze hamster en ons raam:



Zoals je in bovenstaande structuur ziet is de hoofdklasse **Dingen**. Onder de klasse **Dingen** hebben we **Niet-levend** en **Levend**. Deze klassen zijn verder opgedeeld in **Raam** voor **Niet-levend** en voor **Levend** is de opdeling verder gemaakt in **Dier**, **Zoogdier** en **Hamster**.

We kunnen klassen gebruiken om stukken Python code te organiseren. Als we terugdenken aan de module `turtle`, dan zijn de dingen die de schildpad kan doen (vooruit, achteruit, links en rechts bewegen) functies in de klasse `Pen()`. We kunnen een object beschouwen als een onderdeel van een klasse en we kunnen veel objecten aanmaken voor een klasse. Straks

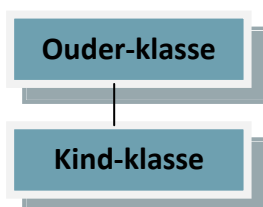
meer daarover. Eerst gaan we dezelfde klassen aanmaken zoals we in onze boomstructuur boven hebben, startende met de top-klasse **Dingen**:

```
>>> class Dingen:
>>>     pass
>>>
```

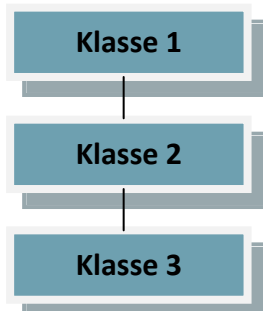
We benoemen de topklasse **Dingen** en gebruiken dan de **pass**-opdracht omdat we Python nu nog niet meer informatie wensen te geven. Nu gaan we de andere klassen aanmaken alsook de onderlinge verbanden bespreken.

Kinderen en ouders

Wanneer een klasse een onderdeel is van een tweede klasse, dan is die eerste klasse een kind van de tweede klasse.



Klassen kunnen zowel ouder als kind zijn ten overstaan van andere klassen. Zo is onderstaand **Klasse 2** kind van **Klasse 1**, maar ouder van **Klasse 3**.



Zo ook in onze boomstructuur. **Niet-levend** en **Levend** zijn beiden kinderen van de klasse **Dingen**. Dus **Dingen** is hun ouder. Om Python te vertellen dat een klasse een kind van een andere klasse is moeten we de naam van de ouder tussen ronde haakjes zetten na de naam van de nieuwe klasse:

```
>>> class Niet_Levend(Dingen):
>>>     pass
>>>
>>> class Levend(Dingen):
>>>     pass
>>>
```

Hier hebben we de klasse **Niet_Levend()** gemaakt en aan Python laten weten dat zijn ouder-klasse **Dingen()** is met de code **class Niet_Levend(Dingen)**. We hebben nadien hetzelfde gedaan met de klasse **Levend()**.

We kunnen ook hetzelfde doen met de klasse `Raam()`. We maken deze klasse met als ouder `Niet_Levend` en we doen hetzelfde met `Zoogdier()`, `Dier()` en `Hamster()` bij hun ouders:

```
>>> class Raam(Niet_Levend):
    pass

>>> class Dier(Levend):
    pass

>>> class Zoogdier(Dier):
    pass

>>> class Hamster(Zoogdier):
    pass

>>>
```

Objecten toevoegen aan klassen

We hebben nu een hoop klassen aangemaakt, maar we hebben nog niets in die klassen gedaan. Stel dat onze hamster Jackie heet. We weten dat hij tot de klasse `Hamster()` behoort, maar wat moeten we gebruiken in programmeer-taal om onze hamster Jackie te noemen? We noemen **Jackie** een *object* van de klasse `Hamster()`. En om dit in code te zetten doen we het volgende:

```
>>> Jackie = Hamster()
>>>
```

Deze code vertelt Python dat in de klasse `Hamster()` een object moet gemaakt worden dat dit object moet toegewezen worden aan **Jackie**. Net als bij een functie wordt een klassenaam gevolgd door ronde haakjes. Later zien we hoe objecten gemaakt worden en hoe parameters gebruikt worden tussen die ronde haakjes.

Maar wat doet dit object **Jackie** nu eigenlijk? Wel, op dit moment nog niets. Om onze objecten bruikbaar te maken wanneer we een klasse aangemaakt hebben, moeten we ook functies aanmaken die met deze objecten in die klasse gebruikt kunnen worden. Dus gaan we het trefwoord `pass` vervangen door functies.

Klasse-functies aanmaken

In hoofdstuk 7 hebben we gezien hoe we functies gebruiken om code te recyclen. Wanneer we een functie aanmaken die bij een klasse hoort, doen we dat op dezelfde manier als wanneer we een andere functie aanmaken, behalve dat we het als blok onder de klassedefinitie plaatsen (met 4 spaties). Laat ons eerst eens een normale functie aanmaken die niets met een klasse vandoen heeft:

```
>>> def normale_functie():
    print('Normale functie')

>>>
```

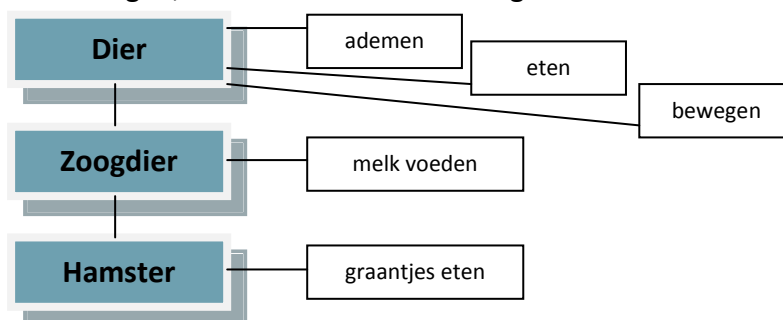

En enkele functies die bij een klasse horen:

```
>>> class Rare_klasse:
    def een_klasse_functie():
        print('Een klassefunctie')
    def nog_een_klasse_functie():
        print('Nog een klassefunctie')
>>>
```

Klasse karakteristiek als functie toevoegen

Als we terugkijken naar de kinderen van de klasse `Levend()` dan kunnen we *karakteristieken* toevoegen aan elke klasse om te beschrijven wat ze is en wat ze kan doen. Een karakteristiek is een karaktertrek of een eigenschap die alle leden van een klasse (en hun kinderen) delen. Net zoals een gewoon kind sommige karaktertrekken heeft van zijn ouders.

Wat hebben alle dieren bijvoorbeeld gemeen met elkaar? Ze kunnen ademen, ze eten en bewegen. En wat hebben zoogdieren gemeen? Ze voeden hun jongen met melk, maar ze ademen ook en ze eten & bewegen. Onze hamster eet graag graantjes, maar geeft ook melk aan zijn kleintjes en kan ook ademen, eten en bewegen. Als we al deze karakteristieken aan onze boomstructuur hangen, dan bekomen we het volgende:



We kunnen over deze karakteristieken denken als functies, dingen die een object van deze klasse kan doen. Om een functie bij een klasse te voegen gebruiken we het trefwoord **def**. Dus de klasse `Dieren()` ziet er als volgt uit:

```
>>> class Dier(Levend):
    def ademen(zelf):
        pass
    def bewegen(zelf):
        pass
    def eten(zelf):
        pass
>>>
```

In de eerste regel definiëren we terug de klasse zoals we eerder deden, maar we voegen in regel 2 de zelfgedefiniëerde functie `ademen()` toe en we geven ze één parameter `zelf`.

Deze parameter **zelf** is een manier voor een functie in de klasse om een andere functie in de klasse (en in de ouder-klassen) op te roepen. Het gebruik van de parameter zien we later wel.

In Regel 3 zeggen we terug tegen Python met het trefwoord **pass** dat we voorlopig niets verder gaan uitvoeren met deze functie. Daarna maken we op identieke wijze de functies **bewegen()** en **eten()** aan, die op dit ogenblik nog niks gaan uitvoeren. Later voegen we code toe. Programmeurs doen dit dikwijls bij het coderen: ze maken reeds klassen aan met functies die nog niets moeten uitvoeren. Hiermee behouden ze een goed overzicht bij complexere programma's.

Ook bij de 2 andere klassen (**Zoogdieren()** en **Hamster()**) moeten we functies toevoegen. Elke klasse kan de karakteristieken (functies) van haar ouder gebruiken. Dit betekent dat je niet één heel complexe klasse moet maken: je kan je functies in de hoogste klasse (de ouder) stoppen vanaf waar de karaktertrek van toepassing is. Elke kind-klasse daaronder erft deze eigenschappen toch. Hierdoor wordt de code eenvoudiger.

```
>>> class Zoogdier(Dier):
        def melk_voeden(zelf):
            pass

>>> class Hamster(Zoogdier):
        def graantjes_eten(zelf):
            pass

>>>
```

Waarom klassen en objecten gebruiken?

We hebben nu wel functies toegevoegd aan onze klassen, maar waarom zouden we eigenlijk klassen en objecten gebruiken als je ook normale functies kan gebruiken? Om dit aan te tonen gaan we onze hamster **Jackie** gebruiken die we reeds als een object van de klasse **Hamster()** gemaakt hebben:

```
>>> Jackie = Hamster()
>>>
```

Omdat **Jackie** een object is kunnen we functies geleverd door deze klasse **Hamster()** of ouder-klassen oproepen en uitvoeren. Functies van een object roepen we op met de *dot*-operator (Engels voor *punt-operator*) met nadien de naam van de functie. Om onze hamster Jackie te zeggen dat ze moet eten en bewegen kunnen we volgende functies oproepen:

```
>>> Jackie.graantjes_eten()
>>> Jackie.bewegen()
>>>
```

Stel dat onze hamster **Jackie** een hamstervriendje heeft met de naam **Johnny**. Dan moeten we een ander hamsterobject aanmaken:

```
>>> Johny = Hamster()
>>>
```

Omdat we objecten en klassen gebruiken, kunnen we perfect tegen Python zeggen over welke hamster we praten wanneer we de functie `bewegen()` willen uitvoeren:

```
>>> Johny.bewegen()
>>>
```

Hier zal enkel hamster **Johny** bewegen.

Als we onze klassen nu een beetje veranderen om het wat duidelijker te maken en we veranderen `pass` door een `print`-opdracht:

```
>>> class Dier(Levend):
    def ademen(zelf):
        print('ademen')
    def bewegen(zelf):
        print('bewegen')
    def eten(zelf):
        print('eten')

>>> class Zoogdier(Dier):
    def melk_voeden(zelf):
        print('melk aan jongen geven')

>>> class Hamster(Zoogdier):
    def graantjes_eten(zelf):
        print('graantjes eten')

>>>
```

Als we nu onze objecten **Jackie** en **Johny** aanmaken en functies bij hen oproepen dan zien we nu het volgende gebeuren:

```
>>> Jackie = Hamster()
>>> Johny = Hamster()
>>> Jackie.bewegen()
bewegen
>>> Johny.graantjes_eten()
graantjes eten
>>> Jackie.melk_voeden()
melk aan jongen geven
>>>
```

Eerst maken we de variabelen **Jackie** en **Johny** die objecten zijn van de klasse `Hamster()`. Daarna roepen we de functie `bewegen()` op bij **Jackie**. Op dezelfde manier roepen we **Johny** op om de functie `graantjes_eten` uit te voeren en laten we **Jackie** `melk_voeden` aan haar jongen.

Objecten en klassen in figuren

Wat als we onze objecten en klassen eens bij figuren zouden toepassen?

We keren nog even terug naar onze schildpad van hoofdstuk 4 waar we `turtle.Pen()` gebruikten. Als we `turtle.Pen()` gebruiken, maakt Python een object van de `Pen()` klasse aan dat door de module `turtle` wordt geleverd (net zoals onze **Jackie** en **Johny** objecten in de vorige paragraaf).

Wat als we nu 2 schildpadobjecten zouden maken met de namen **Stig** en **Siebe**, net zoals we met onze hamsters gedaan hebben?

```
>>> import turtle
>>> Stig = turtle.Pen()
>>> Siebe = turtle.Pen()
>>>
```

Elk schildpadobject (**Stig** en **Siebe**) is lid van de klasse `Pen()`. Nu gaan we zien waarom objecten een goed hulpmiddel zijn om te gebruiken. We hebben net onze schildpadobjecten gemaakt en kunnen er functies mee uitvoeren. Beide zullen onafhankelijk van elkaar tekenen:

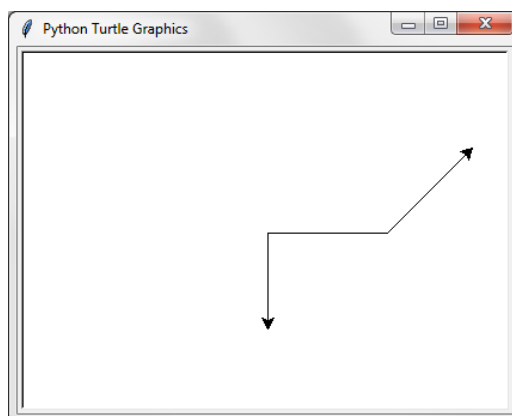
```
>>> Stig.forward(100)
>>> Stig.left(45)
>>> Stig.forward(100)
>>>
```

Met de bovenstaande functies laten we **Stig** bewegen, eerst 100 pixels voorwaarts, daarna moet **Stig** 45° naar links draaien en dan terug 100 pixels vooruitgaan.

En dan is het tijd om **Siebe** te laten bewegen:

```
>>> Siebe.right(90)
>>> Siebe.forward(80)
>>>
```

Eerst moet **Siebe** 90 graden naar rechts draaien en dan 80 pixels vooruit gaan (zie Figuur 14).



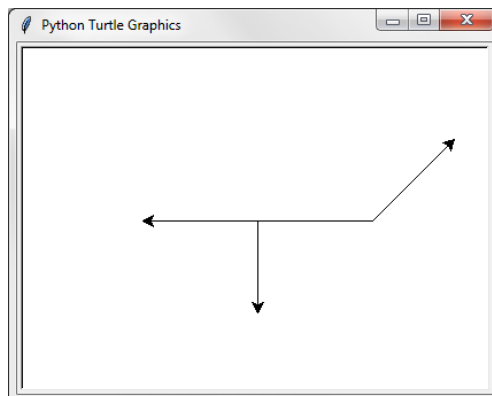
Figuur 14: Bewegingen van Stig en Siebe

En we gaan onze 2 vriendjes nog een vriendje bijgeven. **Stef** wordt onze derde schildpad die ook wil bewegen:

```
>>> Stef = turtle.Pen()
```

```
>>> Stef.right(180)
>>> Stef.forward(100)
>>>
```

Eerst maken we een nieuw `Pen()` object dat we `Stef` noemen. We laten hem 180 graden naar rechts draaien, waarna `Stef` 100 pixels vooruit gaat. En dan zien we 3 schildpadden op ons scherm (zie Figuur 15).



Figuur 15: Met extra vriendje `Stef`

Denk er dus aan dat steeds wanneer we `turtle.Pen()` oproepen, we telkens een nieuw onafhankelijk object aanmaken. Elk object hoort wel bij de klasse `Pen()` en we kunnen bij elk object dezelfde functies gebruiken, maar omdat we objecten gebruiken kunnen we elke schildpad afzonderlijk aansturen.

Net zoals onze onafhankelijke hamsters `Jackie` en `Johnny`, zijn `Siebe`, `Stig` en `Stef` onafhankelijke schildpadobjecten. Als we een nieuw object aanmaken met dezelfde naam als variabele zal het oude object niet noodzakelijk verdwijnen. Probeer maar eens zelf uit met een 3^e schildpad `Senne`.

8.2. Andere eigenschappen van objecten en klassen

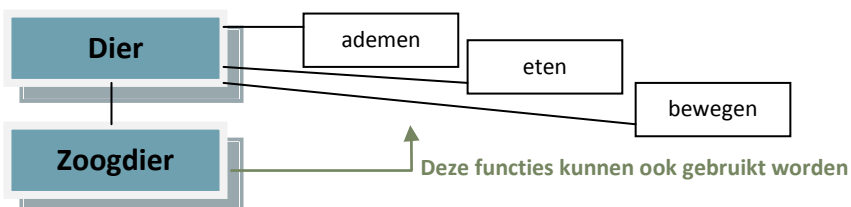
Klassen en objecten maken het gemakkelijk om functies te groeperen. Ze zijn ook goed bruikbaar om programma's in kleinere stukjes op te delen, zodat ze gemakkelijker leesbaar zijn. Als je bijvoorbeeld een zeer groot 3D-game zou hebben dan is het voor veel mensen heel moeilijk om de code te snappen omdat het zo veel is. Als je dit programma in stukjes hakt, dan wordt het gemakkelijker om te lezen en te begrijpen (als je de programmeertaal kent natuurlijk).

Als je een groot programma in stukjes hakt, kan je het werk ook gemakkelijker verdelen over verschillende programmeurs. Zo worden games en andere software ook gemaakt. Het zijn teams van programmeurs die eraan werken.

Stel dat we de klassen die we gemaakt hebben (**Dieren()**, **Zoogdieren()**, **Hamster()**) serieus willen uitbreiden maar onze vrienden moeten helpen omdat het teveel werk is voor ons alleen. Dan kunnen we aan vriendje A vragen om te werken op **Dieren()** en vriendje B om te werken aan **Zoogdieren()** en zelf kunnen we werken aan **Hamster()**.

Functies erven

Als je even goed nadenkt, dan weet je dat we onszelf een beetje lui gemaakt hebben door de vorige zin. Wij werken aan de klasse **Hamster()**. Zoals reeds gezegd kunnen alle functies die gemaakt worden door onze vriendjes voor de klassen **Dieren()** en **Zoogdieren()** ook gebruikt worden door de klasse **Hamster()**. We zeggen dan dat de klasse **Hamster()** functies van de klasse **Zoogdieren()** erft, net zoals die functies van de klasse **Dieren()** erft. Dus als we een object **Hamster()** maken kunnen we functies gebruiken uit de klasse **Hamster()** maar ook functies uit de klassen **Dieren()** en **Zoogdieren()**. Net zo, wanneer we het object **Zoogdieren()** aanmaken, kunnen we de functies uit de ouder-klasse **Dieren()** gebruiken.



