FACULTEIT
BIO-INGENIEURSWETENSCHAPPEN

WETENSCHAPPELIJK PROGRAMMEREN

Syllabus

Prof. dr. ir. Jan Verwaeren
Dr. Demir Ali Köse
Dr. Gauthier Vanhaelewyn
Ir. Lander De Visscher

Eerste Bachelor in de Bio-ingenieurswetenschappen
Academiejaar 2024–2025

UNIVERSITEIT
GENT

Inhoudsopgave

Voorwoord

Genomics, precisielandbouw, precision lifestock farming, robotisatie, big data, artificiële intelligentie en industrie 4.0 zijn voorbeelden van hoog-technologische disciplines en sectoren waar zeer veel ingenieurs in de levende materie in contact mee komen. Bovendien kennen al deze sectoren een hoge graad van informatisering en automatisatie. Een gevolg daarvan is dat ingenieurs vaak zeer grote hoeveelheden ruwe gegevens moeten kunnen verwerken, en op basis van deze verwerking beslissingen nemen, in een beperkte tijdspanne. Enkele voorbeelden:

- Een bedrijf dat sla teelt in hydrocultuur heeft de laatste tijd te kampen met valse meeldauw. Je wordt gevraagd hiervan de oorzaak te achterhalen en beschikt onder andere over metingen van temperatuur, luchtvochtigheid, CO_2 concentratie, fotosynthetisch-actieve-lichtintensiteitsmetingen in de serres op basis van tientallen sensoren van de laatste jaren (meer dan 10000 metingen) …en uiteraard ook je biologische kennis.
- Een plantenveredelingsbedrijf wenst na te gaan of een nieuw tomatenras een verhoogde resistentie heeft tegen bladvlekkenziekte. Om dit gedetailleerd te kunnen opvolgen wordt een proef aangelegd met 200 planten in potten. Deze potten worden op een transportband geplaatst en gedurende 10 weken wordt elke plant twee maal per dag automatisch gefotografeerd. Op basis van de resulterende 28000 foto's wenst men een gedetailleerd beeld te krijgen van resistentie tegen bladvlekkenziekte tijdens de ontwikkeling van de plant.
- Om inzicht te krijgen de infectieproces van *Fusarium graminearum* (een pathogeen op verschillende granen) wenst men een nieuwe mutant te maken waarin men een bepaald gen uitschakelt. Om dit gen gericht te kunnen uitschakelen moet je uiteraard op zoek gaan naar de plaats van dit gen in het genoom van dit organisme (ongeveer 36 miljoen basenparen).

De hoeveelheden gegevens die moeten verwerkt worden om de bovenstaande problemen het hoofd te kunnen bieden zijn van een grootte-orde waarbij een aanpak met pen-en-papier geen optie is. Vaak doet men dan ook beroep op computers om deze problemen op te lossen. Het gebruik van spreadsheets (MS Excel) of specifieke software kan soms een oplossing bieden, maar vaak zijn problemen dermate specifiek dat er geen software bestaat, of dat deze software niet flexibel genoeg is om de nodige verwerking uit te voeren. In dergelijke gevallen zal je zelf software[1] moeten schrijven die je kan helpen bij deze analyses. In deze cursus wordt de basis gelegd voor de programmeervaardigheden die je daarvoor nodig hebt.

[1] De term script is hier meer gepast, zie verder.

Concreet leer je in deze cursus de basisprincipes van gestructureerd, modulair en deels object-georiënteerd programmeren, alsook deze principes toe te passen om problemen op een tijdsefficiënte en geautomatiseerde manier op te lossen. Tijdens de lessen wordt aangeleerd hoe een taak die geformuleerd wordt in een natuurlijke taal kan omgezet worden in een programma dat door een computer kan uitgevoerd worden. Hierbij wordt gebruik gemaakt van de programmeertaal Python. Deze cursus kan je dan ook in de eerste plaats beschouwen als een inleiding tot programmeren in Python 3.

Merk op dat deze cursusnota's deel uitmaken van een lessenpakket, waarbij ook lesslides horen die beschikbaar zijn via het elektronisch leerplatform Ufora.

1

Computers, programma's en Python

Nagenoeg iedereen maakt dagelijks gebruik van een computer. Met bepaalde computers kan je surfen op het web, computerspelletjes spelen, een stuk tekst schrijven, enz. tot zelfs het weer voorspellen. De vragen *'Wat is een computer precies?'* of *'Waarom kan een computer zo veel verschillende taken uitvoeren?'* gebruiken we als startpunt in deze inleidende programmeercursus.

1.1 Een flexibele machine

Een moderne computer kan men definiëren als een '*Machine die informatie opslaat en manipuleert onder controle van een wijzigbaar programma*'. We onderscheiden in deze definitie twee elementen. Ten eerste is een computer een toestel dat informatie manipuleert. Dit wil zeggen dat we informatie in een computer kunnen invoeren en dat de computer deze informatie kan manipuleren en transformeren en tenslotte het resultaat kan tonen aan de gebruiker (bijvoorbeeld op het scherm). In die zin verschilt een computer weinig van een eenvoudig rekentoestel of een digitale chronometer. Ook daarop kan je informatie invoeren, verwerken en het resultaat bekijken.

Het belangrijkste verschil tussen een rekentoestel en een (universele) computer is het deel 'controle door een wijzigbaar programma' in de definitie. De taken die een eenvoudig rekentoestel kan uitvoeren zijn vaak beperkt tot het optellen van een aantal getallen. De verschillende programma's (of software) die op een computer aanwezig zijn daarentegen, stellen dit toestel in staat om meerdere taken, die sterk van elkaar kunnen verschillen, uit te voeren. De mogelijkheid om verschillende programma's te kunnen uitvoeren is dus een belangrijk aspect, maar wat bedoelt men hier precies met ***'een (computer)programma'***?

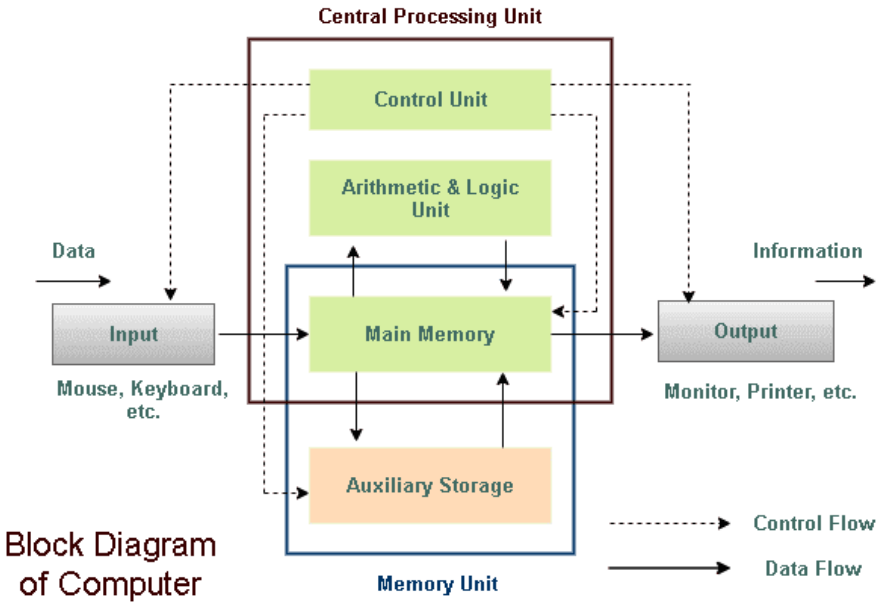
Een *computerprogramma* is een gedetailleerde, stap-voor-stap omschreven, sequentie van instructies die ondubbelzinnig beschrijven wat een computer moet doen. Wanneer een programma gewijzigd wordt zal de computer een andere sequentie van instructies uitvoeren en bijgevolg een andere taak uitvoeren. Het schrijven van deze sequentie van instructies in een taal die voor een mens leesbaar is, maar, mits een aantal tussenstappen, ook door een computer uitvoerbaar is noemt men ***programmeren***.

Software en hardware

In de definitie van een moderne computer die hierboven gegeven werd, staat de mogelijkheid om verschillende programma's of *software* uit te voeren op een centrale plaats. De computer wordt in deze definitie beschreven op basis van zijn functionaliteiten (wat het toestel moet kunnen). Om deze functionaliteiten te voorzien is er *hardware* nodig. De hardware is de verzameling van alle fysieke onderdelen van een computer waaronder de processor, het moederbord, het RAM-geheugen, de harde schijf (HDD) of solid state drive (SSD), het scherm, het toetsenbord, enz. De belangrijkste onderdelen zijn elektronische onderdelen en zijn opgebouwd uit geleiders, transistoren, condensatoren, enz., die vrij complexe schakelingen vormen.

1.2 Onderdelen van een digitale computer

Een digitale computer is een elektronisch toestel dat bestaat uit elektronische basiscomponenten. Het blokschema van een computer in Figuur 1.1 toont welke deze onderdelen zijn en hoe ze samenhangen.



Figuur 1.1: Blokkenschema van een computer.

De input Unit

De input unit bestaat uit input devices zoals een muis, toetsenbord, scanner, microfoon, enz. die gebruikt worden om data of instructies in een computer in te voeren. Zoals andere elektronische toestellen, zal een computer ruwe data en instructies aanvaarden in een binair formaat, deze verwerken en het resultaat als output teruggeven. De input unit is het medium dat ervoor zorgt dat een gebruiker data op een georganiseerde manier kan invoeren in een computer.

De central processing unit (CPU)

CPU of **C**entral **P**rocessing **U**nit is het centrum van de computer. Het is een elektronisch appa-

raat dat alle bewerkingen (zoals rekenkundige en logische bewerkingen) van de computer uitvoert. De CPU is ook verantwoordelijk voor het afhandelen van de operaties van verschillende andere eenheden. De belangrijkste onderdelen zijn:

- *Control Unit.* Deze unit coördineert alle operaties en activiteiten van een computer en bepaalt wat wanneer gebeurt in en welke volgorde. Ze is ook verantwoordelijk voor de controle van de input/output, het geheugen en andere devices die verbonden zijn met de CPU. Ze ontvangt instructies van het geheugen, decodeert en interpreteert ze en zet ze om in een sequentie van operaties (die dan vaak door de ALU zullen worden uitgevoerd).
- *Arithmetic en Logic Unit (ALU).* De data die worden ingevoerd via de input devices wordt bewaard in het main memory. De ALU voert vervolgens rekenkundige en logische operaties uit op deze data (het ophalen van deze data alsook welke operaties er worden op uitgevoerd wordt gecooördineerd door de control unit). De *arithmetic unit* voert eenvoudige rekenkundige bewerkingen uit zoals de optelling, aftrekking deling en vermenigvuldiging. Voor elk van deze bewerkingen bevat ze elektronische schakelingen. Een voorbeeld van zo'n schakeling is een adder, die wanneer ze de binaire voorstelling van twee gehele getallen als input krijgt, ze de binaire voorstelling van de som van deze getallen als uitput teruggeeft[1]. De *logic unit* voert logische operaties uit zoals controleren op gelijkheid, of volgorde (groter dan, kleiner dan) en de logische AND en OR. De ALU zal voor deze operaties data uit het geheugen gebruiken. Nadat de operaties op de data werden uitgevoerd, wordt het resultaat bewaard in het geheugen.
- De ALU staat in nauw contact met het *main memory* (dit is geheugen dat rechtstreeks aanspreekbaar is). Dit geheugen is tijdelijk van aard. De data die hierin bewaard worden gaan verloren als de spanning wegvalt. Een deel van dit geheugen, waaronder het cache geheugen van de CPU, is fysiek aanwezig in de CPU. Dit cache geheugen is zeer beperkt in grootte en minder belangrijk voor een beginnende programmeur. Het Random Access Memory (RAM) dat zich buiten de CPU bevindt en er dus geen deel vanuit maakt, is belangrijker en wordt hierna besproken.

De Memory Unit

De memory unit is een essentieel onderdeel van elke computer. We onderscheiden:

- *Main memory (primair geheugen).* Het Random Access Memory (RAM) is het belangrijkste voorbeeld van primair geheugen, tijdelijk geheugen dat rechtstreeks aanspreekbaar is door de CPU. Het word gebruikt voor de tijdelijke opslag van data en programma's alsook (eventueel tussentijdse) resultaten van berekeningen.
- *Secondary memory (secundair geheugen).* Het secundaire geheugen kan niet rechtstreeks worden aangesproken door de CPU. Data die wordt opgehaald uit secundair geheugen wordt eerst ingeladen in het RAM en dan pas verder verwerkt door de CPU. Dit secundaire geheugen is (meestal) wel permanent. De hard disk en SSD zijn secundair geheugen.

De Output Unit

[1] `https://nl.wikipedia.org/wiki/Adder_(elektronica)`

De output unit bestaat uit een scherm, printer, luidsprekers, etc. en zorgt ervoor dat het resultaat van de operaties die worden uitgevoerd op de data worden voorgesteld op een door een mens waarneembare en interpreteerbare manier.

Meer over geheugen

Geheugen is in verschillende vormen aanwezig in een computer. De harde schijf of SSD, het RAM, de CPU-cache of een USB-stick zijn voorbeelden van geheugendragers. Elk van deze geheugendragers bewaren data in een binair formaat. Dat wil zeggen dat ze bestaan uit een (vaak groot) aantal geheugencellen waarbij elke cel één van twee toestanden kan aannemen. Het zijn dus binaire geheugencellen. Vaak worden deze toestanden genoteerd als 0 of 1. De fysische implementatie van deze toestanden hangt af van het type drager. In het RAM geheugen van een moderne laptop bevinden zich miljarden condensatoren (de elementaire geheugencellen) die elk in staat zijn om een (zeer kleine) lading te dragen. De aan- of afwezigheid van deze lading bepaalt de toestand van de geheugencel. De geheugencellen van een harde schijf maken daarentegen gebruik van magnetische velden (magnetisch noorden N en zuiden Z) als toestanden. Via een kleine electromagneet kunnen deze gelezen worden en omgezet worden in (binaire) elektrische signalen.

Een elementaire geheugencel kan één **bit** aan informatie bewaren (een 0 of 1). Om historische redenen worden deze bits gegroepeerd per 8. Deze 8 bits vormen samen 1 **byte**. Naar analogie met het SI-stelsel spreekt men bovendien de **kilobyte** (1000 bytes), de **megabyte** (1000 000 bytes), de **gigabyte** (1000 000 000 bytes), de **terrabyte** (1000 000 000 000 bytes), enz.

Een computer kan dus enkel omgaan met data die kunnen bewaard worden in binaire geheugencellen, m.a.w. door gebruik te maken van een reeks van de toestanden 0 en 1. In Hoofdstuk 2 van deze syllabus wordt uitvoerig besproken hoe (decimale) getallen binair kunnen voorgesteld worden en welke impact dit heeft op de berekeningen en het geheugen. Later zien we ook hoe tekst binair kan voorgesteld worden.

1.3 De programmeertaal Python

Python is een *programmeertaal*, dit is een formele taal waarin men opdrachten kan schrijven die door een computer kunnen worden uitgevoerd. Net als een natuurlijke taal (zoals bv. het Nederlands) heeft een programmeertaal een woordenschat en een aantal grammaticale regels, waarnaar men vaak verwijst als de syntax(is) van de taal. Iemand die de programmeertaal Python aanleert, moet dus onder andere leren om deze regels correct toe te passen. Een collectie, door een mens leesbare, instructies die men in een programmeertaal neerschrijft noemt noemt men *broncode*. Het onderstaande codefragment is een voorbeeld van broncode die geschreven werd in Python 3.

Fragment 1.1: `voorbeeldSom.py`

```
1 a = 5.0
2 b = 6.0
3 c = a + b
4 print(c)
```

Wanneer deze broncode wordt uitgevoerd[2], zal de waarde 11.0 op het scherm verschijnen.

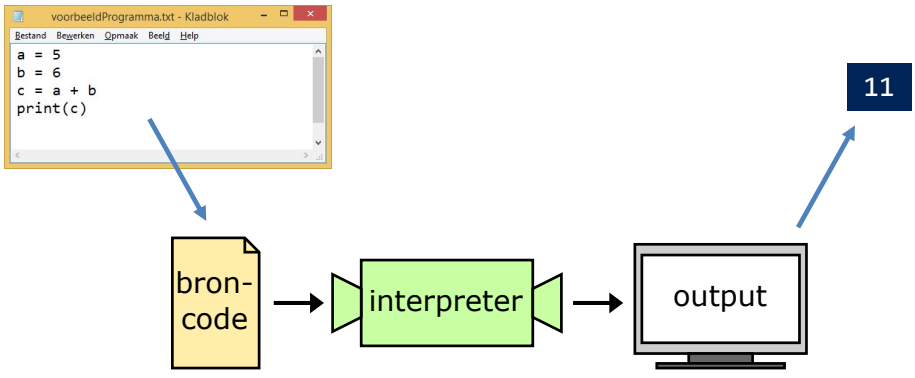
Waarom Python?

De keuze voor een (eerste) programmeertaal hangt af van meerdere factoren. Belangrijke factoren zijn de toepassing(en) waarvoor de taal zal gebruikt worden, de leercurve van de taal (hoe moeilijk een taal aan te leren is), de (voor)kennis van de programmeur. Zonder in detail te gaan worden hieronder een aantal redenen aangehaald die een keuze voor Python motiveren.

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed. (overgenomen van `https://www.python.org/`*)*

1.4 De Python interpreter

Zoals Codefragment 1.1 illustreert, is broncode platte tekst die in een tekstbestand kan worden opgeslagen op de harde schrijf van een computer. Om aan te duiden dat zo'n file Python broncode bevat, gebruikt men vaak de extensie `.py` in de bestandsnaam van deze bestanden (zoals `voorbeeldSom.py` in het voorbeeld). Indien men de instructies in een broncodebestand effectief door de computer wil laten uitvoeren, dan moet men gebruikmaken van een interpreter. Om deze reden noemt men Python een geïnterpreteerde taal (het interpreteren en uitvoeren van de broncode gebeurt in één stap, zoals geïllustreerd in Fig. 1.2), in tegenstelling tot gecompileerde talen waarbij broncode eerst wordt gecompileerd naar machine code en daarna pas wordt uitgevoerd.



Figuur 1.2: Een tekstbestand met de broncode (uit Fragment 1.1) wordt door de Interpreter geïnterpreteerd en uitgevoerd, waarna de output (11) op het scherm verschijnt.

De *Python 3 interpreter* is dus een computerprogramma dat broncode, die geschreven werd in de programmeertaal Python 3, uitvoert. Om gebruik te kunnen maken van Python moet deze interpreter geïnstalleerd zijn op het systeem dat men wil gebruiken. In deze cursus gebruiken we de

[2]We zullen verder zien hoe dit gebeurt.

CPython interpreter (dit is de meest populaire interpreter en werd geschreven in de programmeertaal C, vandaar de letter C in de titel). De meest recente officiële versie van deze interpreter kan teruggevonden worden op `https://www.python.org/downloads/`.

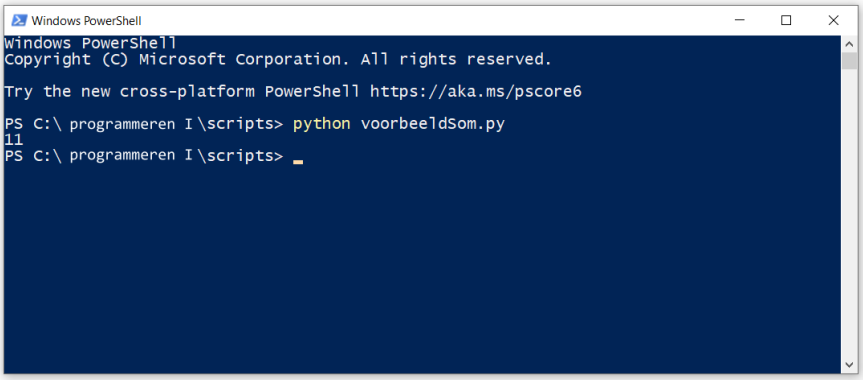
1.5 Programmeren in de praktijk

1.5.1 Programmeren in een shell omgeving

Python broncode kan je schrijven en aanpassen in een eenvoudige teksteditor zoals bijvoorbeeld *Notepad* (Kladblok) en bewaren als een tekstbestand (met de extensie `.py`). Vervolgens kan je deze broncode uitvoeren m.b.v. een *shell*. Dit is een interactief computerprogramma waarmee de gebruiker rechtstreeks opdrachten kan geven aan het besturingssysteem van de computer. Er bestaan voor verschillende besturingssystemen (Microsoft Windows, Apple macOS, Unix…) verschillende *shell* programma's, elk met hun eigen syntax. Echter, voor het uitvoeren van een bestand met Python broncode, kan meestal gebruik gemaakt worden van onderstaand commando:

```
python <bestandsnaam>.py
```

Dit wordt in Figuur 1.3 geïllustreerd voor het bestand `voorbeeldSom.py` in *Windows PowerShell*.

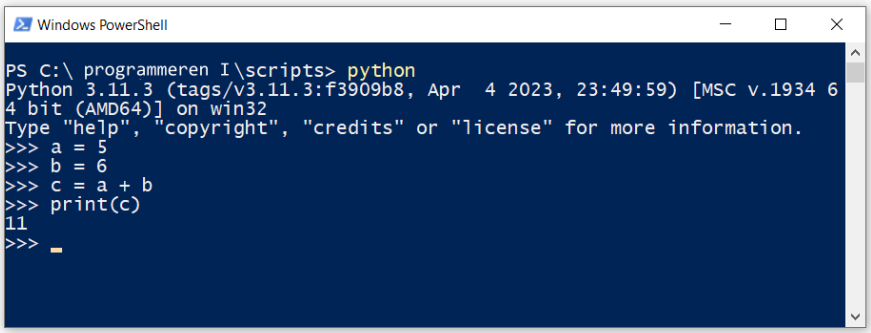


Figuur 1.3: Uitvoering van fragment 1.1 als Python script in een Windows Powershell.

Wanneer de Python Interpreter in de shell wordt opgeroepen zonder een bestand met broncode, wordt een **interactieve sessie** gestart. In deze sessie kunnen Python instructies een voor een worden uitgevoerd, door ze in te typen na de *prompt*-tekens `>>>` en vervolgens op *Enter* te drukken (Figuur 1.4). Daarbij worden alle variabelen[3] bijgehouden gedurende de sessie. Hierdoor kunnen afzonderlijke delen van een programma worden getest en geanalyseerd. Om de interactieve sessie te sluiten, moeten we `exit()` ingeven en uitvoeren.

In de volgende hoofdstukken worden instructies vaak geïllustreerd d.m.v. een interactieve sessie. Dit zal in de codefragmenten worden aangegeven met de prompt-tekens `>>>`.

[3]Zie Hoofdstuk 3 voor meer details over variabelen



Figuur 1.4: De instructies uit Fragment 1.1 worden een voor een uitgevoerd in een interactieve sessie

1.5.2 Integrated Development Environments

De werkwijze uit sectie 1.5.1 kan gebruikt worden om eenvoudige programma's te ontwikkelen. Voor de ontwikkeling van meer uitgebreide programma's wordt vaak gebruikgemaakt van een *Integrated Development Environment (IDE)*. Dit is software die speciaal ontworpen werd om het schrijven en testen van broncode te vereenvoudigen. Zo beschikken de meeste IDE's over *auto-completion* waarbij suggesties worden gegeven terwijl je broncode schrijft. Daarnaast kan broncode in IDE's worden weergegeven met syntaxkleuring (*syntax highlighting*), waarbij bepaalde termen een andere kleur of lettertype krijgen op basis van hun betekenis in de programmeertaal. Dit maakt de broncode veel beter leesbaar.

Naast de hulp bij het schrijven van broncode, kunnen IDE's ook helpen bij het uitvoeren van de code. Zo wordt de interactieve sessie vaak verzorgd door pakketten met extra functionaliteiten t.o.v. standaard Python (zoals IPython) en uitgebreid met een *variable explorer*. Die laatste biedt een grafisch overzicht van de variabelen die zijn gedefinieerd in de huidige sessie[3]. Daarnaast bieden de meeste IDE's ook een *debugging* functionaliteit aan, waarbij een bestand met broncode op een interactieve manier wordt uitgevoerd om sneller fouten te kunnen lokaliseren.

Enkele voorbeelden van IDEs zijn:

- **Visual Studio Code**, met de Python extensie (zie `https://code.visualstudio.com/`)
- PyCharm (zie `https://www.jetbrains.com/pycharm/`)
- Spyder (zie `https://www.spyder-ide.org/`)

Elk van deze IDEs heeft een aantal voor- en nadelen maar de laatste jaren heeft VSCode zich ontpopt tot een van de meest gebruikte IDEs. In deze cursus kiezen we daarom voor deze software als de standaard IDE[4].

[4]De installatie- en configuratieprocedure wordt beschreven in het document "Handleiding Visual Studio Code met Python" op Ufora.

1.5.3 Programmeren en GenAI

Met de publieke release van ChatGPT in 2022 vonden *large language models* (LLMs), en meer algemeen generatieve artificiële intelligentie of GenAI, ingang bij het grotere publiek. Deze taalmodellen (of varianten ervan) zijn ook in staat om ondermeer broncode te genereren o.b.v. een *user-prompt* in een natuurlijke taal, deels afwerkte code aan te vullen (*advanced code completion*) of te helpen bij het opsporen van fouten in broncode. Bovendien bestaan er voor veel populaire IDEs *plug-ins* die het gebruik van GenAI naadloos integreren in je workflow en die reeds geschreven code meenemen in de context van het achterliggende LLM om zo nog meer specifieke aanbevelingen te kunnen doen[5]. De opportuniteiten en aandachtspunten verbonden aan van deze tools zijn gelijkaardig aan die in andere toepassingsdomeinen. Hierna volgt een kort overzicht met de nadruk op programmeerconcepten[6].

Opportuniteiten bij het gebruik van de hiervoor vernoemde tools o.b.v. GenAI door (beginnende) programmeurs:

- GenAI-gebaseerde tools kunnen **beginnende programmeurs** ondersteunen bij het begrijpen van broncode. Men kan aan een GenAI tool vragen om de functie van gegeven broncode regel-per-regel woordelijk toe te lichten.
- GenAI-gebaseerde tools kunnen kunnen **beginnende programmeurs** nieuwe programmeerconcepten aanleren door bijvoorbeeld alternatieve implementaties voor te stellen die meer efficiënt, leesbaar etc. zijn. De beginnende programmeer kan de eigen programmeerstijl daarmee verbeteren.
- GenAI-gebaseerde tools kunnen kunnen **beginnende programmeurs** indicaties geven i.v.m. de aanwezigheid van mogelijke bugs of randgevallen (*edge-cases*) waarbij een programma zou kunnen crashen (*robustness testing*), etc. Het kan zo de finale kwaliteit van het programma verhogen en de programmeur leren attent te zijn voor onverwachte scenario's waarmee een programma moet kunnen omgaan.
- GenAI-gebaseerde tools kunnen kunnen **alle programmeurs** helpen bij het versnellen van minder creatieve, en vaak repetitieve, taken. Een voorbeeld is het genereren van documentatie. Om ervoor te zorgen dat broncode ook bruikbaar is voor anderen (maar op langere termijn ook voor de ontwikkelaar van de code) moet men deze documenteren. Een proces dat vergelijkbaar is met het schrijven van een handleiding. Dit is een vrij repetitief en tijdrovend proces waarbij GenAI tools kunnen helpen door een groot deel van deze documentatie te schrijven. De programmeur hoeft vaak enkel een eenvoudige check uit te voeren op de correctheid van deze documentatie.
- GenAI-gebaseerde tools kunnen kunnen **alle kritische programmeurs** helpen bij het oplossen van problemen die nieuw zijn voor hen. Hoewel het probleemoplossen vermogen van AI systemen door middel van logisch redeneren beperkt is, zijn ze getraind op zeer grote

[5]Tijdens de PC-labs wordt het gebruik van een populaire GenAI plug-in in VSCode kort getoond/aangeleerd

[6]Merk op dat de invloed van GenAI op programmeren (zowel het aanleren als de kwaliteit van broncode in een professionele context) nog niet zeer veel gekend is.

verzamelingen van broncode die wel logische samenhangend is. Veel problemen die voor een programmeur nieuw zijn, werden, vaak in een andere context, reeds opgelost door anderen. GenAI tools maken gebruik van associatie en kunnen patronen in een probleemstelling trachten te associëren met bestaande algoritmen en stukken broncode waarmee het in contact kwam bij het trainen. Zo kunnen vrij complexe algoritmen worden voorgesteld die hun oorsprong vinden in bestaande codebases die mogelijks niet bekend zijn bij de (beginnende) programmeur.

Aandachtspunten bij het gebruik van de hiervoor vernoemde tools o.b.v. GenAI door (beginnende) programmeurs:

- Code die voorgesteld wordt door GenAI is niet foutloos. Voor eenvoudige problemen (type problemen die vaak veelvuldig voorkomen in het materiaal waarop deze GenAI systemen werden getraind) is de kans op fouten eerder klein. Voor problemen die meer complex worden of meer ingekleed zijn binnen een toepassing treden vaker fouten op. Let daarbij op een *halo effect*, waarbij je als programmeur te veel vertrouwen gaat stellen in een GenAI tool omwille van positieve ervaringen in het verleden. Ook bij GenAI gebaseerde code blijft code testing (actief testen of je code in alle mogelijke omstandigheden correcte output geeft) zeer belangrijk!
- Vooral beginnende programmeurs moeten waakzaam zijn voor de nadelige impact op het leerproces. Programmeervaardigheden worden opgedaan door herhaald oefenen. GenAI kan dit oefenproces ondersteunen (onder meer door feedback op maat te voorzien), maar onbedachtzaam gebruik ervan kan ook leerkansen wegnemen.
- Over de invloed van gebruik van GenAI door (professionele) programmeurs op (de kwaliteit van) grote code bases is nog geen consensus. Zo wordt vaak verwezen naar de impact op zogenaamde *code vulnerabilities* (stukken in broncode die vatbaar zijn voor misbruik of kunnen uitgebuit worden door hackers), die kunnen worden geïntroduceerd, maar tevens worden gedetecteerd door GenAI tools. De invloed is waarschijnlijk erg genuanceerd.
- Wees je er, zeker als academisch opgeleide, van bewust dat er blijvende ethische en juridische kwesties zijn rond het gebruik van GenAI. Denk bijvoorbeeld op de impact op het milieu, het auteursrecht of het onbedoeld doorgeven van gevoelige informatie (onder meer aan de AI provider).

2

Binaire voorstelling van getallen en tekst

In het voorgaande hoofdstuk kwam reeds aan bod dat digitale computers data bewaren en verwerken in een binaire vorm. Belangrijke componenten zoals het RAM en de CPU werken immers met binaire geheugencellen om de data te bewaren en bevatten schakelingen die deze binaire voorstellingen verwerken. In dit hoofdstuk bekijken we hoe getallen en tekst binair kunnen worden voorgesteld. We houden daarbij steeds het volgende in het achterhoofd:

- Getallen kunnen op verschillende manieren binair worden voorgesteld. Bepaalde voorstellingen zijn meer flexibel dan andere, vragen meer of minder geheugencapaciteit en kunnen al dan niet aanleiding geven tot benaderingsfouten.
- Het is in deze cursus voornamelijk belangrijk dat we leren kiezen voor een geschikte voorstelling en leren wat de mogelijke consequenties zijn van deze keuze.

Appendix A bij deze cursus (afzonderlijke PDF) bevat een uitgebreide, technische beschrijving van de verschillende manieren waarop numerieke informatie binair kan voorgesteld worden. Deze technische behandeling gaat echter verder dan wat een beginnend programmeur hoeft te weten. Wat hierna volgt zijn de belangrijkste inzichten die we nodig hebben als beginnend programmeur.

2.1 Talstelsels

Een talstelsel is een systeem om getallen voor te stellen. In wat volgt bespreken we het **decimaal**, het **binair** en het **hexadecimaal** talstelsel. Beide talstelsels zijn voorbeelden van *positiestelsels* omdat de positie van een cijfer in een getal de bijdrage van dat cijfer aan de waarde van het getal bepaalt.

2.1.1 Decimaal talstelsel

We gebruiken voor het tellen en rekenen in het dagelijkse leven doorgaans het **decimaal** of tiendelig **talstelsel**. Een getal in het decimaal talstelsel wordt voorgesteld door gebruik te maken van **tien**

symbolen: de cijfers $0, 1, 2, \ldots, 9$ en heeft als **basis** (of grondtal) 10. Elk geheel getal kan ontbonden worden als een veelterm in machten van 10, bijvoorbeeld:

$$\mathbf{2\,023} = \mathbf{2} \times 10^3 + \mathbf{0} \times 10^2 + \mathbf{2} \times 10^1 + \mathbf{3} \times 10^0 \,.$$

Elk cijfer in 2023 wordt een **coëfficiënt** in deze veelterm. Deze ontbinding expliciteert de waarde die we toekennen aan elk van de cijfers in het getal 2023.

Om een getal in dit talstelsel te bewaren in (computer)geheugen heeft dit geheugencellen nodig die elk tien verschillende toestanden kunnen aannemen: één toestand voor elk mogelijk cijfer. Denk bijvoorbeeld aan de cijferwieltjes van een cijferslot die elk één van tien mogelijke toestanden kunnen aannemen. In digitale computers is dit niet het geval, waardoor dit talstelsel niet geschikt is om getallen voor te stellen in het geheugen van digitale computers.

2.1.2 Binair talstelsel

Het **binair** of **tweedelig talstelsel** maakt gebruikt van **twee symbolen**: de cijfers 0 en 1, en heeft als **basis** 2. Men kan *elk* geheel getal *op een unieke manier* ontbinden als een veelterm in machten van het grondtal 2, met coëfficiënten die **enkel** de waarden **0 of 1** aannemen, bijvoorbeeld:

$$13 = \mathbf{1} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{1} \times 2^0 = \mathbf{1101}_2$$

Het resultaat 1101_2 is een binair getal. Om verwarring te vermijden tussen het decimale getal duizend honderdeen (1101) en het binaire getal 1101_2 wordt het **subscript 2** gebruikt. Om expliciet aan te geven dat het tiendelig talstelsel wordt gebruikt, kan het subscript 10 gebruikt worden, maar dat wordt vaak weggelaten.

Dit talstelsel laat toe om getallen uit te drukken a.d.h.v. rijen van twee symbolen, wat toelaat om deze voorstelling te gebruiken in digitale computers.

2.1.3 Hexadecimaal talstelsel

Bij het bestuderen van de interne voorstelling van variabelen in Python zullen we ook met getallen in het **hexadecimaal** of **zestiendelig** talstelsel in aanraking komen. Een getal in het hexadecimaal talstelsel wordt voorgesteld door gebruik te maken van:

- het grontal/basis: $b = 16$
- de **zestien** symbolen: $0, 1, 2, \ldots, 9, \mathrm{A}, \mathrm{B}, \ldots, \mathrm{F}$ waarvan de laatste symbolen geen cijfers zijn en als waarde $A = 10$, $B = 11$, ..., $F = 15$ hebben.

Elk geheel getal kan ontbonden worden in machten van 16, bijvoorbeeld:

$$2\,023 = \mathbf{7} \times 16^2 + \mathbf{14} \times 16^1 + \mathbf{7} \times 16^0 = 7\mathrm{E}7_{16}$$

Merk op dat het subscript 16 wijst op de hexadecimale voorstelling van een getal. **Soms** worden de **kleine letters** $\mathrm{a}, \mathrm{b}, \ldots, \mathrm{f}$ gebruikt i.p.v. de hoofdletters $\mathrm{A}, \mathrm{B}, \ldots, \mathrm{F}$.

Tab. 2.1 bevat de binaire en hexadecimale voorstelling van de getallen 0–15. Merk op dat de conversie van het binaire talstelsel naar het hexadecimale talstelsel zeer eenvoudig kan gebeuren door gebruik te maken van deze tabel op de volgende manier:

1. **groepeer** de bits in de binaire voorstelling per 4, en
2. **converteer** elke groep van 4 bits met Tab. 2.1 naar de binaire vorm.

Dit wordt geïllustreerd in het volgende voorbeeld:

$$0111\ 1110\ 0111_2 \rightarrow 7E7_{16}$$

Tabel 2.1: De eerste 16 getallen in het decimaal, binair en hexadecimaal talstelsel.

decimaal	binair	hexadecimaal	decimaal	binair	hexadecimaal
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

2.2 Gehele getallen in Python

2.2.1 `int`'s en `uint`'s

Gehele getallen (in het Engels *integers*) kunnen op verschillende manieren binair worden voorgesteld in het computergeheugen. Deze voorstellingswijzen zijn steeds gebaseerd op het binaire talstelsel, maar verschillen op twee vlakken:

- Het aantal bits dat ze gebruiken voor een voorstelling en daarmee samenhangend de grootte van de getallen die kunnen voorgesteld worden.
- Of ze al dan niet toelaten om negatieve getallen voor te stellen.

In deze sectie bespreken we de `int`'s en `uint`'s, twee eenvoudige voorstellingswijzen (vaak gegevenstypes of *datatypes* genoemd). Merk op dat, omwille van enkele beperkingen die eigen zijn aan deze gegevenstypes, ze **geen deel uitmaken van** de *Built-in data types* in Python. In veel programmeertalen is dat wel het geval en in **Hoofdstuk 8** zal blijken dat ze alomtegenwoordig zijn in `Numpy`, een belangrijke package[1] voor wetenschappelijk rekenwerk in Python.

[1]Een Python-package is een pakket van broncode (vaak Python code, maar kan ook code in andere talen bevatten en zelfs gecompileerde code) die de functionaliteiten van de Python Standard Library gevoelig kunnen uitbreiden. Deze packages worden bijkomend geïnstalleerd.

De `uint8`'s, `uint16`'s en `uint32`'s

Het binaire talstelsel maakt de opslag van gehele getallen in computergeheugen mogelijk. Het aantal bits dat men nodig heeft om een getal te bewaren hangt af van de grootte van dat getal. Het kleinste aantal bits dat men in de praktijk zal gebruiken is acht, dus 1 byte. In veel programmeertalen zal men kleine natuurlijke getallen opslaan d.m.v. de eerste 8 bits van de binaire voorstelling. Getallen die op deze manier in het geheugen bewaard worden maken gebruik van wat men het `uint8` gegevenstype noemt. `uint8` staat voluit voor *unsigned 8-bit integer*, waarbij *integer* de Engelse term is voor 'geheel'.

Het kleinste getal wordt gevormd door 00000000_2 en heeft waarde 0, het grootste getal is 11111111_2 en heeft waarde 255 ($= 2^8 - 1$). Analoog worden bij gegevenstype `uint16`, worden 16 bits gebruikt en kunnen getallen tussen 0 en 65535 ($= 2^{16} - 1$) bewaard worden. Tab. 2.2 geeft voor een aantal veel voorkomende datatypes het bereik van gehele getallen dat kan voorgesteld worden.

De gegevenstypes `int8`, `int16` en `int32`

De gegevenstypes `int8`, `int16` en `int32` laten ook de opslag van negatieve gehele getallen toe. Ze maken (vaak) gebruik van de 2-complement methode die ook toelaat om negatieve getallen te bewaren (zie Appendix A voor meer details). Getallen tussen -128 en 127 kunnen als `int8` bewaard worden. Wanneer we het bereik van een `int8` vergelijken met dat van een `uint8` wordt duidelijk dat de grootst mogelijke waarde die kan bewaard worden als `int8` (dit is 127) ongeveer de helft is van de grootst mogelijke waarde die een `uint8` kan aannemen (dit is 255). Beide gegevenstypes gebruiken echter 8 bits geheugen. Dit is gelijkaardig voor `int16` en `uint32`.

Tabel 2.2: Bereik van enkele datatypes voor gehele getallen.

datatype	minimumwaarde	maximumwaarde
`int8`	$-2^7 = -128$	$2^7 - 1 = 127$
`uint8`	0	$2^8 - 1 = 255$
`int16`	$-2^{15} = -32768$	$2^{15} - 1 = 32767$
`uint16`	0	$2^{16} - 1 = 65535$
`int32`	$-2^{31} = -2147483648$	$2^{31} - 1 = 2147483647$
`uint32`	0	$2^{32} - 1 = 4294967295$
`int64`	$-2^{63} = -9223372036854775808$	$2^{63} - 1 = 9223372036854775807$
`uint64`	0	$2^{64} - 1 = 18446744073709551615$

Overflow en underflow

Wanneer men een waarde wenst te bewaren die groter is dan de grootste waarde die wordt toegelaten door een gegevenstype, spreekt men van *overflow*. Veronderstel bijvoorbeeld dat men het getal 257 wil bewaren als `uint8` (waarvoor de maximumwaarde 255 is). Er zijn nu verschillende mogelijkheden:

- de interpreter werpt een foutmelding op,
- de waarde wordt afgeknot op de hoogst voorstelbare (in dit geval dus 255),

- in plaats van 257 wordt de rest r berekend na de deling van 257 door 256 en wordt r bewaard, in dit geval dus 1.

In Python worden `int`'s an `uint`'s en voornamelijk gebruikt binnen de module `numpy`, een module die ontworpen werd voor numeriek (reken)werk en die we uitgebreid behandelen in Hoofdstuk 8 in deze cursus. De module `numpy` maakt gebruik van de laatste optie, maar geeft vaak een waarschuwing dat dit mogelijks ongewenst is en dat dit bij updates van deze module in de toekomst mogelijks aanleiding zal gaan geven tot een foutmelding.

Underflow is analoog aan overflow, maar dan voor waarden die te klein zijn. Veronderstel bijvoorbeeld dat men het getal -2 wenst te bewaren als `uint8`. Ook hier zijn er verschillende opties:

- de interpreter werpt een foutmelding op,
- de waarde wordt afgeknot op de kleinst voorstelbare (in dit geval dus 0),
- in plaats van -2 wordt de rest r berekend na de deling van 2 door 256 en wordt $256 - r$ bewaard, in dit geval dus 254.

Bij underflow wordt in `numpy` voor de laatste optie gekozen.

2.2.2 De *bignum* voorstelling en Python *integers*

De *bignum* voorstelling is een alternatief voor `int8`, `int16`, `int32` en `int64` en kent geen begrenzing. In tegenstelling tot de voorgaande gegevenstypes, ligt het aantal bytes dat *bignum*'s gebruiken niet vast. Voor grote getallen zullen meer bytes gebruikt worden dan voor kleine getallen.

In Python worden gehele getallen zeer vaak als *bignum*'s bewaard. Dat wil zeggen dat de programmeur zich weinig zorgen hoeft te maken over het al dan niet passen van een waarde binnen het bereik. Een nadeel van *bignum*'s is hun groot geheugenverbruik. Zo zal men bij het bewaren van de waarde 13 als *bignum* in Python 28 bytes (of dus 224 bits) gebruiken. Waarom dit zo is, wordt uitgebreid beschreven in Appendix A. Belangijk om te weten is dat bij het uitvoeren van een instructie zoals de onderstaande, de aan `a` toegekende waarde zal bewaard worden als een *bignum*.

```
>>> a = 13
```

Deze *bignum* voorstelling is dan ook het *built-in* data type voor gehele getallen in Python. Men verwijst er in Python naar als het gegevenstype `integer` (of afgekort `int`).

Het geheugengebruik kan als volgt opgevraagd worden.

```
>>> import sys
>>> sys.getsizeof(13)
28
```

Het resultaat wordt uitgedrukt in bytes, dus $28 \times 8 = 224$ bits.

2.3 Rationale getallen in Python

De decimale en binaire talstelsels kunnen uitgebreid worden met **negatieve exponenten** om naast gehele ook rationale getallen te kunnen voorstellen. Beschouw als voorbeeld het getal 12.125:

$$\mathbf{12.125} = \mathbf{1} \times 10^1 + \mathbf{2} \times 10^0 + \underbrace{\mathbf{1} \times 10^{-1} + \mathbf{2} \times 10^{-2} + \mathbf{5} \times 10^{-3}}_{\text{negatieve exponenten}}$$

Het decimaalteken geeft daarbij de locatie van de eenheden aan. Merk op dat, door gebruik te maken van vermenigvuldiging met (gehele) machten van 10, men eenzelfde decimaal getal op meerdere manieren kan voorstellen. Zo geldt dat:

$$12.125 = 121.25 \times 10^{-1} = 1212.5 \times 10^{-2} = \mathbf{1.2125 \times 10^1}\,.$$

Men noemt $\mathbf{1.2125 \times 10^1}$ de genormaliseerde voorstelling.

Een **genormaliseerde (decimale) voorstelling** is van de vorm

$$d_1.d_2d_3d_4 \ldots \times 10^e\,,$$

waarin men $d_1.d_2d_3d_4 \ldots$ de mantisse noemt en e de exponent. Er geldt dat:

- d_1 een van nul verschillend cijfer (1–9),
- d_2, d_3, d_4, ...$\in \{0, 1, 2 \ldots 9\}$,
- e is een geheel getal,
- Het teken wordt aangegeven door + (vaak niet expliciet) of -.

Op **dezelfde manier** kan men het **binaire talstelsel** uitbreiden met negatieve exponenten:

$$\mathbf{12.125} = \mathbf{1} \times 2^3 + \mathbf{1} \times 2^2 + \mathbf{0} \times 2^1 + \mathbf{0} \times 2^0 + \underbrace{\mathbf{0} \times 2^{-1} + \mathbf{0} \times 2^{-2} + \mathbf{1} \times 2^{-3}}_{\text{negatieve exponenten}} = 1100.001_2$$

Ook hier kan men, door gebruik te maken van vermenigvuldiging met machten van 2, eenzelfde getal op meerdere manieren voorstellen. Zo geldt dat:

$$1100.001_2 = 110.0001_2 \times 2^1 = 110000.1_2 \times 2^{-2} = \mathbf{1.100001_2 \times 2^3}\,.$$

Men noemt $1.100001_2 \times 2^3$ de genormaliseerde voorstelling.

Een **genormaliseerde (binaire) voorstelling** is van de vorm

$$d_1.d_2d_3d_4 \ldots \times 2^e\,,$$

waarin men $d_1.d_2d_3d_4 \ldots$ de mantisse noemt en e de exponent. Er geldt dat:

- $d_1 = 1$,
- d_2, d_3, d_4, ...$\in \{0, 1\}$,
- e is een geheel getal,

- Het teken wordt aangegeven door + (vaak niet expliciet) of -.

Deze uitbreiding vormt de basis van de drijvendekommagetallen (Eng. *floating points*) die gebruikt worden om rationale getallen voor te stellen in het werkgeheugen.

2.3.1 Floats en de IEEE-754 standaard

De IEEE-754 standaard beschrijft hoe rationale getallen binair kunnen worden voorgesteld. Deze standaard zal de drie componenten van de genormaliseerde binare voorstelling (de mantisse, de exponent en het teken) binair coderen om ze samen voor te stellen in computergeheugen. Het *built-in* gegevenstype voor rationale getallen in Python is de *double-precision floating point* of kortweg `float` en volgt deze standaard. Deze standaard wordt uitgebreid beschreven in Appendix A. Praktisch is het belangrijk om te weten dat:

- Zowel zeer kleine als zeer grote rationale getallen kunnen bewaard worden als `double`. Voorbeelden zijn:

```
>>> a = 0.00065415
>>> a = 13.1815415
>>> a = 15875487.4564
>>> a = -44.0
```

  Elk van deze instructies zal aanleiding geven tot een getal dat bewaard wordt als een `float` in Python. Het bereik is zeer ruim, maar wel begrensd met ondergrens $\pm 2.23 \times 10^{-308}$ en bovengrens $\pm 1.80 \times 10^{308}$.

- Veel rationale getallen kunnen slechts bewaard worden mits een benaderingsfout in acht te nemen. Deze benaderingsfout neemt doorgaans toe met de grootte van het bewaarde getal.

- Een *double-precision* `float` gebruikt een vaste hoeveelheid geheugen: 64 bits, maar Python heeft een extra overhead die zorgt voor een totaal geheugengebruik van 192 bits per `float`, zoals geïllustreerd wordt hieronder:

```
>>> import sys
>>> sys.getsizeof(0.00065415)
24
```

  Het resultaat wordt uitgedrukt in bytes, dus $24 \times 8 = 192$ bits.

Om meer inzicht te krijgen in de benaderingsfout beschouwen we het getal 0.1 (één tiende). De binaire voorstelling van 0.1 kan als volgt opgebouwd worden:

$$0.1 \approx 0 \times 2^0 + 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 1 \times 2^{-4} + 0 \times 2^{-5} + ...$$

Enkel de coëfficiënten noteren geeft, met extra precisie:

$$0.0001001001001001001001001001001001 ...{}_2$$

de ... geven aan dat 0.1 niet exact voorgesteld kan worden door een eindig aantal bits. Omdat een `float` steeds gebruikmaakt van 64 bits, kan het getal 0.1 dus niet exact worden voorgesteld. Bijgevolg ontstaat een **benaderingsfout**. Beschouw nu een rationaal getal x, met genormaliseerde (binaire) voorstelling $1.d_2d_3d_4 \ldots \times 2^e$ en de voorstelling ervan als `float` $\widehat{x}$. We onderscheiden de volgende twee fouten:

$$\textbf{absolute fout} = |x - \widehat{x}|\,, \qquad \textbf{relatieve fout} = \left|\frac{x - \widehat{x}}{x}\right|$$

Voor elke $x \in [\pm 2.23 \times 10^{-308}$ tot $\pm 1.80 \times 10^{308}]$ geldt dat:

$$\left|\frac{x - \widehat{x}}{x}\right| \leq \frac{1}{2}\frac{2^{-(52-e)}}{2^e} = 2^{-53} \approx 1.11 \times 10^{-16}\,.$$

De relatieve fout is dus begrensd. Voor de absolute fout zal deze grens toenemen naarmate x toeneemt:

$$|x - \widehat{x}| \leq |x| \times 1.11 \times 10^{-16}\,.$$

Deze benaderingsfouten verklaren het volgende gedrag:

```
>>> 1.1 + 2.2
3.3000000000000003
```

In het resultaat van deze som is het 16^{e} cijfer na het decimaalteken verschillend van 0. De bovenstaande ongelijkheid garandeert dat het resultaat maar accuraat is tot de 15^{e} positie na het decimaalteken.

```
>>> 1.0 + (1.0*10**20) - (1.0*10**20)
0.0
```

In de bovenstaande instructie zou bij afwezigheid van benaderingsfouten het resultaat gelijk zijn aan 1.0. De benaderingsfout van 1.0×10^{20} is echter groter dan 1.0. Python zal deze instructie van links naar rechts evalueren en bijgevolg zal de waarde 1.0 worden geabsorbeerd in de benaderingsfout van het tussentijdse resultaat `1.0 + (1.0*10**20)` en als het ware verdwijnen. Dit is een voorbeeld van wat men ***catastrophic cancellation*** noemt.

