



# Python Fundamentals

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

Python is een veel gebruikte programmeertaal in de wereld van Geo-data science én software development. De taal is geschikt als eerste kennismaking met programmeren vanwege het intuïtieve en toekanlijke karakter. Python is met name geschikt voor projecten waarbij de snelheid van bouwen belangrijk is. Python is minder geschikt voor het maken van desktopapplicaties, al is het wel mogelijk. In deze leerlijn leer je belangrijke basisprincipes toe te passen in Python, gemaakt in een IDE (Integrated Development Environment). Je gaat experimenteren met diverse voorbeeldprogramma's en na afloop kun je repeterende werkzaamheden automatiseren, handige scripts schrijven en daarmee allerlei processen vereenvoudigen.

## 1. Inleiding - waarom Python?

Wat is Python? En waarom zou je het willen leren?

Python is een objectgeoriënteerde scripttaal, waarmee je computerprogramma's (software) kunt maken. Het is wereldwijd een van de meest populaire computertalen en binnen het onderwijs en wetenschappelijk computergebruik momenteel zelfs de [meest gebruikte taal](#).

Er zijn diverse redenen waarom Python populair is en iedereen het zou moeten leren:

- Het is een (gratis) open-source taal met een grote open-source gemeenschap.
- De taal is makkelijker te leren dan C, C++, C# en Java.
- Er zijn uitgebreide standaard en open-source bibliotheken, waarmee de productiviteit van ontwikkelaars vergroot kan worden. Ook zijn er veel gratis open-source Python applicaties.
- Python is populair in kunstmatige intelligentie (Artificial Intelligence), met name vanwege de bijzondere relatie met dataverwerking (Data Science en Big Data).
- Er is een grote behoefte aan (ervaren) Python programmeurs en deze functies behoren tot de bestbetaalde banen binnen de programmeerwereld.

Alle reden dus om ermee aan de slag te gaan. We beginnen met de installatie.

### 1.1 Python installeren

Om Python te gebruiken heb je een "Python release" nodig. Een Python release bestaat uit 1 of meer Python interpreter(s), diverse bibliotheken met standaard functies en hulpprogramma's.

Python releases zijn gratis verkrijgbaar voor Windows, MacOS X en Linux en kunnen gedownload worden via [deze website](#).

De Python code die je schrijft is voor alle operating systemen gelijk en je kunt dus Python programma's naar een andere computer overzetten, zelfs als die niet draait op hetzelfde operating systeem!

Ervaren programmeurs maken meestal gebruik van een geïntegreerde ontwikkelomgeving of IDE (Integrated Development Environment) omdat die het bouwen, testen, samenvoegen en distribueren van software gemakkelijker maakt. De Anaconda Python distributie is zo'n IDE.

Als student heb je toegang tot JetBrains Pycharm dit is handig voor het ontwikkelen applicatie. Echter wordt in deze cursus gebruik gemaakt van Anaconda Jupyter Lab om eenvoudig codes uit te kunnen voeren. Mocht je Pycharm willen gebruiken i.p.v. anaconda overleg dit dan even met de docent.

In deze cursus gaan we ervan uit dat je de meest recente Anaconda Python distributie hebt geïnstalleerd. Voor details verwijzen we naar het [software installatiedocument](#).

### 1.2 Geschiedenis van Python

Python is begin jaren 90 ontworpen en ontwikkeld door Guido van Rossum, destijds verbonden aan het Centrum voor Wiskunde en Informatica in Amsterdam. De taal is mede gebaseerd op inzichten van professor Lambert Meertens, die een taal genaamd ABC had ontworpen, bedoeld als alternatief voor BASIC, maar

dan met geavanceerde datastructuren. Inmiddels wordt de taal doorontwikkeld door een enthousiaste groep, tot juli 2018 geleid door Van Rossum. Deze groep wordt ondersteund door vrijwilligers op het internet. De ontwikkeling van Python wordt geleid door de Python Software Foundation.

Op 16 oktober 2000 is Python 2 geïntroduceerd en op 3 december 2008 Python 3. Deze versies zijn niet compatibel, dat wil zeggen een Python 2 programma zal niet draaien in een Python 3 omgeving en omgekeerd. De ondersteuning voor Python 2 is op 1 januari 2020 geëindigd. Hoewel er nog veel Python 2 programma's operationeel zijn, moet een nieuwe programmeur starten met het leren van Python 3. Oude Python 2 programma's zullen in de komende jaren worden uitgefaseerd of geconverteerd naar Python 3.

Python heeft zijn naam te danken aan het favoriete televisieprogramma van Guido van Rossum, 4Monty Python's Flying Circus. De documentatie bevat verschillende verwijzingen naar sketches van dit programma.

## 1.3 Opbouw van de cursus

Elk volgend hoofdstuk start met een inleiding en eindigt met een samenvatting. In de tekst staan verschillende opgaven die je echt allemaal moet maken om te oefenen met de nieuwe begrippen. Je kunt niet leren programmeren door erover te lezen; alleen door code te schrijven, fouten te maken en die op te lossen word je een echte programmeur! Vergelijk het maar met leren autorijden. Je kunt wel 100 boeken bestuderen over dat onderwerp, maar als je niet een keer achter het stuur gaat zitten en gaat rijden zal je het nooit leren. Gelukkig is leren programmeren een stuk veiliger dan leren autorijden. Een computercrash is (meestal) minder fataal dan een autocrash!

Na de samenvatting wordt elk hoofdstuk afgesloten met een aantal herhalingsopgaven. Ook deze moet je allemaal maken om te controleren of je de behandelde stof echt hebt begrepen. Het is de tijdsinvestering meer dan waard!

## 1.4 Opgaven

Maak, voordat je verder gaat naar het volgende hoofdstuk, eerst onderstaande opgaven. Succes en veel plezier!

### *Opgave 1*

Installeer de laatste versie van Anaconda3 (zie het software installatiedocument). Test of de Anaconda prompt, JupyterLab en Jupyter Notebook correct werken.

### *Opgave 2*

Start de Python interpreter m.b.v. de Anaconda prompt (CMD.exe) in interactieve modus (dus toets "python" (zonder quotes) in na de prompt). Toets na `>>>` in: `5 + 7` en druk op Enter. Wat is het resultaat?

### *Opgave 3*

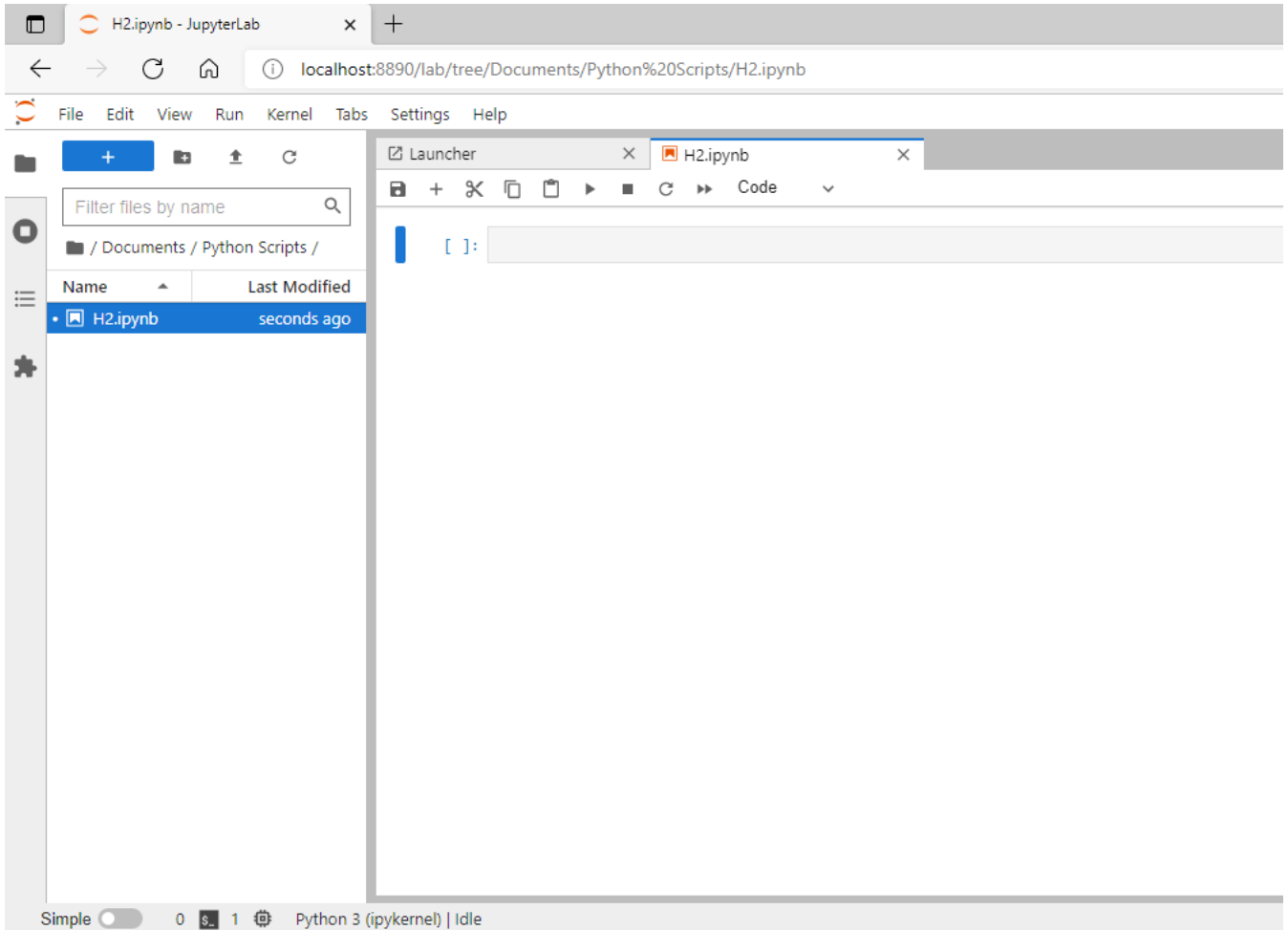
Maak met behulp van het Kladblok een bestand met naam `telop.py` aan dat alleen de regel `5 + 7` bevat. Run dit programma in script mode (dus start de Anaconda prompt (CMD.exe) en toets "python telop.py" (zonder quotes) in na de prompt en druk op Enter. Wat is het resultaat?

## 2. Aan de slag met Interactive Python

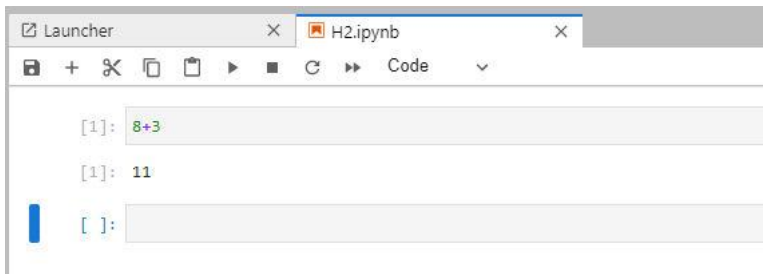
In dit hoofdstuk leer je hoe je Python kunt gebruiken als rekenmachine. We ontdekken ook welke andere simpele bewerkingen er met getallen en stukjes tekst kunnen worden uitgevoerd. Hiermee leggen we de basis voor het programmeren met Python. Aan het eind bekijken we hoe de uitkomst van een berekening kan worden getoond als Python in script mode wordt uitgevoerd.

### 2.1 Python als rekenmachine

Vanuit Anaconda Navigator start je JupyterLab (kan even duren) en vervolgens open je in het Launcher venster een Jupyter Notebook sessie. Er wordt nu een tweede tabblad geopend met de naam `Untitled.ipynb`. Wijzig in het linker venster de naam van dit bestand in `H2.ipynb`.



Typ in het grijze vlak van het tabblad H2.ipynb naast [ ] de tekst '8+3' (zonder quotes) in en druk daarna op het zwarte driehoekje ► in de kopregel. Op een nieuwe regel verschijnt nu het resultaat van de optelling (het getal 11):



Het grijze invoervlak wordt wel een *cel* genoemd en de Python code die we in zo'n cel invoeren heet een *snippet*. Jupyter Notebook maakt gebruik van de ipython interpreter (de i vóór python slaat op 'interactive'), waardoor alle snippets een volgnummer hebben (het getal tussen '[' en ']') en de code verschillende kleuren heeft.

Als je de code van de actieve snippet (aangegeven door een dikke blauwe streep) uitvoert door op de run-knop ► te drukken wordt het resultaat onder die snippet getoond met hetzelfde volgnummer en wordt de cursor naar de volgende cel verplaatst. Je kunt hetzelfde effect bereiken door op de combinatie 'Shift' + 'Enter' te drukken. Als je in dezelfde cel wil blijven moet je op de combinatie 'Ctrl' + 'Enter' drukken.

In het vervolg zullen we in- en uitvoer op de volgende manier weergeven:

In:  
# Hier staat de code van de snippet

Out:  
# Hier staat het resultaat als de code wordt uitgevoerd

Het hekje # (Eng. hashtag) wordt in Python gebruikt voor commentaar. Alles wat rechts van het hekje op dezelfde regel staat wordt genegeerd door de Python interpreter. Het is belangrijk om je code van voldoende commentaar te voorzien zodat een ander (of jijzelf op een later moment) kan begrijpen wat die code doet.

In bovenstaande optelling hadden we de karakters 8, + en 3 direct na elkaar ingetoetst, maar het is toegestaan (en voor de leesbaarheid zelfs aan te raden) om wat spaties toe te voegen. Het maakt Python niets uit en het resultaat is gelijk, voor de toelichting van de spaties is voorafgaand een opmerking toegevoegd:

```
In:
# Extra spaties verhogen de leesbaarheid!
8 + 3
```

```
Out:
11
```

We gaan hierna enkele eenvoudige berekeningen met getallen uitvoeren, maar eerst bekijken we welke getallen in Python zijn toegestaan.

Wil je meer weten over opmerkingen in Python, lees dan blz 63 en 64 van [Crash course programmeren in Python - BUKU](#)

## 2.2 Getallen in Python

We gaan kijken naar getallen in Python. Lees eerst het hoofdstuk getallen en vervolgens de daarbij behorende uitleg.

Python kent 2 soorten getallen: gehele getallen (Eng. integers), zoals 0, 1, -157, 20458901, en getallen met een decimale punt (Eng. floating-point numbers), zoals 2.5, -3.14159, 800.66, enz.

In tegenstelling tot de meeste andere programmeertalen kan een integer in Python willekeurig groot zijn, hoewel er natuurlijk wel een fysieke beperking is door de grootte van het computergeheugen.

Floating-point getallen (of kortweg floats) kennen wel beperkingen in grootte en precisie. Er zijn getallen die niet exact kunnen worden uitgedrukt als getal met een decimale punt, bijvoorbeeld het getal 1 gedeeld door 3 (één derde). Je kunt dit getal benaderen door 0.33, 0.333 of door een getal dat heel veel drieën bevat na de decimale punt, maar je zult met een eindig aantal cijfers nooit de exacte waarde kunnen representeren.

Je kunt meer lezen over de interne representatie, maximale grootte en precisie van floats in de Python documentatie. Voorlopig is het voldoende dat je weet wat een float is. Merk op dat we spreken over getallen met een decimale punt en niet over getallen met een decimale komma! Net als bijna alle andere programmeertalen hanteert Python namelijk de Engelse notatie met een punt als decimaalscheider en niet de Nederlandse, die daarvoor een komma gebruikt.

Je mag geen komma's of meer dan 1 punt in een getal gebruiken, maar tussen 2 cijfers mag je wel een underscore \_ plaatsen om de leesbaarheid te vergroten. We kunnen dus 1 miljard (een 1 met 9 nullen) ook als volgt representeren:

```
In:
1_000_000_000
```

```
Out:
1000000000
```

Python ondersteunt ook de wetenschappelijke notatie voor getallen, zoals 109 of 3.14159 x 10<sup>-4</sup>. We gebruiken daarvoor de kleine letter 'e' of de hoofdletter 'E'. Het resultaat is altijd een float, ook als de waarde feitelijk een geheel getal is.

```
In:
1e9
```

```
Out:
1000000000.0
```

```
In:
3.14159E-4
```

```
Out:
0.000314159
```

## 2.3 Rekenen in Python

Laten we nu eens kijken hoe berekeningen kunt uitvoeren in Python. We gebruiken daarvoor de rekenkundige operatoren '+' (optellen), '-' (aftrekken), '\*' (vermenigvuldigen) en '/' (delen).

In:  
`9 * 7.2 - 5.97`

Out:  
58.83

Je ziet dat je integers en floats door elkaar mag gebruiken. In dit geval is het resultaat natuurlijk een float, maar is dat altijd het geval?

Je hebt eerder gezien dat bij het optellen van 2 integers 8 en 3 het resultaat een integer is 11. Bij aftrekking en vermenigvuldiging van 2 integers is het resultaat ook een integer (ga na!). Maar bij deling zie je het volgende:

In:  
`6 / 3`

Out:  
2.0

Hoewel het resultaat van de deling (precies) 2 is toont Python het toch als een float! Je ziet dat elke gehele waarde gerepresenteerd kan worden als een integer of als een float (dan staat er dus .0 achter de waarde).

Het optellen, aftrekken, vermenigvuldigen of delen van 2 floats (of een float en een integer) levert altijd een float als resultaat. We zullen in het volgende hoofdstuk zien hoe we van een integer een float kunnen maken en omgekeerd.

## Verrijkingstof

Naast de 4 genoemde rekenkundige operatoren kent Python nog 3 andere, namelijk \*\* (machtsverheffen), // (geheeltallige deling) en % (modulo).

De bekendste van deze drie is vermoedelijk machtsverheffen, dus die behandelen we als eerste. Machtsverheffen is in feite herhaald vermenigvuldigen van een getal met zichzelf. Zo kunnen we het product  $5 * 5 * 5 * 5$  noteren als 54. Het getal 5 heet het *grondtal* en het getal 4 de *exponent*.

De exponent bepaalt dus het aantal keren dat het grondtal met zichzelf wordt vermenigvuldigd. In Python wordt 54 geschreven als  $5 ** 4$ :

In:  
`5 ** 4`

Out:  
625

Let op de volgorde van grondtal en exponent. Als je 2 getallen optelt of vermenigvuldigt maakt de volgorde niet uit ( $8 + 3 = 11$  en  $3 + 8 = 11$ ,  $8 * 3 = 24$  en  $3 * 8 = 24$ ), maar  $2 ** 3$  en  $3 ** 2$  geven wel verschillende uitkomsten (namelijk resp.  $2 * 2 * 2 = 8$  en  $3 * 3 = 9$ )!

Zowel het grondtal als de exponent mag een float zijn. Merk op dat worteltrekken overeenkomt met machtsverheffen met exponent 0.5.

In:  
`9 ** 0.5`

Out:  
3.0

Geheeltallige deling '/' (Eng. floor division) en modulo '%' zijn 2 operatoren die vaak in combinatie worden gebruikt. Geheeltallige deling is eigenlijk deling waarbij alles na de decimale punt wordt weggelaten en modulo is de rest als je geheeltallig deelt.

Stel je hebt 13 knikkers en wil die (eerlijk) verdelen over 3 kinderen. Dan geef je elk kind 4 knikkers en blijft er 1 knikker over. Geheeltallige deling van 13 door 3 levert 4 op en 13 modulo 3 levert de rest (dus 1) op. Laten we dat eens controleren in Python:

In:  
`13 // 3`

Out:  
4

In:

```
13 % 3
```

```
Out:  
1
```

Beide operatoren kunnen ook gebruikt worden met floats.

## Opgave

Vervang in bovenstaande snippets 13 door 13.5 en 3 door 3.2 en kijk welke resultaten dat oplevert bij [geheeltallige deling en modulo](#).

## 2.4 Prioriteit en volgorde van berekeningen

Tot nu toe hebben we eenvoudige berekeningen uitgevoerd, maar nu gaan we onderzoeken wat er gebeurt als je de berekeningen wat complexer maakt. Laten we beginnen met het volgende voorbeeld:

```
In:  
3 + 4 * 5
```

Als je deze berekening op een gewone rekenmachine uitvoert door de cijfers en symbolen van links naar rechts in te toetsen krijg je als antwoord 35 (na het intoetsen van het cijfer 4 verschijnt het getal 7 en als dat met 5 wordt vermenigvuldigd levert dat 35 op).

Als je bovenstaande snippet echter in Python gaat runnen is het resultaat:

```
Out:  
23
```

Blijkbaar wordt in Python eerst de vermenigvuldiging van 4 met 5 uitgevoerd en daar wordt dan 3 bij opgeteld. Waarschijnlijk ben je niet verbaasd over dit resultaat omdat je al op de basisschool hebt geleerd dat "vermenigvuldigen voor optellen en aftrekken gaat". We zeggen dat de operator "\*" (vermenigvuldigen) een hogere prioriteit heeft dan '+' (optellen) of '-' (aftrekken).

Als je toch als uitkomst 35 wil krijgen bij bovenstaande uitdrukking moet je haakjes zetten:

```
In:  
(3 + 4) * 5
```

```
Out:  
35
```

Haakjes hebben dus weer een hogere prioriteit dan "\*" (ze hebben zelfs een hogere prioriteit dan alle andere operatoren).

Op dezelfde manier heeft "\*\*" (machtsverheffen) een hogere prioriteit dan "\*" (vermenigvuldigen).

Hoe zit het met delen? Bekijk het volgende voorbeeld:

```
In:  
4 / 2 * 2
```

```
Out:  
4.0
```

Je hebt misschien op de basisschool geleerd dat vermenigvuldigen vóór delen gaat maar in de wiskunde en programmeertalen is dat niet zo; daar hebben ze dezelfde prioriteit. Hier wordt dus eerst  $4 / 2$  bepaald (uitkomst 2.0, want bij deling krijg je altijd een float!) en die waarde wordt vermenigvuldigd met 2.

## Opgave

Wat is de uitkomst van de berekening  $5 - 4 - 2$  in [Python](#)?

Nog een voorbeeld met machtsverheffen:

```
In:
```

```
4 ** 3 ** 2
```

Out:

262144

Dit resultaat (ga na of het klopt) zal je misschien verbazen. Als je van links naar rechts werkt bereken je eerst  $4 * 4 * 4 = 64$  en vervolgens  $642 = 64 * 64 = 4096$ . De reden voor het (grote) verschil is dat bij machtsverheffen van rechts naar links wordt gewerkt in plaats van andersom zoals bij optellen/afrekken en vermenigvuldigen/delen. Als je de uitkomst 4096 wil krijgen moet je dus haakjes zetten om  $4 ** 3$ .

## 2.5 De waarden True en False

Naast berekeningen kunnen we ook nog andere dingen met getallen doen, bijvoorbeeld ze met elkaar vergelijken.

Het wiskundige teken '>' stelt het begrip 'groter dan' voor en je kunt je afvragen of Python 'weet' dat 7 groter is dan 4. Laten we dit eens invullen en uitvoeren:

In:

```
7 > 4
```

Out:

True

Python geeft als resultaat dus True (Engels voor 'Waar'). Als je onderzoekt of 7 kleiner is dan 4 door > te veranderen in < krijg je als resultaat False (Engels voor 'Niet waar').

### Opgave

Zoek zelf uit wat er gebeurt als je in bovenstaande code resp. > vervangt door <, >= (groter of gelijk), <= (kleiner of gelijk), == (gelijk) en != (ongelijk).

N.B. Uit deze opgave blijkt dat je gelijkheid van twee getallen test met behulp van == en dus niet met een enkel =-teken! Dat teken wordt namelijk gebruikt om een waarde toe te kennen, maar dat behandelen we in het volgende hoofdstuk.

True en False zijn *logische waarden* (soms ook wel Boolese waarden genoemd) en het zijn ook de enige logische waarden!

Logische waarden kunnen we met elkaar combineren m.b.v. *logische operatoren*, waarvan NOT, (NIET) AND (EN) en OR (OF) de bekendste zijn. In Python zijn NOT, AND en OR geïmplementeerd door resp. 'not', 'and' en 'or' (zonder quotes).

- de 'not' van een logische waarde geeft de omgekeerde waarde terug, dus 'not True' is 'False' en 'not False' is 'True'.
- de 'and' van 2 logische waarden geeft 'True' terug als ze beide 'True' zijn en anders 'False'
- de 'or' van 2 logische waarden geeft 'True' terug als minstens 1 van beide 'True' is en anders 'False'

### Voorbeeld:

In:

```
False or True
```

Out:

True

Let op dat de logische OR niet hetzelfde is als de 'of' die we meestal in spreektaal gebruiken. Als je vraagt "Wil je koffie of thee?" is het de bedoeling dat de ander aangeeft welke van de 2 dranken gewenst is. Deze vorm van 'of' wordt in de logica de 'exclusieve OF' (Eng. *exclusive OR* of XOR) genoemd.

### Verrijkingstof

Meestal wordt de werking van logische operatoren getoond in zogenaamde *waarheidstabellen*. In deze tabellen zijn p en q beweringen die waar (True) of niet waar (False) kunnen zijn.

p	NOT p
True	False
False	True

p	q	p AND q
True	True	True
True	False	False
False	True	False
False	False	False

p	q	p OR q
True	True	True
True	False	True
False	True	True
False	False	False

## 2.6 Werken met tekst

Tot nu toe hebben we alleen gewerkt met getallen, maar we kunnen ook allerlei bewerkingen met teksten uitvoeren. Een stuk tekst in Python heet een string en moet worden begrensd door enkele of dubbele aanhalingstekens (Eng. quotes), dus 'tekst' of "tekst".

In:  
'hello'

Out:  
'hello'

Als we dezelfde tekst tussen dubbele aanhalingstekens zetten en uitvoeren is het resultaat:

In:  
"hello"

Out:  
'hello'

Als het resultaat een string is zet Python deze bij voorkeur tussen enkele aanhalingstekens, tenzij dit onmogelijk is. Bekijk het volgende voorbeeld:

In:  
"auto's"

Out:  
"auto's"

Als je deze tekst met alleen enkele aanhalingstekens wil uitvoeren krijg je een foutmelding:

In:  
'auto's'

Out:  
Input In []  
'auto's'  
^  
SyntaxError: invalid syntax

Je kunt deze fout oplossen door dubbele aanhalingstekens om de tekst te zetten, maar je kunt ook een "backslash" (\) zetten vóór elk aanhalingsteken in de string. Python interpreteert deze backslash als volgt: het aanhalingsteken na de backslash moet behandeld worden als een teken in de string en niet als een afsluiting van de string. Als we in het vorige voorbeeld een backslash toevoegen voor het aanhalingsteken voor de s krijgen we:

In:  
'auto\'s'

Out:  
"auto's"



Als een string zowel enkele als dubbele aanhalingstekens bevat kun je dus een foutmelding voorkomen door gebruik te maken van backslashes. Je kunt ook drie (enkele of dubbele) aanhalingstekens om de string zetten. Om een backslash in een string op te nemen moet je voor elke backslash een extra backslash zetten. De volgende strings zijn dus correct:

```
In:
'snelle auto\'s zijn \"cool\".'
"""snelle auto's zijn "cool"."""
"deze string bevat een backslash '\\'"
```

Als je bovenstaande strings in een snippet zet en uitvoert zul je in het resultaat toch nog enkele overbodige backslashes zien. Dat is een gevolg van het feit dat het resultaat een string is die in interactieve mode met enkele of dubbele aanhalingstekens moet worden getoond. We zullen in de volgende paragraaf zien hoe we het verwachte resultaat krijgen met behulp van de `print`-functie.

Een bijzondere string is een string waarbij er geen tekst tussen de aanhalingstekens staat, dus `"` of `""`. Dit heet de *lege string*.

We kunnen 2 strings aan elkaar vastplakken (*concateneren*) met behulp van de `+`-operator.

```
In:
'vast' + "plakken"

Out:
'vastplakken'
```

Ook kunnen we een string vermenigvuldigen met een geheel getal

```
In:
'spam' * 3

Out:
'spamspamspam'
```

Let erop dat je alleen “rechte” aanhalingstekens gebruikt en geen “ronde”. Tekstverwerkers zoals Word hebben de neiging om rechte aanhalingstekens automatisch te vervangen door ronde, maar die worden niet door Python herkend. Teksteditors zoals Kladblok doen dat niet, maar pas op bij het kopiëren van Python code naar een tekstverwerker!

## Opgave

Zoek zelf uit wat er gebeurt als je de tekst `'spam'` vermenigvuldigt met [0, -3 of 2.5](#).

Je kunt geen bewerkingen op een string uitvoeren met andere rekenkundige operatoren zoals `**`, `/`, `//`, `%` en `-` (ga na), maar je kunt wel strings “vergelijken”. Bekijk het volgende voorbeeld:

```
In:
'spam' > 'hello'

Out:
True
```

String1 is kleiner dan string2 als string1 in alfabetische volgorde vóór string2 komt. Dit wordt ook wel *lexicografische ordening* genoemd. Als er hoofdletters in de strings voorkomen kan het resultaat anders zijn dan je verwacht! Dat komt omdat hoofdletters vóór kleine letters in de ordening komen (dus `'Z'` is kleiner dan `'a'`).

## Opgave

Verander de kleine `'s'` van `'spam'` in een hoofdletter en bekijk het [resultaat](#).

## Opgave

Experimenteer zelf met de andere vergelijksoperatoren <, >=, <=, == en !=.

## 2.7 De print-functie

Je hebt een aantal opdrachten uitgevoerd met getallen, teksten en de logische waarden True en False. Nadat je op de knop “Uitvoeren” gedrukt had, werd het resultaat (of een foutmelding) op de volgende regel(s) getoond.