Zelfs al is Jackie een object uit de klasse **Hamster()**, dan kunnen we nog steeds de functie **bewegen()** gebruiken die we in de klasse **Dieren()** hebben aangemaakt.

```
>>> Jackie = Hamster()
>>> Jackie.bewegen()
bewegen
>>>
```

Eigenlijk kunnen alle functies die we gemaakt hebben in de klassen **Dieren()** en **Zoogdieren()** opgeroepen worden door ons object **Jackie** omdat deze functies geërfd worden:

```
>>> Jackie = Hamster()
>>> Jackie.ademen()
ademen
>>> Jackie.etten()
eten
>>> Jackie.melk_voeden()
melk aan jongen geven
>>>
```

Functies die andere functies oproepen

Als we functies van een object oproepen, gebruiken we de variabele van dat object.

Bijvoorbeeld om de functie `bewegen()` op te roepen van `Jackie` voeren we volgende code uit:

```
>>> Jackie.bewegen()
bewegen
>>>
```

Om in de klasse `Hamster()` de functie `bewegen()` op te roepen, moeten we de parameter `zelf` gebruiken. Deze parameter `zelf` is een manier om een functie in een klasse een andere functie te laten oproepen. Een voorbeeldje zal het wat duidelijker maken. Stel dat we aan de klasse `Hamster()` een functie `zoek_eten()` willen toevoegen:

```
>>> class Hamster(Zoogdier):
    def zoek_eten(zelf):
        zelf.bewegen()
        print('Ik heb eten gevonden')
        zelf.eten()

>>>
```

We hebben nu een functie gemaakt die 2 andere functies combineert. Dit gebeurt veel bij coderen en is dus belangrijk om te weten. Het zal regelmatig voorkomen dat je een functie aanmaakt die iets uitvoert en dat je deze binnen een andere functie ook wil gebruiken. We gaan de parameter `zelf` gebruiken om nog wat functies aan de klasse `Hamster()` toe te voegen:

```
>>> class Hamster(Zoogdier):
    def zoek_eten(zelf):
        zelf.bewegen()
        print('Ik heb eten gevonden')
        zelf.eten()
    def graantjes_eten(zelf):
        zelf.eten()
    def springen(zelf):
        zelf.bewegen()
        zelf.bewegen()
        zelf.bewegen()

>>>
```

We gebruiken de functies `eten()` en `bewegen()` van de ouder-klasse `Dieren()` om de functies `graantjes_eten()` en `springen()` te definiëren in de klasse `Hamster()`. Dit kan omdat het geërfdde functies zijn. Door nu functies toe te voegen die op deze wijze andere functies oproepen (wanneer we objecten van deze klassen aanmaken) kunnen we 1 functie oproepen die meer doet dan 1 ding. Je ziet wat er gebeurt als we de functie `springen()` oproepen. Ons hamster Jackie beweegt dan 3 keer met 1 functie:

```
>>> Jackie = Hamster()
>>> Jackie.springen()
```

```
bewegen
bewegen
bewegen
>>>
```

Experimenteer zelf maar verder.

8.3. Een object initialiseren

Soms maken we een object waaraan we waarden (of *eigenschappen*, in het Engels *properties*) willen geven om later te gebruiken. Door een object te initialiseren (een beginwaarde te geven) maken we het klaar om te gebruiken.

Stel dat ons hamstertje vlekjes heeft. We willen het aantal vlekjes op ons hamster-object bepalen wanneer we het object aanmaken (dus initialiseren). Om dit te doen, gebruiken we de functie `__init__` (zie je dat er langs beide zijden van het woord `init` twee underscore _ tekens zijn?). Dit is een speciale functie in Python klassen en moet deze naam hebben.

De functie `init` is een manier om de eigenschappen van een object in te stellen wanneer dit object aangemaakt wordt. Python roept automatisch deze functie op wanneer we een nieuw object aanmaken. Hier zie je hoe het te gebruiken:

```
>>> class Hamster(Zoogdier):
    def __init__(zelf, vlekken):
        zelf.hamster_vlekken = vlekken
>>>
```

Eerst maken we de functie `__init__` aan met 2 parameters: `zelf` en `vlekken` via de code `def __init__(zelf, vlekken):`. Net zoals de andere functies die we in deze klasse gedefiniëerd hebben, moet de functie `__init__` ook `zelf` als de eerste parameter hebben. Daarna geven we de parameter `vlekken` aan een objectvariabele (zijn eigenschap) genoemd `hamster_vlekken` met de parameter `zelf`. Dit doen we door de code `zelf.hamster_vlekken = vlekken` in te geven. Dit stuk code is hetzelfde als zeggen dat we de waarde van de parameter `vlekken` nemen en bewaren voor later gebruik bij het object `hamster_vlekken`. Net zoals één functie in een klasse een andere functie kan oproepen met de parameter `zelf`, kunnen variabelen in een klasse ook opgeroepen worden met deze parameter `zelf`.

Als we twee nieuwe hamsterobjecten aanmaken met de namen `Zikki` en `Zakki` en het aantal vlekken dat ze hebben willen afdrukken, dan zie je de initialisatie-functie aan het werk:

```
>>> Zikki = Hamster(150)
>>> Zakki = Hamster(200)
>>> print(Zikki.hamster_vlekken)
150
```



```
>>> print(Zakki.hamster_vlekken)
200
>>>
```

Eerst maken we een instantie van de klasse `Hamster()` met de parameter `150` (een instantie wordt gebruikt om aan te geven dat een object van een bepaalde klasse is afgeleid). Daarbij wordt de functie `__init__` opgeroepen en de waarde van de parameter `vlekken` op `150` gezet. Daarna maken we een tweede instantie `Zakki` van de klasse `Hamster()` met de parameter `200`. We drukken voor onze beide hamsterobjecten de waarde van `hamster_vlekken` af en we bekommen de juiste waarden.

Denk eraan dat wanneer we een object van een klasse aanmaken (zoals `Zikki` en `Zakki` hierboven), we naar haar variabelen of functies kunnen verwijzen met de dot-operator en de naam van de variabele of functie die we willen gebruiken (zoals `Zakki.hamster_vlekken`). Maar als we binnen een klasse functies aanmaken, dan verwijzen we naar diezelfde variabelen (en functies) met de parameter `zelf` (`zelf.hamster_vlekken`).

In dit hoofdstuk hebben we geleerd om klassen aan te maken waarmee we dingen in bepaalde categorieën kunnen stoppen. Van die klassen hebben we objecten (instanties) gemaakt. We hebben gezien hoe een kind-klasse de functies van zijn ouder-klasse erft en zelfs al behoren 2 objecten tot dezelfde klasse, ze zijn niet noodzakelijk hetzelfde (net zoals broers). We leerden ook hoe functies bij een object kunnen opgeroepen of uitgevoerd worden en hoe object variabelen een manier zijn om waarden in die objecten te bewaren. Tenslotte zagen we hoe de parameter `zelf` in functies gebruikt wordt om te verwijzen naar andere functies en variabelen.

9. Iets over bestanden

Als je al eens met de computer gewerkt hebt, weet je zeker wel wat een *bestand* is. In het Engels heet dit een *File*. Bestanden of files hebben steeds een naam, want anders kan je hen niet terugvinden op je computer. Dat is een beetje zoals de naam van je stripverhalen. Als de naam niet op de kaft zou staan, zou je niet weten welke strip het is. En als je verschillende reeksen van stripverhalen hebt, zoals Jommeke, Urbanus, Kiekeboe, Suske en Wiske of andere dan leg je die waarschijnlijk op stapeltjes bij elkaar zodat je direct weet welke reeks het is. Dit is net hetzelfde hoe op je computer folders (soms ook directories genoemd) georganiseerd worden. Je ziet het, bij computers worden dingen op gelijkaardige manier georganiseerd zoals jij zelf zou doen, zodat je alles gemakkelijk terugvindt.

We hebben in een vorig hoofdstuk 7.1 reeds gezien hoe we een bestandsobject moesten openen. Het voorbeeld zag er uit als volgt:

```
>>> bestandje = open('test.txt')
>>> print (bestandje.read())
```

Een bestandsobject of een fileobject kan wel wat meer dan enkel de functie `read()` gebruiken. Anders zouden we alleen maar bestanden kunnen uitlezen. Dit zou hetzelfde zijn als we enkel een stripverhaal van de hoop kunnen nemen, maar het niet meer kunnen terugleggen.

We kunnen ook een nieuw leeg fileobject aanmaken door er een andere parameter in te stoppen wanneer we de functie oproepen:

```
>>> bestandje = open('een_bestand.txt', 'w')
```

Je ziet dat we hier `'w'` gebruikt hebben waarmee we Python vertellen dat we naar het bestandsobject willen schrijven (de `w` staat voor write, “schrijf”) en niet lezen. Hiermee kunnen we dus informatie aan het bestand toevoegen, kijk maar:

```
>>> bestandje = open('een_bestand.txt', 'w')
>>> bestandje.write ('Dit is een bestand met een test')
```

Daarna moeten we Python ook nog vertellen dat we gedaan hebben en willen stoppen met schrijven naar het bestand. Daarvoor gebruiken we de functie `close()`:

```
>>> bestandje = open('een_bestand.txt', 'w')
>>> bestandje.write ('Dit is een bestand met een test')
>>> bestandje.close
```

Je kan dit bestand openen met de tekst editor die je voordien al gebruikt hebt, maar het is nog beter om het te openen en uit te lezen met Python:

```
>>> bestandje = open('een_bestand.txt')
>>> print (bestandje.read())
Dit is een bestand met een test
>>>
```

10. Hier zijn de schildpadden terug

We gaan nog eens kijken naar onze schildpad die we in hoofdstuk 4 gezien hebben. Weet je nog dat we eerst de module moeten importeren en dan het `Pen()`-object moeten aanmaken?

```
>>> import turtle
>>> t = turtle.Pen()
```

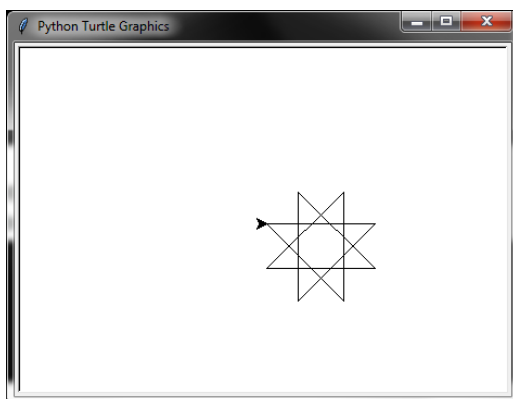
Nu zouden we de basisfuncties kunnen gebruiken om de schildpad rond te laten kruipen en eenvoudige vormen te laten tekenen, maar het is toch wel interessanter om iets te doen wat we in de voorbije hoofdstukken gezien hebben. Zo hebben we volgende code gebruikt om een vierkant te maken:

```
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>>
```

We kunnen dit opnieuw schrijven met een for-loop:

```
>>> t.reset()
>>> for i in range(1,5):
>>>     t.forward(100)
>>>     t.left(90)
>>>
```

Hiervoor moet je heel wat minder intypen, nietwaar. Probeer het volgende ook maar eens en bekijk de figuur die je getekend hebt:



Figuur 16: De schildpad heeft een ster met 8 punten getekend

```
>>> t.reset()
>>> for i in range(1,9):
>>>     t.forward(100)
>>>     t.left(225)
```

```
>>>
```

Deze code maakt een ster met 8 punten zoals je kan zien in Figuur 16 waar de schildpad telkens 225 graden draait naar links nadat ze 100 pixels vooruitgegaan is.

Als de schildpad met een andere hoek draait kan je andere mooie sterren bekomen. Probeer eens met een hoek van 175 graden in 37 stappen (zie Figuur 17):

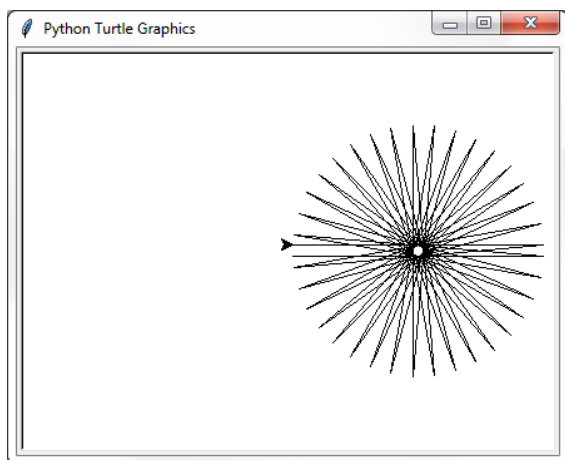
```
>>> t.reset()
>>> for i in range(1,38):
    t.forward(100)
    t.left(175)

>>>
```

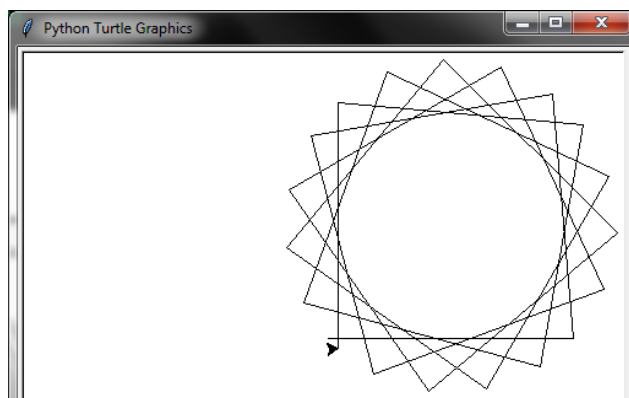
Of nog een andere soort ster (Figuur 18):

```
>>> t.reset()
>>> for i in range(1,20):
    t.forward(100)
    t.left(95)

>>>
```



Figuur 17: Nog een ster



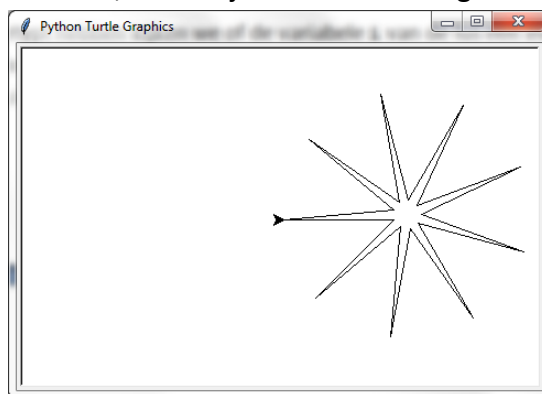
Figuur 18: En nog een andere soort ster

En als we een iets moeilijkere ster willen maken, dan proberen we toch gewoon het volgende:

```
>>> t.reset()
>>> for i in range(1,19):
    t.forward(100)
    if i % 2 == 0:
        t.left(175)
    else:
        t.left(225)
```

>>>

In de code die we zonet ingetypt hebben kijken we of de variabele `i` van de lus een even getal bevat. Daarom gebruiken we een zogezegde modulo-operator (`%`) in de uitdrukking `i % 2 == 0`. Door deze modulo-operator te gebruiken ($x \% 2$ is gelijk aan nul) checken we of het getal in de variabele `i` door twee kan gedeeld worden, zonder dat er een rest is. Misschien is dit nog een beetje te moeilijk om te begrijpen, maar dat is niet zo erg. Onthoud enkel dat je `i % 2 == 0` (`i` kan ook een andere variabele zijn, natuurlijk) kan gebruiken om te kijken of een getal in de variabele een even nummer is (deelbaar door 2 zonder rest). Als je dit programma uitgevoerd hebt, dan zie je iets zoals in volgende figuur:

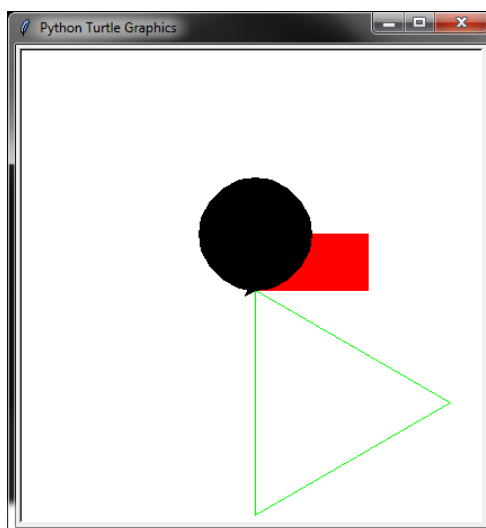


Figuur 19: Een ster met 9 punten.

Je kan ook nog andere dingen tekenen die iets moeilijker lijken en waarbij je nog andere functies kan gebruiken. Zo kan je de kleur van de schildpad veranderen met de functie `color()` met `fill` kan je een vlak opvullen met een kleur. Dat vlak moet je maken tussen de functies `begin_fill()` en `end_fill()`. Je kan ook een cirkel met een bepaalde grootte tekenen met de functie `circle()` en het resultaat zie je in Figuur 20:

```
>>> t.reset()
>>> t.color(1,0,0)
>>> t.begin_fill()
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(50)
>>> t.end_fill()
>>> t.color(0,1,0)
>>> t.forward(200)
>>> t.left(120)
>>> t.forward(200)
>>> t.left(120)
>>> t.forward(200)
>>> t.setheading(0)
>>> t.color(0,0,0)
>>> t.begin_fill()
>>> t.circle(50)
```

```
>>> t.end_fill()
>>>
```



Figuur 20: Een iets moeilijker vorm

10.1. Kleuren gebruiken

De functie `color()` gebruikt 3 parameters:

1. de eerste parameter is een waarde voor de kleur rood;
2. de 2^e is een waarde voor de groene kleur;
3. en de derde parameter is een waarde voor blauw.