2.3.2 Complexe getallen

Python bevat drie built-in numerieke gegevenstypes: `integer`, `float` en `complex`. Dit laatste gegevenstype laat toe om complexe getallen voor te stellen in Python. Dit gegevenstype wordt in deze cursus niet verder gebruikt. Appendix A bevat meer informatie over dit gegevenstype.

2.4 Tekst in Python

Moderne programma's moeten vaak grote hoeveelheden tekstuele data kunnen verwerken. We denken daarbij bijvoorbeeld aan programma's zoals spamfilters die e-mail verwerken en beslissen of die al dan niet bestempeld kunnen worden als spam. Een ander voorbeeld zijn DNA-sequentie analyses. Hierbij beschouwen we tekstuele data in de brede zin van het woord en dus elke vorm van data die geschreven kan worden als een sequentie van karakters. Zo'n sequentie van karakters noemt men in een programmeercontext een **string**. Deze karakters kunnen letters zijn, leestekens, maar bijvoorbeeld ook emoticons.

Een sequentie van karakters kan men in computergeheugen voorstellen door elk karakter afzonderlijk binair te gaan coderen en vervolgens deze binaire coderingen te concateneren. Bij deze codering wordt gebruikgemaakt van een *tekencodering*, als het ware een grote tabel die bepaalt hoe een karakter binair wordt voorgesteld[2].

2.4.1 ASCII

ASCII (American Standard Code for Information Interchange) is een 7-bits tekencodering die toelaat om (Latijnse) letters, (Arabische) cijfers en leestekens (in totaal 94 zichtbare tekens) en de spatie binair voor te stellen. Daarnaast zijn ook 33 stuurcodes, waaronder *tab* en *Carriage return* (of nieuwe lijn karakter) die vaak in tekst voorkomen opgenomen in ASCII. Andere stuurcodes zoals de *escape code*, *shift in*, etc. maken deel uit van ASCII, maar zijn minder belangrijk voor beginnende programmeurs. Tab. 2.3 en Tab. 2.4 tonen een deel van de ASCII-tabel.

Merk op dat de binaire codering bestaat uit 8 bits (1 byte) maar dat de eerste bit steeds waarde 0 heeft. Omdat slechts 7 bits effectief gebruikt worden, kunnen slechts $2^7 = 128$ karakters gecodeerd worden (in plaats van $2^8 = 256$ bij 8 bits). Computersystemen verwerken data steeds per 8 bits. Dit wil zeggen dat 1 bit niet benut wordt met een 'verlies' van 128 karakters tot gevolg. Vaak gebruikt men dan ook *extended-ASCII* tekencoderingen die de extra bit gebruiken om extra karakters te coderen. Binnen deze uitgebreide coderingen wordt *ANSI Latin 1* zeer frequent gebruikt. Tekens zoals Â of μ maken deel uit van *ANSI Latin 1*, maar niet van ASCII.

2.4.2 Unicode

De Unicode standaard `https://www.unicode.org` is een specificatie die (bijna) elk karakter dat gebruikt wordt in menselijke taal toekent aan een unieke code (een Unicode *code point*). Deze code maakt gebruik van de symbolen 0–F (de symbolen gebruikt in het hexadecimale talstelsel), en is dus **geen** binaire voorstelling.

Het onderstaande blok toont enkele van deze karakters en de bijhorende *Unicode code points*.

[2]Strikt genomen is er nog een onderscheid tussen een tekencodering en een karaktercodeerschema, maar deze nuance valt buiten het bestek van deze Syllabus.

Decimaal	Hex	Bin	Waarde	Betekenis
000	00	00000000	NUL	Null character
001	01	00000001	SOH	Start of header
002	02	00000010	STX	Start of text
003	03	00000011	ETX	End of text
004	04	00000100	EOT	End of transmission
005	05	00000101	ENQ	Enquiry
⋮	⋮	⋮	⋮	⋮
029	1D	00011101	GS	Group separator
030	1E	00011110	RS	Record separator
031	1F	00011111	US	Unit separator
032	20	00100000	SP	Space
033	21	00100001	!	Exclamation mark
034	22	00100010	”	Double quote
035	23	00100011	#	Number sign
036	24	00100100	$	Dollar sign
037	25	00100101	%	Percent
038	26	00100110	&	Ampersand
039	27	00100111	’	Single quote
040	28	00101000	(	Left opening parenthesis
041	29	00101001	)	Right closing parenthesis
042	2A	00101010	*	Asterisk
043	2B	00101011	+	Plus
044	2C	00101100	,	Comma
045	2D	00101101	-	Minus or dash
046	2E	00101110	.	Dot
047	2F	00101111	/	Forward slash

Tabel 2.3: Deel van de ASCII-tabel. Bovenaan bevinden zich enkele stuurcodes gevolgd door leestekens.

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
...
2167    'Ⅷ'; ROMAN NUMERAL EIGHT
2168    'Ⅸ'; ROMAN NUMERAL NINE
...
265E    '♞'; BLACK CHESS KNIGHT
265F    '♟'; BLACK CHESS PAWN
...
1F600   '😀'; GRINNING FACE
1F609   '😉'; WINKING FACE
...
```

Merk ook de gelijkenis op met de ASCII-tabel. Het karakter `a` heeft als unicode `0061`. Als we deze

code karakter per karakter converteren naar het binair talstelsel met Tab.2.2 bekomen we

$$0061 \rightarrow 0000\ 0000\ 0110\ 0001\,.$$

De laatste 7 karakters vormen de ASCII-code van `a`. Hetzelfde proces kan echter toegepast worden op elk *Unicode code point* om een binaire voorstelling te bekomen. Dit proces noemen we een *encoding*: een manier om karakters binair voor te stellen.

Deze encoding heeft echter enkele nadelen:

- Veel voorkomende karakters zoals `a`, `b`, etc. nemen zeer veel geheugenruimte in die wordt opgevuld met 0'en.
- Ze is **geen uitbreiding** van ASCII en dus niet compatibel met (oude) ASCII-systemen.
- Een sequentie van 8 opeenvolgende 0'en wordt door veel systemen gezien als een *Null character* en meteen ook het einde van een string.

2.4.3 UTF-8

Wat is UTF-8?

UTF-8 staat voor **U**nicode **T**ransformation **F**ormat - 8-bit. Het is een codering waarmee elk teken in de Unicode-standaard kan worden weergegeven met behulp van 1 tot 4 bytes. De 8 verwijst naar het feit dat de codering gebruikmaakt van 8-bits eenheden (bytes) als basis voor het coderen van tekens. Het werd ontwikkeld om compatibel te blijven met ASCII en tegelijkertijd de mogelijkheid te bieden om alle Unicode-karakters te coderen zonder onnodig veel geheugen te gebruiken, terwijl compatibiliteit met bestaande systemen en efficiëntie behouden blijft.

Ontstaan

UTF-8 is ontstaan in de vroege jaren '90 als een oplossing voor de problemen met bestaande tekencoderingen zoals ASCII (slechts 128 tekens) en andere lokale coderingen (zoals bv. Shift-JIS voor Japans), die beperkt waren in het aantal tekens dat ze konden coderen. Bovendien waren deze coderingen onderling niet compatibel.

In 1992 werd UTF-8 ontworpen door Ken Thompson en Rob Pike. Hun doel was om een tekencodering te ontwikkelen die:

- compatibel was met bestaande ASCII-gebaseerde systemen,
- efficiënt omging met geheugenruimte door alleen meer bytes te gebruiken voor tekens die dat vereisten (variabele lengte codering), en
- in staat was om alle mogelijke Unicode-karakters te coderen.

Werking

De bits in een UTF-8 bitstring worden per 8 (of dus 1 byte) verwerkt. UTF-8 is ontworpen om volledig compatibel te zijn met ASCII, wat betekent dat voor ASCII-tekens (0 tot 127) de code uit de ASCII tabel wordt overgenomen (1 byte per teken).

Voor tekens buiten ASCII (met een Unicode code point hoger dan 127) gebruikt UTF-8 meer dan 1 byte, we spreken van een multi-byte teken. De eerste byte van een multi-byte teken enkele bevat *lead bits* die aangeven hoeveel bytes de multi-byte codering van het karakter zal bevatten, gevolgd door de data bytes die het karakter coderen. De codering is de binaire voorstelling van het Unicode code point van het karakter. Het aantal bytes dat wordt gebruikt voor de codering hangt af van het Unicode-teken:

1 byte: ASCII-tekens (Unicode 0 - 127)

2 bytes: veel voorkomende accenten en symbolen (Unicode 128 - 2047).

3 bytes: niet-Latijnse scripts zoals Chinees, Arabisch, enz. (Unicode 2048 - 65535).

4 bytes: zeer zeldzame en historische tekens (Unicode 65536 - 1114111).

Elke byte heeft een of meerdere *lead bits*, elk met een bijhorende betekenis, die de volgende vorm kunnen aannemen:

1. De **byte begint met** `0`: het karakter is een single-byte karakter. De overige 7 bits volgen de ASCII code-tabel.
2. De **byte begint met** `110`, `1110`, `11110`, enz.: dit is de start van een multi-byte teken. Het aantal 1-en geeft aan uit hoeveel bytes het teken bestaat. In de voorbeelden respectievelijk 2, 3 of 4 bytes.
3. De **byte begint met** `10`: de byte is een **vervolgbyte**. Deze byte moet samen met de voorgaande (en mogelijks ook volgende) bytes samen geïnterpreteerd worden.

Een voorbeeld: het € teken

Een voorbeeld van een UTF-8-codering met 3 bytes is het € teken met Unicode code point `U+20AC`. De stappen om de UTF-8 codering te bekomen, zijn de volgende.

Stap 1: bepaling Unicode code point

Het euroteken heeft het Unicode code point: `U+20AC`.

Stap 2: binaire representatie van het Unicode code point

`U+20AC` in binaire vorm is: 0010 0000 1010 1100 (14 bits).

Stap 3: toepassen van UTF-8 codering

Omdat 14 bits buiten het bereik van 1-byte en 2-bytes karakters valt, gebruikt UTF-8 een 3-byte codering.

De binaire code voor een 3-byte UTF-8 karakter volgt het volgende patroon:

`1110xxxx 10xxxxxx 10xxxxxx`

De eerste 4 bits van het Unicode code point gaan in de eerste byte (op de plaatsen aangeduid met `xxxx`). De volgende 6 bits in de tweede byte en de laatste 6 bits in de derde byte.

We krijgen dan de volgende codering voor het € teken:

- eerste byte: `1110 0010` (`e2` in hexadecimaal),
- tweede byte: `1000 0010` (`82` in hexadecimaal), en
- derde byte: `1010 1100` (`ac` in hexadecimaal)

Stap 4: Resultaat

Het € teken wordt in UTF-8 bijgevolg als volgt gecodeerd: `E2 82 AC` (hexadecimaal) of 11100010 10000010 10101100 (binair).

Op het Computerphile Youtube kanaal kan je meer details vinden over hoe UTF-8 precies tot stand kwam en waarom het uitgegroeid is tot de dominante tekencodering (door Tom Scott):

`https://www.youtube.com/watch?v=MijmeoH9LT4`

2.4.4 Strings in Python

In Python is het built-in gegevenstype `string` (**`str`**) beschikbaar voor de verwerking van sequenties van karakters. In de Python syntax worden strings steeds omgeven door enkele `'` of dubbele `"` quotes. De onderstaande instructies genereren elk een string in het werkgeheugen. Python gebruikt daarbij de UTF-8 codering.

```
>>> a = "ok"
>>> b = "oke"
>>> c = "oké"
>>> d =  ""
```

Het gebruik van UTF-8 zorgt voor een moeilijker voorspelbaar geheugengebruik. Het volgende codefragment illustreert dit:

```
>>> import sys
>>> sys.getsizeof("")
49
>>> sys.getsizeof("ok")
51
>>> sys.getsizeof("oke")
52
>>> sys.getsizeof("oké")
76
```

De eerste instructie toont het geheugengebruik van een lege string `""`, wat aantoont dat de overhead voor strings 49 bytes is per string. Per ASCII-karakter wordt 1 byte extra gebruikt. Voor niet-ASCII karakters is het geheugengebruik veel groter. Hoe groot dit precies is hangt af van het karakter (en wordt bepaald door de UTF-8 codering). Voor het karakter `é` is dit 25 bytes.

Decimaal	Hex	Bin	Waarde	Betekenis
048	30	00110000	0	
049	31	00110001	1	
050	32	00110010	2	
051	33	00110011	3	
052	34	00110100	4	
053	35	00110101	5	
054	36	00110110	6	
055	37	00110111	7	
056	38	00111000	8	
057	39	00111001	9	
058	3A	00111010	:	Colon
059	3B	00111011	;	Semi-colon
060	3C	00111100	<	Less than sign
061	3D	00111101	=	Equal sign
062	3E	00111110	>	Greater than sign
063	3F	00111111	?	Question mark
064	40	01000000	@	AT symbol
065	41	01000001	A	
066	42	01000010	B	
067	43	01000011	C	
068	44	01000100	D	
069	45	01000101	E	
⋮	⋮	⋮	⋮	⋮
097	61	01100001	a	
098	62	01100010	b	
099	63	01100011	c	
100	64	01100100	d	
101	65	01100101	e	
102	66	01100110	f	
103	67	01100111	g	
104	68	01101000	h	
105	69	01101001	i	
106	6A	01101010	j	
107	6B	01101011	k	
108	6C	01101100	l	
109	6D	01101101	m	
110	6E	01101110	n	
111	6F	01101111	o	
112	70	01110000	p	
113	71	01110001	q	
114	72	01110010	r	
115	73	01110011	s	
116	74	01110100	t	
117	75	01110101	u	
118	76	01110110	v	
119	77	01110111	w	
120	78	01111000	x	
121	79	01111001	y	
122	7A	01111010	z	
127	7F	01111111	DEL	Delete

Tabel 2.4: Deel van de ASCII-tabel. De cijfers gevolgd door het alfabet (hoofdletters en kleine letters).

3

Python bouwstenen

In Hoofdstuk 1 werd een computerprogramma omschreven als *een gedetailleerde, stap-voor-stap omschreven, sequentie van instructies die ondubbelzinnig beschrijven wat een computer moet doen.* In dit hoofdstuk wordt beschreven hoe de meest eenvoudige instructies worden opgebouwd.

> Bepaalde onderdelen in dit hoofdstuk en volgende hoofdstukken worden voorafgegaan door een *side-bar*. Deze side-bar geeft aan dat deze concepten technisch uitdagender zijn en niet noodzakelijk voor beginnende programmeurs.

3.1 Objecten, gegevenstypes en variabelen

3.1.1 Numerieke en tekstuele gegevens: een voorbeeld

Als een startpunt hernemen we het codefragment uit de inleiding.

Fragment 3.1: `voorbeeldSom2.py`

```
1 a = 5
2 b = 6
3 c = a + b
4 print(c)
```

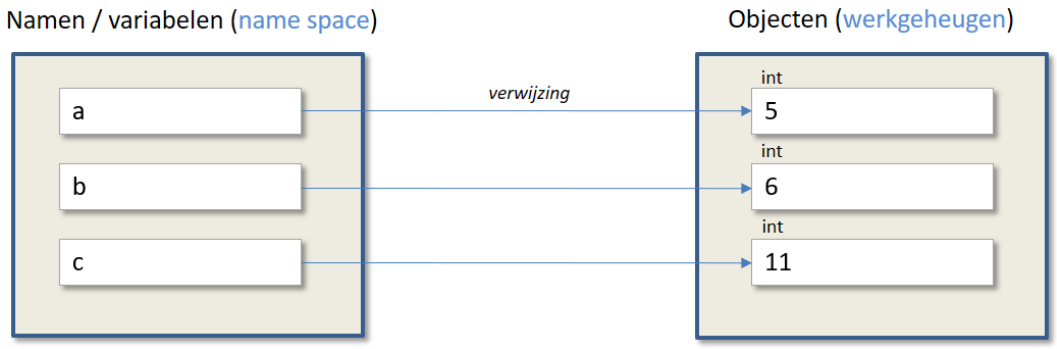
Wanneer dit codefragement wordt uitgevoerd, zullen de instucties sequentieel (van boven naar onder) worden uitgevoerd. Dit codefragment heeft als output:

```
11
```

In dit codefragment noemen we `a`, `b` en `c` **variabelen**. De variabele `a` verwijst naar (een object met) de **waarde** `5` en de variabele `b` verwijst naar (een object met) de waarde `6`. Om de waarde

van `c` te bepalen, zullen de waarden van `a` en `b` opgehaald worden en vervolgens bij elkaar worden opgeteld. Het resultaat, de waarde `11`, wordt toegekend aan de variabele `c`.

De schematische voorstelling van namen en objecten wordt weergegeven in een zogenaamd **toestandsdiagram**. Voor het bovenstaande codefragment wordt deze weergegeven in Figuur 3.1.



Figuur 3.1: Toestandsdiagram voor Codefragment 3.1.

Bij de uitvoer van de instructie `a = 5`, een zogenaamde **toekenningsopdracht** (*assignment statement*), wordt er in de *name space* (zie later) een variabele `a` aangemaakt die verwijst naar een object in het werkgeheugen met het geheel getal 5 als waarde. Idem voor de instructie `b = 6`: nu wordt een variabele `b` aangemaakt die verwijst naar het geheel getal 6. Bij de uitvoer van de instructie `c = a + b` worden eerst de waarden van de variabelen `a` en `b` opgevraagd, bij elkaar opgeteld en wordt een variabele `c` aangemaakt die verwijst naar de som van deze getallen, nl. 11.

Codefragment 3.1 toont dat variabelen kunnen verwijzen naar numerieke waarden en dat deze variabelen kunnen gebruikt worden in bewerkingen. Variabelen kunnen ook verwijzen naar tekstuele gegevens zoals getoond wordt in het onderstaande codefragment:

```
1 zin = "Het is mooi weer vandaag"
2 print(zin)
3 lengte = len(zin)
4 print(lengte)
```

Dit codefragment heeft als output:

```
Het is mooi weer vandaag
24
```

In dit codefragment verwijst de variabele `zin` naar (een object met) de waarde `Het is mooi weer vandaag`. Deze waarde is tekstueel. Om dit duidelijk te maken aan de interpreter wordt deze op regel 1 tussen dubbele aanhalingstekens (`" "`) geplaatst. In regel 3 wordt bepaald hoeveel karakters de zin bevat waarnaar de variabele `zin` verwijst (dit zijn 24 karakters, **inclusief spaties**).

Commentaar in code
Vaak is het aangewezen om broncode te voorzien van **commentaar**. Dit kan je doen door deze commentaar te laten voorafgaan door een hashtag (#). Wat na deze hashtag volgt, wordt **niet geïnterpreteerd** door de interpreter. Code voorzien van commentaar ziet er als volgt uit:

```
a = 5.0    # variabele a verwijst naar object met waarde 5.0
b = 6.0
c = a + b
print(c)   # print functie print waarde van c op scherm
```

Opmerking: de functie `print`
In de eerste codefragmenten wordt vaak gebruikgemaakt van de functie `print()`. Deze functie zorgt ervoor dat de waarde van een variabele die meegegeven wordt naar het scherm wordt geprint bij het uitvoeren van het codefragment. Deze functie zal verder in deze cursus uitvoerig besproken worden (in Sectie 3.5.3).

3.1.2 Objecten en gegevenstypes

Python is een object-georiënteerde programmeertaal. Men zegt vaak dat in Python *alles* een object is. Een gevolg daarvan is na het uitvoeren van de instructie

```
x = 5.0
```

de variabele `x` verwijst naar een **object** met als **waarde** `5.0` (dat in het geheugen bewaard wordt als een double-precision float, zie Hoofdstuk 2). Een object is een vrij abstract begrip en tot op dit moment hebben we nog maar één belangrijk aspect van een object vernoemd: zijn waarde.
De onderstaande instructie maakt een object aan met als waarde `"Het is mooi weer vandaag"`:

```
voorbeeld = "Het is mooi weer vandaag"
```

De voorgaande voorbeelden tonen aan dat Python objecten zowel **numeriek** als **tekstueel** kunnen zijn. Echter, de manier waarop numerieke gegevens bewaard worden in het werkgeheugen verschilt sterk van de manier waarop tekstuele gegevens bewaard worden. Bovendien zijn de bewerkingen die kunnen uitgevoerd worden op beide types gegevens verschillend. Een object met een tekstuele waarde heeft dan ook een ander **gegevenstype** [1] dan een object met een numerieke waarde. Het gegevenstype van een object is, naast zijn waarde, een tweede belangrijk aspect.

Drie belangrijke gegevenstypes zijn: `float`, `int` en `str`.

[1] Vaak wordt ook de benaming **datatype** gebruikt.

Het gegevenstype `float`
Een belangrijk gegevenstype voor numerieke gegevens in Python, en in het bijzonder **decimale getallen**. Het statement `a = 5.0` zorgt ervoor dat in het werkgeheugen een object van het type `float` wordt gecreëerd met als waarde `5.0`. Je kan floats optellen, vermenigvuldigen, enz. Het gegevenstype float wordt gebruikt om rationale getallen efficiënt voor te stellen in het werkgeheugen.
Let op: niet elk rationaal getal kan voorgesteld worden als een `float` (zie Sectie 2.3.1). Indien dit niet mogelijk is, wordt automatisch afgerond. De afrondingsfout is globaal genomen vrij klein.

Het gegevenstype `int` (lees *integer*)
Een belangrijk gegevenstype voor numerieke gegevens in Python, en in het bijzonder **gehele getallen**. Het statement `y = 7` zorgt ervoor dat in het werkgeheugen een object van het type `int` wordt gecreëerd met als waarde `7` (zie Sectie 2.2.2).

Het gegevenstype `str` (lees *string*)
Dit is het standaard gegevenstype voor tekstuele gegevens in Python. Het statement `voorbeeld = "een test"` creëert een object van het type string in het werkgeheugen met als waarde `"een test"`. Tekstuele gegevens zijn in Python een opeenvolging van karakters. Van een string object kunnen we bijvoorbeeld nagaan hoeveel karakters het bevat (met de functie `len()`). Na het uitvoeren van de instructie `len(voorbeeld)` zal 8 op het scherm verschijnen.

3.1.2.1 Objecten aanmaken in Python

Waarden zullen in Python steeds worden ondergebracht in objecten van een gepast type. Bij het uitvoeren van de instructie `a = 5`, wordt het karakter `5` door de interpreter herkend als een geheel getal met waarde 5 dat vervolgens wordt ondergebracht in een object van het type `int`. Hetzelfde geldt voor de instructie `w = "hallo"`. De karakters `"hallo"` worden door de interpreter herkend als tekst en leiden ertoe dat een string object wordt gecreëerd met waarde `hallo`.

Literals

Onderdelen van broncode, zoals `5` en `"hallo"` in de voorgaande voorbeelden, die de interpreter rechtstreeks herkent en aanleiding geven tot de creatie van nieuwe objecten noemt men **literals**.

In de voorbeelden tot nu werden de gecreëerde objecten steeds toegekend aan een variabele. De variabele verwijst dan naar het object zodat je dit object later in je code kan oproepen. Het toekennen van een object aan een variabele is echter niet noodzakelijk. Je kan een object aanmaken door een literal rechtstreeks in een interactieve sessie te typen (zie Sectie 1.5), bv.

```
>>> "ACAAGA"
'ACAAGA'

>>> 6.02214129e+23
```

```
6.02214129e+23
```

De eerste instructie maakt een object van het type **`str`** aan en de tweede instructie een object van het type **`float`**. **Let op**: omdat er geen variabelen verwijzen naar deze objecten kan je ze nadien niet meer aanspreken, maar het kan een handige manier zijn om na te gaan of een bepaalde waarde kan geïnterpreteerd worden.

Indien het object **niet aangemaakt** kan worden, wordt er een **foutmelding**[2] weergegeven:

```
>>> ACAAGA
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ACAAGA' is not defined

>>> 1 2 3 4 5
  File "<stdin>", line 1
    1 2 3 4 5
      ^
SyntaxError: invalid syntax
```

Opdracht 3.1

Voer in een Python console de volgende waarden (literals) in, bestudeer het resultaat en ga na welke waarden succesvol kunnen worden ingevoerd:

- `101300`
- `1013e2`
- `1013hPa`
- `1.0e0.1`
- `1.0e+308`
- `1.0e+309`
- `-122`
- `--122`
- `+122`
- `0b01100110` (probeer ook eens met dubbele quotes: `"0b01100110"`)
- `012`

Opmerking

De notatie `0b01100110` is een voorbeeld van een **binaire** voorstelling van een getal.

[2] Fouten en foutenbehandeling worden in Hoofdstuk 9 besproken.

Het gegevenstype opvragen: de functie `type()`

Elk object heeft een gegevenstype (of datatype). In Python zijn er verschillende soorten gegevenstypes. Voor een oplijsting van de gegevenstypes belangrijk in deze cursus, zie Sectie 3.11. Het gegevenstype van een object kan nagegaan worden met de functie `type()`:

```
>>> type("ACAAGA")
str

>>> type(101300)
int

>>> type(0.0089102)
float
```

Hieruit blijkt dat de objecten `"ACAAGA"`, `101300` en `0.0089102` respectievelijk van het type `str` (*string*), `int` (*integer*) en `float` (*float*) zijn.

Indien gebruik gemaakt wordt van variabelen, dan kan het type van een object opgevraagd worden via de variabele die ernaar verwijst. Dit is vaak de meest aangewezen manier van werken.

```
>>> dna = "ACAAGA"
>>> type(dna)
str

>>> x = 101300
>>> type(x)
int
```

Opdracht 3.2

Ga na wat de gegevenstypes zijn van de objecten die Python creëert voor onderstaande literals. Je zal een aantal nieuwe gegevenstypes zien. Deze zullen later nog uitgebreid aan bod komen. Stemmen de resultaten overeen met je verwachtingen?

- `159`
- `159.0`
- `5+8j`
- `[1, 2, 3, 4, 5]`
- `[]`
- `"B"`
- `"BBBBBBB"`
- `" "`
- `""`
- `(5.332, 'z')`

- `{"Belgium": 384, "Netherlands": 416, "Luxembourg": 243}`
- `True`
- `False`

3.1.3 Variabelen

In de voorgaande secties werd meermaals de term **variabele** gebruikt. Een variabele is een naam die verwijst naar een object. De toekenning van een naam aan een object gebeurt d.m.v. de toekenningsoperator (`=`). Een instructie waarin een dergelijke toekenning gebeurt, noemt men een **toekenningsstatement** (**assignment statement**). Voorbeelden van toekenningsstatements zijn:

```
>>> a = 7                                  # int met waarde 7
>>> zin = "Het is mooi weer vandaag!"      # str
>>> b = 18.2 + 15.1                        # float met waarde 33.3
>>> b_plus_een = b + 1.0                   # float met waarde 34.3
```

De laatste twee regels in het bovenstaande voorbeeld illustreren tevens de **kracht van variabelen**. Ze **verwijzen naar een object zodat we dit object**, en dus ook zijn waarde, **nadien kunnen gebruiken in andere instructies**.

Uit het voorgaande blijkt dat de naam van een variabele kort of lang kan zijn. Daarnaast kan een naam ook bijzondere tekens bevatten zoals underscores (_). Er zijn echter een aantal **regels waaraan de naam van een variabele moet voldoen**:

(1) Elke naam moet beginnen met een (grote of kleine) letter of een underscore (het karakter _):

- Een cijfer is niet toegelaten als eerste karakter.
- Meervoudige woordnamen kunnen gelinkt worden met de underscore karakter (_), bv. `veld_tarwe`, `veld_mais`. Voorlopig zullen we de underscore (_) **niet gebruiken als eerste karakter**[a].

(2) Na de eerste letter mag de naam een willekeurige combinatie zijn van letters, cijfers en underscores:

- De naam mag geen zgn. *keyword* zijn zoals in Tabel 3.1, zie gedeelte 3.11.
- Er mogen geen scheidingstekens (*delimiters*, bv. spaties), punctuatie (bv. `. , ; :`) of operatoren (bv. `+ - / *`) voorkomen in een naam.

(3) Een naam mag een willekeurige lengte hebben.

(4) Python is hoofdlettergevoelig: 'HOOFDLETTERS' zijn verschillend van 'kleine letters':

- `mijn_naam` is verschillend van `mijn_Naam` of `Mijn_naam`

[a] Variabelenamen die starten met een underscore hebben vaak een bijzondere betekenis.

Een voorbeeld van het aanmaken van een geldige variabelenaam in een interactieve sessie:

```
>>> c_plus_plus = "C++ is een programmeertaal."
```

Een voorbeeld van een ongeldige variabelnaam:

```
>>> c++ = "C++ is een programmeertaal."         # bevat operator +
  File "<stdin>", line 1
    c++ = "C++ is een programmeertaal."
        ^
SyntaxError: invalid syntax
```

Opdracht 3.3

Welke van de onderstaande variabelenamen zijn toegelaten in Python? Bij variabelenamen die niet geldig zijn, geef de reden(en).

- `xyzzy`
- `2deVariabele`
- `rich&bill`
- `een_lange_naam`
- `good2go`
- `python-input`

Controleer je antwoord door elk van de variabelen aan te maken en te laten verwijzen naar een object, bv. een *integer*, *float* of *string*.

Python keywords

Keywords zijn speciale woorden in Python die niet mogen gebruikt worden als variabelenaam (zie Tabel 3.1). Deze woorden hebben voor de Python interpreter dan ook een bijzondere betekenis. Verder in de cursus zullen een aantal van deze keywords aan bod komen.

Tabel 3.1: Overzicht van de Python *keywords*

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

Opdracht 3.4

Welke van de onderstaande instructies zijn toegelaten in Python? Geef bij uitdrukkingen die niet geldig zijn de reden(en).

- `define = "#ELEM = 7"`
- `global = 0b0101`
- `boolean_var = "True"`
- `local = 8`
- `True_or_False = 0`
- `yield = 8.0405e+6`

Controleer je antwoord door elk van de instructies in een interactieve sessie te typen en uit te voeren.

Link tussen een variabele(-naam) en object kan wijzigen

De link tussen een naam en het object waar die naam naar verwijst, kan wijzigen, zoals het volgende codefragment illustreert:

```
1 a = 15.1
2 b = 10.1
3 print("De waarden van a en b:")
4 print(a)
5 print(b)
6 a = "Hallo"
7 print("De nieuwe waarde van a:")
8 print(a)
```

Wanneer dit codefragment wordt uitgevoerd, wordt de volgende output bekomen:

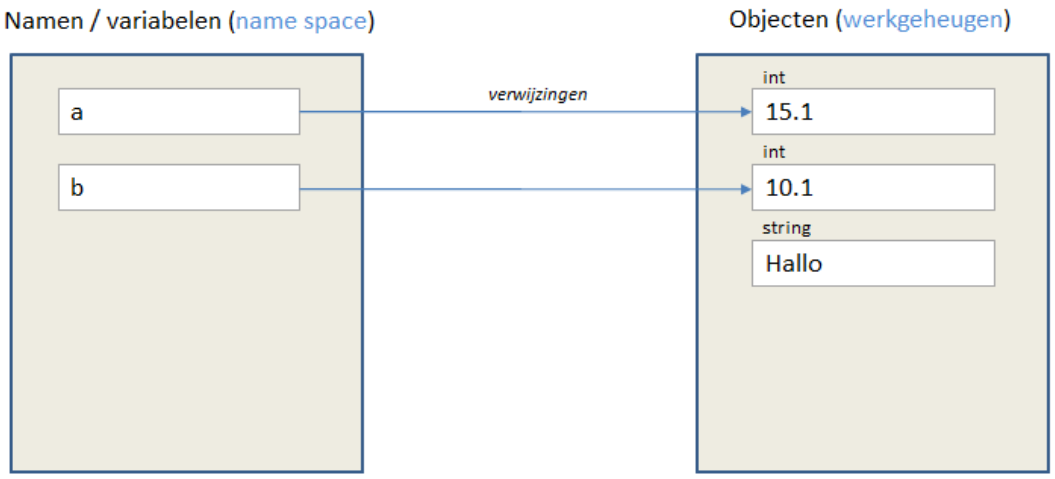
```
De waarden van a en b
15.1
10.1
De nieuwe waarde van a
Hallo
```

De variabele `a` verwijst eerst naar het **`float`**-object met waarde `15.1` (zie Figuur 3.2) en nadien naar het **`str`**-object met waarde `Hallo` (zie Figuur 3.3).

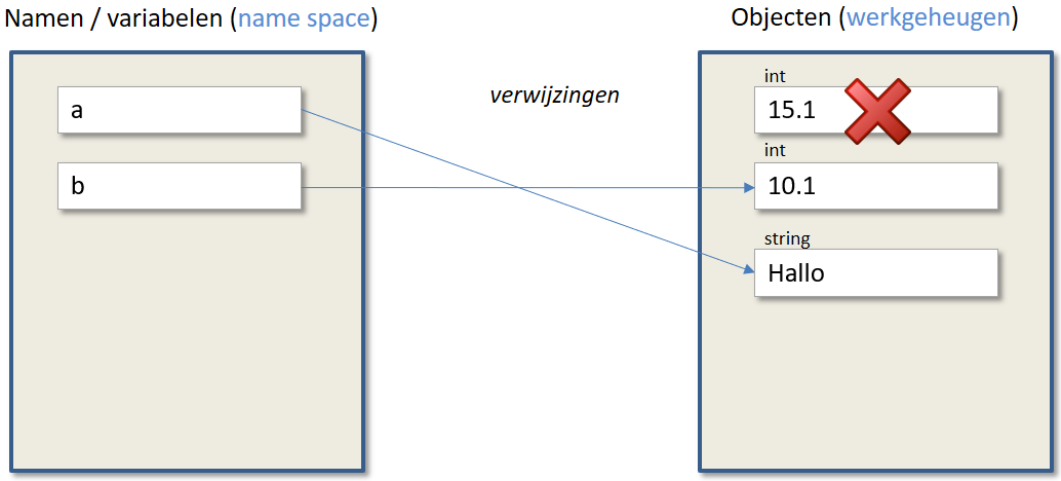
3.1.3.1 De functie `id()`

Telkens wanneer in Python een object aangemaakt wordt, krijgt dit object een identificatienummer (ID). Dit ID kan je terugkrijgen met de ingebouwde functie **`id()`**. Zolang een object in het werkgeheugen bestaat, blijft zijn ID ongewijzigd en is deze bovendien uniek[3].

[3] Je kan zo'n identificatienummer principieel vergelijken met het rijksregisternummer van een persoon.



Figuur 3.2: Het toestandsdiagram na de initiële toekenning.



Figuur 3.3: Het toestandsdiagram na de aanpassing van de variabele `a`.

In het volgende voorbeeld wordt een variabele `a` aangemaakt die verwijst naar een *integer* object met als waarde `7`. Het type wordt nagegaan met `type(a)` en dit stemt inderdaad overeen met `int` (*integer*). Het object ID wordt opgevraagd met `id(a)` en is in dit geval `10455232`.

```
>>> a = 7
>>> type(a)
int
>>> id(a)
10455232
```

De volgende instructies genereren een *float*-object met waarde `2.5` en variabele `b` die naar dit object verwijst.

```
>>> b = 2.5
>>> type(b)
float
```

```
>>> id(b)
140672328648360
```

Merk op dat de IDs van beide objecten **verschillend** zijn.

3.1.3.2 Aliasing

Aliasing treedt op wanneer een variabelenaam gebruikt wordt aan de rechterzijde van de toekenningsoperator. Beschouw het volgende voorbeeld:

```
>>> a = 7
>>> id(a)
10455232
>>> b = 2.5
>>> id(b)
140672328648360     # IDs verschillend

>>> a = b
>>> type(a)
float
>>> id(a)
140672328648360     # IDs gelijk
```

De variabelen `a` en `b` verwijzen nu naar hetzelfde object (de `id` is immers dezelfde). Dit principe heet **aliasing**. Omdat beide variabelen nu verwijzen naar hetzelfde object 2.5, zal de `ob_refcnt` de waarde 2 krijgen (zie Appendix A).

Het toekenningsstatement `a = b` leidt er bovendien toe dat `a` nu verwijst naar het *float*-object met als inhoud `2.5`, immers `a` is nu van het type *float* met hetzelfde ID als `b`. Er is nu geen verwijzing meer naar het oorspronkelijke *integer* object met als inhoud `7` dat niet meer beschikbaar is voor het programma en automatisch uit het werkgeheugen van de computer verwijderd zal worden. Ter info: de `ob_refcnt` van dit object krijgt de waarde 0 (zie Appendix A voor meer toelichting, ter info).

Garbage collection

Het automatisch vrijmaken van geheugenplaatsen waarnaar geen variabelen meer verwijzen heet ***garbage collection***. Dit houdt in dat alle objecten waarvoor de `ob_refcnt` de waarde 0 heeft uit het geheugen worden verwijderd.

Opdracht 3.5

- Voer de volgende instructies uit en bekijk de IDs van de aangemaakte variabelen.

```
>>> a = 5
>>> b = 6
```

- Zet het vorige voort met de volgende instructie en bekijk de inhoud en de IDs van de variabelen. Zoek zelf een verklaring voor de inhoud van de objecten en hun IDs.

```
>>> a = b
```

- Zet het vorige voort met de volgende instructie en bekijk opnieuw de inhoud en de IDs van de variabelen. Zoek zelf een verklaring voor de inhoud van de objecten en hun IDs.

```
>>> b = 10
```

Opdracht 3.6

Voer de volgende instructies in in een console.

```
>>> dier = "poes"
>>> meubel = "stoel"
>>> getal = 4
>>> dier = meubel
>>> meubel = getal
>>> getal = dier
```

- Wat zijn de waarden van de variabelen `dier`, `meubel` en `getal` na het uitvoeren van deze instructies?
- Verklaar waarom er **niet** meer naar het `str`-object `poes` verwezen wordt.

Opdracht 3.7

1. Creëer een variabele `zin` met de *string* `"We behandelen het eerste hoofdstuk."`
2. Creëer een variabele met naam `getal` die verwijst naar de *float* `3.14159`
3. Creëer een variabele `tijdelijk`, die een alias is voor `zin`.
4. Update `zin` zodat het een alias wordt voor `getal`.
5. Update `getal` zodat het een alias wordt voor `tijdelijk`.
6. Bekijk de waarden van `zin` en `getal`: deze zouden omgewisseld moeten zijn.

Het proces dat we in deze opdracht expliciet uitgevoerd hebben heet ***swapping*** (van het Engels *to swap*: wisselen). Bij het "swappen" wordt de inhoud van twee variabelen verwisseld. Dit gebeurt typisch d.m.v. een **tijdelijke** variabele. We zullen later zien dat dit "swappen" ook op andere manieren kan.

3.2 Expressies en statements

3.2.1 Expressies

In de voorgaande voorbeelden hebben we beschreven dat waarden in Python worden ingekapseld in objecten en dat variabelen verwijzingen zijn naar deze objecten. Op deze objecten kunnen vervolgens bewerkingen worden uitgevoerd: bv. twee *floats* optellen. Het resultaat van deze bewerkingen is een nieuw object. Dergelijke instructies worden in Python **expressies** genoemd.

Formeel wordt een **expressie** gedefinieerd als:

een **combinatie** van **waarden**, **variabelen** en **operatoren**.

Voorbeelden van expressies zijn:

```
53.2 + 12.5  # resultaat is de som van 53.2 en 12.5
9 * 6        # resultaat is het product van 9 en 6

a = 3        # dit is een toekenningsstatement (geen expressie, zie verder)
5 + a        # resultaat is de som van 5 en 3 met
             # 3 de waarde van het object waarnaar a verwijst
```

We beschouwen de expressie `53.2 + 12.5` nu in detail:

- Het plusteken (`+`) geeft aan dat de getallen `53.2` en `12.5` moeten opgeteld worden en is een voorbeeld van een **operator**[4]. In de expressie `9 * 6` is `*` de (vermenigvuldigings)operator[5].
- De waarden waarop de operator inwerkt, in dit voorbeeld de getallen `53.2` en `12.5`, zijn de **operanden**.
- Wanneer de expressie `53.2 + 12.5` uitgevoerd wordt door de interpreter, dan zeggen we dat deze **geëvalueerd** wordt.
- Wanneer een expressie geëvalueerd wordt, dan wordt een nieuw object gegenereerd. Men zegt dat dit object **geretourneerd** wordt door de expressie.

De expressie `5 + a` bevat nu ook een variabele. Wanneer deze expressie geëvalueerd wordt, zal het volgende gebeuren:

- De variabele `a` wordt automatisch vervangen door het object waarnaar ze verwijst.
- De expressie wordt verder geëvalueerd met deze waarde.

Wanneer een expressie wordt ingevoerd in een interactieve sessie, dan wordt de **waarde van het geretourneerde object onmiddellijk getoond**:

[4]Ook in de wiskunde noemt men dit een operator. Meer precies is `+` een *binaire* operator. Men noemt deze operator binair omdat hij inwerkt op *twee* waarden, de operanden. Er bestaan ook unaire operatoren.

[5]In de volgende sectie volgt een overzicht van alle mogelijke operatoren in Python.

```
>>> 47 + 53          # expressie: combinatie getallen en operator
100                  # resultaat van de expressie 47 + 53
```

```
>>> x = 4            # statement
>>> x**2 + x - 20    # expressie (de operator ** is de machtoperator)
0                    # resultaat van de expressie
```

In de voorgaande voorbeelden waren de operanden steeds numeriek (type `float` of `int`). Bepaalde operatoren kunnen echter ook inwerken op strings. Twee belangrijke voorbeelden zijn de operatoren `+` en `*`.

3.2.1.1 Concatenatie van strings met de + operator

Wanneer de `+` operator wordt toegepast in een expressie waarin beide operanden *strings* zijn, dan worden de strings **geconcateneerd** (samengevoegd). Hieronder wordt de expressie `"mooi" + "weer"` geëvalueerd in een interactieve sessie:

```
>>> "mooi" + "weer"  # expressie
mooiweer             # resultaat van de expressie
```

De oorspronkelijke strings `"mooi"` en `"weer"` worden samengevoegd tot een **nieuwe** string `"mooiweer"`.

Als één van beide operatoren een *integer* of *float* is, dan geeft dit aanleiding tot een **foutmelding**:

```
>>> 25 + "graden"
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> 18.5 + "MeV"
TypeError: unsupported operand type(s) for +: 'float' and 'str'
```

3.2.1.2 Duplicatie van strings met de * operator

Wanneer de `*` operator wordt toegepast in een expressie waarin één van de operanden een *string* is en de andere een *integer*, dan wordt de string **gedupliceerd**:

```
>>> "mooi" * 5                  # expressie
mooimooimooimooimooi             # resultaat van de expressie

>>> 3 * "hallo"
hallohallohallo
```

Als beide operanden *strings* zijn, dan geeft dit aanleiding tot een **foutmelding**:

```
>>> "7" * "drie"
TypeError: can't multiply sequence by non-int of type 'str'
```

3.2.2 Statements

Een **statement** is een instructie die de Python interpreter kan uitvoeren, maar in tegenstelling tot een expressie, **geen** waarde retourneert.

In wat volgt, bespreken we **toekenningsstatements** en `print`-statements.

3.2.2.1 Het toekenningsstatement

Een belangrijk type statement is het **toekenningsstatement** zoals bijvoorbeeld `a = 5`.

Het resultaat van een expressie kan ook worden toegekend aan een nieuwe variabele. Dit geheel noemt men ook een **toekenningsstatement**. Voorbeelden van dergelijke statements zijn:

```
>>> a = 5
>>> b = 2 + a * 3       % voorrangsregels cfr. wiskunde (en zie verder)
>>> woord = "hallo"
>>> woorden = woord * a
```

De waarden van de objecten waarnaar `b` en `woorden` verwijzen, kunnen bekeken worden met de functie `print()`, of door de naam van de variabele te typen in de console.

```
>>> print(b)
17
>>> woorden
'hallohallohallohallohallo'
```

3.2.2.2 De functie `print()`

Wanneer de functie `print()` gebruikt wordt in een instructie, om de waarde van een variabele op het scherm te printen, dan spreekt men soms van een ***print**- statement*. De functie `print()` retourneert zelf schijnbaar niets[6] [7], maar toont enkel een waarde op het scherm.

```
>>> print(56)          # statement: print het getal 56 naar het scherm
56                     # resultaat van print(56)

>>> print("Faculteit: FBW")    # statement
Faculteit: FBW                 # resultaat van print("Faculteit: FBW")
```

[6] Strikt genomen retourneert elke functie, dus ook de functie `print()`, een object. De functie `print()` retourneert een object van het (gegevens)type `NoneType`. Het `NoneType` object vormt een manier om instructies die niets wezenlijks retourneren toch een waarde te laten retourneren. Het toekenningsstatement `resultaat = print("Hallo")` kan dus geïnterpreteerd worden door de interpreter. De variabele `resultaat` zal dan verwijzen naar een `NoneType`

[7] Omdat de functie `print()` een waarde retourneert, is de oproep van de functie `print()` strikt genomen een expressie! Er wordt nog vaak verwezen naar `print()` als een statement. Deze foute terminologie is nog een restant van de terminologie in Python 2, waar `print()` wel een statement was.

```
>>> mijn_int = 5
>>> print(mijn_int)
5
>>> print("De getalwaarde is:", mijn_int)  # een print statement dat twee
De getalwaarde is: 5                       # objecten op het scherm toont
                                           # (zie verder)
```

Opdracht 3.8

Welke van de onderstaande instructies zijn *statements* en welke zijn *expressies*? Controleer je antwoord door elk van de instructies in een interactieve sessie uit te voeren. Bemerk dat sommige instructies gescheiden zijn door een punt-komma (;) om ze op één regel te kunnen plaatsen.

- `y = "hahaha" + "hihihi"`
- `5 * (1 + 1 / (1 + 8))**3`
- `x = 2; y = 3; x*y`
- `abs(-6.249)`
- `len("Hoe lang is deze zin?")`
- `m = 66.0; g = 9.81; print("F =", m*g, "N")`

Opdracht 3.9

Bekijk de volgende instructies en tracht te verklaren waarom een foutmelding optreedt.

```
>>> x = 2
>>> print(x + 5)        # expressie x + 5 wordt geevalueerd en
7                       # resultaat wordt argument van print
>>> print(y = x + 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'y' is an invalid keyword argument for this function
```

3.3 Operatoren en operanden

In de voorgaande secties werden reeds een aantal operatoren besproken die kunnen inwerken op floats, integers en strings. In deze sectie wordt dit overzicht vervolledigd.

3.3.1 Binaire operatoren voor numerieke operanden

Binaire operatoren zijn operatoren die inwerken op **twee operanden**. De meest gekende binaire operatie (of bewerking) is ongetwijfeld de optelling. In de bewerking

```
5 + 3
```

stelt het plusteken de operator (die de optelling symboliseert) voor en zijn `5` en `3` de operanden.

Voor de wiskundige bewerkingen optelling, aftrekking, vermenigvuldiging, deling en machtsverheffing zijn in Python operatoren voorzien. Tabel 3.2 geeft een overzicht van de binaire operatoren (a.d.h.v hun symbolen) die in Python gedefiniëerd zijn voor numerieke (type `float` en `int`) operanden.

Tabel 3.2: Overzicht van binaire operatoren in Python (met operanden van het type *int* of *float*).

Symbool	Bewerking	Opmerking
`+`	optelling	
`-`	aftrekking	
`*`	vermenigvuldiging	
`/`	deling	
`//`	**gehele** deling (quotiënt)	`17 // 5` heeft als resultaat `3`
`%`	**rest** (modulus operator)	`17 % 5` heeft als resultaat `2`
`**`	machtsverheffing	`2 ** 3` lees je als **twee tot de derde macht** (2^3)

Het gegevenstype van het resultaat van een expressie die één van deze operatoren bevat, hangt af van het type van de operanden.

Hierna volgt een overzicht.

3.3.1.1 Minstens één operand van het type `float`

Indien minstens één van de operanden van het type `float` is, dan zal voor **elke operator** ook het resultaat van het type `float` zijn:

```
>>> a = 3.1 + 2.5       # optelling van twee floats
>>> a
5.6
>>> type(a)
float                   # type is float
```

```
>>> b = 6.0 / 2         # deling van float en integer
>>> b
```

```
3.0
>>> type(b)
float                    # type is float
```

3.3.1.2 Beide operanden van type `int`

Indien beide operanden van het type **`int`** zijn, is het type van het resultaat afhankelijk van de gebruikte operator.

De operatoren `+`, `-`, `*`, `//`, `%` en `**` hebben als resultaat een object van het type **`int`**:

```
>>> 8//3              # gehele deling
2                     # resultaat van 8//3
>>> type(8//3)        # controleer het type
int                   # type is inderdaad 'int'
```

```
>>> 8**3              # machtsverheffing met twee integers
512                   # resultaat 8**3
>>> type(8**3)        # controleer het type
int                   # type is inderdaad 'int'
```

Enkel de operator `/` heeft steeds een object van het type **`float`** als resultaat:

```
>>> 8/2               # (gewone)deling
4.0                   # resultaat (een kommagetal!)
>>> type(8/2)         # expliciete controle van het type
float                 # type is inderdaad 'float'
```

Opdracht 3.10

Stel dat `var_int = 42` en `var_float = 42.0`, voorspel de waarde en het type van de objecten die geretourneerd worden door de volgende expressies en controleer in een interactieve sessie.

- `var_int * 5`
- `var_int * 5.0`
- `var_float + 5.0`
- `var_int / 7`
- `var_float // 6.0`
- `var_float % 5`
- `var_int % 5`

3.3.2 Unaire operatoren voor numerieke operanden

Unaire operatoren werken in op **één operand**. Voorlopig beschouwen we uitsluitend de unaire min (-) en de unaire plus[8] (+).