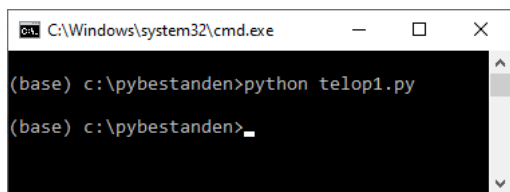
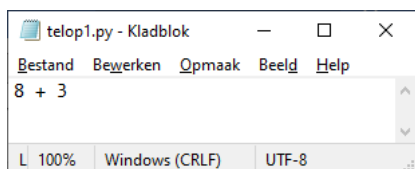
Tot nu toe stond alle code op 1 regel, maar in een snippet kun je ook meerdere regels opnemen. Bekijk het volgende voorbeeld:

In:  
8 + 3  
5 \*\* 2

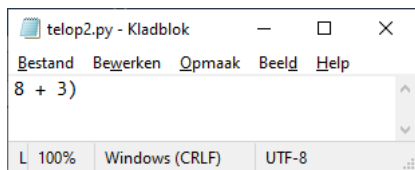
Out:  
25

Je ziet dat Python alleen het resultaat van de machtsverheffing op de laatste regel toont en niet het resultaat van de optelling op de eerste regel.

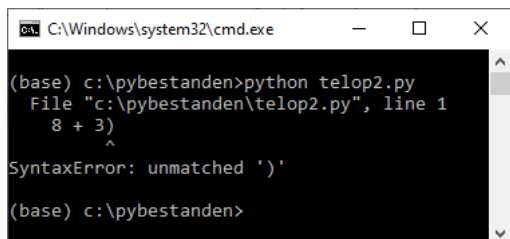
Stel dat je met behulp van Kladblok of een andere teksteditor een bestand telop1.py maakt, waarin uitsluitend de tekst '8' + 3' staat (zonder quotes) en dit runt in script mode (bijvoorbeeld met de CMD,exe Prompt in de Anaconda Navigator).



Zoals je ziet wordt er nu helemaal geen resultaat afgedrukt, terwijl het programma wel normaal is geëindigd. Om dat laatste aan te tonen maken we een nieuw Python bestand telop2.py aan, en voegen na de 3 een sluithaakje toe.



Als we telop2.py runnen krijgen we een foutmelding (een SyntaxError), omdat er niet evenveel openings- als sluitingshaakjes zijn.



Met behulp van de print-functie kun je het resultaat van alle berekeningen in een snippet (in interactieve mode) of in een programma (in script mode) tonen. We passen het eerste voorbeeld van deze paragraaf als volgt aan:

In:

```
print(8 + 3)
print(5 ** 2)
```

Out:

```
11
25
```

Je ziet dat het resultaat van beide berekeningen nu wordt afgedrukt. De print-functie gebruik je dus door het woord 'print' te schrijven (zonder quotes) en daarna wat je wil tonen tussen een rond openings- en sluithaakje. Het is gebruikelijk om het openingshaakje direct na het woord 'print' te plaatsen (dus zonder spatie), maar dat is niet verplicht. Net als bij de eenvoudige berekeningen die we eerder hebben uitgevoerd mag je willekeurig veel spaties toevoegen om de haakjes (maar uiteraard niet in het woord 'print!').

Als je de print-functie gebruikt met een string zie je het volgende gebeuren:

In:

```
print('hello, world!')
```

Out:

```
hello, world!
```

De aanhalingstekens rondom de tekst 'hello, world!' zijn dus verwijderd!

Met behulp van de print-functie kunnen we ook ingewikkelde teksten met backslashes netjes op het scherm tonen (vergelijk dit met het resultaat zonder de print-functie):

In:

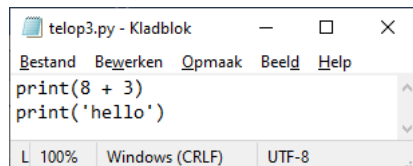
```
print('snelle auto's zijn \"cool\".')
print("""snelle auto's zijn "cool".""" )
print("deze string bevat een backslash '\\')"
```

Out:

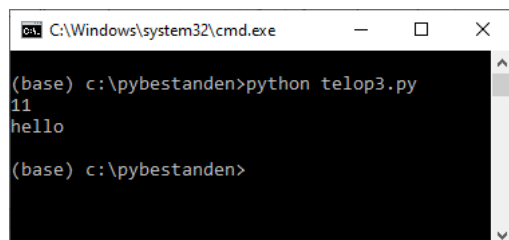
```
snelle auto's zijn "cool".
snelle auto's zijn "cool".
deze string bevat een backslash '\'
```

Hoe werkt de print-functie in script mode?

We maken een nieuw bestand telop3.py aan met de volgende 2 regels code:



Als we dit bestand runnen wordt het resultaat van de optelling op de eerste regel en de tekst van de tweede regel getoond. Ook hier zijn de aanhalingstekens rond de tekst verwijderd.



De print-functie geeft dus precies dezelfde uitvoer in interactieve en script mode!

Overigens kun je ook binnen Jupyter Notebook een .py bestand in script mode runnen. Dat gaat met behulp van het commando %run. (N.B. Dit is in feite een commando van IPython).

In:

```
%run c:\pybestanden\telop3.py
```

Out:

```
11
hello
```

We zullen het begrip functie in latere hoofdstukken uitgebreid behandelen en ook zien welke afdrukmogelijkheden de print-functie nog meer biedt. Nu beperken we ons tot de opmerking dat je ook meerdere berekeningen of bewerkingen tegelijkertijd kan tonen door deze tussen het open- en sluithaakje te zetten, gescheiden door komma's. De print-functie zet automatisch een spatie tussen de verschillende items.

In:  
`print('hello,' , 'world!')`

Out:  
hello, world!

## Opgave

Geeft de code hierboven hetzelfde resultaat als de code hieronder?

In:  
`print('hello,' + 'world!')`

Probeer eerst het antwoord te bedenken en controleer daarna of dat klopt [door de code uit te voeren](#).

Je kunt zelfs een willekeurig aantal waarden, gescheiden door komma's, tussen de haakjes van de print-functie zetten, en dit mogen getallen, berekeningen of stukjes tekst zijn: Deze waarden worden ook wel de *argumenten* van de print-functie genoemd.

In:  
`print('De som van' , 8 , 'en' , 3 , 'is:' , 8 + 3)`

Out:  
De som van 8 en 3 is: 11

Een speciaal geval is de print-functie zonder waarden tussen de haakjes. In dat geval wordt een lege regel afgedrukt.

## 2.8 Samenvatting

In dit hoofdstuk heb je kennis gemaakt met de ontwikkeltool Jupyter Notebook waarin je stukjes Python code (snippets) kun schrijven en uitvoeren. Je hebt geleerd hoe je de interactieve Python interpreter kunt gebruiken als rekenmachine en ook hoe je simpele bewerkingen met logische waarden en tekst kunt uitvoeren. Tevens heb je kennis gemaakt met je eerste Python functie, de print-functie, waarmee je enerzijds meerdere resultaten van een snippet tegelijk kunt tonen en die anderzijds noodzakelijk is om uitvoer te laten zien van een Python programma dat in script mode wordt uitgevoerd.

## 2.9 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studietoestel.

### Opgave 1

Print het volgende stukje text:  
Hoera, nu gaan we echt met Python beginnen!!!

### Opgave 2

We gaan nu een stukje rekenen.  
De formule om temperatuur in Celsius om te rekenen naar Fahrenheit is als volgt:  
`celsius * (9/5) + 32`

Wat is 21° graden Celsius in Fahrenheit?  
En 0° graden?

## Opgave 3

Als je `print("Hallo")` uitvoert krijg je Hallo  
De omliggende quotes worden niet  
getoond.

Kan je de print-functie zo gebruiken dat het "Hallo" toont?

## Opgave 4

`2 < 15` geeft True omdat 2 kleiner is dan 15. Wat is het resultaat als je nu de getallen 2  
en 5 omringd met quotes? Kan je het resultaat verklaren?

# 3. Variabelen van elementaire datatypen

In het vorige hoofdstuk hebben we wat vingeroefeningen gedaan met Python. We hebben enkele kleine berekeningen uitgevoerd en stukjes tekst gemanipuleerd. In een echt programma zul je meestal bepaalde waarden of uitkomsten willen bewaren omdat je die op meerdere plekken nodig hebt. Hiervoor gebruiken we *variabelen*. In dit hoofdstuk leer je wat variabelen zijn, welke namen ze mogen hebben en hoe je ze een waarde kan geven. We zullen nu alleen 'simpele' variabelen introduceren, dat wil zeggen variabelen van een elementair datatype.

## 3.1 Variabelen

Een variabele reserveert een stukje ruimte in het geheugen om gegevens (data) op te slaan.

Elke variabele:

- heeft een naam (Engels: identifier)
- heeft een waarde
- is van een bepaald (data)type

Je kan een variabele ook zien als een doos, waar iets in zit.

In dit hoofdstuk beperken we ons tot variabelen waarbij het datatype een getal (een integer of een float), een logische waarde (True of False) of een string is. In latere hoofdstukken zullen we (variabelen van) complexere datatypen, zoals lijsten, tuples en dictionaries behandelen.

Als de leeftijd van een persoon meerdere keren wordt gebruikt in een Python programma kunnen we die als een variabele definiëren door een naam te verzinnen en die een waarde te geven m.b.v. het '='-teken. Dit heet *toekenning* (Engels: assignment). Aan de linkerkant van het '='-teken staat de naam van de variabele en aan de rechterkant de waarde. Het datatype van de variabele wordt in Python bepaald door de waarde.

```
In:
leeftijd = 25
```

De naam (identifier) van de variabele is dus leeftijd, de waarde is 25 en het datatype is integer.

Als we een variabele in de print-functie gebruiken wordt niet de naam, maar de waarde van de variabele getoond.

```
In:
print(leeftijd)
```

```
Out:
25
```

Ook nu kan de print-functie meerdere waarden (argumenten) hebben:

```
In:
print('Mijn leeftijd is' , leeftijd , 'jaar.')
```

Out:  
Mijn leeftijd is 25 jaar.

Let op dat de naam van de variabele niet tussen quotes staat! Ga zelf na wat er gebeurt als je dat wel doet.

Je kunt de waarde van een variabele veranderen door een nieuwe waarde aan de naam toe te kennen, dus bijvoorbeeld 'leeftijd = 26' (zonder quotes) als je weer een jaartje ouder bent.

In:  
leeftijd = 26  
print('Mijn leeftijd is nu' , leeftijd , 'jaar.')

Out:  
Mijn leeftijd is nu 26 jaar.

Aan de rechterkant van het '='-teken kan ook een berekening staan, waarin zelfs de naam van de variabele mag voorkomen:

In:  
leeftijd = leeftijd + 5  
print('Over 5 jaar ben ik' , leeftijd , 'jaar.')

Out:  
Over 5 jaar ben ik 31 jaar.

Je moet het statement 'leeftijd = leeftijd + 5' als volgt lezen: de nieuwe waarde van leeftijd is de oude waarde van leeftijd plus 5. Het '='-teken stelt dus geen gelijkheid voor, maar toekenning van een (nieuwe) waarde. In het vorige hoofdstuk hebben we gezien dat gelijkheid in Python wordt aangegeven met '==' (dus een dubbel '='-teken).

Op alle plekken waar je een waarde kunt invullen mag je ook de naam van een variabele zetten. Bij de berekening of bewerking wordt dan de waarde van de variabele gebruikt.

## Voorbeeld (salarisberekening):

In:  
salaris = 1000  
opslag = 50  
print('Het nieuwe salaris is' , salaris + opslag)

Out:  
Het nieuwe salaris is 1050

Merk op dat variabele salaris nog steeds waarde 1000 heeft omdat we die niet hebben veranderd!

## Opgave

Kun je de bovenstaande snippet zo aanpassen dat hetzelfde resultaat wordt getoond, maar ook de waarde van salaris is [gewijzigd](#)?

## Voorbeeld

Nu een voorbeeld met float variabelen. Stel dat we 6% korting krijgen op een artikel van € 99.75. Wat is dan de prijs die we moeten betalen voor dat artikel?

In:  
brutoprijs = 99.75  
korting = 0.06  
nettoprijs = brutoprijs \* (1 - korting)  
print('Het artikel kost :' , nettoprijs)

Out:  
Het artikel kost : 93.765

N.B. Let erop dat je korting de waarde 0.06 geeft en niet 6. Bij het rekenen met procenten moet je immers delen door 100!

## Voorbeeld

Een voorbeeld met strings:

```
In:
h = 'hello, '
w = 'world!'
print(h + w)
```

```
Out:
hello, world!
```

## 3.2 Variabelen aanvullend materiaal

Ter aanvulling op de vorige hoofdstukken kan je hoofdstuk 2 doorlezen op blz 49 over variabelen en eenvoudige datatypes

## 3.3 De type-functie

Bij veel programmeertalen moet je het datatype van een variabele vooraf vastleggen (declareren) en kan de variabele alleen maar waarden hebben van dat datatype. Dit heet statische typering, Bij Python wordt het datatype echter bepaald door de waarde van de variabele. We zeggen dat variabelen in Python dynamisch getypeerd zijn.

### Voorbeeld:

```
In:
a = 10
print(a)
a = 2.7
print(a)
a = 'welkom'
print(a)
```

```
Out:
10
2.7
welkom
```

De variabele a krijgt eerst de waarde integer 10 , wordt vervolgens een float 2.7, en is uiteindelijk een "string 'welkom'".

Je kunt het datatype van een waarde opvragen met de typefunctie. Je zet dan de waarde (of een variabele met daarin een waarde) tussen de ( ) haakjes:

```
In:
a = 10
print(type(a), a)
a = 2.7
print(type(a), a)
a = 'welkom'
print(type(a), a)
```

```
Out:
<class 'int'> 10
<class 'float'> 2.7
<class 'str'> welkom
```

Python laat zien dat het type van a achtereenvolgens int (integer), float en str (string) is. Op de betekenis van class komen we later terug. In dit voorbeeld zie je dat een argument van de print-functie ook de uitkomst van een andere functie kan zijn.

## Opgave

Definieer zelf een variabele met een logische waarde (True of False) en controleer met behulp van de type-functie wat het datatype van [die variabele is](#).

## 3.4 De type-casting functies

Soms moet je het datatype van een variabele of waarde veranderen in een ander datatype. Dat kun je doen met een “typecasting” functie. De naam van zo’n functie is het nieuwe datatype en tussen haakjes staat de variabele of de waarde waarvan je het type wil wijzigen.

### Voorbeeld:

```
In:
a = 5
print(type(a), a)
a = float(a)
print(type(a), a)
```

```
Out:
<class 'int'> 5
<class 'float'> 5.0
```

In dit voorbeeld is het type van a dus gewijzigd van int naar float en de waarde van 5 naar 5.0.

## Opgave

Je krijgt hetzelfde resultaat als op regel 3 de toekenning a = 5.0 staat. Kun je bedenken waarom het handig is om ook de [typecasting-functie float\(\)](#) te hebben?

De typecasting moet overigens wel zinvol zijn. Bekijk het volgende voorbeeld:

```
In:
a = 'hallo'
print(type(a), a)
a = float(a)
print(type(a), a)
```

```
Out:
<class 'str'> hallo
-----
ValueError                                Traceback (most recent call last)
Input In [], in <cell line: 3>()
      1 a = 'hallo'
      2 print(type(a), a)
----> 3 a = float(a)
      4 print(type(a), a)
```

ValueError: could not convert string to float: 'hallo'

We bekijken nu nog 2 andere belangrijke typecasting-functies: int() en str().

We beginnen met een voorbeeld van de int-functie:

```
In:
a = int(5.0)
print(type(a), a)
```

Out:



```
<class 'int'> 5
```

In dit geval wordt float waarde 5.0 dus omgezet in gehele waarde 5, die vervolgens wordt toegekend aan de (integer) variabele a.

Wat gebeurt er als we een float waarde invullen die niet eindigt op '.0' (zonder quotes)?

```
In:
a = int(6.7)
print(type(a), a)
```

```
Out:
<class 'int'> 6
```

In dit geval wordt de waarde van de float dus "afgekapt", dat wil zeggen alles na de decimale punt vervalt. We zullen later bij de bespreking van standaard functies behandelen hoe je een float waarde kunt afronden naar het dichtstbijzijnde gehele getal.

## Opgave

Wat gebeurt er als je een string typecast naar int? Probeer dit met de strings 'hallo', '8' en '08' en '8.0'.

Tenslotte bekijken we de str-functie die een variabele of waarde omzet naar een string.

```
In:
a = str(6.7)
print(type(a), a)
```

```
Out:
<class 'str'> 6.7
```

Omdat de print-functie de aanhalingstekens om een string weghaalt lijkt a een float te zijn, maar de uitkomst van type(a) laat zien dat a wel degelijk een string is! De str-functie wordt vaak gebruikt om numerieke waarden in een stuk tekst op te nemen. In het volgende hoofdstuk zullen we een voorbeeld zien.

## 3.5 Variabelenamen

We hebben in de vorige paragraaf een paar variabelenamen gedefinieerd. Er zijn echter enkele regels waaraan ze moeten voldoen.

De naam (identifier) van een variabele:

- mag alleen bestaan uit letters, cijfers en underscores (\_)
- mag niet beginnen met een cijfer
- mag geen Python gereserveerd woord (Eng. keyword) zijn

*Gereserveerde woorden (keywords)* zijn namen die door Python worden gebruikt om de werking en eigenschappen van de taal te implementeren, zoals bijvoorbeeld besturingsstatements.

Tabel 1: Python keywords

and	As	assert	async	await
break	Class	continue	def	del
elif	Else	except	False	finally
for	From	global	if	import
in	Is	lambda	None	nonlocal
not	Or	pass	raise	return
True	Try	while	with	yield

Als je een keyword gebruikt als variabelenaam resulteert dat in een syntaxfout.

```
In:
if = 10

Out:
Input In []
      if = 10
      ^
```

SyntaxError: invalid syntax

Je mag alle soorten letters gebruiken in variabelenamen, maar je moet wel beseffen dat Python onderscheid maakt tussen hoofdletters en kleine letters (Engels: case sensitive). De variabelen a en A zijn dus verschillend, zoals het volgende voorbeeld laat zien!

In:

```
a = 5
print(A)
```

Out:

```
-----
NameError                                Traceback (most recent call last)
Input In [], in <cell line: 2>()
      1 a = 5
----> 2 print(A)

NameError: name 'A' is not defined
```

## 3.6 Conventies

Hoewel je een grote vrijheid hebt bij de keuze van een variabelenaam zul je in de praktijk vaak te maken krijgen met afspraken welke namen wel of niet gebruikt mogen worden. Deze afspraken kunnen op projectniveau zijn vastgelegd, maar er zijn ook enkele algemene richtlijnen of *conventies* voor de namen van variabelen. We noemen de belangrijkste:

- Een variabelenaam is "zelf documenterend", dat wil zeggen de naam sluit zo goed mogelijk aan bij de betekenis van de variabele in het programma. De naam brutoprijs is natuurlijk een stuk duidelijker dan de naam b. En dan moet je natuurlijk niet de nettoprijs van een artikel aan die variabele toekennen!
- Een variabelenaam bestaat in principe alleen uit kleine letters behalve als de naam uit 2 of meer woorden bestaat zoals "aantal cursisten". In dat geval zijn er 2 mogelijkheden: de woorden worden gescheiden door een underscore (\_) of het 2e en volgende woord begint met een hoofdletter. De variabelenaam is dus respectievelijk aantal\_cursisten of aantalCursisten. (let op dat er geen spatie in een variabelenaam kan voorkomen!). Een scheiding van de woorden met een \_ noem je een snake\_case Het tweede woord beginnen met een hoofdletter noem je een camelCase. De laatste komt vooral voor bij Classes.
- Een variabelenaam begint niet met een underscore. Dit soort namen wordt voornamelijk gebruikt door de ontwikkelaars van Python, zodat het niet gelezen kan worden door code in een andere scope/module)

## Opgave

Verzin zelf 10 namen die om verschillende redenen geen variabelenaam kunnen of mogen zijn. Geef bij elke naam aan waarom [niet](#).

## 3.7 Constanten

Een constante is variabele waarvan de initieel toegekende waarde niet kan veranderen. Het is gebruikelijk dat de naam van een constante uitsluitend uit hoofdletters en underscores bestaat. Constanten maken programmacode leesbaarder, betrouwbaarder en beter onderhoudbaar.

## Voorbeeld:

Stel dat een programma werkt met Engelse lengtematen en je wilt een lengte in inches omzetten naar centimeters. Je kunt dan de volgende code schrijven:

```
In:
lengte_in_inches = 3.5
lengte_in_cm = lengte_in_inches * 2.54
print(lengte_in_inches , 'inch =' , lengte_in_cm , 'cm')
```

Out:

```
3.5 inch = 8.89 cm
```

De volgende code is echter een stuk duidelijker:

In:

```
CM_PER_INCH = 2.54
lengte_in_inches = 3.5
lengte_in_cm = lengte_in_inches * CM_PER_INCH
print(lengte_in_inches , 'inch =' , lengte_in_cm , 'cm')
```

Out:

```
3.5 inch = 8.89 cm
```

Constanten kunnen ook gebruikt worden voor waarden die slechts zelden wijzigen, zoals bijvoorbeeld de BTW tarieven. Als die tarieven op meerdere plekken in de code staan moet je bij wijziging van een tarief de hele code doorlopen om de aanpassingen te doen en dan is er nog een grote kans dat je ergens een aanpassing over het hoofd ziet. Het is veel handiger om de tarieven bovenin de code als constanten te definiëren en die overal te gebruiken in plaats van de waarden. Bij wijziging van een tarief hoef je dat slechts op 1 plek aan te passen.

Helaas worden constanten niet actief door Python ondersteund. Sinds Python versie 3.8 kun je met behulp van een zogenaamde statische type checker, zoals mypy, wel vooraf controleren dat bepaalde variabelen niet worden gewijzigd in de code. Deze controle werkt echter niet tijdens de uitvoering van het programma (dus op run-time).

Je moet dus aanleren om een constante een naam in hoofdletters te geven en variabelen met een naam in hoofdletters nergens in de code te wijzigen!

## 3.8 Samengestelde toekenningen

Naast de gewone toekenning van een waarde aan een variabele met behulp van '=' kent Python samengestelde toekenningen, waarmee de code in sommige gevallen iets korter wordt.

Stel dat je 5 wil optellen bij de waarde van de variabele a. De normale code hiervoor is a = a + 5. Met behulp van de samengestelde toekenning '+=' kun je dat ook coderen als: a += 5. Je moet dit lezen als "verhoog de waarde van a met 5".

Voor aftrekken, vermenigvuldigen, delen, machtsverheffen, geheeltallig delen en modulo zijn er overeenkomstige samengestelde toekenningen, resp. '-=', '\*=', '/=', '\*\*=' en '%='.

### Opgave

Wat is het resultaat van onderstaande code?

In:

```
prijs = 10
print(prijs, type(prijs))
prijs *= 1.21
print(prijs, type(prijs))
```

Controleer of je antwoord [klopt](#).

## 3.9 Samenvatting

In dit hoofdstuk heb je geleerd wat variabelen zijn en dat elke variabele een naam (identifier) en waarde heeft en van een bepaald datatype is, namelijk het datatype van de waarde. Je hebt ook gezien hoe je een variabele definieert en hoe je de waarde van een variabele kan wijzigen.

Daarnaast heb je geleerd welke namen gebruikt mogen worden voor variabelen, welke conventies er voor namen gelden en wat gereserveerde woorden (keywords) in Python zijn. Variabelen kunnen een ander datatype krijgen na toekenning van een nieuwe waarde (we zeggen dat het datatype van een variabele in Python dynamisch is). Naast de gewone toekenning (=) kent Python ook zogenaamde samengestelde toekenningen (+=, -=, enz.).

## 3.10 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studiecoach.

### Opgave 1

Schrijf de Python code om de oppervlakte van een tafel in m<sup>2</sup> te berekenen. Maak gebruik van 2 variabelen met de lengte en breedte van de tafel, uitgedrukt in centimeters.

### Opgave 2

De inhoud van een (cilindervormig) blikje is gelijk aan de oppervlakte van de onder- of bovenkant maal de hoogte. De onderkant is een cirkel met een gegeven straal  $r$ , en zoals je misschien nog weet is de oppervlakte van een cirkel gelijk aan  $r^2$  waarbij  $\pi = 3.14159$  (bij benadering).

Schrijf de Python code om de inhoud van het blikje te berekenen, waarbij je gebruik maakt van 3 variabelen voor  $r$ ,  $\pi$  en de hoogte. Let er op dat een constante is!

### Opgave 3

Geef de fouten aan in de onderstaande code:

In:

```
2e_getal = int('35.0') x ((5 - 3) * 7
```

## 4. Expressies en operatoren

In het vorige hoofdstuk hebben we kennis gemaakt met variabelen. We hebben gezien hoe je een enkelvoudige waarde (Engels: literal) toekent aan een variabele, zoals een getal of een string. We kunnen rechts van het '='-teken ook een uitdrukking zetten die eerst berekend (geëvalueerd) moet worden voordat de waarde aan de variabele wordt toegekend, bijvoorbeeld:  $a = 8 + 3$ . De uitdrukking rechts heet een *expressie* (hier is opgebouwd uit 2 literals en de '+'-operator).

### 4.1 Expressies

Een expressie is een stukje Python code dat een waarde oplevert. Expressies worden opgebouwd uit elementaire data-elementen (de zogenaamde atomaire elementen) en operatoren.

De eenvoudigste expressies zijn:

- enkelvoudige waarde (literal)
- variabele met een enkelvoudige waarde

We noemen dit wel atomaire expressies of atomen. Met behulp van operatoren zoals '+', '<' of not kunnen we uit atomaire expressies ingewikkelder expressies maken, zoals we al in eerdere hoofdstukken hebben gezien.

De belangrijkste operatoren zijn:

- Rekenkundige operatoren: `**`, `*`, `/`, `//`, `%`, `+` en `-`
- Vergelijkingsoperatoren: `<`, `<=`, `>`, `>=`, `==` en `!=`
- Logische operatoren: `not`, `and` en `or`
- De conditionele operator: `- if - else -`

Naast deze operatoren zijn er nog andere operatoren, zoals bitbewerkingsoperatoren, shiftoperatoren en de assignment operator. Deze operatoren worden niet behandeld in de cursus.

We hebben ook al gezien dat sommige operatoren een hogere prioriteit hebben dan andere en dat de volgorde waarin de operatoren binnen een expressie worden geëvalueerd van belang is voor de uitkomst. Een bijzonder geval zie je in het volgende voorbeeld:

In:

```
print(7 > 5 or 1 / 0 == 2)
```

Out:

True

Als je de volgorde van de 2 expressies naast de 'or' verwisselt is het resultaat heel anders:

In:

```
print(1 / 0 == 2 or 7 > 5)
```

Out:

```
-----
ZeroDivisionError                                Traceback (most recent call last)
Input In [], in <cell line: 1>()
----> 1 print(1 / 0 == 2 or 7 > 5)

ZeroDivisionError: division by zero
```

Nu krijgen we een foutmelding omdat we delen door nul. Waarom ging het eerst wel goed? Dat heeft te maken met de evaluatievolgorde! In de expressie '7 > 5 or 1 / 0 == 2' bepaalt Python eerst de waarde van de linkerkant van 'or' (True). Omdat 'True or True/False' altijd True is gaat Python de waarde van de expressie aan de rechterkant van 'or' niet meer bepalen. Dit heet lazy evaluation.

Gebruik haakjes als je af moet wijken van de standaard prioriteit en/of evaluatievolgorde en ook als het niet strikt nodig is kan het handig zijn om haakjes te gebruiken voor de leesbaarheid.

## 4.2 De conditionele expressie

Een conditionele expressie is een expressie die verschillende uitkomsten kan hebben afhankelijk van het resultaat van een Boolese expressie (conditie). De syntax is als volgt:

```
expressie1 if conditie else expressie2
```

De waarde van de conditionele expressie is de waarde van expressie1 als de conditie waar (True) is en de waarde van expressie2 als de conditie niet waar (False) is.

### Voorbeeld:

Bepaal de grootste waarde van twee numerieke variabelen a en b.

In:

```
max = a if a > b else b
```

## 4.3 Prioriteit en evaluatie van operatoren

In de hoofdstukken hiervoor hebben we al gezien dat operatoren in een expressie verschillende prioriteiten kunnen hebben. Als operatoren dezelfde prioriteit hebben maakt het uit of ze van links naar rechts of van rechts naar links worden geëvalueerd. In onderstaande tabel staat de prioriteit (van hoog naar laag) en evaluatievolgorde van alle operatoren die we hebben behandeld.

Tabel 2 : Prioriteit (precedentie) en evaluatie van operatoren

Operator	Beschrijving	Evaluatie
()	Haakjes	->
**	Machtsverheffen	<-
+ (unair), - (unair)	positief, negatief	->
*, /, //, %	vermenigvuldigen, delen, geheeltallig delen, modulo	->
+, -	optellen, aftrekken	->
<, <=, >, >=, !=, ==	kleiner dan (of gelijk), groter dan (of gelijk), (on)gelijk	->
not	NOT	->
and	AND	->
or	OR	->
if - else	conditionele expressie	->

## 4.4 De input-functie

Tot nu toe maakten we bij onze berekeningen en bewerkingen gebruik van numerieke waarden en teksten die direct in de code waren ingevuld. Bij interactief gebruik (in de snippets van Jupyter Notebook) is dat niet zo'n probleem omdat je simpel waarden kunt wijzigen, maar bij grote Python programma's die in script mode worden uitgevoerd is dat niet erg handig.