Maar waarom nu toch rood, groen en blauw? Dit noemen we het RGB-kleurenmodel. Als je iets van schilderen kent, dan zal je waarschijnlijk al een deel van het antwoord kennen (de computer gebruikt 1 andere kleur dan schilders als primaire kleur (groen in plaats van geel), maar daar hoeft je jezelf niet veel van aan te trekken). Want als je 2 kleuren mengt, dan bekom je een andere kleur. Als je bijvoorbeeld rood en blauw mengt, dan krijg je paars. En als je heel veel verschillende kleuren mengt dan krijg je een rare kleur, misschien wel vies bruin of vuil grijs. Net zoals bij verf kan je met een computer verschillende kleuren mengen, niet met verf maar met lichtkleuren. Maar je mag nog steeds denken aan de verf. Stel dat we 3 potten verf hebben die 100% vol zijn en de potten bevatten rode, groene en blauwe verf. En 100% stellen we bij het programmeren gelijk aan 1 (denk aan de intrest uit de vorige hoofdstukken). We voegen al de rode verf (dus 100%) uit de pot in een vat. Dan voegen we 100% uit de groene pot toe aan het vat en we beginnen te mengen. Hieruit volgt een gele kleur (bij echte verf is dit wel anders, maar een computer weet dat niet). We gaan dit eens nader onderzoeken in Python en we gaan een cirkel tekenen:

```
>>> t.reset()
>>> t.color(1,1,0)
>>> t.begin_fill()
```

```
>>> t.circle(100)
>>> t.end_fill()
>>>
```

In de bovenstaande code vragen we in de functie `color()` 100% rood, 100% groen en 0% blauw (dus 1, 0 en 0). We willen wel wat oefenen met kleuren en veranderen onze code in een functie:

```
>>> def cirkeltje(r, g, b):
    t.color(r, g, b)
    t.begin_fill()
    t.circle(100)
    t.end_fill()

>>>
```

We willen een felgroene cirkel tekenen door alle groene kleur te gebruiken:

```
>>> cirkeltje(0, 1, 0)
>>>
```

En we kunnen een donkergroene cirkel tekenen als we bijvoorbeeld maar de helft van de groene kleur gebruiken (50% of 0.5 in programmeer-taal):

```
>>> cirkeltje(0, 0.5, 0)
>>>
```

Je ziet dat hoe minder we van een lichtkleur gebruiken, hoe donkerder deze wordt. Je kan hetzelfde eens proberen te oefenen met 100% en 50% voor de rode alsook voor de blauwe kleur. Weet je hoe het moet?

```
>>> cirkeltje(1, 0, 0)
>>> cirkeltje(0.5, 0, 0)
>>> cirkeltje(0, 0, 1)
>>> cirkeltje(0, 0, 0.5)
>>>
```

Experimenteer er maar op los. En vergeet niet dat je het venster kan proper maken met de functie `t.clear()`.

Enkele voorbeelden van kleuren:

goud	(1, 0.845, 0)
licht roze	(1, 0.70, 0.75)
oranje	(1, 0.65, 0)
bruin	(0.60, 0.30, 0.15)

10.2. Het wordt donker

Wat gebeurt er als je in een kamer zonder ramen alle lichten uitdoet? Inderdaad, het wordt donker of ook zwart. Hetzelfde gebeurt bij een computer: zonder licht heb je geen kleur, dus als alle drie de parameters nul zijn bekom je zwart:

```
>>> cirkeltje(0, 0, 0)
>>>
```

In bovenstaande code kleurt onze cirkel zwart. Maar ook het tegenovergestelde is waar. Als de drie basiskleuren allemaal 100% (of beter gezegd 1) zijn, dan krijgen we een witte cirkel (maar de achtergrond is wit, dus gaan we niks meer zien en ook de zwarte cirkel is verdwenen):

```
>>> cirkeltje(1, 1, 1)
>>>
```

10.3. Vormen vullen

Intussen zal je wel al weten dat de vulfunctie `fill()` aangezet wordt door het de parameter 1 te geven en kan afgezet worden door de parameter nul. Het is pas wanneer de functie afgezet wordt dat het vlak dat je getekend hebt, ingevuld wordt met de kleur. We kunnen gemakkelijk het vierkant tekenen dat we eerder al gemaakt hadden. Maar eerst gaan we er een functie van maken:

```
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>>
```

Als we hiervan een functie maken, gaan we de grootte van het vierkant erin stoppen als parameter. Daardoor wordt onze functie meer flexibel in te zetten:

```
>>> def vierkant(grootte):
    t.forward(grootte)
    t.left(90)
    t.forward(grootte)
    t.left(90)
    t.forward(grootte)
    t.left(90)
    t.forward(grootte)
    t.left(90)

>>> vierkant(100)
>>>
```

Nadat we de functie opgeroepen hebben bekomen we een vierkant getekend door onze schildpad. Dit is reeds een start, maar we willen onze code wel wat beter maken. Als je naar de code kijkt, dan zie je een terugkerend patroon: we herhalen enkele keren `forward(grootte)` en `left(90)`. Vervelend om dit 4 keer te moeten typen. Dit kan beter, dus gebruiken we een for-loop om dit voor ons te doen:

```
>>> def vierkant(grootte):
    for i in range(0,4):
```



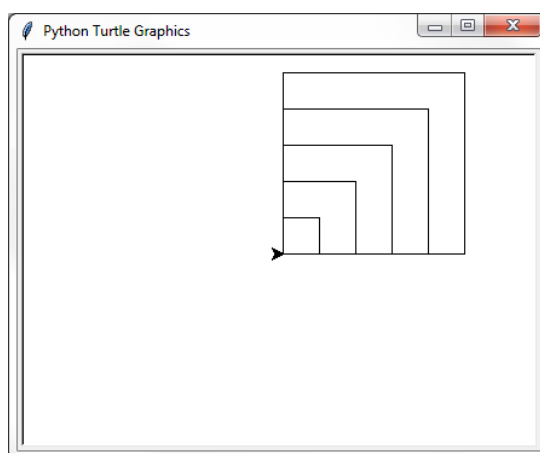
```

        t.forward(grootte)
        t.left(90)

>>> t.reset()
>>> vierkant(30)
>>> vierkant(60)
>>> vierkant(90)
>>> vierkant(120)
>>> vierkant(150)
>>>

```

Dit is heel wat beter en veel mooiere code. Dan resetten we het scherm van onze schildpad en we laten ze enkele vierkanten met verschillende groottes tekenen zoals in het bovenstaande voorbeeld. Daardoor bekommen we Figuur 21.



Figuur 21: Allemaal vierkanten

We kunnen nu een gevuld vierkant uitproberen. Maar we gaan ons venster eerst resetten:

```

>>> t.reset()
>>>

```

Daarna gaan we de vulfunctie oproepen & starten en vervolgens een nieuw vierkant tekenen:

```

>>> t.begin_fill()
>>> vierkant(90)
>>>

```

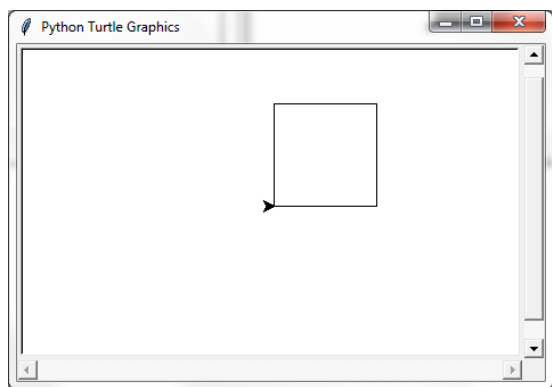
We zien nu wel een vierkant, maar het is nog leeg (zie Figuur 22), tot we de vulfunctie afzetten:

```

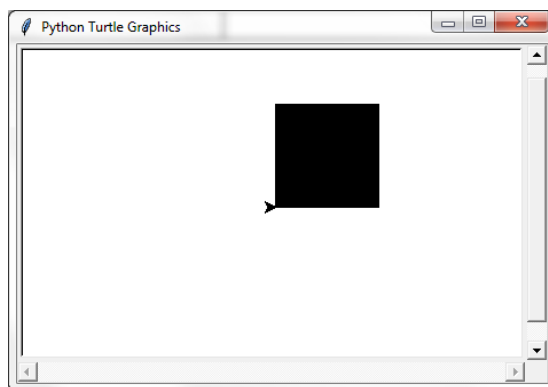
>>> t.end_fill()
>>>

```

Dan wordt het vierkant gevuld met de zwarte kleur zoals te zien is in Figuur 23.



Figuur 22: Leeg vierkant



Figuur 23: Gevuld vierkant

Niet slecht, maar we kunnen de functie ook zodanig aanpassen dat we kunnen kiezen of een leeg ofwel een gevuld vierkant getekend wordt. Dan hebben we nog een bijkomende parameter nodig en een beetje meer ingewikkelde code om dit te doen:

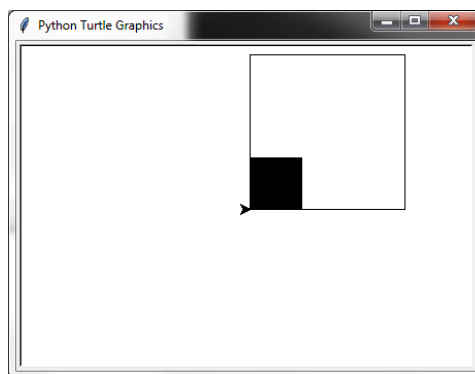
```
>>> def vierkant(grootte, gevuld):
    if gevuld == True:
        t.begin_fill()
    for i in range(0,4):
        t.forward(grootte)
        t.left(90)
    if gevuld == True:
        t.end_fill()

>>>
```

Eerst wordt gekeken of voor de parameter **gevuld** Waar (True) is ingevuld. Als dat zo is, dan wordt de vulfunctie aanzet. Daarna wordt de lus 4 keer doorlopen om het vierkant te tekenen. Op het einde van de functie wordt een 2^e keer gekeken of voor de parameter **gevuld** Waar (True) is ingevuld. En als dit zo is, dan wordt de vulfunctie afgezet. We kunnen eerst een klein gevuld vierkant tekenen en daarna een groter ongevuld vierkant:

```
>>> vierkant(50, True)
>>> vierkant(150, False)
>>>
```

Het resultaat zien we in Figuur 24. Het ziet er een beetje uit als een raar vierkant oog.



Figuur 24: Raar vierkant oog

Het moeten niet alleen vierkanten zijn die we tekenen en vullen met een kleur. Laat ons nog eens terugkeren naar de sterren. De originele code zag er zo uit:

```
>>> t.reset()
>>> for i in range(1,19):
        t.forward(100)
        if i % 2 == 0:
            t.left(175)
        else:
            t.left(225)
>>>
```

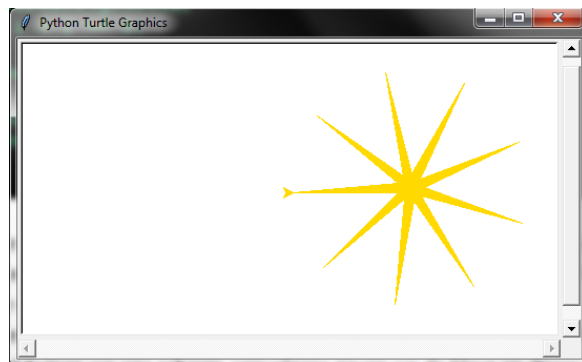
We kunnen hetzelfde if-statement als uit de functie voor het vierkant gebruiken met de grootte-parameter in de forward-functie:

```
1. >>> def ster(grootte, gevuld):
2.         if gevuld == True:
3.             t.begin_fill()
4.         for i in range(1,19):
5.             t.forward(grootte)
6.             if i % 2 == 0:
7.                 t.left(175)
8.             else:
9.                 t.left(225)
10.        if gevuld == True:
11.            t.end_fill()
>>>
```

In regels 2 en 3 wordt de vulfunctie aangezet als de waarde van de parameter Waar (True) is. In de regels 10 en 11 wordt de vulfunctie terug uitgezet als deze waarde Waar is. In regel 5 wordt de grootte van de ster bepaald, naargelang wat we ingeven nadien.

We gaan een gouden (in het Engels *gold*) ster maken met de kleuren die we eerder al zagen: 100% rood, 84,5% groen en 0% blauw. Eerst kuisen we ons scherm op, vervolgens geven we de kleur in en daarna roepen we de functie op:

```
>>> t.reset()
>>> t.color(1, 0.845, 0)
>>> ster(100, True)
>>>
```

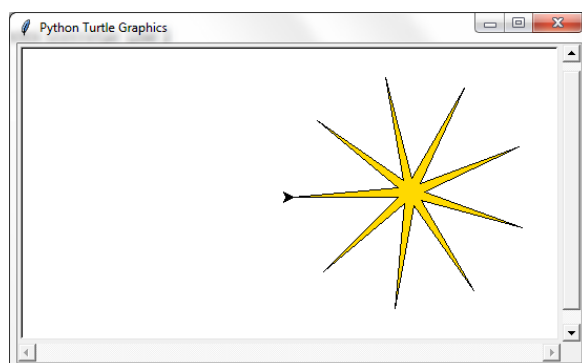


Figuur 25: Een gouden ster

We zien dat we een gouden ster gemaakt hebben in Figuur 25. We kunnen ze nog een zwart randje geven door de kleur te veranderen en ze ongevuld te laten tekenen (met de parameterwaarde False):

```
>>> t.color(0, 0, 0)
>>> ster(100, False)
>>>
```

Dan bekomen we een mooi sterretje (zie Figuur 26)



Figuur 26: Mooi sterretje

10.4. Dingen om zelf eens te doen

In dit hoofdstuk hebben we geleerd om met de schildpad-module basisvormen te tekenen. We hebben ook functies gebruikt om delen van onze code opnieuw te gebruiken en om het gemakkelijker te maken vormen te tekenen met verschillende kleuren.

En nu wat experimentjes:

Experiment 10.1:

We hebben al sterren, cirkels, vierkanten en rechthoeken getekend. Maar wat met een 6-hoek (**hint:** denk aan de graden: 360° gedeeld door 6 hoeken is 60 graden per hoek).

En wat met een achthoek? **Hint:** 360° gedeeld door 8 hoeken is 45 graden per hoek)

Experiment 10.2:

Verander de code van de achthoek nu in een functie zodat deze ook met een kleur gevuld wordt.

11. En nu grafisch

Je hebt het ongetwijfeld al gemerkt. Als je tekent met de schildpad doet deze haar reputatie van snelheid alle eer aan: ze tekent heel traag. Voor een echte schildpad is dat geen probleem, ze worden normaal heel oud, dus ze hebben alle tijd. Maar voor computertekeningen is het toch wel veel te traag. Wanneer je spelletjes speelt op je computer, DS of tablet zie je zeer veel grafische dingen (tekeningen die je op het scherm ziet).

Er zijn verschillende manieren om tekeningen in games op het scherm te brengen. Zo zijn er 2D (2-dimensionale) games waarbij je een vlakke tekening ziet en de figuurtjes op-en-neer of links-en-rechts bewegen. Er zijn ook pseudo-3D games (bijna driedimensioneel) waarbij de figuurtjes iets meer lijken zoals in het echt maar waar ze terug enkel op-en-neer alsook links-en-rechts kunnen bewegen. En uiteindelijk zijn er ook 3D-games waarbij de figuurtjes net bewegen zoals in het echt (ook vooruit-en-achteruit).

Maar al deze tekeningen hebben iets gemeen: ze moeten heel snel op het scherm getekend kunnen worden. Zo snel dat je met het blote oog niet kan zien dat ze getekend worden. Als je een bewegend beeld (een animatie) wil hebben, dan moet het beeld steeds opnieuw heel snel getekend worden met een kleine verschuiving. Daardoor ontstaat de illusie van beweging. Door al die licht verschillende beelden (ook frames genaamd) snel achter elkaar te projecteren bekom je een animatie. Zo worden ook tekenfilms gemaakt.

3D-tekeningen worden op een andere manier gemaakt dan 2D-tekeningen, maar de basisidee is dezelfde. Onze schildpad kunnen we dus echt niet gebruiken, daarvoor is ze veel te traag.

11.1. Snel tekenen

Elke programmeertaal heeft zijn eigen methode om op het scherm te tekenen. Sommige methoden zijn snel en andere zijn traag. De programmeurs die games maken moeten dus zorgvuldig kiezen welke programmeertaal ze gaan gebruiken.

Python heeft meerdere manieren om tekeningen te maken (zoals met onze schildpad), maar de beste methoden om te tekenen is meestal met modules en bibliotheken (in het Engels *libraries*) die niet in Python zelf zitten. Je zal al wat langer moeten programmeren om uit te zoeken hoe je zo'n complexe libraries moet installeren en gebruiken. Maar gelukkig is er een module bij Python die we kunnen gebruiken voor basistekeningen en die toch wel sneller is dan onze schildpad. Misschien moeten we ze de snelle schildpad noemen.

Deze module heet "tkinter". Een rare naam, maar dit staat voor "Tk interface". Ze kan gebruikt worden om ganse applicaties (dit is een andere naam voor programma's) te maken

of simpele tekeningen. Je zou zelfs een eenvoudige tekstverwerker kunnen maken. Laten we eens een simpele applicatie maken met een toets, door de volgende code in te geven:

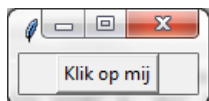
```
1. >>> from tkinter import *
2. >>> tk = Tk()
3. >>> toets = Button(tk, text="Klik op mij")
4. >>> toets.pack()
5. >>>
```


In regel 1 importeren we de inhoud van de Tk-module zodat we deze kunnen gebruiken. De best bruikbare is Tk die een basisvenster aanmaakt, waarin we dingen kunnen stoppen. Nadat je regel 2 aangemaakt hebt, komt dit venster op het scherm. In regel 3 maken we een toets aan en kennen deze toe aan de variabele `toets`. De toets wordt gemaakt door het tk-object als een parameter toe te kennen evenals de genaamde parameter `text` met de waarde `"Klik op mij"`.

Genaamde parameters:

Het is de eerste keer dat we de term genaamde parameter gebruiken. Deze werken net zoals gewone parameters, maar ze kunnen in gelijk welke volgorde verschijnen. Daarom moeten we ze een naam geven. Een voorbeeldje zal het duidelijker maken: Stel dat we een functie `rechthoek()` hebben met 2 parameters; lengte en breedte. Normaal zouden we deze functie als volgt gebruiken: `rechthoek(100, 50)` waarbij we een rechthoek willen tekenen met lengte 100 en breedte 50 pixels. Maar wat als we niet weten welke parameter eerst komt? We zouden dan lengte en breedte kunnen verwisselen en fouten maken. Dan is het beter om direct te zeggen wat gelijk is aan wat: bijvoorbeeld `rechthoek(lengte=100, breedte=50)`. Feitelijk is de idee achter genaamde parameters wat ingewikkelder en kunnen ze op verschillende manieren gebruikt worden om functies veel flexibeler te maken. Maar dit is stof voor als je al gevorderd bent in programmeren.