```
>>> a = 5
>>> type(a)
>>> -a      # de unaire min veroorzaakt een tekenwissel
-5
>>> type(-a)
int         # het gegevenstype blijft behouden
```

3.3.3 Binaire operatoren voor het type `str`

De bewerkingen die gedefiniëerd zijn voor strings in Python zijn **concatenatie** (operator +, zie Sectie 3.2.1.1) en de **duplicatie** (operator *, zie Sectie 3.2.1.2).

De **concatenatie** (operator +) werkt in op twee operanden van het type **str** en heeft als resultaat een object van het type **str** waarin de twee oorspronkelijke strings bij elkaar zijn gevoegd, zoals geïllustreerd wordt in het volgende voorbeeld:

```
>>> woord1 = "hallo"
>>> woord2 = "test"
>>> woord3 = woord1 + woord2  # concatenatie
>>> print(woord3)
hallotest
```

Duplicatie (operator *) werkt in op één operand van het type **str** en één operand van het type **int**. Het resultaat is steeds een object van het type **str**.

```
>>> "hallo" * 3
hallohallohallo
>>> 4 * "test"
testtesttesttest
>>> 60 * '-'
'------------------------------------------------------------'
```

Opdracht 3.11

Stel dat `mijn_str = "Hello"` en `jouw_str = "World"`, ga na in een interactieve sessie wat het resultaat zal zijn van de volgende expressies:

- `mijn_str + jouw_str`
- `mijn_str - jouw_str`

[8] Dit is in Python ook een unaire operator, maar het gebruik ervan bij een numeriek operand heeft weinig nut.

- `mijn_str * jouw_str`
- `jouw_str + mijn_str`
- `mijn_str + '' + jouw_str`
- `mijn_str + ', hello '+ jouw_str + '!'`

Opdracht 3.12

Stel dat `mijn_str = "Hello"`, ga na in een interactieve sessie wat het resultaat is van de volgende expressies:

- `mijn_str * 3`
- `3 * mijn_str`
- `(mijn_str + '') * 3`
- `mijn_str + 3`
- `6 * '#'+ '' + mijn_str + ''+ 6 * '#'`
- `'5' * 3`
- `5 * 3`

3.3.4 Bitsgewijze operatoren

Pyhton bevat 4 operatoren die bitsgewijs werken: de ***AND*** (&), de ***OR*** (|), de ***XOR*** (∧) en de ***not*** (∼) operatoren.

In Python worden bitsgewijze operatoren gebruikt om bitsgewijze berekeningen uit te voeren op gehele getallen. De gehele getallen worden eerst omgezet naar hun binaire voorstelling en vervolgens worden bewerkingen uitgevoerd op elk bit of corresponderend paar bits. Vandaar de naam bitsgewijs. Het resultaat wordt teruggegeven in decimaal formaat.

Enkele voorbeelden:

```
>>> a = 10   # binair: 10 = 1010_2
>>> b = 4    # binair:  4 = 0100_2
>>> a & b
0                          # 0000_2 = 0

>>> a | b
14                         # 1110_2 = 14
```

3.3.5 Operator voorrang

Indien een expressie meerdere operatoren bevat, dan moet je rekening houden met **voorrangsregels** (zie Tabel 3.3) die de volgorde bepalen waarin de operatoren inwerken op hun operanden. Om af te wijken van de standaardvolgorde, en vaak ook voor de duidelijkheid, worden haakjes gebruikt. Bewerkingen die in de tabel op gelijke hoogte staan, zoals optellen en aftrekken, zijn gelijkwaardig. **Gelijkwaardige bewerkingen worden van links naar rechts uitgevoerd.**

Tabel 3.3: Voorrangsregels van de operatoren en haakjes, geordend volgens prioriteit, van hoog (1) naar laag (5).

Prioriteit	Operator(en)	Betekenis van de operator(en)
1	`()`	haakjes
2	`**`	machtsverheffing
3	`+x`, `-x`	unaire plus of min
4	`*`, `/`, `%`, `//`	vermenigvuldiging, deling, rest, gehele deling
5	`+`, `-`	optellen, aftrekken

Enkele voorbeelden:

```
>>> 3.2 * 5 + 2
18.0
>>> 3.2 * (5 + 2)
22.400000000000002   # merk ook de afronding tgv float op
>>> 5**2*3
75
>>> -5**2*3
-75
>>> 5 * 3 // 2
7
```

Opdracht 3.13

Gegeven de volgende expressie: `30 - 3 ** 2 + 8 // 3 ** 2 * 10`.

(a) Wat is het resultaat van de expressie? Controleer je antwoord in een interactieve sessie.

(b) Rekeninghoudend met de voorrangsregels, geef dezelfde bewerking met haakjes weer om de uitdrukking te verduidelijken. Ga na dat je hetzelfde resultaat bekomt als in (a).

Opdracht 3.14

Voorspel het resultaat en controleer dit in een interactieve sessie.

(a) `2**2**3`

(b) `2**(2**3)`

(c) `(2**2)**3`

Waarom hebben twee van de bovenstaande expressies dezelfde uitkomst?
Herschrijf expressie (c) met één machtsverheffingsoperator (`**`) en één vermenigvuldigingsoperator (`*`).

Opdracht 3.15

Je wenst het volgende uit te rekenen: $6^{3/2}$

1. Welke van de onderstaande expressies levert het juiste resultaat op?

   (a) `6**3/2`

   (b) `(6**3)/2`

   (c) `6**(3/2)`

   (d) `6**(3//2)`

2. Welke twee expressies zijn **gelijkwaardig**? ..... en .....

3.4 Eerste codescripts – programma's

Tot nu toe hebben we alle instructies rechtstreeks via een interactieve sessie ingegeven. Dit is handig als we een reeks instructies interactief willen uitvoeren of indien we Python gebruiken als een uitgebreide rekenmachine.

In vele gevallen wensen we meerdere instructies te bundelen in een broncode-bestand om deze (eventueel ook op een later moment) sequentiëel te laten uitvoeren door de interpreter. In deze gevallen kunnen we de instructies onder elkaar plaatsen in een tekstfile (en deze opslaan met de **extensie** `.py`). Een dergelijke file wordt ook wel een Python *script* genoemd. We beschouwen ter illustratie het volgende probleem.

Een elektrisch verwarmingselement wordt aangesloten op een gelijkspanning van $U = 120.0\,V$. In de kring meet je een stroomsterkte van $I = 4.8\,A$. Bereken de weerstand van het verwarmingselement, het vermogen en de hoeveelheid warmte-energie geproduceerd in 1 uur. De weerstand R, het vermogen P en de warmte-energie Q worden berekend m.b.v. de volgende formules:

$$R = \frac{U}{I} \qquad P = U \cdot I \qquad Q = P \cdot \Delta t\,.$$

Het onderstaande codefragment is een voorbeeld van een Python script, dat werd opgeslagen in het bestand `warmte_energie.py`, en deze drie grootheden berekent:

Fragment 3.2: `warmte_energie.py`

```
1  U = 120.0   # spanning in V
2  I = 4.8     # stroomsterkte in A
3  dt = 1      # tijdsduur in uur (1 h = 3600 s)
4
5  R = U / I   # berekening van de weerstand in Ohm
6  print("Weerstand R =", R, "Ohm")
7
8  P = U * I   # berekening van het vermogen in W
9  print("Vermogen P =", P, "W")
10
11 Q = P * dt  * 3600   # warmte-energie in J
12 print("Warmte-energie Q =", Q, "J")
```

Wanneer het wordt uitgevoerd, levert het programma `warmte_energie.py` volgende output:

```
Weerstand R = 25.0 Ohm
Vermogen P = 576.0 W
Warmte-energie Q = 2073600.0 J
```

Opmerking: notatie
De lijnnummers aan de linkerkant van de code in Fragment 3.2 maken **geen deel** uit van de code.

Opdracht 3.16 (`weerstand_serie_parallel.py`)
Schrijf een script met als naam `weerstand_serie_parrallel.py` dat de vervangingsweerstand (substitutieweerstand) berekent van twee weerstanden $R_1 = 3.0\,\Omega$ en $R_2 = 9.0\,\Omega$ in het geval van **serie**- (R_s) en **parallel**schakeling (R_p). Generieke formules voor het berekenen van de vervangingsweerstand van N serie- of parallelgeschakelde weerstanden zijn de volgende:

$$R_s = \sum_{i=1}^{N} R_i \qquad R_p = \frac{1}{\sum\limits_{i=1}^{N} \frac{1}{R_i}}\,.$$

Wanneer je dit script uitvoert, moet de volgende informatie op het scherm verschijnen:

```
Waarden elektrische weerstand:
R1 = 3.0 Ohm
R2 = 9.0 Ohm
Vervangingsweerstanden:
Rs = 12.0 Ohm  (serieschakeling)
Rp = 2.25 Ohm  (parallelschakeling)
```

Vóór je begint:

- Open de map `Wetenschappelijk programmeren` als werkruimte in VS Code[a] en maak een nieuw Python script `weerstand_serie_parallel.py` aan.

- Voor de berekening van de vervangingsweerstanden, werden de generieke formules gegeven voor N weerstanden. In deze opdracht hebben we maar twee weerstanden, schrijf deze formules expliciet uit voor $N = 2$.

$$R_s = \dots\dots\dots\dots\dots\dots\,, \qquad R_p = \dots\dots\dots\dots\dots\dots\dots\dots\dots$$

- Gebruik Fragment 3.2 als leidraad voor deze opdracht.

[a]Hierbij gaan we ervan uit dat je de installatie- en configuratiestappen in het document "**Handleiding Visual Studio Code met Python**" correct uitgevoerd hebt.

Opdracht 3.17 (`massacentrum.py`)
Schrijf een script met als naam `massacentrum.py` dat de positie x_C van het massacentrum berekent voor drie (punt)massa's $m_1 = 5.3\ kg$, $m_2 = 2.6\ kg$ en $m_3 = 1.7\ kg$ die zich langs eenzelfde as respectievelijk op posities $x_1 = 0.9\ m$, $x_2 = 1.3\ m$ en $x_3 = -0.2\ m$ bevinden.

De generieke formule voor het berekenen van de positie x_C van het massacentrum van N (punt)massa's (m_i) op posities (x_i) voor $i = 1, \cdots, N$ langs een as wordt gegeven door:

$$x_C = \frac{\sum_{i=1}^{N} m_i x_i}{\sum_{i=1}^{N} m_i}$$

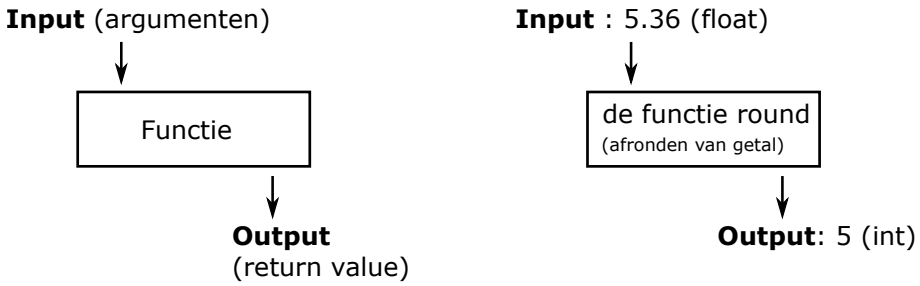
Wanneer je dit script uitvoert, moet de volgende informatie op het scherm verschijnen:

```
xC = 0.8135416666666667 m
```

3.5 Gebruik van functies

3.5.1 Functies oproepen in Python

Een Python-functie is een benoemde sequentie van instructies (een stuk broncode) dat bij naam aangeroepen kan worden in een programma en dat een specifieke taak uitvoert. Een functie kan gezien worden als een structuur die **input** (bijvoorbeeld een `float` of `str`) aanvaardt en verwerkt tot **output**. Figuur 3.4 illustreert dit concept.



Figuur 3.4: (links) Schematische voorstelling van een functie als een structuur die input vertaalt naar output. (rechts) Toepassing van dit schema voor de functie `round` die een decimaal (type `float`) getal afrondt tot een geheel getal (type `int`).

In Python wordt de functie `round()` als volgt opgeroepen:

```
>>> round(5.36)
5
```

Wanneer een functie als onderdeel van een expressie gebruikt wordt, dan zegt men dat deze functie wordt **opgeroepen** (*function call*). Bij een functie-oproep zijn 3 aspecten van een functie belangrijk. Dit zijn de naam, input en output. Deze worden in Python als volgt benoemd.

1. Elke functie heeft een **naam**: in het voorbeeld is `round` de naam van de functie. **Een functie wordt steeds opgeroepen o.b.v. zijn naam.**
2. De objecten (of de variabelen die ernaar verwijzen) die de input vormen van de functie, zoals de `float` `5.36` in het voorbeeld, noemen we de **argumenten** van de functie. De argumenten worden bij de functie-oproep tussen ronde haakjes `()` geplaatst na de naam. Indien er meerdere argumenten zijn, worden deze gescheiden door komma's (`,`).
3. Het object dat de **output** vormt, noemen we de **return value**.

In het voorbeeld wordt de functie `round()` opgeroepen met als argument 5.36. De return value is 5.

Een ander voorbeeld is de functie `abs()` (wiskundige notatie: $|.|$). Van een numeriek argument (type `float` of `int`) zal deze functie de absolute waarde berekenen en retourneren, waarbij het type van de *return value* hetzelfde is als dat van het argument.

```
>>> abs(-5.36)
5.36
```

Functies met meerdere argumenten

De meeste functies in Python kunnen meerdere argumenten aanvaarden. Een eenvoudig voorbeeld is de functie `min()`. Deze functie bepaalt de kleinste waarde van zijn argumenten en geeft deze terug als return value. Indien er **meerdere argumenten** worden meegegeven, worden deze steeds **gescheiden door komma's**.

```
>>> min(3.16, 4.23, 1.2)
1.2
```

Merk op dat niet elke functie meerdere argumenten kan aanvaarden. Indien het aantal meegegeven argumenten niet overeenstemt met het verwachte aantal argumenten, wordt een foutmelding opgeworpen:

```
>>> abs(-5, -3)
TypeError: abs() takes exactly one argument (2 given)
```

Gebruik van variabelen bij de functie-oproep

In de voorgaande voorbeelden zijn de argumenten bij de functie-oproep waarden. De argumenten kunnen ook variabelen zijn en de return value kan worden toegekend aan een nieuwe variabele. Dit wordt geïllustreerd in het volgende voorbeeld:

```
>>> a = 5.36
>>> b = round(a)
>>> a
5.36              # waarde van a blijft ongewijzigd
>>> b
5                 # variabele b verwijst naar nieuw object met waarde 5
```

Merk op dat de waarde van de variabele die als argument wordt gebruikt niet wijzigt! Dit is bij zeer veel functies in Python het geval.

Een functie-oproep met meerdere argumenten laat eenzelfde manier van werken toe:

```
>>> a = 5.36
>>> b = 3.2
>>> c = min(a, b)
>>> c
3.2
```

In zijn meest algemene vorm is de syntaxis van een functie-oproep de volgende:

```
returnValue = functieNaam(argument1, argument2, argument3, ...)
```

Nesten (of samenstellen) van functie-oproepen

In de wiskunde kan men functies samenstellen. Wenst men bijvoorbeeld aan te geven dat de sinus van de absolute waarde van de variabele x moet bepaald worden, dan kan men dit noteren als $\sin(|x|)$. Eerst zal dan de absolute waarde bepaald worden van x en van deze absolute waarde wordt vervolgens de sinus berekend. Ook in Python kan men deze functies gaan samenstellen. Men noemt dit het nesten van functie-oproepen.

Als voorbeeld bepalen we $\min(|-5|, |-3|)$:

```
>>> min(abs(-5), abs(-3))
3
```

Vergelijk dit met $|\min(-5,-3)|$:

```
>>> abs(min(-5, -3))
5
```

Opdracht 3.18 (`gebruikfuncties.py`)
Schrijf een script met als naam `gebruik_functies.py` waarin het volgende gebeurt.

- Maak twee variabelen a en b aan die verwijzen naar twee `float` objecten met waarden die je zelf mag kiezen.
- Bepaal het minimum van $(a\sqrt{|b|}+1)^2$ en $(a^2+1)(|b|+1)$ en ken het resultaat toe aan de variabele c.
- Breng de volgende boodschap op het scherm: `Het minimum is ....`

Test je code: voor $a = 1.2$ en $b = -2.3$ is het minimum bij benadering 7.95178.

Voor je begint

- Lees eerst sectie 3.5.1

Ter info: als gevolg van de ongelijkheid van Cauchy-Schwarz zou het minimum steeds de eerste van deze twee uitdrukkingen moeten zijn.

3.5.2 De function signature, parameters, positionele en keyword argumenten

Python bevat een beperkt aantal ingebouwde (*built-in*) functies (71 in versie 3.11.2). Een alfabetisch overzicht van deze ingebouwde functies wordt gegeven in Tabel 3.4.

Tabel 3.4: Een alfabetisch overzicht van alle ingebouwde functies in Python (versie 3.11.5).

A	**E**	**L**	**R**
`abs()`	`enumerate()`	`len()`	`range()`
`aiter()`	`eval()`	`list()`	`repr()`
`all()`	`exec()`	`locals()`	`reversed()`
`any()`			`round()`
`anext()`	**F**	**M**	
`ascii()`	`filter()`	`map()`	**S**
	`float()`	`max()`	`set()`
B	`format()`	`memoryview()`	`setattr()`
`bin()`	`frozenset()`	`min()`	`slice()`
`bool()`			`sorter()`
`breakpoint()`	**G**	**N**	`staticmethod()`
`bytearray()`	`getattr()`	`next()`	`str()`
`bytes()`	`globals()`		`sum()`
		O	`super()`
C	**H**	`object()`	
`callable()`	`hasattr()`	`oct()`	**T**
`chr()`	`hash()`	`open()`	`tuple()`
`classmethod()`	`help()`	`ord()`	`type()`
`compile()`	`hex()`		
`complex()`		**P**	**V**
	I	`pow()`	`vars()`
D	`id()`	`print()`	
`delattr()`	`input()`	`property()`	**Z**
`dict()`	`int()`		`zip()`
`dir()`	`isinstance()`		
`divmod()`	`issubclass()`		`_`
	`iter()`		`__import__()`

De volledige lijst van de ingebouwde functies en links naar de helppagina's is terug te vinden via de volgende link: `https://docs.python.org/3/library/functions.html`.

De modules en packages die in een environment aanwezig zijn (zie Sectie 3.6) verhogen het aantal beschikbare functies al snel tot enkele duizenden. Het is dus, zelfs voor een ervaren programmeur, onmogelijk om alle functies te kennen. Bij het eerste gebruik van een functie is het dan ook belangrijk de **functie-documentatie** te raadplegen. In de eerste plaats kan dit door gebruik te maken van de functie `help()`. De functie `help()` kan eenvoudig worden opgeroepen in de console met als argument de naam van de functie.

`help(`*functieNaam*`)`

Voor de functie `round()` bekomen we de volgende informatie:

```
>>> help(round)
Help on built-in function round in module builtins:

round(number, ndigits = None)
    Round a number to a given precision in decimal digits.

    The return value is an integer if ndigits is omitted or None. Otherwise
    the return value has the same type as the number. ndigits may be negative.
```

Uit de woordelijke beschrijving is het duidelijk dat de functie `round` een getal afrondt. Daarnaast is vooral de volgende regel belangrijk:

```
round(number, ndigits = None) -> number
```

Deze regel noemt men vaak de **function signature** omdat hij aangeeft welke (types) argumenten een functie kan aanvaarden en wat het type van de return value zal zijn[9]. We ontleden deze regel nu stap-voor-stap.

De parameters van `round()`

De functie `round()` heeft twee **parameters**: `number` en `ndigits`. Dit zijn twee namen die tijdelijk (en automatisch) worden toegekend aan de argumenten tijdens de functie-oproep. Dit wil zeggen dat de functie `round()` twee argumenten kan aanvaarden:

- `number`: het getal (`float` of `int`) dat men wenst af te ronden.
- `ndigits`: (staat voor number of digits) het aantal decimalen (cijfers na de komma), een `int`.

De volgorde waarin deze parameters voorkomen, bepaalt ook de volgorde van de argumenten. De link tussen parameters en argumenten wordt gelegd o.b.v. positie. We noemen deze argumenten daarom **positionele argumenten**:

```
>>> round(5.32434, 2)
5.32
```

Een alternatieve manier om argumenten door te geven aan een functie maakt gebruik van **keyword argumenten**. Dit gebeurt via de toekenning `parameter = waarde`.

```
>>> round(number = 5.32434, ndigits = 2)
5.32
```

Het resultaat is hetzelfde, maar de functie-oproep wordt iets langer. Een voordeel is dat de broncode mogelijks duidelijker wordt omdat de programmeur duidelijk aangeeft wat de functie is van de argumenten. Bovendien is het mogelijk om ook de **positie van de argumenten vrij te kiezen**.

[9]Merk op dat de term *function signature* in Python iets soepeler dient te worden geïnterpreteerd dan in bv. Java of C. In deze talen ligt het gegevenstype van de argumenten en de return value vast. In Python is dit niet noodzakelijk het geval. De functie `abs()` kan bijvoorbeeld zowel een argument van het type `int` als van het type `float` aanvaarden.

```
>>> round(ndigits = 2, number = 5.32434)
5.32
```

De optionele argumenten van `round()`

Als tweede argument staat `, ndigits = None` om aan te geven dat dit een **optioneel argument** is. Dat wil zeggen dat er een standaard (*default*) waarde wordt toegekend aan de parameter `ndigits` indien er geen waarde wordt voorzien bij de functie-oproep. In de documentatie lezen we dat wanneer maar één argument wordt meeggeven de return value een `int` is. Merk het **subtiele verschil** op wanneer `ndigits = 0` wordt meegegeven (de return value is nu een `float`).

```
>>> round(5.32434)
5         # int
>>> round(5.32434, ndigits = 0)
5.0       # float
>>> round(5.32434, ndigits = None)
5         # int
```

Het voorgaande voorbeeld geeft aan dat de default waarde voor `ndigits` niet 0 is, maar `None`[10].

Opdracht 3.19 (`waterdebiet.py`)

De functies `pow()` en `round()` zijn ingebouwde functies. Schrijf een script `waterdebiet.py` waarin deze functies gebruikt worden om het debiet door een open kanaal te berekenen. In een open kanaal wordt het waterdebiet Q (in m^3s^{-1}) berekend met de formule:

$$Q = \frac{A^{5/3} \cdot S_0^{1/2}}{n \cdot P^{2/3}}$$

waarbij $A = 18.0\ m^2$ de dwarsdoorsnede is van het water, $S_0 = 0.001$ de relatieve helling, $n = 0.015$ een getal afhankelijk van het soort kanaal en de wand, $P = 13.94\ m$ de zogenaamde natte omtrek.

Bereken het waterdebiet Q tot op 1 cijfer na de komma en geef je resultaat weer op het scherm gevolgd door zijn eenheid. Wanneer je dit script uitvoert, moet de volgende informatie op het scherm verschijnen:

```
Q = 45.0 m^3/s
```

Uitbreiding:

Als alternatief op de ingebouwde functie `pow()` kan je uiteraard ook de machtverheffingsoperator (`**`) gebruiken. Kopieer en herschrijf de expressie voor het berekenen van `Q` door gebruik te maken van `**` i.p.v. `pow()` voor het berekenen van de machten. Let op de voorrangsregels en gebruik haakjes (cf. Opdracht 3.15)! Je resultaat zou identiek moeten zijn.

[10] `None` is in Python een bijzonder object dat je het makkelijkst kan interpreteren als *niets*.

Vóór je begint:

- Lees Sectie 1.5 en Subsectie 1.5.1 aandachtig.
- Bekijk de specificaties van de functies `pow()` en `round()`. Je kan dit doen door `help(pow)` in te typen in de console. Informatie over een functie kan je ook online zoeken op bv. `www.google.com` met de zoekterm *python 3 pow*.

Opdracht 3.20 (`dieselmotor.py`)

In een dieselmotor ontsteekt de brandstofnevel door temperatuurstijging als gevolg van samendrukking door een zuiger in een cilinder. Zij V_i het initieel volume van de cilinder vóór de samendrukking bij een temperatuur $T_i = 293\ K$, en zij V_f het finaal volume van de cilinder na de samendrukking bij een temperatuur $T_f = 773\ K$ is. Indien je voor zo'n proces de volgende uitdrukking mag gebruiken: $T_i V_i^{\gamma-1} = T_f V_f^{\gamma-1}$, met $\gamma = 1.4$, bereken dan de verhouding V_i/V_f.

- Leid een uitdrukking af voor het berekenen van de verhouding V_i/V_f:

$$\frac{V_i}{V_f} = \ldots\ldots\ldots\ldots\ldots\ldots$$

- Schrijf een script dat deze verhouding berekent en het resultaat op het scherm toont afgerond **zonder cijfers na de komma**. Wanneer je dit script uitvoert, moet de volgende informatie op het scherm verschijnen:

```
Vi/Vf = 11
```

3.5.3 De functie `print()`

De functie `print()` maakt het mogelijk om in code aan te geven dat objecten of hun waarden op het scherm moeten worden weergegeven. Deze functie kan één argument aanvaarden:

```
>>> print("Hallo")
Hallo
```

of meerdere argumenten:

```
>>> print("een", "twee", "drie")     # drie argumenten (gescheiden door ,)
een twee drie
```

Indien meerdere argumenten (van het type `str`) worden meegegeven als argumenten, worden deze naast elkaar op het scherm getoond. **Merk op dat automatisch een spatie wordt tussengevoegd als separator.** Ook numerieke objecten kunnen deel uitmaken van de argumenten.

```
>>> print("Ik ben", 45, "jaar oud.") # drie argumenten (gescheiden door ,)
Ik ben 45 jaar oud.
```

Merk op dat 45 in de output als een string wordt weergegeven en niet langer numeriek is.

Via de **help** bestuderen we de function signature van **print()**.

```
>>> help(print)
Help on built-in function print in module builtins:

print(...)
   print(value, ..., sep = " ", end = "\n", file = sys.stdout, flush = False)

   Prints the values to a stream, or to sys.stdout by default.
   Optional keyword arguments:
   file:  a file-like object (stream); defaults to the current sys.stdout.
   sep:   string inserted between values, default a space.
   end:   string appended after the last value, default a newline.
   flush: whether to forcibly flush the stream.
```

Merk op dat de documentatie vrij beperkt is. Laten we daarom enkele parameters wat meer in detail bekijken.

De meest gebruikte parameters zijn:

- `value`: stelt de waarde(n) voor die geprint zullen worden. Met ... geeft men aan dat het aantal *values* dat kan geprint worden **variabel** is (zoals uit de voorbeelden ook blijkt).
- `sep`: bepaalt wat de *separator* (het scheidingsteken) is tussen de values die geprint worden. De default waarde is één spatie `" "`, maar deze kan een **str** naar keuze zijn.
- `end`: is een **str** die geprint wordt nadat alle values zijn geprint. De default value is `"\n"` en stelt één nieuwelijnkarakter voor. Dit wil zeggen dat, indien meerdere print statements na elkaar worden uitgevoerd, elk statement op een **nieuwe regel** zal starten.
- `file`: bepaalt de *stream* waarnaar wordt geprint. De default waarde `sys.stdout` stelt (meestal) het scherm voor, maar men kan deze ook vervangen door een file object (zie Hoofdstuk 9) zodat de values worden weggeschreven naar een bestand.

Het onderstaande codefragment illustreert een aantal mogelijkheden:

```
1 a = "een"
2 b = "twee"
3 c = "drie"
4 print(a, b, c)
5 print(a, b, c, sep = "x")   # 'x' als scheidingsteken
6 print("---------")
7 print(a, b, c, end = "xx") # 'xx' op het einde
8 print("---------")
```

De output van dit codefragment is:

```
1 een twee drie
2 eenxtweexdrie
3 ---------
4 een twee driexx---------
```

Merk op dat in regel 2 van de output de woorden gescheiden worden door x'en (in plaats van spaties). Omdat bij de eerste drie `print`-statements de optie `end` de default waarde `"\n"` heeft, wordt telkens een nieuwe regel gestart voor de volgende printopdracht. Bij het voorlaatste `print`-statement is `end = "xx"`. Dit heeft tot gevolg dat de karakters `xx` **achteraan** worden toegevoegd. Bovendien wordt **geen nieuwe lijn gestart**. Daarom wordt de streepjeslijn die volgt op dezelfde lijn toegevoegd.

Opdracht 3.21 (`printoefening.py`)
Geef de volgende instructies in in een script, maar vul de … aan zodat de output van het script dezelfde is als de onderstaande.

```
a = "een"
b = "twee"
c = "drie"
print(a, b, c, sep = ......, end = .......)
print(a, b, c, sep = ......, end = .......)
print("-einde-")
```

```
een+twee+drie = een plus twee plus drie!
-einde-
```

Opdracht 3.22

- Welke **foutmelding** verschijnt er bij het uitvoeren van de onderstaande instructie?

  ```
  print("de" 3, "biggetjes")
  ```

  Foutmelding: .....
  Pas het `print`-statement aan zodat er geen foutmelding meer verschijnt.

  .....

- Wat verschijnt op het scherm bij het uitvoeren van volgende instructie (en waarom)?

  ```
  print("Het is\n mooi weer vandaag")
  ```

  .....

3.5.4 De functie `len()`

Een string is een opeenvolging van karakters. Vaak is het handig om te kunnen bepalen uit hoeveel karakters een string bestaat. De functie **`len()`** biedt deze functionaliteit aan[11]. Ook spaties `" "`, leestekens zoals komma's `","` bijvoorbeeld, en nieuwelijnkarakters `"\n"` zijn karakters en dragen dus ook bij tot de lengte.

```
>>> len("abc")  # lengte van de string "abc"
3
```

```
>>> len("Wie niet sterk is, moet slim zijn.")  # lengte van de string
34
```

Opdracht 3.23 (`lenoefening.py`)
Bepaal (eerst zonder Python) het resultaat van de volgende expressies:

- `len("het is\nmaandag")`
- `len("abc") * 2 * "abc"`

3.6 Python modules

3.6.1 Modules in Python

Naast een beperkt aantal ingebouwde (*built-in*) functies bevat de *Python Standard Library*[12] een groot aantal functies die worden aangeboden via **modules** en **packages**. Een module kan, voorlopig, beschouwd worden als een verzameling van ondermeer functiedefinities die door de programmeur kunnen gebruikt worden. Een package is dan weer een verzameling van modules. Vaak situeren de functies in een module zich **rond eenzelfde topic**. Zo zijn er bijvoorbeeld modules die voornamelijk elementaire) wiskundige functies bevatten (zoals de module `math`) of om multidimensionale array objecten aan te maken (zoals de module `numpy`) of functies om webpagina's in te laden (zoals de module `urllib`) of functies om te werken met datums en tijd (zoals de module `time`)[13]. Om de functionaliteiten van een module te kunnen gebruiken, moet deze eerst **geïmporteerd** worden. Dit kan met een **`import`**-statement. Dit is een statement gevormd door het keyword **`import`** gevolgd door de naam van de module.

[11] Later zullen we zien dat **`len()`** ook de kan gebruikt worden om de lengte van andere types te bepalen zoals lists, zie Hoofdstuk 7.

[12] De Python Standard Library is de bibliotheek die bij alle installaties van de Python interpreter wordt geïnstalleerd. De programmeur kan er dus van uit gaan dat alle gegevenstypes, functies en modules die tot de *standard library* behoren beschikbaar zijn op het systeem waarop de code zal worden uitgevoerd.

[13] De beschikbare *modules* zijn te vinden via de zgn. *Python Module Index* op `https://docs.python.org/3/py-modindex.html`

Deze **`import`-statements** worden vaak **bovenaan in het broncodebestand** geplaatst. Expressies en statements die onder deze imports staan, kunnen gebruikmaken van deze modules.
Enkele voorbeelden van `import`-statements.

```
import math         # importeren math library
import numpy        # importeren numpy library
import time         # importeren time library
```

De functies die via deze modules beschikbaar zijn, kunnen worden opgeroepen door hun naam te laten voorafgaan door de naam van de module waartoe ze behoren, gevolgd door een punt. Zo kunnen de functies `sin()` en `cos()`, die behoren tot de `math` module en gebruikt worden om de sinus en cosinus van een hoek (in radialen) te berekenen, als volgt worden opgeroepen.

```
import math         # importeren math library
a = math.sin(0.6)   # oproepen van de functie sin, hoek 0.6 in radialen
b = math.cos(0.8)   # oproepen van de functie cos, hoek 0.8 in radialen
print(a, b)
```

Dit codefragment heeft als output:

```
0.5646424733950354 0.6967067093471654
```

Gebruik van modules

- **importeren** van een module:

  `import` *naam_module*

- **functie-oproep** naar een functie uit die module:

  naam_module.*naam_functie*(*argument1*, *argument2*, ...)

3.6.2 Voorbeeld 1: de module `math`

De module `math` bevat een aantal elementaire wiskundige functies en constanten. De volledige lijst kan teruggevonden op `https://docs.python.org/3/library/math.html`.

In wat volgt, geven we een kort overzicht van enkele veelgebruikte functies uit die module:

- Functies uit de getallenleer, bv. `floor()`, `ceil()`
- Machtsfuncties, logaritmische functies, en exponentiële functies, bv. `pow()` (machtsverheffing), `log()` (natuurlijke logaritme), `log10` (logaritme met grondtal 10), `exp()` (macht met grondtal `e`)
- Trigonometrische functies, bv. `sin()`, `asin()`

- Omzetting van hoeken, bv. `degrees()`, `radians()`
- Hyperbolische functies, bv. `cosh()`, `acosh()`
- Wiskundige constanten, bv. `pi` (π), `e` (≈ 2.7182818)
- Bijzondere functies, bv. `erf()`, `gamma()`

De signatuur (*signature*) van elke van deze functies kan worden opgevraagd met de functie **`help()`**.

Hieronder wordt ter illustratie de sinus berekend van $\pi/3$:

```
>>> import math           # importeren math library
>>> math.sin( math.pi/3 ) # hoek als argument van sin staat in radialen
0.8660254037844386
```

Wensen we de sinus van een hoek in graden te berekenen met de `sin` functie, dan moeten we die eerst in radialen omzetten[14] met de functie `radians()`:

```
>>> import math
>>> math.sin( math.radians(60) )
0.8660254037844386
```

Een hoek in radialen omzetten naar een hoek in graden kan met de functie `degrees()`:

```
>>> import math
>>> math.degrees( math.pi/3 )
59.99999999999999
```

Het volgende codefragment illustreert het gebruik van de `math` module in een uitgebreider script, waarin het geluidsniveau (in decibel, dB) berekend wordt voor een gemeten geluidsintensiteit $I = 10^{-4} Wm^{-2}$. De referentie-intensiteit $I_0 = 10^{-12} Wm^{-2}$. Het geluidsniveau in dB wordt gedefinieerd als

$$B = 10 \log \left(\frac{I}{I_0} \right).$$

Fragment 3.3: `geluidsniveau_db.py`

```
1 import math
2
3 I0 = 1.0e-12    # referentie intensiteit
4 I  = 1.0e-4     # gemeten intensiteit
5
6 B = 10 * math.log10(I / I0)   # geluidsniveau in dB
7
8 print("De geluidsintensiteit is", I, "W/m^2")
9 print("Het geluidsniveau is", B, "dB")
```

[14] Het verband tussen een hoek in radialen en een hoek in graden is $rad = graden \times \frac{\pi}{180}$.

Dit levert het volgende resultaat

```
De geluidsintensiteit is 0.0001 W/m^2
Het geluidsniveau is 80.0 dB
```

Merk op dat de functie `log10()` de **logaritme berekent met grondtal 10** (cf. $log()$). De functie `log()` **berekent de natuurlijke logaritme** (cf. $ln()$).

Opdracht 3.24 (`hellingshoek.py`)

De hellingsgraad is een maat om de steilheid van een hellend vlak weer te geven. De hellingsgraad wordt veelal uitgedrukt in percentages (%). De hellingsgraad is gelijk aan het hoogteverschil Δh gedeeld door de horizontale afstand d maal 100%:

$$\text{hellingsgraad} = \frac{\Delta h}{d} \times 100\%.$$

Een andere maat voor de steilheid is de hellingshoek (meestal uitgedrukt in graden °) die het wegdek maakt met het horizontale vlak. Deze kan je bekomen door de inverse van de tangens (de functie `atan()`) te nemen van $\Delta h/d$:

$$\text{hellingshoek} = \tan^{-1}\left(\frac{\Delta h}{d}\right)$$

Schrijf een script `hellingshoek.py` waarin je voor 3 hellingsgraden (5, 10 en 15%) de overeenstemmende hellingshoeken berekent. Wanneer je dit script uitvoert, moet de volgende info op het scherm verschijnen.

```
De hellingsgraad is 5.0 %
De overeenstemmende hellingshoek is 2.9 graden
--------------------------------------------------
De hellingsgraad is 10.0 %
De overeenstemmende hellingshoek is 5.7 graden
--------------------------------------------------
De hellingsgraad is 15.0 %
De overeenstemmende hellingshoek is 8.5 graden
--------------------------------------------------
```

Tip: Met `print(60*'-')` kunnen een 60-tal koppeltekens - naast elkaar geprint worden (zie 3.3.3).

Vóór je begint:

- Lees Sectie 3.6 aandachtig.
- Gebruik Codefragment 3.3 als voorbeeld.

Opdracht 3.25 (`brekingsindex.py`)
Een monochromatische lichtstraal gaat over van lucht naar een bepaalde glassoort. De invalshoek in $\theta_1 = 43^\circ$ en de brekingshoek is $\theta_2 = 26^\circ$

$$n_{glas} = \frac{\sin(\theta_1)}{\sin(\theta_2)}.$$

Schrijf een script `brekingsindex.py` waarin je de invalshoek en brekingshoek definieert en de brekingsindex berekent. Wanneer je dit script uitvoert, moet de volgende info op het scherm verschijnen:

```
theta1 = 43.0 graden (invalshoek)
theta2 = 26.0 graden (brekingsshoek)
De brekingsindex van het glassoort:
n_glas = 1.5558
```

Vóór je begint

- Lees Sectie 3.6 aandachtig.
- Gebruik Codefragment 3.3 als voorbeeld.

3.6.3 Voorbeeld 2: de module `random`

De module `random` kan gebruikt worden om (pseudo-)random getallen te genereren uit een groot aantal distributies. Een voorbeeld is de simulatie van een worp met een dobbelsteen. Het resultaat van een worp is een geheel getal tussen 1 en 6 (**grenzen inbegrepen**, type `int`), waarbij elk getal eenzelfde kans op voorkomen moet hebben. De functie `randint` uit de `random` module kan daarvoor gebruikt worden.

We importeren eerst de random module en bekijken de documentatie van `randint`.

```
>>> import random
>>> help(random.randint)
Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
```

Uit deze beschrijving blijkt dat deze methode twee argumenten aanvaardt en een integer retourneert tussen `a` en `b` (beide inbegrepen).

```
>>> worp = random.randint(1, 6)
>>> print(worp)
5
```

3.7 Typeconversie

De waarden (strikt genomen noemen we dit *literals*) `5`, `5.0` en `"5"` stellen allen het getal vijf voor. De literal `5` zal door Python echter geïnterpreteerd worden als een integer waarde, `5.0` als een floating point waarde en `"5"` als een string. Men kan echter een object van het ene type omzetten in een (nieuw) object van een ander type met dezelfde waarde[15]. Een dergelijke omzetting noemen we een **typeconversie**.

De functies `int()`, `float()` en `str()` kunnen gebruikt worden om een object om te zetten naar respectievelijk een integer, float of string. Het **oorspronkelijke object** wordt daarbij **niet gewijzigd**.

We illustreren het gebruik van deze functies hieronder:

```
>>> a = "5"
>>> b = int(a)
>>> c = float(a)

>>> type(a)     # type van a blijft str
str
>>> type(b)     # type van b is int
int
>>> type(c)     # type van c is float
float

>>> b
5               # waarde van b is 5
>>> c
5.0             # waarde van c is 5.0
```

```
>>> d = 5**(1/2)
>>> str(d)
'2.23606797749979'    # aantal decimalen kan voorspeld
                      # worden, maar niet triviaal
```

Merk op dat het object dat moet omgezet worden eenvoudig wordt meegegeven als argument aan de functie die de typeconversie uitvoert.

Met behulp van typeconversie kan het `print`-statement `print("de" 3, "biggetjes")` uit Opdracht 3.22 ook als volgt verbeterd worden:

```
>>> print("de " + str(3) , "biggetjes")   # extra spatie na "de"
de 3 biggetjes
```

[15] In sommige gevallen kan, bv. door afronding of een beperking in het aantal decimalen, de geconverteerde waarde ietwat afwijken van de oorspronkelijke.

Opmerkingen:

- Typeconversie is echter niet steeds mogelijk, zo kan een een string die geen geheel getal voorstelt niet geconverteerd worden naar een `int`:

```
>>> int("hallo")    # str stelt geen getal voor
ValueError: invalid literal for int() with base 10: 'hallo'

>>> int("5.1234")   # str stelt geen wel getal voor, maar niet geheel
ValueError: invalid literal for int() with base 10: '5.1234'
```

- Anderzijds kan de waarde na conversie sterk afwijken van de oorspronkelijke waarde:

```
>>> int(5.123)   # decimaal deel valt weg
5
>>> int(5.789)   # decimaal deel valt weg
5                # volgt NIET de afrondingsregels
```

Opdracht 3.26 (`typeconversieOefening.py`)
Vul het onderstaande codefragment aan met de nodige functies voor typeconversie (gebruik tevens de functie `round()` zodat de weergegeven output correct op het scherm verschijnt.

```
temp_juni = 16.57
temp_juli = "18.66"

b1 = "Gemiddelde temperatuur juni: " + ..................... + " graden."

b2 = "Na afronding is dit " + ............................. + " graden."

b3 = "Gemiddelde temperatuur juli: " + ..................... + " graden."

b4 = "Na afronding is dit " + ............................. + " graden."

print(b1, b2, b3, b4, sep = "\n")
```

Output:

```
Gemiddelde temperatuur juni: 16.57 graden.
Na afronding is dit 16.6 graden.
Gemiddelde temperatuur juli: 18.66 graden.
Na afronding is dit 18.7 graden
```

3.8 Input van keyboard

Beschouw het volgende script waarin de snelheid van een voorwerp, uitgedrukt in kmh^{-1} wordt omgezet naar een snelheid in ms^{-1}:

```
snelheid_km_h = 120.0
snelheid_m_s = round(120 / 3.6, 2)
print("De snelheid in SI-eenheden is:", snelheid_m_s, "m/s")
```

Merk op dat de programmeur hier zelf de waarde 120.0 **heeft gekozen.** Zonder de broncode te wijzigen is het niet mogelijk om deze omzetting ook uit te voeren voor andere snelheden.

De functie `input()` laat toe om, bij het uitvoeren van een script, de gebruiker om input te vragen. Deze input kan worden ingevoerd via het keyboard. Wanneer de input-functie wordt uitgevoerd zal de gebruiker een boodschap te zien krijgen op het scherm. De uitvoer wordt vervolgens onderbroken tot de gebruiker input heeft ingevoerd via het keyboard. Wanneer de gebruiker op de Enter-toets drukt wordt de uitvoer weer hervat.

Beschouw ter illustratie het volgende codefragment.

```
woord = input("Typ een woord: ")
print("Je voerde", woord, "in.")
```

Bij het uitvoeren van dit fragment verschijnt de volgende output.

```
Typ een woord in: Snackbar    # gebruiker voert Snackbar in en drukt Enter
Je voerde Snackbar in.        # print-statement wordt uitgevoerd
```

Zoals het voorbeeld illustreert, heeft de functie `input()` de volgende signatuur:

```
returnValue = input( prompt )
```

met

- `input`: de naam van de functie,
- `prompt`: de (enige) parameter met de boodschap die de gebruiker te zien krijgt bij de vraag om input (type `str`), en
- `returnValue`: de *return value*, een object met daarin de gebruikersinput (type `str`).

Belangrijke opmerking

Gebruikersinput die numeriek is van aard (getallen) wordt door de functie `input()` ook als een `str` geretourneerd! We moeten deze waarden dus converteren naar `int` of `float` om er bv. rekenkundige bewerkingen op uit te kunnen voeren:

```
>>> getal_str = input("Typ een getal: ")
Typ een getal: 56.3
>>> type(getal_str)
```

```
str                                     # getal_str is van type str
>>> getal_float = float(getal_str)      # converteer naar float
>>> type(getal_float)                   # getal_float is van type float
float
```

Het onderstaande script laat de gebruiker toe om zelf een snelheid in te geven in kmh^{-1}, zet deze om in ms^{-1} met twee cijfers na de komma en geeft deze weer op het scherm.

Fragment 3.4: `snelheid_km_ms.py`

```
1 # vraag naar de snelheid in km/h
2 snelheid_km_str = input("Geef een snelheid in km/h: ")
3 # converteer str naar float
4 snelheid_km_float = float(snelheid_km_str)
5 # zet om naar m/s en rond af tot 2 cijfers na de komma
6 snelheid_ms_float = round(snelheid_km_float / 3.6, 2)
7 # print het resuiltaat naar het scherm
8 print("De snelheid in SI-eenheden is:", snelheid_ms_float, "m/s")
```

Dit levert het volgende resultaat indien je `120` ingeeft als snelheid

```
Geef een snelheid in km/h: 120
De snelheid in SI-eenheden is: 33.33 m/s
```

De volgende twee (kortere) coderingsmogelijkheden hebben dezelfde output.

Fragment 3.5: `snelheid_km_ms.py`

```
1 snelheid_km_float = float(input("Geef een snelheid in km/h: "))
2 snelheid_ms_float = round(snelheid_km_float / 3.6, 2)
3 print("De snelheid in SI-eenheden is:", snelheid_ms_float, "m/s")
```

Fragment 3.6: `snelheid_km_ms.py`

```
1 snelheid_ms_float = \
2 round(float(input("Geef een snelheid in km/h: ")) / 3.6, 2)
3 print("De snelheid in SI-eenheden is:", snelheid_ms_float, "m/s")
```

Instructies spreiden over meerdere regels

Soms is een instructie te lang om op één regel te passen (zie bv. Fragment 3.6). Om de leesbaarheid te vergroten, kan het interessant zijn om de instructie te spreiden over meerdere regels. De **backslash** (\) wordt gebruikt om een instructie op een volgende regel verder te zetten.

Zo is het statement

```
x = 2 + 5
```

hetzelfde als

```
x \
= 2 \
+ 5
```

Opdracht 3.27 (`schuine_zijde.py`)

Implementeer een script dat achtereenvolgens de lengtes vraagt van de rechthoekszijden van een rechthoehige driehoek (op een interactieve manier m.b.v. de functie `input()`), de lengte van de schuine zijde berekent en deze vervolgens op het scherm weergeeft. Gebruik `sqrt()` en `pow()` uit de `math`-module. Een mogelijke output is:

```
Geef de lengte van de rechthoekszijde 1: 3
Geef de lengte van de rechthoekszijde 2: 4
De lengte van de schuine zijde is: 5.0
```

Vóór je begint:

- Lees Sectie 3.8 aandachtig.
- Gebruik codefragment 3.4 als voorbeeld.

Opdracht 3.28 (`haversine_toepassen.py`)

De Haversine formule bepaalt de kortste afstand tussen twee posities op een sfeer, langs het oppervlak van de sfeer, bij gegeven latitude en longitude coördinaten. Overeenkomstig de conventie gebruiken we de symbolen ϕ voor latitude en λ voor longitude. Indien (ϕ_1, λ_1) en (ϕ_2, λ_2) de latitude en longitude coördinaten (in decimale graden $^\circ$) zijn van twee posities op Aarde met (straal $R = 6371\ km$) dan wordt de afstand d (in kilometers) bepaald door:

$$\begin{aligned} a &= \sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos\phi_1 \cdot \cos\phi_2 \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right) \\ c &= 2 \cdot \arctan\left(\frac{\sqrt{a}}{\sqrt{1-a}}\right) \\ d &= R \cdot c \end{aligned}$$

Schrijf een script dat achtereenvolgens de latitude en longitude coördinaten (in graden) vraagt van twee posities en vervolgens de afstand d berekent volgens bovenstaande Haversine formule. Een mogelijke output is:

```
Geef latitude  van positie 1: 50.85
Geef longitude van positie 1: 4.35
Geef latitude  van positie 2: 55.75
Geef longitude van positie 2: 37.62
De afstand tussen ( 50.85, 4.35 ) en ( 55.75, 37.62 ) is 2253.28 km
```

Vóór je begint:

- Lees Sectie 3.6 aandachtig.
- Zet de latitude en longitude coördinaten om naar radialen.
- Bereken `a`, bereken daarna `c` en bereken tenslotte de afstand `d`.

3.9 Het gegevenstype `bool` en logische expressies

3.9.1 Proposities of beweringen

Een **propositie** of bewering is een declaratieve zin die **waar** (`True`) of **vals** (`False`) kan zijn. De volgende proposities zijn **waar**:

'Een schaap is een zoogdier.'

'Het getal 5 is groter dan het getal 2.'

'Na het uitvoeren van de toekenningsstatements `x = 2` en `y = 2` zijn de waarden waar `x` en `y` naar verwijzen gelijk.'

De volgende proposities zijn daarentegen **vals**:

'Een paard is een vis.'

'Het getal 7 is kleiner dan het getal 3.'

Het al dan niet waar (`True`) of vals (`False`) zijn van een propositie noemt men de **waarheidswaarde** van de propositie.

3.9.2 Het gegevenstype `bool`

Het aangeven van de waarheidswaarde van een expressie gebeurt in Python met het gegevenstype `bool` (afkomstig van *boolean*). Dit is een gegevenstype dat maar twee mogelijke waarden onderscheidt: `True` en `False` (let op de hoofdletters `T` en `F`). Expressies die als resultaat een object van het type `bool` hebben, noemt men **logische expressies**. Logische expressies bevatten (nagenoeg) altijd vergelijkingsoperatoren of relationele operatoren. Dit zijn operatoren die de relatie tussen twee objecten bekijken. Zo kan bekeken worden of de waarden van twee objecten gelijk zijn aan elkaar met de `==` operator of kan men bekijken of een waarde groter is dan een andere waarde met de `>` operator. Een compleet overzicht wordt gegeven in Tabel 3.5.

Tabel 3.5: Tabel met relationele operatoren.

Operator	Betekenis
<	kleiner dan
<=	kleiner dan of gelijk aan
>	groter dan
>=	groter dan of gelijk aan
==	gelijk aan (vergelijkingsoperator)
!=	niet gelijk aan (verschillend van)

Merk het verschil op tussen de **vergelijkingsoperator** == en de **toekenningsoperator** =. De vergelijkingsoperator == wordt gebruikt om te controleren of twee waarden gelijk zijn aan elkaar. De toekenningsoperator = wordt gebruikt om een waarde toe te kennen aan een variabele.

Het onderstaande codefragment illustreert het gebruik van deze operatoren voor numerieke operanden (type `float` en `str`).