Daarom introduceren we nu de input-functie waarmee je een waarde of tekst kan invoeren. In het volgende programma voer je een woord in en daarna toont Python met een passende tekst welk woord dat was.

In:

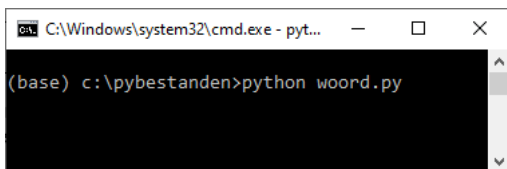
```
woord = input()
print('je hebt het volgende woord ingevoerd: ' , woord)
```

Als je deze code runt in Jupyter Notebook wordt een klein venster geopend, waarin je het woord kunt invoeren. Stel dat je 'hallo' (zonder quotes) invoert en op 'Enter' drukt dan is het resultaat:

Out:

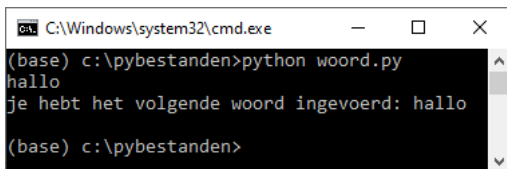
```
je hebt het volgende woord ingevoerd: hallo
```

Als je bovenstaande code in een bestand zet (hier: woord.py) en dit runt in CMD.exe lijkt er niets te gebeuren, maar knippert de cursor op de eerstvolgende lege regel.



```
C:\Windows\system32\cmd.exe - pyt...
(base) c:\pybestanden>python woord.py
```

Je voert nu 'hallo' in (zonder quotes) en drukt op 'Enter' waarna je hetzelfde resultaat krijgt als in interactieve mode.



```
C:\Windows\system32\cmd.exe
(base) c:\pybestanden>python woord.py
hallo
je hebt het volgende woord ingevoerd: hallo
(base) c:\pybestanden>
```

Dit is natuurlijk niet erg gebruiksvriendelijk! Hoe weet je dat je een woord moet invoeren en niet een hele zin of een getal? Daarvoor kun je de input-functie met "prompt" gebruiken. Een prompt is tekst die vóór het invoervenster (in interactieve mode) of op de eerste lege regel (in script mode) wordt gezet.

## Voorbeeld:

In:

```
naam = input('Voer je naam in :')
print('Welkom' , naam)
```

Als je na de prompt je naam invoert dan word je door Python persoonlijk welkom geheten. In dit voorbeeld heeft Melissa dat gedaan.

Out:

```
Voer je naam in : Melissa
Welkom Melissa
```

Stel dat je nu het volgende programma schrijft om een getal dat je invoert te kwadrateren:

In:

```
getal = input('Voer een getal in :')
print('Het kwadraat van' , getal , 'is : ' , getal ** 2)
```

Nadat we het getal 5 invoeren en op 'Enter' drukken krijgen we helaas een foutmelding:

Out:

```
-----
TypeError                                 Traceback (most recent call last)
Input In [], in <cell line: 2>()
      1 getal = input('Voer een getal in :')
```

```
----> 2 print('Het kwadraat van' , getal , 'is :' , getal ** 2)
```

TypeError: unsupported operand type(s) for \*\* or pow(): 'str' and 'int'

Wat is hier aan de hand? In de laatste regel van het uitvoerblok zie je dat we de macht van een string hebben proberen te bepalen en dat lukt natuurlijk niet. De reden is dat de input-functie altijd een string oplevert! Met de opgedane kennis uit het vorige hoofdstuk weten we dat we het type van een stringvariabele kunnen wijzigen in een integer met behulp van de cast-functie `int()`. We passen ons programma dus als volgt aan:

```
In:
getal = int(input('Voer een getal in :'))
print('Het kwadraat van' , getal , 'is :' , getal ** 2)
```

Nu krijgen we het verwachte resultaat:

```
Out:
Voer een getal in : 5
Het kwadraat van 5 is : 25
```

## Opgave

Schrijf een programma dat de gebruiker vraagt 2 getallen in te voeren en als resultaat de som en het product van die getallen toont met een [passende tekst](#).

Het bovenstaande voorbeeld is natuurlijk niet erg robuust. Als een gebruiker van jouw programma per ongeluk of expres een ongeldige waarde invoert na de prompt (bijvoorbeeld de letter 'a') zal het programma "crashen". Daarom moet invoer altijd gecontroleerd worden op geldigheid. Later in de cursus zullen we manieren behandelen om correcte invoer af te dwingen.

## 4.5 Samenvatting

In dit hoofdstuk heb je geleerd wat expressies en operatoren zijn. De waarde van een expressie kan toegekend worden aan een variabele, maar kan ook gebruikt worden als argument van een functie, zoals bijvoorbeeld de print-functie.

Expressies worden opgebouwd uit atomaire expressies en operatoren. Prioriteit en evaluatievolgorde van operatoren met dezelfde prioriteit bepalen de waarde van een expressie.

Tenslotte heb je geleerd hoe je gegevens kunt invoeren in een Python programma.

## 4.6 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studiecoach.

### Opgave 1

Maak een programma wat een getal inleest met `input()`, En vervolgens dit verandert naar een integer met de `int()`functie.

Wat gebeurt er als er geen geldig getal ingevoerd wordt?

### Opgave 2

Maak een programma wat iemand om zijn leeftijd vraagt en vervolgens de leeftijd van die persoon over 30 jaar weergeeft.

### Opgave 3

De formule om temperatuur in celsius om te rekenen naar fahrenheit is als volgt:  $celsius * (9/5) + 32$

Maak nu een programma wat de gebruiker vraagt om een temperatuur in celsius in te voeren, en vervolgens de temperatuur in fahrenheit weergeeft.

## 5. Algoritmen

In de vorige hoofdstukken heb je kennis gemaakt met de programmeertaal Python. Nu doen we een stapje terug en gaan we bekijken wat programmeren is.

Je kunt programmeren vergelijken met autorijden. Als je deze cursus met succes afrondt heb je feitelijk je (Python) rijbewijs gehaald en kun je zelfstandig aan de slag als programmeur. Iedere automobilist weet echter dat je pas na enkele jaren ervaring op de weg goed leert rijden (en sommigen leren het nooit!). Als programmeur moet je dus heel veel code schrijven, bestuderen en testen om een professional te worden.

Een tweede overeenkomst is dat een ervaren chauffeur zonder grote problemen in een andere auto kan rijden. Het is hoogstens even wennen aan de plek van bepaalde knopjes en schakelaars. Net zo kan een software professional vrij simpel overstappen naar een nieuwe programmeertaal omdat programmeerervaring veel belangrijker is dan de kennis van een specifieke taal!

Het doel van deze cursus is dus om te leren programmeren *met* Python (let op de volgorde!) In dit hoofdstuk maak je kennis met een techniek die je kan helpen om een correct programma te maken: het Programma Structuur Diagram of PSD. Merk op dat deze techniek onafhankelijk is van de programmeertaal!

### 5.1 Algoritmen en programma's

Als we een persoon of een machine een taak willen laten uitvoeren moeten we duidelijk maken hoe dat moet worden gedaan, met andere woorden welke handelingen moeten worden verricht. Zo'n beschrijving heet een algoritme. Een algoritme bestaat uit een reeks stappen (of instructies) die, wanneer ze precies worden gevolgd of uitgevoerd, als resultaat hebben dat de taak (of het proces) verricht wordt. De uitvoerder van een algoritme heet een processor.

Voorbeelden van algoritmen zijn: een breipatroon om een das te breien, een handleiding om een model te bouwen, een recept om een brood te bakken en bladmuziek om een sonate te spelen. Bij deze voorbeelden kan sprake zijn van een menselijke processor, maar de processor kan ook een elektronische of mechanische machine zijn (denk aan breimachines, industriële robots, broodmachines, pianola's). Bij de meeste taken die te maken hebben met de verwerking van gegevens kan de processor zowel een mens als een computer zijn.

In alle gevallen moet het algoritme op zo'n manier worden geformuleerd dat de processor het kan begrijpen en uitvoeren. We zeggen dat de processor het algoritme moet kunnen *interpreteren*, wat inhoudt dat de processor in staat moet zijn om:

- te begrijpen wat elke stap betekent en
- de bijbehorende handeling te verrichten

Als de processor een computer is moet het algoritme in de vorm van een *programma* worden uitgedrukt. Een programma wordt geschreven in een *programmeertaal* en het formuleren van een algoritme als een programma heet *programmeren*. Elke stap (instructie) van het algoritme wordt door een statement in het programma uitgedrukt. Een programma bestaat dus uit statements die elk een bepaalde handeling specificeren die de computer moet uitvoeren.

De aard van zulke statements in een programma hangt af van de programmeertaal die gebruikt wordt. Er bestaat een groot aantal programmeertalen en elke taal heeft zijn eigen repertoire van statements. De eenvoudigste talen, die *machinetalen* heten, zijn zo ontworpen dat elk statement direct door de computer kan worden geïnterpreteerd, dat wil zeggen dat de CPU (Central Processing Unit) in staat is elk statement te begrijpen en de bijbehorende handelingen te verrichten. Omdat de statements echter zo eenvoudig zijn (bijvoorbeeld tel twee getallen op), kunnen ze slechts een zeer klein deel van een algoritme uitdrukken. Dat betekent dat er voor het uitdrukken van de meeste algoritmen een groot aantal statements nodig is. Programmeren in een machinetaal gaat daarom zeer moeizaam.

Om het programmeren te vergemakkelijken zijn er andere soorten programmeertalen ontwikkeld, zoals FORTRAN, COBOL, ALGOL, C, Basic, Pascal, C++, Java, C#, Python, e.d.. Deze hogere talen zijn geschikter dan machinetalen, omdat met ieder statement een grotere stap van het algoritme wordt geformuleerd. Natuurlijk moet elk statement nog steeds door de computer worden geïnterpreteerd, maar hiervoor hoeft niet noodzakelijk de CPU te worden aangepast. Een goedkoper en flexibeler alternatief is het vertalen van programma's naar machinetaal voordat ze uitgevoerd worden. De vertaling bestaat uit de verandering van elk statement uit het programma in de hogere taal in een gelijkwaardige reeks machinetaalstatements. Deze machinetaalstatements kunnen dan door de CPU worden geïnterpreteerd.

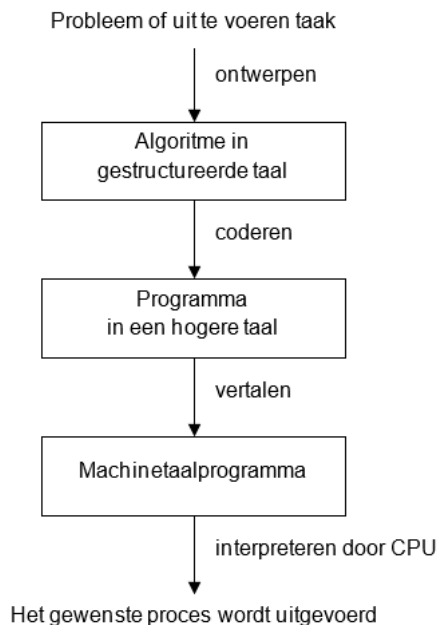
Het vertalen van een hogere taal naar een machinetaal is een taak die de computer zelf kan verrichten. Een programma hiervoor wordt een vertaler (*compiler*) genoemd. Hogere talen vergemakkelijken in feite het programmeerwerk, ten koste van de interpretatie en de vertaling, die moeilijker worden. Omdat het eerste door mensen gedaan wordt en het laatste door computers is de keuze niet moeilijk.



Omdat het - zelfs bij het gebruik van een hogere programmeertaal - meestal niet zo eenvoudig is een correct programma te schrijven doen we dit vaak in 2 stappen:

- Eerst beschrijven we het uit te voeren algoritme in pseudo-taal of gestructureerde taal, waarbij een beperkt aantal besturingsstructuren wordt gebruikt. We noemen deze stap in het vervolg ontwerpen.
- Vervolgens zetten we het algoritme om in een programma (in een hogere taal). Deze stap noemen we hierna coderen.

Het gehele proces van probleem naar uitvoering is te zien in onderstaande figuur.



## 5.2 Algoritmen en PSD's

Bij de beschrijving van een algoritme in gestructureerde taal of pseudo-taal wordt gebruik gemaakt van een natuurlijke taal (bijv. Nederlands), maar is slechts een beperkt aantal taalelementen (sleutelwoorden) en besturingsstructuren toegestaan.

Pseudo-taal lijkt op een programmeertaal, maar kan niet gecompileerd worden. Gestructureerde taal is minder formeel omdat in de beschrijving ook bijvoeglijke naamwoorden, voegwoorden en andersoortige weinigzeggende woorden mogen voorkomen, om voor de lezer een beter lopend verhaal te krijgen.

In de beschrijving kan men gebruik maken van werkwoorden, objecten en gegevenselementen, besturingsstructuren (DOE .. ZOLANG .. EIND-DOE, HERHAAL .. TOT, ALS .. DAN .. ANDERS .. EIND-ALS), vergelijkende bewerkingen (gelijk, groter, ..) en rekenkundige bewerkingen (optellen, vermenigvuldigen, ..). Specifieke symbolen en constructies uit een programmeertaal moeten zoveel mogelijk worden vermeden.

In 1966 hebben Böhm en Jacopini bewezen dat 3 basisstructuren nodig en voldoende zijn om elk willekeurig algoritme te construeren, nl. sequentie (openvolging), selectie (keuze) en iteratie (herhaling).

Bij het ontwerpen van een algoritme maken we gebruik van *top-down decompositie* en *stapsgewijze verfijning*. Top-down decompositie ('verdeel en heers') houdt in dat het algoritme in steeds kleinere delen (subalgoritmen) wordt opgesplitst. Stapsgewijze verfijning houdt in dat elk deel steeds gedetailleerder (nauwkeuriger) wordt beschreven.

Om de leesbaarheid van een algoritme in gestructureerde taal te vergroten kan het ook grafisch worden weergegeven in een zgn. *Programma Structuur Diagram* (PSD). Een dergelijk diagram wordt ook vaak aangeduid met *Nassi-Schneiderman* diagram, naar de bedenkers ervan.

Elke besturingsstructuur heeft in een PSD een grafisch symbool, zodat het aantal noodzakelijke sleutelwoorden in de tekst beperkter is dan bij gestructureerde taal. In de volgende paragrafen worden deze grafische symbolen beschreven.

We bekijken eerst de drie basisstructuren. Daarna behandelen we enkele andere besturingsstructuren, die in bijna elke programmeertaal voorkomen.

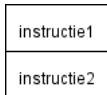
Om PSD's eenvoudig te kunnen vervaardigen en snel te kunnen wijzigen is het handig over een specifiek tekenprogramma te beschikken met meer mogelijkheden dan een pure tekstverwerker (in moderne case-tools zijn dergelijke programma's meestal opgenomen). Wij maken gebruik van het (gratis) programma Structorizer (zie handleiding installatie software).

## 5.3 De 3 basisstructuren van een PSD

Hieronder bespreken we de 3 basisstructuren van een PSD, te weten: sequentie, selectie en iteratie.

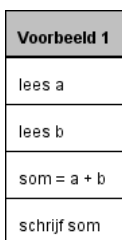
### Sequentie (opvolging)

*Sequentie* is het na elkaar uitvoeren van instructies. In een PSD wordt de sequentie van instructie1 en instructie2 aangegeven met onderstaand symbool.



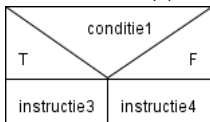
### Voorbeeld 1

Bepaal de som van 2 getallen en druk die af.



### Selectie (keuze)

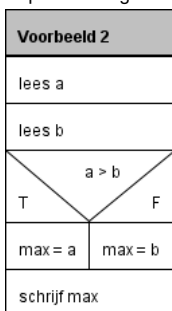
*Selectie* is een tweevoudige keuze op basis van een conditie, die wel of niet waar is. In een PSD wordt selectie met onderstaand symbool aangegeven. Als conditie1 waar is (T) wordt instructie3 uitgevoerd, zo niet (F) dan wordt instructie4 uitgevoerd.



Een speciaal geval is de enkelvoudige selectie waarbij instructie4 niet voorkomt. In dat geval wordt er alleen iets uitgevoerd (namelijk instructie3) als conditie1 waar is.

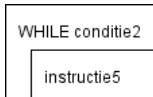
### Voorbeeld 2

Bepaal van 2 getallen welke de grootste is en druk die af.



### Iteratie (herhaling)

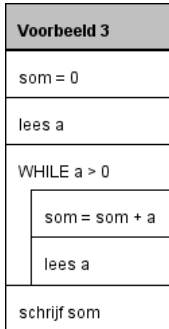
*Iteratie* is het meerdere malen uitvoeren van een instructie afhankelijk van een zekere conditie. In een PSD wordt iteratie met onderstaand symbool aangegeven. Zolang conditie2 waar is wordt instructie5 uitgevoerd.



Let erop dat instructie5 helemaal niet wordt uitgevoerd als conditie2 bij het bereiken van de while-structuur onwaar is. We spreken daarom ook wel over iteratie met test vooraf (vergelijk dit met iteratie met test achteraf in de volgende paragraaf). Als conditie2 altijd waar blijft wordt instructie5 steeds weer uitgevoerd en komt er geen einde aan de uitvoering van het algoritme. We spreken in dat geval van een 'oneindige loop'.

## Voorbeeld 3

Tel een (willekeurig grote) rij positieve<sup>1</sup> getallen bij elkaar op. Na een 0 wordt de som afgedrukt en stopt het algoritme.



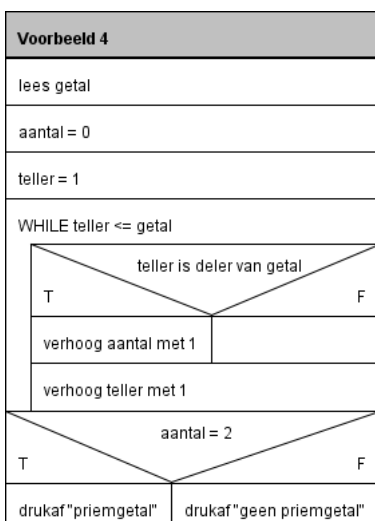
Zoals uit de voorbeelden blijkt mogen we in de 3 besturingsstructuren een instructie vervangen door een besturingsstructuur. We kunnen dus elk algoritme construeren met slechts drie soorten 'blokjes' (vergelijk dit met Lego).

## Voorbeeld 4

Bepaal of een gegeven getal een priemgetal<sup>2</sup> is en druk het resultaat af.

We kunnen dit probleem oplossen door te bekijken hoeveel delers (opgeslagen in de variabele aantal) het getal heeft. Daartoe gaan we van elk geheel getal (bijgehouden in de variabele teller) tussen 1 en het gegeven getal bekijken of het een deler is (iteratie). Zo ja, dan verhogen we het aantal delers met 1, zo nee, dan gebeurt er niets (enkelvoudige selectie).

Vooraf moeten we natuurlijk het aantal delers op 0 zetten. Als na afloop van de iteratie het aantal delers 2 is drukken we af "priemgetal" en anders "geen priemgetal".<sup>3</sup>



<sup>1</sup>Onder een positief getal wordt hier verstaan een getal dat groter of gelijk aan nul is.

<sup>2</sup>Een *priemgetal* is een positief geheel getal met precies 2 verschillende delers (nl. 1 en zichzelf). De getallen 2, 3, 5, 7, 11, ... zijn dus priemgetallen. Let erop dat het getal 1 geen priemgetal is!

<sup>3</sup>Het gegeven algoritme is zeker niet het meest efficiënt maar dient uitsluitend als voorbeeld.

## 5.4 Overige besturingsstructuren in een PSD

### Meervoudige selectie

Als er verschillende selecties na elkaar moeten worden uitgevoerd waarbij de condities elkaar uitsluiten, dan is er sprake van *geneste selectie*. Vaak kan deze structuur eenvoudiger worden gerepresenteerd m.b.v. een *meervoudige selectie*. De laatstgenoemde structuur wordt in een PSD aangegeven met onderstaand symbool.

conditieA	conditieB	...	conditieZ	anders
instructieA	instructieB	...	instructieZ	instructieAnders

### Voorbeeld 5

De brutoprijs van een artikel wordt ingelezen. Afhankelijk van het soort klant wordt er een korting gegeven van resp. 20% (soort = 1), 10% (soort = 2), 5% (soort = 3) of 0% (overige soorten). Bepaal de nettoprijs van het artikel en druk deze af.

Voorbeeld 5			
lees brutoprijs			
lees soort (klant)			
soort =			
1	2	3	overig
korting = 20%	korting = 10%	korting = 5%	korting = 0%
nettoprijs = brutoprijs * (1 - korting / 100)			
schrijf nettoprijs			

### Iteratie met test achteraf

Bij de *iteratie met test achteraf* wordt een instructie meerdere malen uitgevoerd afhankelijk van een zekere conditie. Deze structuur lijkt op de eerder behandelde (gewone) iteratie, maar controle vindt nu pas achteraf plaats. In een PSD wordt hiervoor onderstaand symbool gebruikt. Instructie6 wordt uitgevoerd totdat conditie3 waar is.

instructie6
UNTIL conditie3

Let op: instructie6 wordt altijd minstens eenmaal uitgevoerd, zelfs als conditie3 bij het bereiken van de structuur waar is!

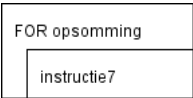
### Voorbeeld 6

Lees een willekeurige rij karakters in totdat je een 'Q' tegenkomt. Druk het totaal aantal gelezen karakters af.

Voorbeeld 6	
aantal = 0	
lees karakter	
verhoog aantal met 1	
UNTIL karakter is 'Q'	
schrijf aantal	
drukaf "karakters gelezen"	

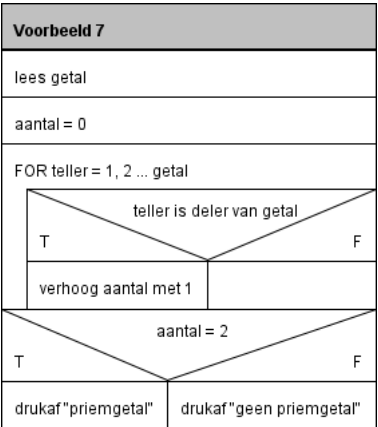
### Iteratie met bekend aantal herhalingen

Als het aantal malen dat een instructie moet worden uitgevoerd vooraf bekend is kunnen we een *iteratie met een bekend aantal herhalingen* toepassen. In een PSD wordt deze structuur aangegeven met onderstaand symbool. De klassieke werking van deze structuur is als volgt: eerst wordt een teller op een startwaarde gezet en begint de iteratie (d.w.z. wordt instructie7 uitgevoerd). Na elke iteratie wordt de waarde van de teller gewijzigd totdat de eindwaarde is bereikt en instructie7 voor de laatste keer is uitgevoerd. De opsomming wordt meestal genoteerd als: start ... einde, waarbij start en einde resp. de start- en eindwaarde voorstellen. Als start en eind gelijk zijn wordt de iteratie dus precies één keer uitgevoerd. Meer algemeen kan een opsomming ook worden bepaald door een gegeven (eindige) verzameling. In dat geval wordt instructie7 uitgevoerd voor elk element van die verzameling.



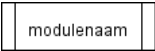
Voorbeeld 7

Bepaal of een gegeven getal een priemgetal is (zie voorbeeld 4)



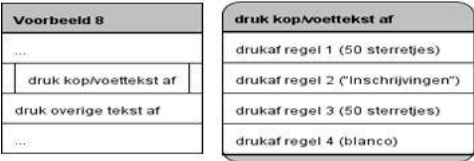
5.5 Modulen

Al eerder is vermeld dat de werkwijze die gevolgd wordt bij het ontwerpen van een algoritme bestaat uit top-down decompositie en stapsgewijze verfijning. We kunnen uiteraard de PSD's van de subalgoritmen (of modulen) die hierbij ontstaan combineren tot één groot PSD, maar het is vaak handiger en overzichtelijker om de 'losse' PSD's te laten bestaan en één overkoepelend PSD te maken. Hierin wordt dan de samenhang tussen de PSD's getoond, en wordt m.b.v. een speciale instructie aangegeven dat een module moet worden uitgevoerd (of aangeroepen). De speciale instructie wordt met onderstaand symbool aangegeven. Modulenaam moet overeenkomen met de PSD-naam van het subalgoritme, en de PSD van een module heeft ronde hoeken i.p.v. rechte hoeken (zie onderstaande voorbeelden).



Voorbeeld 8

Druk aan het begin van een overzicht van inschrijvingen een aantal standaard regels af.



Een extra voordeel van deze aanpak is dat een subalgoritme dat meerdere keren voorkomt slechts éénmaal in een PSD hoeft te worden opgenomen. Als we in bovenstaand voorbeeld ook aan het eind van het overzicht de regels willen afdrukken kunnen we dus volstaan met het toevoegen van één instructie in het linker PSD.

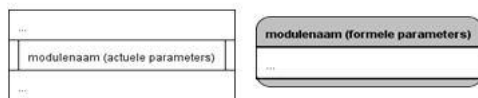
Een nog grotere flexibiliteit kan bereikt worden met behulp van parameters. Als de vier regels uit voorbeeld 8 meerdere keren moeten worden afgedrukt, maar de tekst op regel 2 in die gevallen verschillend is, wordt bij de aanroep de juiste tekst meegegeven.

## Voorbeeld 9

Druk aan het begin van elk hoofdstuk een aantal standaard regels af, waarin ook de titel van het hoofdstuk is opgenomen.



We onderscheiden hierbij de zogenaamde *formele* en *actuele parameters*. Formele parameters worden in de kop van de module na de modulenaam tussen haakjes vermeld. Ze mogen in de module worden gebruikt, maar hun waarde is onbekend totdat de module daadwerkelijk wordt aangeroepen. Actuele parameters worden in de speciale instructie tussen haakjes vermeld en nemen bij aanroep van de module de plaats in van de formele parameters. Het aantal formele en actuele parameters moet uiteraard wel gelijk zijn! In onderstaande figuur is een en ander samengevat.

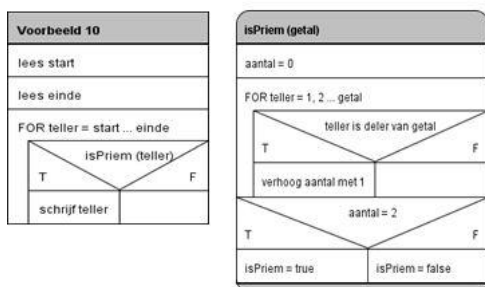


In bovenstaande voorbeelden is sprake van eenrichtingsverkeer. Een module wordt aangeroepen en krijgt soms aanvullende informatie (de actuele parameters). In veel gevallen zal de module ook informatie willen teruggeven (bijvoorbeeld het resultaat van een ingewikkelde berekening). Hiervoor is geen apart symbool gedefinieerd. We kunnen zelf beslissen of we in een PSD bij de aanroep van een dergelijke module wel of niet het symbool van de speciale instructie gebruiken<sup>4</sup>.

In de module moet worden aangegeven welke informatie wordt geretourneerd. Ook hiervoor bestaat geen apart symbool. In onderstaand voorbeeld is ervoor gekozen aan het eind van de module een waarde toe te kennen aan de modulenaam. Deze waarde wordt dan teruggegeven aan het aanroepende algoritme (soms wordt een instructie gebruikt van de vorm 'RETURN waarde').

## Voorbeeld 10

Druk alle priemgetallen af tussen twee gegeven gehele getallen.



## Toelichting

In het hoofdalgoritme worden eerst de twee grensgetallen ingelezen (resp. start en einde). We maken gebruik van een teller die in een iteratie alle gehele getallen tussen start en einde doorloopt. Als einde kleiner is dan start wordt de iteratie geen enkele keer uitgevoerd. Voor elke waarde van teller wordt in module isPriem gecontroleerd of het een priemgetal betreft. In deze module is getal de formele parameter, terwijl teller in het hoofdalgoritme de actuele parameter is. De module retourneert de waarde true als de actuele parameter een priemgetal is, respectievelijk false als dit niet het geval is. Deze waarde fungeert als conditie voor de selectie in het hoofdalgoritme.

Het afdrucken van de priemgetallen had natuurlijk ook door de module kunnen worden verzorgd (hoe?). Als we daarna echter hadden besloten om het algoritme alleen het aantal priemgetallen tussen de twee gegeven getallen te laten bepalen hadden we beide PSD's moeten aanpassen (ga na). Bij de hierboven gegeven PSD's kunnen we volstaan met een paar kleine aanpassingen in het hoofdalgoritme (welke?).

De conclusie is dat we ernaar moeten streven modules zo simpel mogelijk te houden om de kans op hergebruik te vergroten en in principe slechts één opdracht, controle, berekening en dergelijke te laten uitvoeren (dus bijvoorbeeld niet controle én afdrukken).

Merk tenslotte op dat teller zowel in het hoofdalgoritme als in de module voorkomt, maar dat het uiteraard verschillende data-elementen betreft. In principe is elk PSD autonoom en hebben de data-elementen in verschillende PSD's (met uitzondering van de parameters en de returnwaarde) niets met elkaar te maken (dus ook niet als ze toevallig dezelfde naam hebben).

<sup>4</sup>Als we het PSD tekenen met behulp van Structorizer is het soms niet mogelijk om de speciale instructie voor de aanroep van een module te gebruiken (zie voorbeeld 10).