De vierde regel is een instructie om de toets te melden dat ie zichzelf moet tekenen op het scherm. Hierdoor verkleint het scherm uit regel 2 tot ongeveer de grootte van de toets en ziet er ongeveer uit als volgt:



Deze toets doet niet zoveel. Je kan er enkel op klikken en dan gebeurt er niks. We kunnen de toets wel een nuttige invulling geven door het vorige voorbeeld een beetje te veranderen. Maar sluit eerst het venster van de toets. Dit doe je door op de  in de rechterbovenhoek van het venster te klikken. We kunnen eerst een functie aanmaken om tekst te drukken:

```
>>> def dag():
    print('Dag allemaal')
>>>
```

Dan gaan we ons eerste voorbeeld aanpassen:

```
>>> from tkinter import *
>>> tk = Tk()
>>> toets = Button(tk, text="Klik op mij", command=dag)
>>> toets.pack()
>>>
```

De genaamde parameter `command` duidt aan dat we de functie `dag` willen gebruiken wanneer op de toets geklikt wordt. Als je klikt op de toets dan zie je het antwoord verschijnen in de Python-console:

```
>>> Dag allemaal
```

11.2. Simpel tekenen

Enkel maar een toets weergeven is niet zo handig als we iets op het scherm willen tekenen. Daarom moeten we iets anders gebruiken, namelijk een canvas (dat is iets zoals een doek om op te schilderen). Wanneer we een canvas aanmaken, moeten we de *breedte* (*width* in het Engels) en de *hoogte* (in het Engels is dit *height*) in pixels aangeven, zoals we bij het aanmaken van de toets de tekst en het commando moesten bepalen:

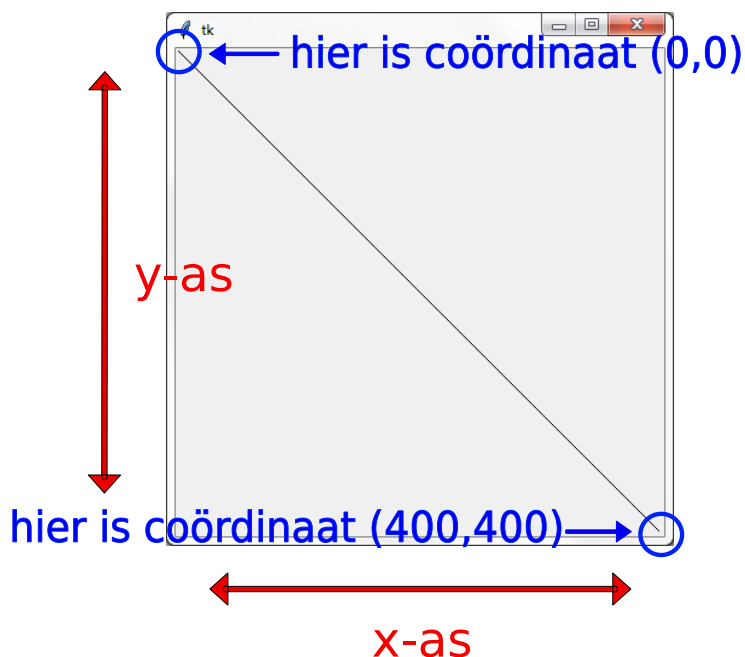
```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>>
```

Net zoals bij het voorbeeld met de toets, verschijnt na het ingeven van regel 2 een nieuw venster op het scherm. In regel 4 vergroot het scherm plotseling. In het canvas kunnen we een lijn tekenen door pixelcoördinaten te gebruiken. Coördinaten zijn niets anders dan 2 getallen om een plaats te bepalen in een vlak. Hiermee bepaal je de plaats van de pixels op het doek. Als je verder leest en de oefening maakt zal het je wel duidelijker worden. Op een Tk-canvas geven de coördinaten aan hoe ver van links naar rechts en van boven naar onder bewogen kan worden. Alles wat links-rechts gaat noemen we de x-as en alles wat boven-onder gaat, noemen we de y-as.

We hebben hierboven een canvas gemaakt van 400 pixels breed en 400 pixels hoog. Daarom zijn de coördinaten van de rechterbenedenhoek van het canvas op het scherm (400, 400). De linkerbovenhoek is de referentie van ons systeem en heeft de coördinaten (0, 0). We tekenen nu een lijn diagonaal door het canvas. De lijn start dus bij de coördinaten (0, 0) en eindigt bij de coördinaten (400, 400) zoals je ziet in Figuur 27. De code hiervoor is:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_line(0, 0, 400, 400)
1
```


>>>



Figuur 27: Canvas met de x-as en y-as

Als we dit met onze schildpad hadden moeten doen, dan zou volgende code moeten gebruikt worden:

```
>>> import turtle
>>> turtle.setup(width= 400, height= 400)
>>> t = turtle.Pen()
>>> t.up()
>>> t.goto(-200, 200)
>>> t.down()
>>> t.goto(400, -400)
>>>
```

Hierbij zie je wel dat de tkinter code reeds een verbetering is. Ze is korter en duidelijker. Er zijn zeer veel methodes mogelijk met het canvas-object. Sommigen zijn voor ons niet direct nuttig, maar anderen dan weer wel. Laten we eens kijken naar enkele nuttige functies in de volgende paragrafen.

11.3. Vierkantjes tekenen

Toen we met de schildpad tekenden, maakten we een vierkant door vooruit te gaan, te draaien, enz. Je moest toch wel wat opletten. Met tkinter is een vierkant of een rechthoek tekenen wel wat simpeler; je moet enkel de coördinaten van de hoeken kennen:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
```

```
>>> canvas.create_rectangle(50, 50, 100, 100)
1
>>>
```

Bemerk dat een vierkant eigenlijk een speciale rechthoek is, daarom gebruikt Python enkel *rechthoek* (in het Engels *rectangle*). In het bovenstaande voorbeeld hebben we een canvas van 400 pixels breed bij 400 pixels hoog gemaakt. Daarin hebben we vervolgens een vierkant getekend met de links-bovenste hoek op 50 pixels van de linkerbovenhoek van het canvas. De rechterbenedenhoek ligt op 100 pixels van de links-bovenste hoek en dit zowel op de x-as (de breedte) als op de y-as (de hoogte). De parameters die ingegeven moeten worden bij **create_rectangle** zijn daarom: bovenste-linker x-plaats, bovenste-linker y-plaats, onderste-rechter x-plaats, onderste-rechter y-plaats. Dit is echt wel veel typwerk, daarom maken we het onszelf wat gemakkelijker (we zijn een beetje lui, nietwaar) en gebruiken we x1, y1 en x2, y2.

Je hebt misschien gemerkt dat na **create_rectangle** en voorheen bij **create_line** er een getal verscheen in Python? Dit is een identificatienummer voor de vorm die je net getekend hebt. Het speelt geen rol of het een lijn, een rechthoek, een cirkel of wat dan ook is. We komen later nog terug op dat getal.

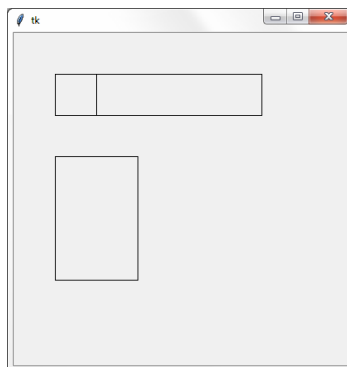
We kunnen in plaats van een vierkant een rechthoek tekenen door x2 groter te nemen:

```
>>> canvas.create_rectangle(50, 50, 300, 100)
2
>>>
```

Of we kunnen ook y2 groter nemen (en eventueel ook y1 veranderen):

```
>>> canvas.create_rectangle(50, 150, 150, 300)
3
>>>
```

Deze laatste rechthoek wordt getekend door eerst 50 pixels naar rechts te gaan van het begin van ons canvas en dan 150 pixels naar beneden. Dit is de linkerbovenhoek. Dan wordt de rechthoek getekend met als tegenoverliggende hoek x2=150 en y2=300. Voor de 3 bovenstaande rechthoeken bekom je een tekening zoals in Figuur 28.



Figuur 28: Rechthoeken met tkinter

En als we nu eens wat experimenteerden? We kunnen ook ons canvas met een heleboel willekeurige rechthoeken invullen. Hiervoor gebruiken we de module **random** (*random* is Engels voor *willekeur*). Hiervoor moeten we eerst deze module importeren.

```
>>> import random
>>>
```

Dan kunnen we een functie maken die een willekeurig getal weergeeft voor de coördinaten van de bovenste en onderste hoeken van onze rechthoeken. De functie die we hiervoor gebruiken is **randrange()** (wat neerkomt op het bereik tot waar het willekeurig getal mag gaan):

```
1. >>> tk = Tk()
2. >>> canvas = Canvas(tk, width=400, height=400)
3. >>> canvas.pack()
4. >>> def will_rechthoek(breedte, hoogte):
5.     x1 = random.randrange(breedte)
6.     y1 = random.randrange(hoogte)
7.     x2 = random.randrange(x1 + random.randrange(breedte))
8.     y2 = random.randrange(y1 + random.randrange(hoogte))
9.     canvas.create_rectangle(x1, y1, x2, y2)
10.
11. >>>
```

In de twee regels op plaats 5 en 6 creëren we de variabelen voor de linkerbovenhoek van de rechthoek met de functie **randrange()** met de waarden voor de breedte en de hoogte. De **randrange()** functie neemt een getal als een argument (bekijk ook Bijlage C: Modules voor meer uitleg over **randrange()**). Dus **randrange(9)** geeft je een getal tussen 0 en 8 (denk eraan, het laatste nummer in een bereik wordt niet gebruikt, het is 0 tot 9; niet 0 tot en met 9), **randrange(101)** geeft je een getal tussen 0 en 100.

In regels 7 en 8 maken we variabelen voor de rechteronderhoek van de rechthoek. We gebruiken de coördinaten van de linkerbovenhoek (x1 of y1) en tellen er een willekeurig getal bij op (tussen 0 en de breedte & tussen 0 en de hoogte).

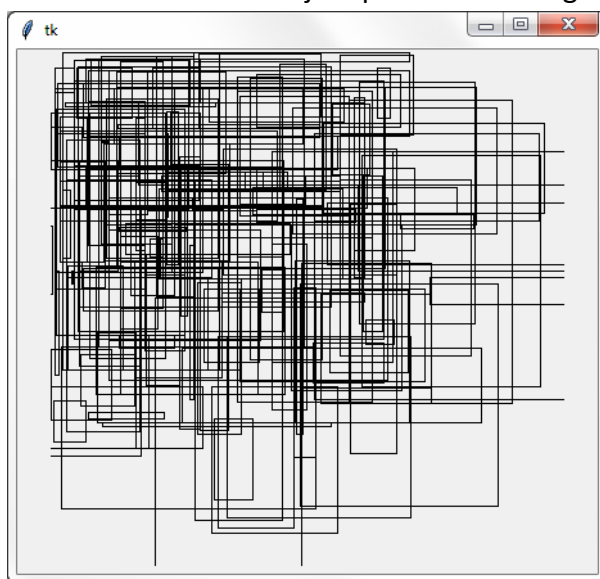
In regel 9 roepen we de `create_rectangle()` functie op met de variabelen die we net gemaakt hebben. Je kan onze functie `will_rechthoek()` die we net gemaakt hebben eens uitproberen met de breedte en de hoogte van het canvas:

```
>>> will_rechthoek(400, 400)
>>>
```

We kunnen ook het venster vullen met rechthoeken door een for-lus te gebruiken en die 150 keer te doorlopen:

```
>>> for i in range(0,150):
    will_rechthoek(300, 300)
>>>
```

Dan bekom je een venster dat misschien wat lijkt op de chaos in Figuur 29:



Figuur 29: Heleboel rechthoeken

Weet je nog dat we in het vorige hoofdstuk de kleur die de schildpad tekende bepaalden door de percentages van de 3 basiskleuren rood, groen en blauw te nemen? Bij tkinter kan je de kleuren gelijkaardig instellen, maar het is jammer genoeg wel iets moeilijker dan met onze schildpad. Laat ons eerst eens de willekeurige rechthoek functie `will_rechthoek()` veranderen zodat we een kleur aan de rechthoek kunnen toevoegen:

```
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> def will_rechthoek(breedte, hoogte, vul_kleur):
    x1 = random.randrange(breedte)
    y1 = random.randrange(hoogte)
    x2 = random.randrange(x1 + random.randrange(breedte))
    y2 = random.randrange(y1 + random.randrange(hoogte))
    canvas.create_rectangle(x1, y1, x2, y2, fill=vul_kleur)
>>>
```

De canvas functie `create_rectangle()` kan ook een parameter `fill` bevatten waarmee we de kleur aangeven waarmee de rechthoek gevuld dient te worden. Enkele van de kleuren die kunnen gebruikt worden vind je terug in Tabel 4 en gaan we eens uitproberen:

Nederlands	Waarde voor <code>fill</code>	Hex-code	RGB-code
rood	red	#FF0000	255, 0, 0
groen	green	#008000	0, 128, 0
blauw	blue	#0000FF	0, 0, 255
oranje	orange	#FFA500	255, 165, 0
violet	violet	#EE82EE	238, 130, 238
geel	yellow	#FFFF00	255, 255, 0
purper	purple	#800080	128, 0, 128
roze	pink	#FFC0CB	255, 192, 203
magenta (rood-paars)	magenta	#FF00FF	255, 0, 255
cyaan (groen-blauw)	cyan	#00FFFF	0, 255, 255
gold	goud	#FFD700	255, 215, 0

Tabel 4: kleuren met Hex-codes en RGB-codes

```
>>> will_rechthoek(400,400,'red')
>>> will_rechthoek(400,400,'green')
>>> will_rechthoek(400,400,'blue')
>>> will_rechthoek(400,400,'orange')
>>> will_rechthoek(400,400,'violet')
>>> will_rechthoek(400,400,'yellow')
>>> will_rechthoek(400,400,'purple')
>>> will_rechthoek(400,400,'pink')
>>> will_rechthoek(400,400,'magenta')
>>> will_rechthoek(400,400,'cyan')
>>> will_rechthoek(400,400,'gold')
>>>
```

Waarschijnlijk zullen al deze kleurbenamingen wel werken, maar sommige kunnen misschien een foutboodschap opleveren (dit is afhankelijk van met welk besturingssysteem je computer werkt). Dan kunnen we beter met een algemeen bruikbare methode werken die door alle besturingssystemen begrepen wordt. Want als we nu, zoals in het vorige hoofdstuk, goud willen gebruiken? Bij de schildpad maakten we de gouden kleur door 100% rood, 84,5% groen en geen blauw te gebruiken. Nu zouden we bij tkinter 'gold' kunnen gebruiken maar het is eigenlijk beter om hexadecimale code te gebruiken.

Met hexadecimale of hex-code hebben we veel meer mogelijkheden. Om te weten welke code met welke kleur overeenkomt moet je in een zoekrobot op internet maar eens "hex color" ingeven. Dan bekom je voldoende sites waar je veel meer kleuren vindt dan in Tabel 4. Om een gouden rechthoek in tkinter te maken kunnen we dus ook het volgende ingeven:

```
>>> will_rechthoek(400,400,'#FFD700')
>>>
```

Dit lijkt nogal raar om een kleur te maken, maar geloof me, het is de manier die het meest juist is en ook het meest gebruikt wordt. Die hex-code zoals '#FFD700' is een andere manier dan we gewoon zijn om getallen in te geven. Het bespreken van hoe we moeten werken met hexadecimale getallen zou te ver leiden in dit boek. Als je er meer over wil weten kan je gerust eens op Wikipedia kijken. Als je wil weten welke hex-code gebruikt wordt voor welke kleur, zoals bij de schildpad, kan je zelf ook een programma maken in Python. Natuurlijk, wat had je gedacht?

```
>>> def hex_kleur(rood, groen, blauw):
    rood = 255*(rood/100.0)
    groen = 255*(groen/100.0)
    blauw = 255*(blauw/100.0)
    return '%02x%02x%02x' % (rood, groen, blauw)

>>>
```

Als we de hex-code willen weten voor 100% rood, 84,5% groen en 0% blauw dan moeten we volgend commando ingeven en we bekommen direct het resultaat:

```
>>> print(hex_kleur(100, 84.5, 0))
#ffd700
>>>
```

Als je toch wat meer wil weten: voor goud hebben we 84,5% groen nodig. In Tabel 4 zie je bij de **G** 215 staan. Het maximum dat bij een kleur voorkomt is 255. Als we 84,5% van 255 nemen bekommen we 215. In hexadecimale code is 255 gelijk aan FF en 215 is gelijk aan D7. Dikwijls wordt in hex-code op het einde een kleine “h” of “16” geschreven om aan te duiden dat het hexadecimaal is, bijvoorbeeld D7_h of D7₁₆.