```
>>> 5 > 3
True
>>> 6.0 > 10.0
False
>>> 7 == 7
True
```

Het resultaat van een logische expressie kan men toekennen aan een nieuwe variabele.

```
>>> a = 5 >= 3
>>> print(a)
True
>>> type(a)
bool
```

De literals `True` en `False` zijn tevens Python keywords die kunnen gebruik worden om een object van het type `bool` aan te maken.

```
>>> a = True
>>> print(a)
True
>>> type(a)
bool
```

Opdracht 3.29

Controleer in een Python console welke waarde geretourneerd wordt door de volgende logische expressies:

- `3 > 2`
- `5+3 < 3-2`
- `'1' < 2`
- `1 + 2 == 3`
- `1.0 + 2.0 == 3.0`
- `1.1 + 2.2 == 3.3`

Kan je verklaren waarom `1.1 + 2.2 == 3.3` de waarde `False` teruggeeft?
Tip: bekijk eens het resultaat van `1.1 + 2.2` afzonderlijk.

3.9.3 Samengestelde proposities

De voorgaande proposities waren steeds enkelvoudige proposities. Men kan deze proposities echter samenstellen door gebruik te maken van voegwoorden. Deze voegwoorden hebben in deze samengestelde proposities de rol van een operator, we noemen deze dan ook **booleaanse operatoren**.

Voegwoord EN (Engels: `and`).
De propositie

$$\underbrace{\text{Een schaap is een zoogdier}}_{p} \ \textbf{en} \ \underbrace{\text{het getal 5 is groter dan het getal 2}}_{q}$$

is een voorbeeld van een samengestelde propositie, waarbij p en q werden samengesteld met het voegwoord **en**. We noemen dit de *conjunctie* van p en q. Omdat zowel p als q waar zijn, is ook de conjunctie p en q waar.

De conjunctie van twee proposities is waar indien beide samenstellende delen waar zijn. Zodra minstens één van deze delen vals is, is de conjunctie vals. Dit wordt samengevat in waarheidstabel 3.6.

Tabel 3.6: Waarheidstabel van de **`and`** operator.

p	q	p **and** q
True	True	True
True	False	False
False	True	False
False	False	False

De conjunctie wordt in Python geïmplementeerd door de **`and`** operator.

```
>>> a = (5 > 3) and (7 > 1)   # True and True -> True
>>> print(a)
True
```

```
>>> b = (5 == 7) and (9 < 11) # False and True -> False
>>> print(b)
False
```

Voegwoord OF (Engels: `or`).
De propositie

$$\underbrace{\text{Een schaap is een vis}}_{p} \ \textbf{of} \ \underbrace{\text{het getal 5 is groter dan het getal 2}}_{q}$$

is een voorbeeld van een samengestelde propositie, waarbij p en q werden samengesteld met het voegwoord **of**. We noemen dit de *disjunctie* van p en q. Omdat minstens één van p en q waar zijn, is ook de disjunctie p of q waar.

De disjunctie van twee proposities is waar indien minstens één van beide samenstellende delen waar zijn. Indien beide samenstellende delen vals zijn, is de disjunctie ook vals. Dit wordt samengevat in waarheidstabel 3.7.

Tabel 3.7: Waarheidstabel van de `or` operator.

p	q	p or q
True	True	True
True	False	True
False	True	True
False	False	False

De disjunctie wordt in Python geïmplementeerd door de `or` operator.

```
>>> a = (5 < 3) or (7 > 1)      # False or True -> True
>>> print(a)
True

>>> b = (5 != 5) or (9 >= 11)  # False or False -> False
>>> print(b)
False
```

Negatie[16]: NIET (Engels: `not`).
De propositie

$$\underbrace{\text{het getal 7 is}}_{p} \ \textbf{niet} \ \underbrace{\text{groter dan het getal 10}}_{p}$$

is een voorbeeld van een propositie die een negatie bevat. De negatie van een propositie p is waar indien p vals is (en omgekeerd). Dit wordt samengevat in waarheidstabel 3.8.

[16] `not` is strikt genomen geen voegwoord.

Tabel 3.8: Waarheidstabel van de **not** operator.

p	not p
True	False
False	True

De negatie wordt in Python geïmplenteerd door de **not** operator.

```
>>> 5 > 7
False
>>> not (5 > 7)
True
```

3.9.4 Combineren van numerieke en logische operatoren

Rekenkundige (bv. +, /), relationele (bv. >, ==) en booleaanse operatoren (**and**, **or**, **not**) kunnen samen voorkomen in een expressie. Bij het evalueren van deze expressies worden voorrangsregels gevolgd die worden weergegeven in Tabel 3.9.

Tabel 3.9: Samenvatting voorrangsregels van hoogste naar laagste prioriteit.

Operator	Beschrijving
()	Haakjes
**	Machtsverheffing
+x, -x	Unaire plus/min
*, /, //, %	Vermenigvuldiging, Deling, Rest
+, -	Som, Verschil
<, <=, >, >=, !=, ==	Vergelijking
not x	*Boolean* NOT
and	*Boolean* AND
or	*Boolean* OR

In de onderstaande expressie zijn a, …, g variabelen die verwijzen naar numerieke objecten.

```
a < b and c + d == e or f <= g
```

In deze expressie komen drie types operatoren voor. De volgende expressie heeft hetzelfde resultaat, maar maakt gebruik van haakjes die de prioriteit van de operatoren benadrukken.

```
((a < b) and (c + d == e)) or (f <= g)
```

Opdracht 3.30

Ga het resultaat na van de volgende expressies

- `(0 <= a_int) and (a_int <= 5)` waarbij `a_int = 5`
- `(X > 2) or (X > 5)` waarbij `X = 3`
- `2 > 5 or 6 * 2 <= 12 and 7 > 3`
- `True and False or True and False`

Algemene tip: gebruik haakjes, maar overdrijf niet in het gebruik ervan!

3.9.5 Relationele operatoren voor operanden van het type string

De operanden van relationele operatoren kunnen in Python ook van het type **`str`** zijn. Zo kan men van **twee strings nagaan of ze gelijk zijn** met de `==` operator:

```
>>> "Hallo" == "Test"
False                    # strings hebben verschillende waarde
>>> "Hallo" == "Hallo"
True                     # strings hebben dezelfde waarde
>>> "Hallo" == "hallo"
False                    # HOOFDLETTERGEVOELIG
```

De operatoren `<`, `>`, `<=` en `>=` kunnen ook inwerken op operanden van het type **`str`**. Bij het vergelijken van strings wordt gewerkt volgens de **regels** die gebruikt worden bij het **lexicografische ordening van woorden**, maar met een **uitgebreid alfabet** dat naast letters ook karakters zoals komma's, punten, enz. bevat. De plaats van deze karakters in het alfabet wordt gepaald door hun ASCII/Unicode code[ab]. Leestekens en cijfers hebben een eerder lage ASCII/Unicode code (de spatie heeft de laagste ASCII code van de printbare karakters), gevolgd door hoofdletters (volgens alfabet) en tenslotte kleine letters (volgens alfabet).

```
>>> "olifant" > "muis"
True                 # o heeft hogere ascii waarde dan m (idem alfabet)
>>> "kip" < "ei"
False                # k heeft hogere ascii code dan e (idem alfabet)
>>> "Kip" < "ei"
True                 # K (hoofdletter) heeft lagere ascii code dan e
>>> "plaats" < "plas"
True                 # a (positie 4) heeft lagere ascii code dan s
                     #                                (idem alfabet)
```

[a]Meer precies wordt Unicode gebruikt, die een uitbreiding van ASCII (die alle gangbare karakters bevat) is

en waarbij de eerste 128 *code points* de ASCII karakters bevatten.
[b]Voor een volledige lijst zie `https://en.wikipedia.org/wiki/ASCII`.

Het volgende voorbeeld toont de invloed van spaties, die een zeer lage ASCII-waarde hebben:

```
>>> "Paul Van den Berghe" < "Paul Vanden Berghe"
True                # spatie tussen 'Van den' heeft lage ASCII
```

Indien men het Unicode *code point* expliciet wenst op te vragen kan dit met de functie `ord()`.

```
>>> ord("a")
97
>>> ord("b")
98
>>> ord("A")
65
>>> ord(" ")   # spatie heeft lage ASCII/unicode
32
```

3.9.6 Typeconversie

Objecten van het gegevenstype `bool` kunnen geconverteerd worden naar numerieke objecten of stringobjecten, met de vertrouwde functies voor typeconversie `int`, `float` en `str`. Bij dergelijke conversies, die in eenvoudige Python code niet vaak voorkomen, worden volgende regels gebruikt.

- `True` wordt bij omzetting naar een numeriek gegevenstype 1 (één).
- `False` wordt bij omzetting naar een numeriek gegevenstype 0 (nul).
- Typeconversie van de booleans `True` en `False` naar een string resulteert in de strings `'True'`, resp. `'False'`.

```
>>> int(True)
1
>>> str(False)
'False'
```

Typeconversie naar een object van het gegevenstype `bool` kan met de functie `bool()`. Daarbij worden voor alle numerieke gegevenstypes **niet-nul waarden** geconverteerd naar `True`. **Nullen** worden geconverteerd naar `False`.

```
>>> bool(12)
True
>>> bool(0.0)
False
```

Indien een string object geconverteerd wordt naar een `bool` object, dan zal het resultaat `True` zijn:

```
>>> bool("hallo")
True
```

De **enige uitzondering** is de **lege** string `" "` die wordt geconverteerd naar `False`:

```
>>> bool("")
False
```

Opdracht 3.31 (`even_getal.py`)

Een manier om na te gaan of een geheel getal (*integer*) even is, bestaat erin dit getal te delen door 2 en de rest te bekijken. Schrijf een programma `even_getal.py` dat:

- De gebruiker vraagt een geheel getal in te geven.
- Controleert of dit getal kleiner is dan of gelijk aan 100.
- Controleert of dit getal even is.
- Indien beide voorwaarden voldaan zijn de waarde `1` op het scherm toont. Indien minstens één voorwaarde niet voldaan is, moet `0` op het scherm verschijnen.

Een mogelijke input/output is

```
Geef een geheel getal: 23
0
```

Het controleren of het getal kleiner is dan 100 en even is, en het tonen van `0` of `1` op het scherm kan in één instructie.

Vóór je begint:

- Lees Sectie 3.9.
- Creëer een variabele `a` die verwijst naar een integer object met een waarde naar keuze.
- Ken het resultaat van de logische expressie `a <= 100` toe aan variabele `b`: `b = a <= 100`.
- Voer de instructie `print(b)` uit.
- Voer de instructie `print(int(b))` uit.
- Gebruik de verworven inzichten om de opdracht uit te voeren.

3.9.7 Toekenning en typeconversie

De waarde van een logische uitdrukking kan je toekennen aan een variabele en deze waarde converteren naar een *string*.

Opdracht 3.32
Schrijf de logische expressie uit `2 + 3 <= 15 or 4 == 5` met haakjes. Stel in een interactieve sessie deze logische expressie gelijk aan een variabele. Ga na welke waarde de variabele heeft en converteer deze waarde naar een *string*.

3.10 Gemengde opdrachten

Opdracht 3.33 (`kogel_max_hoogte.py`)
Een kogel wordt verticaal omhoog geschoten met een beginsnelheid van $v_0 = 300\,ms^{-1}$. De maximale hoogte wordt berekend met:

$$h_{max} = \frac{v_0^2}{2 \cdot g}$$

waarbij $g = 9.81\,ms^{-2}$ de valversnelling is. Schrijf een programmaatje die deze hoogte berekent. Een mogelijke output is:

```
Een kogel wordt verticaal afgevuurd.
Begin snelheid:  300 m/s
Valversnelling:  9.81 m/s^2
Maximale hoogte: 4587.155963302752 m
```

Opdracht 3.34 (`rente_na_10_jaar.py`)
Een spaarrekening heeft een rente van 0.50 % per jaar. Je plaatst een initieel bedrag van €10000 op deze spaarrekening. Het bedrag A_n na n jaar kan berekend worden met de volgende formule:

$$A_n = A_0 \left(1 + \frac{p}{100}\right)^n$$

hierin is A_0 het initieel bedrag, n het aantal jaren, p de rente in % en A_n het eindbedrag. Schrijf een programma waarin je het eindbedrag na 10 jaar berekent. Een mogelijke output is:

```
Rente:           0.5 %
Initieel bedrag: 10000.0 euro
Aantal jaar:     10 jaar
Eindbedrag:      10511.4 euro
```

Opdracht 3.35 (`twee_getallen.py`)
Schrijf een programma die achtereenvolgens twee getallen vraagt en vervolgens die twee getallen, hun som, verschil, product, en quotiënt weergeeft op het scherm. Een mogelijke output is:

```
Geef het eerste getal: 9
Geef het tweede getal: 4
9.0 + 4.0 = 13.0
9.0 - 4.0 = 5.0
9.0 * 4.0 = 36.0
9.0 / 4.0 = 2.25
```

Opdracht 3.36 (`uren_minuten_seconden.py`)
Een dag bevat 86400 s ($= 24 \times 60 \times 60$ s). Schrijf een programma die een tijd in seconden tussen 0 en 86400 vraagt en deze omzet naar uren, minuten en seconden. Een mogelijke input/output is

```
Geef een tijd in seconden tussen 0 en 86400 s: 56335
56335 seconden is 15 uren, 38 minuten en 55 seconden.
```

Opdracht 3.37 (`oppervlakte_trapezium.py`)
De oppervlakte A van een trapezium wordt berekend met

$$A = \frac{1}{2} \cdot h \cdot (a + b)$$

waarbij a en b de lengtes zijn van de basis en h de hoogte. Schrijf een programma dat vraagt naar deze lengtes (in meter), de oppervlakte berekent en deze weergeeft naar het scherm. Een mogelijke input/output is

```
Geef de lengte (in m) van de ene basis: 5.0
Geef de lengte (in m) van de andere basis: 4.0
Geef de hoogte (in m): 3.0
De oppervlakte van de trapezium is 13.5 m^2
```

Opdracht 3.38 (`hoeken_driehoek.py`)
In een driehoek geldt $c^2 = a^2 + b^2 - 2 \cdot a \cdot b \cdot \cos(\hat{c})$ met a, b en c de lengtes van de zijden en $\hat{c}$ de hoek gevormd door a en b. Schrijf een programma dat de lengtes a, b en c vraagt aan de gebruiker vraagt als input. Op basis van de gebruikersinput moet eerst nagegaan worden of de ingegeven lengtes een driehoek kunnen vormen. Vervolgens moet de hoek C berekend worden. Een mogelijke input/output is:

```
Geef lengte van zijde a: 3
Geef lengte van zijde b: 7
Geef lengte van zijde c: 9
```

```
Controle geldige driehoek: True
Hoek tussen zijden a en b is: 123.2 graden

Geef lengte van zijde a: 6
Geef lengte van zijde b: 7
Geef lengte van zijde c: 1
Controle geldige driehoek: False
Hoek tussen zijden a en b is: 0.0 graden
```

Vóór je begint:

- Zoek op het internet op aan welke voorwaarde(n) a, b en c moeten voldoen opdat ze de lengtes van de zijden van een driehoek zouden kunnen zijn.
- Implementeer deze voorwaarden als een (vrij uitgebreide) logische expressie (je hebt mogelijks de logische operatoren `and`, `or` en `not` nodig).

Opdracht 3.39 (`isbn01.py`)

De International Standard Book Number (ISBN) is een unieke numerieke commerciële boek identificatienummer. Als dit nummer toegekend is aan een boek na 1 januari 2007, dan is dit ISBN-nummer 13 cijfers lang (ISBN-13). De eerste 12 daarvan geven informatie over het boek zelf, terwijl het laatste louter een controlecijfer is dat dient om foutieve ISBN-13 codes te detecteren. Indien x_1, x_2, ..., x_{12} de eerste 12 cijfers van een ISBN-13 code voorstellen, dan wordt het controle cijfer x_{13} als volgt berekend:

$$x_{13} = (10 - (x_1 + 3x_2 + x_3 + 3x_4 + x_5 + 3x_6 + x_7 + 3x_8 + x_9 + 3x_{10} + x_{11} + 3x_{12}) \mod 10) \mod 10$$

Het cijfer x_{13} kan m.a.w. een waarde aannemen tussen 0 en 9 ($0 \leq x_{13} \leq 9$). Schrijf een programma dat achtereenvolgens de eerste 12 cijfers vraagt van een ISBN-13 code en het 13de cijfer berekent en weergeeft naar het scherm. Een mogelijke input/output is:

```
9
7
8
1
2
9
2
0
2
1
0
3
Controle cijfer:  4
ISBN-13 code:  9781292021034
```

Opdracht 3.40 (`ontwikkelingsindex.py`)

De index van de menselijke ontwikkeling (ontwikkelingsindex) of Human Development Index (HDI) van de Verenigde Naties meet voornamelijk armoede, analfabetisme, onderwijs en levensverwachting in een bepaald land of gebied. De index werd in 1990 ontwikkeld door de Pakistaanse econoom Mahbub ul Haq en wordt sinds 1993 door VN-Ontwikkelingsprogramma gebruikt in haar jaarlijks rapport. Noorwegen staat vaak op de eerste plaats. In 2014 was Nederland goed voor een vierde plaats en stond België op de 21ste plaats. Onderaan staan de Afrikaanse landen Tsjaad, de Centraal Afrikaanse Republiek, Congo en Niger.
De index meet de gemiddelde prestaties van een land, opgedeeld in drie categorieën:

- **Volksgezondheid**: deze wordt gemeten aan de hand van de levensverwachtingsindex (Life Expectancy Index; LEI) die de gemiddelde levensverwachting bij geboorte uitdrukt

  $$LEI = \frac{LE - 20}{82.3 - 20}$$

  waarbij LE (Life Expectancy) staat voor de levensverwachting bij de geboorte.

- **Kennis**: deze wordt gemeten aan de hand van de onderwijsindex (Education Index; EI) die een maat is voor het analfabetisme en het deel van de bevolking dat primair, secundair en tertiair onderwijs doorloopt

  $$EI = \frac{\sqrt{MYSI \cdot EYSI}}{0.951}$$

  hierbij wordt de gemiddelde scholingsjarenindex (Mean Years of Schooling Index; $MYSI$) gegeven door

  $$MYSI = \frac{MYS}{13.2}$$

  waarbij MYS (Mean Years of Schooling) staat voor het gemiddeld aantal jaren scholing voor een 25-jarige, en wordt de verwachte scholingsjarenindex (Expected Years of Schooling Index; $EYSI$) gegeven door

  $$EYSI = \frac{EYS}{20.6}$$

  waarbij EYS (Expected Years of Schooling) staat voor het verwacht aantal jaren scholing voor een 5-jarige.

- **Levensstandaard**: deze wordt gemeten a.d.h.v. de inkomensindex (Income Index; II) die het bruto nationaal product uitdrukt per hoofd van de bevolking, in koopkrachtpariteit in dollars

  $$II = \frac{\ln(GNI) - \ln(100)}{\ln(107721) - \ln(100)}$$

  waarbij GNI (Gross National Income at purchasing power parity per capita) staat voor het bruto nationaal inkomen per hoofd van de bevolking, en ln staat voor de natuurlijke logaritme.

De uiteindelijke index is het meetkundig gemiddelde van de drie genormaliseerde indices:

$$HDI = \sqrt[3]{LEI \cdot EI \cdot II}$$

Schrijf een programma die de volgende invoer vraagt:

- Naam van het land
- Levensverwachting bij geboorte (LE)
- Gemiddeld aantal jaren scholing voor een 25-jarige (MYS)
- Verwacht aantal jaren scholing voor een 5-jarige (EYS)
- Bruto nationaal inkomen per hoofd van de bevolking ($GNIpc$)

waarbij je LE, MYS, EYS en $GNIpc$ beschouwt *floats*. Het programma moet uiteindelijk een regel op het scherm weergeven waarin de naam van het land staat en de HDI als *float*. Een mogelijke input/output is

```
Naam land: Belgie
LE:  80.548
MYS: 10.86875064
EYS: 16.2
GNI: 39470.90422
De HDI van Belgie bedraagt 0.8896358328268209
```

Bereken de ontwikkelingsindex van enkele andere landen:

Land	LE	MYS	EYS	GNI
Noorwegen	81.5	12.6	17.6	63909
Nederland	81.0	11.9	17.9	42397
Duitsland	80.7	12.9	16.3	43049
Groot Brittannië	80.5	12.3	16.2	35002

3.11 Belangrijkste concepten – samenvatting

- Object(en), (gegevens)type(s), variabele(n)
  - Types, bv. *int*, *float*, *str*, *bool*
- Geldige variabelenamen voldoen aan bepaalde regels:
  - begin met een letter; anders letters, cijfers en underscore
  - beginnend met een underscore heeft een speciale betekenis voor later
- *Keywords*: Tabel 3.1
- *Statement*
  Een *statement* is een instructie; geeft geen waarde terug.
- Expressie
  Een expressie is een combinatie van waarden, variabelen en operatoren; geeft een waarde terug.

- Operatoren voor operanden *int* en *float*: Tabel 3.2
  - Voorrangsregels voor rekenkundige bewerkingen: Tabel 3.3
- Operator voor operanden *string* en *string*: `+`
- Operator voor operanden *string* en *int*: `*`
- Functies (eerste kennismaking)
  - *Built-in* functies: `https://docs.python.org/3/library/functions.html`
  - `math`-module: `https://docs.python.org/3/library/math.html`
- Typeconversiefuncties: **`int()`**, **`float()`**, **`str()`**, **`bool()`**
- Input van keyboard en output naar scherm: **`input()`**, **`print()`**
- Gegevenstype *Boolean*: `True`, `False`
- *Boolean* vergelijkingsoperatoren: Tabel 3.5
- Basis *Boolean* operatoren: **`and`**, **`or`**, **`not`**
- Samenvatting voorrangsregels: Tabel 3.9
- De *string*-methoden: *str*.`isalpha()`, *str*.**`isnumeric()`**
  (`https://docs.python.org/3/library/stdtypes.html#string-methods`)
- Andere te beheersen functies:
  - *Built-in Functions*:
    `abs()`, **`complex()`**, **`id()`**, **`len()`**, **`max()`**, **`min()`**, **`pow()`**, **`round()`**, **`pow()`**, **`type()`**
    (zie `https://docs.python.org/3/library/functions.html`)
  - *math - Mathematical functions*:
    `fabs(x)`, `exp(x)`, `log(x)`, `log10(x)`, **`pow(x, y)`**, `sqrt(x)`, `acos(x)`, `asin(x)`, `atan(x)`, `cos(x)`, `sin(x)`, `tan(x)`,
    `degrees(x)`, `radians(x)`, `pi`, `e`
    (zie `https://docs.python.org/3/library/math.html`)

4

Controlestructuren

Tot nu toe hebben we Python-code geschreven die toeliet instructies sequentieel uit te voeren, d.i., de ene instructie na de andere. In dit hoofdstuk bekijken we hoe de volgorde waarin instructies worden uitgevoerd (de *control flow*) kan beïnvloed worden. De structuren die gebruikt worden om de *control flow* te beïnvloeden heten **controlestructuren**.

4.1 Het *if-else* - statement

Een `if-else` statement kan gebruikt worden om, o.b.v. de evaluatie van een (al dan niet samengestelde) conditie, te beslissen om bepaalde instructies wel of niet uit te voeren.

4.1.1 Het *if* - statement

Het `if`-statement vormt de meest eenvoudige controlestructuur. Op basis van een `conditie` (dit is een logische expressie) wordt bepaald of de instructies die zich in de **if-suite**, ook wel de **body** van het `if`-statement genoemd, worden uitgevoerd.

```
if conditie:
    # dit is de if-suite
    # instructie(s) die hier staan worden enkel uitgevoerd
    #  indien de conditie evalueert tot True
```

De `conditie` in het `if`-statement is een logische (of booleaanse) expressie. Enkel indien deze na evaluatie `True` retourneert, zullen de instructies in de `if`-suite worden uitgevoerd[1]. Fragment 4.1 illustreert het gebruik van een `if`-statement.

[1] De `if`-suite moet minstens één instructie bevatten.

Fragment 4.1: `positief_getal.py`

```
1 geheel_getal = int(input("Geef een geheel getal: "))
2
3 if geheel_getal > 0:
4     print("Je hebt een positief getal ingegeven!")
5
6 print("Programma stopt.")
```

In dit fragment is:

- De ***conditie***: de logische expressie `geheel_getal > 0`
- De *if-suite* (**body**): het print statement `print("Je hebt een positief getal ingegeven!")`
- Het statement `print("Programma stopt!")` is geen deel van het `if`-statement en zal **altijd** worden uitgevoerd.

Mogelijke input/output zijn:

```
Geef een geheel getal: 4
Je hebt een positief getal ingegeven!
Programma stopt.
```

```
Geef een geheel getal: -3
Programma stopt.
```

Gebruik van indentatie (inspringen)

De `if`-suite in Fragment 4.1 bestaat uit slechts één instructie. Men kan deze eenvoudig uitbreiden met meerdere instructies zoals in Fragment 4.2.

Fragment 4.2: `positief_getal2.py`

```
1 geheel_getal = int(input("Geef een geheel getal: "))
2
3 if geheel_getal > 0:
4     print("Je hebt een positief getal ingegeven!")
5     kwadraat = geheel_getal**2
6     print("Het kwadraat van", geheel_getal, "is", kwadraat)
7
8 print("Programma stopt.")
```

In dit fragment bestaat de `if`-suite uit de drie instructies op regels 4, 5 en 6. Merk op dat deze drie regels zijn **ingesprongen of geïndenteerd**. Python gebruikt indentatie om aan te geven welke regels tot de `if`-suite behoren.

Belangrijk is dat:

- Alle instructies die tot **eenzelfde `if`-suite** behoren (op eenzelfde niveau, zie verder) eenzelfde **indentatie hebben**. **Standaard worden 4 spaties gebruikt.**

- Instructies die niet meer tot de `if`-suite behoren worden opnieuw links uitgelijnd (of in overeenstemming met het voorgaande niveau, zie verder)

Mogelijke input/output zijn:

```
Geef een geheel getal: 4
Je hebt een positief getal ingegeven!
Het kwadraat van 4 is 16
Programma stopt.
```

```
Geef een geheel getal: -3
Programma stopt.
```

Opdracht 4.1 (`getal_deelbaar_door.py`)
Schrijf een script dat vraagt naar een positief geheel getal. Controleer of dit getal deelbaar is door 2 en geef weer op het scherm indien dit het geval is. Voer vervolgens eenzelfde controle uit voor deelbaarheid door 3. Gebruik hierbij twee `if`-statements. Een mogelijke input/output is:

```
Geef een positief geheel getal: 18
Het getal is deelbaar door 2
Het getal is deelbaar door 3
Programma stopt.

Geef een positief geheel getal: 27
Het getal is deelbaar door 3
Programma stopt.

Geef een positief geheel getal: 97
Programma stopt.
```

Vóór je begint: lees Sectie 4.1.1 aandachtig.

4.1.2 Het *if-else*-statement

Het `if-else`-statement is een uitbreiding van het `if`-statement. Indien de conditie voldaan is, worden enkel de instructies in de `if`-suite uitgevoerd. Indien de conditie niet voldaan is, worden enkel de instructies in de `else`-suite uitgevoerd.

```
if conditie:
    # if suite, uit te voeren indien conditie True is
else:
    # else suite, uit te voeren indien conditie False is
```

De werking van het **if-else**-statement wordt toegelicht a.d.h.v. een voorbeeld. In Fragment 4.3 wordt nagegaan welke van twee ingegeven getallen, het grootste getal is:

Fragment 4.3: grootste_getal.py

```
1 eerste_getal = float(input("Geef het eerste getal: "))
2 tweede_getal = float(input("Geef het tweede getal: "))
3
4 if eerste_getal > tweede_getal:
5     print("Het eerste ingegeven getal is het grootst.")
6 else:
7     print("Het tweede ingegeven getal is het grootst.")
```

Mogelijke input/output is:

```
Geef het eerste getal: 6.5
Geef het tweede getal: 5.6
Het eerste ingegeven getal is het grootst.
```

Ook hier wordt indentatie gebruikt om duidelijk te maaken waar de **else**-suite eindigt.

Opdracht 4.2 (raad_getal.py)
Schrijf een script waarin:

- Automatisch een willekeurig getal tussen 1 en 5 (inclusief) gegenereerd wordt. Gebruik daarvoor de functie `randint()` uit de module `random`, zie Sectie 3.6.3 voor een voorbeeld.
- De gebruiker gevraagd wordt om een waarde (een gok) in te geven.
- Nagegaan wordt of de gebruiker correct gegokt heeft, en een gepaste boodschap op het scherm verschijnt. Maak daarbij o.a. gebruik van een **if-else** statement.

Mogelijke input/output is:

```
Raad een getal tussen 1 en 5
Jouw keuze: 4
Juist geraden!
```

4.1.3 Geneste *if-else*-statements

Binnen een **if**- of **else**-statement kunnen we opnieuw een **if**- of **if-else**-statement gebruiken:

```
if conditie_1:
    # if-suite 1 bevat nieuwe if-else
    if conditie_2:
        # if suite 2
    else:
        # else suite 2
else:
    # else suite 1
```

Men zegt dat deze `if`-statements **genest** zijn. In het bovenstaande voorbeeld bevat de `if`-suite bij `conditie_1` opnieuw een `if-else` statement. Enkel indien `conditie_1 True` is zal `conditie_2` geëvalueerd worden. Indien bijvoorbeeld `conditie_1 True` is, en `conditie_2` is `False`, dan zel **enkel** `else-suite 2` worden uitgevoerd.

In Fragment 4.4 berekent een postorderbedrijf de prijs voor het verzenden van lichtgewicht pakketen. Het bedrijf rekent € 5 aan voor de eerste 300 g en € 2 voor iedere bijkomende 100 g (afgerond tot op 100 g), tot een maximum van 1000 g. Het programma vraagt naar de massa van een pakket. Indien het pakket lichter is dan 1000 g moet de kostprijs berekend worden, indien het pakket zwaarder weegt moet het verzendingspersoneel daarover ingelicht worden.

Fragment 4.4: `verzendingskost.py`

```
1  massa = float(input("Geef de massa in gram van het pakket: "))
2
3  if massa <= 1000:
4      if massa <= 300:
5          kostprijs = 5
6      else:
7          kostprijs = 5 + 2 * round((massa - 300)/100)
8
9      print("De verzendingskost is:", kostprijs)
10 else:
11     print("Maximaal gewicht voor lichte pakketen overschreden.")
12     print("Gebruik de zware pakketendienst.")
```

Merk op dat regels 4, 6 en 9 **eenzelfde indentatie** hebben! Dit is noodzakelijk omdat deze regels tot dezelfde `if`-suite behoren.

Mogelijke input/output zijn:

```
Geef de massa in gram van het pakket: 225
De verzendingskost is: 5 euro.
```

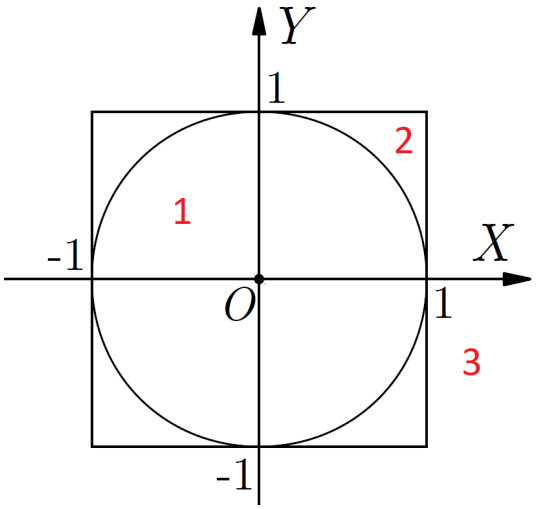
```
Geef de massa in gram van het pakket: 700
De verzendingskost is: 13 euro.
```

```
Geef de massa in gram van het pakket: 575
De verzendingskost is: 11 euro.
```

```
Geef de massa in gram van het pakket: 1600
Maximaal gewicht voor lichte pakketen overschreden.
Gebruik de zware pakketendienst.
```

Opdracht 4.3 (`cirkel_vierkant.py`)

Beschouw een cirkel met straal 1 en een vierkant met zijde 2 gecentreerd rond de oorsprong van een XY-assenkruis (zie Figuur 4.1), met zijden die evenwijdig zijn aan de assen.



Figuur 4.1: Een cirkel met straal 1 binnen een vierkant met zijde 2.

Schrijf een script dat:

- De x- en y-coördinaten van een punt aan de gebruiker vraagt.
- Nagaat of dit punt binnen/buiten de cirkel en het vierkant liggen door gebruik te maken van geneste **`if-else`** statements. Er zijn drie opties (zie rode cijfers in Figuur 4.1):
  (1) Binnen cirkel.
  (2) Binnen vierkant, maar niet binnen cirkel.
  (3) Zowel buiten vierkant als cirkel.
- De gepaste optie op het scherm toont.

Mogelijke input/output zijn:

```
Geef x: 0.99
Geef y: 0
Het punt (0.99, 0) ligt binnen de cirkel.
```

```
Geef x: 0.99
Geef y: 0.99
Het punt (0.99, 0.99) ligt tussen (of op) de cirkel en het vierkant.
```

```
Geef x: 1.01
Geef y: 1
Het punt (1.01, 1) ligt buiten het vierkant.
```

Vóór je begint:

- Bekijk Figuur 4.1 aandachtig.
- Schrijf wiskundig neer wat de condities zijn opdat de coördinaten van een punt binnen het vierkant of de cirkel liggen.
- Vertaal deze condities naar Python expressies.

4.1.4 Controle van gebruikersinput met if-statements

Wanneer gebruikersinput gevraagd wordt, dan wordt vaak verondersteld dat de gebruiker (min of meer) invoert wat men van hem verwacht. Dit is echter niet steeds het geval. Bekijk hiertoe het volgende codefragement:

Fragment 4.5: `even_getal.py`

```
1 geheel_getal = int(input("Geef een geheel getal: "))
2
3 if geheel_getal % 2 == 0:
4     print("Je hebt even getal ingegeven!")
5
6 print("Programma stopt.")
```

Op de eerste regel wordt de gebruikersinput onmiddellijk geconverteerd naar een `int` object. Zoals we reeds gezien hebben bij conversiefuncties is dit niet steeds mogelijk, zoals geïllustreerd wordt in het volgende venster

```
Geef een geheel getal: Hallo

ValueError: invalid literal for int() with base 10: 'Hallo'
```

De string `'Hallo'` is immers niet converteerbaar naar een integer object. Om dergelijke fouten op te vangen, kunnen we gebruik maken van de **methode** `isnumeric()`. Het gebruik van deze methode wordt hieronder geïllustreerd:

```
>>> a = "12"
>>> a.isnumeric()
True
>>> b = "Hallo"
>>> b.isnumeric()
False
```

Deze methode[2] retourneert `True` indien de string waarop ze wordt toegepast **enkel uit cijfers bestaat**: i.e. de karakters `0`, `1`, `2`, ... `9`. Deze methode kan als volgt gebruikt worden om het codefragment meer robuust te maken.

Fragment 4.6: `even_getal2.py`

```
1  geheel_getal_str = input("Geef een geheel getal: ")
2
3  if geheel_getal_str.isnumeric():
4      geheel_getal = int(geheel_getal_str)
5      if geheel_getal % 2 == 0:
6          print("Je hebt een even getal ingegeven!")
7  else:
8      print("Je gaf geen geldig getal in")
9
10 print("Programma stopt.")
```

Opdracht 4.4 (`getal_deelbaar_door2.py`)
Pas je antwoord van Opdracht 4.1 (`getal_deelbaar_door.py`) aan zodat gecontroleerd wordt of de gebruikersinput geldig (m.a.w. **numeriek**) is.

Vóór je begint:

- Bekijk Sectie 4.1.4.

```
Geef een positief geheel getal: a26
Je gaf geen geldig getal in.
Programma stopt.
```

4.1.5 Het *if-elif-elif-...-else* - statement

Het Python sleutelwoord **`elif`** is de afkorting van *else if*. Het **`elif`**-statement heeft ook een *logische expressie* als bijhorende conditie, en een **`elif`**-suite: de geïndenteerde blok code die uitgevoerd wordt indien de conditie `True` retourneert. Het **`if-elif-elif-...-else`**-statement heeft de volgende vorm:

```
if conditie_1:
    # suite 1, uit te voeren indien conditie_1 True is
elif conditie_2:
    # suite 2, uit te voeren indien conditie_2 True is,
    # maar conditie_1 niet
```

[2] De syntax die hier gebruikt wordt, zal later meer in detail besproken worden. Belangrijk om weten is dat **`isnumeric()`** wordt opgeroepen door **`.isnumeric()`** te laten voorafgaan door een string object.

```
elif conditie_3:
    # suite 3, uit te voeren indien conditie_3 True is,
    # maar conditie_1 en 2 niet

# zoveel elif-statements als nodig

else:
    # laatste suite, uit te voeren wanneer alle vorige
    # condities False waren
```

Het `if-elif-elif-...-else`-statement zal de suite uitvoeren die hoort bij de **eerste** *conditie* die `True` is. Indien geen enkele conditie `True` is, wordt het `else`-blok uitgevoerd.

In Fragment 4.7 wordt een lettergraad toegekend aan bepaalde percentage-intervallen (A voor $[90, 100]$; B voor $[80, 90[$; C voor $[70, 80[$; D voor $[60, 70[$; en geen lettergraad indien percentage < 60). Het programmaatje vraagt naar het percentage en schrijft de lettergraad naar het scherm.

Fragment 4.7: `letter_graad.py`

```
1  percentage = float(input("Wat is jouw percentage? "))
2
3  if percentage > 100:
4      print("Onmogelijk!")
5  elif 90 <= percentage <= 100:
6      print("Je krijgt de graad A")
7  elif 80 <= percentage < 90:
8      print("Je krijgt de graad B")
9  elif 70 <= percentage < 80:
10     print("Je krijgt de graad C")
11 elif 60 <= percentage < 70:
12     print("Je krijgt de graad D")
13 else:
14     print("Percentage te laag, je krijgt geen graad")
```

Mogelijke input/output zijn:

```
Wat is jouw percentage? 70
Je krijgt de graad C
```

```
Wat is jouw percentage? 52.5
Percentage te laag, je krijgt geen graad
```

Opmerking: de expressie `(90 <= percentage) and (percentage <= 100)` kan verkort genoteerd worden als `90 <= percentage <= 100`. Men noemt dit het ***chainen*** van vergelijkingsoperaties. Deze manier van coderen leidt tot minder broncode maar leidt bij beginnende programmeurs tot

onverwacht gedrag en bijgevolg onverwachte resultaten. Gebruik deze syntax **bij voorkeur enkel** in situaties zoals bovenstaande, waarbij nagegaan wordt of een getal tussen bepaalde grenzen ligt (zie `https://peps.python.org/pep-0535/`).

Opdracht 4.5 (`bmi_besluit.py`)
Om een indicatie te krijgen over het lichaamsgewicht van een persoon wordt meestal de Body Mass Index (BMI) gebruikt. De BMI wordt ook wel de Queteletindex genoemd, bedacht door de Gentenaar Adolphe Quételet. In de BMI formule wordt er geen onderscheid gemaakt tussen man en vrouw. De BMI bereken je door de massa in kilogram te delen door het kwadraat van de lengte in meter.

$$BMI = \frac{\text{massa } (kg)}{(\text{lengte } (m))^2} .$$

Het resultaat stemt overeen met een bepaalde categorie (leeftijd > 17 jaar):

BMI	**Categorie**
< 18.5	ondergewicht
$18.5 \leq BMI < 25.0$	gezond gewicht
$25.0 \leq BMI < 30.0$	overgewicht
$30.0 \leq BMI < 40.0$	obesitas
≥ 40.0	morbide obesitas

Schrijf een programma dat vraagt naar de massa (in kg) en de lengte (in m) van een persoon, de BMI berekent en de categorie weergeeft naar het scherm. Mogelijke input/output zijn:

```
Geef de massa (in kg): 67
Geef de lengte (in m): 1.73
BMI: 22.4
Categorie: Gezond gewicht
```

```
Geef de massa (in kg): 96
Geef de lengte (in m): 1.81
BMI: 29.3
Categorie: Overgewicht
```

4.2 *Intermezzo*: Fouten en foutenbehandeling

Op dit punt in de syllabus kunnen we reeds een eenvoudig programma schrijven. Bij het ontwikkelen van deze programma's zullen we vaak fouten maken. Soms leiden deze tot een fout resultaat, maar in andere gevallen zal dit aanleiding geven tot het opwerpen van een foutmelding door de Python

interpreter. Deze meldingen goed kunnen interpreteren is een belangrijke vaardigheid. In dit intermezzo komen fouten voor de eerste keer aan bod (Hoofdstuk 9 gaat hier nog dieper op in).

Fouten worden ingedeeld in drie grote categorieën: **syntax** fouten, **runtime** fouten en **logische** fouten.

4.2.1 Syntax fouten

Syntax fouten zijn fouten die optreden wanneer de **broncode niet voldoet aan de Python codeerregels**. Als voorbeeld zie je hieronder in een IPython console de output wanneer het dubbele punt (`:`) ontbreekt aan het einde van het `if`-statement op regel 2:

```
>>> geheel_getal_str = input("Geef een geheel getal: ")
>>> if geheel_getal_str.isnumeric()
    geheel_getal = int(geheel_getal_str)
    print(geheel_getal)

  File "<ipython-input-7-431c70a132e5>", line 2
    if geheel_getal_str.isnumeric()
                                   ^
SyntaxError: invalid syntax
```

De `IndentationError` wordt opgeworpen als de indentatie niet syntactisch correct is. Dit is een bijzonder geval van een syntax fout waarvoor een afzonderlijke exception werd voorzien.

4.2.2 Runtime fouten

Runtime fouten zijn **fouten die optreden tijdens het uitvoeren van syntactisch correcte broncode**. We kunnen bijvoorbeeld een syntactisch juist Python programma schrijven dat tracht een geheel getal te delen door 0. Geen enkele grammaticale Python-regel weerhoudt ons van zo'n programma te schrijven, maar de uitvoering ervan zal tot een fout leiden:

```
>>> x = 5
>>> y = 0
>>> z = x/y

  File "<ipython-input-8-ec1eaf47f76b>", line 3, in <module>
    z = x/y

ZeroDivisionError: division by zero
```

In beide gevallen zal de Python *interpreter* een ***exception*** opwerpen. **Een *exception* is een signaal dat aangeeft dat er een fout (of iets ongewoons) is opgetreden**. Wanneer een *exception* wordt opgeworpen, wordt de **uitvoering van het programma onderbroken**. De *interpreter* zal eerst deze *exception* afhandelen. De programmeur kan deze afhandeling zelf voorzien,

maar als dit niet het geval is, zal het programma onmiddellijk beëindigd worden. Het afhandelen van de *exception* bestaat dan eenvoudigweg uit het printen van een foutboodschap op het scherm (in de console) én een onmiddellijke beëindiging van de uitvoering.

Er kunnen tal van *exceptions* opgeworpen worden. Hier beperken we ons tot de volgende:

- `SyntaxError`: wordt opgeworpen als de *interpreter* een *syntax* fout ontdekt,
- `ZeroDivisionError`: wordt opgeworpen als de noemer van een deling nul is (vb. `5/0`),
- `NameError`: wordt opgeworpen als in een instructie een naam wordt gebruikt die **niet** aanwezig is in de *namespace*,
- `TypeError`: wordt opgeworpen als men tracht een bewerking uit te voeren met een gegevenstype dat deze bewerking **niet** ondersteunt (vb. `"hallo" * "twee"`),
- `ValueError`: wordt opgeworpen als een bewerking of een functie input krijgt die wel het correcte type heeft, maar een **verkeerde** waarde (vb. `(-5)**(1/2)`),

In Sectie 9.9 komen andere *exceptions* aan bod en gaan we dieper in op het opwerpen en afhandelen van *exceptions*.

4.2.3 Logische fouten

Logische fouten treden op in Python wanneer de code wordt uitgevoerd zonder enige syntax- of runtime-fouten, maar onjuiste resultaten oplevert vanwege **verkeerde logica in de code**. Dit soort fouten wordt vaak veroorzaakt door onjuiste aannames, een onvolledig begrip van het probleem of de onjuiste toepassing van een algoritme of formule. In tegenstelling tot syntax- of runtimefouten, kunnen logische fouten een uitdaging zijn om te detecteren en te verbeteren. Dit komt omdat de **code wordt uitgevoerd zonder foutmeldingen weer te geven**. De resultaten lijken misschien correct, maar de code kan in bepaalde situaties onjuiste uitvoer produceren.

Enkele fouten die leiden tot logische fouten zijn:

- één of meer instructies die een verkeerde indentatie hebben (maar **geen** aanleiding geven tot een `IndentationError`),
- de verkeerde variabele gebruiken,
- gehele deling in plaats van normale deling,
- fout in een logische expressie,
- voorrangsregels voor operatoren (zie Tabel 3.3) niet nageleefd,
- ...

Stel bijvoorbeeld dat de formule voor de berekening van de BMI als volgt werd gegeven:

$$BMI = \frac{\text{massa}}{\text{lengte} \times \text{lengte}}$$

In onderstaand codefragment wordt deze formule gebruikt om de BMI te berekenen voor twee lengtes:

```
>>> massa = 67
>>> L1 = 1.0
>>> L2 = 1.73
>>> BMI1 = massa/L1*L1
>>> print(BMI1)
67.0

>>> BMI2 = massa/L2*L2
>>> print(BMI2)
67.0
```

We krijgen geen foutmelding maar de waarde van `BMI2` is wel degelijk fout: deze zou 22.3863 moeten zijn. Dit is een gevolg van het feit dat de deling en de vermenigvuldiging dezelfde precedentie (voorrang) hebben (zie Tabel 3.3). Dit betekent dat er eerst wordt gedeeld door de lengte en vervolgens het resultaat wordt vermenigvuldigt met opnieuw de lengte. Globaal gezien is er dus niets gebeurd. Een oplossing bestaat er in haakjes `()` te plaatsen rond de noemer zodat eerst de lengte gekwadrateerd wordt. Uiteraard kan ook de `**` operator gebruikt worden. Deze heeft voorrang op de deling.

Opdracht 4.6

Bekijk de onderstaande codefragmenten. Tracht eerst voor jezelf uit te maken wat de code doet. Voer de code uit. Geef aan welke foutmelding je krijgt en waarom. Gaat het om een *syntax* of *runtime* fout?

1. Codefragment 1:

```
a = 3
b = 4
c = 5
D = b**2 - 4*a*c
if D > 0:
print("Twee verschillende oplossingen!")
```

   Type foutmelding: _____
   Type fout (omcirkel): *syntax* / *runtime*

2. Codefragment 2:

```
L = 1.75
G = 65
BMI = G/L**2
print(bmi)
```

   Type foutmelding: _____
   Type fout (omcirkel): *syntax* / *runtime*

3. Codefragment 3:

```
x = "3.14"
y = int(x)
```

Type foutmelding: _____
Type fout (omcirkel): *syntax* / *runtime*

4. Codefragment 4:

```
prijs1 = input("Geef de prijs van het eerste product: ")
prijs2 = input("Geef de prijs van het tweede product: )
gemiddelde = (float(prijs1) + float(prijs2))/2
print("De gemiddelde prijs is", gemiddelde)
```

Type foutmelding: _____
Type fout (omcirkel): *syntax* / *runtime*

5. Codefragment 5:

```
getal1 = 81.70
getal2 = "12.34"
getal1 + getal2
```

Type foutmelding: _____
Type fout (omcirkel): *syntax* / *runtime*

Een veel voorkomende `TypeError` treedt op wanneer bepaalde operatoren vergeten worden en het lijkt alsof er een functie-oproep wordt uitgevoerd. De volgende opdracht illustreert dit a.d.h.v enkele voorbeelden.

Opdracht 4.7

1. Omschrijf met eigen woorden waarom hier een fout optreedt. Wat betekent *not callable* hier?