## 5.6 Samenvatting

In dit hoofdstuk heb je geleerd wat programmeren is: het bedenken van correcte algoritmen om bepaalde problemen op te lossen of bepaalde resultaten (overzichten) te produceren en die algoritmen om te zetten naar de juiste code in de gebruikte programmeertaal.

Voor het ontwerpen van algoritmen kun je gebruik maken van Programma Structuur Diagrammen of PSD's. Deze techniek is met name bedoeld voor beginnende programmeurs, maar kan tevens nuttig zijn om ingewikkelde algoritmen te documenteren.

Elk algoritme kan opgebouwd worden uit 3 basisstructuren (sequentie, selectie en iteratie), maar de meeste programmeertalen ondersteunen ook nog enkele aanvullende structuren.

Grote algoritmen kunnen gesplitst worden in kleinere (sub)algoritmen (volgens het principe van "verdeel en heers"), die steeds concreter worden beschreven ("stapsgewijze verfijning").

## 5.7 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studietoelichting.

### Opgave 1

Maak een PSD voor het afdrukken van een driehoek van sterretjes:

```
*  
**  
***  
****
```

### Opgave 2

Voer een geheel getal in. Zolang het getal niet tussen 1 en 4 ligt wordt de boodschap "verkeerde invoer" getoond en opnieuw om invoer gevraagd. Tenslotte wordt het ingevoerde getal afgedrukt. Maak een PSD voor dit algoritme.

### Opgave 3

Voer een willekeurig aantal karakters in en stop na invoer van het '\$'-teken. Bepaal hoe vaak de letter 'a' of 'A' is ingevoerd en druk dit aantal af. Maak een PSD voor dit algoritme.

### Opgave 4

De vakantie-uitkering bedraagt 8% van het jaarsalaris, maar is minimaal € 1.000,- en maximaal € 2.500,-. Na invoer van het jaarsalaris wordt de vakantie-uitkering berekend en afgedrukt. Maak hiervoor een PSD.

### Opgave 5

Voer een willekeurig aantal (positieve en/of negatieve) getallen in en sluit af met een 0 (NUL). Bepaal het gemiddelde van alle positieve getallen en het gemiddelde van alle negatieve getallen en druk die af. Maak een PSD voor dit algoritme.

## Opgave 6

Een startkapitaal wordt vastgezet tegen een vast jaarlijks rentepercentage. Startkapitaal, rentepercentage en looptijd (in gehele jaren) worden ingevoerd. Bepaal het eindkapitaal en druk dit af. Maak hiervoor een PSD.

## Opgave 7

Voer een (onbekend) aantal gehele getallen in en sluit af met een 0 (NUL). Maak een PSD om de som te bepalen van MAXIMAAL de eerste 10 getallen. Let op: er kunnen ook minder dan 10 of zelfs helemaal geen getallen zijn ingevoerd vóór de 0!

## Opgave 8

Voer een (onbekend) aantal gehele getallen in en sluit af met een 0 (NUL). Maak een PSD waarin het volgnummer en de waarde van zowel het grootste als het kleinste getal wordt bepaald en afgedrukt. Let op: er kan ook onmiddellijk een afsluit-0 zijn ingevoerd !

## Opgave 9 (wordt in de les behandeld)

De kosten voor het huren van een auto moeten worden bepaald. Gebruik de volgende gegevens:

- Huur per dag: € 50.00
- Per dag is 100 km inbegrepen; elke extra km kost € 0.25
- Bij terugkomst wordt de tank gevuld op kosten van de huurder: per liter € 2.30

Het aantal huurdagen, de totaal gereden kilometers en de getankte liters benzine bij terugkomst worden ingevoerd. Bepaal het te betalen huurbedrag m.b.v. een PSD.

## Opgave 10 (wordt in de les behandeld)

Naast de vele andere producten verkoopt "Bouwmarkt 'Doe het Zelf'" ook multiplexplaten. Van die platen zijn de volgende twee standaardmaten beschikbaar:

- 150 cm breed en 450 cm lang
- 60 cm breed en 150 cm lang

Je kunt zo'n plaat tegen een geringe vergoeding op maat laten zagen. Bij de zaagafdeling geef je dan de gewenste maten van jouw plaat op. Deze maten (lengte en breedte) worden in de computer ingevoerd. Op grond van de ingevoerde maten berekent de computer vervolgens welke standaard plaat gebruikt moet worden en hoeveel dat gaat kosten. De prijs die je bij de kassa moet betalen is als volgt opgebouwd:

De klant betaalt voor een volledige standaard plaat; de prijzen daarvan zijn:

- 150 cm breed en 450 cm lang: € 25,-
- 60 cm breed en 150 cm lang: € 15,-

De klant betaalt voor het zagen een vast bedrag: € 10,-. Als er niet wordt gezaagd hoeft dat bedrag uiteraard niet te worden betaald. Analyseer dit probleem en maak een PSD. Je hoeft geen rekening te houden met een klant die meerdere platen uit een standaard plaat wil laten zagen.

# 6. Besturingsstructuren

In het vorige hoofdstuk heb je gezien welke besturingsstructuren gebruikt kunnen worden om een algoritme te ontwerpen. We gaan nu bekijken hoe die besturingsstructuren in Python zijn



geïmplementeerd. Eerst behandelen we de statements voor de basisstructuren (sequentie, selectie en iteratie) en daarna de belangrijkste overige besturingsstatements in Python.

De syntax van een statement geeft een gedetailleerde en precieze beschrijving van de opbouw van het statement. We zullen de volgende conventies hanteren:

- **Vetgedrukte** woorden en tekens zijn verplicht.
- *Cursieve woorden* geven aan waar iets moet worden ingevuld. Dit kan een variabelenaam, een waarde, een expressie, een conditie, een statement of iets anders zijn.
- Het "inspringen" in de syntax moet in de code worden gevolgd.

Onder inspringing (Engels: Indentation) verstaan we dat een regel minstens 1 extra voorloopspatie bevat dan de voorafgaande regel. Meestal gebruik je hiervoor de 'Tab'-toets, waarbij per tab 4 spaties wordt ingesprongen.

## Voorbeeld:

Het toekenningsstatement heeft de volgende syntax:

```
variabelenaam = expressie
```

## 6.1 Statements voor de basisstructuren

Python kent enkelvoudige (Engels: simple) en samengestelde statements (Engels: compound statements). Een enkelvoudig statement staat (bijna) altijd op 1 regel, terwijl een samengesteld statement meer dan 1 regel omvat. Het toekenningsstatement is een enkelvoudig statement.

Onderstaande code bestaat dus uit 2 statements, namelijk `a = 8` en `b = 3`. In tegenstelling tot de meeste andere programmeertalen hoeven statements in Python niet afgesloten te worden met een punt, puntkomma of een ander (zichtbaar) symbool. Regeleinden scheiden enkelvoudige statements.

```
In:
a = 8
b = 3
```

Het is mogelijk om meerdere enkelvoudige statements op 1 regel te zetten, maar dan moet je ze wel scheiden met behulp van een puntkomma (;). Voor de leesbaarheid van je code raden we dit echter af!

## 6.2 Het statementblok

Zet nu eens vóór de 2e regel van het vorige voorbeeld een spatie:

```
In:
a = 8
 b = 3
```

Als je deze snippet runt is het resultaat:

```
Out:
Input In []
      b=3
      ^
IndentationError: unexpected indent
```

Dit mag dus niet! Inspringing is een belangrijk onderdeel van de Python syntax en mag/moet alleen in bepaalde situaties worden toegepast.

Een *statementblok* bestaat uit 1 of meerdere statements (op verschillende regels) die evenveel inspringen. Dat is de Python implementatie van sequentie (opeenvolging).

Als we het PSD van Voorbeeld 1 uit het vorige hoofdstuk omzetten in Python code krijgen we:

```
In:
```

```
# Voorbeeld 1 : bepaal de som van 2 getallen (a en b) en druk die af

a = float(input('a ='))
b = float(input('b ='))
som = a + b
print('de som van a en b is :', som)
```

Je ziet dat we een paar aannames hebben gedaan, namelijk dat er een invoerprompt wordt afgedrukt bij het inlezen van de getallen en dat het decimale getallen betreft. Het PSD had eigenlijk nog iets gedetailleerder moeten zijn!

## Opgave

Werkt deze code ook als je op de 3e en 4e regel 'float' vervangt door ['int'](#)?

## 6.3 IF-statement

Selectie is in Python geïmplementeerd middels het (gewone) if-statement. Dit is een samengesteld statement met de volgende syntax:

```
if conditie : # conditie is een Boolese expressie
statementblok1 # wordt uitgevoerd als conditie True is
else :
statementblok2 # wordt uitgevoerd als conditie False is
```

Toelichting:

- Let op de verplichte dubbelepunten na de conditie en het keyword 'else'.
- Let op de verplichte inspringing van statementblok1 en statementblok2
- Het is niet verplicht om haakjes om de conditie te zetten (soms wel handig!)

Als we het PSD van Voorbeeld 2 uit het vorige hoofdstuk omzetten in Python code krijgen we:

```
In:
# Voorbeeld 2 : Bepaal van 2 getallen welke de grootste is en druk die af

a = float(input('a ='))
b = float(input('b ='))

if a > b :
    max = a
else :
    max = b

print('De grootste waarde is :', max)
```

Let op dat het statement op de laatste regel (de print-functie) niet inspringt! Dit statement valt dus niet onder het statementblok van de else-tak, en wordt altijd uitgevoerd.

De else-tak is overigens niet verplicht. Als je alleen statements hoeft uit te voeren indien conditie True is kun je de hele else-tak weglaten.

Voorbeeld:

```
In:
getal = float(input('Voer een positief getal in :'))
if getal <= 0 :
    print('Het getal is niet positief')
```

Als je na de prompt een positief getal invoert wordt er dus niets afgedrukt.

De regels voor het inspringen zorgen er overigens wel voor dat Python niet het bekende probleem van de hangende-else (Engels: dangling-else) kent. Bekijk het volgende voorbeeld om de prijs van een kaartje te berekenen:

```
In:
prijs = 12.50
lid = input('lid (J/N) :')
leeftijd = int(input('leeftijd :'))
```

```

if lid == 'N' :
    if leeftijd >= 65 :
        korting = 10
    else :
        korting = 25

print('De prijs is :', prijs * (1 - korting / 100))

```

Uit de inspringing blijkt dat de 'else' bij de eerste 'if' hoort en niet bij de tweede!

## 6.4 WHILE-statement

Iteratie (met test vooraf) is in Python geïmplementeerd middels het while-statement. Dit is een samengesteld statement met de volgende syntax:

**while** *conditie* : # conditie is een Boolese expressie  
*statementblok* # wordt uitgevoerd zolang *conditie* True is

Let ook hier op de verplichte dubbelpunt na de conditie en de verplichte inspringing van het statementblok. En ook hier zijn haakjes niet verplicht om de conditie.

In:

```

# Voorbeeld 3 : Tel een (willekeurig grote) rij positieve getallen bij
# elkaar op. Na een 0 wordt de som afgedrukt en stopt het programma.

som = 0
a = float(input('a = '))

while a > 0 :
    som += a
    a = float(input('a = '))

print('De som is :', som)

```

## Opgave

Maak nu zelf een Python programma om te bepalen of een ingevoerd positief geheel getal een priemgetal is. Gebruik het PSD van voorbeeld 4 uit het vorige hoofdstuk. [Test of het werkt!](#)

## 6.5 Uitgebreide IF-statement

Meervoudige selectie is eigenlijk het na elkaar uitvoeren van (gewone) tweevoudige selecties. In het vorige hoofdstuk (voorbeeld 5) zagen we hoe de nettoprijs van een artikel berekend wordt als er verschillende soorten klanten zijn met een eigen kortingspercentage. We zouden dat algoritme als volgt naar Python code kunnen omzetten:

In:

```

# Voorbeeld 5 : Oplossing met gewone selectie

brutoprijs = float(input('brutoprijs = '))
soort = int(input('soort afnemer = '))

if soort == 1 :
    korting = 20
else :
    if soort == 2 :
        korting = 10
    else :
        if soort == 3 :
            korting = 5
        else :
            korting = 0

nettoprijs = brutoprijs * (1 - korting / 100)
print('De nettoprijs voor deze klant is :', nettoprijs)

```

Door het verplichte inspringen "loopt de code steeds verder naar rechts". Veel programmeertalen hebben een apart statement voor meervoudige selectie, maar bij Python is dit geïmplementeerd door uitbreiding van het if-statement. De syntax is als volgt:

```
if conditie1 :
    statementblok1 # wordt uitgevoerd als conditie1 True is
elif conditie2 :
    statementblok2 # wordt uitgevoerd als conditie2 True is
...
elif conditieL :
    statementblokL # wordt uitgevoerd als conditieL True is
else :
    statementsblokE # wordt uitgevoerd als conditieL False is
```

Met behulp van het uitgebreide if-statement krijgen we de volgende oplossing:

```
In:
# Voorbeeld 5 : Oplossing met meervoudige selectie

brutoprijs = float(input('brutoprijs ='))
soort = int(input('soort afnemer ='))

if soort == 1 :
    korting = 20
elif soort == 2 :
    korting = 10
elif soort == 3 :
    korting = 5
else :
    korting = 0

nettoprijs = brutoprijs * (1 - korting / 100)
print('De nettoprijs voor deze klant is :', nettoprijs)
```

## 6.6 FOR-statement

Voor een iteratie met een bekend aantal herhalingen gebruiken we in Python het for-statement. Dit is een samengesteld statement met de volgende syntax:

```
for variabelenaam in rij : # rij is een geordende collectie waarden
    statementblok # wordt uitgevoerd voor elke waarde in de rij
```

De waarden van de rij worden achtereenvolgens aan de variabele toegekend. Na elke toekenning worden de statements in het statementblok uitgevoerd.

De rij wordt meestal gemaakt met de range-functie. Deze functie genereert een rij opeenvolgende getallen en heeft de volgende syntax:

```
range(start, einde)
```

Dit levert de rij start, start+1, start+2, ..., einde-1 op. De waarde van einde zit er dus niet in!

Voorbeeld:

```
range(2, 8) geeft de rij: 2, 3, 4, 5, 6, 7
```

Als je de startwaarde weglaat neemt Python aan dat die 0 is (de default). De aanroep range(5) levert dus de rij: 0, 1, 2, 3, 4 op.

Je kunt ook de stapgrootte toevoegen. De aanroep range(1, 8, 2) levert de rij: 1, 3, 5, 7 op. In dit geval moet je de startwaarde altijd meegeven, omdat Python anders niet snapt wat je bedoelt.

We kunnen nu het algoritme van voorbeeld 7 uit het vorige hoofdstuk omzetten naar Python code:

```
In:
# Voorbeeld 7 : Bepaal of een positief geheel getal een priemgetal is

getal = int(input('getal ='))
aantal = 0

for teller in range(1, getal+1, 1) :
    if getal % teller == 0 :
```

```
aantal += 1

if aantal == 2 :
    print("priemgetal")
else :
    print("geen priemgetal")
```

## 6.7 BREAK-statement

Het break-statement heeft de volgende syntax:

### break

Dit statement kan alleen gebruikt worden in een lus (dus binnen een while- of een for-statement) en zorgt ervoor dat je 'uit de lus springt'. In onderstaand voorbeeld spring je uit de for-lus als teller de waarde 4 heeft.

In:

```
for teller in range(1, 6, 1) :
    if teller == 4 :
        break
    print(teller)
```

Out:

```
1
2
3
```

## 6.8 CONTINUE-statement

Het continue-statement heeft de volgende syntax:

### continue

Net als het break-statement kan het continue-statement alleen gebruikt worden in een lus. Dit statement zorgt ervoor dat resterende statements binnen de lus worden overgeslagen en met de volgende lusuitvoering wordt gestart. In onderstaand voorbeeld wordt het print-statement dus overgeslagen als teller de waarde 4 heeft.

In:

```
for teller in range(1, 6, 1) :
    if teller == 4 :
        continue
    print(teller)
```

Out:

```
1
2
3
5
```

## Opgave

Stel dat we dit voorbeeld implementeren met behulp van een while-statement (zie onderstaande code).

In:

```
teller = 1
while teller < 6 :
    if teller == 4 :
        continue
    print(teller)
    teller += 1
```

[Is het resultaat gelijk?](#)

## 6.9 Samenvatting

In dit hoofdstuk zijn de belangrijkste besturingsstatements van Python behandeld. In principe kun je hiermee elk algoritme omzetten in Python code. In volgende hoofdstukken zullen nog enkele aanvullende statements worden behandeld, die bijvoorbeeld nodig zijn als je bibliotheekfuncties wil gebruiken of nuttig zijn bij foutafhandeling.

## 6.10 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studietoelichting.

### Opgave 1

Schrijf een programma waarbij je twee gehele positieve getallen kunt invoeren. Het programma bepaalt en toont of het ene getal een veelvoud is van het andere getal.

### Opgave 2

De basisprijs van een bioscoopkaartje is 12 euro. Kinderen tot 5 jaar zijn gratis en kinderen van 5 tot 12 jaar betalen de halve prijs. Personen tussen 13 en 54 jaar moeten de volle prijs betalen en vanaf 55 jaar is de toegang weer gratis.

Maak een programma dat de te betalen prijs afdruckt nadat je de leeftijd hebt ingevoerd.

### Opgave 3

Schrijf een programma dat 3 gehele getallen sorteert. De getallen worden ingevoerd en respectievelijk opgeslagen in de variabelen num1, num2 en num3. Het programma sorteert de getallen zodanig dat na afloop  $\text{num1} \leq \text{num2} \leq \text{num3}$ .

### Opgave 4 (wordt in de les behandeld)

Schrijf een programma dat gehele getallen leest, het totaal en het gemiddelde van deze getallen berekent en afdruckt, waarbij 0 niet wordt meegeteld. Het programma eindigt met invoer 0.

### Opgave 5 (wordt in de les behandeld)

Schrijf een programma dat de tafel van "factor" afdruckt, nadat je een geheel getal tussen 0 en 10 (de "factor") hebt ingevoerd.

Bijvoorbeeld als factor = 5:

Out:

1 x 5 = 5

2 x 5 = 10

3 x 5 = 15    # enz.

### Opgave 6

Schrijf een programma met de volgende functionaliteit:

- Eerst moet je een willekeurig getal invoeren (het "zoekgetal").
- Daarna moet je een geheel getal tussen 0 en 10 invoeren (het "aantal").
- Tenslotte voer je "aantal" getallen in (dus als aantal = 5 moet je 5 getallen invoeren).

Het programma drukt alle getallen af die kleiner zijn dan het zoekgetal.

## 7. Functies

In de vorige hoofdstukken hebben we al verschillende functies gebruikt, zoals de print-functie, de type-functie, de input-functie en de typecasting-functies. Een Python functie is de implementatie van een sub algoritme (module) en heeft dezelfde doelstelling: hergebruik en het verminderen van complexiteit.

In dit hoofdstuk leren we eerst hoe we zelf een functie kunnen maken en gebruiken. Daarna gaan we kijken welke functies al in Python aanwezig zijn.

### 7.1 Zelf gedefinieerde functies

Er bestaan functies met en zonder parameters. In deze paragraaf wordt er gekeken naar hoe deze omgezet kunnen worden naar Python code.

De syntax van een functie is als volgt:

```
def functienaam(formele parameters) : # header
statementblok # body
```

Een functienaam moet aan dezelfde regels voldoen als een variabelenaam: alleen letters, cijfers en underscores ('\_') zijn toegestaan en de naam mag niet beginnen met een cijfer. Bovendien mag een functienaam niet al in gebruik zijn als variabelenaam.

Tussen de haakjes na de functienaam staan 0, 1 of meer parameters, gescheiden door komma's (de *formele parameters*). De combinatie van naam en parameters wordt de *header* of *signatuur* van de functie genoemd. Het statementblok heet de *body* van de functie. Bij aanroep van de functie moet voor elke formele parameter een *actuele parameter* (een waarde of een expressie) worden ingevuld. Dit worden ook wel de *argumenten* van een functie genoemd. Daarna worden alle statements van het statementblok uitgevoerd.

De syntax voor het *aanroepen* (Engels: *callen*) van een functie is als volgt:

```
functienaam(actuele parameters)
```

Als een functie geen parameters heeft bestaat de aanroep dus uit de functienaam gevolgd door een openings- en een sluihaakje. De volgorde van de actuele parameters moet aansluiten bij de volgorde van de formele parameters. De eerste waarde (dus de eerste actuele parameter) wordt toegekend aan de eerste formele parameter, de tweede waarde aan de tweede formele parameter, enzovoorts. Er zijn echter enkele bijzondere situaties bij de toekenning van waarden aan parameters die we later in dit hoofdstuk zullen behandelen. Het betreft formele parameters met een default waarde en de toepassing van keyword argumenten.

De definitie van een functie moet in de code vóór de aanroep van de functie staan. Python slaat echter de statements van de functiebody over en voert die pas uit bij aanroep van de functie. We bekijken nu hoe de voorbeelden 8 en 9 uit het vorige hoofdstuk (respectievelijk een functie zonder en met parameters) kunnen worden omgezet naar Python code:

#### Voorbeeld 8: functie zonder een parameter

In:

```
def printKop() :
    print(" " * 50)
    print("Inschrijvingen")
    print(" " * 50)
    print()

# hoofdprogramma
print("tekst vooraf")
print()
printKop()
print("tekst achteraf")
```

Out:

```
tekst vooraf
```

```
*****
Inschrijvingen
*****
```

tekst achteraf

### Voorbeeld 9: functie met een parameter

In:

```
def printKop(tekst) :
    print("*"*50)
    print(tekst)
    print("*"*50)
    print()

# hoofdprogramma
print("tekst vooraf")
print()
printKop("Inleiding")
print("tekst inleiding")
print()
printKop("Hoofdstuk 1")
print("tekst hoofdstuk 1")
```

Out:

tekst vooraf

\*\*\*\*\*

Inleiding

\*\*\*\*\*

tekst inleiding

\*\*\*\*\*

Hoofdstuk 1

\*\*\*\*\*

tekst hoofdstuk 1

Een functie kan een waarde teruggeven met behulp van het return-statement. Dit statement heeft de volgende

**return *expressie***

Als er geen expressie achter het keyword 'return' staat geeft een functie de waarde None terug. Dit is ook het geval als er helemaal geen return-statement in de body van de functie voorkomt.

**Voorbeeld 10 uit het vorige hoofdstuk kan als volgt worden omgezet in Python code:**

***Functie met parameter en een returnwaarde***

In:

```
def isPriem(getal) :
    aantal = 0
    for teller in range(1, getal + 1, 1) :
        if getal % teller == 0 :
            aantal += 1
    if aantal == 2 :
        return True
    else :
        return False

# hoofdprogramma
start = int(input('start = '))
einde = int(input('einde = '))

print("priemgetallen tussen", start, "en", einde, ":")

for getal in range(start, einde + 1, 1) :
    if isPriem(getal) :
        print(getal)
```



```

Out:
start = 70
einde = 90
priemgetallen tussen 70 en 90 :
71
73
79
83
89

```

## 7.2 Functies nader bekeken

In de al eerder genoemde Python Style Guide staat dat de eerste regel van de functiebody een zogenaamde *docstring* moet zijn met een korte omschrijving van de functie. Een docstring is tekst (die mag bestaan uit meerdere regels) tussen 3 dubbele aanhalingstekens:

```

"""tekst"""

```

Je kunt in de definitie van een functie 1 of meerdere parameters een default waarde geven. In een eerdere opgave heb je de inhoud van een cilinder met straal  $rr$  en hoogte  $hh$  berekend:

$inhoud = \pi r^2 h$  waarbij  $\pi = 3.14159$  (bij benadering).

Stel dat we hiervoor de volgende functie hebben gemaakt:

```

In:
PI = 3.14159    # PI is een constante

def bereken_inhoud_cilinder(straal, hoogte) :
    """Bereken de inhoud van een cilinder"""
    inhoud = PI * hoogte * straal ** 2
    return(inhoud)

```

Als je deze functie aanroept met 2 argumenten (straal=0.5 en hoogte=2) dan gaat het goed:

```

In:
print(bereken_inhoud_cilinder(0.5, 2))

```

```

Out:
1.570795

```

Maar als je een van de argumenten vergeet krijg je een foutmelding:

```

In:
print(bereken_inhoud_cilinder(3))

```

```

Out:
-----
TypeError                                 Traceback (most recent call last)
Input In [], in <cell line: 1>()
----> 1 print(bereken_inhoud_cilinder(3))

TypeError: bereken_inhoud_cilinder() missing 1 required positional argument: 'hoogte'

```

We kunnen onze inhoudsfunctie als volgt aanpassen, waarbij we zowel de straal als de hoogte default waarde 1 geven:

```

In:
PI = 3.14159    # PI is een constante

def bereken_inhoud_cilinder(straal=1, hoogte=1) :
    inhoud = PI * hoogte * straal ** 2
    return(inhoud)

```

Je kunt deze functie nu aanroepen met 0, 1 of 2 argumenten. Als een argument ontbreekt wordt de defaultwaarde van de parameter gebruikt, waarbij de aanwezige argumenten van links naar rechts aan de parameters worden toegekend.

```
In:
print(bereken_inhoud_cilinder())          # gebruikt beide defaultwaarden
print(bereken_inhoud_cilinder(0.5))      # gebruikt defaultwaarde hoogte
print(bereken_inhoud_cilinder(0.5, 2))   # gebruikt geen defaultwaarden
```

```
Out:
3.14159
0.7853975
1.570795
```

Als een functie parameters heeft met en zonder defaultwaarde is het gebruikelijk om de parameters zonder defaultwaarde aan de linkerkant te zetten (vraag: waarom is dat logisch?).

Stel dat we voor de straal de defaultwaarde willen gebruiken en alleen de waarde van de hoogte bij de aanroep van de functie willen meegeven. Dat lukt niet met uitsluitend defaultwaarden!

Je kunt echter ook keyword argumenten (Engels: keyword arguments) gebruiken om argumenten aan een functie door te geven:

```
In:
print(bereken_inhoud_cilinder(0.5, 2))
print(bereken_inhoud_cilinder(hoogte=2, straal=0.5))
```

```
Out:
1.570795
1.570795
```

Beide aanroepen van de functie `bereken_inhoud_cilinder()` leveren dus precies hetzelfde resultaat op, maar bij de eerste aanroep hebben we gebruikgemaakt van positionele argumenten en bij de tweede aanroep van keyword argumenten.

Als een functieaanroep zowel positionele als keyword argumenten bevat moeten alle keyword argumenten na alle positionele argumenten (waarvoor we dus geen parameternamen meegeven) worden geplaatst. Doen je dit niet dan krijg je een foutmelding (`SyntaxError`):

```
In:
def bereken_inhoud_blok(lengte, breedte, hoogte) :
    inhoud = lengte * breedte * hoogte
    return(inhoud)
```

```
bereken_inhoud_blok(hoogte=3, 1, 2)
```

```
Out:
Input In []
      bereken_inhoud_blok(hoogte=3, 1, 2)
                              ^
```

```
SyntaxError: positional argument follows keyword argument
```

## Opgave

Doen een functieaanroep van `bereken_inhoud_cilinder()` waarbij alleen de waarde van de hoogte als argument wordt meegegeven en voor de straal de defaultwaarde wordt gebruikt.

[Controleer hier je antwoord.](#)

### Voorbeeld

Bij de print-functie worden vaak keyword argumenten gebruikt om de uitvoer op een bepaalde manier weer te geven. Zo hebben we al diverse keren gezien dat de print-functie een spatie zet tussen de waarden van de argumenten en dat de uitvoer na elke aanroep begint op een nieuwe regel. Dit zijn namelijk de defaultwaarden van de parameters 'sep' en 'end'.

```
In:
for teller in range(6) :
    print(teller, end='; ')
```

```
Out:
0; 1; 2; 3; 4; 5;
```

In plaats van de defaultwaarde '\n' (wat nieuwe regel betekent) hebben we 'end' de waarde ';' (een puntkomma gevolgd door een spatie) toegekend.

Op dezelfde manier kun je 'sep' (een afkorting voor het Engelse separator) een andere waarde toekennen:

```
In:
print(1, 2, 3, sep=';')
```

```
Out:
1;2;3
```

In veel objectgeoriënteerde talen hoeft een functienaam niet uniek te zijn maar alleen de signatuur (naam en parameters). Als twee functies dezelfde naam hebben spreken we van *overladen* (Engels: overloading). In Python moet de functienaam wel uniek zijn, wat geïllustreerd wordt door het volgende voorbeeld:

```
In:
def hallo() :
    print('hallo')

def hallo(tekst) :
    print ('hallo', tekst)

hallo()
hallo('Larissa')
```

```
Out:
```

```
-----
TypeError                                 Traceback (most recent call last)
Input In [], in <cell line: 7>()
      4 def hallo(tekst) :
      5     print ('hallo', tekst)
----> 7 hallo()
      8 hallo('Larissa')
```

```
TypeError: hallo() missing 1 required positional argument: 'tekst'
```

De tweede hallo-functie heeft de eerste overschreven, wat vergelijkbaar is met het toekennen van een nieuwe waarde aan een variabele. Bij variabelen gaat de oude waarde verloren en bij functies vervalt de oude definitie.

Door het bestaan van defaultwaarden kunnen we in Python toch hetzelfde effect als overloading bereiken. We passen het bovenstaande voorbeeld als volgt aan:

```
In:
def hallo(tekst='') :
    print ('hallo', tekst)

hallo()
hallo('Larissa')
```

```
Out:
```

```
hallo
hallo Larissa
```

Python kent wel overloading van operatoren. Zo wordt de '+'-operator bijvoorbeeld gebruikt voor het optellen van 2 getallen, maar ook voor het samenvoegen (concateneren) van 2 strings.