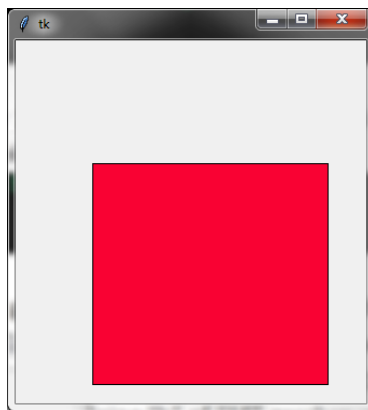
We kunnen ook een soort rode kleur maken met 98% rood, 1% groen en 20% blauw:

```
>>> print(hex_kleur(98, 1, 20))
#f902c4
>>>
```

We kunnen dit gebruiken bij de functie `will_rechthoek()` die we eerder gebruikten:

```
>>> will_rechthoek(400,400, hex_kleur(98, 1, 20))
>>>
```

En dit resulteert in de volgende figuur:



Figuur 30: een rode rechthoek

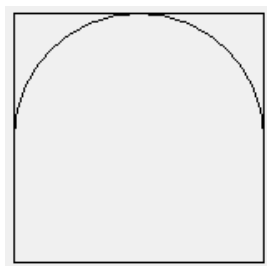
11.4. Bogen tekenen

Een *boog* (in het Engels *arc*) is een onderdeel van een cirkel of een ellips, maar om een boog te tekenen met tkinter moet je een rechthoek tekenen. Raar, hé. Maar het wordt je duidelijker als je probeert een rechthoek te tekenen en daarin een boog. De code om de boog te tekenen kan op onderstaande lijken:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(50, 50, 200, 200, extent=180, style=ARC)
1
>>>
```

De coördinaten van `create_arc` plaatsen een rechthoek met linkerbovenhoek op $x_1=50$ & $y_1=50$. Dit betekent, zoals je nog weet, dat er van de linkerbovenhoek van het canvas 50 pixels naar rechts geschoven wordt, en 50 pixels naar onder om deze hoek van de rechthoek te bekomen. De rechteronderhoek staat op coördinaten $x_2=200$ en $y_2=200$. De volgende parameter is terug een genaamde parameter `extent`, die gebruikt wordt om de graden van de hoek van de boog te bepalen. Hier hebben we 180 graden genomen en dat kan je zien in de Figuur 31. Om het gemakkelijk te zien heb ik er ook een rechthoek bijgetekend met de onderstaande code:

```
>>> canvas.create_rectangle(50, 50, 200, 200)
2
>>>
```



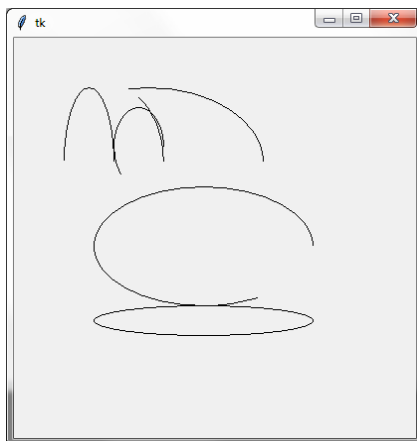
Figuur 31: Een boog die past in het vierkant

Zoals je ziet is onze boog eigenlijk een halve cirkel. Dit komt omdat de rechthoek feitelijk een vierkant is en het aantal graden 180 bedraagt. Als we de genaamde parameter **extent** op 90 zouden gezet hebben, dan bekwamen we een kwart cirkel. Indien deze 359 zou zijn dan is er een volledige cirkel getekend. Neen, we mogen geen 360 graden nemen, want dat is gelijk aan 0 graden en dan zien we niks op het scherm verschijnen.

We gaan nu wat verschillende bogen tekenen zodat het allemaal duidelijker wordt:

```
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(50, 50, 100, 200, extent=180, style=ARC)
1
>>> canvas.create_arc(20, 50, 250, 200, extent=100, style=ARC)
2
>>> canvas.create_arc(50, 50, 150, 200, extent=60, style=ARC)
3
>>> canvas.create_arc(100, 70, 150, 150, extent=225, style=ARC)
4
>>> canvas.create_arc(300, 150, 80, 270, extent=300, style=ARC)
5
>>> canvas.create_arc(300, 300, 80, 270, extent=359, style=ARC)
6
>>>
```

Het resultaat van bovenstaande zie je in Figuur 32.



Figuur 32: Verschillende bogen

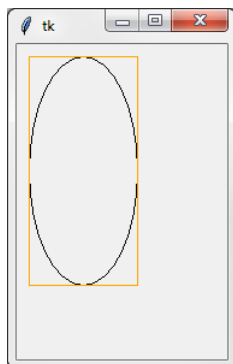
11.5. Ovalen tekenen

Eerst iets duidelijk maken: een ellips is een ovaal, maar een ovaal hoeft geen ellips te zijn. Dit is net zoals een vierkant een rechthoek is, maar een rechthoek geen vierkant moet zijn.

Je hebt misschien gemerkt dat in de laatste opdracht hierboven je een ovaal getekend hebt. Je kan ook ovalen tekenen met de `create_oval()` functie. Net zoals bij bogen wordt een ovaal ook binnen een rechthoek getekend. Bijvoorbeeld:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_oval(10, 10, 100, 200)
1
>>> canvas.create_rectangle(10, 10, 100, 200, outline='#FFA500')
2
>>>
```

We vinden in Figuur 33 onze ovaal terug in een oranje rechthoek (met de genaamde parameter `outline`).



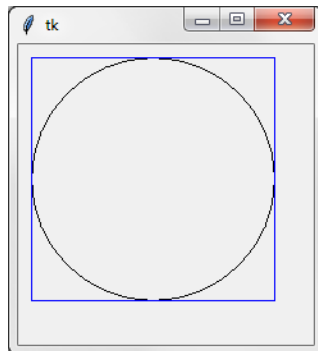
Figuur 33: Een ovaal in een rechthoek

Maar we kunnen ook een speciaal geval maken en een cirkel aanmaken. Daarvoor hebben we de coördinaten van een vierkant nodig, dat we blauw kleuren:

```
>>> tk = Tk()
```

```
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_oval(10, 10, 200, 200)
1
>>> canvas.create_rectangle(10, 10, 200, 200, outline='#0000FF')
2
>>>
```

En dit vinden we terug in Figuur 34.



Figuur 34: Een cirkel in een vierkant

11.6. Veelhoeken tekenen

Een veelhoek (soms ook polygoon genaamd) is elke vorm met 3 of meer zijden: driehoeken, vierhoeken, vijfhoeken, zeshoeken en ga zo maar een tijdje door. Dat zijn allemaal voorbeelden van polygonen. Er zijn veelhoeken met regelmatige vormen. Maar je kan ook veelhoeken maken met onregelmatige vormen. We zullen die aan de hand van een voorbeeldje laten zien. We willen een driehoek tekenen. Daarvoor zijn 3 sets coördinaten nodig (een x-plaats en een y-plaats) voor elk punt van de driehoek die we blauw maken:

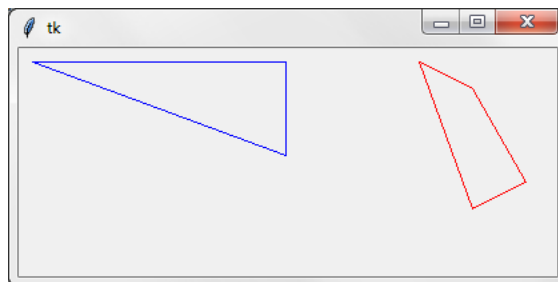
```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 200, 10, 200, 80, fill='',
outline='#0000FF')
1
>>>
```

Bemerk dat de breedte van ons boek te smal is om de regel waarin de polygoon aangemaakt wordt achtereen te schrijven. Je moet dus gewoon verder typen.

Dan gaan we een onregelmatige rode polygoon aanmaken met volgende code:

```
>>> canvas.create_polygon(300, 10, 340, 30, 380, 100, 340, 120,
fill='', outline='#FF0000')
2
>>>
```

Beide bovenstaande veelvormen vind je terug in Figuur 35.



Figuur 35: Verschillende polygonen

11.7. Tekeningen maken

Je kan een beeldje op het canvas tekenen met tkinter door eerst het beeld in te laden en daarna de `create_image()` functie op het canvas-object te gebruiken. Zet eerst een beeldje in gif-formaat in de python-directory. Ik heb gekozen voor het beeldje “Tux_and_friends.gif”, maar je kan eender welke foto kiezen. Dat klinkt allemaal raar, maar het werkt als volgt:

```
1. >>> from tkinter import *
2. >>> tk = Tk()
3. >>> canvas = Canvas(tk, width=400, height=400)
4. >>> canvas.pack()
5. >>> beeldje = PhotoImage(file='Tux_and_friends.gif')
6. >>> canvas.create_image(0, 0, image=beeldje, anchor=NW)
7. 1
8. >>>
```

En daaruit bekomen we Figuur 36:



Figuur 36: Het ingeladen beeldje

In regels 1 tot en met 4 zetten we ons canvas op zoals we voorheen ook deden. In regel 5 laden we onze foto in de variabele `beeldje`. Het is belangrijk dat je het beeld dat je wil inladen staat in een directory waar Python aankan. Meestal is dat de directory waarin Python werkt. Als je het niet weet, dan kan je nog volgend commando uitvoeren:

```
>>> import os
>>> print (os.getcwd())
```

```
/home/Johan
>>>
```

Dan bekom je iets zoals boven, maar waarschijnlijk met jouw naam in plaats van Johan. Als je Siegrid heet en je besturingssysteem is zo geconfigureerd dan is je werkdirectory waarschijnlijk /home/Siegrid. Zorg dus dat je beeldje dan daar staat.

In regel 6 gebruiken we de functie `create_image()` op het canvas om het beeld op het scherm te plaatsen.


Met de functie `PhotoImage` kan je slechts een beperkt aantal types beeldjes aan (met extensie GIF en nog enkele andere), maar JPG is er niet bij. Jouw camera neemt beelden op in JPG, dus zou je kunnen denken dat deze niet bruikbaar zijn, maar niet getreurd, er is een oplossing: er bestaat een extensie die deze mogelijkheid aan Python toevoegt. Dit is de Python Imaging Library (PIL), waarmee je veel formaten kan inlezen, maar ook uitrekken, inkrimpen, kleuren veranderen, en nog een heleboel meer. Maar het installeren en gebruiken van PIL is een beetje een brug te ver om nu reeds te bekijken. Je kan wel meer info rond PIL vinden door een zoektocht op internet met als zoekterm “Python Imaging Library”.

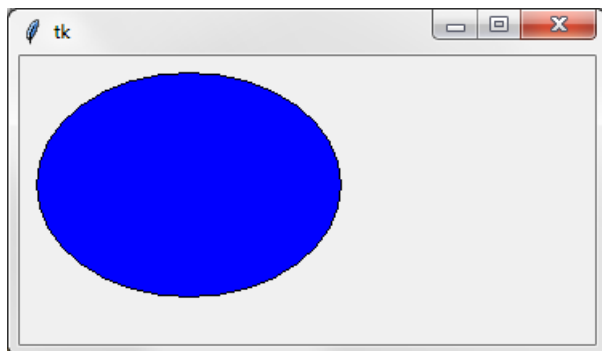
11.8. Basis animaties

Tot nu toe hebben we enkel niet-bewegende beelden gemaakt, dit worden ook statische beelden genoemd (ze staan daar maar te staan). Maar wat met bewegende beelden of animaties? Onze Tk is niet heel sterk in animaties, maar je kan er wel basishandelingen mee uitvoeren. Zo kunnen we een blauwe ovaal maken en die over het scherm laten bewegen met onderstaande code:

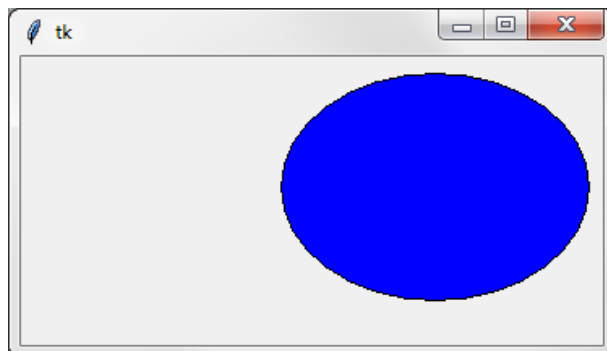
```
1. >>> import time
2. >>> from tkinter import *
3. >>> tk = Tk()
4. >>> canvas = Canvas(tk, width=400, height=400)
5. >>> canvas.pack()
6. >>> canvas.create_oval(10, 10, 200, 150, fill='#0000FF')
7. 1
8. >>> for i in range (0, 50):
9.     canvas.move(1, 3, 0)
10.    tk.update()
11.    time.sleep(0.02)

>>>
```

Zodra je op het einde op de -toets drukt begint de blauwe ovaal naar rechts te bewegen. Het verschil tussen start en einde zie je in Figuur 37 en Figuur 38.



Figuur 37: Start beweging



Figuur 38: Einde beweging

Hoe werkt dit nu? Regels 1 tot en met 5 hoef ik niet meer uit te leggen want dat is de basis om een canvas op te maken. In regel 6 wordt de blauwe ovaal gemaakt, met in regel 7 zijn identiteit (het getal "1") dat Python teruggeeft na ingave van deze functie. In regel 8 maken we een lus tussen getallen 0 en 49. Het blok dat daarop volgt (regels 9 tot en met 11) is de code die onze ovaal doet *bewegen* (in het Engels *move*).

De functie `canvas.move()` doet een getekende vorm op het canvas bewegen door bij de x- en y-coördinaten een waarde op te tellen. We hebben het er reeds voorheen over gehad dat de identiteit soms nodig is om te kennen. Wel, hier is dit het geval. Met de functie `canvas.move(1, 3, 0)` in regel 9 doen we het object met identiteit 1 (onze blauwe ovaal) telkens verschuiven over de x-as (horizontaal) met 3 pixels en over de y-as (vertikaal) met nul pixels (geen verticale verschuiving dus). Als we de ovaal dan in de tegengestelde richting zouden willen terugschuiven, dan gebruiken we een negatieve waarde `canvas.move(1, -3, 0)`.

De functie `update()` op het `tk`-object, `tk.update()`, maakt dat er een update gebeurt van de beweging. Als we deze functie niet zouden gebruiken, dan zou de lus eindigen zonder dat de ovaal bewoog.

In regel 11 vertellen we tegen Python dat na elke stap van de lus $1/50^{\text{ste}}$ van een seconde of 0,02 seconden gewacht moet worden (`sleep` is Engels voor *slapen*) alvorens de volgende stap te nemen. Als we dit niet zouden doen en we startten onze code, dan zouden we direct het einde van de beweging zien omdat de computer heel snel rekent. Hoe groter het getal dat je hier neemt hoe trager de beweging gaat, want hoe langer de computer moet slapen.

We kunnen de blauwe ovaal ook diagonaal naar rechts beneden laten bewegen. We laten telkens de x-plaats met 3 vermeerderen en de y-plaats met 4 in `canvas.move(1, 3, 4)`. Hiervoor passen we onze code aan:

```
>>> import time
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_oval(10, 10, 200, 150, fill='#0000FF')
1
```

```
>>> for i in range (0, 50):
    canvas.move(1, 3, 4)
    tk.update()
    time.sleep(0.02)

>>>
```

En we willen terug naar de startlokatie van onze ovaal, dus veranderen we de x- en y-parameters in `canvas.move(1, -3, -4)` door hun negatieve waarden:

```
>>> for i in range (0, 50):
    canvas.move(1, -3, -4)
    tk.update()
    time.sleep(0.02)

>>>
```

11.9. Actie en reactie

We kunnen zorgen dat onze blauwe ovaal reageert als iemand een actie uitvoert, zoals bijvoorbeeld op een toets drukken. Dit noemen we in het Engels *event bindings*, wat staat voor het *verbinden van gebeurtenissen* in het Nederlands. Gebeurtenissen kunnen voorkomen terwijl het programma loopt, zoals het indrukken van een toets, een beweging met de muis, het sluiten van een venster. We kunnen Tk zodanig instellen dat het uitkijkt naar zo'n events en eventueel reageert als zo een event gebeurt. Om gebeurtenissen te behandelen moeten we beginnen met een functie aan te maken. We kunnen onderstaande functie aanmaken om onze ovaal te bewegen:



```
>>> def beweeg_ovaal(event):
    canvas.move(1, 3, 0)

>>>
```




De functie heeft slechts één parameter nodig (**event**) die door Tk gebruikt wordt om info naar de functie te sturen over wat er gebeurd is. Dan vertellen we Tk dat deze functie dient voor een specifiek event, door de `bind_all()` functie op het canvas. De volledige code ziet er dan zo uit:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_oval(10,10,200,150,fill='#0000FF')
1
>>> def beweeg_ovaal(event):
    canvas.move(1, 3, 0)

>>> canvas.bind_all('<KeyPress-Return>', beweeg_ovaal)
'46595968beweeg_ovaal'
>>>
```

Bij de `bind_all()` functie beschrijft de eerste parameter het event waar Tk naar moet kijken. In ons geval moet Tk kijken naar het *indrukken* (in het Engels *press key* of in de code `KeyPress`) van de -toets (`<KeyPress-Return>`). We vertellen aan Tk dat de functie `beweeg_ovaal()` moet uitgevoerd worden als deze toetsaanslag gebeurt. Als je de code uitvoert moet je naar het Tk-canvas gaan door er met de muis op te klikken en dan moet je eens proberen op de -toets te drukken (meerdere keren mag ook) op je toetsenbord.

We kunnen ook wat spelen met onze ovaal en deze verschillende richtingen laten uitgaan afhankelijk van welke toets we indrukken. Maar misschien eerst enkele vertalingen zodat je weet welke toets waarvoor dient:

Toets	Engels	Nederlands
	Up	Omhoog
	Down	Omlaag
	Left	Links
	Right	Rechts

Tabel 5: Enkele toetsen

We moeten eerst de functie `beweeg_ovaal()` veranderen naar het onderstaande:

```
>>> def beweeg_ovaal(event):
    if event.keysym == 'Up':
        canvas.move(1, 0, -3)
    elif event.keysym == 'Down':
        canvas.move(1, 0, 3)
    elif event.keysym == 'Left':
        canvas.move(1, -3, 0)
    else:
        canvas.move(1, 3, 0)

>>>
```

Het event-object dat aan `beweeg_ovaal` doorgegeven wordt, bevat een aantal properties (Engels voor *eigenschappen*). Properties zijn genaamde waarden die iets beschrijven, zoals bijvoorbeeld het gras is groen, of modder is vuil. Properties bij het programmeren hebben steeds een naam en een waarde (vandaar een genaamde waarde). Eén van deze properties is `keysym`, wat een string is die de waarde van de ingedrukte toets heeft. Als `keysym` de string 'Up' heeft, dan roepen we `canvas.move` op met de parameters (1, 0, -3); bij de string 'Down' worden de parameters (1, 0, 3) doorgegeven, enzovoort. Herinner jezelf dat de eerste parameterwaarde de identiteit is van de vorm die we getekend hebben. Dus enkel die vorm zal de actie uitvoeren. De 2^e parameter is de waarde die bij de x-positie (horizontaal) moet geteld worden en de 3^e parameter wordt bij de y-plaats (vertikaal) geteld.

Dan gaan we Tk vertellen dat de functie `beweeg_ovaal()` moet gebruik maken van de events die bij deze 4 toetsen uit Tabel 5 horen:

```

>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_oval(10,10,200,150,fill='#0000FF')
1
>>> def beweeg_ovaal(event):
    if event.keysym == 'Up':
        canvas.move(1, 0, -3)
    elif event.keysym == 'Down':
        canvas.move(1, 0, 3)
    elif event.keysym == 'Left':
        canvas.move(1, -3, 0)
    else:
        canvas.move(1, 3, 0)