```
>>> a = 1
>>> a(1)
Traceback (most recent call last):

  File "<ipython-input-3-5645faaf81f3>", line 1, in <module>
    a(1)

TypeError: 'int' object is not callable
```

2. Een student voert de (wiskundige) uitdrukking $z = x(y - 1)$ met $x = 3.14$ en $y = 2.5$ in de console als volgt in:

```
>>> x = 3.14
>>> y = 2.5
>>> z = x(y - 1)
Traceback (most recent call last):

  File "<ipython-input-8-639ebea6e118>", line 3, in <module>
    z = x(y - 1)

TypeError: 'float' object is not callable
```

Omschrijf met eigen woorden waarom hier een fout optreedt. Verbeter de fout.

4.2.4 Foutenbehandeling - ter info

Python voorziet een `try-except`-statement dat de programmeur toelaat een *exception* op te vangen en deze af te handelen, en vertoont enkele gelijkenissen met het `if-else`-statement. Dit statement bestaat uit twee delen: een `try`-blok en één of meerdere `except`-blokken. Hun functies zijn de volgende:

- Het `try`-blok bevat code waarover we (als programmeurs) wensen te waken op mogelijke runtime fouten. Met andere woorden, we zijn bezorgd over een deel van onze code die een *exception* kan opwerpen.
- Elk `except` blok wordt geassocieerd met een bepaalde *exception* en een code-blok dat zal uitgevoerd worden als deze *exception* optreedt in het `try`-blok.

Een eenvoudig voorbeeld van een `try-except` constructie wordt in Fragment 4.8 gegeven.

Fragment 4.8: `omzetting.py`

```
1 getal_str = input("Geef een geheel getal: ")
2 try:
3     getal = int(getal_str)
4     print("Het getal is: ", getal)
5 except ValueError:
6     print("Fout bij het converteren naar een int.")
```

In Fragment 4.8 wordt in het `try`-blok de variabele `getal_str` omgezet (gecast) naar een `int`. Indien dit mogelijk is, wordt het omgezette getal naar het scherm geschreven. Indien de

omzetting niet mogelijk is, wordt een `ValueError` opgeworpen door regel 3. Met het opwerpen van deze `ValueError` wordt het `try`-blok **onmiddellijk beëindigd**. De instructie op regel 4 wordt dus **niet** uitgevoerd. Het `except`-blok zal vervolgens deze exceptie opvangen en afhandelen door regel 6 uit te voeren.

We verwijzen opnieuw naar Sectie 9.9 voor een meer gedetailleerde bespreking van fouten en hun behandeling.

4.3 Het *while* - statement

4.3.1 Het basis *while* - statement

Het `while`-statement voert een sequentie van instructies (de `while`-suite of body) meerdere keren uit. Het aantal keer dat de sequentie uitgevoerd wordt, wordt bepaald door een (eventueel samengestelde) **conditie**. Naar analogie met het if-statement is deze conditie een logische expressie. De structuur van het `while`-statement wordt getoond in Fragment 4.9.

Fragment 4.9: Structuur `while`-statement

```
1 # (i) initialisatie van de loop-control variabele
2 while conditie:   # (ii) test de loop_controle_variabele in een conditie
3     # (iii) de while-suite: uit te voeren indien conditie True is
4     # (iv)  update loop_controle_variabele
```

Zoals blijkt uit deze structuur, bestaat het `while`-statement uit een aantal componenten die (bijna) altijd voorkomen. Om de bespreking concreter te maken, beschouwen we Fragment 4.10. De broncode in dit fragment zal, bij uitvoeren, er toe leiden dat de output `Waarde van i: 1`, …, `Waarde van i: 4` naar het scherm geprint wordt.

Fragment 4.10: Voorbeeld `while`-statement

```
1 i = 1 # (i) initialisatie  loop-control variabele i
2 while i <= 4:   # (ii) conditie (logische expressie)
3     print("Waarde van i: ", i) # (iii) begin while-suite
4     i = i + 1                 # (iv)  update loop-control variabele
5
6 print("Programma stopt")       # behoort niet meer tot while suite
```

De output van dit script is:

```
Waarde van i: 1
Waarde van i: 2
Waarde van i: 3
Waarde van i: 4
Programma stopt
```

We bespreken de uitvoer van dit codefragment stap-voor-stap:

Stap (1) De *loop-control* variabele `i` wordt **geïnitialiseerd** door er de waarde 1 aan toe te kennen.

Stap (2) De logische expressie `i <= 4` wordt geëvalueerd. Er zijn twee opties:

(1) **De logische expressie retourneert** `True`: De `while`-suite (regels 3 en 4) wordt uitgevoerd. Merk op dat als onderdeel van de `while`-suite de waarde van loop-control variabele `i` gewijzigd wordt. Nadat alle instructies in de `while`-suite werden uitgevoerd, herbegint **Stap 2**.

(2) **De logische expressie retourneert** `False`: De `while`-lus wordt beëindigd en men gaat verder naar **Stap 3**

Stap (3) Instructies die na de `while`-suite komen worden uitgevoerd.

Het (eenmalig) uitvoeren van alle instructies in de `while`-suite noemen we een **iteratie**. In het voorgaande voorbeeld werden dus vier iteraties uitgevoerd.

Opmerking: nadat de `while`-suite één of meerdere keren is uitgevoerd moet de *loop-control variabele* geüpdatet worden om ervoor te zorgen dat de `while`-lus eindigt! Indien de *Boolean expressie* nooit `False` wordt, dan zal de `while`-lus oneindig doorgaan (*infinite loop*) en dat is meestal niet gewenst.

TIP: uitvoer onderbreken
Bij het schrijven en testen van broncode komen *infinite loops* vrij vaak voor omwille van logische fouten in de code. Bij het uitvoeren zal dit ervoor zorgen dat de uitvoer nooit eindigt (en de console onbruikbaar wordt). Deze uitvoer kan echter geforceerd onderbroken worden op volgende manieren:

- Optie 1 (scripts): Klik in de terminal en druk **CTRL-SHIFT-C**. Het actieve proces zal beëindigd worden.
- Optie 2 (interactieve sessie in VS Code): Kies voor *interrupt* in VS-code bovenaan het interactieve sessie-venster. De actieve cell wordt beëindigd.
- Optie 3 (interactieve sessie in VS Code): Kies voor *Restart*. Deze optie zal ervoor zorgen dat de interactieve sessie (en de achterliggende kernel die de instructies uitvoert) herstart wordt. Alle variabelen in het werkgeheugen gaan verloren.

Opdracht 4.8 (`machtenvan2.py`)
Vul de onderstaande `while`-lus aan zodat, bij het uitvoeren:

- De gebruiker gevraagd wordt een geheel getal in te geven (met de functie `input()`).
- Alle (gehele) machten van 2 die kleiner dan of gelijk zijn aan dit getal onder elkaar op het scherm print. Als de gebruiker bijvoorbeeld 80 ingeeft, dan moeten de waarden 1, 2, 4, 8, 16, 32, 64 onder elkaar op het scherm geprint worden.

```
getal = int(input("Geef een geheel getal in: "))
macht = .........              # initialisatie loop-control variabele macht
while ................. :      # de conditie
    print( ................. ) # begin while-suite
    macht = ..........         # update loop-control variabele

print("Programma stopt")       # behoort niet meer tot while suite
```

Vóór je begint:

- Lees Sectie 4.3.1 aandachtig.

Opdracht 4.9 (typ_woorden.py)

Schrijf een script dat:

- Aan de gebruiker vraagt hoeveel woorden hij wenst in te typen (variabele `max_aantal`).
- Aan de gebruiker `max_aantal` keer vraagt om een woord in te voeren.
- Bij elk woord bepaalt hoeveel karakters het telt (de lengte kan je bepalen met de functie `len()`) en dit aantal op het scherm brengt.
- Na het beëindigen van de `while`-lus op het scherm toont hoeveel karakters er in totaal werden ingegeven.
- (moeilijker) Na het beëindigen van de `while`-lus op het scherm alle ingegeven woorden na elkaar weergeeft, gescheiden door komma's.

Een mogelijke input/output is:

```
Aantal woorden die je wenst in te typen: 3
Typ een woord: hond
Woord telt 4 karakters
Typ een woord: kat
Woord telt 3 karakters
Typ een woord: paard
Woord telt 5 karakters

Totaal aantal karakters: 12
Volgende woorden werden ingegeven:
hond, kat, paard                    # GEEN komma na laatste woord
```

Merk op dat er **na het laatste woord geen komma** staat.

4.3.2 Nesten van *while* en *if* - statements

Een `while`-statement kan een of meerdere `if`-, `if-else`- en `if-elif-else`-statements bevatten. We illustreren dit principe a.d.h.v. het vermoeden van Collatz[3]. Beschouw de rij van positieve gehele getallen $x_0, x_1, x_2, x_3, ...$ waarbij x_0 zelf kan gekozen worden en $x_1, x_2, x_3, ...$ recursief worden gedefinieerd als volgt:

$$x_{n+1} = \begin{cases} \dfrac{x_n}{2} & \text{als } x_n \text{ even is,} \\ 3x_n + 1 & \text{als } x_n \text{ oneven is.} \end{cases}$$

Het vermoeden van Collatz stelt dat er voor elke (gehele) startwaarde x_0 een n bestaat waarvoor geldt dat $x_n = 1$.

We de kleinste n waarvoor $x_n = 1$ berekenen met het volgende script:

```
1  x = int(input("Geef een startwaarde x0: ")) # waarde voor x
2  n = 0                                       # bijhorende waarde voor n
3
4  while x != 1:
5      if x % 2 == 0:      # even getal --> x/2
6          x = x // 2
7      else:               # oneven getal --> 3x + 1
8          x = 3 * x + 1
9      n = n + 1
10
11 print("Na", n, "iteraties wordt 1 bereikt")
```

Dit codefragment geeft volgende output voor $x_0 = 46$ en $x_0 = 687$:

```
Geef een startwaarde x0: 46
Na 16 iteraties wordt 1 bereikt
```

```
Geef een startwaarde x0: 687
Na 38 iteraties wordt 1 bereikt
```

Merk op dat indentatie gebruikt wordt om aan te geven in welke constrolestructuur bepaalde instructies zich bevinden.

- De instructie `n = n + 1` bevindt zich in het `while`-statement maar **niet** binnen het `if-else` statement. Dit wil zeggen dat deze instructies in elke iteratie wordt uitgevoerd en dus **niet beïnvloed** wordt door het resultaat van `x % 2 == 0`.
- Het print statement `print("Na", n, ...)` maakt geen deel uit van de `while`-suite en wordt steeds eenmalig uitgevoerd nadat de `while`-lus werd doorlopen.

[3]Dit is een gekend vermoeden in de wiskunde, maar geen bewezen stelling. Daar is men tot op heden nog niet in geslaagd.

Opdracht 4.10 (`perfecte_getallen_checker.py`)
Een perfect getal is een natuurlijk getal waarvoor de som van zijn (natuurlijke) gehele delers, **met uitzondering van het getal zelf**, gelijk is aan het natuurlijk getal. Een tweetal voorbeelden zijn:

$$\begin{aligned} 6 &= 1+2+3 \\ 28 &= 1+2+4+7+14 \end{aligned}$$

Schrijf een script dat nagaat of een ingegeven getal een perfect getal is. Volg de volgende stappen:

1. Vraag naar een natuurlijk getal.
2. Zoek naar de (natuurlijke) gehele delers van dat getal.
3. Hou een variabele `som` bij die je telkens verhoogt wanneer een nieuwe deler gevonden wordt.
4. Controleer of het getal perfect is en print het resultaat op het scherm.

Mogelijke input/output zijn:

```
Geef een natuurlijk getal: 6
6 is een perfect getal
```

```
Geef een natuurlijk getal: 28
28 is een perfect getal
```

```
Geef een natuurlijk getal: 128
128 is geen perfect getal
```

Kan je nog perfecte getallen vinden? Probeer eens 496 en 8128.

Vóór je begint:

- Lees Sectie 4.3.2 aandachtig.
- Nagaan of een getal een deler is, kan eenvoudig met een (korte) logische expressie

4.3.3 Nesten van `while`-statements

De **`while`**-suite kan één of meerdere nieuw **`while`**-statements bevatten. In dit geval zeggen we dat deze statements **genest** worden. In dit geval worden (meestal) twee loop-variabelen gebruikt, zoals geïllustreerd wordt in Fragment 4.11.

Fragment 4.11: Voorbeeld `while`-statement

```
i = 1          # initialisatie loop-variabele outer-loop
while i <= 2:
    j = 1      # initialisatie loop-variabele inner-loop
    while j <= 3:
        print("i =", i, ", j =", j)
        j = j + 1   # update loop variabele inner-loop
    i = i + 1       # update loop variabele outer-loop
```

Volgend output wordt bekomen:

```
i = 1 , j = 1
i = 1 , j = 2
i = 1 , j = 3
i = 2 , j = 1
i = 2 , j = 2
i = 2 , j = 3
```

We bespreken dit resultaat stap-voor-stap:

- De eerste `while`-lus wordt de **outer loop** genoemd, de `while`-lus die deel uitmaakt van de `while`-suite van de outer loop heet de **inner loop**.
- Uit de output blijkt duidelijk dat in elke iteratie van de outer loop de inner loop volledig (dus alle iteraties) wordt doorlopen.
- Omdat de initialisatie van `j` binnen de `while`-suite van de outer loop staat, zal in elke iteratie van de outer loop `j` opnieuw starten bij 1.

Opdracht 4.11 (`restmatrix.py`)

Een restmatrix is een tabel (of matrix) waarvoor geldt dat het element in rij i en kolom j als waarde de rest bij deling van i door j bevat. Deze rest bereken je met `i % j`. De onderstaande tabel is een restmatrix met 9 rijen en kolommen. Schrijf een script dat deze tabel genereert op basis van geneste while-lussen

```
0  1  1  1  1  1  1  1  1
0  0  2  2  2  2  2  2  2
0  1  0  3  3  3  3  3  3
0  0  1  0  4  4  4  4  4
0  1  2  1  0  5  5  5  5  # element op rij 5, kolom 3 is 2 omdat
0  0  0  2  1  0  6  6  6  # 5 % 3 als resultaat 2 heeft (rest is 2)
0  1  1  3  2  1  0  7  7
0  0  2  0  3  2  1  0  8
0  1  0  1  4  3  2  1  0
```

Vóór je begint:

- Lees Sectie 4.3.3 aandachtig.
- Tracht eerst alle rijen onder elkaar op het scherm te brengen, niet in tabelvorm. Dit is eenvoudiger.
- Met `print(..., end = "")` print je getallen naast elkaar.
- Met `print()` start je een nieuwe rij.

4.3.4 Herhaalde vraag om gebruikersinput

In bepaalde programma's wordt een gebruiker gevraagd om input te geven via het toetsenbord, vervolgens wordt deze input verwerkt, en wordt tenslotte opnieuw om input gevraagd. Indien de gebruiker het programma wenst te beëindigen, kan hij dit doen door bv. `exit` in te geven. Fragment 4.12 toont een mogelijke implementatie van een dergelijke strategie.

Fragment 4.12: `gebruikersinput_herhaald.py`

```
1 gebruikersinput = input("Geef een woord in: ")    # initialiseer de input
2 while gebruikersinput != "exit":                  # (stop)conditie
3     print("Het woord telt", len(gebruikersinput), "karakters")
4     gebruikersinput = input("Geef een woord in: ") # vraag nieuwe input
5 print("Programma stopt")
```

Mogelijke input/output is:

```
Geef een woord in: Hallo
Het woord telt 5 karakters

Geef een woord in: iedereen
Het woord telt 8 karakters

Geef een woord in: exit
Programma stopt
```

Opmerking: de `while`-lus zal stoppen van zodra **exact** `exit` ingegeven wordt. Dit is niet het geval als er `Exit` of `EXIT` werd ingegeven. Immers, Python is **hoofdlettergevoelig**.

Opdracht 4.12 (gebruikersinput.py)
Beschouw het volgende codefragment:

Fragment 4.13: som_even_getallen.py

```
1  getal_str = input("Geef een getal: ")
2  som = 0
3
4  while getal_str != "999":
5     getal = int(getal_str)
6     if getal % 2 == 1:
7        som = som + getal
8     getal_str = input("Geef een getal: ")
9
10 print("De som is", som)
```

Omschrijf in één zin wat de functionaliteit is van dit codefragment:

4.3.5 Het *while-else* - statement en het `break` keyword

In de voorgaande secties werd het aantal interaties van de **`while`**-lus gecontroleerd door een *conditie*. Enkel wanneer deze condities `False` retourneert, wordt de **`while`**-lus beëindigd. In deze sectie bespreken we het Python keyword **`break`**. Wanneer dit keyword wordt geëvalueerd zal de **`while`**-lus waarin **`break`** zich bevindt onmiddellijk beëindigd worden. Fragment 4.14 illustreert het gebruik van dit keyword.

Fragment 4.14: while_break.py

```
1  i = 0
2  while i < 10:
3      i = i + 1
4      if i == 3:
5          print("Break statement")
6          break
7      print("Waarde van i: ", i)
8
9  print("Programma stopt")
```

De output van dit codefragment is:

```
Waarde van i:  1
Waarde van i:  2
Break statement
Programma stopt
```

Het bovenstaande fragment illustreert dat, zodra `i` de waarde 3 heeft, het **`break`**-statement wordt uitgevoerd en de iteratie beëindigd wordt. Merk op dat de boodschap `Waarde van i: 3` niet op het scherm geprint wordt, wat aangeeft dat de **`while`**-suite onmiddellijk verlaten wordt en dat de **huidige iteratie dus niet volledig wordt afgewerkt**.

Opmerking: indien een **`break`**-statement voorkomt in een **`while`**-suite die deel uitmaakt van een **`while`**-statement dat genest is, dan zal enkel de *inner-loop* beëindigd worden.

Opdracht 4.13 (`gebruikersinput_break.py`)
Het onderstaande codefragment vraagt herhaaldelijk aan de gebruiker om input (dit is een kopie van Codefragment 4.12).

```
gebruikersinput = input("Geef een woord in: ")
while gebruikersinput != "exit":
    print("Het woord telt", len(gebruikersinput), "karakters")
    gebruikersinput = input("Geef een woord in: ")
print("Programma stopt")
```

De functie **`input()`** komt tweemaal voor in dit codefragment. Maak gebruik van een **`break`** keyword (en herschik indien nodig de code) zodat de functie **`input()`** op één plaats wordt opgeroepen.

```
gebruikersinput = ..........................
while gebruikersinput != "exit":
        ............................
        ............................
        ............................
        print("Het woord telt", len(gebruikersinput), "karakters")
        ............................
        ............................
        ............................

print("Programma stopt")
```

Naar analogie met het **`if-else`**-statement, kan ook het **`while`**-statement uitgebreid worden met een **`else`**-suite. De instructies die zich in de **`else`**-suite bevinden worden (maximaal) één keer uitgevoerd nadat de *conditie* `False` retourneert bij evaluatie.

```
while conditie:
    # while-suite # uitgevoerd indien conditie True retourneert
else:
    # else-suite  # uitgevoerd indien conditie False retourneert
```

Het volgende voorbeeld illustreert dit gebruik.

```
1 i = 0
2 while i < 5:
3     i = i + 1
4     print("Waarde van i: ", i)
5 else:
6     print("De else suite!")
7
8 print("Programma stopt")
```

De output van dit fragment is:

```
Waarde van i:  1
Waarde van i:  2
Waarde van i:  3
Waarde van i:  4
Waarde van i:  5
De else suite!
Programma stopt
```

Uit deze output blijkt, wanneer `i` verhoogd wordt tot 5 op regel 3, de boodschap `Waarde van i: 5` op het scherm geprint wordt. Bij de volgende evaluatie van `i < 5` wordt `False` geretourneerd en wordt overgegaan naar de **`else`**-suite.

Het `while-else`-statement met het `break` keyword

In het voorgaande voorbeeld is er weinig verschil tussen het uitvoeren van instructies die in de **`else`**-suite staan en instructies die buiten het **`while-else`** statement staan. Een belangrijk verschil ontstaat echter wanneer het **`break`** keyword gebruikt wordt. In dit geval zal de iteratie beëindigd worden zonder dat *conditie* `False` wordt. In dergelijke gevallen zal de **`else`**-suite dan ook **niet** uitgevoerd worden. Fragment 4.15 illustreert het gebruik van het **`while-else`**-statement.

Fragment 4.15: `while_else.py`

```
1 i = 0
2 while i < 5:
3     i = i + 1
4     print("Waarde van i: ", i)
5     if i == 3:
6         break
7 else:
8     print("De else suite!")
9
10 print("Programma stopt")
```

De output van dit fragment is:

```
Waarde van i:  1
Waarde van i:  2
```

```
Waarde van i:  3
Programma stopt
```

Merk op dat bij de output de regel `De else suite!` **niet** aanwezig is. Dit toont aan dat de `else`-suite niet werd uitgevoerd.

Opdracht 4.14 (`check_priemgetal.py`)

Een priemgetal is een natuurlijk getal, verschillend van 1, dat niet deelbaar is door een ander natuurlijk getal met uitzonderling van 1 en het getal zelf. Om te testen of een getal `n` een priemgetal is, kan voor `i = 2, ..., int(n**(1/2))` bepaald worden of `i` een deler is. Indien geen deler gevonden wordt, kan besloten worden dat n een priemgetal is. Schrijf een script dat deze strategie uitvoert. Wanneer een deler gevonden wordt, is het uiteraard overbodig om nog verdere kandidaat-delers te evalueren, de iteratie kan op dat moment reeds beëindigd worden. Volg de volgende stappen:

- Vraag de gebruiker om een natuurlijk getal in te geven.
- Ga na voor elke `i = 2, ..., int(n**(1/2))` of n deelbaar is door i (met `while`).
- Indien een deler gevonden, dan is het getal **geen** priemgetal en wordt dit gemeld aan de gebruiker (`break` de `while`-lus op dat moment).
- Indien geen deler gevonden, dan is het getal **wel** een priemgetal en wordt dit gemeld aan de gebruiker (in de `else`-suite).

Voorbeeld input/output is:

```
Geef getal: 15
15 is geen priemgetal, 3 is de kleinste deler.

Geef getal: 457
457 is een priemgetal.
```

Opdracht 4.15 (`chipkaart_pincode.py`)

Bij het ingeven van je PIN-code aan een geldautomaat, heb je exact 3 pogingen om je correcte PIN-code in teven. Na drie mislukte pogingen wordt de chipkaart geblokkeerd. Schrijf een script dat een gebruiker vraagt om een PIN-code in te geven. Indien deze correct is wordt getoond wat het saldo is van zijn rekening. Na drie mislukte pogingen krijgt de gebruiker de melding `Chipkaart geblokkeerd!` en wordt het programma beëindigd.

Volg de volgende stappen:

- Kies een 'geheime' PIN-code bv. `pin = "4567"` en een saldo, bv. `saldo = 1223.5`
- Maak een variabele aan die het aantal **resterende** pogingen bijhoudt (initieel 3).
- Maak gebruik van `while-else` met `break` om de gevraagde functionaliteit te bekomen.

Voorbeeld input/output is:

```
Typ uw PIN-code: 3258
PIN-code incorrect! Aantal resterende pogingen: 2
Typ uw PIN-code: 0066
PIN-code incorrect! Aantal resterende pogingen: 1
Typ uw PIN-code: 1258
PIN-code incorrect! Aantal resterende pogingen: 0
Chipkaart geblokkeerd!
```

```
Typ uw PIN-code: 4567
PIN-code correct! Uw saldo is 1223.5 Euro.
```

4.4 Het *for*-statement

Het `for`-statement is, naar analogie met het `while`-statement, een structuur die toelaat om bepaalde instructies meerdere keren uit te voeren.

4.4.1 Het basis *for*-statement

Het `for`-statement laat toe om te itereren over de elementen van een sequentie. Het volgende codefragment illustreert het gebruik van dit statement:

```
1 woord = "Hallo"
2
3 for k in woord:
4     print("De waarde van k is:", k)
```

De output van dit fragment is:

```
De waarde van k is: H
De waarde van k is: a
De waarde van k is: l
De waarde van k is: l
De waarde van k is: o
```

Op basis van deze output kunnen we het volgende concluderen:

- Het `print`-statement wordt vijf keer uitgevoerd (evenveel als het aantal karakters in de string `"Hallo"`).
- De variabele `k` noemen we de **loop variabele**. Bij elke iteratie zal de waarde van `k` wijzigen. We zeggen dat `k` **itereert** over de karakters van de string `"Hallo"`.

In het voorgaande voorbeeld werd geïtereerd over de karakters van een string object. Het `for`-statement laat echter ook toe om te itereren over andere objecten.

De algemene vorm van het `for`-statement is de volgende:

```
for loop_variable in iterable_object:
    # for-suite
```

We bespreken hierna elk van de onderdelen:

- Een `iterable_object` is een **object van een type waarover men kan itereren**, zoals bijvoorbeeld een object van het type string. Strings zijn niet de enige **iterable types**. In de volgende sectie worden een aantal bijkomende iterable types besproken.
- De **loop variable** is een variabele die in elke iteratie zal verwijzen naar een (volgende) element uit het `iterable_object`.
- De `for`-suite (of body) is een sequentie van instructies[4] die in elke iteratie zal worden uitgevoerd. De syntax van de `for` suite is identiek aan die van de `while`-suite. **Indentatie** bepaalt welke statements tot de `for`-suite behoren, en net zoals bij de `while`-suite kan ook de `for`-suite op zijn beurt `if-else` statements, `while` statements of andere `for` statements bevatten.

Opdracht 4.16 (`basen_tellen.py`)
Schrijf een script dat vraagt naar een DNA-sequentie (dus enkel met de basen: A, C, G of T) en de frequentie/aantal weergeeft voor elke base. Een mogelijke input/output is:

```
Geef een DNA-sequentie (enkel A, C, G of T gebruiken):
ACGTGTACAGFCTAAAGTCCAGTCCGAATA
F is niet toegelaten!
Aantal A: 10
Aantal C: 7
Aantal G: 6
Aantal T: 6
```

Vóór je begint:

- Lees Sectie 4.4.1 aandachtig.
- Uiteraard kan je een variabele `teller` initialiseren en vervolgens updaten in de `for`-suite:

```
woord = "Hallo"
teller = 1
for k in woord:
    print("Iteratie", teller, "- De waarde van k is:", k)
    teller = teller + 1
```

[4] Ook nu bevat de `for`-suite **minstens één** instructie.

Uitbreiding: zorg ervoor dat de aantallen enkel worden weergeven op het scherm indien de DNA-sequentie enkel uit A, C, G of T bestaat. Indien een letter in de DNA-sequentie niet A, C, G of T is, moet je een gepaste melding teruggeven.

Mogelijke input/output is:

```
Geef een DNA-sequentie (enkel A, C, G of T gebruiken):
AAAXTCGAATCGGGT
X is niet toegelaten!
```

Opdracht 4.17 (`omgekeerde_zin.py`)
Schrijf een script dat naar een zin vraagt en deze zin omgekeerd op het scherm weergeeft. Een mogelijke input/output is:

```
Typ een zin:
De zon schijnt vandaag.
.gaadnav tnjihcs noz eD
```

4.4.2 Geneste *for*-statements

De body van een `for`-statement kan op zijn beurt één of meerdere nieuwe `for`-statements bevatten. In dit geval spreken we, net als bij `while`-statements, van **geneste** `for`-lussen. Ook hier maken we een onderscheid tussen de **outer loop** en de **inner loop** (die deel uitmaakt van de suite van de outer loop). Bij elke iteratie van de outer loop zal de inner loop **volledig** worden doorlopen.

In Fragment 4.16 wordt dit principe geïllustreerd. Daarbij wordt er over DNA-basen A, G, C, T geïtereerd m.b.v. twee geneste `for`-lussen:

Fragment 4.16: `twee_basen_sequenties.py`

```
1 basen = "AGCT"
2 for ch1 in basen:
3     for ch2 in basen:
4         tweebasen = ch1 + ch2        # concatenatie van twee karakters
5         print(tweebasen, end = " ")
6     print() # nieuwe regel
```

De output is:

```
AA AG AC AT     # alle koppels die beginnen met A
GA GG GC GT     # alle koppels die beginnen met G
CA CG CC CT     # alle koppels die beginnen met C
TA TG TC TT     # alle koppels die beginnen met T
```

Opdracht 4.18 (`tekens_tellen.py`)

Vul het onderstaande codefragment aan, zodat de gebruiker te zien krijgt hoeveel *speciale tekens* de opgegeven tekst bevat.

```
tekst = input("Geef een tekst in: ")
speciale_tekens = "!/*+-"
teller = 0

for kar in tekst:
    for .......  in .........:
        if ..................... :
            ..........................
            ..........................

print("De tekst bevat", teller, "speciale tekens.")
```

De output is:

```
Geef een tekst in: Het is de 21-ste vandaag!
De tekst bevat 2 speciale tekens.
```

Opdracht 4.19 (`verschillende_codons.py`)

Een codon is een opeenvolging van drie basen (de vier basen zijn `A`, `C`, `T` en `G`). De string `"ATC"` stelt bv. een codon voor. Schrijf een script dat alle codons, die bestaan uit **verschillende** nucleotiden, op het scherm print. Het codon `ACT` is toegestaan, maar bv. `ACC` is niet toegestaan. Geef de codons weer in tabelvorm, zoals hieronder getoond. **Opmerking**: laat de codons eerst onder elkaar op het scherm verschijnen, dat is eenvoudiger.

De mogelijke output is:

```
AGC AGT ACG ACT ATG ATC
GAC GAT GCA GCT GTA GTC
CAG CAT CGA CGT CTA CTG
TAG TAC TGA TGC TCA TCG
```

Vóór je begint:

- Gebruik Fragment 4.16 als startpunt

4.4.3 Iterable objects

Iteratoren zijn een groep van gegevenstypes die een **iteratieprotocol** implementeren. Dit wil zeggen dat ze, wanneer gevraagd, het volgende element in een rij van elementen kunnen berekenen/bepalen en retourneren. Een volledige discussie van iteratoren valt buiten het bestek van deze cursus[5].

Een *iterable object* is een **object dat** een *iterator* **kan retourneren**. Een voorbeeld zijn strings. Een string object kan een *string iterator* retourneren die kan itereren over alle karakters in een string. Daarbij zijn twee functies belangrijk:

- de functie `iter()`: deze functie aanvaardt een **iterable object als argument** en retourneert de iterator van dit object.
- de functie `next()`: deze functie aanvaardt een **iterator als argument** en retourneert het volgende element in de sequentie (berekend door de iterator).

Een illustratie:

```
>>> woord = "Hallo"
>>> woord_iterator = iter(woord)  # aanmaken iterator
>>> next(woord_iterator)          # opvragen volgende element in sequentie
'H'
>>> next(woord_iterator)
'a'
```

Indien de iterator geen nieuwe elementen meer kan opwerpen, wordt een exception (een foutmelding) opgeworpen:

```
>>> next(woord_iterator)
'o'
>>> next(woord_iterator)
Traceback (most recent call last):
    next(woord_iterator)
StopIteration
```

Objecten van het type `int`, `float` en `bool` zijn niet iterable.
De functie `iter()` kan men ook gebruiken om na te gaan of een bepaald object iterable is. Zo kan men vaststellen dat objecten van het type `int`, `float` of `bool` niet iterable zijn. Dit is geen onverwacht resultaat omdat de waarden van deze objecten, in tegenstelling tot strings, geen sequentie definiëren waarover geïtereerd zou kunnen worden.

[5]Op dit punt in de cursus is een volledige beschrijving van iteratoren nog niet aan de orde. Strikt genomen is een iterator een object waarvoor een `__next__` methode geïmplementeerd is. Wanneer deze methode wordt opgeroepen, zal een waarde (meestal de volgende waarde in een rij van waarden) geretourneerd worden. Voor meer info zie, `https://docs.python.org/3/library/stdtypes.html#iterator-types`

```
>>> iter(5)
TypeError: 'int' object is not iterable

>>> iter(True)
TypeError: 'bool' object is not iterable
```

De functie `range()` en het gegevenstype `list`.
Van de gegevenstypes die tot op dit ogenblik aan bod kwamen, zijn enkel strings iterable. In wat volgt, komen nog twee andere gegevenstypes aan bod die iterable zijn.

1. Objecten van het type **`range`**: dit zijn objecten waarvan de iteratoren *range iteratoren* zijn, en die toelaten om te itereren over een rij van getallen. Een **`range`**-object kan eenvoudig worden aangemaakt met de functie **`range()`**.

   ```
   >>> a = range(1, 6) # een range object (rij van getallen 1, 2,..., 5)
   >>> a_iterator = iter(a)  # een range iterator
   >>> next(a_iterator)
   1
   >>> next(a_iterator)
   2
   ```

2. Objecten van het type **`list`**: dit zijn objecten van een collectietype. Dat wil zeggen dat het zich gedraagt als een **container** voor andere objecten. **`list`** objecten kunnen een list iterator genereren, die kan gebruikt worden om te itereren over de elementen van de **`list`**.

   ```
   >>> a = ["ma", "di", "wo", "do",\
            "vrij", "za", "zo"]  # list waarvan elementen str zijn
   >>> a_iterator = iter(a)  # een list iterator
   >>> next(a_iterator)
   "ma"
   >>> next(a_iterator)
   "di"
   ```

Merk op dat we de functies **`iter()`** en **`next()`** **niet nodig hebben in de syntax van het `for`-statement**. Het **`for`**-statement zal de bijhorende iteratoren zelf, op de achtergrond, aanmaken en oproepen. Dit wordt hieronder geïllustreerd voor **`range`**-iteratoren:

```
for i in range(1, 6):
    print(i, end = "   ")       # 3 spaties
```

Met output:

```
1   2   3   4   5
```

Eenzelfde manier van werken kan gebruikt worden om te itereren over de elementen van een **list**:

```
dagen = ["ma", "di", "wo", "do", "vrij", "za", "zo"]
for dag in dagen:
    print(dag, end = "   ")     # 3 spaties
```

Dit levert de volgende output:

```
ma   di   wo   do   vrij   za   zo
```

4.4.4 Iteratie met `range` iteratoren

In veel toepassingen moet men itereren over een rij van getallen van de vorm:

$$a, \quad a+k, \quad a+2k, \quad a+3k, \quad ...$$

Een voorbeeld is een toepassing waarbij men de som wenst te bereken van alle **oneven** getallen tussen 1 en 100 (100 inclusief). Het onderstaande fragment print deze getallen naar het scherm en berekent de som:

```
som = 0
for i in range(1, 100, 2):   # 2 --> stapgrootte
    print(i)
    som = som + i
print("De som is:", som)
```

De output is:

```
1
3
5
...
99
De som is: 2500
```

De functie **range()** creëert in het voorbeeld hierboven een **range**-iterator die de getallen 1, 3, 5, ..., 99 retourneert. De functie signature van **range()** kan worden afgeleid uit de output van **help(range)**:

```
>>> help(range)
range(start, stop[, step]) -> range object
 |
 |  Return an object that produces a sequence of integers from start
 |  (inclusive) to stop (exclusive) by step.
```

We bespreken hierna de parameters:

- `start`: de **kleinste** waarde van de sequentie. Dit argument is optioneel, de default waarde is 0.
- `stop`: bepaalt de **grootste** waarde van de sequentie. **Let op**: de waarde die aan `stop` wordt toegekend, maakt **nooit** deel uit van de sequentie. De iteratie zal steeds eindigen vóór `stop`.
- `step`: de stapgrootte. Dit argument is optioneel, de default waarde is 1.

Uit de bespreking van de parameters blijkt dat de functie `range()` kan opgeroepen worden met één, twee of drie parameters. De volgende voorbeelden illustreren dit gebruik:

- Voorbeeld 1:

```
for i in range(5):
    print(i, end = " ")
```

  Met output:

```
0 1 2 3 4
```

- Voorbeeld 2:

```
for i in range(3, 10):
    print(i, end = " ")
```

  Met output:

```
3 4 5 6 7 8 9
```

- Voorbeeld 3:

```
for i in range(1, 20, 4):
    print(i, end = " ")
```

  Met output:

```
1 5 9 13 17
```

- Voorbeeld 4:

```
for i in range(35, 0, -3):
    print(i, end = " ")
```

  Met output:

```
35 32 29 26 23 20 17 14 11 8 5 2
```

Opdracht 4.20 (`product.py`)

Beschouw het onderstaande codefragment:

```
i = 0
product = 1
while i < 20:
    i = i + 1
    if i % 3 == 0:
        product = product * i
print(product)  # waarde is 524880
```

Vul het onderstaande fragment aan zodat de output van het script dezelfde is als die van het voorgaande.

```
product = 1
for i in range(..........., ..........., ..........):
    product = product * i
print(product)
```

Opdracht 4.21 (`getallenreeks.py`)

Door gepast gebruik te maken van een **`for`** lus, de range functie, type conversie (van **`int`** naar **`str`** en omgekeerd) en concatenatie kan men met vier regels code de onderstaande output (links) bekomen. Pas vervolgens deze code aan om de output rechts te bekomen.

```
1
12
123
1234
12345
123456
1234567
12345678
123456789
```

```
1 * 8 + 1 = 9
12 * 8 + 2 = 98
123 * 8 + 3 = 987
1234 * 8 + 4 = 9876
12345 * 8 + 5 = 98765
123456 * 8 + 6 = 987654
1234567 * 8 + 7 = 9876543
12345678 * 8 + 8 = 98765432
123456789 * 8 + 9 = 987654321
```

Opdracht 4.22 (`perfecte_getallen_checker_for.py`)

In Opdracht 4.10 werd nagegaan of een getal een perfect getal is: een getal is perfect indien de som van de delers gelijk is aan het getal zelf, bv. $6 = 1 + 2 + 3$. Indien een getal niet perfect is, dan maakt men verder onderscheid tussen **abundante** en **deficiënte** getallen:

- Een getal is *abundant* indien de som van zijn delers **groter** is dan het getal zelf.
- Een getal is *deficient* indien de som van zijn delers **kleiner** is dan het getal zelf.

Schrijf een script dat een getal als input vraagt aan de gebruiker, en op het scherm print of dit getal perfect, abundant of deficiënt is. Maak hierbij gebruik van een `for`-statement.

```
Geef een natuurlijk getal: 18
18 is abundant

Geef een natuurlijk getal: 213
213 is deficient
```

4.4.5 Het `for-else` statement en het `break` keyword

Wanneer het `break` keyword zich in een `for`-suite bevindt, zal bij het evalueren van dit keyword de `for`-lus onmiddellijk beëindigd worden. Het keyword `break` heeft hier dus dezelfde functie als bij de `while`-lus. Het onderstaande voorbeeld (rechts) illustreert het gebruik van dit keyword. Merk op dat in dit voorbeeld de rekentijd die nodig is om te bepalen of een getal een priemgetal is gevoelig lager wordt door het gebruik van `break`.

```
a = input("Geef een getal: ")
is_priemgetal = True

for i in range(2, int(a)):
    if a % i == 0:
        is_priemgetal = False
        # zonder break

if is_priemgetal:
    print("Priemgetal")
```

```
a = input("Geef een getal: ")
is_priemgetal = True

for i in range(2, int(a)):
    if a % i == 0:
        is_priemgetal = False
        break     # met break

if is_priemgetal:
    print("Priemgetal")
```

Naar analogie met het `while-else` statement, kan ook het `for`-statement worden uitgebreid met een `else`-statement. Dit statement heeft de volgende structuur:

```
for loop_variable in iterable_object:
    # for suite
else:
    # else suite
```

Wanneer de *loop variabele* **alle waarden in** het ***iterable object*** heeft **doorgelopen**, zal de `else`-suite **eenmalig** worden **uitgevoerd**. Wanneer het iteratieproces beëindigd wordt d.m.v. het `break`-statement, zal de `else`-suite **niet** worden uitgevoerd.

Opdracht 4.23 (`nucleotideAtellen.py`)

Een DNA sequentie bestaat uit de nucleotiden A, C, G en T. Het onderstaande codefragment kan gebruikt worden om het aantal keer dat `A` voorkomt in een DNA sequentie te tellen. Men zou

kunnen stellen dat zodra een DNA string één of meerdere karakters bevat die geen A, C, G of T zijn, deze string niet geldig is. Pas het onderstaande codefragment aan zodat in dergelijke gevallen de berekening onmiddellijk stopt, een foutboodschap verschijnt en het totaal aantal A's **niet** op het scherm verschijnt. Maak gebruik van `break` en `for-else`:

```
DNA = input("Geef een DNA-string: ")
teller_A = 0
for nucl in DNA:
    if nucl == "A":
        teller_A += 1   # analoog aan teller_A = teller_A + 1
.........................................
.........................................
.........................................
.........................................
.........................................
.........................................
```

Voorbeeld input/output:

```
Geef een DNA-string: AACCTTAACG
Aantal A's: 4

Geef een DNA-string: AfACTTG
Geen geldige sequentie
```

4.5 *Intermezzo*: file input/output

In de voorgaande secties werd (gebruikers)input steeds ingegeven via het keyboard en werd output steeds getoond als tekst in de console. Nu breiden we deze input/output uit naar (data)bestanden (Engels: *files*). In Hoofdstuk 9 zullen we bestanden meer in detail bespreken. Hier geven we alvast een eerste introductie.

4.5.1 Bestanden en bestandslocaties

De Python interpreter moet soms tekstbestanden of binaire bestanden inlezen of wegschrijven, bijvoorbeeld bij

- Het gebruik van een (nieuwe) module. De module wordt gelezen met het statement `import`.
- Het inlezen van tekst uit bv. een gegevensbestand (file input).
- Het wegschrijven van tekst naar bv. een gegevensbestand (file output).

Een bestands-*path* is een sequentie van karakters die verwijst naar een locatie op een (harde)schijf. Indien we veronderstellen dat het tekstbestand `bloemen.txt` op de C-schijf van je computer staat in de map `WetProg`, dan wordt het **absolute** bestands-*path* aangegeven met de volgende string

- Op een **Windows** systeem: `C:\WetProg\bloemen.txt`
- Op een **Linux/MacOS** systeem: `C:/WetProg/bloemen.txt`

Indien men in broncode naar een bestand verwijst, dan is het belangrijk dat de interpreter het volledige bestands-*path* kan achterhalen, en dus niet enkel de bestandsnaam `bloemen.txt`. Het *path* is steeds een *string* in Python. Deze *paths* kunnen soms vrij lang zijn. Daarom is het praktisch om je huidige map (*Current Directory*) te veranderen naar de map waar je werkbestanden in staan (werkmap of *Working Directory*). Deze map waarin je dan "werkt", zullen we de huidige werkmap (*Current Working Directory*) noemen.

Om in Python naar de huidige werkmap te navigeren moet je eerst de module `os` importeren (`os` is de afkorting van *operating system*).

Met de functie `os.getcwd()`, kan je de naam van de huidige werkmap opvragen:

```
>>> import os                # importeren van module os
>>> os.getcwd()              # opvragen van werkmap
'D:\\Users\\dkose'
```

Indien je systeem een C-schijf heeft waarop de map `WetProg` zich bevindt, dan kan deze map als werkmap ingesteld worden met de volgende instructies:

```
>>> import os                 # importeren van module os
>>> os.chdir("C:/WetProg")    # wijzigen van werkmap
```

De werkmap werd door deze instructies gewijzigd naar `"C:/WetProg"`. Wanneer men informatie uit bestanden wenst in te lezen, dan zal de interpreter automatisch in de werkmap gaan zoeken naar deze bestanden. Je hoeft in dit geval dus enkel de bestandsnaam in te geven[6].

4.5.2 Inlezen uit en wegschrijven naar een tekstbestand (I/O)

Met de termen **inlezen** en **wegschrijven** bedoelen we

- **inlezen**: tekst uit een tekstbestand inlezen vanop de *hard disk* en converteren naar een Python-object.
- **wegschrijven**: een Python-object converteren naar een (gestructureerd) tekstbestand

[6]Merk op dat, bij het uitvoeren van het statement `import mijnmodule`, de interpreter op zoek gaat naar het bestand `mijnmodule.py` (of de map `mijnmodule` voor packages) in de mappen opgelijst in `sys.path`. De working directory wordt niet automatisch aan deze lijst toegevoegd. Bij het uitvoeren van een script wordt de map waarin dat script zich bevindt wel toegevoegd aan `sys.path`.

Python biedt uitgebreide I/O-functionaliteiten die in Hoofdstuk 9 in detail besproken worden. De module (of package) `infoFunWP` is een eenvoudige wrapper (ontwikkeld voor deze cursus) rond deze I/O functionaliteiten en bevat onder andere de volgende functies: `listRead()`, `listReadValues()` en `listWrite()`. Deze functies laten op een eenvoudige wijze toe tekst in te lezen uit een tekstbestand en tekst weg te schrijven naar een tekstbestand:

- `naam_list = listRead("bestandsnaam.txt")`
  leest het tekstbestand `bestandsnaam.txt` in en retourneert een *list* van *strings* waarbij elke regel in het tekstbestand een element is van de lijst. Nieuwelijnkarakters worden verwijderd.
- `naam_list = listReadValues("bestandsnaam.txt")`
  leest het tekstbestand `bestandsnaam.txt` in en retourneert en *list* van *floats* waarbij elke regel in het tekstbestand gecast wordt naar een *float* en een element wordt van de lijst.
- `listWrite("bestandsnaam.txt", naam_list)`
  schrijft de *list* `naam_list` weg naar een tekstbestand `bestandsnaam.txt`. **Elk element** uit de *list* wordt **op** een **nieuwe regel** geplaatst in `bestandsnaam.txt`.

Met deze functionaliteiten kunnen we op een meer praktische en realistische manier omgaan met gegevens, zoals bv. grotere gegevenshoeveelheden inlezen en onze programma-output opslaan in een tekstbestand.

De module `infoFunWP` maakt deel uit van een package met dezelfde naam dat niet behoort tot de Python Standard Library (zie Sectie 3.6). Om gebruik te kunnen maken van functies uit de module `infoFunWP`, moet het package `infoFunWP` **geïnstalleerd** worden voor het kan **geïmporteerd** worden.

Installeren

1. Start een *Terminal* of (Anaconda) *Command Prompt*:
   - In **VS Code**: klik op Terminal in het bovenste menu. Selecteer *New Terminal*.
   - Op een **Windowstoestel**: tik *(Anaconda) Command Prompt* of *cmd* in in de zoekbalk op het bureaublad.
   - Op een **macOS**: gebruik de toetscombinatie `Cmd + Spatie` en typ `Terminal`.
2. Voer het volgende commando uit:

   ```
   pip install infoFunWP
   ```

   Deze instructie zal de package `infoFunWP` downloaden en installeren in de **actieve** Python environment (of *system wide*).

Importeren

- Onderstaande instructie zal de module importeren

```
import infoFunWP as infoFun
```

Illustratie van het gebruik van de module `infoFunWP`

Het bestand `aminozuren.txt` bevat een 20-tal regels met op elke regel de naam van een aminozuur. Voer de volgende voorbeeldinstructies uit in een Python console:

```
>>> import infoFunWP as infoFun   # importeren van de module infoFunWP
```

Lees het bestand `aminozuren.txt` in en plaats de inhoud in een *list*:

```
>>> aminozuren_list = infoFun.listRead("aminozuren.txt")
```

Geef de inhoud van de *list* weer naar het scherm:

```
>>> print(aminozuren_list)
['alanine', 'arginine', 'asparagine', 'aspartic', 'cysteine', 'glutamine',
'glutamic', 'glycine', 'histidine', 'isoleucine', 'leucine', 'lysine',
'methionine', 'phenylalanine', 'proline', 'serine', 'threonine',
'tryptophan', 'tyrosine', 'valine']
```

Het resultaat is een lijst van *strings*. In Sectie 4.7 bekijken we hoe we met deze strings verder kunnen werken.

Het bestand `data_waarden.txt` bevat een 6-tal regels met op elke regel één numerieke waarde. Lees dit bestand in en plaats de inhoud in een *list*:

```
>>> waarden_lst = infoFun.listReadValues("data_waarden.txt")
```

Geef de inhoud van de *list* weer naar het scherm:

```
>>> print(waarden_lst)
[12.584, 15.33, 16.397, 11.005, 14.379, 12.045]
```

Het resultaat is een lijst van *floats*.

Wensen we een lijst te bewaren in een (nieuw) bestand, dan kan daarvoor de functie `listWrite()` gebruikt worden. Maak de volgende *list* aan

```
>>> celcyclus_list = ["G1", "S", "G2", "Pro", "Meta", "Ana", "Telo"]
```

Schrijf de *list* `celcyclus_list` weg naar het tekstbestand `celcyclus.txt`:

```
>>> infoFun.listWrite("celcyclus.txt", celcyclus_list)
```

Bekijk nu de inhoud van het bestand `celcyclus.txt` met een teksteditor. Elk element van de *list* `celcyclus_list` wordt bewaard als een afzonderlijke regel in dit bestand.

4.6 Intermezzo: formatteren van strings met f-string

Opmerking: de bespreking van f-strings komt uitgebreid aan bod in het hoofdstuk over strings (Hoofdstuk 6). Wat hier volgt, is een korte bespreking omdat ze soms in de oplossing van een aantal opdrachten gebruikt worden.

Het gebruik van de functie `print()` is relatief eenvoudig maar heeft geen controle over het **formatteren** van de output. Met formatteren bedoelen we o.a.:

- **hoeveel plaats** (karakters) er voorzien moet worden voor een getal,
- **hoeveel cijfers na de komma** (precisie) moeten er getoond worden bij een *float*,
- links of rechts **uitlijnen**,
- **lege karakterplaatsen opvullen** met bv. nullen, enz.

Met f-strings kunnen we op een flexibele en intuïtieve manier strings opmaken in Python, hetgeen de code vaak eenvoudiger en leesbaarder maakt. De f staat voor *formatted* en geeft aan de string die er op volgt delen bevat die geformatteerd moeten worden. **Om een f-string te gebruiken, plaatsen we de letter f (of F) vóór de aanhalingstekens van de string en gebruiken we accolades { } om de variabelen aan te geven die we willen formatteren en invoegen in de string.**

Een eerste voorbeeld

```
>>> codon = "CGT"
>>> fractie = 0.2095
>>> f"De fractie van het codon {codon} is {fractie}"
```

Hierin betekent:

- `{codon}`: plaats hier de inhoud van de variabele `codon` (die een string voorstelt),
- `{fractie}`: plaats hier de inhoud van de variabele `fractie` (die een decimaal getal voorstelt).

Het resultaat van deze f-string is:

```
'De fractie van het codon CGT is 0.2095'
```

Opmerking: in hetgeen volgt, stellen `s`, `d` en `f` *format specifiers* voor. Deze worden gebruikt om de opgegeven waarden te formatteren.

Een tweede voorbeeld

In veel toepassingen worden getallen met een **beperkt aantal cijfers na de komma** weergegeven. We kunnen dit realiseren door *format specifiers* te gebruiken:

```
>>> codon = "CGT"
>>> fractie = 0.2095
>>> f"De fractie van het codon {codon:s} is {fractie:6.2f}"
```

Hierin betekent:

- `{codon:s}`: plaats hier de inhoud van de variabele `codon` die een string voorstelt (aangeduid met `s`, staat voor *string*),
- `{fractie:6.2f}`: voorzie 6 karakterplaatsen voor de variabele `fractie` die een decimaal getal voorstelt (aangeduid met `f`, staat voor *float*) en gebruik 2 plaatsen voor het deel na de komma.

Het resultaat van deze f-string is:

```
'De fractie van het codon CGT is   0.21'
```

Merk op dat in het resultaat

- de fractie werd afgerond en er 2 extra spaties vóór 0.21 staan: er werden 6 plaatsen voorzien waarvan er 4 werden gebruikt voor 0.21, de **overige 2 plaatsen werden opgevuld met spaties**, en
- een punt ook een plaats inneemt.

Andere voorbeelden

Getallen worden *default rechts* uitgelijnd, *strings* worden *default links* uitgelijnd:

```
>>> codon = "CGT"
>>> f"{codon:7s}"
>>> aantal = 53
>>> f"{aantal:7d}"
```

Hierin staat `{aantal:7d}`: voorzie 7 karakterplaatsen voor de variabele `aantal` die een geheel getal voorstelt (aangeduid met `d`, staat voor *digit*).

Het resultaat van deze f-strings is:

```
'CGT    '        # links uitgelijnd: 4 spaties na CGT
'     53'        # rechts uitgelijnd: 5 spaties vóór 53
```

Getallen links uitlijnen kan met de specifier `<`, rechts uitlijnen met de specifier `>` en centreren met ∧:

```
>>> codon = "CGT"
>>> f"{codon:>7s}"
>>> f"{codon:^7s}"
>>> aantal = 53
>>> f"{aantal:<7d}"
```

Het resultaat van deze f-strings is:

```
'    CGT'        # rechts uitgelijnd: 4 spaties vóór CGT
'  CGT  '        # gecentreerd: 2 spaties vóór en 2 spaties na CGT
'53     '        # links uitgelijnd: 5 spaties na 53
```

4.7 Itereren over de elementen van een lijst

In Sectie 4.4.3 werd reeds aangehaald dat lijsten (objecten van type `list`) *iterable objecten* zijn, waaruit volgt dat ze kunnen gebruikt worden in een `for`-statement. In deze sectie gaan we dieper in op een aantal toepassingen van dit principe.

Beschouw het volgende codefragment:

```
1 aminozuren_list = ["alanine", "arginine", "asparagine"]
2
3 for az in aminozuren_list:
4     print(az)
```

De output is:

```
alanine
arginine
asparagine
```

Hieruit blijkt dat de variabele `az` zal itereren over de elementen van de lijst `aminozuren_list`. In dit voorbeeld zijn deze elementen strings. Men kan dus, indien gewenst deze strings verder analyseren. In het onderstaande codefragment wordt bepaald hoeveel keer het karakter `a` voorkomt in deze strings in `aminozuren_list`.