Naast gewone functies kent Python ook methoden (Engels: methods). Een *methode* is een functie die niet los wordt aangeroepen, maar in combinatie met een variabele van een bepaald datatype.

De syntax voor het *aanroepen* (Engels: callen) van een methode is als volgt:

```
variabelenaam.methodenaam(actuele parameters)
```

In de volgende hoofdstukken over samengestelde datatypen zullen we verschillende methoden behandelen waarmee bewerkingen op een variabele van dat datatype kunnen worden uitgevoerd. Je zou een methode als een gewone functie kunnen zien waarbij de eerste (extra) parameter een variabele van een bepaald datatype is. In het hoofdstuk over Objectoriëntatie zullen we leren hoe we zelf methoden bij nieuwe datatypen (klassen) kunnen definiëren en implementeren.

## 7.3 Scope van variabelen

Elke naam (identifier) heeft een scope die bepaalt waar je deze kan gebruiken in je programma. Voor dat deel van het programma zeggen we dat de naam 'in scope' is.

De naam van een *lokale variabele*, dat is een variabele die in een functie is gedefinieerd, heeft een *lokale scope* (Engels: local scope). De naam is alleen in scope tussen de definitie en het einde van de functiebody. Zodra de functie een retourwaarde heeft verstrekt aan de aanroeper is de naam 'buiten scope' (Engels: out of scope). Een lokale variabele kan dus alleen gebruikt worden binnen de functie waarin de variabele is gedefinieerd.

Namen die gedefinieerd zijn buiten een functie (of klasse) hebben *globale scope* (Engels: global scope). Dit betreft namen van functies, variabelen en klassen. Globale variabelen zijn variabelen met globale scope. Namen met een globale scope kunnen in een .py bestand of een interactieve sessie overal gebruikt worden nadat ze gedefinieerd zijn.

#### Voorbeelden

De variabele is gedefinieerd in een functie en wordt daarin ook gebruikt:

```
In:
def print_a() :
    a = 10
    print(a)

print_a()
```

```
Out:
10
```

Lokale variabelen zijn niet bekend buiten de functie waarin ze gedefinieerd zijn:

```
In:
def print_a() :
    a = 10
    print('1:', a)

print_a()
print('2:', a)
```

```
Out:
1: 10
```

```
-----
NameError                                Traceback (most recent call last)
Input In [], in <cell line: 6>()
      3     print('1:', a)
      5     print_a()
----> 6     print('2:', a)
```

```
NameError: name 'a' is not defined
```

Daarentegen zijn globale variabelen na definitie overal bekend (ook in functies):

```
In:
def print_a() :
    print('2:', a)

a = 5
print('1:', a)
print_a()
print('3:', a)
```

```
Out:
1: 5
2: 5
3: 5
```

Een globale variabele kan echter wel door een lokale variabele worden 'gecamoufleerd':

```
In:
def print_a() :
    a = 10
```

```

    print('2:', a)

a = 5
print('1:', a)
print_a()
print('3:', a)

```

Out:

```

1: 5
2: 10
3: 5

```

Als er een lokale variabele is wordt de globale variabele met dezelfde naam al vanaf het begin van de functie gecamoufleerd:

```

In:
def print_a() :
    print('2:', a)
    a = 10
    print('3:', a)

a = 5
print('1:', a)
print_a()
print('4:', a)

```

Out:

```

1: 5

```

```

-----
UnboundLocalError                                Traceback (most recent call last)
Input In [33], in <cell line: 8>()
      6 a = 5
      7 print('1:', a)
----> 8 print_a()
      9 print('4:', a)

Input In [33], in print_a()
      1 def print_a() :
----> 2     print('2:', a)
      3     a = 10
      4     print('3:', a)

```

UnboundLocalError: local variable 'a' referenced before assignment

Globale variabelen kunnen niet (zondermeer) in een functie worden gewijzigd:

```

In:
def print_a() :
    a *= 2
    print('2:', a)

a = 5
print('1:', a)
print_a()
print('3:', a)

```

Out:

```

1: 5

```

```

-----
UnboundLocalError                                Traceback (most recent call last)
Input In [], in <cell line: 7>()
      5 a = 5
      6 print('1:', a)
----> 7 print_a()
      8 print('3:', a)

Input In [69], in print_a()
      1 def print_a() :
----> 2     a *= 2

```

```
3     print('2:', a)
```

```
UnboundLocalError: local variable 'a' referenced before assignment
```

Wel kunnen we de waarde van een globale variabele binnen een functie te wijzigen moeten we vooraf het global-statement gebruiken:

## global *variabelenaam*

```
In:
def print_a() :
    global a
    a *= 2
    print('2:', a)
```

```
a = 5
print('1:', a)
print_a()
print('3:', a)
```

```
Out:
```

```
1: 5
2: 10
3: 10
```

Je kunt een variabele verwijderen met behulp van het del-statement:

## del *variabelenaam*

Je zult dit waarschijnlijk alleen gebruiken voor globale variabelen omdat alle lokale variabelen na beëindiging van de functie waarin ze gedefinieerd zijn automatisch worden verwijderd.

```
In:
a = 5
print('1:', a)
del a
print('2:', a)
```

```
Out:
```

```
1: 5
```

```
-----
NameError                                Traceback (most recent call last)
```

```
Input In [], in <cell line: 4>()
```

```
      2 print('1:', a)
      3 del a
----> 4 print('2:', a)
```

```
NameError: name 'a' is not defined
```

## 7.4 Python bibliotheek

In deze paragraaf gaan we in op de verzameling van alle functies die voorbij zijn gekomen, namelijk de Python bibliotheek.

In de vorige paragrafen hebben we geleerd hoe we zelf Python functies kunnen maken. We hebben echter ook al verschillende functies gebruikt zonder dat we die zelf hadden geprogrammeerd, zoals de print-functie, de input-functie, enzovoorts. Het grote voordeel om bestaande functies te gebruiken is natuurlijk dat je ze niet zelf hoeft te maken en dat ze over het algemeen zeer goed zijn getest! We noemen de verzameling van al deze functies de Python bibliotheek (Engels: Python library) en die bibliotheek breidt nog voortdurend uit.

Er zijn zelfs zoveel Python functies beschikbaar dat je ze nooit allemaal zult leren kennen, ook al doe je tijdens de rest van je leven niets anders dan programmeren in Python. Dat is niet erg, want het belangrijkste is dat je (gericht) leert zoeken in de Python bibliotheek of er een functie bestaat die jij nodig hebt.

Mocht zo'n functie bestaan dan is het voldoende als je weet hoe je deze moet gebruiken, maar het is niet nodig om de code in de functiebody te begrijpen (of zelfs maar te kennen). We zien de functie als een "zwarte doos" (Engels: black box): je kunt alleen de buitenkant bekijken, maar niet de binnenkant! In het dagelijks leven gebruiken we natuurlijk heel veel apparaten, waarvan we de interne werking niet kennen en dat levert (meestal) geen problemen op.

De buitenkant van een functie bestaat uit de signatuur (naam en parameters) en de retourwaarde. De aanroeper van een functie is verantwoordelijk voor het meegeven van de juiste parameterwaarden. Als de functie bijvoorbeeld een integer verwacht maar een string als argument krijgt zal het programma hoogstwaarschijnlijk abrupt eindigen met een run-time fout.

We kunnen de bibliotheek van functies in 3 categorieën verdelen:

1. Ingebouwde functies
2. Functies in de standaard bibliotheek
3. Functies in externe packages

## 7.5 Ingebouwde functies (built-in functions)

Een *ingebouwde functie* (Engels: built-in function) is een functie die altijd in Python beschikbaar is. [Hier vind je een overzicht van de functies.](#)

Built-in Functions (Python 3.9)				
abs()	<u>delattr()</u>	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	<u>issubclass()</u>	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Bij elke functie kun je een uitgebreide beschrijving van de syntax (parameters, defaultwaarden en een toelichting op het gebruik) vinden. Hieronder staat de beschrijving van de print-functie.

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end*, *file* and *flush*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is true, the stream is forcibly flushed.

Changed in version 3.3: Added the *flush* keyword argument.

## 7.6 Python standaard bibliotheek

De Python Standaard bibliotheek (Engels: Python Standard Library) bevat veel meer functies dan de ingebouwde functies. Al die functies zijn georganiseerd in modules en een overzicht van die modules vind je op <https://docs.python.org/3.9/library/>.

Om een functie uit zo'n module te gebruiken moet je eerst de module importeren:

```
import modulenaam
```

Een functie uit zo'n module kan daarna als volgt worden aangeroepen:

```
modulenaam.functienaam(parameters)
```

Een belangrijke module is random, waarin functies staan voor het genereren van willekeurige (random) getallen. Als voorbeeld gebruiken we de functie randrange() om een (gewone) dobbelsteen te simuleren, waarmee we 10 keer gooien:

In:

```
import random

for worp in range(10) :
    print(random.randrange(1, 7), end=' ')
```

Out:

```
5 2 3 6 1 6 4 2 6 5
```

Je kunt ook een specifieke functie importeren:

```
from modulenaam import functienaam
```

In dit geval hoeft je niet meer de modulenaam voor de functienaam op te nemen. Pas wel op voor conflicten met bestaande namen!

In:

```
from random import randrange

for worp in range(10) :
    print(randrange(1, 7), end=' ')
```

Out:

```
5 2 5 6 2 2 6 2 2 2
```

Tenslotte kun je ook nog de geïmporteerde functie een andere naam geven:

```
from modulenaam import functienaam as nieuwe_functienaam
```

In:

```
from random import randrange as rr

for worp in range(10) :
    print(rr(1, 7), end=' ')
```

Out:

```
2 1 3 6 1 3 5 4 1 4
```

## 7.7 Python packages

Naast de Standaard Python bibliotheek zijn er nog andere bibliotheken met functies, die worden aangeduid als *packages*. Een package bestaat uit 1 of meer gerelateerde modules met functies voor een bepaald aandachtsgebied. Er zijn inmiddels honderden packages en het aantal neemt nog steeds toe! Veel packages zijn al standaard door Anaconda geïnstalleerd. Welke dat zijn kun je in de Anaconda Navigator zien als je aan de linkerkant op de knop Environments drukt. In deze cursus zullen we de packages NumPy en Matplotlib gebruiken om respectievelijk arrays en plots (plaatjes) te kunnen maken.

## 7.8 Samenvatting

In dit hoofdstuk hebben we geleerd hoe we zelf Python functies kunnen maken en hoe gegevens tussen de functie en de aanroeper van de functie kan worden uitgewisseld. We hebben gezien hoe defaultwaarden en keyword parameters gebruikt kunnen worden om het aanroepen te vereenvoudigen, wat met name handig is als een functie heel veel parameters heeft.

We hebben de scope van namen behandeld en het verschil tussen lokale en globale variabelen. Programma's moeten zo weinig mogelijk globale variabelen hebben en bij voorkeur worden deze alleen gebruikt om constante waarden op te slaan.

Het adagium 'goed gejat is beter dan slecht gemaakt' geldt zeker voor het schrijven van code. De Standaard Python bibliotheek en alle aanvullende packages bevatten zoveel nuttige functies dat het altijd zinvol is om eerst te kijken of een bepaalde functie als bestaat, voordat je hem zelf gaat schrijven. Het spaart tijd en meestal zijn die functies al uitgebreid getest.



In de volgende hoofdstukken gaan we 4 samengestelde datatypen bespreken die ingebouwd zijn in Python: de lijst, de tuple, het woordenboek en de verzameling.

## 7.9 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studietoestel.

### Opgave 1

Schrijf een functie die de tafel van "factor" afdrukt (zie ook opgave 5 uit het hoofdstuk Besturingsstructuren). Bijvoorbeeld als factor = 5:

Out:

```
1 x 5 = 5
2 x 5 = 10
3 x 5 = 15    # enz.
```

De header van de functie is:

In:

```
def print_tafel(factor) :
```

Opmerking: Maak een testset om de functie te testen. Gebruik het hoofdprogramma om die test uit te voeren.

### Opgave 2

Schrijf een functie die bepaalt of 2 getallen een veelvoud van elkaar zijn. De functie geeft True of False terug. Ga ervan uit dat de twee getallen positief en geheel zijn.

De header van de functie is:

In:

```
def is_veelvoud(a, b) :
```

Schrijf een eenvoudig hoofdprogramma om je functie in te gebruiken.

### Opgave 3

Schrijf een functie die een "productregel" afdrukt.

Bijvoorbeeld:

Out:

```
2 x 5 = 10 is de productregel bij 2 en 5
6 x 8 = 48 is de productregel bij 6 en 8
```

De header van de functie is:

In:

```
def print_productregel(aantal, tafel) :
```

Schrijf een eenvoudig hoofdprogramma om je functie in te gebruiken.

aannames:  $\text{aantal} \geq 0$  en  $0 \leq \text{tafel} \leq 10$

Pas vervolgens de body van de functie `print_tafel()` uit opgave 1 aan. Maak in deze body gebruik van de functie `print_productregel()`.

Test vervolgens de nieuwe versie van `print_tafel` met hetzelfde testprogramma dat je bij opgave 1 gebruikt hebt. In het geval van een blackbox test kun je dus ook dezelfde testset gebruiken.

## Opgave 4 (wordt in de les behandeld)

Schrijf een functie die Celsius naar Fahrenheit converteert, met de volgende header:

In:

```
def cels_to_fahr (celsius) :
```

N.B. De formule voor de conversie is als volgt:  $fahrenheit = (9 / 5) * celsius + 32$

Schrijf een programma dat een for-statement gebruikt en hierbij de functie `cels_to_fahr()` aanroept om de volgende uitvoer te produceren:

Out:

Celsius	Fahrenheit
40.00	104.00
39.00	102.20
38.00	100.40
37.00	98.60
36.00	96.80
35.00	95.00
34.00	93.20
33.00	91.40
32.00	89.60
31.00	87.80

## 8. Lijsten (Lists)

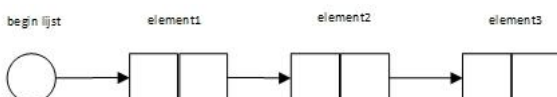
In dit hoofdstuk behandelen we het samengestelde datatype list.

Tot nu toe hebben we alleen gewerkt met variabelen, waarden en argumenten van elementaire datatypen. Dit soort datatypen, zoals `int`, `float`, `bool` en `str`, hebben waarden die niet zonder verlies van betekenis uit elkaar gehaald kunnen worden. Je kunt het getal 31 natuurlijk splitsen in de cijfers 3 en 1, maar daarbij gaat de waarde van het getal verloren. Zo kun je ook het woord 'hallo' (zonder quotes) splitsen in de letters 'h', 'a', 'l' en 'o'. maar dan gaat de betekenis van het woord verloren! In dit hoofdstuk en de volgende hoofdstukken bekijken we samengestelde datatypen, zoals lijsten (Engels: lists), tupels (Engels: tuples), woordenboeken (Engels: dictionaries), verzamelingen (Engels: sets) en arrays. Samengestelde datatypen worden vaak met de verzamelnaam *collecties* aangeduid.

Een variabele van een samengesteld datatype bevat meerdere (enkelvoudige) waarden die op een bepaalde manier gegroepeerd zijn. In dit hoofdstuk behandelen we het samengestelde datatype list. Dit type is ingebouwd in Python en hoeft dus niet geïmporteerd te worden.

### 8.1 Wat is een lijst?

Een (simpele) lijst is opgebouwd uit elementen die uit 2 delen bestaan: in het eerste deel zit de data van het element en het tweede deel bestaat uit een verwijzing naar het volgende element. De lijst heeft een begin die naar het eerste element verwijst en het laatste element van de lijst heeft geen verwijzing. De lijst in onderstaande afbeelding heeft dus 3 elementen.



Een *lijst* (Engels: list) in Python wordt aangegeven door middel van rechte haken om de gehele collectie waarden, met komma's tussen de verschillende elementen. Meestal zijn de waarden van hetzelfde datatype, zoals een lijst van integers of een lijst van strings. De waarden mogen echter ook van verschillende typen zijn.

## 8.2 Een lijst aanmaken

In deze paragraaf leer je hoe je een lijst kan aanmaken.

We maken als volgt een lijst met de 5 getallen 2, 5, 7, 11 en 15 (in die volgorde) aan:

```
In:
nummers = [2, 5, 7, 11, 15]
print(nummers)
print(type(nummers))
```

```
Out:
[2, 5, 7, 11, 15]
<class 'list'>
```

De *lege lijst* is een lijst zonder elementen en wordt aangegeven met `[]`.

```
In:
lege_lijt = []
print(type(lege_lijt), lege_lijt)
```

```
Out:
<class 'list'> []
```

Lijstelementen kunnen individueel benaderd worden met behulp van een index. Python begint te tellen vanaf index 0. Dit geldt niet alleen voor lijsten maar ook voor de meeste andere geordende collecties.

```
In:
print(nummers[0])
```

```
Out:
2
```

Je kunt ook een negatieve index gebruiken. In dat geval begin je bij het laatste element (dat heeft index -1) en wordt de index steeds met 1 verlaagd als je de lijst in omgekeerde volgorde doorloopt.

```
In:
print(nummers[-2])
```

```
Out:
11
```

Als je voor de index een niet-gehele waarde gebruikt (of een expressie die een niet-gehele waarde oplevert) krijg je een foutmelding (TypeError):

```
In:
print(nummers[1.5])
```

```
Out:
-----
TypeError                                Traceback (most recent call last)
Input In [], in <cell line: 1>()
----> 1 print(nummers[1.5])
```

TypeError: list indices must be integers or slices, not float

Als je een niet-bestaande index (een te grote of te kleine waarde) gebruikt krijg je eveneens een foutmelding (IndexError):

```
In:
print(nummers[10])
```

```
Out:
-----
IndexError                                Traceback (most recent call last)
Input In [], in <cell line: 1>()
----> 1 print(nummers[10])
```

IndexError: list index out of range

Om te controleren of een waarde al voorkomt in een lijst wordt in Python het keyword 'in' gebruikt. Door het keyword 'not' voor 'in' te zetten kun je controleren of een waarde niet in een lijst zit.

```
In:
nummers = [2, 5, 7, 11, 15]
print('7', 7 in nummers)
print('8', 8 in nummers)
print('9', 9 not in nummers)
```

```
Out:
7 True
8 False
9 True
```

Met de methode `index()` kun je het eerste voorkomen van een gegeven waarde in de lijst bepalen. In het volgende voorbeeld zoeken we de index van het eerste lijstelement met waarde 11:

```
In:
nummers = [2, 5, 7, 11, 15]
print(nummers.index(11))
```

```
Out:
3
```

Het aantal elementen in de lijst kan worden opgevraagd met behulp van de `len`-functie:

```
In:
nummers = [2, 5, 7, 11, 15]
print('aantal elementen :', len(nummers))
```

```
Out:
aantal elementen : 5
```

Je kunt de waarde van een element wijzigen met behulp van een gewone toekenning:

```
In:
nummers[0] = 3
print(nummers)
```

```
Out:
[3, 5, 7, 11, 15]
```

Hier is de waarde van het eerste element (dus het element met index 0) gewijzigd van 2 in 3. Je kunt lijstelementen ook gebruiken in expressies:

```
In:
print(nummers[0] + nummers[1])
```

```
Out:
8
```

Hier zijn de waarden van het eerste element (met index 0) en het tweede element (met index 1) bij elkaar opgeteld. Wat is volgens jou het resultaat van de volgende regel code? Test het!

```
In:
print(nummers[2] * nummers[3])
```

## Opgave

Bepaal en print de som van alle elementen in de lijst `nummers` met behulp van een `for`-statement.

[Controleer hier je antwoord](#)

Dit programma kan echter veel korter als je gebruikmaakt van de ingebouwde `sum`-functie:

[Zie hier hoe dat kan.](#)

## 8.3 Een lijst groter of kleiner maken

Soms wil je meer dan alleen de elementen uit de lijst aanpassen. Met de methode `append()` kun je elementen toevoegen aan het eind van de lijst.

```
In:
nummers = [2, 5, 7, 11, 15]
nummers.append(17)
print(nummers)
```

```
Out:
[2, 5, 7, 11, 15, 17]
```

Met de methode `extend()` kun je meerdere elementen toevoegen aan het eind van de lijst.

```
In:
nummers = [2, 5, 7, 11, 15]
nummers.extend([19, 21, 26])
print(nummers)
```

```
Out:
[2, 5, 7, 11, 15, 19, 21, 26]
```

Je kunt hetzelfde resultaat bereiken door de lijsten aan elkaar te plakken (concateneren) met behulp van de `'+'`-operator:

```
In:
nummers = [2, 5, 7, 11, 15]
extra_nummers = [19, 21, 26]
print(nummers + extra_nummers)
```

```
Out:
[2, 5, 7, 11, 15, 19, 21, 26]
```

Met de `'*'`-operator kun je vanuit een (kleine) lijst een grote lijst maken, die een herhaling is van de oorspronkelijke lijst:

```
In:
nummers = [2, 5, 7, 11, 15]
print(nummers * 2)
```

```
Out:
[2, 5, 7, 11, 15, 2, 5, 7, 11, 15]
```

Met de methode `insert()` kun je elementen toevoegen op een willekeurige plek in de lijst. Bij aanroep van de methode geef je de gewenste index en de waarde van het element als argumenten mee. Het element dat eerst die index had en alle elementen daarna schuiven 1 plaats naar rechts. Het toevoegen van element 13 op index 4 (begin met tellen vanaf 0!) gaat dus als volgt:

```
In:
nummers.insert(4, 13)
print(nummers)
```

```
Out:
[2, 5, 7, 11, 13, 15, 17]
```

Als we de lijst kleiner willen maken kunnen we de methode `remove()` gebruiken:

```
In:
nummers.remove(11)
print(nummers)
```

```
Out:
[2, 5, 7, 13, 15, 17]
```

`remove()` verwijdert alleen de eerste instantie van het element dat hij tegenkomt. Bekijk het volgende voorbeeld:

```
In:
nummers.insert(4,7)
```

```
print(numbers)
numbers.remove(7)
print(numbers)
```

Out:

```
[2, 5, 7, 13, 7, 15, 17]
[2, 5, 13, 7, 15, 17]
```

Eerst voegen we een tweede 7 toe op index 4. Vervolgens verwijdert `remove()` de eerste 7 die hij tegenkomt (op index 2). Als je alle 7's in één keer wil verwijderen kun je dit doen door middel van een for-loop. Later in dit hoofdstuk zullen wij echter efficiëntere methodes behandelen.

Als je een element met een bepaalde index wil verwijderen kun je dit net als bij variabelen doen met behulp van het `del`-statement.

```
In:
del numbers[1]
print(numbers)
```

Out:

```
[2, 13, 7, 15, 17]
```

We hebben het element met index 1 (en waarde 5) verwijderd. Let goed op de verschillen tussen een methode (`numbers.remove(7)`), een 'gewone' functie (`print(numbers)`) en een statement (`del numbers[1]`).

## Opgave

Maak de lijst `tafel_van_drie = [3, 6, 9, 12, 16, 18, 24, 27, 32]` aan. Je ziet al dat er een aantal waardes niet kloppen.

1. Gebruik een toekenning om 16 in 15 te veranderen
2. Gebruik de methode `remove()` om 32 te verwijderen
3. Gebruik de methode `append()` om 30 toe te voegen aan het eind van de lijst
4. Gebruik de methode `insert()` om 21 toe te voegen tussen 18 en 24

Print de lijst. [Klopt de tafel nu?](#)

Zoals je ziet kan er veel mis gaan als je zelf een lange rij getallen met een bepaalde logische volgorde (zoals een tafel) moet typen. Je hebt in een eerder hoofdstuk al de `range`-functie geleerd. Als je makkelijk een lijst wil aanmaken van een `range` kan dat door middel van de `list()` functie:

```
In:
print(list(range(3,31,3)))
```

Out:

```
[3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
```

Simpelweg rechte haken om de `range()` heen zetten maakt nog niet een lijst aan met alle waardes. Dit doet Python om geheugen te besparen:

```
In:
print([range(3,31,3)])
```

Out:

```
[range(3, 31, 3)]
```

Lijsten kunnen in omgekeerde volgorde gezet worden met behulp van de methode `reverse()` en ze kunnen gesorteerd worden met behulp van de methode `sort()`.

Lijstelementen kunnen zelf ook lijsten zijn. We spreken dan over *geneste lijsten*.

## 8.4 Slicing

Vaak willen we meerdere elementen uit een lijst halen. Dit kan door middel van 'slicing'.

Bij slicing typen we de naam van de lijst, met daaraan vast rechte haken die de start en stop indices bevatten van de gewenste subset, gescheiden door een dubbele punt:

```
In:
print(nummers[1:4])
```

```
Out:
[13, 7, 15]
```

Hier hebben we de elementen van de lijst `nummers` op index 1 tot, (maar niet met!) 4 uit de oorspronkelijke lijst gehaald. Net als bij de `range`-functie die je al eerder hebt gezien rekent Python wel de startindex mee, maar niet de eindindex. Belangrijk om te weten is ook dat we de lijst zelf niet hebben aangepast:

```
In:
print(nummers)
```

```
Out:
[2, 13, 7, 15, 17]
```

We kunnen ook slicen zonder de begin- of eindindex mee te geven. Als we de beginindex weg laten neemt Python aan dat we vanaf het begin van de lijst (index 0) willen slicen. Als we de eindindex weg laten neemt Python aan dat we tot EN MET het eind willen slicen (equivalent aan de lengte van de lijst als eindindex meegeven):

```
In:
print(nummers[:2])
print(nummers[2:])
```

```
Out:
[2, 13]
[7, 15, 17]
```

We kunnen ook met een andere stapgrootte of achteruit slicen:

```
In:
print(nummers[:2])
print(nummers[::-1])
```

```
Out:
[2, 7, 17]
[17, 15, 7, 13, 2]
```

Slicing kan niet alleen waarden ophalen, het kan ook gebruikt worden om waarden te wijzigen:

```
In:
nummers[0:2] = [2,3,5]
print(nummers)
```

```
Out:
[2, 3, 5, 7, 15, 17]
```

Merk op dat de lijst langer is geworden! We hebben namelijk alleen de eerste twee elementen (op index 0 en 1) vervangen door een lijst met 3 elementen.

## Opgave

Neem de lijst `namen = ['Alfred', 'Bob', 'Charlie', 'David', 'Erik']` over.

1. Print de eerste 3 namen
2. Vervang David en Erik door ['Daphne', 'Eva', 'Frederique'] en print de uitkomst

[Zie hier de uitwerking.](#)

## 8.5 List comprehensions

Een 'list comprehension' (er is geen goed Nederlands woord voor) kan op een korte manier hetzelfde bewerkstelligen als meerdere regels code

Neem het volgende voorbeeld van list comprehension:

```
In:
lijst1 = []
for getal in range(1,6):
    lijst1.append(getal)
print(lijst1)
```

```
Out:
[1, 2, 3, 4, 5]
```

Dat zijn 3 regels code voor de simpele taak van het aanmaken van een lijst met de getallen 1, 2, 3, 4 en 5. Met een list comprehension kunnen we het aanmaken in 1 regel code doen:

```
In:
lijst2 = [getal for getal in range(1,6)]
print(lijst2)
```

```
Out:
[1, 2, 3, 4, 5]
```

Nu kun je zeggen 'leuk en aardig, maar met `list(range(1, 6))` kan ik dit ook in 1 regel'. Dat is juist, maar met list comprehension kun je een stuk meer. Zie bijvoorbeeld het volgende voorbeeld:

```
In:
lijst3 = [getal ** 3 for getal in range(1,6)]
print(lijst3)
```

```
Out:
[1, 8, 27, 64, 125]
```

Dit is een voorbeeld van *mapping*: het uiteindelijke resultaat bevat evenveel elementen als de originele data, maar in een andere vorm. In dit voorbeeld is elk getal gemapt naar de derde macht (dus 2 naar  $2^3 = 8$ , enzovoort). Een andere toepassing van list comprehension is *filtering*: hierbij blijven er in het resultaat (meestal) minder waarden over dan in de oorspronkelijke data:

```
In:
lijst4 = [getal for getal in range(1,21) if getal not in (3,5,7)]
print(lijst4)
```

```
Out:
[1, 2, 4, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

Hier hebben we de range van getallen 1 tot en met 20 teruggekregen, zonder de getallen 3,5 en 7. Je kunt dus hele blokken code met for- en if-statements verwerken in een enkele regel met behulp van een list comprehension. Als je vrij grote blokken hebt is het wellicht beter om dat zo te houden, zodat je code leesbaar blijft. In het bovenstaande voorbeeld staan 3, 5 en 7 binnen ronde haken, dus in een tuple. Tupels worden in het volgende hoofdstuk behandeld.

## Opgave

Gebruik list comprehension om een lijst te maken van alle even getallen tussen 1 en 20.

[Controleer je lijst.](#)

## 8.6 Samenvatting

In dit hoofdstuk hebben we het belangrijke samengestelde datatype lijst behandeld. Je gebruikt lijsten om samenhangende gegevens op te slaan en te bewerken. Lijsten zijn heel flexibel omdat ze geordend zijn en je overal data kunt toevoegen, wijzigen en verwijderen. De individuele lijstelementen zijn te benaderen met behulp van een index. In het volgende hoofdstuk bekijken we een ander samengesteld datatype met een ordening: de tuple.

## 8.7 Opgaven



Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studietoetscoach.

## Opgave 1

Maak de lijst 'getallen' aan: `getallen = [2, 4, 7, 11, 19]`

Voer de volgende opdrachten uit:

1. Voeg het getal 22 toe aan (het einde van) de lijst
2. Voeg het getal 6 toe tussen 4 en 7
3. Vervang het getal 4 door het getal 5

## Opgave 2

In de Fibonacci rij bestaat elk getal uit de som van de twee voorgaande getallen: 1, 1, 2, 3, 5... De som van 1 en 1 is 2, de som van 1 en 2 is 3, enzovoorts. Implementeer de functie 'fibonacci' die als parameter de lijst 'fibonacci\_reeks' = [1, 1] krijgt aangeleverd, en een element toevoegt aan de lijst, bestaande uit de vorige twee elementen. Roep de functie meerdere keren aan (Bijvoorbeeld met een for-loop).

## Opgave 3 (wordt besproken in de les)

Maak een lijst 'kwadraten' die de kwadraten bevat van de getallen 1 tot en met 10. Gebruik een for loop.

# 9. Tupels (Tuples)

## 9.1 Tuples

Een tuple is een onveranderlijke verzameling van elementen, en wordt aangegeven door middel van ronde haken om de gehele verzameling, met komma's tussen de verschillende elementen.