>>> canvas.bind_all('<KeyPress-Up>', beweeg_ovaal)
'5741464beweeg_ovaal'
>>> canvas.bind_all('<KeyPress-Down>', beweeg_ovaal)
'47082392beweeg_ovaal'
>>> canvas.bind_all('<KeyPress-Left>', beweeg_ovaal)
'47082432beweeg_ovaal'
>>> canvas.bind_all('<KeyPress-Right>', beweeg_ovaal)
'48805288beweeg_ovaal'
>>>

```

Telkens je het commando `canvas.bind_all` ingegeven hebt, moet je eens klikken op het Tk-venster en de toets die beschreven is indrukken. Zo zie je de ovaal bewegen in alle richtingen.

12. Postlude

Heel fijn. Je bent tot het einde van dit boek geraakt. Ik hoop dat je wat bijgeleerd hebt over programmeren en in het bijzonder over coderen in Python. Het is heel fijn je eigen programma te kunnen maken of de programma's die anderen gemaakt hebben te begrijpen. Je hebt zeker nog een weg af te leggen alvorens je heel complexe programma's kan maken, maar ik hoop dat je met dit boek een degelijke basis gelegd hebt in het maken van code. Het vergt veel experimenteren en oefenen. Soms zal het je misschien niet goed lukken om een goed resultaat te bekomen, maar geef de moed zeker niet op. Na falen komt succes, al dan niet met de hulp van iemand anders. Als je een succes boekt is de voldoening des te groter. Je kan je programmeerkunsten ook toepassen op andere dingen dan enkel je computer, denk maar aan een robot, een lichtshow voor je muziek of een drone of iets helemaal anders. Ik wens je heel veel plezier met het maken van fijne en goede code.

Hoe moet je nu verder? Wel er is gelukkig heel wat te vinden over Python. Jammer genoeg is het meeste geschreven in het Engels. Maar als je reeds wat Engels kent (of via een vertaalprogramma, of misschien ma of pa) zijn onder andere volgende werken zeker interessant (sommige zijn betalend, andere onder Creative Commons-licentie):

Boek en/of website	Auteur
A byte of Python	Swaroop C H
Dive Into Python	Mark Pilgrim
Invent Your Own Computer Games with Python	Al Sweigart
Hacking Secret Ciphers with Python	Al Sweigart
Making Games with Python & Pygame	Al Sweigart
Learn Python The Hard Way	Zed A. Shaw
Python Programming for the Absolute Beginner	Michael Dawson
Programming Python	Mark Lutz
www.python.org	
www.pygame.org	

Er is met veel zorg getracht dit boek op te stellen. Het is nooit de ambitie geweest om volledig te zijn. Net zoals software en hardware errors bevatten kunnen ook in dit boek fouten niet vermeden worden. Indien je onjuistheden zou ontdekken kan je deze steeds melden op python.voor.kids@gmail.com.

Ik hoop dat dit boek ook anderen zal inspireren om zelf een Nederlandstalig boek voor kinderen over programmeertalen te schrijven, ofwel om een der Engelstalige werken uit bovenstaande lijst te vertalen en vervolgens via een CC-licentiemodel te verspreiden (indien opportuun). Er zijn genoeg vervolgböeken te maken, er is keuze te over aan thema's. Je hoeft het niet alleen te doen, heden zijn er genoeg tools om samenwerking te bevorderen.

13. Bijlage A: Trefwoorden

Keywords of *trefwoorden* zijn belangrijke woorden die door de programmeertaal zelf gebruikt worden. Je mag deze niet op een andere manier gebruiken, zoals bijvoorbeeld als variabele. Als je dat toch zou doen kan je rare foutboodschappen krijgen in de Python-console.

Als je Python zelf de trefwoorden wil laten weergeven, dan voer je onderstaande code uit:

```
>>> import keyword
>>> print (keyword.kwlist)
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class',
'continue', 'def', 'del', 'elif', 'else', 'except', 'finally',
'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda',
'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while',
'with', 'yield']
>>>
```

Denk eraan dat alle Python trefwoorden in kleine letters geschreven worden, behalve **None**, **True** en **False**. In Python v3.4 bestaan 33 trefwoorden.

Hieronder geven we de belangrijkste Python3 trefwoorden en een korte beschrijving. Hou er rekening mee dat Python v3 en Python v2.7 op meerdere vlakken serieus verschillen. Dit boek behandelt enkel Python v3.

and

Het trefwoord **and** wordt gebruikt om 2 uitdrukkingen aan elkaar te hangen in een opdracht (zoals bijvoorbeeld het if-statement) om te zeggen dat beiden samen moeten waar zijn.

Vb:

```
>>> if leeftijd > 9 and leeftijd < 12:
```

Hier moet de leeftijd groter zijn dan 9 en kleiner dan 12 jaar.

Stel:

A = `leeftijd > 9` en

B = `leeftijd < 12`

Onderstaande tabel geeft een waarheidstabel voor **and**:

A	B	Resultaat
True	True	True
True	False	False
False	True	False
False	False	False

Tabel 6: A and B geeft Resultaat

as

Het trefwoord **as** wordt gebruikt om een andere naam te geven aan een geïmporteerde module.

Vb:

een_Python_module_maar_minder_gemakkelijk_bruikbaar

Als je dit steeds opnieuw zou moeten typen, dan zou je veel werk hebben:

```
>>> import een_Python_module_maar_minder_gemakkelijk_bruikbaar
>>>
>>> een_Python_module_maar_minder_gemakkelijk_bruikbaar.doe_iets()
Ik heb iets gedaan
>>>
>>> een_Python_module_maar_minder_gemakkelijk_bruikbaar.doe_nog_iets()
Ik heb nog iets anders gedaan
>>>
```

Daarom is het beter een andere naam te geven als je importeert en nadien kan je die nieuwe naam gebruiken:

```
>>> import een_Python_module_maar_minder_gemakkelijk_bruikbaar as
niet_bruikbaar
>>>
>>> niet_bruikbaar.doe_iets()
Ik heb iets gedaan
>>> niet_bruikbaar.doe_nog_iets()
Ik heb nog iets anders gedaan
>>>
```

Maar waarschijnlijk zal je dit trefwoord niet heel dikwijls gebruiken.

assert

Dit is een trefwoord voor gevorderde programmeurs dat gebruikt wordt om te zeggen dat code moet True zijn. Het is een andere manier om fouten en problemen in code op te vangen en wordt in complexe programma's gebruikt.

break

Het keyword **break** wordt gebruikt om het uitvoeren code te stoppen. Je kan het bijvoorbeeld in een for-loop steken om deze te onderbreken:

```
>>> for a in range(0, 20):
    print ('Hallo %s' % a)
    if a > 9:
        break

Hallo 0
Hallo 1
Hallo 2
Hallo 3
Hallo 4
Hallo 5
Hallo 6
Hallo 7
Hallo 8
Hallo 9
Hallo 10
>>>
```

Als de variabele **a** groter is dan 9 stopt de code.

class

Het **class** keyword wordt gebruikt om een type object te definiëren. Dit wordt in veel programmeertalen gebruikt bij het maken van ingewikkelde programma's, maar het is nog een beetje te vergevorderd voor ons boek.

del

De functie **del ()** laat ons toe iets te verwijderen. Het is de afkorting van delete. Als je bijvoorbeeld een cake wil maken, maar je wil abrikozen in plaats van appelen.

Lijstje voor cake:

- eieren
- bloem
- boter
- melk
- ~~appelen~~
- abrikozen

In Python geeft dit:

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>>
```

We verwijderen appelen door **del** te gebruiken en voegen de abrikozen toe met de functie **append ()**:

```
>>> del lijstje[4]
>>> lijstje.append('abrikozen')
>>>
```

En dan bekijken we de nieuwe boodschappelijst voor onze cake:

```
>>> print (lijstje)
['eieren', 'bloem', 'boter', 'melk', 'abrikozen']
>>>
```

elif

Het trefwoord **elif** (wat staat voor else if) wordt gebruikt als deel van het if-statement. Kijk hieronder bij **if**.

else

Ook het trefwoord **else** wordt gebruikt als deel van het if-statement. Kijk hieronder bij **if**.

except

Nog een keyword dat gebruikt wordt om problemen in code op te vangen. Ook dit wordt gebruikt bij heel ingewikkelde programma's, die nog te moeilijk zijn voor de stof in dit boek.

exec

De functie `exec()` is een speciale functie die gebruikt wordt om naar een string te kijken alsof het een stuk Python-code was en dit dan uit te voeren (*execute*). Bijvoorbeeld, je kan een variabele maken met een string:

```
>>> tekstje = 'Joehoe'
>>>
```

Daarna wordt de inhoud afgedrukt op het scherm:

```
>>> print (tekstje)
Joehoe
>>>
```

Maar je zou ook wat Python-code in die string kunnen stoppen:

```
>>> tekstje = 'print("Joehoe")'
>>>
```

Waarna je de functie `exec()` kan gebruiken om deze string in een mini-Pythonprogramma te stoppen en dit uit te voeren:

```
>>> exec(tekstje)
Joehoe
>>>
```

Het is misschien wel een raar idee, maar het heeft wel degelijk zin om dit te gebruiken in meer ingewikkelde programma's.

finally

Terug hebben we hier een geavanceerd trefwoord dat, wanneer een fout in een programma optreedt, een stuk speciale code uitvoert.

for

Dit trefwoord `for` wordt gebruikt om een lus of de zogenaamde for-loop te maken.

Bijvoorbeeld:

```
>>> for i in range (3,7):
    print ('De waarde van i is %s' %i)

De waarde van i is 3
De waarde van i is 4
De waarde van i is 5
De waarde van i is 6
>>>
```

from

Wanneer we een module importeren kunnen we enkel het deel dat we nodig hebben importeren met het **from** trefwoord. Zoals je nog weet heeft de Python schildpad een module **turtle** met een functie **Pen()** die gebruikt wordt om een pen-object te maken (eigenlijk een canvas waarop de schildpad kan bewegen). Je zou de volledige **turtle** module kunnen importeren en dan de **Pen()** functie gebruiken door volgende code:

```
>>> import turtle
>>> t = turtle.Pen()
>>>
```

Of je kan enkel de **Pen()** -functie zelf importeren en dan direct gebruiken zonder naar de **turtle**-module te moeten verwijzen:

```
>>> from turtle import Pen
>>> t = Pen()
>>>
```

Dit betekent wel dat je de andere functies van de module niet kan gebruiken omdat je ze niet geïmporteerd hebt, maar het spaart geheugen uit. Kijk maar eens naar volgend voorbeeld waar we een deel van de module **time** willen gebruiken. Deze module heeft naast de functie **localtime()** ook onder andere **gmtime()**. Als we enkel **localtime()** importeren en dan **gmtime()** proberen uit te voeren, bekommen we een foutboodschap:

```
>>> from time import localtime
>>> print(localtime())
time.struct_time(tm_year=2015, tm_mon=1, tm_mday=25, tm_hour=20,
tm_min=51, tm_sec=18, tm_wday=6, tm_yday=25, tm_isdst=0)
>>> print(gmtime())
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    print(gmtime())
NameError: name 'gmtime' is not defined
>>>
```

Door **name 'gmtime' is not defined** weer te geven vertelt Python ons dat het deze functie niet kent (want ze is niet geïmporteerd).

Als er een heleboel functies in een bepaalde module zijn die je wil gebruiken en je wil niet steeds de modulenaam intypen (zoals bijvoorbeeld: **time.localtime()** en **time.gmtime()**, enz) dan kan je alles in de module importeren door een asterisk (*) te gebruiken:

```
>>> from time import *
>>> print(localtime())
time.struct_time(tm_year=2015, tm_mon=1, tm_mday=25, tm_hour=20,
tm_min=57, tm_sec=41, tm_wday=6, tm_yday=25, tm_isdst=0)
>>> print(gmtime())
time.struct_time(tm_year=2015, tm_mon=1, tm_mday=25, tm_hour=19,
tm_min=57, tm_sec=48, tm_wday=6, tm_yday=25, tm_isdst=0)
>>>
```

We hebben hier dus alles geïmporteerd van de module `time` en we kunnen naar de individuele functies verwijzen door hun naam te gebruiken.

global

In hoofdstuk 7 hebben we over *scope* gepraat. Dit is het *bereik* of de *zichtbaarheid* van een variabele. Wanneer een variabele buiten een functie gedefinieerd is, kan de variabele gewoonlijk binnen die functie gezien worden. Wanneer een variabele in een functie gedefinieerd is, kan die variabele normaal niet buiten deze functie gezien worden. Maar het trefwoord `global` verandert de hele zaak. Een variabele die als `global` gedefinieerd is, kan overal gezien worden. *Global* is Engels voor *globaal*, *wereldwijd* of *universeel*. Als je jezelf inbeeldt dat Python een mini-wereldje is voor je code, dan betekent de functie `global()` echt wereldwijd. Een voorbeeldje:

```
>>> def testje():
        global x
        x = 1
        y = 2

>>> testje()
>>> print (x)
1
>>> print (y)
Traceback (most recent call last):
  File "<pyshell#27>", line 1, in <module>
    print (y)
NameError: name 'y' is not defined
>>>
```

Het afdrukken met `print(x)` geeft geen problemen, maar met `print(y)` wel. Dit komt omdat we in module `testje` van de variabele `x` een `global` gemaakt hebben en niet van variabele `y`. Let wel op dat je van een variabele eerst een `global` moet maken vooraleer je aan deze variabele een waarde geeft. Let ook op met het gebruik van `global` in complexe programma's. Er zijn programmeurs die, zeker niet onterecht, tegen het gebruik van globale variabelen zijn. Ze moeten met veel zorg gebruikt worden.

if

Dit trefwoord wordt gebruikt bij een opdracht waarbij een beslissing moet genomen worden over iets. Soms worden ook de trefwoorden `else` en `elif` bij dit keyword gebruikt. Een if-statement zegt feitelijk het volgende: als iets waar is, doe dan deze actie. Bijvoorbeeld:

```
>>> prijs = 170
>>> if prijs > 200:
        print ('Veel te duur')
    elif prijs > 150:
        print ('Te duur')
    else:
```

```
print ('Ik koop het')
>>>
```

Dit if-statement zegt dat als de prijs groter is dan 200, dan is het veel te duur; als de prijs hoger is dan 170, dan is het nog te duur. Als de prijs gelijk of lager dan 150 is dan koop ik het. De prijs is 170, dus het is te duur.

import

het **import** trefwoord zegt Python dat een module moet ingeladen worden zodat deze gebruikt kan worden. Bijvoorbeeld:

```
>>> import time
>>>
```

Bovenstaande code zegt dat we de module **time** willen gebruiken.

in

Het keyword **in** wordt in uitdrukkingen gebruikt om te checken of een item binnen de collectie items valt. Enkele voorbeelden om te verduidelijken:

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> if 'taart' in lijstje:
    print('taart zit in het lijstje')
else:
    print('taart zit niet in het lijstje')
taart zit niet in het lijstje
>>>
```

```
>>> if 3 in [1, 2, 3, 4]
    print('Getal zit in de lijst')
Getal zit in de lijst
>>>
```

is

het keyword **is** kan je zien als iets zoals het is-gelijk-aan teken (==) dat gebruikt wordt om te checken of 2 zaken gelijk aan elkaar zijn. Bijvoorbeeld `5 == 5` is True, maar `7 == 8` is False. Maar let op, er is toch een verschil tussen **is** en **==**. Als je twee zaken vergelijkt dan kan **==** de waarde True teruggeven, maar het trefwoord **is** niet (ondanks dat je denkt dat beide hetzelfde betekenen). Dit is wel enkel voor geavanceerde gebruikers van belang. Onthoud nu enkel het gebruik van **==**.

lambda

een geavanceerd trefwoord dat we nu zeker nog niet gaan gebruiken, want het is nog veel te moeilijk.

not

Als iets True is, dan maakt het trefwoord **not** dit iets False zoals je onder kan zien in de tabel.

A	not A
True	False
False	True

Tabel 7: Waarheidstabel not

```
>>> a = True
>>> print (not a)
False
>>>
```

Dit lijkt niet echt nuttig, tot je denkt aan **not** in if-statements.

Stel dat je een game hebt waar je exact 15 stappen moet zetten vooraleer je kan springen. Dan is 15 stappen voor jou belangrijk om te kunnen springen en niet 12 stappen of 13 of 14, maar ook niet 16 stappen. Als je dat nu in een in-statement zou stoppen dan zouden we iets kunnen schrijven als volgt (je hoeft het niet in een Python-console te typen):

```
if stappen == 1:
    print ("1 is verkeerd")
elif stappen == 2:
    print ("2 is verkeerd")
elif stappen == 3:
    print ("3 is verkeerd")
```

en dit gaat zo verder. Het bovenstaande is natuurlijk veel eenvoudiger te schrijven als:

```
if stappen <15 or stappen >15:
    print ("%s is verkeerd" % stappen)
```

maar het is nog eenvoudiger dit alles op te maken met **not**:

```
if not stappen == 15
    print ("%s is verkeerd" % stappen)
```

or

Het trefwoord **or** wordt gebruikt om minstens 2 uitdrukkingen aan elkaar te hangen in een opdracht (zoals een if-statement) waarbij ten minste 1 van de uitdrukkingen de waarde True (waar) moet hebben.

Bijvoorbeeld:

```
>>> if stad == 'Gent' or stad == 'Leuven':
    print ('Vlaamse stad')
elif stad == 'Utrecht' or stad == 'Groningen':
    print ('Nederlandse stad')
```

Als de variabele stad het woord Gent of Leuven bevat zal Vlaamse stad geantwoord worden door Python, als deze variabele echter Utrecht of Groningen is dan krijgen we Nederlandse stad te zien.

Onderstaande tabel geeft een waarheidstabel voor **or**:

A	B	Resultaat
True	True	True
True	False	True
False	True	True
False	False	False

Tabel 8: A or B geeft Resultaat

pass

Soms als je aan het coderen bent heb je enkel een stukje code nodig om eens iets uit te testen. Soms is dit wat vervelend omdat je na bijvoorbeeld een for-loop een blok code nodig hebt die wordt uitgevoerd. Hetzelfde geldt voor een if-statement als de voorwaarde True is. Bijvoorbeeld:

```
>>> if i < 6:
    print('Bijna genoeg')
```

Bovenstaande zal wel functioneren, maar als je onderstaande zou ingeven:

```
>>> if i < 6:
```

of:

```
>>> for i in range (0,8):
```

Dan krijg je (afhankelijk van de situatie) ofwel Python die je de prompt niet teruggeeft en dus verder input verwacht, ofwel een foutboodschap zoals:

```
IndentationError: expected an indented block
```

Python verwacht dus een blok code. `Indented block` betekent zoiets als *ingesprongen blok*.

In zo'n geval kunnen we het trefwoord **pass** gebruiken waarbij we het statement maken, maar de code erachter niet hoeven in te vullen. Meestal wordt dit gebruikt als we nog niet goed weten wat er moet volgen, of als we iets willen uitproberen. Laten we een voorbeeld nemen:

```
>>> for i in range (0,8):
    print ('De waarde van i is %s' %i)
    if i == 6:
        pass
```

```
De waarde van i is 0
```

```

De waarde van i is 1
De waarde van i is 2
De waarde van i is 3
De waarde van i is 4
De waarde van i is 5
De waarde van i is 6
De waarde van i is 7
>>>

```

Als we nadien weten wat we in het blok na `if i == 6:` willen zetten, dan vervangen we gewoon `pass` door dat blok.

print

Het keyword `print` schrijft iets naar de Python-console. Dit iets kan een string zijn, een waarde of een variabele:

```

>>> print ('Joehoe')
Joehoe
>>> print (250)
250
>>> print (i)
7
>>>

```

raise

Een trefwoord voor gevorderden. Dit keyword wordt gebruikt om een error (*fout*) te laten gebeuren. Raar, hé dat je wil dat je programma een fout maakt. Wel, goede programmeurs willen soms dat fouten gemaakt worden omdat ze dan hun code kunnen verbeteren en willen weten waar er misschien problemen in hun code kunnen opduiken.

return

Het trefwoord `return` wordt gebruikt om een waarde terug te krijgen van een functie. Je kan bijvoorbeeld een functie maken die de leeftijd teruggeeft:

```

>>> def leeftijd ():
        return ouderdom
>>>

```

Als je dan deze functie oproept, kan de waarde gegeven worden aan een andere variabele of afgedrukt worden:

```

>>> print (leeftijd())
12
>>> oud = leeftijd()
>>>

```

try

Dit keyword wordt gebruikt door geavanceerde programmeurs. Het is het begin van een blok code dat eindigt met de trefwoorden **except** en/of **finally**. Dit soort codeblok wordt gebruikt om fouten in programma's te behandelen, zodat de gebruiker weet wat er foutgelopen is in plaats dat Python een ingebouwde foutboodschap weergeeft die weinig informatief is voor de eindgebruiker.

while

Net zoals een for-loop is **while** (Engels voor *zolang*) ook een manier om een lus te maken. Een for-loop telt een exact aantal getallen, terwijl een while-loop blijft lopen zolang de uitdrukking True is. Je moet wel een beetje opletten hiermee, want als een uitdrukking alleen True is, dan blijft de lus doorlopen (een oneindige lus):

```
>>> a = 5
>>> while a == 5:
    print ('ik blijf doorlopen')
Ik blijf doorlopen
...
Ik blijf doorlopen
>>>
```

De lus wordt enkel onderbroken als je de Python-console afsluit of als je de toetsencombinatie [CTRL]+[C] samen indrukt. Je kan beter bovenstaande code vervangen door :

```
>>> a = 1
>>> while a < 5:
    print ('Ik blijf doorlopen')
    a = a + 1

Ik blijf doorlopen
Ik blijf doorlopen
Ik blijf doorlopen
Ik blijf doorlopen
>>>
```

Hier wordt 4 keer bovenstaande afgedrukt (zolang **a < 5**), waarna één opgeteld wordt bij de variabele **a**.

with

with is een trefwoord voor gevorderden.

yield

Ook **yield** is een trefwoord voor gevorderden.

14. Bijlage B: Functies

Python heeft een aantal ingebouwde functies. Dit zijn functies die niet eerst moeten geïmporteerd worden. Python 3.4 bevat de in de onderstaande tabel ingebouwde functies. We zullen er een aantal van bespreken.

<code>abs()</code>	<code>dict()</code>	<code>help()</code>	<code>min()</code>	<code>setattr()</code>
<code>all()</code>	<code>dir()</code>	<code>hex()</code>	<code>next()</code>	<code>slice()</code>
<code>any()</code>	<code>divmod()</code>	<code>id()</code>	<code>object()</code>	<code>sorted()</code>
<code>ascii()</code>	<code>enumerate()</code>	<code>input()</code>	<code>oct()</code>	<code>staticmethod()</code>
<code>bin()</code>	<code>eval()</code>	<code>int()</code>	<code>open()</code>	<code>str()</code>
<code>bool()</code>	<code>exec()</code>	<code>isinstance()</code>	<code>ord()</code>	<code>sum()</code>
<code>bytearray()</code>	<code>filter()</code>	<code>issubclass()</code>	<code>pow()</code>	<code>super()</code>
<code>bytes()</code>	<code>float()</code>	<code>iter()</code>	<code>print()</code>	<code>tuple()</code>
<code>callable()</code>	<code>format()</code>	<code>len()</code>	<code>property()</code>	<code>type()</code>
<code>chr()</code>	<code>frozenset()</code>	<code>list()</code>	<code>range()</code>	<code>vars()</code>
<code>classmethod()</code>	<code>getattr()</code>	<code>locals()</code>	<code>repr()</code>	<code>zip()</code>
<code>compile()</code>	<code>globals()</code>	<code>map()</code>	<code>reversed()</code>	<code>__import__()</code>
<code>complex()</code>	<code>hasattr()</code>	<code>max()</code>	<code>round()</code>	
<code>delattr()</code>	<code>hash()</code>	<code>memoryview()</code>	<code>set()</code>	

abs()

De functie **abs()** geeft de absolute waarde terug van een getal. Een absolute waarde is een getal dat niet negatief is. Bijvoorbeeld, de absolute waarde van 4 is 4, maar de absolute waarde van -2,3 is 2,3. In code gegoten:

```
>>> print (abs(4))
4
>>> print (abs(-2.3))
2.3
>>>
```

bool()

De functie **bool()** geeft ofwel de waarde `True` ofwel de waarde `False` terug afhankelijk van de parameterwaarde.

Als de parameterwaarde uit getallen bestaat, dan wordt bij het getal 0 `False` teruggegeven en bij alle andere getallen de waarde `True`:

```
>>> print (bool(0))
False
>>> print (bool(1))
True
>>> print (bool(80.20))
True
>>> print (bool(-80.20))
True
>>>
```

Voor parameterwaarden die geen getallen zijn, wordt enkel bij **None** en **False** de waarde `False` teruggegeven, al de rest krijgt de waarde `True`:

```
>>> print (bool(False))
False
>>> print (bool(None))
False
>>> print (bool(True))
True
>>> print (bool('xyz'))
True
>>>
```

dir()

De functie `dir()` geeft een lijstje met informatie over de ingegeven waarde. Je kan deze ingebouwde functie `dir()` gebruiken met strings, getallen, functies, modules, objecten, klassen. Eigenlijk met alles. Bij sommige waarden zal de functie enkel zinloze informatie geven:

```
>>> dir(10)
['_abs_', '_add_', '_and_', '_bool_', '_ceil_',
'_class_', '_delattr_', '_dir_', '_divmod_', '_doc_',
'_eq_', '_float_', '_floor_', '_floordiv_', '_format_',
'_ge_', '_getattr_', '_getnewargs_', '_gt_',
'_hash_', '_index_', '_init_', '_int_', '_invert_',
'_le_', '_lshift_', '_lt_', '_mod_', '_mul_', '_ne_',
'_neg_', '_new_', '_or_', '_pos_', '_pow_', '_radd_',
'_rand_', '_rdivmod_', '_reduce_', '_reduce_ex_',
'_repr_', '_rfloordiv_', '_rlshift_', '_rmod_', '_rmul_',
'_ror_', '_round_', '_rpow_', '_rrshift_', '_rshift_',
'_rsub_', '_rtruediv_', '_rxor_', '_setattr_', '_sizeof_',
'_str_', '_sub_', '_subclasshook_', '_truediv_',
'_trunc_', '_xor_', 'bit_length', 'conjugate', 'denominator',
'from_bytes', 'imag', 'numerator', 'real', 'to_bytes']
>>>
```

Nogal een lijst met speciale functies, maar de functie `dir()` oproepen met een string resulteert in betere informatie:

```
>>> dir('aa')
['_add_', '_class_', '_contains_', '_delattr_', '_dir_',
'_doc_', '_eq_', '_format_', '_ge_', '_getattr_',
'_getitem_', '_getnewargs_', '_gt_', '_hash_', '_init_',
'_iter_', '_le_', '_len_', '_lt_', '_mod_', '_mul_',
'_ne_', '_new_', '_reduce_', '_reduce_ex_', '_repr_',
'_rmod_', '_rmul_', '_setattr_', '_sizeof_', '_str_',
'_subclasshook_', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace',
'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'maketrans', 'partition', 'replace', 'rfind', 'rindex', 'rjust',
'rstrip', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
>>>
```

Hier zien we dat er functies zijn zoals `capitalize()`, die de eerste letter in een hoofdletter verandert:

```
>>> print ('wij gaan spelen'.capitalize())
Wij gaan spelen
>>>
```

Of een functie zoals `split()` die onze zin in woorden opdeelt:

```
>>> print ('wij gaan spelen'.split())
['wij', 'gaan', 'spelen']
>>>
```

Of de functie `upper()` die alle letters omvormt tot hoofdletters:

```
>>> print ('wij gaan spelen'.upper())
WIJ GAAN SPELEN
>>>
```

eval()

De functie `eval()` neemt een string als een parameter en voert deze uit net alsof het een Python commando was. Dit is gelijkaardig aan het trefwoord `exec`, behalve dat het ietwat anders werkt. Met `exec` kan je een mini Python programma maken, met `eval()` kunnen enkel eenvoudige stukjes commando uitgevoerd worden:

```
>>> eval ('100+200')
300
>>>
```

float()

Deze functie verandert een string of een geheel getal (bij programmeurs ook een integer genoemd) in een getal met een decimale punt (in het Engels *floating point number*) wat we uit de wiskunde ook kennen als een reëel getal.

Het getal 5 is bijvoorbeeld een integer, maar 5.1 en 85.3215 zijn reële getallen (of *floating point numbers*). Je kan een string omvormen tot een 'float' door:

```
>>> float ('5')
5.0
```

In een string kan je natuurlijk ook een decimale punt plaatsen:

```
>>> float ('321.654987')
321.654987
>>>
```

Een getal kan in een reëel getal omgevormd worden:

```
>>> float (150)
150.0
>>> float (150.365)
150.365
>>>
```

int()

De functie `int()` verandert een string of een getal in een geheel getal (wat we ook *integer* noemen):

```
>>> int(12.34)
12
>>> int('12')
12
>>>
```

Deze functie werkt licht verschillend van de functie `float()`. Als je probeert om een decimaal getal in een string om te vormen, wordt een foutboodschap bekomen:

```
>>> int('12.34')
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int('12.34')
ValueError: invalid literal for int() with base 10: '12.34'
>>>
```

len()

De functie `len()` geeft de lengte van een object terug. In het geval van een string, geeft het het aantal karakters terug uit deze string (spaties inclusief!):

```
>>> len('Dit is een string waar we karakters tellen')
42
>>>
```

In het geval van een lijst of een tuple geeft deze functie het aantal items terug:

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> print (len(lijstje))
5
>>> tup = (1, 2, 3)
>>> print (len(tup))
3
>>>
```

Deze `len()` functie vindt ook een goede toepassing in lussen waar je het aantal items in een lijst wil afdrukken:

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> for items in lijstje:
    print (items)

eieren
bloem
boter
melk
appelen
>>>
```


Maar wat als je ook het aantal items uit je lijstje wil tellen, naast afdrukken:

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> lengte = len(lijstje)
>>> for teller in range(0,lengte):
    print ('Het item op indexplaats %s is %s' % (teller,
lijstje[teller]))

Het item op indexplaats 0 is eieren
Het item op indexplaats 1 is bloem
Het item op indexplaats 2 is boter
Het item op indexplaats 3 is melk
Het item op indexplaats 4 is appelen
>>>
```

max()

De functie **max()** geeft het grootste item terug uit een lijst, tuple of string. Hiermee wordt bedoeld, het hoogste getal, of het verst in het alfabet. Je moet zelfs geen lijst, tuple of string gebruiken, want direct uitlezen kan ook. Bijvoorbeeld:

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> print (max(lijstje))
melk
>>> lijst = [6, 8, 12, 5, 3, 7]
>>> print (max(lijst))
12
>>> print (max(21, 65, 45, 98, 321, 25))
321
>>>
```

min()

De functie **min()** werkt op net dezelfde manier als **max()**, enkel geeft het de kleinste waarde terug van een string, tuple of lijst:

```
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> print (min(lijstje))
appelen
>>> lijst = [6, 8, 12, 5, 3, 7]
>>> print (min(lijst))
3
>>> print (min(21, 65, 45, 98, 321, 25))
21
>>>
```

open()

Dit is een functie om bestanden te openen en een bestandsobject met functies die toegang hebben tot informatie in het bestand terug te geven (cf. Hoofdstuk 9).

range()

De **range()** functie wordt voornamelijk bij for-loops gebruikt als je enkele keren een lus wil doorlopen. We hebben deze functie reeds meermaals gebruikt met 2 argumenten, maar 3 argumenten kunnen evengoed gebruikt worden. Een voorbeeldje van deze functie met 2 argumenten:

```
>>> for teller in range(0,4):
        print (teller)

0
1
2
3
>>>
```

Wat je tot hertoe waarschijnlijk nog niet wist is dat de **range()** functie een speciaal object teruggeeft, dat we een *iterator* noemen, waar de for-loop zich doorheen werkt. Deze iterator kunnen we in een lijst stoppen, zodat je de getallen ziet als je deze afdruckt:

```
>>> print (list(range(0,4)))
[0, 1, 2, 3]
>>>
```

Je bekomt eigenlijk een lijstje met getallen dat toegekend kan worden aan bijvoorbeeld variabelen en die elders in jouw programma gebruikt kunnen worden:

```
>>> lijstje_getallen = list(range(0,25))
>>> print (lijstje_getallen)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24]
>>>
```

In bovenstaande hebben we telkens 2 argumenten gebruiken bij de functie **range()**. De eerste waarde is de startwaarde en de tweede waarde de stopwaarde (niet inclusief!) Als we nu een 3^e argument toevoegen, dan is dit de stap. Bij 2 argumenten is de stap (het verhogen van het getal) na elke iteratie telkens 1 geweest. Maar als we nu eens niet per 1 maar per 2 willen verhogen? Daarvoor dient deze stap:

```
>>> lijstje_getallen = list(range(0,25,2))
>>> print (lijstje_getallen)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
>>>
```

Als we de stap vergroten:

```
>>> lijstje_getallen = list(range(0,25,4))
>>> print (lijstje_getallen)
[0, 4, 8, 12, 16, 20, 24]
>>>
```

sum()

Deze functie telt de items in een lijst op en geeft de som ervan terug:

```
>>> lijstje_getallen = list(range(0,25,4))
```

```
>>> print (lijstje_getallen)
[0, 4, 8, 12, 16, 20, 24]
>>> print(sum(lijstje_getallen))
84
>>>
```

15. Bijlage C: Modules

Python bevat veel modules om vanalles uit te kunnen voeren. De lijst is veel te lang om in dit boek op te nemen, maar je kan hem vinden op docs.python.org/3/py-modindex.html. Deze lijst is heel uitgebreid en zeker niet alle modules ga je nodig hebben. Sommige modules zijn ook complex, terwijl andere eenvoudig zijn. Enkele modules gaan we in deze bijlage bespreken om je een idee te geven van de mogelijkheden ermee en zodat je weet hoe ermee om te gaan. Een module moet steeds eerst geïmporteerd worden.

Module random

Je kent het spelletje wel waar je een willekeurig getal moet raden tussen 1 en 10. Wel, deze module doet exact hetzelfde, ze raadt een willekeurig getal tussen de waarden die jij opgeeft. Deze module bevat een aantal functies met de meest interessante voor ons `randint()`, `choice()` en `shuffle()`.

Met de functie `randint()` wordt een willekeurig getal gekozen tussen een startwaarde en een eindwaarde:

```
>>> import random
>>> print (random.randint(0, 100))
35
>>> print (random.randint(100, 1000))
405
>>> print (random.randint(1000, 10000))
7587
>>>
```

Dit kunnen we gebruiken om een simpel raadspelletje te maken met een while-loop:

```
>>> import random
>>> import sys
>>> getal = random.randint(1,10)
>>> while True:
    print('Raad een getal tussen 1 en 10')
    checken = sys.stdin.readline()
    raden = int(checken)
    if raden == getal:
        print('Juist geraden')
        break
    elif raden < getal:
        print('Hoger')
    elif raden > getal:
        print('Lager')
```

```
Raad een getal tussen 1 en 10
3
Hoger
```

De functie `choice()` wordt gebruikt als je een lijst hebt en er een willekeurig item uit wil pikken:

```
>>> import random
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> print (random.choice(lijstje))
```

```

appelen
>>> print (random.choice(lijstje))
eieren
>>> print (random.choice(lijstje))
melk
>>>

```

Gebruik de functie `shuffle()` als je een lijstje dooreen wil schudden:

```

>>> import random
>>> lijstje = ['eieren', 'bloem', 'boter', 'melk', 'appelen']
>>> random.shuffle(lijstje)
>>> print(lijstje)
['appelen', 'boter', 'eieren', 'bloem', 'melk']
>>> random.shuffle(lijstje)
>>> print(lijstje)
['eieren', 'boter', 'appelen', 'melk', 'bloem']
>>>

```

Module sys

Deze module bevat zeer goed bruikbare systeemfuncties. Het is een beetje een rare manier om te zeggen dat dit zeer belangrijke functies zijn binnen Python. Een aantal van diegene die we veelal kunnen gebruiken in de module `sys` zijn: `exit()`, `stdin()`, `stdout()`.

Met de `exit()` functie vertellen we de Python-console om te stoppen (de manier waarop hangt af van Windows, Linux of Mac):

```

>>> import sys
>>> sys.exit()
>>>

```

De functie `stdin()` hebben we reeds in dit boek gebruikt om te vragen om input in te typen via het toetsenbord:

```

>>> import sys
>>> mijn_variabele = sys.stdin.readline()
testje
>>> print (mijn_variabele)
testje
>>>

```

De functie `stdout()` doet net het tegengestelde, deze schrijft boodschappen naar de console. Dit lijkt sterk op de `print`-opdracht maar het werkt eerder als een bestand. Soms is het beter om `stdout()` te gebruiken dan `print`:

```

>>> import sys
>>> mijn_variabele = sys.stdout.write('testje')
testje
>>> print(mijn_variabele)
6
>>> mijn_variabele = sys.stdout.write('testje\n\n')
testje

```

```
>>>
```

De functie `stdout.write()` geeft een getal terug dat het aantal karakters telt. Dit kan je visueel maken door deze variabele `mijn_variabele` af te drukken met de `print`-opdracht.

Door aan de string zogenaamde *escape karakters* (zoals bovenstaand `\n\n`) toe te voegen, kan je speciale eigenschappen toekennen aan deze string. Zo is het escape karakter `\n` een speciaal karakter waardoor naar een nieuwe regel gesprongen moet worden.