```
1 aminozuren_list = ["alanine", "arginine", "asparagine"]
2 aantal_a = 0
3
4 for az in aminozuren_list:
5     for karakter in az:       # tellen aantal a's
6         if karakter == "a":
7             aantal_a += 1
8 print("Aantal a's", aantal_a)
```

Opdracht 4.24 (`aminozuren.py`)

Schrijf een script dat het bestand `"aminozuren.txt"` met de aminozuren inleest en de aminozuren voorzien van een nummering op het scherm weergeeft op de onderstaande manier.

```
[ 1]  alanine
[ 2]  arginine
[ 3]  asparagine
[ 4]  aspartic
....
[19]  tyrosine
[20]  valine
```

Tip: Let op de uitlijning van de nummering!

Vóór je begint

- Lees Secties 4.5.2 en 4.6 aandachtig.

Opdracht 4.25 (`bloedgroepen_tellen.py`)

Het bestand `bloedgroep_data_belgie.txt` bevat de bloedgroep van een steekproef van 1000 Belgen. Bepaal, o.b.v. deze gegevens, de geobserveerde (procentuele) frequentie van de bloedgroepen `O`, `A`, `B` en `AB`. Schrijf een script dat het bestand `bloedgroep_data_belgie.txt` inleest, de gevraagde percentages berekent en weergeeft zoals hieronder getoond:

```
Percentages bloedgroepen in Belgie:
Percentage O:  XY.Z
Percentage A:  XY.Z
Percentage B:  XY.Z
Percentage AB: XY.Z
```

Het formaat `XY.Z` duidt op een *float* weergegeven in een ruimte van vier karakters met één cijfer na de komma.

4.8 Enumerate

Beschouw opnieuw het volgende codefragment:

```
for az in aminozuren_list:
    print(az)
```

Met output:

```
alanine
arginine
asparagine
```

Veronderstel nu dat we, naast de aminozuren ook de index (het nummer) van het item willen weergeven, zoals in de volgende output:

```
1 alanine
2 arginine
3 asparagine
```

Een eerste manier om deze output te bekomen is te werken met een teller en deze te updaten binnen de `for`-lus:

```
nr = 0  # initialisatie teller
for az in aminozuren_list:
    nr = nr + 1    # update teller
    print(nr, az)
```

Een tweede manier is gebruik te maken van de functie `enumerate()`. Deze functie verwacht als input een *iterable object* (bv. `aminozuren_list`) en retourneert:

1. de index (het nummer) van de huidige iteratie, en
2. de waarde van het item in de huidige iteratie.

Met de functie `enumerate()` kan de `for`-lus in bovenstaand codefragment herschreven worden als:

```
for i, az in enumerate(aminozuren_list):
    print(i, az)
```

De output is:

```
0 alanine
1 arginine
2 asparagine
```

Merk op dat het tellen default bij 0 start en niet bij 1. Om deze startwaarde aan te passen, kan de waarde van de parameter `start` gewijzigd worden:

```
    print(i, az)
```

De output is nu:

```
1 alanine
2 arginine
3 asparagine
```

Opdracht 4.26 (`aminozuren.py`)

Pas je script `aminozuren.py` uit Opdracht 4.24 aan zodat je nu gebruik maakt van de functie `enumerate()` om de aminozuren te nummeren. Je zou hetzelfde resultaat moeten bekomen.

4.9 Gemengde opdrachten

Opdracht 4.27 (`vierkantsvergelijking.py`)
De wortels van een vierkantsvergelijking $ax^2 + bx + c = 0$ worden gegeven door

$$x_{\pm} = \frac{-b \pm \sqrt{D}}{2a}$$

waarin $D = b^2 - 4ac$ de discriminant voorstelt. Indien de discriminant positief is, zijn er twee reële wortels, indien de discriminant nul, is er slechts één reële wortel en als de discriminant negatief is, zijn er geen reële wortels.

Schrijf een programma dat vraagt naar de coëfficiënten a, b en c van een vierkantsvergelijking en afhankelijk van de waarde van de discriminant twee of één reële oplossingen teruggeeft naar het scherm. Je programma moet ook controleren of a niet nul is (anders hebben we geen vierkantsvergelijking). Indien er geen reële oplossingen zijn moet de gebruiker ook ingelicht worden.

Een mogelijke input/output is:

```
Beschouw ax^2 + bx + c = 0 met a (niet nul), b en c reele coefficienten.
Geef (niet nul) coefficient a: -4.2
Geef coefficient b: 6.7
Geef coefficient c: 8
Twee reele wortels.
x1 = -0.7964192360763666 en x2 = 2.391657331314462
```

```
Beschouw ax^2 + bx + c = 0 met a (niet nul), b en c reele coefficienten.
Geef (niet nul) coefficient a: 0
De coefficient a is nul!
```

Opdracht 4.28 (`loterij_twee_cijfers.py`)
Bij een loterijspel moet je twee cijfers $(0 - 9)$ raden. Indien je dezelfde cijfers raadt in de juiste volgorde, win je € 10000. Indien je dezelfde cijfers raadt maar niet in de juiste volgorde, win je € 3000. Indien je slechts één van de cijfers raadt, win je € 1000. Indien je geen van de cijfers raadt, win je niets. Schrijf een programmaatje dat twee willekeurige loterij cijfers genereert, de gebruiker achtereenvolgens vraagt naar twee cijfers, de loterij cijfers weergeeft en ook weergeeft hoeveel je gewonnen hebt. Gebruik de `random.randint()` functie (zie Sectie 3.6.3 voor een illustratie).

Mogelijke input/output zijn:

```
Geef het eerste cijfer (0 - 9): 7
Geef het tweede cijfer (0 - 9): 6
De loterij cijfers zijn: 1 en 9
Jammer, geen zelfde cijfers.
```

```
Geef het eerste cijfer (0 - 9): 0
Geef het tweede cijfer (0 - 9): 5
De loterij cijfers zijn: 5 en 5
Een zelfde cijfer: je wint 1000 euro!
```

Opdracht 4.29 (`natuurlijk_getal.py`)

Schrijf een script dat vraagt naar een natuurlijk getal en dit getal als *int* print naar het scherm. Echter, indien geen natuurlijk getal ingevoerd wordt, moet het programma blijven vragen naar een natuurlijk getal. Het programma wordt pas beëindigd wanneer een natuurlijk getal werd ingegeven. Maak hierbij gebruik van de *string*-methode `isnumeric()`.

```
>>> "789k3".isnumeric()
False
>>> "78903".isnumeric()
True
```

Een mogelijke input/output is:

```
Geef een natuurlijk getal: 45.0
Fout: probeer opnieuw. Geef een natuurlijk getal: 4a
Fout: probeer opnieuw. Geef een natuurlijk getal: -90
Fout: probeer opnieuw. Geef een natuurlijk getal: 123
Het natuurlijk getal is: 123
```

Opdracht 4.30 (`caesar_vercijfering.py`)

Schrijf een programma dat naar een zin vraagt, alle letters in dat woord omzet naar hoofdletters en vervolgens deze zin vercijfert volgens de zgn. Caesarvercijferingsmethode-3 (3 is de zgn. sleutelparameter). De vercijferde zin wordt dan op het scherm weergegeven.

- Letters omzetten naar hoofdletters kan je doen met de *string*-methode *str*.`upper()` (`https://docs.python.org/3/library/stdtypes.html`)
- Bij het vercijferen moet je enkel rekening houden met de alfabetische (hoofd)letters, dit kan je doen met de *string*-methode *str*.`isalpha()` die nagaat of een karakter een alfabetische letter is (`https://docs.python.org/3/library/stdtypes.html`)
- Bij de Caesarvercijferingsmethode-3 moet je het alfabet 3 letters naar links verschuiven: het alfabet moet je cyclisch beschouwen, dit doe je met de modulo-operator `%`. De letter `A` wordt dan `D`, de letter `B` wordt dan `E`, ..., de letter `Z` wordt dan `C`. Hieronder zie je de twee alfabetten (in klare tekst en vercijferd) uitgelijnd onder elkaar

  ```
  Klare tekst:        ABCDEFGHIJKLMNOPQRSTUVWXYZ
  Vercijferde tekst:  DEFGHIJKLMNOPQRSTUVWXYZABC
  ```

Gebruik verder de ingebouwde functie `ord()` inwerkend op een karakter om de overeenkomstige *integer* notatie van dit karakter te bekomen, en de ingebouwde functie `chr()` inwerkend op een *integer* om het overeenkomstige karakter te bekomen (`https://docs.python.org/3/library/functions.html#ord`).
Kortom, volgend stukje code moet je in je programma implementeren: als `kar` een bepaalde karakter is in het woord, dan wordt de nieuwe (geëncrypteerde) karakter `nieuwe_kar`:

```
nieuwe_kar = chr(((ord(kar) - ord("A")) + 3) % 26 + ord("A"))
```

Een mogelijke input/output is:

```
Typ een zin:
Ontmoeting: 14h30, aan station!
Klare tekst:         ONTMOETING: 14H30, AAN STATION!
Vercijferde tekst:   RQWPRHWLQJ: 14K30, DDQ VWDWLRQ!
```

Opdracht 4.31 (`perfecte_getallen_zoeken.py`)
Gebruik de code in Opdracht 4.22 om te zoeken met een `for`-lus en de functie `range()` naar alle perfecte getallen tussen 2 en een bepaalde bovengrens ingegeven door de gebruiker. Geef verder ook aan hoeveel getallen *abundant* en *deficient* zijn tussen 2 en de ingegeven bovengrens.

- Een getal is *abundant* indien de som van zijn delers **groter** is dan het getal zelf.
- Een getal is *deficient* indien de som van zijn delers **kleiner** is dan het getal zelf.

Een mogelijke input/output is:

```
Geef een bovengrens om te zoeken: 5000
6 is een perfect getal
28 is een perfect getal
496 is een perfect getal
In het bereik 2 tot 5000 :
Aantal perfecte getallen   : 3
Aantal abundante getallen  : 1239
Aantal deficiente getallen : 3757
```

Opdracht 4.32 (`data_gemid_std.py`)
Het bestand `data_waarden.txt` bevat een aantal decimale getallen. Bepaal, van deze waarden de gemiddelde waarde en de standaarddeviatie (een maat voor de spreiding van deze waarden). Voor de waarden x_i $(i = 1, ..., n)$ bereken je het gemiddelde x_{gem} en standaarddeviatie σ met

$$x_{gem} = \frac{\sum_{i=1}^{n} x_i}{n} \qquad \text{en} \qquad \sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_{gem} - x_i)^2}{n-1}}$$

Schrijf een script dat de waarden uit `data_waarden.txt` inleest (met een geschikte functie uit de module `infoFun`) de gemiddelde waarde en standaarddeviatie ervan berekent en deze op het scherm toont in de volgende vorm:

```
----------------
      Waarden
----------------
       12.584
       15.330
       16.397
       11.005
       14.379
       12.045
----------------
Gemid: 13.623
Stdev:  2.079
```

Tip: gebruik de functie `listReadValues` uit de module `infoFun` om de decimale waarden, die je kan terugvinden in het bestand `data_waarden.txt`, in een lijst op te slaan.

Vóór je begint:

- Lees Sectie 4.5.2 aandachtig. In het bijzonder het deel met betrekking tot het gebruik van de functie `listReadValues()` uit de module `infoFun`.

Opdracht 4.33 (`analyseer_codon_data.py`)

Een codon is een sequentie van drie nucleotiden. Er zijn vier verschillende nucleotiden (A, G, C en T) waardoor er dus theoretisch 64 verschillende codons mogelijk zijn (zie `codons64.txt`). In een experiment werd een lijst samengesteld bestaande uit duizenden codons die teruggevonden werden in een organisme (zie `codon_data.txt`). De meeste codons komen dus vele malen voor. Bereken de frequentieverdeling van de codons in `codon_data.txt`. Dit is een tabel (zie onder) die voor elk van de 64 mogelijke codens de frequentie van voorkomen uitdrukt (uitgedrukt in percentages). Per rij worden 8 codons en bijhorende percentages geprint.

De output is:

```
Percentages codons in bestand: codon_data.txt
AAA:1.13 AAG:0.89 AAC:1.49 AAT:0.71 AGA:2.01 AGG:0.36 AGC:1.45 AGT:1.39
ACA:2.96 ACG:0.52 ACC:1.15 ACT:2.96 ATA:0.66 ATG:2.70 ATC:0.93 ATT:1.89
GAA:1.01 GAG:2.38 GAC:0.36 GAT:2.22 GGA:0.62 GGG:2.92 GGC:2.60 GGT:0.60
GCA:1.91 GCG:0.66 GCC:2.44 GCT:0.44 GTA:2.56 GTG:1.27 GTC:1.01 GTT:2.60
CAA:0.26 CAG:0.32 CAC:2.14 CAT:0.91 CGA:2.58 CGG:2.18 CGC:1.51 CGT:2.84
CCA:1.83 CCG:0.77 CCC:0.48 CCT:1.27 CTA:2.64 CTG:0.99 CTC:1.59 CTT:1.59
TAA:2.46 TAG:2.32 TAC:1.33 TAT:2.64 TGA:1.89 TGG:1.57 TGC:2.58 TGT:0.38
TCA:2.04 TCG:2.92 TCC:1.79 TCT:2.04 TTA:1.05 TTG:0.22 TTC:1.19 TTT:0.87
```

Tip: Je zal gebruik moeten maken van twee (geneste) `for`-lussen om te itereren over de codons in beide bestanden en deze met elkaar te vergelijken.

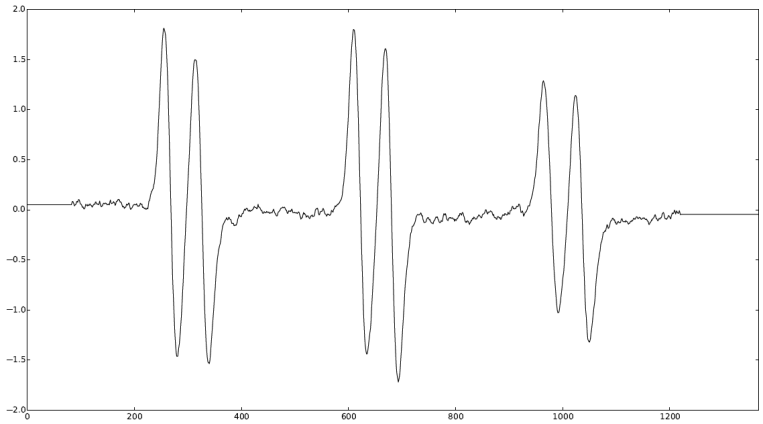
Opdracht 4.34 (`isbn02.py`)

Schrijf een programma dat alle opgelijste ISBN13-codes uit het bestand `isbn13_codes.txt` inleest, nagaat welke codes er foutief zijn, alle codes weergeeft op het scherm waarin de foutieve codes op dezelfde regel getagged worden met `FOUT!`. Verder moet dit aangevuld worden met het totaal aantal, het juiste aantal en het foute aantal ISBN-codes. Een mogelijke (scherm)output is:

```
9780321706600   FOUT!
9780123865137
9780596526788
9780471741564   FOUT!
9781292025933
9781482299281
9780131747189   FOUT!
9783642024740
Totaal aantal ISBN13-codes:   8
Aantal juiste ISBN13-codes:   5
Aantal foute ISBN13-codes:    3
```

Opdracht 4.35 (`spectrum_proc.py`)

Radicalen spelen een belangrijke rol in het verouderingsproces van bier. Deze radicalen kunnen *getrapped* worden met een zgn. *spin trap* en vervolgens gedetecteerd worden met Elektronen Paramagnetische Resonantie (EPR). Een veel gebruikte spin trap is α-(4-Pyridyl N-oxide)-N-tert-butylnitrone (POBN) dat aanleiding geeft tot een 6 “lijnen” spectrum.



Figuur 4.2: POBN-adduct EPR-spectrum

Zo’n EPR-spectrum heeft dus zes lokale maxima en minima. Schrijf een programma dat de volgende instructies uitvoeren.

1. Het bestand `eprspectrum02.txt` inleest met de functie `listReadValues()` uit de infoFun module. Deze functie is analoog aan de functie `listRead()`, maar zal in plaats van een lijst van `str` objecten een lijst van `float` objecten retourneren.
2. Het (absolute) maximum en minimum van het spectrum bepaalt met een zelf geïmplementeerd algoritme.
3. Het (absolute) maximum en minimum van het spectrum bepaalt met de *built-in* functies `max()` en `min()`. Je zou dezelfde waarden moeten bekomen als hierboven.
4. Uit Figuur 4.2 blijkt dat dit spectrum 6 lokale maxima bevat die groter zijn dan 0.5. Schrijf een script dat de hoogte van deze maxima bepaalt[7]. In een rij is een getal een lokaal maximum als geldt dat dit getal groter is dan het voorgaande getal **én** groter is dan het volgende getal. In de rij `[1, 2, 5, 3, 1, 7, 2]` zijn 5 en 7 lokale maxima omdat ze groter zijn dan hun buren.

Een mogelijke output is:

```
Max en min waarde (eigen implementatie):
max:  1.813138
min:  -1.7182275

Max en min waarde (met min(), max() functies):
max:  1.813138
min:  -1.7182275

Lokale maxima:
[01]  1.81313800
[02]  1.49919560
[03]  1.79931330
[04]  1.60865030
[05]  1.28854150
[06]  1.14191440
```

Opdracht 4.36 (`lening.py`)

Een annuïteitenlening is een lening waarbij de lener jaarlijks een vast geldbedrag, d.i. *annuïteit*, moet betalen aan de geldschieter. Zo'n lening omvat een jaarlijkse *aflossing* (d.i. het jaarlijkse geldbedrag waarmee de schuld kleiner wordt) en *rente* (d.i. het jaarlijkse geldbedrag dat je aan de geldschieter verschuldigd bent). De *annuïteit* wordt zo gekozen dat aan het einde van de looptijd de volledige lening is terugbetaald.
Noem B_0 het geleende bedrag, dan wordt de annuïteit A berekend met:

$$A = \frac{p}{1-(1+p)^{-n}}\, B_0\,,$$

[7] Opmerking: dit is niet zo eenvoudig maar wel mogelijk op basis van wat we tot nu toe gezien hebben. Later zullen wezien dat dit eenvoudiger kan door ge bruik te maken van indexering

waarin p de (vaste) rentevoet is en n het aantal jaren. De **resterende schuld** B_j na j jaren kan je berekenen uit

$$B_j = (1+p)^j B_0 + \frac{A}{p}(1-(1+p)^j)\,.$$

De *aflossing* a_j kan je berekenen met

$$a_j = (1+p)^{j-1}(A - p\,B_0)$$

en de *rente* r_j kan je halen uit:

$$r_j = A - a_j\,.$$

Zowel de aflossing a_j als de rente r_j kan je pas berekenen vanaf het eerste jaar, dus voor $j > 0$. Schrijf een script dat:

- Vraagt aan de gebruiker wat het geleende bedrag B_0 is (cf. dit is een bedrag in Euro).
- Vraagt aan de gebruiker wat de (jaarlijkse) rentevoet p is (cf. dit is een getal tussen 0 en 1).
- Vraagt aan de gebruiker wat de looptijd n in jaren van de lening is.
- De annuïteit A berekent en op het scherm toont.
- De jaarlijkse aflossingstabel op het scherm weergeeft met de volgende structuur:
  - 1^{ste} kolom: jaar
  - 2^{de} kolom: rente
  - 3^{de} kolom: aflossing
  - 4^{de} kolom: resterende schuld

  Rond alle geldbedragen af tot op de euro.

Je hoeft in deze opdracht de getalwaarden in de tabel niet speciaal uit te lijnen maar als extra oefening mag je dit uiteraard wel doen.

Een mogelijke input/output is:

```
Geef het geleende bedrag: 100000
Geef de rentevoet: 0.04
Geef de looptijd: 10
De annuiteit is: 12329 euro
Aflossingstabel (bedragen in euro):
 0 ---- ----- 100000
 1 4000  8329  91671
 2 3667  8662  83009
 3 3320  9009  74000
 4 2960  9369  64631
 5 2585  9744  54887
 6 2195 10134  44753
 7 1790 10539  34214
 8 1369 10961  23254
 9  930 11399  11855
10  474 11855      0
```

Vóór je begint

- Lees Sectie 4.3.1 aandachtig.

4.10 Belangrijkste concepten – samenvatting

- Controlestructuren - selectie: `if-elif-...-else`

```
if boolean_expressie:
    # if-suite
```

```
if boolean_expressie:
    # if-suite
else:
    # else-suite
```

```
if boolean_expressie_1:
    # suite 1
elif boolean_expressie_2:
    # suite 2
elif boolean_expressie_3:
    # suite 3
...
else:
    # else-suite
```

- Controlestructuren - herhaling: `while`

```
while boolean_expressie:
    # while-suite
```

```
while boolean_expressie_1:
    # suite 1
    if boolean_expressie_2:
        # suite 2
    elif boolean_expressie_3:
        # suite 3
    ...
    else:
        # else-suite
```

```
while boolean_expressie:
    # while-suite
else:
    # else-suite
```

```
while boolean_expressie_1:
    # while-suite
    if boolean_expressie_2:
        # if-suite
        break  # verlaat while-lus en negeer else-statement
else:
    # else-suite
```

- Controlestructuren - iteratie: `for`

```
for een_element in een_collectie:
    # for-suite
```

```
for element_1 in collectie_1:
    # suite 1
    for element_2 in collectie_2:
        # suite 2
        ...
```

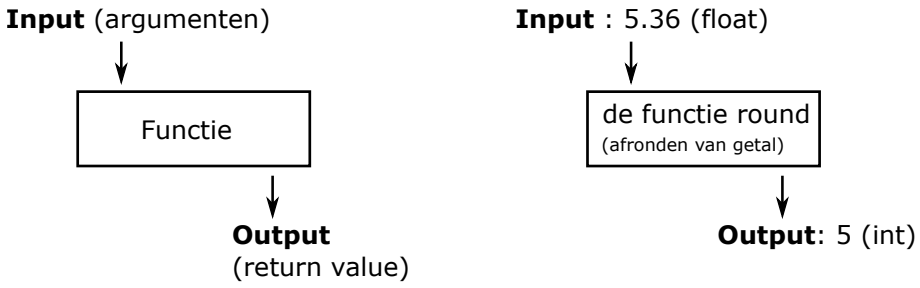
```
for een_element in een_collectie:
    # for-suite
    if boolean_expressie_1:
        # suite 1
    elif boolean_expressie_2:
        # suite 2
    ...
    else:
        # else-suite
```

```
for een_element in een_collectie:
    # for-suite
    if boolean_expressie_1:
        # if-suite
        break  # verlaat for-lus en negeer else-statement
else:
    # else-suite
```

5

Functies

In Hoofdstuk 2, Sectie 3.5 werden functies geïntroduceerd als structuren die input aanvaarden, op basis van deze input berekeningen uitvoeren en het resultaat retourneren als output. Figuur 5.1 illustreert deze denkwijze.



Figuur 5.1: (links) Schematische voorstelling van een functie als een structuur die input vertaalt naar output. (rechts) Toepassing van dit schema voor de functie `round()` die een decimaal (type `float`) getal afrondt tot een geheel getal (type `int`).

Bij het gebruik van functies onderscheidden we reeds de **naam**, de **parameters** en **argumenten**, alsook de **return value**. In de help van de functie `round()` vinden we terug:

```
round(number[, ndigits]) -> number
```

De bovenstaande begrippen worden, ter opfrissing, hierop toegepast:

- `number` en `ndigits` zijn de **parameters** van de functie. Dit zijn de namen die gegeven worden aan de inputs.
- Bij de functie-oproep[1] (*function call*) `a = round(number = 5.262, ndigits = 2)` noemen we de waarden 5.262 en 2 de **argumenten**. De variabele `a` zal verwijzen naar de **return value**.
- De parameter `ndigits` is **optioneeel** en heeft als **default waarde** `None`. Indien bij de function call geen tweede argument wordt meegegeven, zal deze default gebruikt worden (afronden tot op gehele waarde).

[1] We zullen in deze curusus de benamingen function call, functie-oproep of kortweg oproep, door elkaar gebruken.

In de voorgaande hoofdstukken hebben we steeds functies opgeroepen die *built-in* waren, of deel uitmaakten van een module (bv. de module `math`). In dit hoofdstuk wordt beschreven hoe we **zelf *eigen* functies** kunnen aanmaken en gebruiken.

5.1 User-defined functions

Wanneer een programmeur *eigen* functies definieert, en deze vervolgens oproept, dan noemen we deze functies *user-defined*, in tegenstelling tot *built-in* functies of functies die deel uitmaken van (bestaande) modules.

5.1.1 Nut van user-defined functions

Vaak zal men **eenzelfde sequentie van instructies op meerdere plaatsen** in de code nodig hebben. Bijvoorbeeld in een toepassing waarin DNA-sequenties moeten geanalyseerd worden, moet men vrij vaak de `GC`-inhoud berekenen (het percentage van de nucleotiden in een DNA-sequentie dat `G` of `C` is). Indien deze berekening meerdere malen moet gebeuren, dan zouden we kunnen opteren om de statements die nodig zijn om deze aantallen te berekenen meerdere malen te dupliceren (of te copy-pasten).

Het gebruik van **functies** vormt een **alternatief** voor de duplicatie van deze code, en brengt bovendien meer structuur aan in de code. Daarnaast laten functies de gebruiker toe om abstractie te maken van bepaalde taken. Indien bv. een functie-oproep `bepaalGC("AACCTTGGA")` toelaat om de `GC`-inhoud te berekenen, dan hoeft de programmeur zich niet te bekommeren over de manier waarop deze functie precies werd geïmplementeerd. Hij hoeft enkel te weten hoe hij deze functie moet/kan gebruiken.

5.1.2 Een voorbeeld

Fragment 5.1 bevat een *user-defined function* alsook de functie-oproep ervan.

Fragment 5.1: Voorbeeld functiedefinitie met oproep

```
1  #%%  Functiedefinitie
2  def maximum(a, b):
3      if a > b:
4          c = a
5      else:
6          c = b
7      return c
8
9  #%%  Instructies
10 m = maximum(10, 12)      # functie-oproep
11 print("Oproep 1:", m)
12
```

```
13 x = 7; y = 5
14 z = maximum(x, y)
15 print("Oproep 2:", z)
```

De output van dit codefragment is:

```
Oproep 1: 12
Oproep 2: 7
```

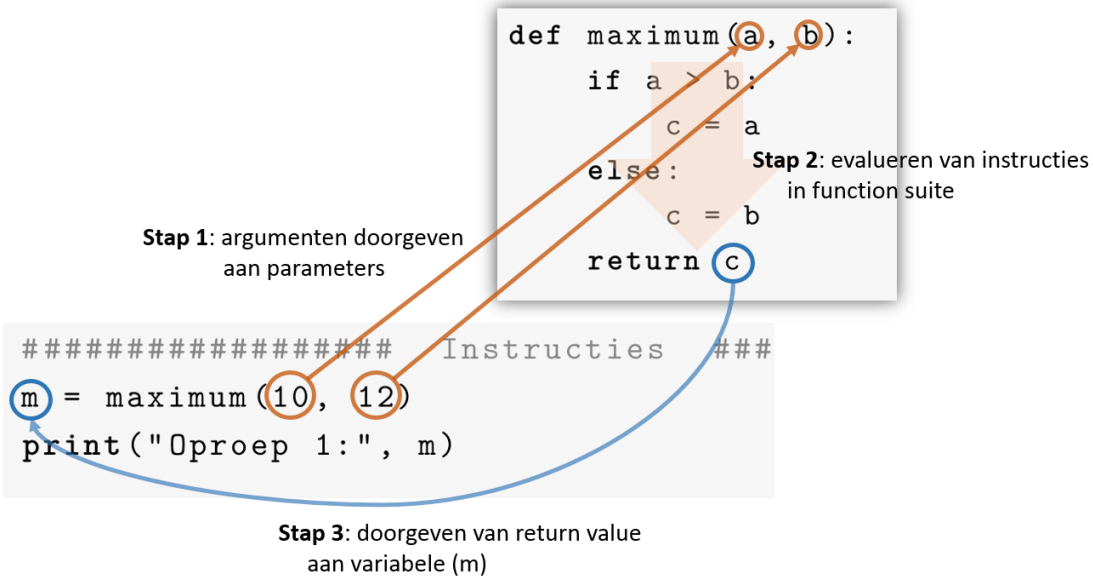
Laten we even de tijd nemen om het bovenstaande codefragment te bestuderen en zelf te achterhalen hoe de output tot stand komt. In de volgende secties wordt dit in detail besproken.

Merk op dat de hashtags (#) **commentaartekens** zijn en dus geen invloed hebben op de uitvoer. Ze worden hier gebruikt om visueel duidelijk te maken dat dit codefragment principieel uit **twee** delen bestaat:

1. een **functiedefinitie**, en
2. meerdere **functie-oproepen** en/of andere instructies.

Merk op dat de **oproep** van user-defined functions **volledig analoog** is **aan de oproep van built-in functies**.

Figuur 5.2 geeft schematisch weer hoe argumenten uit de functie-oproep worden doorgegeven aan de parameters in de functiedefinitie en de volgorde waarin de instructies worden uitgevoerd (*control flow*). In de volgende secties gaan we hier verder op in, maar deze figuur licht het algemene principe toe.



Figuur 5.2: Schematische voorstelling van de *control flow*.

5.1.3 De functiedefinitie

In Fragment 5.1 bevatten regels 2–7 een **functiedefinitie**. Vóór de syntax van deze functiedefinitie in detail besproken wordt, bekijken we eerst wat men in een programmeercontext bedoelt met een *functie*.

> *A **function** is a named block of (reusable) code that performs a specific task*[a].
>
> [a]Merk op dat er zeer veel definities bestaan voor een *functie*. Bovendien hangt de definitie (deels) af van de programmeertaal, maar deze definitie is vrij algemeen toepasbaar.

Uit deze beschrijving volgt dat een functie, in essentie, een stuk broncode is met een naam. Dit is tevens wat de functiedefinitie op regels 2–7 in Codefragment 5.1 bevatten. De naam van deze functie is *maximum* en het bijhorende blok code (de function-suite of de body) is een `if-else`-statement. Meer algemeen volgt een functiedefinitie de volgende **structuur**:

```
def functie_naam(parameter_1, parameter_2, ...):
    #
    # De function suite of body
    #
    return ret_variabele
```

We bespreken nu in detail de onderdelen van deze functiedefinitie.

5.1.3.1 De functiedefinitiehoofding (*function header*)

De functiedefinitiehoofding of *function header*[2] begint steeds met het Python keyword `def`, gevolgd door de naam van de functie en de parameters. Een (vrij) algemene vorm van deze hoofding is de volgende.

> De **functiedefinitiehoofding** of *function header*.
>
> `def functienaam(parameter_1, parameter_2, ...) :`

De **parameters** zijn namen die de programmeur kan kiezen, maar moeten voldoen aan de regels voor variabelenamen (zie Sectie 3.1.3). De parameters zijn de namen die worden toegekend aan de argumenten wanneer de functie wordt opgeroepen.

Beschouwen we de volgende hoofding:

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

[2]De function header is een courante term om de hoofding (typisch eerste regel) van een functiedefinitie mee te benoemen in Python, maar deze term is afkomstig uit (gecompileerde) talen waarbij de gegevenstypes van de argumenten en de *return value* van een functie vastgelegd worden door de programmeur. Bij Python is dit **niet** het geval.

dan zal bij de functie-oproep:

```
>>> m = maximum(10, 12)
```

de parameter `a` verwijzen naar een `int` met waarde 10, en zal `b` verwijzen naar een `int` met waarde 12. **Binnen de body** van de functie wordt dus naar de argumenten 10 en 12 verwezen via de parameters `a` en `b`.

Belangrijk om hierbij op te merken is dat:

- **alle communicatie**[3] tussen het globale werkgeheugen en de objecten die daarin aanwezig zijn en de function body gebeurt **via de parameters**.
- de **parameters enkel betekenis** hebben **binnen de function-body**. Indien men in Fragment 5.1 onder regel 15 het `print`-statement `print(a)` zou toevoegen, dan zou bij het uitvoeren van deze regel de volgende foutmelding[4] optreden:

```
print(a)
Traceback (most recent call last):
 ...
NameError: name 'a' is not defined
```

5.1.3.2 De function-suite of body

De function-suite of body (in Fragment 5.1 regels 3–7) bevat een aantal statements en/of expressies die de waarden waarnaar de parameters verwijzen, vertalen in een *return value*. Bij het oproepen van de functie zullen, nadat de argumenten werden gelinkt aan de parameters, de statements en expressies in de function suite sequentieel worden uitgevoerd. Als onderdeel van deze instructies zal (vaak) een nieuwe variabele gecreëerd worden die verwijst naar de *return value*. In het voorbeeld is dit de variabele `c` (zie verder).

De regels waaraan de function suite of body moet voldoen zijn dezelfde als die waaraan ook andere suite's, zoals `while` of `for` moeten voldoen:

- **Indentatie** bepaalt waar de function suite start en eindigt.
- De *function suite* kan statements en expressies bevatten, maar ook samengestelde (*compound*) statements zoals `if-else`, `for-` of `while`-statements.
- De parameters kunnen in de body gebruikt worden op dezelfde manier als variabelen.
- Binnen de function-suite kunnen **bijkomende variabelen** gecreëerd worden. Zoals bijvoorbeeld de variabele `c` in Fragment 5.1. **Let op**: deze variabelen zijn **enkel beschikbaar binnen de function-suite**, **niet daarbuiten**.

[3] Om correct te zijn *bijna alle* communicatie, zie verder.

[4] Behalve als er buiten de functie `maximum()` een variabele `a` zou gedefinieerd worden. In dat geval zal de waarde van die variabele geprint worden (zie Sectie 5.1.4.5).

Functies bevatten heel vaak een **return** statement. Dit statement ziet er als volgt uit:

```
return variabeleNaam
```

Dit statement bepaalt welke waarde de *return value* van de functie wordt. Wanneer bij het uitvoeren van een functie een **return** statement wordt geëvalueerd, dan zal de functie uitvoer onmiddellijk eindigen en wordt het object waarnaar `variabeleNaam` verwijst, geretourneerd[a].

[a] Sommige functiedefinities bevatten geen of meerdere **return** statements. We bespreken dergelijke functies later.

5.1.3.3 Het evalueren van een functiedefinitie

Een functiedefinitie is een statement dat door een interpreter wordt geëvalueerd. Pas nadat een functiedefinitie werd geëvalueerd kan ze in daarna volgende statements en expressies worden opgeroepen o.b.v. haar naam. Dit proces is heel gelijkaardig aan het creëren en gebruiken van variabelen, waarbij variabelen worden gecreëerd d.m.v. een *assignment statement* (bv. `s = 5`) en nadien worden gebruikt in andere statements en expressies (bv. **print**`(s + 2)`). Het functiedefinitie-statement kan dus gezien worden als een bijzonder assignment statement.

Het verschil tussen functiedefinities en functie-oproepen kan men ook terugvinden in de wiskunde. Daar wordt een functie gedefinieerd o.b.v. van een functievoorschrift. Het evalueren van de functie gebeurt ook hier o.b.v. de naam en argument dat tussen haakjes wordt geplaatst. Deze gelijkenis wordt hieronder aangehaald:

Functievoorschrift van f:

$$f(x) = x^2 + 3x + 1$$

Evalueren van f met argument 5.

$$f(5)$$

```
# functiedefintie
def fun(x):
    y = x**2 + 3*x + 1
    return y

# functie-oproep
fun(5)
```

Wanneer de functiedefinitie wordt geëvalueerd, wordt (net zoals bij een assignment-statement) een object gecreëerd in het werkgeheugen waarin de functie wordt ondergebracht. Het gegevenstype van dit object is `function`. Dit wordt hieronder geïllustreerd:

```
>>> def fun(x):              # uitvoeren van functie-definitie
 ...    y = x**2 + 3*x + 1
 ...    return y

>>> type(fun)                # opvragen van type
function
```

```
>>> fun(5)                    # uitvoeren van functie-oproep
41
```

5.1.3.4 De functie-oproep

Een functie die in het werkgeheugen aanwezig is, kan men vervolgens oproepen (*function call*) o.b.v. zijn naam en de nodige argumenten. Nadat de oproep voltooid is, kan de return-value gebruikt worden in instructies die daarna volgen.

Functie**definitie**:

```
def fun(a):
    b = 10
    c = a * b
    return c
```

Functie-**oproep**:

```
>>> z = fun(5)
>>> print(z)
50
```

5.1.4 Namespaces

In de voorgaande hoofdstukken en secties werd reeds beschreven dat alle structuren in Python objecten zijn. Bij het uitvoeren van het assignment `a = 2` wordt een integer object met waarde 2 gecreëerd in het werkgeheugen en is `a` de variabele waarmee we dit object associëren. Deze variabele (of naam) kan dan verder gebruikt worden in andere statements en expressies. Ook functies (zowel *built-in* als *user-defined*) zijn objecten in die we kunnen aanspreken via hun naam.

Tijdens het uitvoeren van een programma zal de interpreter een collectie aanleggen[5] van alle gedefinieerde *namen* en de objecten waarnaar ze verwijzen: zowel namen van traditionele variabelen als die van functies. Deze collectie heet een ***namespace***. Afhankelijk van de plaats in de broncode waar namen worden gecreëerd, komen ze terecht in

1. de *built-in namespace*,
2. de *globale namespace*,
3. de *enclosing namespace* of
4. in een *lokale (function) namespace*.

We bekijken deze vier namespaces hierna meer in detail.

[5]Strikt genomen is dit eerder een *dictionary*, maar dit is niet van belang hier.

5.1.4.1 De built-in namespace

Deze namespace bevat de namen van alle built-in functies en variabelen, die automatisch worden ingeladen bij het opstarten van de interpreter. Voorbeelden van namen die tot deze namespace behoren zijn `round()`, `len()`, …. Namen die tot deze namespace behoren zijn overal toegankelijk.

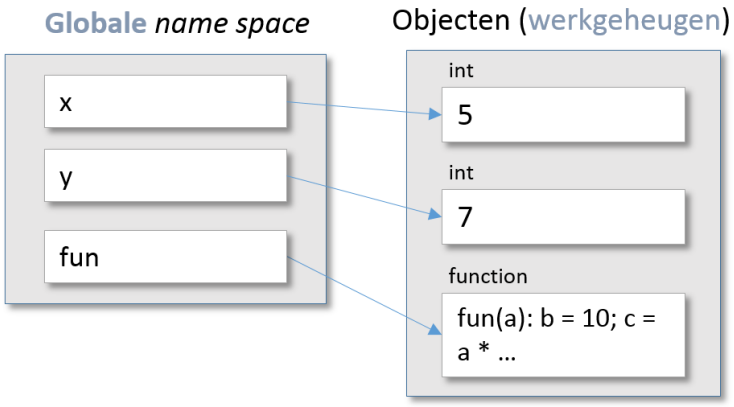
5.1.4.2 De globale namespace

Namen van variabelen (of namen van functies) die worden gecreëerd (in de console of) in het Python script dat men uitvoert (maar **niet** in een functie) komen terecht in de globale namespace. Beschouw de volgende instructies, die uitgevoerd worden in de console of in een script:

```
>>> x = 5
>>> y = 7
>>> def fun(a):        # een functiedefinitie creeert
  ...   b = 10         # ook een naam (in dit geval fun)
  ...   c = a * b
  ...   return c
```

Na het uitvoeren van deze instructies komen de variabelen `x`, `y` en de functie `fun()` terecht in de globale namespace.

Het **toestandsdiagram** is een overzicht van de globale (en/of lokale) namespace samen met de objecten waar de namen in de namespace naar verwijzen. Het toestandsdiagram, na het uitvoeren van bovenstaande instructies, wordt getoond in Figuur 5.3.



Figuur 5.3: Toestandsdiagram na uitvoeren instructies `x = 5`, `y = 7` en `def fun(a): ...`.

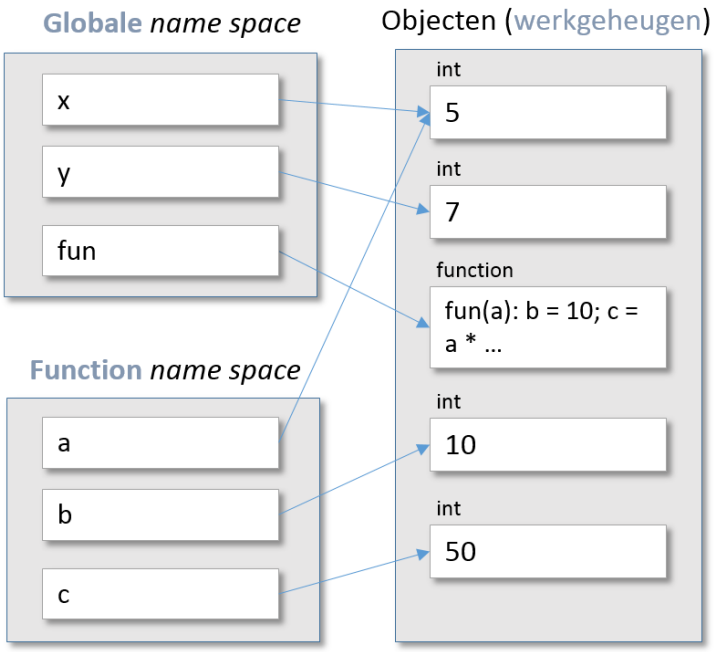
5.1.4.3 De lokale function namespace

Wanneer een functie-oproep wordt geëvalueerd dan zal, tijdens het uitvoeren van deze oproep, een **tijdelijke** lokale function namespace worden gecreëerd. Tijdens de functie evaluatie zal de interpreter gebruik kunnen maken van de namen die tot deze lokale function namespace behoren. **Nadat de evaluatie van de functie is afgelopen, zal deze function namespace weer**

verdwijnen. De namen die deel uitmaken van de function namespace, worden dan ook automatisch verwijderd. Het volgende codefragment sluit aan bij het voorgaande fragment.