```
In:
lege_tuple = ()
print(lege_tuple)
print(type(lege_tuple))
```

```
Out:
()
<class 'tuple'>
```

De haakjes zijn optioneel zolang de elementen van elkaar gescheiden worden door middel van komma's:

```
In:
student_tuple = 'Jan', 'Groen', 3.3
print(student_tuple)
print(type(student_tuple))
```

```
Out:
('Jan', 'Groen', 3.3)
<class 'tuple'>
```

De komma's zijn niet optioneel, probeer maar eens te raden wat de types zijn van beide variabelen. Test het daarna.

```
In:
variabele1 = ('rood',)
variabele2 = ('geel')
print(type(variabele1))
print(type(variabele2))
```

Tupel elementen kunnen net als lijstelementen bereikt worden met een index. Ook hier heeft het eerste element van de tuple index 0. Tevens kun je weer 'achterwaarts' indexeren met behulp van negatieve indices:

```
In:
tuple1 = ('hello', 'world', 'Monty', 'Python')
print(tuple1[0])
print(tuple1[1])
print(tuple1[-1])
print(tuple1[-2])
```

```
Out:
hello
world
Python
Monty
```

Een tuple is niet-wijzigbaar. De elementen van een tuple kunnen niet zomaar veranderd worden zoals je gewend bent van lijsten:

```
In:
tuple1[0] = 'spam'
```

```
Out:
```

```
-----
TypeError                                Traceback (most recent call last)
Input In [], in <cell line: 2>()
      1 tuple1 = ('hello', 'world', 'Monty', 'Python')
----> 2 tuple1[0] = 'spam'
```

```
TypeError: 'tuple' object does not support item assignment
```

## 9.2 Samenvatting

In dit hoofdstuk hebben we het samengestelde datatype tuple behandeld dat net als een lijst is geordend. Een tuple bestaat meestal uit heterogene data die met elkaar samenhangt, zoals de naam, het studentnummer, het adres en de behaalde resultaten van een student.

## 9.3 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studiecoach.

### Opgave 1

Maak de lijst 'getal\_kwadraat\_paar' aan voor getallen 1 tot en met 5 waarin elk element bestaat uit een tuple die het getal en het bijbehorende kwadraat bevat. Gebruik een list comprehension.

### Opgave 2

Schrijf een dobbelspel voor een dobbelsteen met waardes 1 t/m 6. Als de speler 5 of 6 gooit, heeft hij gewonnen. Als hij 1 of 2 gooit, heeft hij verloren. Als hij 3 of 4 gooit, mag hij nog een keer gooien. Gebruik de random.randrange functie uit de module random voor de dobbelsteenwaarde. Hou de game status bij in een variabele die gecheckt wordt door een while loop. Gebruik if/elif waarde in tuple om te checken of een waarde gegooit is. Print de uitkomst.

## 10. Woordenboeken (Dictionaries)

In de vorige hoofdstukken hebben we twee samengestelde datatypen behandeld met een ordening, namelijk lijsten (lists) en tupels (tuples). In dit hoofdstuk maken we kennis met een samengesteld datatype waarin de elementen niet geordend zijn: het woordenboek (Engels: dictionary).

## 10.1 Wat is een woordenboek?

Een woordenboek bestaat uit een aantal sleutel-waarde paren (Engels: key-value pairs), waarbij een onveranderlijke sleutel een waarde oplevert. Vergelijk dit met een 'echt' woordenboek waarin bij elk woord een omschrijving (definitie) wordt gegeven.

Een sleutel-waarde paar heeft de volgende syntax:

*sleutel* : *waarde* (de spatie voor en na de dubbelepunt mogen worden weggelaten)

De sleutel moet een niet te wijzigen datatype zijn, zoals een int, float of een str. Een woordenboek wordt in Python genoteerd met behulp van 2 accolades '{' en '}' waartussen de sleutel-waarde paren staan, gescheiden door komma's.

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5}
print(type(vakken), vakken)
```

Out:

```
<class 'dict'> {'Python': 10, 'Java': 8, 'C#': 5}
```

Het lege woordenboek wordt genoteerd als {}.

Twee woordenboeken zijn gelijk als ze precies dezelfde sleutel-waarde paren bevatten, waarbij die paren niet in dezelfde volgorde hoeven te staan. We kunnen dit testen met behulp van '=='.

In:

```
vakken1 = {'Python':10, 'Java':8, 'C#':5}
vakken2 = {'Java':8, 'C#':5, 'Python':10}
print(vakken1 == vakken2)
```

Out:

```
True
```

Omdat woordenboeken niet geordend zijn is de volgorde van de sleutel-waarde paren onbepaald als je de print-functie aanroept met een woordenboek als argument. Als je een sleutel-waarde paar toevoegt met een bestaande sleutel wordt één van de paren met dezelfde sleutel uit het woordenboek verwijderd. Welke van de twee dat is weet je niet vooraf!

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'Java':7}
print(vakken)
```

Out:

```
{'Python': 10, 'Java': 7, 'C#': 5}
```

Om te controleren of een sleutel al voorkomt in een woordenboek wordt in Python het keyword 'in' gebruikt. Door het keyword 'not' voor 'in' te zetten kun je controleren of een sleutel niet in een woordenboek zit.

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5}
print('Python', 'Python' in vakken)
print('SQL', 'SQL' in vakken)
print('Pascal', 'Pascal' not in vakken)
```

Out:

```
Python True
SQL False
Pascal True
```

Het aantal sleutel-waarde paren kan worden opgevraagd met behulp van de len-functie.

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
print('aantal paren :', len(vakken))
```

Out:

```
aantal paren : 4
```

Je kunt de sleutel-waarde paren in een woordenboek niet benaderen met behulp van een index, zoals bij lijsten en tupels. Als je dat probeert krijg je een foutmelding (een `KeyError`).

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
print(vakken[1])
```

Out:

```
-----
KeyError                                Traceback (most recent call last)
Input In [], in <cell line: 2>()
      1 vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
----> 2 print(vakken[1])
```

```
KeyError: 1
```

Wel kun je een sleutel-waarde paar benaderen met behulp van de sleutel:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
print(vakken['Java'])
```

Out:

```
8
```

Als je een ongeldige sleutelwaarde gebruikt krijg je weer een `KeyError`:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
print(vakken['Pascal'])
```

Out:

```
-----
KeyError                                Traceback (most recent call last)
Input In [], in <cell line: 2>()
      1 vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
----> 2 print(vakken['Pascal'])
```

```
KeyError: 'Pascal'
```

Je kunt deze foutmelding voorkomen door gebruik te maken van de methode `get()`. Als de sleutel niet gevonden wordt geeft `get` 'None' terug. Als je een tweede argument bij de aanroep van `get()` meegeeft wordt die waarde getoond als de sleutel niet gevonden wordt:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
print(vakken.get('Pascal', 'Pascal staat niet in het woordenboek'))
```

Out:

```
Pascal staat niet in het woordenboek
```

De waarde van een sleutel-waarde paar kun je met een gewone toekenning wijzigen:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
vakken['Java'] = 7
print(vakken)
```

Out:

```
{'Python': 10, 'Java': 7, 'C#': 5, 'SQL': 6}
```

Als je een waarde toekent aan een niet-bestaande sleutel wordt er een nieuw sleutel-waarde paar aan het woordenboek toegevoegd:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
vakken['Pascal'] = 4
print(vakken)
```

Out:

```
{'Python': 10, 'Java': 8, 'C#': 5, 'SQL': 6, 'Pascal': 4}
```

Om een sleutel-waarde paar uit het woordenboek te verwijderen gebruik je het del-statement:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
del vakken['C#']
print(vakken)
```

Out:

```
{'Python': 10, 'Java': 8, 'SQL': 6}
```

Naast bovengenoemde manieren om een sleutel-waarde paar toe te voegen of te wijzigen bestaat er nog een methode update() die ook hiervoor gebruikt kan worden. In de volgende 2 voorbeelden zie je hoe je met behulp van update() een nieuw sleutel-waarde paar maakt of de waarde bij een bestaande sleutel wijzigt:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
vakken.update(Pascal = 4)
print(vakken)
```

Out:

```
{'Python': 10, 'Java': 8, 'C#': 5, 'SQL': 6, 'Pascal': 4}
```

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
vakken.update(Java = 7)
print(vakken)
```

Out:

```
{'Python': 10, 'Java': 7, 'C#': 5, 'SQL': 6}
```

Je kunt het hele woordenboek doorlopen met behulp van het for-statement. Als we alle vakken willen afdrukken waarvan de waarde minstens 6 is kan dat met de volgende code:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6, 'Pascal':4}
```

```
for vak,aantal in vakken.items() :
    if aantal >= 6 :
        print(vak, '->', aantal)
```

Out:

```
Python -> 10
Java -> 8
SQL -> 6
```

De methode items() geeft alle sleutel-waarde paren terug als een rij van tupels, waarbij we elk tupel uitpakken in de variabelen vak (voor de sleutel) en aantal (voor de waarde).

Woordenboeken hebben ook de methoden keys() en values() die alle sleutels respectievelijk alle waarden teruggeven als een rij. De volgende code drukt de namen (de sleutels) van alle vakken uit het woordenboek alfabetisch gesorteerd af, gescheiden door een spatie:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6, 'Pascal':4}
```

```
for vak in sorted(vakken.keys()) :
    print(vak, end=' ')
```

Out:

C# Java Pascal Python SQL

De methoden `items()`, `keys()` en `values()` geven elk een *view* van de data in het woordenboek. Als je een *view* doorloopt zie je de huidige inhoud van het woordenboek, niet een kopie van de data!

We tonen dit aan met het volgende voorbeeld. De variabele `vakken_view` krijgt alle sleutels in het woordenboek als waarde. Nadat er nieuw sleutel-waarde paar is toegevoegd blijkt de sleutel aan `vakken_view` te zijn toegevoegd.

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
vakken_view = vakken.keys()
print(vakken_view)
```

```
vakken['Pascal'] = 4
print(vakken_view)
```

Out:

```
dict_keys(['Python', 'Java', 'C#', 'SQL'])
dict_keys(['Python', 'Java', 'C#', 'SQL', 'Pascal'])
```

Je kunt alle items, sleutels of waarden van een woordenboek ook converteren naar een lijst. Dan gebruik je de ingebouwde `list`-functie:

In:

```
vakken = {'Python':10, 'Java':8, 'C#':5, 'SQL':6}
vakkenlijst = list(vakken.keys())
print(type(vakkenlijst), vakkenlijst)
```

Out:

```
<class 'list'> ['Python', 'Java', 'C#', 'SQL']
```

Ook bij woordenboeken kunnen we *comprehension* gebruiken om snel een nieuw woordenboek uit een bestaande collectie te maken. In het volgende voorbeeld starten we met een woordenboek met de namen en cijferlijsten van studenten en creëren daaruit een nieuw woordenboek met de namen en gemiddelde cijfers van de studenten.

In:

```
cijfers = {'Jan': [7.3, 8.5, 6.4], 'Piet': [6.9, 7.2, 9.1, 8.1]}
gemiddelde_cijfers = {s : sum(w) / len(w) for s, w in cijfers.items()}
print(gemiddelde_cijfers)
```

Out:

```
{'Jan': 7.400000000000001, 'Piet': 7.825000000000001}
```

## 10.2 Zoeksnelheid in woordenboeken

Soms kun je zowel een lijst als een woordenboek gebruiken voor een stukje code in een Python script. Voor de functionaliteit maakt het meestal niet uit welk datatype je kiest, maar er zijn situaties waarin er wel een groot verschil in snelheid is.

In het onderstaande voorbeeld worden een lijst en een woordenboek gemaakt met alle tientallen tussen 0 en 1 miljoen. Vervolgens wordt een willekeurig geheel getal tussen 0 en 1 miljoen gegenereerd en wordt zowel in de lijst als het woordenboek gezocht of dat getal erin voorkomt. Voor beide zoekacties meten we de benodigde tijd, waarbij we de methode `default_timer()` uit module `timeit` gebruiken. Zoeken in het woordenboek blijkt veel sneller te gaan dan in de lijst. Dat komt omdat de lijst sequentieel wordt doorlopen, terwijl het zoeken in het woordenboek met 'hashing' technieken gaat.

In:

```
import random
import timeit

getallist = list(getal*10 for getal in range(100_000))
getaldict = {getal*10 : 0 for getal in range(100_000)}

zoekgetal = random.randrange(1_000_000)

starttime = timeit.default_timer()
if zoekgetal in getallist :
    print('gevonden in lijst')
else :
```

```

    print('niet gevonden in lijst')
print("zoektijd in de lijst :", timeit.default_timer() - starttime)

starttime = timeit.default_timer()
if zoekgetal in getaldict :
    print('gevonden in woordenboek')
else :
    print('niet gevonden in woordenboek')
print("zoektijd in het woordenboek :", timeit.default_timer() - starttime)

```

Out:

```

niet gevonden in lijst
zoektijd in de lijst : 0.00493579999965732
niet gevonden in woordenboek
zoektijd in het woordenboek : 0.0003118999993603211

```

## 10.3 Visualisatie met Matplotlib

De uitspraak “Een plaatje zegt meer dan 1000 woorden” geldt ook als je veel data hebt en daaruit nuttige informatie probeert te halen, wat de kern is van het vakgebied Data Science.

In deze paragraaf leer je hoe je een eenvoudig staafdiagram (Engels: bar chart) kunt maken met behulp van module pyplot in het Python package Matplotlib. Dit package is onderdeel van Anaconda3 en hoeft dus niet meer apart geïnstalleerd te worden.

In het onderstaande voorbeeld wordt 6000 keer gegooid met een gewone dobbelsteen en wordt het aantal geworpen enen, tweeën, ..., zessen opgeslagen in een woordenboek. De sleutel van een element uit dit woordenboek is dus een van de gehele getallen tussen 1 en 6 en de waarde het aantal keren dat het betreffende getal is geworpen. Tenslotte wordt een staafdiagram van de geworpen waarden gemaakt.

In:

```

import matplotlib.pyplot as plt
import random

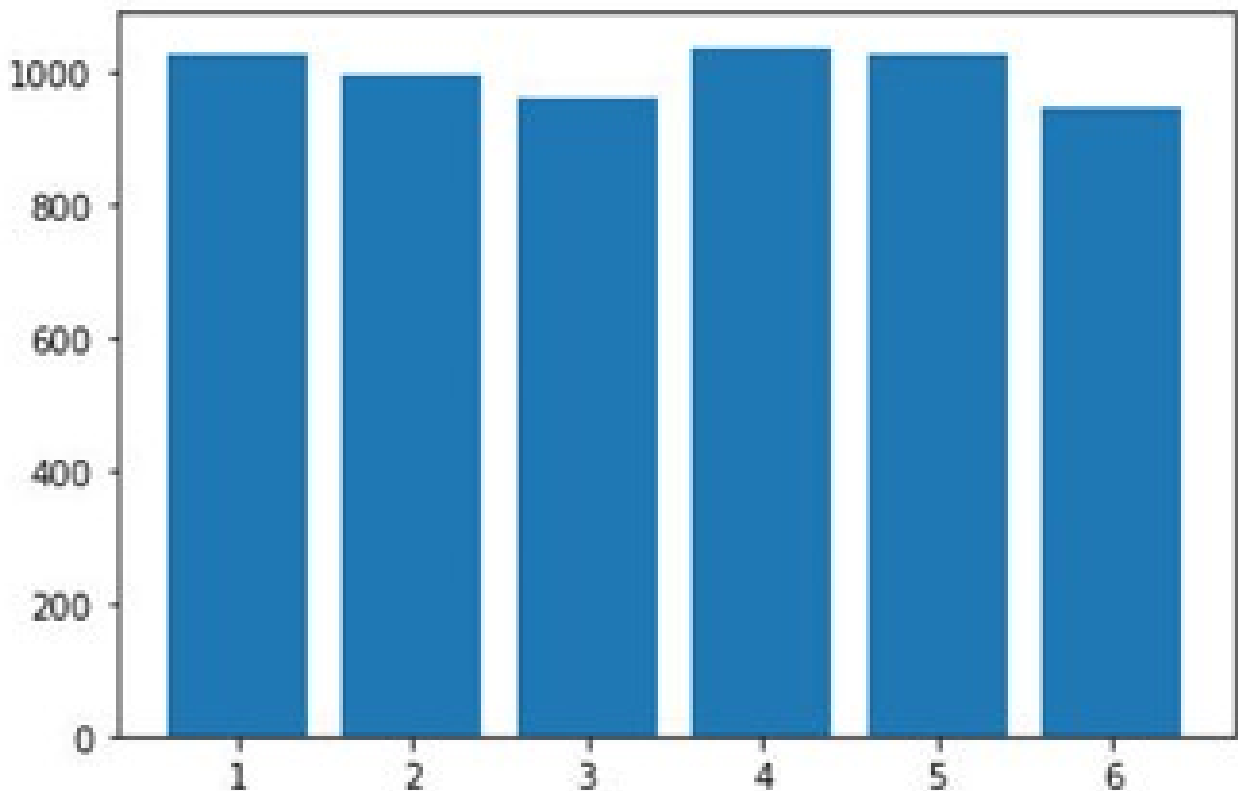
worpen = {i:0 for i in range(1,7)}

for i in range(6000):
    worp = random.randrange(1,7)
    worpen[worp] += 1

plt.bar(worpen.keys(), worpen.values())
plt.show()

```

Out:



## 10.4 Samenvatting

In dit hoofdstuk hebben we het samengestelde datatype woordenboek behandeld. Woordenboeken bestaan uit een ongeordende collectie sleutel-waarde paren. Met behulp van de sleutel kan een waarde worden opgezocht of gewijzigd. Het zoeken van een sleutel in een woordenboek gaat meestal veel sneller dan het zoeken van een waarde in een ongeordende lijst. In het volgende hoofdstuk bespreken we een ander ongeordend samengesteld datatype: de verzameling.

## 10.5 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studiecoach.

### Opgave 1

Gegeven een woordenboek van Internet toplevel domeinen:

In:

```
tlds = {'Nederland': 'nl', 'Verenigde Staten': 'us', 'Duitsland': 'de'}
```

Voer de volgende opdrachten uit en toon de resultaten:

1. Controleer of het woordenboek de sleutel 'Duitsland' bevat.
2. Controleer of het woordenboek de sleutel 'Frankrijk' bevat.
3. Itereer door de sleutel-waarde paren en toon ze in 2-kolommen.
4. Voeg het sleutel-waarde paar 'Zweden': 'sw' toe.
5. Wijzig de waarde van de sleutel 'Zweden' in 'se'.
6. Gebruik een dictionary comprehension om sleutels en waarden te verwisselen.
7. Uitgaande van het resultaat van f) gebruik een dictionary comprehension om alle landnamen te converteren naar hoofdletters.

### Opgave 2 (wordt in de les behandeld)



Maak een woordenboek van vrienden met hun voornaam als sleutel en hun leeftijd als waarde (we nemen even aan dat alle voornamen verschillend zijn).

Voer de volgende opdrachten uit en toon de resultaten:

1. Druk naam en leeftijd af van al je vrienden, alfabetisch gesorteerd op naam.
2. Druk de naam af van al je vrienden die 30 jaar of ouder zijn.
3. Verhoog alle leeftijden in het woordenboek met 1.
4. Verwijder alle vrienden waarvan de naam begint met de letter 'B'

## Opgave 3

Breid het programma waarin we de zoeksnelheid in een lijst en woordenboek vergelijken uit zodat je 1000 random gehele getallen tussen 0 en 1 miljoen genereert en die probeert op te zoeken in de lijst en het woordenboek. Wat is het resultaat?

# 11. Verzamelingen (Sets)

Hoewel verzamelingen (Engels: sets) de basis vormen voor de wiskunde (en dus ook de informatica) worden ze maar door weinig programmeertalen in zuivere vorm ondersteund. Bedenk echter wel dat alle samengestelde datatypen die we in eerdere hoofdstukken hebben behandeld beschouwd kunnen worden als (combinaties van) verzamelingen met enkele extra eigenschappen (zoals een ordening).

## 11.1 Wat is een verzameling?

Een verzameling is een ongeordende collectie elementen, waarin elk element slechts 1 keer voorkomt. Een verzameling wordt in Python (net als in de wiskunde) genoteerd met 2 accolades '{' en '}' waartussen de elementen staan, gescheiden door komma's. Hoewel Python sets aanpasbaar zijn kunnen de afzonderlijke elementen niet gewijzigd worden.

```
In:
kleuren = {'rood', 'blauw', 'geel'}
print(type(kleuren), kleuren)
```

```
Out:
<class 'set'> {'blauw', 'rood', 'geel'}
```

De *lege verzameling* (Engels: empty set) is een verzameling zonder elementen, en wordt in de wiskunde aangeduid met het symbool  $\emptyset$ . Omdat {} al gebruikt wordt om een leeg woordenboek aan te duiden wordt de lege verzameling in Python genoteerd als set().

Je kunt een verzameling creëren uit een andere collectie met behulp van de ingebouwde functie set(). In het volgende voorbeeld maken we een verzameling met de getallen 1 tot en met 5 uit de rij 1, 2, 3, 4, 5 die door de range-functie wordt gegenereerd:

```
In:
getallen = set(range(1,6))
print(type(getallen), getallen)
```

```
Out:
<class 'set'> {1, 2, 3, 4, 5}
```

In plaats van range(1, 6) hadden we in het vorige voorbeeld ook de lijst [1, 2, 3, 4, 5] als argument van de set-functie mogen gebruiken.

Twee verzamelingen zijn gelijk als ze dezelfde elementen bevatten. In Python kunnen we testen of verzamelingen gelijk zijn met behulp van '=='.

```
In:
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'blauw', 'geel', 'rood'}

print('gelijk' if kleuren1 == kleuren2 else 'niet gelijk')
```

```
Out:
Gelijk
```

Omdat elk element per definitie slechts 1 keer in een verzameling kan voorkomen worden dubbele elementen automatisch verwijderd:

```
In:
kleuren = {'rood', 'blauw', 'geel', 'blauw'}
print(kleuren)
```

```
Out:
{'blauw', 'rood', 'geel'}
```

Let op dat de kleuren van de verzameling in een andere volgorde worden afgedrukt dan waarin ze stonden bij de definitie van de verzameling!

Om te controleren of een element in een verzameling zit (wiskundig aangegeven met  $\in$ ) wordt in Python het keyword 'in' gebruikt. Door het keyword 'not' voor 'in' te zetten kun je controleren of een element niet in een verzameling zit.

```
In:
kleuren = {'rood', 'blauw', 'geel'}
print('geel', 'geel' in kleuren)
print('wit', 'wit' in kleuren)
print('oranje', 'oranje' not in kleuren)
```

```
Out:
geel True
wit False
oranje True
```

Het aantal elementen in een verzameling kan worden opgevraagd met behulp van de len-functie.

```
In:
kleuren = {'rood', 'blauw', 'geel'}
print('aantal elementen :', len(kleuren))
```

```
Out:
aantal elementen : 3
```

Om een element toe te voegen aan een verzameling gebruiken we methode add():

```
In:
kleuren = {'rood', 'blauw', 'geel'}
kleuren.add('wit')
print(kleuren)
```

```
Out:
{'rood', 'geel', 'wit', 'blauw'}
```

Met behulp van de methode remove() kun je een bestaand element verwijderen:

```
In:
kleuren = {'rood', 'blauw', 'geel'}
kleuren.remove('blauw')
print(kleuren)
```

```
Out:
{'rood', 'geel'}
```

Als je een element probeert te verwijderen dat niet in de verzameling voorkomt krijg je een foutmelding (een KeyError):

```
In:
kleuren = {'rood', 'blauw', 'geel'}
kleuren.remove('oranje')
print(kleuren)
```

```
Out:
-----
KeyError                                Traceback (most recent call last)
Input In [], in <cell line: 2>()
      1 kleuren = {'rood', 'blauw', 'geel'}
----> 2 kleuren.remove('oranje')
```

```
3 print(kleuren)
```

```
KeyError: 'oranje'
```

Met behulp van de methode `clear()` kun je een verzameling leeg maken (dus alle elementen uit de verzameling verwijderen):

```
In:
kleuren = {'rood', 'blauw', 'geel'}
kleuren.clear()
print(kleuren)
```

```
Out:
set()
```

Je kunt de elementen in een verzameling niet benaderen met behulp van een index, zoals bij lijsten en tupels. Als je dat probeert krijg je een foutmelding (een `TypeError`):

```
In:
kleuren = {'rood', 'blauw', 'geel'}
print(kleuren[0])
```

```
Out:
```

```
-----
TypeError                                 Traceback (most recent call last)
Input In [], in <cell line: 2>()
      1 kleuren = {'rood', 'blauw', 'geel'}
----> 2 print(kleuren[0])
```

```
TypeError: 'set' object is not subscriptable
```

Wel kun je de hele verzameling doorlopen met behulp van het `for`-statement. Als we alle kleuren in de verzameling in hoofdletters willen afdrukken, gescheiden door puntkomma's (en een spatie), kan dat met de volgende code:

```
In:
kleuren = {'rood', 'blauw', 'geel'}

for kleur in kleuren :
    print(kleur.upper(), end = '; ')
```

```
Out:
ROOD; GEEL; BLAUW;
```

Omdat verzamelingen aanpasbaar zijn kun je een verzameling niet als element in een (andere) verzameling opnemen. Daarop bestaat één uitzondering: de `frozenset`.

Een `frozenset` is een verzameling die na creatie niet kan worden aangepast. Je maakt een nieuwe `frozenset` met behulp van de ingebouwde functie `frozenset()`.

```
In:
kleuren = frozenset({'rood', 'blauw', 'geel'})
print(kleuren)
```

```
Out:
frozenset({'rood', 'geel', 'blauw'})
```

Met `frozensets` kun je dezelfde bewerkingen doen als met gewone verzamelingen, behalve als ze de verzameling willen aanpassen. In dat geval krijg je een foutmelding:

```
In:
kleuren = frozenset({'rood', 'blauw', 'geel'})
kleuren.add('wit')
print(kleuren)
```

```
Out:
```

```
-----
AttributeError                             Traceback (most recent call last)
Input In [], in <cell line: 2>()
      1 kleuren = frozenset({'rood', 'blauw', 'geel'})
----> 2 kleuren.add('wit')
```

```
3 print(kleuren)
```

AttributeError: 'frozenset' object has no attribute 'add'

## 11.2 Wiskundige operaties op verzamelingen

Om in Python te controleren of een verzameling deelverzameling is van een andere verzameling kun je gebruik maken van de ' $\leq$ '-operator:

```
In:
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'geel', 'rood'}

print(kleuren2 <= kleuren1)
```

Out:  
True

Je kunt ook gebruik maken van de methode `issubset()` om dit te controleren:

```
In:
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'geel', 'rood'}

print(kleuren2.issubset(kleuren1))
```

Out:  
True

Om te controleren of het een echte deelverzameling is maak je gebruik van de ' $<$ '-operator. Er is geen corresponderende methode.

Op dezelfde manier kun je controleren dat een verzameling een andere verzameling (echt) omvat met behulp van de ' $>$ '-operator en de ' $\geq$ '-operator. Als verzameling A de verzameling B omvat zeggen we ook wel dat A een superset is van B. In plaats van de ' $\geq$ '-operator kunnen we ook de methode `superset()` gebruiken. Merk op dat A een superset van B is dan en slechts dan als B een deelverzameling van A is!

```
In:
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'geel', 'rood'}

print(kleuren2.issuperset(kleuren1))
```

Out:  
False

De *vereniging* (Engels: union) van twee verzamelingen A en B bestaat uit alle elementen die in A of in B voorkomen. De 'of' is hier de wiskundige of inclusieve-of en de vereniging bevat dus ook alle elementen die in beide verzamelingen voorkomen. Wiskundige notatie:  $A \cup B$ .

Je zou misschien verwachten dat je twee verzamelingen kunt verenigen met de '+'-operator, maar dat blijkt niet te werken (ga na!). In plaats daarvan moet je de '|'-operator gebruiken:

```
In:
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'wit', 'geel', 'oranje', 'rood'}
kleuren3 = kleuren1 | kleuren2
print(kleuren3)
```

Out:  
{'blauw', 'rood', 'wit', 'geel', 'oranje'}

Een andere manier om de vereniging te realiseren gaat met behulp van de methode `union()`. Het verschil met de '|'-operator is dat het argument van `union()` niet persé een verzameling hoeft te zijn. Het mag ook een lijst zijn of een andere collectie die kan worden doorlopen (een zogenaamd iterabel object).