```
>>> lijn = 'regell bevat tekst
SyntaxError: EOL while scanning string literal
>>> lijn = 'regell bevat tekst\n\nen na een lege regel terug een
regel'
>>> print(lijn)
regell bevat tekst

en na een lege regel terug een regel
>>>
```

Enkele escape karakters die regelmatig voorkomen vind je in onderstaande tabel

Escape karakter	Uitleg
<code>\b</code>	Backspace
<code>\n</code>	Nieuwe lijn
<code>\r</code>	Return of Enter
<code>\t</code>	Horizontale tabulatie

Tabel 9: Escape karakters

Module time

De module `time` van Python bevat functies om, jawel, de tijd weer te geven. Maar de meest gemakkelijke functie `time()` doet iets anders dan je eerst zou verwachten.

```
>>> import time
>>> print (time.time())
1422993929.666906
>>>
```

Deze geeft een nogal raar getal terug. Dit getal is het aantal seconden verstreken sinds 1 januari 1970 om 00u00min00sec. Je zou kunnen denken dat dit echt niet belangrijk is, maar vergis je niet. Soms wil je weten welke tijd er verlopen is tussen een startmoment en een eindmoment. Dan zijn het aantal seconden die verstreken zijn wel van belang. Stel dat je wil weten hoe lang het duurt om alle getallen van 0 tot 100 op je computerscherm te printen:

```
>>> def getallen(max):
    for tel in range (0, max):
        print (tel)

>>> getallen (1000)
0
1
...
```

```
998
999
>>>
```

Hierboven printen we de getallen, maar we hebben zonder chronometer geen idee van de tijd hoelang het duurt. Met de `time()` functie weten we dit wel:

```
>>> def getallen(max):
    tijd0 = time.time()
    for tel in range(0, max):
        print(tel)
    tijd1 = time.time()
    print('Er zijn %s seconden overgegaan om te printen' %
(tijd1-tijd0))

>>> getallen(1000)
0
1
...
998
999
Er zijn 2.5272040367126465 seconden overgegaan om te printen
>>>
```

Maar hoe werkt dit nu? We nemen onze begintijd `tijd0` op bij de start. Denk eraan, dit zijn het aantal seconden sinds het begin van 1 januari 1970. Dan gaat ons programma doorheen de lus waarbij telkens de waarde `tel` afgedrukt wordt. Op het einde van de for-loop nemen we opnieuw de tijd op en stoppen deze in de variabele `tijd1`. Door het verschil te maken tussen `tijd1` en `tijd0` weten we hoelang het geduurd heeft om onze lus te doorlopen en af te drukken.

Andere functies die binnen de module `time` ter beschikking staan zijn `asctime()`, `ctime()`, `localtime()`, `sleep()`, `strftime()` en `strptime()`.

De functie `asctime()` neemt de datum als tuple (een lijst van waarden die niet veranderd kan worden) en verandert deze in een leesbare vorm. Je kan het ook oproepen zonder argument en dan geeft het de huidige datum en tijd in leesbare vorm:

```
>>> import time
>>> print(time.asctime())
Wed Feb  4 19:20:53 2015
>>>
```

Om deze functie wel met een argument op te roepen, moeten we eerst een tuple aanmaken met de correcte waarden voor datum en tijd. We nemen als tuple de variabele `tup`. De volgorde van de getallen bij deze variabele zijn: Jaar, Maand, Dag, Uur, Minuut, Seconde, Dag van de week (0 = maandag, 1 = dinsdag ... 6 = zondag), Dag van het jaar, Zomeruur (0 = geen zomeruur, 1 = zomeruur):

```
>>> import time
>>> tup = (2015, 1, 14, 20, 33, 15, 2, 0, 0)
```

```
>>> print (time.asctime(tup))
Wed Jan 14 20:33:15 2015
>>>
```

Let wel op met de waarden die je in de tuple stopt. Als je er fouten instopt, bekom je een onzinnige datum (vergelijk onderstaande met bovenstaande):

```
>>> import time
>>> tup = (2015, 1, 14, 20, 33, 15, 1, 0, 0)
>>> print (time.asctime(tup))
Tue Jan 14 20:33:15 2015
```

De functie `ctime()` wordt gebruikt om een aantal seconden in een leesbare vorm te zetten:

```
>>> import time
>>> var = time.time()
>>> print (var)
1423083872.234206
>>> print(time.ctime(var))
Wed Feb 4 22:04:32 2015
>>>
```

De functie `localtime()` geeft de huidige datum en tijd weer als tuple in dezelfde volgorde die we hierboven hebben gebruikt:

```
>>> import time
>>> print (time.localtime())
time.struct_time(tm_year=2015, tm_mon=2, tm_mday=4, tm_hour=22,
tm_min=5, tm_sec=26, tm_wday=2, tm_yday=35, tm_isdst=0)
>>>
```

En we kunnen deze waarde ook in de functie `asctime()` stoppen:

```
>>> import time
>>> var = time.localtime()
>>> print(time.asctime(var))
Wed Feb 4 22:05:49 2015
>>>
```

De functie `sleep()` is goed bruikbaar als we een pauze van een bepaalde tijd in ons programma willen inlassen. Het argument bevat het aantal te wachten seconden:

```
>>> import time
>>> for tel in range (1, 11):
    print(tel)
    time.sleep(1)

1
2
3
4
5
6
7
8
9
```



```
10
>>>
```

In bovenstaande code wordt na elke print-opdracht 1 seconde gewacht alvorens de volgende print-opdracht uit te voeren. Zonder de pauze, zou dit programma direct volledig uitgevoerd worden.

Je zou kunnen denken dat een pauze-functie niet zo handig is, maar denk eens aan je wekker. Daar heb je ook een toets die als 's morgens het alarm afgaat je nog een pauze van enkele minuten kan inlassen om wakker te worden. Wel, dit is net hetzelfde.

De functie `strftime()` wordt gebruikt om de weergave van datum en tijd te veranderen en de functie `strptime()` wordt gebruikt om een string te nemen en deze in een datum/tijd-tuple te veranderen. Laat ons eerst eens naar `strftime()` kijken. Daarstraks hebben we gezien hoe we een tuple in een string konden veranderen met `asctime()`:

```
>>> import time
>>> tup = (2015, 1, 14, 20, 33, 15, 2, 0, 0)
>>> print (time.asctime(tup))
Wed Jan 14 20:33:15 2015
>>>
```

In de meeste gevallen is dit goed bruikbaar, maar wat als je dit allemaal niet wil? Als je bijvoorbeeld enkel de datum wil en niet de tijd? Dat kunnen we met `strftime()`:

```
>>> import time
>>> tup = (2015, 1, 14, 20, 33, 15, 2, 0, 0)
>>> print (time.strftime('%d %b %Y', tup))
14 Jan 2015
>>>
```

Zoals je merkt heeft de functie `strftime()` 2 argumenten: eerst is er een string met het datum/tijd-formaat die beschrijft hoe dit op het scherm moet weergegeven worden. Het tweede argument is een tuple die de tijdwaarden bevat. Het formaat '%d %b %Y' is een manier om te zeggen geef me de dag, de maand en het jaar in een bepaald formaat (zie Tabel 10). We zouden de maand ook als een getal kunnen weergeven:

```
>>> import time
>>> tup = (2015, 1, 14, 20, 33, 15, 2, 0, 0)
>>> print (time.strftime('%d/%m/%Y', tup))
14/01/2015
>>>
```

Met de '/' maken we een scheiding tussen de getallen van de dag, de maand en het jaar. Je zou hier ook een ander teken kunnen zetten dan de '/'.

Code	Uitleg
%a	Verkorte naam van de weekdag (Mon, Tues, Wed, Thurs, Fri, Sat, Sun)
%A	Volledige naam van de weekdag (Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday)

%b	Verkorte naam van de maand (Jan, Feb, Mar, Apr, May, Jun, Jul, Aug, Sep, Oct , Nov, Dec)
%B	Volledige naam van de maand (January, February, March, April, May, June, July, August, September, October, November, December)
%c	Volledige datum en tijd in het formaat van <code>asctime()</code>
%d	Dag van de maand als een getal (01 tot en met 31)
%H	Uur van de dag in 24-uursformaat (00 tot en met 23)
%I	Uur van de dag in 12-uursformaat (01 tot en met 12)
%j	Dag van het jaar als een getal (001 tot en met 366)
%m	Maand als een getal (01 tot en met 12)
%M	Minuten als een getal (00 tot en met 59)
%p	Voormiddag (AM) of namiddag (PM)
%S	Seconden als een getal
%U	Weeknummer van het jaar als een getal (00 tot en met 53) met Zondag als eerste dag van de week
%w	Dag van de week als een getal (Zondag = 0; maandag = 1 ... Zaterdag = 6)
%W	Weeknummer van het jaar als een getal (00 tot en met 53) met Maandag als eerste dag van de week
%x	Simpel datum formaat (jaar/maand/dag , vb. 15/02/16)
%X	Simpel tijdsformaat (uur:minuut:seconde , vb. 15:26:01)
%Y	Jaar in 2 cijfers (vb. 2015 is dan 15)
%Y	Jaar in 4 cijfers (vb. 2015)

Tabel 10: Datum/tijd formattering

De functie `strptime()` is zowat het omgekeerde van `strftime()`. Deze functie neemt een string en converteert die naar een tuple met de datum en tijd:

```
>>> import time
>>> var = time.strptime('14 Jan 2015', '%d %b %Y')
>>> print (var)
time.struct_time(tm_year=2015, tm_mon=1, tm_mday=14, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=2, tm_yday=14, tm_isdst=-1)
>>>
```

Als de datum in onze string dag/maand/jaar is (vb. 14/01/2015) kunnen we het volgende gebruiken:

```
>>> import time
>>> var = time.strptime('14/01/2015', '%d/%m/%Y')
>>> print (var)
time.struct_time(tm_year=2015, tm_mon=1, tm_mday=14, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=2, tm_yday=14, tm_isdst=-1)
```

Of als we werken met jaar/maand/dag typen we het volgende in:

```
>>> import time
>>> var = time.strptime('2015/01/14', '%Y/%m/%d')
>>> print (var)
time.struct_time(tm_year=2015, tm_mon=1, tm_mday=14, tm_hour=0,
tm_min=0, tm_sec=0, tm_wday=2, tm_yday=14, tm_isdst=-1)
```

```
>>>
```

Wat natuurlijk hetzelfde resultaat moet geven.

16. Bijlage D: Oplossingen

16.1. Hoofdstuk 3

Mogelijke oplossing 3.1:

```
>>> (5*25)+(7*8)
181
>>>
```

Mogelijke oplossing 3.2:

```
>>> speelgoed = ['computer', 'voetbal', 'pijl en boog']
>>> dieren = ['hamster', 'kip', 'poes']
>>> favorieten = speelgoed + dieren
>>> print (favorieten)
['computer', 'voetbal', 'pijl en boog', 'hamster', 'kip', 'poes']
>>>
```

Mogelijke oplossing 3.3:

```
>>> voornaam1 = 'Pepijn'
>>> voornaam2 = 'Niel'
>>> naam = '%s en %s'
>>> print (naam % (voornaam1, voornaam2))
Pepijn en Niel
>>>
```

16.2. Hoofdstuk 4

Mogelijke oplossing 4.1:

```
>>> import turtle
>>> t = turtle.Pen()
>>> t.forward(200)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(200)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)                #(deze laatste stap is niet noodzakelijk)
>>>
```

Mogelijke oplossing 4.2:

```
>>> t.reset()
>>> t.forward(100)
>>> t.left(120)
>>> t.forward(100)
>>> t.left(120)
>>> t.forward(100)
>>> t.left(120)                #(deze laatste stap is niet noodzakelijk)
>>>
```

16.3. Hoofdstuk 5

Oplossing 5.1:

De laatste lijn wordt ook afgedrukt omdat de indentatie van `print ('Deze lijn ook')` weggevallen is.

16.4. Hoofdstuk 6

Mogelijke oplossing 6.1:

```
Hallo 0
```

De variabele `a` start bij 0. Er zal dus maar 1 print-opdracht uitgevoerd worden, want de voorwaarde die nadien volgt `a < 9` is niet vervuld. Dan wordt de code in het blok van de lus onderbroken en daar er geen code nadien staat, stopt de uitvoering van het programma.

Mogelijke oplossing 6.2:

```
>>> gespaard = 200
>>> for jaar in range (0, 11):
    print ('Jaar %s geeft €%s totaal' % (jaar, gespaard))
    gespaard = gespaard + gespaard * 0.03

Jaar 0 geeft €200 totaal
Jaar 1 geeft €206.0 totaal
Jaar 2 geeft €212.18 totaal
Jaar 3 geeft €218.5454 totaal
Jaar 4 geeft €225.101762 totaal
Jaar 5 geeft €231.85481486 totaal
Jaar 6 geeft €238.8104593058 totaal
Jaar 7 geeft €245.974773084974 totaal
Jaar 8 geeft €253.35401627752321 totaal
Jaar 9 geeft €260.95463676584893 totaal
Jaar 10 geeft €268.7832758688244 totaal
>>>
```

Zoals eerder al gemeld kan het zijn dat er afrondingsfouten voorkomen. Daar gaan we nu nog geen rekening mee houden.

Mogelijke oplossing 6.3:

```
16 31
17 32
18 33
19 34
20 35
21 36
22 37
23 38
24 39
```

```

25 40
26 41
27 42
28 43
29 44
30 45
31 46
32 47
33 48
34 49
35 50
>>>

```

16.5. Hoofdstuk 7

Mogelijke oplossing 7.1:

```

>>> def bereken_intrest(gespaard, intrest):
    for jaar in range(0, 11):
        print ('Jaar %s geeft €%s totaal' % (jaar, gespaard))
        gespaard = gespaard + gespaard * intrest

>>> bereken_intrest(200, 0.03)
Jaar 0 geeft €200 totaal
Jaar 1 geeft €206.0 totaal
Jaar 2 geeft €212.18 totaal
Jaar 3 geeft €218.5454 totaal
Jaar 4 geeft €225.101762 totaal
Jaar 5 geeft €231.85481486 totaal
Jaar 6 geeft €238.8104593058 totaal
Jaar 7 geeft €245.974773084974 totaal
Jaar 8 geeft €253.35401627752321 totaal
Jaar 9 geeft €260.95463676584893 totaal
Jaar 10 geeft €268.7832758688244 totaal
>>>

```

Mogelijke oplossing 7.2:

```

>>> def bereken_intrest(gespaard, intrest, periode):
    for jaar in range(0, (periode+1)):
        print ('Jaar %s geeft €%s totaal' % (jaar, gespaard))
        gespaard = gespaard + gespaard * intrest

>>> bereken_intrest(200, 0.03, 7)
Jaar 0 geeft €200 totaal
Jaar 1 geeft €206.0 totaal
Jaar 2 geeft €212.18 totaal
Jaar 3 geeft €218.5454 totaal
Jaar 4 geeft €225.101762 totaal
Jaar 5 geeft €231.85481486 totaal
Jaar 6 geeft €238.8104593058 totaal
Jaar 7 geeft €245.974773084974 totaal
>>>

```

Mogelijke oplossing 7.3:

```
>>> import sys
>>> def bereken_intrest():
    print ('Geef je gespaard bedrag in:')
    gespaard = int(sys.stdin.readline())
    print ('Geef de intrest in:')
    intrest = float(sys.stdin.readline())
    print ('Geef de periode in:')
    periode = int(sys.stdin.readline())
    for jaar in range(0, (periode+1)):
        print ('Jaar %s geeft €%s totaal' % (jaar, gespaard))
        gespaard = gespaard + gespaard * intrest

>>> bereken_intrest()
Geef je gespaard bedrag in:
100
Geef de intrest in:
0.03
Geef de periode in:
4
Jaar 0 geeft €100 totaal
Jaar 1 geeft €103.0 totaal
Jaar 2 geeft €106.09 totaal
Jaar 3 geeft €109.2727 totaal
Jaar 4 geeft €112.550881 totaal
>>>
>>>
```

16.6. Hoofdstuk 10Mogelijke oplossing 10.1:

Zeshoek:

```
>>> def zeshoek(grootte):
    for i in range(0,6):
        t.forward(grootte)
        t.left(60)

>>> t.reset()
>>> zeshoek(100)
>>>
```

Achthoek:

```
>>> def achthoek(grootte):
    for i in range(0,8):
        t.forward(grootte)
        t.left(45)

>>> t.reset()
>>> achthoek(100)
>>>
```

Mogelijke oplossing 10.2:

```
>>> def achthoek(grootte, gevuld):
    if gevuld == True:
        t.begin_fill()
    for i in range(0,8):
        t.forward(grootte)
        t.left(45)
    if gevuld == True:
        t.end_fill()

>>> t.reset()
>>> achthoek(100,True)
>>>
```


17. Versiebeheer

Versie	Datum	Items	Auteur
1.0.0	Sept. 2015	Originele versie	Johan Vereecken
1.0.1	Sept. 2015	Inhoudstafel, spelfouten, tabelkleuren aangepast	Johan Vereecken