```
>>> z = fun(x)       # oproep van functie fun() (definitie hiervoor)
```

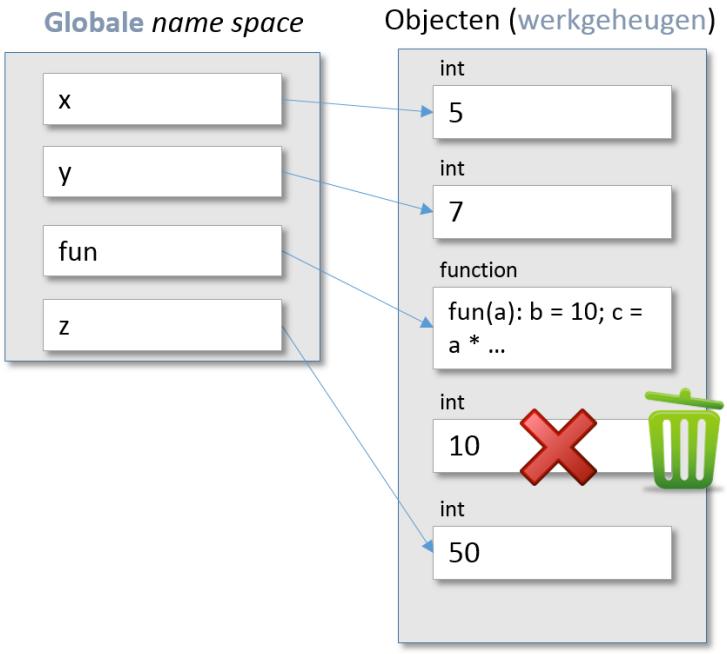
Figuur 5.4 toont het bijhorende toestandsdiagram net voordat het `return`-statement van de functie `fun` wordt uitgevoerd. Figuur 5.5 toont het toestandsdiagram nadat de evaluatie van de functie-oproep is afgerond.



Figuur 5.4: Toestandsdiagram tijdens het uitvoeren van `fun(x)`, net voor het `return`-statement `return c`

Uit Figuur 5.4 blijkt dat de parameter `a` zich gedraagt als een variabele die verwijst naar het integer object met waarde `5` (dit is hetzelfde object waarnaar ook `x` verwijst). **De argumenten geven dus hun referentie (of verwijzing) door aan de parameters.** De variabelen die gecreëerd worden door de toekenningsstatements in de body van de functie komen terecht in de lokale function namespace.

Uit Figuur 5.5 blijkt dat, na het beëindigen van de functie-evaluatie, de lokale function namespace verdwijnt. Merk op dat de variabele `z` nu deel uitmaakt van de globale namespace en dat deze variabele verwijst naar de return value `50`, die het resultaat is van de functie-oproep.



Figuur 5.5: Toestandsdiagram na het uitvoeren van `z = fun(x)`.

5.1.4.4 De enclosing namespace

Beschouw de **geneste** functie-definities in Fragment 5.2:

Fragment 5.2: Geneste functies

```
1 def funF(x):
2     a = 1
3     def funG(y)
4         b = 2
5         resG = (b + y)*a
6         return resG
7     resF = a + g(x)
8     return resF
```

Bij de evaluatie van een oproep naar `funF()` zal een tijdelijke lokale namespace worden gecreëerd. Vermits de functie `funF()` een oproep doet naar de functie `funG()` zal er binnen de lokale namespace van `funF()` een nieuwe lokale namespace voor `funG()` worden gecreëerd. De namespace van `funF()` wordt de ***enclosing namespace*** genoemd, de namespace van `funG()` wordt de ***enclosed namespace*** genoemd.

Opmerking: in het vervolg van deze cursus zullen we **geen geneste functies gebruiken of zelf schrijven**.

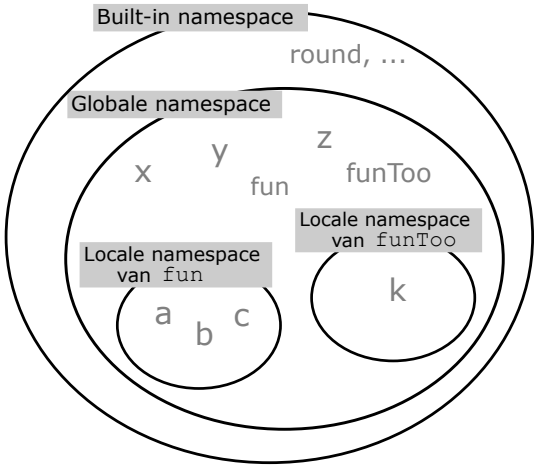
5.1.4.5 De scope

Een variable die wordt gecreëerd binnen een functie, is enkel beschikbaar binnen die functie. Men zegt daarom dat de **scope** van deze variabele beperkt is tot de body van de functie. De scope van

een namespace is het deel van het programma van waaruit de namespace rechtstreeks toegankelijk is.

In de voorgaande secties werden binnen een functie enkel variabelen gebruikt die lokaal (dus binnen de functie) gecreëerd werden. Dit is echter niet noodzakelijk. **Ook variabelen die tot de built-in of de globale namespace behoren zijn beschikbaar binnen een functie.** Deze namespaces zijn genest. Om dit te benadrukken worden namespaces vaak voorgesteld als een Venn-diagram. Dit wordt hieronder geïllustreerd:

```
1  # functiedefintie
2  def fun(a):
3      b = 10
4      c = a * b
5      return c
6
7  def funToo(k):
8      return k**x
9  # functie-oproep
10 x = 5
11 y = fun(x)
12 z = funToo(3)
```



Hiërarchie van de namespaces. Vb. variabele x is toegankelijk vanuit `fun`.

Merk op: function namespaces worden onmiddellijk verwijderd na beëindigen functie-oproep.

Omdat de variabele `x` behoort tot de globale namespace is ze ook toegankelijk binnen de functie `funToo()`. Bij het evalueren van de expressie `k**x` zal de waarde `5` dus gebruikt kunnen worden.

Een aantal belangrijke opmerkingen

1. Alle built-in functions (zoals bv. `round()` en `max()`) behoren automatisch tot de built-in namespace. Dit wil zeggen dat deze functies ook kunnen worden opgeroepen vanuit scripts én vanuit *user-defined* functies.
2. Beperk het gebruik van het flexibele scoping mechanisme voor variabelen. Functies zoals `funToo()` zijn vaak geen *good practices*. Bij het programmeren tracht men, in de mate van het mogelijke, communicatie tussen de lokale en globale namespace te laten verlopen via de parameters/argumenten.
3. Eenzelfde naam kan tegelijk voorkomen in een meerdere namespaces. De regel hierbij is dat de naam in de lokale namespace voorrang krijgt op de globale namespace, en dat als laatste pas gekeken wordt naar de built-in namespace. Zie verder voor een voorbeeld.
4. Wanneer meerdere **functies** gedefinieerd worden **in eenzelfde script**, komen hun namen in de globale namespace terecht. Dit wil zeggen dat deze functies **voor elkaar bereikbaar zijn** en dat men in de functiedefinitie van de ene functie de andere functie kan oproepen.

5.1.4.6 Invloed van name-space: voorbeeld

Indien een variabele zowel voorkomt in de lokale als in de globale namespace of de built-in namespace, dan heeft de lokale namespace voorrang. De volgorde waarin name-spaces worden geraadpleegd wordt beschreven door de **LEGB-regel** genoemd: eerst **L**okaal, dan **E**nclosing, vervolgens **G**lobaal en tenslotte **B**uilt-in.

We illustreren dit a.d.h.v. een concreet voorbeeld. De warmte Q toegevoegd aan n mol van een ideaal gas bij een constant volume (V = constant) door een verandering van de temperatuur T wordt berekend met

$$Q = nC_V\Delta T$$

waarbij $C_V = 3R/2$ indien het gas één-atomig is en R de gasconstante is ($R \approx 8.31\,J/(mol\,K)$). In Fragment 5.3 implementeren we deze formule in een functie `warmte_aan_gas()` en berekenen meteen de warmte Q voor een bepaalde waarde van n en ΔT:

Fragment 5.3: `warmte.py`

```
1 def warmte_aan_gas(n, dT):
2     R = 8.3144621
3     Cv = 3/2*R
4     warmte = n*Cv*dT
5     return warmte
6 n  = 2
7 dT = 10
8 print("Q = ", warmte_aan_gas(n, dT))
```

Dit geeft als output:

```
Q =  249.433863
```

Merk op dat de variabelen `R = 8.3144621` en `Cv = 3 / 2 * R` zich **binnen de functiedefinitie** bevinden. Dit zijn dus **lokale** variabelen.

We maken nu gebruik van de verworven kennis om na te gaan (en te verklaren) of het berekende resultaat wijzigt of niet wijzigt, en of we een foutmelding krijgen, in de volgende twee gevallen.

1. `R = 8.3144621` wordt **buiten** de functiedefinitie gedefinieerd en komt terecht in de **globale namespace**:

```
#%% Functiedefinities
def warmte_aan_gas(n, dT):
    Cv = 3 / 2 * R
    Q = n * Cv * dT
    return Q
#%% Instructies
R = 8.3144621
```

```
n  = 2
dT = 10
print("Q = ", warmte_aan_gas(n, dT))
```

Heeft als output:

```
Q =  249.433863   # zelfde resultaat als voorgaande
```

2. `Cv = 5 / 2 * R` wordt (bijkomend) **ook buiten** de functiedefinitie gecreëerd. De variabele `Cv` is dus **zowel binnen de lokale als de globale namespace** aanwezig, maar met een **andere waarde**.

```
#%% Functiedefinities
def warmte_aan_gas(n, dT):
    Cv = 3 / 2 * R
    Q = n * Cv * dT
    return Q
#%% Instructies
R = 8.3144621
Cv = 5 / 2 * R
n  = 2
dT = 10
print("Q = ", warmte_aan_gas(n, dT))
```

```
Q =  249.433863   # zelfde resultaat als voorgaande
                  #  ==> lokale variabele Cv gebruikt
```

5.1.4.7 Voortgezette hiërarchie van namespaces

In de voorgaande voorbeelden werd de hiërarchie van de namespaces beperkt tot de built-in namespace, één globale (module) namespace en één lokale (function) namespace. De hiërarchie is echter meer uitgebreid. Zo zullen ook lokale function namespaces verder genest worden (*enclosing namespaces*), wanneer de suite van een functie een nieuwe functiedefinitie bevat cf. Fragment 5.2). Daarnaast heeft ook elke module zijn eigen globale *module namespace*. Inzichten in dit topic zijn vooral belangrijk bij het ontwikkelen van nieuwe modules en packages. Deze zaken vallen echter buiten het bestek van deze cursus.

5.2 Voorbeeld nuttig gebruik van functies: GC-inhoud

Het gebruik van functies biedt meer voordelen naarmate code langer wordt. Fragment 5.4[6] is wat meer uitgebreid. Bij uitvoeren zal het volgende gebeuren:

[6] Merk op dat dit fragment sterk kan ingekort worden, en dat ook efficiëntie kan worden verhoogd. Daar komen we later in dit hoofdstuk op terug.

1. De gebruiker wordt gevraagd een DNA sequentie in te voeren.
2. Er wordt nagegaan of deze DNA sequentie geldig is (en dus enkel bestaat uit `A`, `C`, `T` en `G`).
3. Voor geldige sequenties wordt de `GC` inhoud berekend ((aantal `G` + aantal `C`) / lengte × 100) en getoond op het scherm.
4. De voorgaande stappen worden herhaald tot de gebruiker `stop` invoert.

Fragment 5.4: `GC`-inhoud

```
#%% Functiedefinities
def isGeldig(DNA):
    controle = True
    for nucl in DNA:
        if not (nucl == "A" or nucl == "C" or nucl == "T" or nucl == "G"):
            controle = False
    return controle

def bepaalGC(DNA):
    GC = 0
    for nucl in DNA:
        if nucl == "G" or nucl == "C":
            GC += 1
    return GC/len(DNA)*100
#%% Instructies
DNA = input("Geef een DNA-seq in: ")
while DNA != "stop":
    if isGeldig(DNA):
        GC_inhoud = bepaalGC(DNA)
        print('GC-inhoud is:', GC_inhoud)
    else:
        print("Geen geldige sequentie!")

    DNA = input("Geef een DNA-seq in: ")

else:
    print("Programma beeindigd!")
```

Mogelijke input/output is:

```
Geef een DNA-seq in: AACTGGACCT
GC-inhoud is: 50.0

Geef een DNA-seq in: ACCAATT
GC-inhoud is: 28.57142857142857

Geef een DNA-seq in: AXXACT
Geen geldige sequentie!
```

```
Geef een DNA-seq in: stop
Programma beeindigd!
```

Opdracht 5.1 (verticale_worp.py)

Een voorwerp met massa m wordt vanaf de grond verticaal omhoog geworpen met een beginsnelheid v_0. De kinetische K en potentiële U energie wanneer het voorwerp hoogte y bereikt kunnen in dit geval berekend worden met de volgende formules:

$$K = \frac{1}{2}m\left(v_0^2 - 2gy\right), \qquad U = mgy,$$

waarbij $g = 9.81\ ms^{-2}$. Vul onderstaande code verder aan zodat na uitvoeren het daaronder getoonde resultaat op het scherm wordt getoond.

```
#%% Functiedefinities
def energie_kin(m, v0, y):
    ..............................
    ..............................
    return energie

def energie_pot(m, y):
    ..............................
    ..............................
    return energie

#%% Instructies
massa = 5.0           # kg
beginsnelheid = 10.0  # m/s
hoogte = 2.5          # m

K = energie_kin(massa, beginsnelheid, hoogte)  # de functie-oproep
U = energie_pot(massa, hoogte)
print("Kinetische energie:", K)
print("Potentiele energie:", U)
```

De output is:

```
Kinetische energie: 127.37499999999999
Potentiele energie: 122.62500000000001
```

Vóór je begint:
Lees Sectie 5.1 aandachtig!

Opdracht 5.2 (`verticale_worp.py`)

Maak gebruik van je functies `energie_kin()` en `energie_pot()` die je in Opdracht 5.1 schreef om, voor een voorwerp met massa $m = 0.5\ kg$ en beginsnelheid $v_0 = 10.00 ms^{-1}$, voor elk van de hoogtes $y = 0, 1, 2, 3, 4$ en $5\,m$ de waarden voor kinetische (K) en potentiële (U) energie te berekenen en op het scherm te tonen in tabelvorm.

De output zou er als volgt moeten uitzien (let op de **uitlijning**):

```
 y (m)     K (J)     U (J)
--------------------------   # --> bekom je met de instructie print("-"*26)
  0.00    25.000     0.000
  1.00    20.095     4.905
  2.00    15.190     9.810
  3.00    10.285    14.715
  4.00     5.380    19.620
  5.00     0.475    24.525
```

Tip: gebruik een `for`-lus waarin je de functies `energie_kin()` en `energie_pot()` oproept.

5.3 Functies die andere functies oproepen

5.3.1 Inleiding

Elke functie waarvan de naam in de globale namespace aanwezig is, kan opgeroepen worden in de function-suite van een andere functie waarvan de naam in de globale namespace aanwezig is.

Een heel eenvoudig voorbeeld vinden we in Fragment 5.5 waar de stelling van Pythagoras[7] wordt toegepast.

Fragment 5.5: Voorbeeld: functie die andere functie oproept

```
1  #%% Functiedefinities
2  def kwadrateer(getal):
3      return getal**2
4
5  def pythagoras(a, b):
6      c_kwadraat = kwadrateer(a) + kwadrateer(b)
7      return c_kwadraat
8  #%% Instructies
9  c_kwadr = pythagoras(3, 4)
10 print("Lengte schuine zijde:", c_kwadr**(1/2))
```

[7] In een rechthoekige driehoek met rechthoekszijden a en b en schuine zijde c geldt dat $a^2 + b^2 = c^2$.

De output is:

```
Lengte schuine zijde: 5.0
```

Bij het oproepen van de functie `pythagoras()` zal de functie `kwadrateer()` twee keer worden opgeroepen. Dit is mogelijk omdat de definitie van de functie `kwadrateer()` in dit script voorkomt. Wanneer het script wordt uitgevoerd, komt deze functie terecht in de *globale namespace* en is ze dus toegankelijk voor andere functies (zie ook Sectie 5.1.4).

Oproepen van functies
Opdat een functie opgeroepen zou kunnen worden, moeten de **functie-oproepen nà de functiedefinities** komen.

Dit betekent dat indien in het voorbeeld hierboven de instructies 9–10 vóór de definitie van de functie `pythagoras()` (lijn 5) zouden staan, de instructie `c_kwadr = pythagoras(3, 4)` een **foutmelding** zal genereren **omdat** de functie `pythagoras()` op dat moment nog **niet in de globale namespace aanwezig** is.

5.3.2 Functionele decompositie

Oplossingsstrategieën voor (complexere) problemen zullen het probleem vaak **ontbinden** in een aantal meer **eenvoudige deelproblemen**, en de oplossing van deze deelproblemen samenstellen om zo de oplossing van het hoofdprobleem te bekomen. Wanneer de oplossingen van deze deelproblemen worden geïmplementeerd in twee afzonderlijke functies, dan spreekt men van **functionele decompositie**.

We illustreren het gebruik van functionele decompositie a.d.h.v. de berekening de oppervlakte van een cilinder o.b.v. de diameter van het grondvlak en de hoogte. Men kan deze oppervlakte bepalen als:

$$\text{Oppervlakte cilinder} = 2 \times \text{Oppervlakte grondvlak} + \underbrace{\text{Oppervlakte mantel}}_{= \text{Omtrek grondvlak} \times \text{hoogte}}$$

Om tot de oppervlakte van een cilinder te komen, moet de oppervlakte en de omtrek van een cirkel berekend worden. Dit zijn twee afzonderlijke taken, die kunnen geïmplementeerd worden in twee functies. Vervolgens kunnen deze functies worden opgeroepen door een derde functie die hun output samenstelt en retourneert. Dit wordt geïllustreerd in Fragment 5.6:

Fragment 5.6: Functionele decompositie: oppervlakte van een cilinder

```
1 import math
2 #%% Functiedefinities
3 def oppervlakte_cirkel(straal):
4     oppervlakte = math.pi * straal**2
5     return oppervlakte
6
```

```
7  def omtrek_cirkel(straal):
8      omtrek = 2 * math.pi * straal
9      return omtrek
10
11 def oppervlakte_cilinder(diameter, hoogte):
12     opp = 2 * oppervlakte_cirkel(diameter/2)\
13         + omtrek_cirkel(diameter/2) * hoogte
14     return opp
15 #%% Instructies
16 A = oppervlakte_cilinder(2, 5)
17 print("Oppervlakte is:", A)
```

De output is:

```
Oppervlakte is: 37.69911184307752
```

Opdracht 5.3 (`statistieken.py`)

Beschouw het bestand `data_waarden.txt`. Dit bestand bevat een aantal waarden die kunnen ingelezen worden als een lijst van floats met de functie `listReadValues()` uit de module `infoFun`. De standaarddeviatie van deze getallen kan berekend worden met de volgende formule:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2} \quad \text{met} \quad \bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i\,,$$

waarbij $x_1, x_2, \ldots, x_n$ de waarden voorstellen en $\bar{x}$ het gemiddelde van deze waarden.

Vul het bestand `statistieken.py` aan.

1. Implementeer een functie `gemiddelde()` die een lijst van floats als argument aanvaardt en het **gemiddelde** van de waarden in deze lijst retourneert.

   ```
   >>> gem = gemiddelde([7, 5, 3, 9])
   >>> gem
   6.0
   ```

2. Implementeer een functie `std()` die een lijst van floats als argument aanvaardt en de standaarddeviate `stddev` van deze waarden retourneert. Bij de berekening van de standaarddeviatie heb je het gemiddelde nodig, roep daarom de functie `gemiddelde()` op in de function-suite van de functie `std()`.

   ```
   >>> stddev = std([7, 5, 3, 9])
   >>> stddev
   2.581988897471611
   ```

Indien deze functies correct geïmplementeerd werden, dan bekom je, na het uitvoeren van

```
waarden = infoFun.listReadValues("data_waarden.txt")
gemid_waarde = gemiddelde(waarden)
stdev_waarde = std(waarden)
print("Gemiddelde:", gemid_waarde)
print("Standaarddeviatie:", stdev_waarde)
```

de volgende output:

```
Gemiddelde: 13.623
Standaarddeviatie: 2.079
```

5.4 Default waarden voor parameters

Reeds in Hoofdstuk 3 werd beklemtoond dat een functie-oproep kan gebeuren o.b.v. **positionele** argumenten of o.b.v. **keyword**-argumenten (of een combinatie van beide). Dit principe wordt hieronder geïllustreerd voor de user-defined functie `volume_balk()`.

```
def volume_balk(lengte, breedte, hoogte):
    volume = lengte * breedte * hoogte
    return volume
```

De functie `volume_balk()` heeft drie parameters: (1) `lengte`, (2) `breedte` en (3) `hoogte`. We roepen deze positioneel en keyword-based op:

- Oproep met **enkel positionele** argumenten:

  ```
  >>> volume_balk(2, 3, 5)
  30
  ```

- Oproep met **enkel keyword** argumenten (`parameter = waarde`):

  ```
  >>> volume_balk(hoogte = 5, lengte = 2, breedte = 3)
  30
  ```

  Opmerking: bij gebruik van keyword argumenten wordt de link tussen waarde en parameter niet gelegd o.b.v. positie, en kan de **positie** (volgorde) dus **willekeurig gekozen** worden.

- **Combinatie** van beide:

  ```
  >>> volume_balk(2, hoogte = 5, breedte = 3)
  30
  ```

Belangrijke opmerking
Indien een combinatie van positionele en keyword argumenten gebruikt wordt, komen de **positionele argumenten steeds vooraan**. Zoniet wordt een foutmelding weergegeven:

```
>>> volume_balk(hoogte = 5, 2, breedte = 3)
    volume_balk(hoogte = 5, 2, breedte = 3)
                            ^
SyntaxError: positional argument follows keyword argument
```

In elk van de bovenstaande gevallen, werd de functie opgeroepen met drie argumenten. Indien minder argumenten worden meegegeven, dan treedt een foutmelding op:

```
>>> volume_balk(2, 3)
TypeError: volume_balk() missing 1 required positional argument: 'hoogte'
```

Default waarden voor parameters
In de functiehoofding kan de programmeur default waarden toekennen aan de parameters via de toekenning

```
parameter = defaultwaarde
```

Deze waarde wordt enkel gebruikt indien bij de functieoproep **geen argument** voorzien wordt.

De onderstaande definitie bevat een default waarde voor de hoogte.

```
def volume_balk(lengte, breedte, hoogte = 10):
    volume = lengte * breedte * hoogte
    return volume
```

Een oproep van deze (aangepaste) functie kan nu met twee argumenten:

```
>>> volume_balk(2, 3)
60
```

Indien de functie-oproep toch drie argumenten bevat, dan wordt de default waarde genegeerd.

```
>>> volume_balk(2, 3, 4)
24
```

Opmerking: parameters waarvoor een default waarde voorzien wordt, kunnen nooit gevolgd worden door parameters zonder default.

Plaats parameters met default waarden dus steeds achteraan in de functiehoofding.

Opdracht 5.4 (`formatteer_mac.py`)

Een MAC-adres is een uniek identificatienummer dat aan een apparaat in een ethernetnetwerk is toegekend dat ervoor zorgt dat apparaten met elkaar kunnen communiceren. Het MAC-adres bestaat uit 12 karakters (cijfers: `0` t.e.m. `9`, en (hoofd- of kleine) letters: `a`, `b`, `c`, `d`, `e` en `f`) en wordt in de volgende vormen genoteerd, bv. `000c6ed211e6` of `00:0c:6e:d2:11:e6` of `00-0c-6e-d2-11-e6`.

Implementeer de functie `formatteer_mac()` die twee argumenten aanvaardt:

- `mac`: een MAC-adres (zonder tussenvoegsels)
- `tussenvoegsel`: één karakter, met default ":"

Deze functie retourneert een geformatteerd MAC-adres, dit wil zeggen het originele MAC-adres, mits tussenvoegen van het `tussenvoegsel` op de correcte plaats:

```
>>> mac1 = formatteer_mac("000C6Ed211e6")
>>> print(mac1)
'00:0c:6e:d2:11:e6'

>>> mac2 = formatteer_mac("000C6Ed211e6", tussenvoegsel = "-")
>>> print(mac2)
'00-0c-6e-d2-11-e6'
```

5.5 Bijzondere return-statements

In de voorgaande voorbeelden werd de *function call* steeds beëindigd na het uitvoeren van een `return`-statement. We bekijken nu enkele **bijzondere gevallen**.

5.5.1 Meerdere return statements

Een functiedefinitie kan **meerdere return statements** bevatten. Wanneer zo'n functie wordt opgeroepen, dan wordt de **oproep beëindigd onmiddellijk na het eerste uitgevoerde `return`-statement**. De onderstaande functie `maximum()` bevat twee `return`-statements.

```
1 #%% Functiedefinities
2 def maximum(a, b):
3     if a > b:
4         return a
```

```
    else:
        return b

#%% Instructies
m = maximum(10, 12)
print("Oproep 1:", m)
```

De output is:

```
Oproep 1: 12
```

Het gebruik van meerdere `return`-statements kan, in sommige gevallen, de tijd die nodig is om een bepaalde taak te voltooien verkorten. De functie `bevatX()` in het onderstaande voorbeeld gaat na of een gegeven string (argument van de functie) het karakter `X` bevat. Om dit na te gaan wordt geïtereerd over alle karakters van de string (d.m.v. een `for`-lus). Zodra een karakter met waarde `"X"` gevonden wordt, retourneert de functie `True`. Indien, na het volledig doorlopen van de `for`-lus, geen `"X"` gevonden werd, wordt tenslotte `False` geretourneerd.

```
#%% Functiedefinities
def bevatX(zin):
    for karakter in zin:
        if karakter == "X":
            return True
    return False

#%% Instructies
res1 = bevatX("de K is het 4de karakter van deze zin")
print("Zin 1:", res1)
res2 = bevatX("de X is het 4de karakter van deze zin")
print("Zin 2:", res2)
```

De output is:

```
Zin 1: False
Zin 2: True
```

Opdracht 5.5 (`getaltheorie.py`)

Implementeer een functie `is_priemgetal` die één getal (parameter `a`) aanvaardt en dat nagaat of `a` een priemgetal is of niet. De functie moet `True` retourneren indien `getal` een priemgetal is, en `False` in het andere geval. Controleer hiervoor of 2, 3, 4, ..., `int(a**(1/2))` delers zijn van `a`. **Let op**: voor getallen met kleine delers (het getal 54643213212 is een **even** getal en dus duidelijk geen priemgetal) kan je reeds heel snel besluiten dat het getal in kwestie geen priemgetal is, zorg dat je functie dan ook snel retourneert door gebruik te maken van meerdere return-statements.

```
>>> resultaat = is_priemgetal(453)
>>> print(resultaat)
False
>>> is_priemgetal(17)
True
>>> is_priemgetal(54643213212) # mag niet lang duren
False
```

5.5.2 Functies zonder return-statement

Het `return`-statement is **geen noodzakelijk onderdeel** van een functiedefinitie in Python. Wanneer men een functie oproept die geen `return`-statement bevat, dan zal, na het uitvoeren van alle instructies in de body van de functie, `None` geretourneerd worden.

Functies met als doel het tonen (op het scherm) of bewaren (in een bestand) van gegevens, bevatten vaak geen `return`-statement. Het volgend voorbeeld illustreert dit:

```
#%% Functiedefinities
def toonGrootsteGetal(a, b):
    if a > b:
        print(a, "is groter dan", b)
    else:
        print(a, "is kleiner dan of gelijk aan", b)

#%% Instructies
toonGrootsteGetal(3, 5)
```

De output is:

```
3 is kleiner dan of gelijk aan 5
```

Merk op dat de return value van de functie `toonGrootsteGetal()` steeds `None` is.

```
>>> resultaat = toonGrootsteGetal(3, 5)
3 is kleiner dan of gelijk aan 5         # dit is NIET de return value
>>> print(resultaat)
None                                     # de return value is None
```

Opdracht 5.6 (`statistieken2.py`)

Deze opdracht is een vervolg op Opdracht 5.3. Roep de functies `listReadValues()`, `gemiddelde()` en `std()` op in een **nieuwe** functie `toonStatistieken()`. Deze functie aanvaardt één argument (parameter `bestandsnaam`) die de naam van een bestand met getallen bevat (zoals bv. `data_waarden.txt`). Wanneer deze functie opgeroepen wordt, moet de volgende informatie op het scherm verschijnen:

```
>>> toonStatistieken("data_waarden.txt")
----------------
      Waarden
----------------
       12.584
       15.330
       16.397
       11.005
       14.379
       12.045
----------------
Gemid: 13.623
Stdev:  2.079
```

Deze functie bevat **geen `return`-statement.**

5.5.3 Meerdere waarden retourneren

De return value in Python is steeds (een referentie naar) exact één object. Wensen we meerdere waarden te retourneren, dan moeten deze **ingekapseld** worden in een **container**, zoals bijvoorbeeld een `list` of een `tuple`, en wordt deze container geretourneerd. De Python syntax laat dit, bijna onopgemerkt, eenvoudig toe.

Beschouw de volgende instructies:

```
>>> a  = (3, "biggetjes")  # waarden gescheiden door komma's
                           # RONDE haakjes zijn optioneel
>>> type(a)
tuple                      # gegevenstype tuple

>>> print(a)
(3, 'biggetjes')
```

Tuple packing/unpacking
Het bovenstaande voorbeeld toont dat een tuple een container kan zijn voor objecten van verschillende gegevenstypes.

- Het inkapselen van objecten in een tuple noemt men **tuple packing**.
- Daartegenover staat **tuple unpacking**: daarbij zal men de objecten die zitten ingekapseld in een tuple toekennen aan (een even groot aantal) variabelen.

Tuple unpacking wordt geïllustreerd in het onderstaande voorbeeld.

```
>>> a  = (3, "biggetjes")   # tuple packing
>>> x, y = a                # tuple unpacking
>>> print(x)
3
>>> print(y)
biggetjes
```

Tuple packing/unpacking kan ook toegepast worden op de **return**-value van een functie:

```
#%% Functiedefinities
def somEnVerschil(a, b):
    som = a + b
    verschil = a - b
    return (som, verschil)   # tuple packing --> haakjes zijn optioneel

#%% Instructies
x, y = somEnVerschil(5, 2)   # tuple unpacking toegepast op return value
print("De som is:", x)
print("Het verschil is:", y)
```

De output is:

```
De som is: 7
Het verschil is: 3
```

Opdracht 5.7 (`transformaties2D.py`)

Beschouw een cartesiaans coördinatenstelsel en een punt met coördinaten (x, y). Wanneer men dit punt roteert rond de oorsprong over een hoek θ, dan zijn de nieuwe coördinaten van dit punt (x', y'). Men kan deze coördinaten berekenen met de volgende transformatieformules:

$$\begin{aligned} x' &= x\cos(\theta) - y\sin(\theta) \\ y' &= x\sin(\theta) + y\cos(\theta) \end{aligned}$$

Implementeer de functie `roteer()` die vier argumenten aanvaardt:

- `x`: de x-coördinaat van het punt (float)
- `y`: de y-coördinaat van het punt (float)
- `theta`: de **rotatiehoek** (float), met default waarde 0.0.
- `eenheid`: de **eenheid** waarin de hoek werd uitgedruk, een string met waarde `"radialen"` (de default waarde) of `"graden"`.

Deze functie retourneert (x', y'). Het onderstaande voorbeeld illustreert het gebruik van de functie `roteer()`.

```
>>> x_uit, y_uit = roteer(2, 3, theta = 1/2*math.pi)
>>> print(x_uit, y_uit)
-3.0 2.0

>>> x_uit, y_uit = roteer(2, 3, theta = 1/2*math.pi, eenheid = "graden")
>>> print(x_uit, y_uit)
1.9170120329428304 3.0536969177625335
```

Alternatief voor tuple unpacking: indexering

Een tuple is een geordende verzameling van objecten. Elk object heeft dus een bepaalde positie in de tuple. Deze positie wordt aangeduid met een index. Deze begint steeds bij 0. Toegang tot de objecten waaruit een tuple bestaat, gebeurt o.b.v. deze index en door gebruik te maken van de index operator `[]`. Het toepassen van deze operator noemt men **indexering** (*indexing*).

```
>>> data_tuple  = ("Een", 2, "drie")
>>> print(data_tuple)
('Een', 2, 'drie')

>>> print(data_tuple[0])   # eerste element heeft index 0
Een

>>> print(data_tuple[2])   # derde element heeft index 2
drie
```

Opmerking: indexering kan op verschillende gegevenstypes toegepast worden zoals strings (zie Hoofdstuk 6) en lists (zie Hoofdstuk 7).

5.6 Anonieme functies: *lambda functions*

In Python is het ook mogelijk om functies te creëren die geen naam hebben. De functie wordt aan een variabele toegekend die vervolgens kan gebruikt worden alsof het een functie is.

De syntax voor de aanmaak van een zogenaamde *lambda function* is:

```
variabeleNaam = lambda argumenten: expressie
```

Hierin is

- `variabeleNaam`: de variabele waaraan de anonieme functie toegekend wordt,
- `lambda`: een *key word* dat aangeeft dat het om een anonieme functie gaat,
- `argumenten`: de parameters waarvan de anonieme functie afhangt, **van elkaar gescheiden door komma's**, en

- `expressie`: één (al dan niet geneste) expressie die het resultaat van de anonieme functie bepaalt.

Een voorbeeld is de anonieme functie `schuin` die, gegeven de twee rechthoekszijden van een rechthoekige driehoek, de lengte van de schuine zijde berekent:

```
>>> schuin = lambda x, y: (x**2 + y**2)**0.5
```

Dit is equivalent met:

```
def schuin(x, y):
    z = (x**2 + y**2)**0.5
    return z
```

Om de anonieme functie op te roepen, volstaat het om aan `schuin` twee getallen mee te geven:

```
>>> schuin(3, 4)
5.0
```

5.7 Function docstrings en type hints

Docstrings en *type hints* maken code duidelijker, makkelijker te hergebruiken en te onderhouden (te updaten). Zowel docstrings als type hints hebben geen invloed op de werking van de functie, maar volgen wel een specifieke syntax.

Function docstrings zijn een gestructureerde manier om functies te documenteren. Ze worden systematisch onder de *header* van de functie geplaatst. Het onderstaande voorbeeld[8] toont een functie `add()` die de som berekent van twee inputs:

```
1  def add(a: int, b: int) -> int:
2      """
3      Add two numbers.
4
5      Parameters
6      ----------
7      a : int
8          The first number.
9      b : int
10         The second number.
11
12     Returns
13     -------
14     int
```

[8] Dit voorbeeld gebruikt de NumPy-style docstring, een variant is de Google style docstring.

```
15          The sum of the two numbers.
16
17      Examples
18      --------
19      >>> add(2, 3)
20          5
21      """
22      return a + b
```

De *docstring* bevindt zich tussen de drievoudige dubbele quotes (zie de """ op regels 2 en 21) en bevat de volgende onderdelen:

- een korte **beschrijving** van de functie,
- een **oplijsting (overzicht) van alle parameters**, inclusief hun **gegevenstype** en een korte beschrijving van elke parameter. Let op de uitlijning/formatting,
- een beschrijving van de `return` value, inclusief het gegevenstype,
- (**optioneel**) één of meerdere voorbeelden van **functie-oproepen**.

Type hints zijn aanduidingen van het gegevenstype van de argumenten en de return value in de header van de functie. Ze worden aangegeven door een `:` te plaatsen na elke parameter gevolgd door het gegevenstype. De type hint voor de return value wordt na een `->` geplaatst. Merk op dat type hints niet worden afgedwongen bij het oproepen van een functie, er gebeurt dus geen type-check o.b.v. deze hints. De gebruiker van een functie kan er dus voor kiezen deze type hints te negeren.

In bovenstaand voorbeeld is het gegevenstype steeds `int`. Dit hoeft niet steeds zo te zijn. Andere mogelijkheden zijn:

- `float`: decimaal getal,
- `str`: string,
- `bool`: logische waarden,
- `list`: lijsten (zie Hoofdstuk 7),
- `dict`: dictionary (zie Hoofdstuk 10)
- `numpy.ndarray`: een numpy array (zie Hoofdstuk 8), enkel beschikbaar indien de module numpy werd geïmporteerd.

Bij collection types kan eveneens het gegevenstype van de elementen van de collection aangeduid worden, bijvoorbeeld `list[int]`. Analoog kan bij dictionaries het gegevenstype van de *keys* en de *values* gespecificeerd worden bijvoorbeeld `dict[str, list]`.

Type hints kunnen ook buiten de functie header gebruikt worden. Bij het intialiseren van een dictionary kan men er bijvoorbeeld gebruik van maken om aan te geven dat de variabele `x` verwijst naar een dictionary waarvan de keys van het type `str` zijn en de values van het type `list`.

```
>>> x : dict[str, list] = {}
```

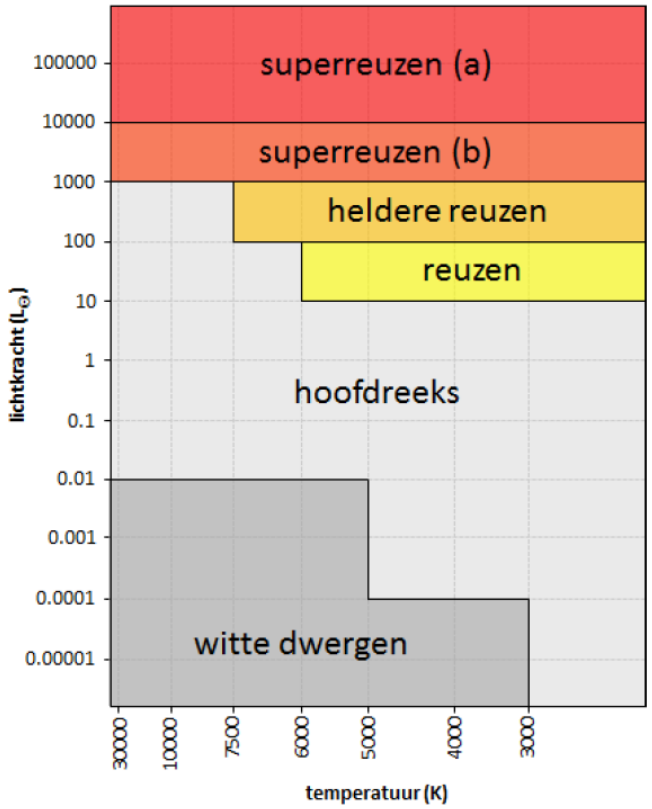
Enkele voordelen van het gebruik van type hints zijn:

- **Verbeterde leesbaarheid**: type hints geven aan welk type waarde een variabele, een parameter of de return value zou moeten hebben. Dit maakt de code begrijpelijker voor de gebruiker.
- **Duidelijkheid**: als je type hints gebruikt, wordt de bedoeling van de code duidelijker zonder dat je de details van de implementatie moet doornemen.
- **Ondersteuning door IDE's**: IDE's gebruiken type hints om betere codeaanvullingen en suggesties te geven tijdens het schrijven van code. Dit versnelt de implementatie en vermindert (de kans op) fouten.

5.8 Overkoepelende oefeningen

Opdracht 5.8 (`sterren.py`)

Men kan sterren classificeren[9] o.b.v. hun lichtkracht en temperatuur. Deze de classificatieregels worden voorgesteld in een zogenaamd Hertzsprung-Russelldiagram:



Op basis van dit diagram worden volgende klassen onderscheiden: superreuzen (a), superreuzen (b), heldere reuzen, reuzen, de hoofdreeks en witte dwergen. Een ster met een lichtkracht van 230 L_Θ en een temperatuur van 6500 K wordt bijvoorbeeld geclassificeerd als de klasse `'heldere reuzen'`.

Schrijf een functie `classificeer()` waaraan de temperatuur en de lichtkracht van een ster moeten doorgegeven worden. Deze functie retourneert de klasse waartoe deze ster behoort (string) volgens het Hertzsprung-Russelldiagram.

```
>>> classificeer(6500, 230)
'heldere reuzen'

>>> classificeer(3700, 0.007)
'hoofdreeks'
```

[9] Idee ontleend aan een oefening van Prof. Peter Dawyndt (Programmeren, Faculteit Wetenschappen, UGent).

Opdracht 5.9 (caesar02.py)

Schrijf een programma dat een zin kan vercijferen of ontcijferen volgens de Caesarcijfermethode. De sleutelparameter moet via de gebruiker ingegeven worden, evenals de keuze vercijferen of ontcijferen. Baseer je voor de vercijfering en ontcijfering op Opdracht 4.30 (caesar_vercijfering.py). Vul het bestand caesar02.py aan. Implementeer in je programma de volgende functies:

1. verschuif_karakter(kar, sleutel) die een karakter cyclisch naar links verschuift volgens de sleutelparameter. Volgende functiedefinitie kan je gebruiken:

```
def verschuif_karakter(kar, sleutel):
    if kar.islower():
        nieuwe_kar = \
            chr(((ord(kar) - ord("a") + sleutel) % 26) + ord("a"))
    elif kar.isupper():
        nieuwe_kar = \
            chr(((ord(kar) - ord("A") + sleutel) % 26) + ord("A"))
    else:
        nieuwe_kar = kar
    return nieuwe_kar
```

   De methode isupper() onderzoekt of het karakter kar een hoofdletter is. De methode islower() onderzoekt of het karakter kar een kleiner letter is.

2. de functie caesar_vercijfering(klare_tekst, sleutel) die klare tekst vercijfert volgens de sleutelparameter en de vercijferde tekst retourneert.

3. de functie caesar_ontcijfering(vercijferde_tekst, sleutel) die vercijferde tekst ontcijfert volgens de sleutelparameter en de ontcijferde tekst retourneert.
   Tip: het algoritme bij ontcijfering is analoog aan dat van vercijfering, het enige verschil is dat een **negatieve** sleutelparameter moet gebruikt worden in de implementatie van de functie.

Een mogelijke input/output voor vercijfering is:

```
Typ een zin:
Ontmoeting: 14h30, aan station!
Geef de sleutelparameter: 6
Vercijfer (1) of ontcijfer (2)? 1
De vercijferde tekst is:
Utzsukzotm: 14n30, ggt yzgzout!
```

Een mogelijke input/output voor ontcijfering is:

```
Typ een zin:
Utzsukzotm: 14n30, ggt yzgzout!
Geef de sleutelparameter: 6
Vercijfer (1) of ontcijfer (2)? 2
De ontcijferde tekst is:
Ontmoeting: 14h30, aan station!
```

6

Strings

6.1 Inleiding

Strings werden reeds in Hoofdstuk 3 geïntroduceerd als een gegevenstype waarin men tekstuele informatie kan onderbrengen. Dit gegevenstype is bovendien *built-in*, wat wil zeggen dat het automatisch beschikbaar is bij het opstarten van de interpreter.

Omdat strings reeds meermaals aan bod kwamen in de voorgaande hoofdstukken, worden hierna de belangrijkste aspecten van strings, die **reeds gekend** zijn, samengevat.

- Strings worden in Python geïmplementeerd door het *built-in* gegevenstype `str`.
- Een `str` object kan men aanmaken door meerdere karakters tussen dubbele quotes (aanhalingstekens) `""` te plaatsen[1]:

```
>>> a = "Het lijkt altijd onmogelijk, totdat het gedaan is."
>>> type(a)
str
```

- Een `str` object bevat een **sequentie van karakters**, wat erop duidt dat een string meerdere karakters bevat in een bepaalde volgorde[2].
- Een string is *iterable*. Dit wil zeggen dat ze een iterator kunnen genereren die kan gebruikt worden in een `for`-statement.

```
>>> spreuk = "Op de digitale snelweg kan je ook verongelukken."
>>> for ch in spreuk:
...     print(ch)
```

[1] Het is ook toegelaten om karakter te omgeven door enkele quotes (`''`), maar in deze cursus geven we de voorkeur aan dubbele quotes.

[2] Het gegevenstype `str` is wat men noemt een *sequence type*. Verder wordt dieper ingegaan op deze eigenschap.

- Het aantal karakters dat een string bevat kan bepaald worden met de functie `len()`.

```
>>> len(spreuk)
48
```

De volgende operatoren werden reeds beschreven:

- **Concatenatie** van strings met de `+` operator.

```
>>> "2" + " " + "micro" + "-" + "organismen"
'2 micro-organismen'
```

- **Duplicatie** van strings met de `*` operator.

```
>>> "organismen" * 2
'organismenorganismen'
```

- **Vergelijken** van strings met de relationele operatoren `<`, `>`, `<=`, `>=`, `==`, en `!=`.

```
>>> "test" == "testje" # strings zijn gelijk als ze even lang zijn en
False                  # alle karakters gelijk zijn
>>> "reus" > "dwerg"
True                   # orde bepaald door een (uitgebreid) alfabet
```

6.2 Indexering en slicing

Een string is een **geordende verzameling** van karakters. Elk karakter heeft dus een bepaalde **positie** in de string. Deze positie wordt aangegeven door de **index**. Figuur 6.1 toont de indices van de karakters in de string `"Bacillus anthracis"`. Merk op dat de **index start bij 0**. Merk ook op dat een spatie ook een karakter voorstelt en dus ook een index heeft.

Figuur 6.1: Indices van de karakters in de string `"Bacillus anthracis"`.

6.2.1 Indexering

Men kan toegang krijgen tot de karakters waaruit een string bestaat o.b.v. van de index, door gebruik te maken van de **index operator** `[]`. Het toepassen van deze operator noemt men **indexering**. Hieronder wordt indexering geïllustreerd:

```
>>> woord = "Bacillus anthracis"
>>> karakter = woord[2]
>>> karakter
'c'                              # karakter met index 2 is c

>>> type(karakter)               # resultaat is nieuwe string
str                              # bestaande uit 1 karakter
```

Uit dit voorbeeld blijkt dat het toepassen van indexering een nieuw string object genereert. Deze string bevat één karakter.

Indexering - voor index binnen bereik van string.
Beschouw een `str` object `s` en een geheel getal (`int`) `i`. De expressie

`s[i]`

retourneert:

- **indien** $0 \leq$ `i` $<$ `len(s)` $\rightarrow$ het karakter met index `i`
- **in alle andere gevallen** $\rightarrow$ een `IndexError`

Opmerkingen

1. Het eerste karakter in een string heeft index `0` en het laatste karakter heeft index `len(s) - 1`. Indien de index buiten dit bereik valt, dan wordt een **foutmelding** van het type `IndexError` opgeworpen:

   ```
   >>> woord = "hallo"
   >>> woord[7]
   IndexError: string index out of range
   ```

2. Elk karakter heeft een index, dus ook spaties (`" "`) en nieuwe lijn karakters (`"\n"`).

Indexering biedt een eenvoudige tool om de karakters van een string aan te spreken. Vaak wordt dit type indexering gebruikt in combinatie met `range`-iteratoren in een `for`-lus. In het volgend voorbeeld wordt gezocht naar de positie(s) van het karakter `"a"` in een string:

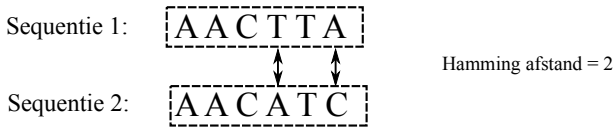
```
woord = "maandag"
for i in range(0, len(woord)):   # in laatste iteratie i = len(woord)-1
    if woord[i] == "a":
        print("Karakter", i, "is a.")
```

Het resultaat is:

```
Karakter 1 is a.
Karakter 2 is a.
Karakter 5 is a.
```

Opdracht 6.1 (`hamming.py`)

Indien twee DNA-sequenties even lang zijn, dan is de **Hamming afstand** tussen deze twee sequenties **gelijk aan het aantal posities waarop** deze **twee sequenties verschillen van elkaar**. Bekijk ter illustratie het onderstaande voorbeeld, waarin de Hamming afstand 2 is:



Schrijf een programma dat de gebruiker vraag om **twee DNA-strings** in te geven die **even lang** zijn (de lengte hoef je dus niet te controleren). Nadat deze sequenties werden ingegeven wordt de Hamming afstand berekend en op het scherm getoond. Voorbeeld input/output is:

```
Gelieve een 1e DNA sequentie in te geven: AACTTA
Gelieve een 2e DNA sequentie in te geven: AACATC
De Hamming afstand is 2
```

Indexeren met een negatieve index

Beschouw de string `"maandag"`. De index van het laatste karakter van deze string is 6. Algemeen is de index van het laatste karakter gelijk aan de lengte van de string verminderd met één. Toegepast op dit voorbeeld geeft dit `len("maandag") - 1`. Door gebruik te maken van **negatieve indices**, kan men dit vereenvoudigen. Index `-1` extraheert bijvoorbeeld het laatste element van een string, index `-2` het voorlaatste, enz.

```
>>> woord = "maandag"
>>> woord[len(woord) - 1]  # index laatste karakter is 7 - 1 = 6
'g'
>>> woord[-1]              # gebruik negatieve index (laatste karakter)
'g'
>>> woord[-2]
'a'                        # gebruik negatieve index (voorlaatste karakter)
```

Indexering - met een negatieve index

Beschouw een **`str`** object `s` en een geheel getal (**`int`**) `i`. De expressie

`s[i]`

retourneert:

- **indien** $0 \leq$ `i` $<$ `len(s)` $\rightarrow$ het karakter op index `i` .

- **indien** $-$`len(s)` $\leq$ `i` $\leq -1 \rightarrow$ het karakter met index `i + len(s)`
- **in alle andere gevallen** $\rightarrow$ een `IndexError`

6.2.2 Slicing

In de voorgaande sectie werd *indexering* gebruikt om één karakter te selecteren (kopiëren) uit een string. Men kan deze indexering uitbreiden zodat **meerdere karakters gelijktijdig** worden geselecteerd (of gekopiëerd). Om te benadrukken dat (vaak) meerdere karakters worden geëxtraheerd noemt men dit proces **slicing**.

Slices van de vorm `[i:j]`
Slicing gebeurt met dezelfde operator (`[]`) als indering. Maar in plaats van een enkele, gehele, index `i`, wordt nu gebruik gemaakt van een bereik. Dit wordt geïllustreerd in het onderstaande voorbeeld.

```
>>> woord = "Bacillus anthracis"
>>> woord[4:11]
'llus an'
```

Het resultaat is een **kopie van een deel** van de originele string, en dit noemt men vaak een **substring**. De karakters met een index gaande van `4` tot en met `10` worden **gekopieerd**.

Slicing - voor indices binnen bereik van string
Beschouw een **`str`** object `s` en twee gehele getallen (**`int`**) `i` (startindex) en `j` (eindindex). De expressie

`s[i:j]`

retourneert

- indien `i` $<$ `j` $\rightarrow$ een nieuwe string met daarin **een kopie** van de karakters met indices `i`, `i+1`, …, `j-1`
- indien `i` $\geq$ `j` $\rightarrow$ een **lege string**

Merk op dat het karakter met index `j` **niet** tot de slice zal behoren.

Opdracht 6.2 (`rijksregisternummer.py`)
Een Rijksregisternummer bestaat steeds uit 11 cijfers. De eerste 9 cijfers vormen samen een identificatiecode, en de laatste 2 cijfers zijn een controlegetal.

1. Schrijf een programma dat aan de gebruiker vraagt om een rijksregisternummer in te voeren en vervolgens de identificatiecode en het controlegetal op het scherm print. Voorbeeld input/output is:

```
Geef een RRnummer in: 78100601459
De identificatiecode is: 781006014
Het controlegetal is: 59
```

2. Bij een geldig rijksregisternummer geldt dat het resultaat van de bewerking

   ```
   97 - (identificatiecode % 97)
   ```

   gelijk moet zijn aan het controlegetal. Voor personen geboren in of na het jaar 2000 moet je het cijfer 2 plaatsen vóór de cijfers van de identificatiecode alvorens je de bovenstaande bewerking uitvoert. Bijvoorbeeld, iemand die in 2009 geboren is heeft een identificatiecode `091011097`, de bovenstaande bewerking moet je dan uitvoeren op `2091011097`. Breid je script uit zodat deze controle uitgevoerd wordt en tevens op het scherm verschijnt of een Rijksregisternummer geldig is (je mag ervan uitgaan dat de gebruiker **steeds exact 11 cijfers** ingeeft)[3].

```
Geef RRnumer in: 73091403970
Geboren in of na 2000? [J/n]: n
CONTROLE: RRnummer is geldig
```

```
Geef RRnumer in: 09101109755
Geboren in of na 2000? [J/n]: J
CONTROLE: RRnummer is geldig
```

```
Geef RRnumer in: 09101109755
Geboren in of na 2000? [J/n]: n
CONTROLE: RRnummer is NIET geldig
```

Slices van de vorm `[i:]`, `[:j]` en `[:]`

Slices die starten bij het begin van een string, of slices die doorgaan tot het einde van een string komen vrij vaak voor. Daarom werd voor een aantal van dit type slices een **verkorte syntax** voorzien, waarbij men de begin/eind index weglaat. Hierna volgen enkele voorbeelden:

```
>>> s = "Bacillus anthracis"
>>> s[9:]             # karakter met index 9 tot einde string
'anthracis'
>>> s[:8]             # begin tot en met karakter met index 7
'Bacillus'
>>> s[:]
'Bacillus anthracis'  # begin tot einde (dit heet een copy slice)
```

Tabel 6.1 vat deze opties samen (versie met uitgebreide syntax en verkorte syntax naast elkaar).

[3]Verder in dit hoofdstuk volgen oefeningen waarbij ook andere fouten moeten worden opgevangen, zoals ontbrekende cijfers, ongeldige karakters enz.

Tabel 6.1: Enkele bijzondere slices.

Slice	Verkorte syntax	Voorbeeld
`s[i:len(s)]`	`s[i:]`	`s[9:] --> 'anthracis'`
`s[0:j]`	`s[:j]`	`s[:8] --> 'Bacillus'`
`s[0:len(s)]`	`s[:]`	`s[:] --> 'Bacillus anthracis'`

De laatste optie van het type `s[:]` heet een **copy slice**. Merk het verschil op tussen een copy slice en een alias. Enkele voorbeelden:

```
>>> s = "Bacillus anthracis"
>>> x = s              # x is een alias (verwijst naar zelfde object als s)
>>> x                  # id(x) is gelijk aan id(s)
'Bacillus anthracis'

>>> y = s[:]           # y is een kopie (zelfde waarde maar ander object)
>>> y                  # id(y) is NIET gelijk aan id(s)
'Bacillus anthracis'
```

Opdracht 6.3 (`name_phone.py`)

De onderstaande strings zijn een aantal voorbeelden van records (uit een lange lijst) van een telefoonboek. Elke record bevat een achternaam, voornaam en telefoonnummer, gescheiden door komma's. Omdat de achternamen en voornamen **geen constante lengte** hebben, zijn de **indices** van deze **komma's variabel**.