```
In:
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'wit', 'geel', 'oranje', 'rood'}
kleuren3 = kleuren1.union(kleuren2)
print(kleuren3)
```

Out:

```
{'blauw', 'rood', 'wit', 'geel', 'oranje'}
```

De *doorsnede* (Engels: intersection) van twee verzamelingen A en B bestaat uit alle elementen die in de beide verzamelingen voorkomen. Wiskundige notatie:  $A \cap B$ .

Om de doorsnede van 2 Python verzamelingen te bepalen moet je de '&'-operator gebruiken:

In:

```
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'wit', 'geel', 'oranje', 'rood'}
kleuren3 = kleuren1 & kleuren2
print(kleuren3)
```

Out:

```
{'geel', 'rood'}
```

Ook hier bestaat de mogelijkheid om de doorsnede met behulp van een methode te realiseren, in dit geval met behulp van de methode `intersection()`. Net als bij de methode `union()` hoeft het argument van `intersection()` geen verzameling te zijn, maar voldoet elk iterabel object. Stel dat we alle kleuren van verzameling `kleuren2` in een tuple `kleuren2_tupel` zetten:

In:

```
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2_tupel = ('wit', 'geel', 'oranje', 'rood')
kleuren3 = kleuren1.intersection(kleuren2_tupel)
print(kleuren3)
```

Out:

```
{'geel', 'rood'}
```

Het *verschil* (Engels: difference) van twee verzamelingen A en B bestaat uit alle elementen die wel in A maar niet in B voorkomen. Wiskundige notatie:  $A \setminus B$ .

Het verschil van 2 Python verzamelingen bepaal je met behulp van de '-'-operator:

In:

```
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'wit', 'geel', 'oranje', 'rood'}
kleuren3 = kleuren1 - kleuren2
print(kleuren3)
```

Out:

```
{'blauw'}
```

De alternatieve manier om het verschil te realiseren is met behulp van de methode `difference()`. Ook hier hoeft het argument geen verzameling te zijn, maar voldoet elk iterabel object.

## Opgave

Bepaal zelf het verschil van `kleuren1` en `kleuren2` met behulp van de methode [difference\(\)](#).

De volgorde van de verzamelingen is van belang voor het verschil. Als we in het bovenstaande voorbeeld het verschil van `kleuren2` en `kleuren1` bepalen is het resultaat:

In:

```
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'wit', 'geel', 'oranje', 'rood'}
kleuren3 = kleuren2 - kleuren1
print(kleuren3)
```

Out:

```
{'oranje', 'wit'}
```

Het *symmetrisch verschil* (Engels: symmetric difference) van twee verzamelingen A en B bestaat uit alle elementen die óf in A óf in B voorkomen. Hier gebruiken we dus de *exclusieve-of* (Engels: exclusive-or). Wiskundige notatie:  $A \oplus B$ .

Je kunt het symmetrisch verschil bepalen met behulp van de '^'-operator:

```
In:
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'wit', 'geel', 'oranje', 'rood'}
kleuren3 = kleuren1 ^ kleuren2
print(kleuren3)
```

```
Out:
{'oranje', 'blauw', 'wit'}
```

De alternatieve manier om het symmetrisch verschil te realiseren is met behulp van de methode `symmetric_difference()`. Ook nu mag het argument een willekeurig iterabel object zijn.

## Opgave

Bepaal zelf het symmetrisch verschil van `kleuren1` en een lijst met alle kleuren die in `kleuren2` voorkomen met behulp van de methode [symmetric\\_difference\(\)](#).

Voor elke besproken set-operator is er ook een samengestelde operator. We hebben dus `|=`, `&=`, `-=` en `^=`, waarbij de verzameling aan de linkerkant van de operator wordt aangepast.

## Voorbeeld:

```
In:
kleuren1 = {'rood', 'blauw', 'geel'}
kleuren2 = {'wit', 'geel', 'oranje', 'rood'}
kleuren1 |= kleuren2
print(kleuren1)
```

```
Out:
{'blauw', 'rood', 'wit', 'geel', 'oranje'}
```

Net als dictionary comprehensions definieer je *set comprehensions* met behulp van accolades. Stel dat we een nieuwe verzameling creëren met alleen de unieke even getallen van een lijst.

```
In:
getallen = [1, 2, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 10]
even_getallen = {getal for getal in getallen if getal % 2 == 0}
print(even_getallen)
```

```
Out:
{2, 4, 6, 8, 10}
```

## 11.3 Samenvatting

In dit hoofdstuk hebben de verzameling behandeld, het vierde ingebouwde samengestelde datatype in Python. Hoewel verzamelingen niet vaak gebruikt worden in Python scripts zijn ze van fundamenteel belang voor de wiskunde en theoretische informatica. In het volgende hoofdstuk bestuderen we het samengestelde datatype array, dat niet is ingebouwd in standaard Python.

## 11.4 Opgaven

Neem de tijd om onderstaande opgaven te maken, zoals je weet kun je enkel door veel te oefenen een goede programmeur worden. De antwoorden zullen gedeeld worden in de les en na afloop hiervan in Teams. Volg je de gehele cursus zelfstandig? Vraag de antwoorden dan op bij je studiecoach.

### Opgave 1

Gegeven de verzameling {3, 44, 17, 23, 58, 9, 36}

Voer de volgende opdrachten uit:

1. Voeg de waarde 27 aan de verzameling toe.
2. Verwijder de waarde 23 uit de verzameling.
3. Druk alle waarden in de verzameling tussen 20 en 50 af.

## Opgave 2

Gegeven de verzamelingen {11, 22, 33} en {5, 11, 16, 22}

Gebruik wiskundige verzamelingenoperatoren om de volgende verzamelingen te creëren:

1. {33}
2. {5, 16, 33}
3. {5, 11, 16, 22, 33}
4. {11, 22}

## Opgave 3 (wordt behandeld in de les)

Maak 2 verzamelingen met kleuren aan en controleer dat het symmetrisch verschil van die verzamelingen gelijk is aan de vereniging van de verschillen tussen beide verzamelingen. In een wiskundige formule:  $A \oplus B = (A \setminus B) \cup (B \setminus A)$

## Opgave 4

Maak nogmaals 2 verzamelingen met kleuren aan en controleer dat het symmetrisch verschil van die verzamelingen gelijk is aan het verschil tussen de vereniging en de doorsnede van beide verzamelingen. In een wiskundige formule:  $A \oplus B = (A \cup B) \setminus (A \cap B)$

# 12. Arrays met Numpy

Je hebt in de vorige hoofdstukken kennism gemaakt met enkele samengestelde datatypen die in Python zijn ingebouwd. Opvallend genoeg kent standaard Python geen arrays, terwijl dit datatype wel in bijna alle andere programmeertalen is geïmplementeerd. Sinds 2006 bestaat er echter een package waarmee je in Python met arrays kunt werken, namelijk NumPy (Numerical Python).

De functionaliteiten van arrays kunnen ook met lists worden gerealiseerd, maar bewerkingen met arrays zijn veel sneller! Bovendien zijn veel Python packages afhankelijk van NumPy, waaronder een aantal belangrijke data science bibliotheken zoals Pandas, SciPy en Keras.

Tijdens de installatie van Anaconda3 wordt naast de Python release en allerlei hulpprogramma's ook een (groot) aantal packages geïnstalleerd, waaronder NumPy. Je kunt dit in Anaconda Navigator controleren door in het menu aan de linkerkant op 'Environments' te klikken, waarna aan de rechterkant een lijst wordt getoond met alle (niet) geïnstalleerde packages.

## 12.1 Wat is een array?

Een array is een samengesteld datatype met een vaste grootte, waarvan alle elementen hetzelfde datatype hebben.

Om arrays in Python te kunnen gebruiken moeten we eerst de module numpy uit het package NumPy importeren. Het is gebruikelijk om deze module de naam np te geven:

In:

```
import numpy as np
```

Met behulp van de array-functie uit numpy kunnen we een nieuw array creëren vanuit een andere collectie. Hieronder gebruiken we een lijst om een array van integers te maken.

In:

```
getallen = np.array([6, 2, 15, 3, 9, 7])
print(type(getallen), getallen)
```

Out:

```
<class 'numpy.ndarray'> [ 6  2 15  3  9  7]
```

Allereerst valt op dat het type van getallen numpy.ndarray is, waarbij ndarray een afkorting is voor n-dimensionaal array. Verder zie je dat alle array-elementen tussen de rechte haken staan, maar zonder komma's tussen de elementen! Tenslotte zie je dat de witruimte tussen 2 elementen niet altijd gelijk is. Python bepaalt namelijk welk getal de meeste posities nodig heeft en formatteert elk (ander) getal met hetzelfde aantal posities, waarbij de waarden rechts-uitgelijnd worden. In bovenstaand voorbeeld gebruikt het getal 15 twee posities en wordt elk (ander) getal dus rechts-uitgelijnd in 2 posities.

Een individueel array-element kan bereikt worden met behulp van een index, waarbij het eerste element zoals gebruikelijk index 0 heeft. Het tweede element (met index 1!) kan op de volgende manier worden afgedrukt:

```
In:
print(getallen[1])
Out:
2
```

Je kunt de waarde van een element wijzigen met behulp van een gewone toekenning:

```
In:
getallen[1] = 4
print(getallen)
Out:
[ 6 4 15 3 9 7]
```

Tot zover lijken arrays heel erg op lijsten, maar er zijn ook belangrijke verschillen. Arrays hebben een vaste grootte (die bij creatie is bepaald) en je kunt geen elementen toevoegen of verwijderen. Verder is NumPy geoptimaliseerd voor arrays van getallen (int of float).

Een array is een object met een aantal attributen waarvan we belangrijkste hieronder bespreken:

```
In:
getallen = np.array([6, 2, 15, 3, 9, 7])
print('dtype :', getallen.dtype)
print('ndim :', getallen.ndim)
print('shape :', getallen.shape)
print('size :', getallen.size)
print('itemsize :', getallen.itemsize)
Out:
dtype : int32
ndim : 1
shape : (6,)
size : 6
itemsize : 4
```

Je ziet dat het inderdaad array attributen betreft en geen methoden omdat er geen haakjes na de namen staan!

**dtype** : het datatype van de array elementen: int32 betekent integers van 32 bits

**ndim** : het aantal dimensies van het array

**shape** : de vorm van het array uitgedrukt in een tuple, met het aantal elementen per dimensie

**size** : het totale aantal array elementen

**itemsize** : de grootte van elk array element uitgedrukt in bytes

We gaan nu een 2-dimensionaal array met floats maken en bekijken wat dan de attribuutwaarden zijn:

```
In:
fgetallen = np.array([[6.1, 2.35, 15.9], [3, 9.77, 7.68]])
print(fgetallen)
print('dtype :', fgetallen.dtype)
print('ndim :', fgetallen.ndim)
print('shape :', fgetallen.shape)
print('size :', fgetallen.size)
print('itemsize :', fgetallen.itemsize)
Out:
[[ 6.1  2.35 15.9 ]
 [ 3.   9.77 7.68]]
dtype : float64
ndim : 2
shape : (2, 3)
size : 6
itemsize : 8
```

In dit geval hebben we een array van floats bestaande uit 2 rijen en 3 kolommen. Zoals je ziet worden de waarden keurig onder elkaar gezet (alle decimale punten staan in dezelfde kolom) en het aantal posities is bepaald door de getallen met de meeste cijfers voor en na de decimale punt.

Floats worden default in arrays opgeslagen in 8 bytes (64 bits) volgens een indeling die is vastgelegd in de IEEE 754 normering.



Een array met uitsluitend nullen of enen kan als volgt worden aangemaakt:

```
In:
nullen = np.zeros(5)
enen = np.ones((2, 3), dtype=int)
print(nullen)
print(enen)
```

```
Out:
[0. 0. 0. 0. 0.]
[[1 1 1]
 [1 1 1]]
```

Met behulp van de `arange`-functie in NumPy kun je een array vullen met een range van integers:

```
In:
array_rij = np.arange(3, 9)
print(array_rij)
```

```
Out:
[3 4 5 6 7 8]
```

De vorm van een array kan veranderd worden met de methode `reshape()`. Daarbij is een eis dat het totale aantal elementen in het array niet wijzigt.

```
In:
getallen = np.array([6, 2, 15, 3, 9, 7])
getallen = getallen.reshape(2, 3)
print(getallen)
```

```
Out:
[[ 6  2 15]
 [ 3  9  7]]
```

#### Opgave

Wat is het resultaat als je in de tweede regel van bovenstaande snippet geen toekenning doet (de tweede regel is dus: `getallen.reshape(2, 3)`)?

```
Out:
[ 6  2 15  3  9  7]
```

De methode `reshape()` maakt dus een view aan van het array, maar wijzigt het array zelf niet!

## 12.2 Vergelijking van snelheden: array vs. lijst

Naast het grote aantal bewerkingen dat op arrays kan worden toegepast is ook de snelheid een belangrijke reden om arrays te gebruiken bij data analyse en andere toepassingen.

Als voorbeeld bekijken we hoe lang het duurt om een lijst en een array van 6 miljoen worpen van een gewone dobbelsteen te creëren:

```
In:
import random
import timeit
import numpy as np

starttime = timeit.default_timer()
getallijst = [random.randrange(1, 7) for i in range(6_000_000)]
print("tijd om lijst te maken :", timeit.default_timer() - starttime)
```

```
starttime = timeit.default_timer()
getalarray = np.random.randint(1, 7, 6_000_000)
print("tijd om array te maken :", timeit.default_timer() - starttime)
```

```
Out:
tijd om lijst te maken : 16.94141140001011
tijd om array te maken : 0.18764370000280906
```

Er is dus een snelheidsverschil van ruwweg een factor 100 en dat is significant als je met heel grote dataverzamelingen werkt!

Er zijn diverse operatoren en functies voor arrays. We verwijzen hiervoor naar de officiële [NumPy documentatie](https://numpy.org/doc/stable/). Net als bij lijsten en tupels kun je slicing toepassen om een deel van de array elementen te selecteren.

Tenslotte nog een voorbeeld hoe je leuke plaatjes kunt maken van numerieke data in arrays met behulp van het package Matplotlib.

```
In:
import matplotlib.pyplot as plt
import random
```

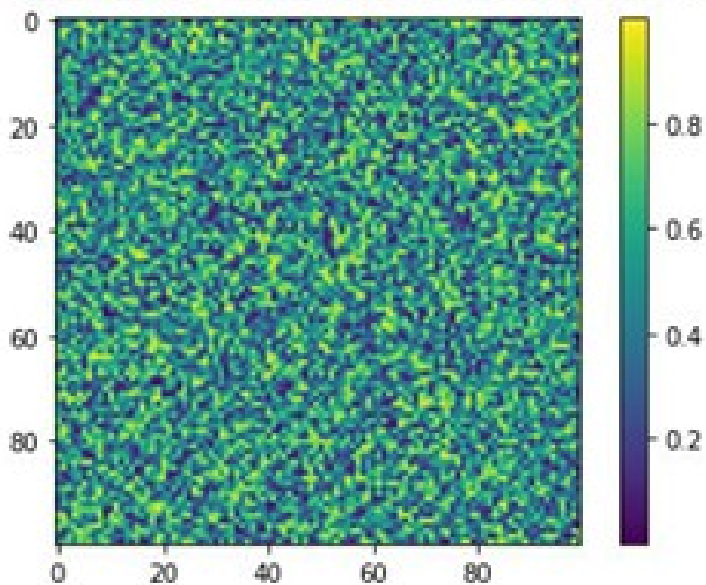
```
import numpy as np
```

```
getalarray = np.random.rand(100,100)
print(getalarray)
```

```
plt.imshow(getalarray)
plt.colorbar()
plt.show()
```

Out:

```
[ [0.37887347 0.5263136 0.44882055 ... 0.27018364 0.27693329 0.18514739]
  [0.159068 0.32202988 0.26022402 ... 0.7080022 0.32499211 0.9190673 ]
  [0.4853757 0.96618262 0.87063809 ... 0.68089733 0.15889747 0.45128022]
  ...
  [0.80898557 0.06189394 0.25744789 ... 0.20242276 0.86317712 0.94662074]
  [0.12557852 0.16414672 0.86874645 ... 0.02956456 0.05496384 0.76794391]
  [0.4486958 0.68731186 0.77810091 ... 0.25079692 0.96282132 0.59735621]]
```



In deze snippet hebben we met behulp van de methode `rand()` in de module `numpy.random` een 2-dimensionaal array van 100 bij 100 willekeurige floats gemaakt met een waarde tussen 0 en 1.

De methode `imshow()` in module `matplotlib.pyplot` laat een plaatje van deze waarden zien in een vierkant van 100 maal 100 pixels. De methode `colorbar()` verklaart de verschillende kleuren.

## 12.3 Samenvatting

In dit hoofdstuk hebben we het datatype `ndarray` uit het package `NumPy` behandeld. Deze arrays worden veel gebruikt bij het onderzoek van grote dataverzamelingen (Big Data) en kunstmatige intelligentie (Engels: Artificial Intelligence), maar zijn ook nuttig voor veel andere toepassingen.

We hebben gezien dat bewerkingen op arrays veel sneller zijn dan overeenkomstige bewerkingen op lijsten en hoe je ook met arrays leuke plaatjes kunt maken. In het volgende hoofdstuk gaan we bestuderen hoe je gegevens permanent kunt opslaan, zodat ze bewaard blijven na afloop van het programma.

## 12.4 Opgaven

### Opgave 1

Maak het array `{3 8 2 5}` en wijzig vervolgens de waarde van het tweede element in 7.

### Opgave 2

Gebruik de `arange`-functie om een array van 20 even getallen te maken van 2 tot en met 40 en wijzig de vorm vervolgens naar een 2-dimensionaal array van 4 bij 5.

### Opgave 3 (wordt besproken in de les)

Maak een array met 50 willekeurige gehele getallen tussen 1 en 6 (inclusief 1 en 6). Bepaal de index van het eerste getal 3 in het array. Als 3 niet voorkomt moet er een passende boodschap worden afgedrukt.

#### Opgave 4

We gaan de snelheid van optelling in een lijst en een array vergelijken. Maak eerst met behulp van respectievelijk list comprehension en de `arange`-functie een lijst en een array met de waarden 0, 1, 2, ..., 999999 aan.

Gebruik vervolgens de methode `default_timer()` uit de module `timeit` om te meten hoe snel alle getallen uit de lijst en het array worden opgeteld. Hierbij gebruik je de ingebouwde `sum`-functie voor de lijst en de methode `sum()` uit module `numpy` voor het array.

## 13. Bestanden (files)

Tot nu toe hebben we alleen datatypes gebruikt die slechts tijdelijk gegevens kunnen opslaan. De waarde van een lokale variabele gaat verloren als de variabele 'buiten scope' is en de waarde van een globale variabele gaat verloren als het programma eindigt. Data die bewaard moet worden ('persistent' is) moet in een bestand worden gezet, dat wordt opgeslagen op een harde schijf, SSD, USB-stick, enzovoorts.

In dit hoofdstuk leer je hoe je de data uit bestanden kunt gebruiken in je Python applicatie en hoe je een nieuw bestand kunt aanmaken of de gegevens van een bestaand bestand kunt wijzigen. Omdat er bij het werken met bestanden allerlei fouten kunnen optreden wordt in dit hoofdstuk ook het afhandelen van bijzondere foutmeldingen (Engels: Exceptions) behandeld.

### 13.1 Werken met bestanden

Python kent twee soorten bestanden: *tekstbestanden* (Engels: text files) die bestaan uit een rij van karakters en *binair bestanden* (Engels: binary files) voor het opslaan van plaatjes, video's, en dergelijke, die bestaan uit een rij van bytes. We zullen alleen tekstbestanden bekijken.

Om een bestand te kunnen gebruiken moet je hem eerst openen, waarbij Python een bestandsobject creëert voor de interactie. De karakters in een tekstbestand kunnen benaderd worden via hun indexnummer, waarbij het eerste karakter zoals gebruikelijk index 0 heeft. Het einde van een bestand wordt meestal aangegeven met behulp van een speciaal *einde-bestandsteken* (Engels: end-of-file marker).

Bij de start van een Python programma worden 3 standaard bestandsobjecten gemaakt:

- `sys.stdin` : invoer vanaf het toetsenbord, gebruikt door de `input`-functie
- `sys.stdout` : uitvoer naar de commandoregel, gebruikt door de `print`-functie
- `sys.stderr` : uitvoer naar de commandoregel, gebruikt bij programmafouten

### 13.2 Het with-statement

Bij bestandsbewerkingen gebruiken we het `with`-statement, dat de volgende syntax heeft:

```
with bestandsobject as bestandsvariabele :           #
statementblok                                     #
```

Het `with`-statement kent een variabelenaam toe aan een bestandsobject, allocceert het bestand en sluit het bestand aan het einde van het statementblok.

De ingebouwde `open`-functie opent het bestand waarvan de naam als argument wordt meegegeven en geeft een bestandsobject als return-waarde. De keyword parameter `mode` geeft aan of het bestand geopend wordt voor lezen ('r') of schrijven ('w').

### 13.3 Inlezen van gegevens uit een tekstbestand

Gewoonlijk bestaat een tekstbestand uit een aantal records met vaste lengte, waarbij in elk record enkele samenhangende gegevens zijn opgeslagen.

Onderstaand bestand bevat 3 records met studentgegevens: het studentnummer, de naam, de geboortedatum, de geboorteplaats en het jaar van inschrijving. Elk gegeven (of veld) staat op een vaste plaats in het record en heeft een vast aantal (karakter)posities ter beschikking (dit heet de *lay-out* van het record).

Tekstbestand studenten.txt:

0001	Rianne de Zwart	20000314	Den Haag	2021
0002	Ed Feunekes	19990822	Delft	2020
0003	Brigitte Bergsma	20010509	Haarlem	2021

Elke regel in bovenstaand tekstbestand correspondeert met 1 record. Records zijn dus onderling gescheiden door het nieuwe-regelteken ('\n'), dat echter niet zichtbaar is!

Stel dat je een overzicht wil maken met het studentnummer en de naam van alle studenten die zich in 2021 hebben ingeschreven dan kan dat als volgt:

In:

```
with open('c:\\pybestanden\\studenten.txt', mode='r') as studenten :
    for record in studenten :
        if int(record[57:61]) == 2021 :
            print(record[0:4], record[4:29])
```

Out:

0001 Rianne de Zwart

0003 Brigitte Bergsma

Het bestand c:\\pybestanden\\studenten.txt wordt hier dus geopend voor lezen (mode='r') en krijgt als naam studenten (in de Python code). Per iteratie in de for-lus wordt 1 record uit het bestand opgehaald en in de variabele record (van type str) gezet. Met behulp van string slicing kun je nu de waarden van de afzonderlijke velden uit het record halen. Let op het gebruik van casting om het inschrijfaar van de student te vergelijken met de (int) waarde 2021!

## 13.4 Maken van een nieuw tekstbestand

Vervolgens creëren we een nieuw tekstbestand met gegevens van (informatica)vakken. Per vak maken we een record met de code, de naam, het aantal studiepunten en het aantal inschrijvingen.

We gebruiken hierbij de methode write() om een record in het bestand te maken:

In:

```
with open('c:\\pybestanden\\vakken.txt', mode='w') as vakken :
    vakken.write('PYT Programmeren in Python  5 12\\n')
    vakken.write('DBS Databases en SQL      4 9\\n')
    vakken.write('OPS Operating systemen    5 4\\n')
```

In de map c:\\pybestanden wordt het tekstbestand vakken.txt gemaakt met de volgende records:

Vakken.txt:

PYT Programmeren in Python 5 12

DBS Databases en SQL 4 9

OPS Operating systemen 5 4

Let op de dubbele backslashes in de volledige naam van het bestand!

Een andere manier om dit bestand te creëren met behulp van de print-functie gaat als volgt:

In:

```
with open('c:\\pybestanden\\vakken.txt', mode='w') as vakken :
    print('PYT Programmeren in Python  5 12', file=vakken)
    print('DBS Databases en SQL      4 9', file=vakken)
    print('OPS Operating systemen    5 4', file=vakken)
```

Hier hoeven we geen 'n' aan het eind van elke string te zetten omdat de print-functie dat al doet.

Let op: Als je een bestand opent met mode='w' dan wordt een nieuw bestand gemaakt. Een bestaand bestand met dezelfde naam wordt vervangen door het nieuwe bestand!

## 13.5 Een bestaand tekstbestand wijzigen

Je kunt records aan het eind van een tekstbestand toevoegen door het bestand te openen met mode='a' (a is de afkorting voor het Engelse append). Met de onderstaande code wordt een 4<sup>e</sup> record in het bestand vakken.txt gezet:

In:

```
with open('c:\\pybestanden\\vakken.txt', mode='a') as vakken :
    print('PM Project Management      3 3', file=vakken)
```

Als nieuwe records tussen bestaande records moeten worden geplaatst en/of bestaande records moeten worden gewijzigd of verwijderd werkt deze aanpak niet en moet een nieuw (tijdelijk) bestand gemaakt worden. Ongewijzigde records worden daarbij gekopieerd. Aan het eind wordt het oude tekstbestand verwijderd en wordt de naam van tijdelijke bestand gewijzigd naar de originele naam van het tekstbestand.

Stel dat we in het bestand vakken.txt als eerste record een vak met code INL, naam Inleiding informatica, aantal studiepunten 2 en aantal inschrijvingen 5 willen toevoegen en alle records van vakken waarvan de code begint met een 'P' willen verwijderen. Dat kan als volgt:

In:

```
import os
```

```
vakken = open('c:\\pybestanden\\vakken.txt', mode='r')
```

```
temp = open('c:\\pybestanden\\temp.txt', mode='w')
```

```
with vakken, temp :
    temp.write('INL Inleiding informatica    2 5\n')
    for record in vakken :
        if record[0] != 'P' :
            temp.write(record)

os.remove('c:\\pybestanden\\vakken.txt')
os.rename('c:\\pybestanden\\temp.txt', 'c:\\pybestanden\\vakken.txt')
Na afloop bevat het bestand vakken.txt de volgende inhoud:
```

Tekstbestand vakken.txt:

```
INL Inleiding informatica    2 5
DBS Databases en SQL        4 8
OPS Operating systemen      5 3
```

In deze snippet hebben we de module os geïmporteerd, waarin een aantal functies staan om met het besturingssysteem (Engels: Operating System) samen te werken. Hier hebben we de remove-functie gebruikt om het oorspronkelijke vakken.txt bestand te verwijderen en de rename-functie om de naam van het tijdelijke bestand temp.txt te wijzigen in vakken.txt.

## 13.6 Werken met csv-bestanden

Een csv-bestand (csv staat voor comma-separated values) is een bestand waarin de waarden van de velden in een record gescheiden zijn door komma's (of een ander leesteken). De module csv in de Standaard Python bibliotheek bevat functies om csv-bestanden te maken en te lezen.

Zo kunnen we een csv-bestand met de gegevens van de informaticavakken als volgt maken:

```
In:
import csv

with open('c:\\pybestanden\\vakken.csv', mode='w', newline='') as vakken :
    csvwrite = csv.writer(vakken)
    csvwrite.writerow(('PYT', 'Programmeren in Python', 5, 12))
    csvwrite.writerow(('DBS', 'Databases en SQL', 4, 9))
    csvwrite.writerow(('OPS', 'Operating systemen', 5, 4))
```

Het csv-bestand vakken.csv heeft de volgende inhoud:

```
Csv-bestand vakken.csv:
PYT,Programmeren in Python,5,12
DBS,Databases en SQL,4,9
OPS,Operating systemen,5,4
```

We kunnen dit csv-bestand weer inlezen met de reader-functie in de module csv, waarbij we de afzonderlijke veldwaarden in een tuple zetten en de gegevens van alle vakken netjes onder elkaar afdrukken met de print-functie, waarbij we een geformatteerde string gebruiken.

Zo'n string bestaat uit de letter f en tussen quotes de afdruginformatie van alle velden. Die informatie staat per veld tussen accolades ('{}') en bevat de veldnaam, een dubbelepunt, een teken voor de uitlijning en het aantal posities dat het veld mag gebruiken. Het '<'-teken staat voor links uitlijnen en het '>'-teken voor rechts uitlijnen.

```
In:
import csv

with open('c:\\pybestanden\\vakken.csv', mode='r', newline='') as vakken :
    csvread = csv.reader(vakken)
    for record in csvread :
        code, naam, stp, ins = record
        print(f'{code:<4}{naam:<25}{stp:>1}{ins:>3}')
```

Out:

```
PYT Programmeren in Python  5 12
DBS Databases en SQL        4 9
OPS Operating systemen     5 4
```

Let erop dat de afzonderlijke veldwaarden in een record geen komma's mogen bevatten (of algemener: het leesteken dat als scheidingsteken in het csv-bestand fungeert) omdat bij het inlezen dan een onjuist aantal velden in dat record wordt gemaakt!

NumPy bevat functies om csv-bestanden in arrays te kunnen laden. In het volgende voorbeeld wordt een csv-bestand met behaalde cijfers ingelezen. Omdat dit bestand een kopregel bevat met de kolomnamen wordt de eerste regel van het bestand overgeslagen door de keyword parameter skiplines de waarde 1 te geven (default is 0).

```
Csv-bestand cijfers.csv:
"INL", "PYT", "DBS", "OPS"
8.3,4.7,6.2,5.6
9.0,8.5,8.2,7.7
7.4,6.6,7.1,5.9
```

In:

```
import numpy as np
```

```
cijferarray = np.loadtxt('c:\\pybestanden\\cijfers.csv', delimiter=',', skiprows=1)
print(cijferarray)
Out:
[[8.3 4.7 6.2 5.6]
 [9.  8.5 8.2 7.7]
 [7.4 6.6 7.1 5.9]]
```

## 13.7 Excepties (Exceptions)

We hebben in de voorgaande hoofdstukken al veel voorbeelden gezien waarbij een foutmelding wordt getoond als we de Python code runnen. Soms is de oorzaak een syntaxfout, maar de reden kan ook onjuiste invoer zijn, delen door 0, een onbekende bestandsnaam, of een andere logische fout die voorkomen kan worden door een controle vooraf.

Speciale aandacht is nodig bij invoer door gebruikers. Als jouw programma een int verwacht en de gebruiker voert per ongeluk een float in of moedwillig een willekeurige string dan moet die fout door het programma worden opgevangen zodat het niet crasht.

Als er tijdens het runnen van een stuk code een onverwachte fout optreedt genereert Python een *exceptie* (Engels: exception). Excepties kun je *afvangen* (Engels: exception handling) met behulp van het try-statement, dat de volgende syntax heeft:

```
try :
    statementblok          # wordt uitgevoerd totdat een exceptie optreedt
except exception1 :
    statementblok1         # wordt uitgevoerd als exception1 optreedt
    ...
except exceptionL :
    statementblokL         # wordt uitgevoerd als exceptionL optreedt
else :
    statementsblokE        # wordt uitgevoerd als geen enkele exceptie optreedt
finally :
    statementsblokF        # wordt altijd uitgevoerd
```

De else- en finally-clausules zijn optioneel en hoeven dus niet voor te komen in een try-statement.

Stel dat we 10 willen delen door een geheel getal dat de gebruiker moet invoeren:

In:

```
try :
    noemer = int(input('noemer :'))
    deling = 10 / noemer
except ValueError :
    print('U heeft geen geheel getal ingevoerd')
except ZeroDivisionError :
    print('U mag niet delen door 0')
else:
    print ('10 / noemer =', deling)
```

Out:

```
noemer : 4.5
U heeft geen geheel getal ingevoerd
Out:
noemer : hallo
U heeft geen geheel getal ingevoerd
Out:
noemer : 0
U mag niet delen door 0
Out:
noemer : 3
10 / noemer = 3.3333333333333335
```

Als je wilt afdwingen dat de gebruiker een geldige waarde invoert (in bovenstaand voorbeeld dus een geheel getal ongelijk 0) dan moet je het try-statement in een oneindige lus zetten (met behulp van het while-statement), waaruit je alleen maar kunt ontsnappen via een break-statement in de else-clausule.

Bij het werken met bestanden kun je ook tegen bepaalde excepties aanlopen. Een exceptie die vaak optreedt is `FileNotFoundError` als het programma een bestand niet kan vinden omdat het bestand niet bestaat (of in een andere map staat) of de naam niet helemaal correct is.

## 13.8 Samenvatting

In dit hoofdstuk hebben we geleerd hoe we tekstbestanden kunnen maken, inlezen en wijzigen. Ook hebben we gezien hoe Python bepaalde run-time fouten kan opvangen en afhandelen.

## 13.9 Opgaven

### *Opgave 1*

Maak een tekstbestand met de naam, de geboortedatum en het telefoonnummer van je vrienden.

### *Opgave 2*

Maak een programma waarbij je de naam van een vriend invoert en het programma de gegevens van die vriend afdruckt met behulp van het tekstbestand dat je in opgave 1 hebt gemaakt.

### *Opgave 3 (wordt besproken in de les)*

Bepaal het gemiddelde cijfer per vak uit het bovengenoemde csv-bestand met behaalde cijfers.

## 14. Objectoriëntatie

In dit hoofdstuk leer je meer (software)systemen realiseren tijdens het ontwikkelproces door middel van objectoriëntatie (OO)

Objectoriëntatie (OO) is een manier om (software)systemen te realiseren en kan gebruikt worden tijdens het gehele ontwikkelproces (analyse, ontwerp en implementatie). Een objectgeoriënteerd systeem bestaat uit samenwerkende bouwstenen (objecten) die onderling communiceren via berichten.

Objectoriëntatie heeft zijn wortels in modulaire systemen, maar breidt die uit met nieuwe voorzieningen als instantiatie, overerving en dynamische binding.

### 14.1 Begrippen van objectoriëntatie

De begrippen object, identiteit, klasse, inkapseling, overerving en dynamische binding vormen de kern van de objectgeoriënteerde benadering. In deze paragraaf bespreken we eerst de betekenis van elk begrip en in de volgende paragrafen kijken we hoe ze in Python zijn geïmplementeerd.

#### Object

Een object is een zelfstandige systeemcomponent die communiceert met andere objecten door berichten te versturen en verwerken. Objecten verenigen toestand (data) en gedrag (operaties) in zich. Elk object heeft interne variabelen (meestal attributen of instantievariabelen genoemd) waarin gegevens zijn opgeslagen. Verder is er een verzameling operaties of methoden die andere objecten kunnen aanroepen om iets van het object gedaan te krijgen. Een van de ideeën van objectgeoriënteerd ontwikkelen is dat de objecten waaruit een informatiesysteem is opgebouwd, corresponderen met de reële wereld en dus herkenbare concepten in het betreffende domein zijn.

#### Identiteit en verwijzing

Een belangrijk kenmerk van objecten is dat ze een identiteit hebben. Dit garandeert dat twee objecten die instanties zijn van dezelfde klasse en ook dezelfde waarden hebben voor de attributen, toch verschillend zijn. Identiteit is een dienst die wordt geleverd door de omgeving waarin objecten zijn gecreëerd. De identiteit van een object is van belang als men objecten onderling wil koppelen. Door in een attribuut van een object de identiteit van een ander object op te nemen kunnen we verwijzingen tussen objecten creëren. Hierdoor kunnen we dus ook vanuit het ene object berichten sturen naar een ander object (het zogenaamde delegeren).

#### Klasse

In de meeste talen en omgevingen beschrijven we geen individuele objecten, maar klassen van objecten. Een klasse definieert zowel de structuur als het gedrag voor een bepaalde objectfamilie. Objecten zijn instanties van een klasse. De creatie of instantiatie van een nieuw object van een klasse is een dienst die de taal of het systeem biedt. De attribuutdefinities van de klasse dienen om de inhoud van een object te initialiseren.

## Inkapseling

De moderne kijk op objecten is sterk beïnvloed door ideeën rond modulaire systemen. Het centrale begrip is inkapseling (encapsulation) of information hiding. In feite is een object een black box: andere objecten zien de buitenkant, maar niet de binnenkant. Een object heeft nooit zomaar toegang tot de interne zaken van een ander object, maar kan alleen de voor de buitenwereld zichtbare (publieke) methoden aanroepen die voor dat andere object zijn gedefinieerd. De publieke methoden vormen de interface, die de functionaliteit weergeeft van een object, dus wat met een object kan worden gedaan. De privé-methoden in de black box bepalen hoe de interface wordt gerealiseerd. Om de samenwerking tussen de verschillende delen van een OO-systeem vast te leggen volstaat dus in principe de bepaling van de systeemdelen, de beschrijving van hun interfaces en de vastlegging van wie met wie berichten uitwisselt. In feite beschrijven we een systeem als een verzameling diensten zonder dat we hoeven uit te wijden over hoe die worden gerealiseerd.

## Overerving

Via overerving ontstaat een nieuwe klasse, gebaseerd op één (of meer) andere klassen. De nieuwe klasse is een subklasse van een bestaande (super)klasse. In de subklasse kunnen we extra attributen definiëren, maar ook nieuwe methoden toevoegen en/of bestaande methoden anders implementeren. Instanties van een subklasse bevatten waarden voor de attributen uit de subklasse en alle superklassen. Op zo'n object kunnen andere objecten zowel de interfacemethoden uit de subklasse als die uit de superklasse(n) aanroepen. Het vinden van de goede methode bij een bericht wordt binding genoemd. We onderscheiden enkelvoudige overerving, waarbij een klasse maximaal één directe superklasse mag hebben, en meervoudige overerving, waarbij een klasse kan erven van meerdere klassen. De ontwikkeling van een goede klassenhiërarchie is een van de beste, maar ook moeilijkste, vormen om tot goed herbruikbare componenten te komen. Een goede klassenhiërarchie voor een bepaald probleemdomain ontwikkelen kost tijd. Het is een iteratief proces, waarbij bestaande klassen algemener worden gemaakt (generalisatie) en verdere verfijningen worden toegevoegd (specialisatie).

## Dynamische binding en polymorfie

Een aspect dat bij (abstracte) klassen en overerving naar voren komt, is dynamische binding van berichten aan methoden. Dynamische binding maakt het makkelijk een systeem aan te passen of uit te breiden. Als een ander deel van het systeem een object van een bepaalde klasse verwacht, mag het ook met een object worden geconfronteerd van een nieuwe subklasse, zo lang die de interface van de superklasse maar blijft waarmaken. Door de aanpassingen en uitbreidingen te plaatsen in nieuwe subklassen en gebruik te maken van dynamische binding hoeft men de rest van het systeem niet meer aan te passen. De consequentie van overerving is dus dat we methoden met dezelfde naam (en betekenis) kunnen toepassen op objecten van verschillende klassen. Dit heet polymorfie. Het voordeel is dat minder verschillende namen voor operaties nodig zijn, hetgeen de overzichtelijkheid van het systeem bevordert.

## Hergebruik

Overerving is een techniek die op verschillende manieren kan worden gebruikt en misbruikt. Het is in ieder geval een middel voor hergebruik. We definiëren een nieuwe klasse van objecten door iets toe te voegen aan of iets te veranderen ten opzichte van een bestaande klasse. Van de superklasse erven (en dus hergebruiken) we alle niet veranderde methoden.

Hergebruik via overerving kan nuttig zijn, maar moet niet als doel op zich worden gezien. Er zijn risico's voor onderhoudbaarheid, omdat een subklasse (meestal) toegang heeft tot de implementatie van de superklasse. Als de implementatie van de superklasse wordt veranderd, moeten we dus ook de subklasse veranderen. We spreken daarom vaak van white-box reuse, in tegenstelling tot black-box reuse, dat ontstaat als een object een ander object gebruikt.

## 14.2 Klassen, objecten en inkapseling in puton

Een klasse wordt in Python gedefinieerd middels het keyword `class`. De syntax is als volgt:

```
class klassenaam :      #
initialisatiemethode    #
overige methoden      #
```

De initialisatiemethode heeft de volgende syntax:

```
def __init__(parameters) :      # dubbele underscore ('_') voor en na 'init'
statementblok                  # definitie en initialisatie van alle attributen
```

De syntax van een overige methode is gelijk aan de syntax van een functie:

```
def methodenaam(parameters) :    # methodeheader
statementblok                  # methodebody
```

De eerste parameter van de initialisatiemethode en overige methoden moet een verwijzing naar het object zijn. Bijna altijd wordt hiervoor de aanduiding **self** gebruikt. De attributen van een klasse worden in het statementblok van de initialisatiemethode gedefinieerd, en zijn herkenbaar door de aanduiding: **self.attribuutnaam**. Als je een attribuut gebruikt in een van de overige methoden moet je eveneens deze aanduiding hanteren.



In het volgende voorbeeld maken we een klasse `Dobbelsteen`. Een object van deze klasse stelt een 'gewone' dobbelsteen voor die na een worp de waarde 1, 2, 3, 4, 5 of 6 heeft. De klasse heeft 1 attribuut (`waarde`) en 1 overige methode (`werp()`). Waarde is initieel 0 en krijgt na aanroep van `werp()` een random (geheeltallige) waarde tussen 1 en 6.

Een `Dobbelsteen` object wordt gemaakt door een toekenning van de vorm:

```
variabelenaam = Dobbelsteen()
```

Hierbij wordt automatisch de initialisatiemethode uitgevoerd.

In:

```
import random
```

```
class Dobbelsteen :
    def __init__(self) :
        self.waarde = 0

    def werp(self) :
        self.waarde = random.randrange(1, 7)
```

```
# hoofdprogramma
d = Dobbelsteen()
d.werp()
print('worp :', d.waarde)
Out:
worp : 3
```

In het hoofdprogramma wordt dus 1 object (met identiteit 'd') van de klasse `Dobbelsteen` gemaakt en daarmee wordt eenmaal 'geworpen'. Uiteraard kun jij een ander resultaat krijgen als je deze snippet uitvoert!

Het nadeel van bovenstaande definitie van `Dobbelsteen` is dat het attribuut `waarde` overal in het programma benaderd en gewijzigd kan worden. Het is daarom gebruikelijk om attribuutnamen te beginnen met een underscore ('\_'), waarmee wordt aangegeven dat de naam alleen intern in de klasse gelezen en gewijzigd mag worden. Dit is echter een conventie en voorkomt niet dat een onvoorzichtige of kwaadwillende ontwikkelaar direct in een programma attribuutwaarden ophaalt of aanpast.

Iets veiliger is het om attribuutnamen met 2 underscores te beginnen. Hiermee kun je in Python privé-gegevens (Engels: private data) simuleren. Bekijk het volgende voorbeeld:

In:

```
import random
```

```
class Dobbelsteen :
    def __init__(self) :
        self.__waarde = 0

    def werp(self) :
        self.__waarde = random.randrange(1, 7)
```

```
# hoofdprogramma
d = Dobbelsteen()
d.werp()
print('worp :', d.__waarde)
```

Out:

```
-----
AttributeError                                Traceback (most recent call last)
Input In [], in <cell line: 13>()
      11 d = Dobbelsteen()
      12 d.werp()
--> 13 print('worp :', d.__waarde)
```

```
AttributeError: 'Dobbelsteen' object has no attribute '__waarde'
```

Python kan het attribuut `waarde` dus niet direct benaderen. Voor zo'n attribuut hebben we daarom methoden nodig om de inhoud te lezen of te wijzigen. Deze methoden worden wel de getter en setter van het attribuut genoemd.

De get methode (de getter) heeft de volgende header:

```
def getAttribuutnaam(self) :
```

De set methode (de setter) heeft de volgende header:

```
def setAttribuutnaam(self, nieuweWaarde) :
```

In het voorbeeld met de Dobbelsteen hebben we alleen een getter nodig om de geworpen waarde op te vragen. Het attribuut waarde kan immers alleen gewijzigd worden door het werpen van de dobbelsteen.

In:

```
import random

class Dobbelsteen :
    def __init__(self) :
        self.__waarde = 0

    def getWaarde(self) :
        return self.__waarde

    def werp(self) :
        self.__waarde = random.randrange(1, 7)

# hoofdprogramma
d = Dobbelsteen()
d.werp()
print('worp :', d.getWaarde())
```

Out:

4

Stel dat we nu een algemene dobbelsteen willen definiëren die alle gehele waarden bevat tussen een gegeven minimum waarde en een maximum waarde.

In:

```
import random

class Dobbelsteen :
    def __init__(self, min_waarde=1, max_waarde=6) :
        self.__MIN_WAARDE = min_waarde
        self.__MAX_WAARDE = max_waarde
        self.__waarde = 0

    def getWaarde(self) :
        return self.__waarde

    def werp(self) :
        self.__waarde = random.randrange(self.__MIN_WAARDE, self.__MAX_WAARDE + 1)
```

Omdat de minimum en maximum waarde na initialisatie niet veranderd mogen worden noteren we ze als constanten (dus met hoofdletters).

We kunnen zo bijvoorbeeld het werpen van een twaalfkantige dobbelsteen simuleren waarop de waarden 3 tot en met 14 staan.

In:

```
d = Dobbelsteen(3, 14)
d.werp()
print('worp :', d.getWaarde())
```

Out:

Worp : 11

Een gewone dobbelsteen kun je op 2 manieren maken. Enerzijds kun je de minimale en maximale waarde 1 en 6 meegeven, maar omdat dit de defaultwaarden zijn mag je ze ook weglaten. In de onderstaande snippet worden 2 gewone dobbelstenen d1 en d2 gecreëerd:

In:

```
d1 = Dobbelsteen(1, 6)
d2 = Dobbelsteen()
```

Vervolgens willen we een overzicht kunnen maken van het aantal keren dat een mogelijke waarde van de dobbelsteen is geworpen. Bij een gewone dobbelsteen is dit dus het aantal keren dat een 1 is geworpen, het aantal keren dat een 2 is geworpen, ..., het aantal keren dat een 6 is geworpen. Dit overzicht

wordt gemaakt met behulp van de methode `toonWaarden()`.

Je kunt deze methode op verschillende manieren implementeren. Een manier is alle geworpen waarden in een lijst opslaan en bij aanroep van de methode het aantal keren dat elke waarde in de lijst voorkomt te tellen:

In:

```
import random

class Dobbelsteen :
    def __init__(self, min_waarde=1, max_waarde=6) :
        self.__MIN_WAARDE = min_waarde
        self.__MAX_WAARDE = max_waarde
        self.__waarde = 0
        self.__worpen = []

    def getWaarde(self) :
        return self.__waarde

    def werp(self) :
        self.__waarde = random.randrange(self.__MIN_WAARDE, self.__MAX_WAARDE + 1)
        self.__worpen.append(self.__waarde)

    def toonWaarden(self) :
        for i in range(self.__MIN_WAARDE, self.__MAX_WAARDE + 1) :
            print(i, ': ', self.__worpen.count(i))
```

Je zou nu in het hoofdprogramma bijvoorbeeld 600000 worpen kunnen simuleren en kijken wat het resultaat is:

In:

```
d = Dobbelsteen()

for i in range(600_000) :
    d.werp()

d.toonWaarden()
```

Out:

```
1 : 99803
2 : 99888
3 : 99995
4 : 100336
5 : 99766
6 : 100212
```

De methode `toonWaarden()` kan ook geïmplementeerd worden met behulp van een array:

In:

```
import random
import timeit
import numpy as np

class Dobbelsteen :
    def __init__(self, min_waarde=1, max_waarde=6) :
        self.__MIN_WAARDE = min_waarde
        self.__MAX_WAARDE = max_waarde
        self.__waarde = 0
        self.__worpen = np.zeros(self.__MAX_WAARDE - self.__MIN_WAARDE + 1, dtype=int)

    def getWaarde(self) :
        return self.__waarde

    def werp(self) :
        self.__waarde = random.randrange(self.__MIN_WAARDE, self.__MAX_WAARDE + 1)
        self.__worpen[self.__waarde - self.__MIN_WAARDE] += 1

    def toonWaarden(self) :
        for i in range(self.__MIN_WAARDE, self.__MAX_WAARDE + 1) :
            print(i, ': ', self.__worpen[i - self.__MIN_WAARDE])
        self.__worpen.append(self.__waarde)
```

```
def toonWaarden(self) :
    for i in range(self.__MIN_WAARDE, self.__MAX_WAARDE + 1) :
        print(i, ': ', self.__worpen.count(i))
```

## 14.3 Overerving en binding in Python

Stel dat je nu een nieuwe klasse wil maken waarbij dobbelstenen ook een kleur hebben. Dan kun je een groot deel van de code van de klasse Dobbelssteen kopiëren, maar dat is natuurlijk geen elegante oplossing. Een wijziging in de code van klasse Dobbelssteen zal waarschijnlijk ook in de code van de nieuwe klasse moeten worden uitgevoerd!

Gelukkig kunnen we gebruikmaken van *overerving* om dit soort problemen te voorkomen. Bedenk dat een gekleurde dobbelsteen in feite een dobbelsteen is met een extra eigenschap (kleur). Dit 'is-een' (Engels: is-a) verband tussen verschillende klassen is kenmerkend voor overerving. Een andere manier om het verband uit te drukken is te zeggen dat de klasse GekleurdeDobbelssteen een *subklasse* is van de klasse Dobbelssteen. Nog een andere formulering is dat de verzameling objecten van de klasse GekleurdeDobbelssteen een deelverzameling van de verzameling objecten van de klasse Dobbelssteen is.

De definitie van de klasse GekleurdeDobbelssteen is als volgt:

```
In:
class GekleurdeDobbelssteen(Dobbelssteen) :
    def __init__(self, min_waarde=1, max_waarde=6, kleur='wit') :
        super().__init__(min_waarde, max_waarde)
        self.__kleur = kleur

    def getKleur(self) :
        return self.__kleur
```

Na de klassenaam staat tussen haakjes de naam van de superklasse (hier dus: Dobbelssteen).

De klasse heeft een aangepaste initialisatiemethode, waarin de kleur is toegevoegd (default: wit). Het eerste statement van deze methode is een aanroep van de initialisatiemethode van de superklasse met behulp van de methode `super().__init__(...)`. Let erop dat je daarbij niet `self` als argument meegeeft!

De enige methode die we nu nog hoeven toe te voegen is een getter voor de kleur. De aanname is dat de kleur van een dobbelsteen niet gewijzigd kan worden.

De klasse GekleurdeDobbelssteen erft alle attributen en methoden van de klasse Dobbelssteen en we kunnen nu dus een rode dobbelsteen creëren en daarmee werpen:

```
In:
d = GekleurdeDobbelssteen(kleur='rood')
d.werp()
print(d.getKleur(), d.getWaarde())
Out:
rood 3
```

We gaan vervolgens aan de klassen Dobbelssteen en GekleurdeDobbelssteen een methode `toon()` toevoegen. Bij aanroep van `toon()` drukt een Dobbelssteen object de boodschap "De ongekleurde dobbelsteen heeft waarde <w>" af, waarbij <w> de laatst geworpen waarde is. Bij aanroep van `toon()` drukt een object van GekleurdeDobbelssteen de boodschap "De dobbelsteen met kleur <k> heeft waarde <w>" af waarbij <k> de kleur en <w> de laatst geworpen waarde is.

In de volgende snippet staat de volledige code van beide klassen. In het hoofdprogramma maken we een gewone en een gekleurde dobbelsteen aan en gooien met beide dobbelstenen één keer. Tenslotte roepen we voor beide dobbelstenen de methode `toon()` aan:

```
In:
import random
import timeit

class Dobbelssteen :
    def __init__(self, min_waarde=1, max_waarde=6) :
        self.__MIN_WAARDE = min_waarde
        self.__MAX_WAARDE = max_waarde
        self.__waarde = 0
        self.__worpen = []

    def getWaarde(self) :
        return self.__waarde
```

```

def werp(self) :
    self.__waarde = random.randrange(self.__MIN_WAARDE, self.__MAX_WAARDE + 1)
    self.__worpen.append(self.__waarde)

def toonWaarden(self) :
    for i in range(self.__MIN_WAARDE, self.__MAX_WAARDE + 1) :
        print(i, ': ', self.__worpen.count(i))

def toon(self) :
    print('De ongekleurde dobbelsteen heeft waarde', self.__waarde)

class GekleurdeDobbelsteen(Dobbelsteen) :
    def __init__(self, min_waarde=1, max_waarde=6, kleur='wit') :
        super().__init__(min_waarde, max_waarde)
        self.__kleur = kleur

    def getKleur(self) :
        return self.__kleur

    def toon(self) :
        print('De dobbelsteen met kleur', self.__kleur, 'heeft waarde', self.getWaarde())

# hoofdprogramma
d1 = Dobbelsteen()
d2 = GekleurdeDobbelsteen(kleur='rood')
d1.werp()
d2.werp()
d1.toon()
d2.toon()

```

Out:

De ongekleurde dobbelsteen heeft waarde 2

De dobbelsteen met kleur rood heeft waarde 3

We zien hier dat het aanroepen van de methode toon() voor een Dobbelsteen object een ander effect heeft dan voor een GekleurdeDobbelsteen object. Hier is dus sprake van polymorfie. We zeggen dat methode toon() in klasse Dobbelsteen wordt *overschreven* (Engels: overridden) door methode toon() in klasse GekleurdeDobbelsteen.

Merk op dat je in de methode toon() van GekleurdeDobbelsteen de getter self.getWaarde() moet gebruiken omdat het attribuut self.\_\_waarde niet bereikbaar is (ga na!).

## 14.4 Verbanden tussen objecten vastleggen

In de vorige paragraaf is al gezegd dat verbanden tussen objecten kunnen worden vastgelegd door verwijzing-en op te nemen, waarbij een verwijzing naar een object niets anders is dan de identificatie van dat object. In deze paragraaf gaan we daar verder op in.

Als voorbeeld bekijken we een dobbelspel, waarbij we 2 gekleurde dobbelstenen werpen en hun waarden optellen om de uitkomst te bepalen. Stel dat we met 2 gewone dobbelstenen de waarden 5 en 3 gooien dan is de uitkomst 8. Met 2 gewone dobbelstenen kun je dus als uitkomst 2, 3, ..., 12 krijgen, maar vermoedelijk weet je dat niet elke uitkomst even waarschijnlijk is. De kans op de uitkomst 7 is namelijk 6 keer zo groot als de kans op uitkomst 2 (of 12)!

We maken nu een nieuwe klasse Dobbelspel, waarbij elk Dobbelspel object 2 Dobbelsteen objecten gebruikt om mee te werpen. De klasse heeft een methode werpDobbelstenen() die de som van de geworpen waarden opslaat in het attribuut uitkomst, een getter voor uitkomst en een methode toonUitkomsten() die een samenvatting geeft van alle worpen.

In het hoofdprogramma worden eerst 2 dobbelstenen gecreëerd die aan het dobbelspel worden toegevoegd en vervolgens wordt met beide dobbelstenen 360000 maal geworpen. Aan het eind wordt het overzicht getoond.

In:

```

class Dobbelspel :
    def __init__(self, dobbelsteen1, dobbelsteen2) :
        self.__dobbelsteen1 = dobbelsteen1
        self.__dobbelsteen2 = dobbelsteen2
        self.__uitkomst = 0

```

```

        self.__uitkomsten = []

    def getUitkomst(self) :
        return self.__uitkomst

    def werpDobbelstenen(self) :
        self.__dobbelsteen1.werp()
        self.__dobbelsteen2.werp()
        self.__uitkomst = self.__dobbelsteen1.getWaarde() + self.__dobbelsteen2.getWaarde()
        self.__uitkomsten.append(self.__uitkomst)

    def toonUitkomsten(self) :
        print('dobbelsteen 1 :', self.__dobbelsteen1.getKleur())
        self.__dobbelsteen1.toonWaarden()
        print('dobbelsteen 2 :', self.__dobbelsteen2.getKleur())
        self.__dobbelsteen2.toonWaarden()
        print('uitkomsten')
        for i in range(2, 13) :
            print(f'{i:>2} : {self.__uitkomsten.count(i)}')

# hoofdprogramma
d1 = GekleurdeDobbelsteen(kleur='geel')
d2 = GekleurdeDobbelsteen(kleur='rood')
spel = Dobbelspel(d1, d2)

for i in range(360_000) :
    spel.werpDobbelstenen()

```

```
spel.toonUitkomsten()
```

Out:

```
dobbelsteen 1 : geel
```

```
1 : 60148
```

```
2 : 59903
```

```
3 : 59503
```

```
4 : 60221
```

```
5 : 60269
```

```
6 : 59956
```

```
dobbelsteen 2 : rood
```

```
1 : 59744
```

```
2 : 60019
```

```
3 : 59935
```

```
4 : 60380
```

```
5 : 60033
```

```
6 : 59889
```

```
uitkomsten
```

```
2 : 10065
```

```
3 : 19812
```

```
4 : 29920
```

```
5 : 40151
```

```
6 : 49851
```

```
7 : 60205
```

```
8 : 49795
```

```
9 : 40016
```

```
10 : 30031
```

```
11 : 20170
```

```
12 : 9984
```

## 14.5 Samenvatting

In dit hoofdstuk heb je geleerd hoe je een objectgeoriënteerd Python programma kunt maken. Bij het ontwikkelen van een nieuwe klasse moet je eerst bedenken welke functionaliteiten die klasse moet aanbieden. Met andere woorden: je bepaalt eerst de methoden die gebruikers kunnen aanroepen voor een object van die klasse en vervolgens implementeer je deze methoden met behulp van een aantal interne variabelen (attributen).

## 14.6 Opgaven

*Opgave 1*

Wijzig de code van de klasse Dobbelsteen zodat intern een woordenboek (dictionary) gebruikt wordt in plaats van een lijst of een array.

#### *Opgave 2*

Maak een klasse Docent waarbij van elke docent de naam en het salaris kan worden vastgelegd. Maak getters en setters voor beide attributen en een methode toonGegevens, waarbij in een zin de naam en het salaris van een docent worden afgedrukt.

#### *Opgave 3*

Maak een subklasse DeeltijdDocent van Docent, waarbij voor elke deeltijddocent ook een factor met een waarde tussen 0 en 1 is vastgelegd. Factor 0.2 betekent bijvoorbeeld dat een deeltijddocent een aanstelling van 20% heeft. Maak een getter en setter voor dit attribuut. Maak ook een aangepaste methode toonGegevens waarbij van elke deeltijddocent de naam, het salaris en de factor worden afgedrukt.

#### *Opgave 4*

Maak een klasse Student waarbij van elke student het studentnummer, de naam, de geboortedatum en het jaar van inschrijving kan worden vastgelegd. Maak getters en setters voor de attributen en een methode toonGegevens, waarbij alle gegevens van een student worden afgedrukt.

#### *Opgave 5*

Elke student kan maximaal één docent als studiebegeleider hebben. Anderzijds kan een docent 0, 1 of meerdere studenten begeleiden. Pas de klassen uit opgaven 2, 3 en 4 aan zodat je kunt vastleggen welke studenten door een docent worden begeleid.