```
"Eeckhout,Mia,09 242 42 76"
"Baert,Geert,09 242 42 78"
"Wouters,Geert,09 242 46 81"
```

Schrijf een programma dat aan de gebruiker vraagt om een achternaam, voornaam, telefoonnummer in te geven, gescheiden door komma's. Vervolgens moeten de achternaam, voornaam, telefoonnummer afzonderlijk op het scherm getoond worden, zoals hieronder geïllustreerd wordt.

```
>>> Geef familienaam, voornaam en tel in: Baert,Geert,09 242 42 78
Voornaam:       Geert
Familienaam:    Baert
Telefoonnummer: 09 242 42 78
```

Tips:

- zoek eerst de index van de komma's in de opgegeven string, en
- maak vervolgens gebruik van slicing om de verschillende substrings te extraheren.

Opdracht 6.4 (`name_phone_ctd.py`)

In de voorgaande opdracht heb je (vermoedelijk) geen *user-defined* functie gebruikt. Implementeer de functie `get_personal_info()` die:

- een string `infostring` als argument aanvaardt (zoals bv. `"Baert,Geert,09 242 42 78"`),
- als *return value* drie strings met daarin de voornaam, familienaam en telefoonnummer retourneert.

```
#%% Functiedefinities
def get_personal_info(infostring):
    #
    # vul aan
        #
    return (familienaam, voornaam, telefoonnummer) # () niet verplicht

#%% Instructies
infostr = input("Geef familienaam, voornaam en tel in:")
fn, vn, tel = get_personal_info(infostr)
```

In combinatie met de functie `range()` kan men een string eenvoudig onderverdelen in een sequentie van substrings met **vaste lengte** (hier 5). Beschouw de volgende string van opeenvolgende tijdstippen.

```
tijdstippen = "12u45 13u15 19u55 17u12 15u30"
```

Het volgende codefragment toont hoe men deze tijdstippen één voor één kan extraheren uit deze string.

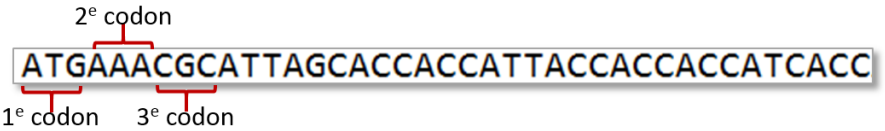
```
for i in range(0, len(tijdstippen), 6): # spatie wordt niet mee geselecteerd
    tijd = tijdstippen[i:i+5]
    print(tijd)
```

Opmerkingen:

- De stapgrootte is 6 en niet 5 omdat de spatie tussen de tijdstippen wordt meegerekend.
- Met `tijdstippen[i:i+5]` worden exact 5 karakters geselecteerd. De spatie achter elk tijdstip wordt dus **niet** geslecteerd.

Opdracht 6.5 (`telCodons.py`)

Beschouw de DNA-string uit de onderstaande figuur.



Indien dergelijke DNA-sequentie een eiwit codeert, dan worden de nucleotiden in de transscriptiefase per drie samengenomen en vertaald naar een aminozuur. Een opeenvolging van 3 nucleotiden noemen we een codon.

Implementeer een functie `telCodon()` die twee argumenten aanvaardt:

- `dna`: een DNA-sequentie (type **`str`**, ga ervan uit dat de lengte een veelvoud is van 3),
- `zoekcodon`: een codon (type **`str`**, bestaande uit drie karakters) waarvan we willen nagaan hoeveel keer het voorkomt in `dna`.

Deze functie overloopt de codons en `dna` telt hoeveel keer `zoekcodon` voorkomt. Het gebruik van deze functie wordt hieronder geïllustreerd.

```
>>> dna = "ATGAAATGAAAAAGG"  # ter illustratie: ATG AAA TGA AAA AGG
>>> telCodon(dna, "AAA")
2
>>> telCodon(dna, "ATG")
1
```

De indexing operator kan tevens gebruikt worden om karakters te extraheren die **niet aaneensluitend** zijn.

Slices van de vorm `[i:j:k]`
Voor een string `s` en drie integers i (startindex), j (eindindex) en k (stapgrootte) zal de expressie `s[i:j:k]` de karakters met volgende indices

$$i, \quad i+k, \quad i+2k, \quad i+3k, \quad \ldots, \quad < j$$

retourneren in een **nieuwe** string.

Merk op dat het karakter met index j **niet** tot de slice zal behoren.

Opmerkingen:

- Indien de **startindex afwezig** is, dan wordt deze impliciet vervangen door 0.
- Indien de **eindindex afwezig** is, dan wordt deze impliciet vervangen door de **lengte** van de string `len(s)`.
- In `[i:j:k]` kan, op voorwaarde dat `i > j` (**of beide afwezig** zijn) de **stapgrootte** (k) **negatief zijn**. De string zal dan van achter naar voor overlopen worden.
- De startindex `i` en/of eindindex `j` kunnen negatief zijn, op voorwaarde dat `i < j`.

Enkele voorbeelden

- `i < j` en **positieve** stapgrootte `k`:

```
>>> s = "Bacillus anthracis"
>>> s[2:12:3]          # karakters met index 2, 5, 8, 11
'cl t'                 # merk op dat karakter met index 8 een spatie is

>>> s[2:12:2]          # karakters met index 2, 4, 6, 8, 10  (12 NIET)
'clu n'
```

- Start- of eindindex afwezig:

```
>>> s = "Bacillus anthracis"
>>> s[::3]
'Biuahc'

>>> s[1::2]            # start bij index 1, stapgrootte 2
'ailsatrcs'
```

- `i > j` en **negatieve** stapgrootte `k`:

```
>>> s = "Bacillus anthracis"
>>> s[7:0:-1]          # karakters met index 7, 6, 5, ..., 1
'sullica'              # karakter met index 0 NIET

>>> s[7::-1]           # karakters met index 7, 6, 5, ..., 0
'sullicaB'             # karakter met index 0 WEL
```

- Negatieve startindex `i` en eindindex `j`:

```
>>> s = "Bacillus anthracis"
>>> s[-9:-2]          # len(s) - 9, ..., len(s) - 3
'anthrac'
```

6.3 De `in` operator

Beschouw de string

```
s = "parallellogram"
```

De string `"alle"` is een substring[4] van de string `s` omdat de sequentie van karakters `alle` voorkomt in `"parallellogram"`. De **`in`** operator werkt in op twee operanden van het type **`str`** en retourneert `True` indien het eerste operand een substring is van het tweede operand en retourneert `False` indien dit niet het geval is.

[4]Formele definitie van een substring: Beschouw de sequentie van karakters $s = k_1 k_2 k_3 \ldots k_n$. Elke sequentie $k_i k_{i+1} \ldots k_j$, zodat $1 \leq i \leq j \leq n$, noemen we een substring van s.

De `in` operator.
Beschouw twee strings `s1` en `s2`. De expressie

```
s1 in s2
```

retourneert `True` indien `s1` een substring is van `s2` en `False` indien dit niet het geval is.

De voorbeelden hieronder illustreren het gebruik van deze operator.

```
>>> "gram" in "parallellogram"
True
>>> "hello" in "parallellogram"
False     # "ello komt wel voor maar "hello" NIET
```

```
>>> "a" in "aeiou"
True
>>> "" in "aeiou"   # lege string is steeds substring
True
```

Opdracht 6.6 (`telefoonlijst01.py`)
Het bestand `telefoonlijst.csv`[5] bevat de namen en werk-telefoonnummers van de personeelsleden van de FWB op campus Schoonmeersen. Als je met een teksteditor de inhoud van dit bestand bekijkt, dan krijg je het volgende te zien:

```
Eeckhout,Mia,09 242 42 76
Everaert,Helena,09 242 42 96
Fonseca,Maria,09 242 26 50
...
```

We wensen een eenvoudig zoekprogramma te implementeren dat de gebruiker toelaat om personen te zoeken in deze lijst.

1. Schrijf een script dat de gebruiker vraagt om een (deel van een) naam in te geven. Alle regels waarin deze naam terug te vinden is, moeten op het scherm getoond worden:

   ```
   >>> Geef een (deel van een) naam in: Geer
   Baert,Geert,09 242 42 78
   Haesaert,Geert,09 242 42 70
   ```

2. Breid je script uit: gebruik de functie `get_personal_info()` uit Opdracht 6.3 zodat de output een **betere formatting** krijgt.

[5] De extensie van dit bestand is `.csv` wat staat voor *Comma Seperated Values*. Dit is een tekstbestand waarin elke regel een rij voorstelt in een tabel. De verschillende waarden zijn gescheiden door komma's.

```
>>> Geef een (deel van een) naam in: Geer
* Voornaam:       Geert
* Familienaam:    Baert
* Telefoonnummer: 09 242 42 78
=======================================
* Voornaam:       Geert
* Familienaam:    Haesaert
* Telefoonnummer: 09 242 42 70
=======================================
```

6.4 String methoden

Het gegevenstype **str** dat strings implementeert, voorziet, naast de mogelijkheid tot indexeren, nog een aantal andere functionaliteiten die het werken met strings vereenvoudigen. Deze functionaliteiten worden aangeboden onder de vorm van **methoden**.

6.4.1 Inleiding

Methoden kunnen beschouwd worden als bijzondere functies. Ze onderscheiden zich van *gewone* functies doordat ze gebonden zijn aan (een object van) een bepaald gegevenstype. Net als functies kunnen methoden opgeroepen worden (method-call), kunnen ze argumenten aanvaarden, enz.

Een uitgebreid overzicht van de methoden die beschikbaar zijn voor **str** objecten volgt later. Hieronder illustreren we het gebruik van methode op basis van de **str.upper()**.

```
>>> s = "Bacillus anthracis"
>>> x = s.upper()           # oproep van de methode
>>> x
'BACILLUS ANTHRACIS'        # genereert nieuwe string --> hoofdletters
>>> type(x)
str

>>> s
'Bacillus anthracis'        # originele string s blijft ongewijzigd
```

De expressie `s.upper()` roept de string methode `upper()` op en past deze toe op de string `s`. Deze oproep retourneert een **kopie** van `s` waarin alle kleine letters vervangen werden door grote letters. **Merk op** dat deze methode **geen** argumenten aanvaardt.

Een tweede voorbeeld is de methode **str.find()**. Deze methode aanvaardt als **verplicht** argument een substring van `s` en retourneert de **index van het eerste karakter** van deze substring in `s`. Indien de substring meerdere keren voorkomt, wordt enkel de index van het **eerste voorkomen** geretourneerd.

```
>>> s = "Bacillus anthracis"
>>> ind = s.find("cil")
>>> ind
2                          # de c van "cil" heeft index 2 in s

>>> s.find("cis")
15                         # de c van "cis" heeft index 15 in s
```

Uit deze voorbeelden leiden we de algemene syntax voor het oproepen van een string methode af.

String methoden oproepen
Beschouw een string `s` en een methode *methodeNaam*. De expressie:

`s.`*methodeNaam*`(`*argument1*`,` *argument1*`, ...)`

past de methode *methodeNaam* toe op de string `s` en retourneert het resultaat.

Opmerking: string methoden zullen het object `s` zelf **nooit wijzigen**.

De belangrijkste string methoden met betrekking tot deze cursus worden weergegeven in Tabel 6.2. Een uitgebreide versie kan je terugvinden in Tabel 6.3.

Tabel 6.2: Enkele veelgebruikte string methoden.

Methode	Beschrijving
count()	Returns the number of times a specified value occurs in a string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
isalpha()	Returns True if all characters in the string are in the alphabet
isnumeric()	Returns True if all characters in the string are numeric
lower()	Converts a string into lower case
replace()	Returns a string where a specified value is replaced with a specified value
split()	Splits the string at the specified separator, and returns a list
strip()	Returns a trim version of the string
upper()	Converts a string into upper case
join()	Joins the elements of an iterable to the end of the string

Deze methoden worden hieronder kort besproken.

Opmerking: het gebruik van deze methoden is vrij eenvoudig. De uitdaging bij het gebruik van deze methoden ligt vaak in de vertaling van een (tekstbewerkings)probleemstelling in een reeks van expressies waarin ze gebruikt worden.

6.4.2 Methoden met een `str` object als return value

6.4.2.1 De methodes `upper()` en `lower()`

Method help/signature:

```
s.upper() -> str
s.lower() -> str
```

- De methode `upper()` zet een string `s` om in **hoofd**letters.
- De methode `lower()` zet een string `s` om in **kleine** letters.
- Cijfers en leestekens worden **niet** gewijzigd.

Enkele voorbeelden:

```
>>> a = "Goede MORGEN, het is 10u15!"
>>> a.upper()
'GOEDE MORGEN, HET IS 10U15!'
>>> a.lower()
'goede morgen, het is 10u15!'
```

Opmerking: De originele string `a` blijft ongewijzigd. De methoden `upper()` en `lower()` retourneren enkel een **kopie** van de string `a` waarin de karakters zijn omgezet naar hoofdletters resp. kleine letters.

Opdracht 6.7 (`gebruikerinput_stoppen.py`)
Vul het onderstaande codefragment aan zodat de gebruiker herhaaldelijk om een woord gevraagd wordt. Dit woord moet onmiddellijk getoond worden op het scherm. Indien de gebruiker `stoppen` of `stoPPen` of `STopPen` of ... (elke mogelijk combinatie van kleine letters en grote letters) ingeeft moet het programma beëindigd worden.
Vul het onderstaande codefragment aan zodat de gebruiker herhaaldelijk om een woord gevraagd wordt. Dit woord moet onmiddellijk getoond worden op het scherm. Indien de gebruiker `stoppen` of `stoPPen` of `STopPen` of ... (elke mogelijke combinatie van kleine letters en grote letters) ingeeft, moet het programma beëindigd worden. Gebruik de volgende structuur:

```
woord = input("Geef een woord in: ")

while ............................ :

    print("Je gaf", woord, "in")
    woord = input("Geef een woord in: ")
```

6.4.2.2 De methode `replace()`

Method help/signature:

```
s.replace(old, new[, count]) -> str
```

- Deze methode retourneert een kopie van `s` waarin elk voorkomen van `old` wordt vervangen door `new`.
- Indien een waarde (`int`) voor de parameter `count` wordt meegegeven, dan worden enkel de eerste `count` vervangingen doorgevoerd.

- Deze methode retourneert een **kopie** van `s` waarin elk voorkomen van `old` wordt vervangen door `new`.
- Indien een waarde (`int`) voor de parameter `count` wordt meegegeven, dan worden enkel de eerste `count` vervangingen doorgevoerd.

Enkele voorbeelden:

```
>>> a = "banaan"
>>> b = a.replace("a", "o")  # vervang "a" door "o"
>>> b
'bonoon'

>>> a
'banaan'                     # originele string blijft onaangepast

>>> "Hiep, hiep, hiep, Hoera!".replace("ie", "o", 2)
'Hop, hop, hiep, Hoera!'     # eerste twee "ie" vervangen door "o"
```

Een **bijzondere toepassing** van deze methode bestaat uit het **verwijderen** van een specifiek karakter uit een string.

```
>>> zin = "Hiep! hiep! hiep! ... Hoera!".replace("!", "")  # "": lege string
'Hiep hiep hiep ... Hoera'      # uitroepteken vervangen door lege string
```

6.4.2.3 De methode `strip()`

Method help/signature:

```
s.strip(chars) -> str
```

- Indien `chars` (type `str`) **niet** wordt meegegeven, retourneert deze methode een **kopie** van `s` waarin alle whitespace-characters (spaties, tabs, nieuwelijnkarakters) die **vooraan**

en/of **achteraan** de string voorkomen, zijn verwijderd.

- Indien `chars` (type **`str`**) wordt meegegeven, worden de karakters in `chars` vooraan en achteraan verwijderd uit `s`.

Enkele voorbeelden:

```
>>> a ="!! Hallo! Is het mooi weer vandaag?!"
>>> a.strip("!?")
' Hallo! Is het mooi weer vandaag'  # spatie voor Hallo blijft

>>> b = "   Dag    allemaal   "     # bemerk de spaties vooraan en achteraan
>>> b.strip()
'Dag    allemaal'                   # spaties werden verwijderd
```

6.4.2.4 De methode `join()`

Method help/signature:

```
sep.join(iterable) -> str
```

- Deze methode zal alle strings in `iterable` concateneren, waarbij `sep` als scheidingsteken gebruikt wordt, en retourneren als een nieuwe string.
- Heel vaak is `iterable` een **lijst** van strings.
- Deze methode zal alle strings in `iterable` concateneren, waarbij `sep` als scheidingsteken gebruikt wordt, en retourneren als een **nieuwe** string.
- Heel vaak is `iterable` een **lijst** van strings.

Enkele voorbeelden:

```
>>> woordenlijst = ["Het", "is", "maandag"]  # list
>>> scheidingsteken = "++"
>>> scheidingsteken.join(woordenlijst)
'Het++is++maandag'

>>> "".join(woordenlijst)   # scheidingsteken is lege string
'Hetismaandag'
```

6.4.3 Methoden met een `list` object als return value

6.4.3.1 De methode `split()`

Method help/signature:

```
s.split(sep = None, maxsplit = -1) -> list of strings
```

Deze methode splits `s` in een lijst van substrings waarbij `sep` het scheidingsteken is waarop gesplitst wordt.

- De default `sep = None` geeft aan dat er wordt gesplitst op whitespace-characters (spaties, tabs, nieuwelijnkarakters).
- De parameter `maxsplit` geeft aan hoeveel keer maximaal gesplitst mag worden. De **default** `maxsplit = -1` geeft aan dat het aantal splitsingen **onbeperkt** is.

Enkele voorbeelden:

```
>>> een_string = "Hallo! het is maandag!"
>>> een_string.split()                # geen scheidingsteken opgegeven
['Hallo!', 'het', 'is', 'maandag!']   # leestekens worden niet verwijderd

>>> bewerking = "5 + 7 = 12"
>>> bewerking.split("=")              # splits op gelijkheidsteken
['5 + 7 ', ' 12']
```

6.4.4 Methoden met een `bool` object als return value

6.4.4.1 De methoden `isnumeric()` en `isalpha()`

Method help/signature:

```
s.isalpha() -> bool
s.isnumeric() -> bool
```

- De methode `isalpha()` gaat na of **alle** karakters in `s` **letters** zijn (a, b, …, z of A, B, …, Z) en retourneert `True` indien dit het geval is. Zodra minstens één karakter geen letter is (bv. leesteken, spatie of cijfer) retourneert de methode `False`.
- De methode `isnumeric()` gaat na of **alle** karakters in `s` **cijfers** (i.e. `'0'`, `'1'`, `'2'`,…, `'9'`) voorstellen en retourneert `True` indien dit het geval is.

Enkele voorbeelden:

```
>>> "hallo".isalpha()
True
>>> "hallo!".isalpha()       # uitroepteken ! is geen letter
```

```
False

>>> "123".isnumeric()
True
>>> "123.4".isnumeric()      # decimaalteken . is geen cijfer
False
>>> "123piano".isnumeric()   # letters aanwezig
False
```

Opdracht 6.8 (`toespraak01.py`)

Het tekstbestand `toespraak_koning_filip_wo_1.txt` bevat een toespraak van Koning Filip. De functie `stringRead()` uit de module `infoFun` kan je gebruiken om dit tekstbestand in te lezen als één lange string:

```
>>> import infoFunWP as infoFun
>>> toespraak = infoFun.stringRead("toespraak_koning_filip_wo_1.txt")
>>> print(toespraak[:50])    # eerste 50 karkters
Majesteit, Dames en Heren Staatshoofden, Koninklij
```

1. Schrijf een programma waarin deze tekst wordt ingelezen (zie boven) en vervolgens alle woorden waaruit de tekst bestaat onder elkaar geprint worden op het scherm. Hou er rekening mee dat:
   - In de getoonde lijst **geen leestekens** zoals punten, komma's en dubbele punten (. , :) **of cijfers** (i.e. `'0'`, `'1'`, `'2'`, ..., `'9'`) voorkomen.
   - **Alle woorden uit kleine letters** moeten bestaan.

   De output van je script is:

   ```
   ...
   echte
   verzoening
   en
   een
   gemeenschappelijk
   project
   ```

   Vóór je begint:
   Je zal gebruik moeten maken van de string methoden die hiervoor besproken werden, lees dus eerst Sectie 6.4 aandachtig.

2. **Breid je script uit**: wrap de code die je schreef in een functie `toonWoordenlijst()`, die een (lange) string aanvaardt als argument, en de individuele woorden in deze string onder elkaar op het scherm brengt (zoals hiervoor). Je functie retourneert niets.

```
>>> toespraak = infoFun.stringRead("toespraak_koning_filip_wo_1.txt")
>>> toonWoordenlijst(toespraak)
...
een
gemeenschappelijk
project
```

6.4.5 Methoden met een `int` object als return value

6.4.5.1 De methode `count()`

Method help/signature:

```
s.count(sub[, start[, end]]) -> int
```

Deze methode telt hoeveel keer de substring `sub` voorkomt in `s`.

- De **optionele** parameters `start` en `end` (**niet** inbegrepen) laten toe om de **start- en eindindex** van het deel van de string waarover de telling moet gebeuren te specifiëren.

Enkele voorbeelden:

```
>>> zin = "Hiep, hiep, hoera!"
>>> zin.count("ie")
2

>>> zin.count("h")
2                                  # hoofdlettergevoeligheid

>>> zin.count("ie", 5, len(zin))  # start = 5, end = lengte zin
1
```

6.4.5.2 De methode `find()`

Method help/signature:

```
s.find(sub[, start[, end]]) -> int
```

Deze methode retourneert de laagste (eerste) index in `s` waarop men substring[a] `sub` kan terugvinden.

- De **optionele** parameters `start` en `end` (inbegrepen) laten toe om de **start- en eindindex** van het deel van de string waarop de zoekoperatie wordt toegepast te beperken.

- Indien `sub` niet voorkomt, wordt `-1` geretourneerd.

[a]De substring kan ook uit slechts 1 karakter bestaan.

Enkele voorbeelden:

```
>>> zin = "Hiep, hiep, hoera!"
>>> zin.find("ep")              # de "e" van "ep" heeft index 2 in zin
2

>>> zin.find("ep", 5)           # zoekproces start pas bij index 5
8                               # eerste voorkomen van "ep" vanaf index 5
                                # start op index 8
>>> zin.find("hop")
-1                              # "hop" is geen substring
```

Opdracht 6.9 (`substring_posities.py`)

Implementeer een functie `toon_substring_posities()` die alle posities op het scherm toont waarop een `substring` voorkomt in een gegeven `tekststring`[6]. Je functie moet de volgende vorm hebben:

```
def toon_substring_posities(tekststring, substring, hoofdletgev = True):
    #
    # vul zelf aan ( gebruik o.a. find, print, ... )
    #
    # GEEN return statement (functie heeft geen return value)
```

Zorg ervoor dat de gebruiker kan kiezen om het zoekproces al dan niet **hoofdlettergevoelig** te maken (default `hoofdletgev` is `True`, dus wel hoofdlettergevoelig).

```
>>> zin = "Hiep, hiep, hiep hoera!"
>>> toon_substring_posities(zin, "hie")        # hoofdlettergevoelig
Posities: 6   12    # 3 spaties tussen de posities
>>> toon_substring_posities(zin, "hie", False) # niet hoofdlettergevoelig
Posities: 0   6   12

>>> toespraak = infoFun.stringRead("toespraak_koning_filip_wo_1.txt")
>>> toon_substring_posities(toespraak, "oorlog")
Posities: 1015   1833   2020   2441   2479   2521   3229
```

[6]Je mag ervan uitgaan dat de substring geen repetitieve delen bevat (bv. indien `tekststring = "BAAAABCA"` en `substring = "AAA"`, dan is niet ondubbelzinnig bepaald hoeveel keer `"AAA"` voorkomt.)

6.4.6 Uitgebreid overzicht van string methoden

Tabel 6.3: Uitgebreid overzicht van string methoden.

Methode	Beschrijving
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Joins the elements of an iterable to the end of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trim version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

6.5 Formatteren van strings

In verschillende situaties moeten strings, integers, floats, enz. op een - voor een mens duidelijk **geformatteerde** (gestructureerde) manier - samengebracht worden in een string, vaak met als doel deze string te **tonen op het scherm** of te **bewaren in een tekstbestand**.

Met **formatteren** bedoelen we o.a.:

- **hoeveel plaats** (karakters) er voorzien moet worden voor een bepaald getal,
- **hoeveel cijfers na de komma** (precisie) moeten er getoond worden bij een *float*,
- **uitlijnen**: links, rechts of centreren,
- **lege karakterplaatsen opvullen** met bv. nullen, ...

Er bestaan in Python verschillende tools om strings te formatteren:

- **f-strings**,
- de modulo operator %, en
- de methode `format()`.

In hetgeen volgt, zullen we de laatste twee technieken kort toelichten[7], maar het gebruik van f-strings uitgebreid behandelen.

6.5.1 f-strings

In Hoofdstuk 4 hebben we de syntax en het gebruik van f-strings geïntroduceerd (zie Sectie 4.6). We zullen hier een meer gedetailleerde beschrijving geven. Voor een volledige beschrijving verwijzen we naar: `https://realpython.com/python-f-strings/`.

Om een string als f-string te gebruiken, plaatsen we de letter f (of F) vóór de aanhalingstekens van de string en gebruiken we accolades { } (zogenaamde *placeholders*) om de variabelen aan te geven die we willen formatteren en invoegen in de string.

Een eerste voorbeeld

```
>>> codon = "CGT"
>>> fractie = 0.2095
>>> f"De fractie van het codon {codon} is {fractie}"
```

In de praktijk wordt het **type** van de variabele steeds aangegeven door de variabele te laten volgen door een dubbelpunt, en een

- `s` van *string* voor **tekstuele** data, of

[7] Er staan vertikale grijze lijnen bij deze subsecties. Dit betekent dat deze subsecties louter ter informatie dienen en zonder problemen overgeslagen kunnen worden.

- `d` van *digit* voor **gehele** getallen, of
- `f` van *float* voor **decimale** getallen.

De *placeholders* (accolades) kunnen verschillende zaken bevatten, waaronder:

- tekstuele en/of numerieke waarden,
- variabelen,
- expressies,
- functie-oproepen, en
- *modifiers*

Met *modifiers* kunnen het aantal voorziene **plaatsen**, de **preciesie**, de **uitlijning** en de **opvulling** gespecificeerd worden.

Voorbeelden

```
# functie definitie
def berekenBMI(gewicht, lengte):
    bmi = gewicht/lengte**2
    return bmi

# variabelen
naam = "John"
lengte = 1.73
gewicht = 65
bmi = gewicht/lengte**2

# variabelen in f-string
f"De BMI van {naam} is {bmi}"

# expressie in f-string
f"De BMI van {naam} is {gewicht/lengte**2}"

# functie-oproep in f-string
f"De BMI van {naam} is {berekenBMI(gewicht, lengte)}"
```

Het resultaat van alle f-strings is dezelfde, namelijk:

```
'De BMI van John is 21.71806608974573'
```

6.5.2 Aantal voorziene plaatsen

We kunnen wat **meer structuur in de layout** brengen en bijvoorbeeld een vaste kolombreedte voorzien voor een betere uitlijning door het aantal karakters (plaatsen) te specificeren. Dat kan door na het dubbelpunt het aantal plaatsen op te geven. De syntax hiervoor is:

`{variable:xs}` of `{variable:xd}` of `{variable:xf}`

met

- `variable`: de te formatteren variabele,
- `x`: het **totaal aantal karakters** dat voorzien moet worden, en
- `s`, `d` of `f`: het **type** van de te formatteren variabele (`s` → string, `d` → geheel getal of `f` → decimaal getal).

Enkele voorbeelden:

```
>>> f"Naam van de patiënt: {naam:10s}"
>>> f"Gewicht (in kg): {gewicht:5d}"
```

Het resultaat van deze f-strings is:

```
'Naam van de patiënt: John      '
'Gewicht (in kg):    65'
```

Vaststellingen:

- **Getallen** worden default **rechts** uitgelijnd, **strings** worden default **links** uitgelijnd.
- getallen worden **vooraan** automatisch aangevuld met spaties: er staan 3 extra spaties vóór 65
- Strings worden **achteraan** automatisch aangevuld met spaties: er staan 6 extra spaties na John.

6.5.3 Precisie

De precisie, i.e. het aantal cijfers na de komma, kan **enkel** bij het gegevenstype `float` gespecificeerd worden. De syntax hiervoor is:

`{variable:x.yf}`

met

- `variable`: de te formatteren variabele,
- `x`: het **totaal aantal karakters** dat voorzien moet worden, en
- `y`: het **aantal cijfers na de komma**.

Een voorbeeld:

```
f"De BMI van {naam:s} is {bmi:6.2f}"
```

Met als resultaat:

```
'De BMI van John is  21.72'
```

Vaststellingen:

- Het **decimaal teken (.) neemt ook een plaats in**: bv. `21.72` neemt 5 plaatsen in.
- Er wordt **afgerond** volgens de gekende afrondingsregels.
- Er wordt **vooraan aangevuld** met spaties: er staat 1 extra spatie vóór `21.72`.

6.5.4 Uitlijning

We hebben gezien dat

- **getallen** default **rechts** worden uitgelijnd, en,
- **strings** default **links** worden uitgelijnd.

De uitlijning kan gewijzigd worden door net na het dubbelpunt een **aligneringssymbool** te plaatsen:

- `<` → **links** uitlijnen (default voor strings),
- `>` → **rechts** uitlijnen (default voor getallen),
- $\wedge$ → **centreren**,

Enkele voorbeelden:

```
>>> codon = "CGT"
>>> f"{codon:>7s}"    # RECHTS uitlijnen
>>> f"{codon:^7s}"    # CENTREREN
>>> aantal = 53
>>> f"{aantal:<7d}"   # LINKS uitlijnen
```

Het resultaat van deze f-strings is:

```
'    CGT'        # rechts uitgelijnd: 4 spaties vóór CGT
'  CGT  '        # gecentreerd: 2 spaties vóór en 2 spaties na CGT
'53     '        # links uitgelijnd: 5 spaties na 53
```

Opmerking: indien bij het **centreren** het aantal plaatsen verminderd met de lengte van de string of het getal oneven is, wordt een extra spatie na de string of het getal geplaatst:

```
>>> f"{codon:^8s}"
```

Het resultaat van deze f-strings is:

```
'  CGT   '        # gecentreerd: 2 spaties vóór en 3 spaties na CGT
```

Er worden 8 plaatsen voorzien en de lengte van de string is 3. Vermits 8 - 3 = 5 oneven is, worden 2 spaties vóór de string en 3 spaties na de string geplaatst.

6.5.5 Opvulling

Bij het uitlijnen worden standaard extra spaties geplaatst. Men kan ervoor kiezen een ander opvulsymbool (letter, cijfer of ander karakter) te gebruiken:

```
>>> codon = "CGT"
>>> f"{codon:-<7s}"    # opvullen met KOPPELTEKENS
>>> f"{codon:_>7s}"    # opvullen met UNDERSCORES
>>> f"{codon:.^7s}"    # opvullen met PUNTEN
>>> aantal = 53
>>> f"{aantal:0>7d}"   # opvullen met NULLEN
```

Het resultaat van deze f-strings is:

```
'CGT----'        # links uitgelijnd: 4 koppeltekens na CGT
'____CGT'        # rechts uitgelijnd: 4 underscores vóór CGT
'..CGT..'        # gecentreerd: 2 punten vóór en 2 punten na CGT
'0000053'        # rechts uitgelijnd: vijf 0'en vóór 53
```

Opmerking: vermits **strings default links** worden uitgelijnd, is het karakter < in de eerste f-string overbodig. Een gelijkaardige redenering geldt voor het karakter > in de laatste f-string: deze is overbodig daar **getallen steeds rechts** worden uitgelijnd.

6.5.6 De modulo operator %

De modulo operator (%) was de eerste tool voor string formatting in Python. Het maakt gebruik van *conversion specifiers* om te bepalen wat en hoe de variabelen moeten geformatteerd worden. Een eerste voorbeeld zal het gebruik verduidelijken:

```
>>> codon = "CGT"
>>> fractie = 0.2095
>>> "De fractie van het codon %s is %f." % (codon, fractie)
```

Om gebruik te maken van de modulo operator hebben we 3 zaken nodig:

- een string met één of meer *conversion specifiers*,
- de modulo operator, en

- een tuple met de te formatteren variabelen/waarden

In bovenstaand voorbeeld werd de specifier `%s` gebruikt voor strings (de `s` staat voor string) en de specifier `%f` voor decimale getallen (de `f` staat voor float).

Het resultaat is:

```
'De fractie van het codon CGT is 0.209500.'
```

We stellen vast dat decimale getallen default met 6 cijfers na de komma worden weergegeven. De precisie kan als volgt gespecificeerd worden:

```
>>> "De fractie van het codon %s is %4.2f." % (codon, fractie)
```

Hierin betekent `%4.2f`: voorzie 4 plaatsen voor het decimaal getal en gebruik 2 plaatsen voor het deel na de komma.

Het resultaat hiervan is:

```
'De fractie van het codon CGT is 0.21.'
```

Merk op dat de gekende afrondingsregels werden toegepast: 0.2095 werd afgerond tot 0.21.

Een uitgebreide beschrijving van het gebruik van de modulo operator voor string formatting is terug te vinden op: `https://realpython.com/python-modulo-string-formatting/`.

6.5.7 De methode `format()`

De syntax van de methode `format()` wordt beschreven in het volgende kader.

Method help/signature:

```
s.format(*args, **kwargs) -> str
```

- De notatie `*args` geeft aan dat deze methode een **variabel aantal positionele** argumenten kan aanvaarden.
- De notatie `**kwargs` geeft aan dat de methode bovendien een **variabel aantal keyword** argumenten kan aanvaarden.

De methode `format()` laat toe om het samenstellen of formatteren van strings te vereenvoudigen. Hierbij wordt gebruik gemaakt van placeholders. Deze herkennen we aan de accolades `{}` en formattering binnen die accolades.

Een uitgebreide beschrijving van het gebruik van `format()` is terug te vinden op: `https:`

`//www.w3schools.com/python/ref_string_format.asp`. Hier beperken we ons tot het volgende.

6.5.7.1 Placeholders en formatting types

Placeholders zijn te herkennen aan de accolades `{}` en bepalen **wat** en **hoe** er geformatteerd moet worden. We beperken ons tot variabelen van het type `int`, `float` en `str`. De placeholders voor deze gegevenstypes zijn:

- `{:d}` → gehele getallen (`int`),
- `{:f}` → decimale getallen (`float`),
- `{:s}` → strings (`str`).

Enkele voorbeelden:

```
>>> zin = "Vandaag is het {:d} {:s}"
>>> zin.format(26, "september")  # {:d} vervangen door 26,
'Vandaag is het 26 september'    # {:s} vervangen door september

>>> getal = 3/8        # = 0.375
>>> "{:f}".format(getal)
'0.375000'
```

Vaststellingen:

- Wordt het aantal plaatsen dat voorzien moet worden niet gespecificeerd, dan worden er bij gehele getallen en strings **evenveel karakters gebruikt als nodig**.
- Bij decimale getallen worden **standaard 6 cijfers na de komma** weergegeven.

6.5.7.2 Aantal voorziene karakters

Indien we wat **meer structuur in de layout** willen brengen en bijvoorbeeld een vaste kolombreedte willen voorzien voor een betere uitlijning, dan kan het aantal karakters (plaatsen) gespecificeerd worden.

Het opgeven van het aantal voorziene karakters is **gelijkaardig** als bij f-strings (zie 6.5.2): het volstaat `variable` weg te laten.

Aantal karakters
Om het aantal karakters te specificeren, volstaat het om dat aantal tussen het dubbelpunt (`:`) en het type:

- `{:3d}` → 3 karakters voor het geheel getal,

- `{:10f}` → 10 karakters voor het kommagetal,
- `{:6s}` → 6 karakters voor de string.

Enkele voorbeelden:

```
>>> zin = "Vandaag is het {:3d} {:6s}"
>>> zin.format(1, "mei")
'Vandaag is het   1 mei   '

>>> getal = 3/8        # = 0.375
>>> "{:10f}".format(getal)
'  0.375000'
```

Vaststellingen:

- **Getallen** worden default **rechts** uitgelijnd, **strings** default **links** uitgelijnd.
- Gehele getallen en kommagetallen worden **vooraan** automatisch aangevuld met spaties.
- Strings worden **achteraan** automatisch aangevuld met spaties.

6.5.7.3 Precisie

De precisie, i.e. het aantal cijfers na de komma, kan enkel bij het gegevenstype `float` gespecificeerd worden. Ook de specificatie van de precisie is **gelijkaardig** als bij f-strings (zie 6.5.3).

Aantal cijfers na de komma
Het aantal cijfers na de komma kan als volgt gespecificeerd worden:

`{:x.yf}`

met

- `x`: het **totaal aantal karakters** dat voorzien moet worden, en
- `y`: het **aantal cijfers na de komma**.

Enkele voorbeelden:

```
>>> getal = 12.365
>>> "{:10.3f}".format(getal)
'    12.365'   # 4 extra spaties vooraan

>>> "{:10.1f}".format(getal)
'      12.4'   # 6 extra spaties vooraan en afronding!
```

Vaststellingen:

- Het **decimaal teken (.) neemt ook een plaats in**: bv. `12.345` neemt 6 plaatsen in.
- Er wordt **afgerond** volgens de gekende afrondingsregels.
- Er wordt **vooraan aangevuld** met spaties.

6.6 Immutable types

String objecten zijn **immutable**. Dit wil zeggen dat een string object niet meer kan gewijzigd worden nadat het werd gecreëerd. Dit blijkt ook duidelijk uit de voorgaande sectie, waarin de methoden die inwerken op strings (zoals bv. `replace()` of `strip()`) het oorspronkelijke string object nooit zullen aanpassen, maar wel een (gewijzigde) kopie zullen retourneren. In het volgende hoofdstuk bespreken we gegevenstypes de wel mutable zijn, en wat de gevolgen daarvan zijn.

Het onderstaande voorbeeld illustreert dat men karakters niet kan wijzigen met indexering:

```
>>> s = "schoen"
>>> s[3] = "e"   # poging tot wijzigen van "o" door "e" met indexering
TypeError: 'str' object does not support item assignment
```

De string `s` wijzigen kan dus niet. Wat wel mogelijk is, is het volgende:

```
>>> s = "schoen"             # toekenningsstatement
>>> s = s[0:3] + "e" + s[4:] # de variabele s verwijst naar een NIEUW object
>>> print(s)
'scheen'
```

Deze manier van updaten van variabelen wordt hieronder gebruikt in een uitgebreider voorbeeld, waarin de `replace()` methode meermaals opgeroepen wordt om alle leestekens uit een string te verwijderen:

```
zin = "Hallo, allemaal! Is programmeren leuk???"
for x in "!,?;":   # x itereert over karakters in "!,?;"
    zin = zin.replace(x, "")   # vervang x door lege string en
                               # ken resultaat toe aan 'nieuwe' zin
print(zin)
```

De output van dit fragment is:

```
Hallo allemaal Is programmeren leuk
```

Let op het feit dat zin in de `for`-lus niet gewijzigd wordt, maar dat de `replace()` methode een **kopie** van de string retourneert waarin de leestekens vervangen zijn door een lege string. Deze kopie wordt dan aan de variabele `zin` toegekend. De variabele `zin` wordt dus steeds overschreven binnen de `for`-lus.

6.7 Gemengde opdrachten

Opdracht 6.10 (`timestamp.py`)

Schrijf een programma waarin je de volgende stappen uitvoert:

- Lees het tekstbestand `"datetime.txt"` in met daarin de regel:

  ```
  Date and time [DD/MM/YYYY hh:mm:ss]: 28/06/2015 15:27:56
  ```

  en plaats de inhoud daarvan in bv. een variabele `date_time_str` (gebruik hiervoor een geschikte functie uit de module `infoFun`).

- Schrijf een functie `get_timestamp()` dat een *string* neemt zoals hierboven (`date_time_str`) een zgn. timestamp van de vorm `YYYYMMDD_hhmmss` en retourneert.

Je programma geeft de volgende structuur:

```
import infoFunWP as infoFun

def get_timestamp(date_time_str):
    # implementeer zelf de functie
    return timestamp

# lees tekstbestand "datetime.txt" en plaats inhoud in date_time_str
date_time_str = ... # aan te vullen

# roep de fuctie get_timestamp() op
timestamp = get_timestamp(date_time_str)
# toon timestamp op scherm
print("Timestamp:",timestamp)
```

Een mogelijke output is:

```
Timestamp: 20150628_152756
```

Opdracht 6.11 (`lees_dna_seq.py`)

Een DNA-sequentie in een tekstbestand staat in vele gevallen op verschillende regels met een bepaalde (vaste) lengte. Elke regel wordt beëindigd door een `\n` (*end-of-line* 'karakter'). Bij het inlezen van zo'n DNA-sequentie wil je vermijden dat die *end-of-line* 'karakters' mee in de DNA string worden opgenomen. Schrijf daarom een functie `lees_dna_seq_uit_bestand()` die een DNA-sequentie als *string* uit een tekstbestand inleest en de `"\n"` verwijdert (vervangt door `""`, d.i. een lege *string*). De functie retourneert de DNA-sequentie (`dna_seq`) waarin alle `\n` verwijderd werden. Gebruik een functiedefinitie van de volgende vorm:

```
def lees_dna_seq_uit_bestand(bestandsnaam):
    # lees bestand in met stringRead() uit module infoFun
    # verwijder alle end-of-line karaketers "\n" uit de tekst
    # retourneer restultaat
    return dna_seq
```

```
>>> dna_seq = lees_dna_seq_uit_bestand("E_coli_Gene_ddp3_2089-2511.txt")
>>> print(dna_seq)
ATGACGTCTCTGACGCCATTCCGCAAC... GTATGA
```

Opdracht 6.12 (`toon_codons_tabel.py`)

Schrijf een functie `toon_codons()` die een DNA-sequentie als *string* aanvaardt en de codons (3 opeenvolgende basen) in een tabel weergeeft naar het scherm in 5 kolommen. Gebruik een functiedefinitie van de volgende vorm:

```
def toon_codons(dna_seq):
    # deze functie toont gegevens op het scherm
    # maar retourneert niets
```

Mogelijke output is:

```
>>> dna = "ATGACGTCTCTGACGCCATTCCGCAACGTTTGCCCGATGACCTAC"
>>> toon_codons(dna)
[  1] ATG   [  2] ACG   [  3] TCT   [  4] CTG   [  5] ACG
[  6] CCA   [  7] TTC   [  8] CGC   [  9] AAC   [ 10] GTT
[ 11] TGC   [ 12] CCG   [ 13] ATG   [ 14] ACC   [ 15] TAC
```

Tips

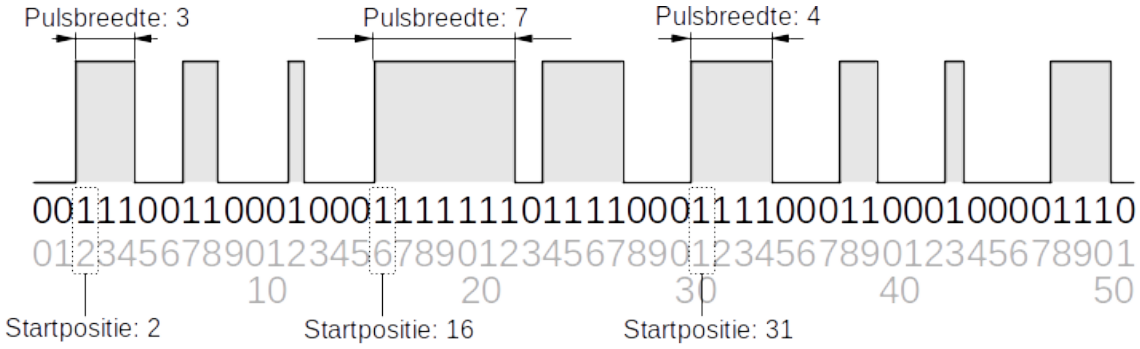
- Gebruik de `end` optie van de functie **`print()`** output op eenzelfde regel te tonen.
- Gebruik f-string om de uitlijning (in het bijzonder voor de teller) te verzorgen.

Uitbreiding: indien je deze opracht combineert met Opdracht 6.12 kan je het bestand `E_coli_Gene_ddp3_2089-2511.txt` inlezen als een string (zonder `"\n"`) en vervolgens de codons afzonderlijk tonen. De output is dan:

```
[  1] ATG   [  2] ACG   [  3] TCT   [  4] CTG   [  5] ACG
[  6] CCA   [  7] TTC   [  8] CGC   [  9] AAC   [ 10] GTT
...
[136] GAA   [137] AAT   [138] TTA   [139] TTA   [140] GTA
[141] TGA
```

Opdracht 6.13 (`signaalverwerking.py`)

In bepaalde signaalverwerkingsprocessen is het nodig om de startposities en breedtes te bepalen van elektrische pulsen. In deze opdracht bestaan signalen uit `'0'`-en en `'1'`-en. Eén of meerdere opeenvolgende `'1'`-en vormen samen een puls. Een puls begint bij een bepaalde startpositie (index) en heeft een zekere pulsbreedte (het aantal opeenvolgende `'1'`-en). In het onderstaande voorbeeld kan je 9 pulsen waarnemen, de eerste puls begint bij positie 2 en heeft pulsbreedte 3, de tweede puls begint bij positie 7 en heeft pulsbreedte 2, enz.



Een signaal stellen we voor door een *string* die bestaat uit de karakters `'0'` en `'1'`. Het signaal in het bovenstaande voorbeeld wordt dus:

```
signaal = "00111001100010001111111011110001111000110001000011 10"
```

Opmerking: je mag ervan uitgaan dat het eerste en het laatste karakter van een signaal steeds een `'0'` is.

1. Schrijf een script dat herhaaldelijk aan de gebruiker vraagt om een signaal in te gegeven tot een *lege string* ingegeven wordt. Het script moet tellen hoeveel pulsen aanwezig zijn in het signaal en dit aantal naar het scherm printen.

```
Geef een signaal van 1-en en 0-en:
011100
Aantal pulsen: 1

Geef een signaal van 1-en en 0-en:
01110011000
Aantal pulsen: 2

Geef een signaal van 1-en en 0-en:
00111001100010001111111011110001111000110001000011 10
Aantal pulsen: 9

Geef een signaal van 1-en en 0-en:

Programma stopt.
```

2. Implementeer de functie `geefEigenschappen()` die als argument een signaal (type **`str`**) aanvaardt en waarbij voor elke puls de volgende informatie op het scherm wordt geprint:

   - De pulspositie is de index van de eerste `1` van de puls.
   - De pulsbreedte is het aantal (opeenvolgende) karakters `'1'`.

   Deze functie retourneert niets.

```
>>> geefEigenschappen("011100")
Puls 1: startindex = 1, breedte = 3

>>> geefEigenschappen("01110011000")
Puls 1: startindex = 1, breedte = 3
Puls 2: startindex = 6, breedte = 2

>>> geefEigenschappen("01110011000100011110")
Puls 1: startindex = 1, breedte = 3
Puls 2: startindex = 6, breedte = 2
Puls 3: startindex = 11, breedte = 1
Puls 4: startindex = 15, breedte = 4
```

Opdracht 6.14 (`diana.py`)

Encryptie[8] is het coderen of versleutelen van gegevens (vaak tekst) o.b.v. een bepaald algoritme. Het Diana Cryptosysteem is zo een versleutelingstechniek, die in theorie onkraakbaar is. Deze techniek maakt gebruik van twee principes. De eerste is een **trigraph**. Dit is een eenvoudige manier om o.b.v. twee letters een derde letter te vormen, o.b.v. hun plaats in het alfabet. `A` heeft plaats 0, `B` heeft plaats 1, enzoverder. Stel dat men twee letters heeft met plaatsen p_1 en p_2, men kan dan p_3 als volgt berekenen:

$$p_3 = (25 - p_1 - p_2) \text{ modulus } 26\,.$$

Voor de letters `C` (dus $p_1 = 2$) en `F` (dus $p_2 = 5$) krijgt men dat $p_3 = 18$ wat overeenkomt met de letter `S`. Merk op dat enkel hoofdletters gebruikt worden. Indien men een tekst wil coderen, mag deze enkel uit hoofdletters bestaan (geen kleine letters, spaties, punten, komma's, cijfers, …). Indien men een tekst krijgt, moeten dus eerst alle niet-letters verwijderd worden. De overige letters moeten omgezet worden in hoofdletters zodat één lange string van hoofdletters ontstaat.

1. Schrijf een functie `trigraph()` die de omzetting die hierboven geïllustreerd wordt implementeert. Deze functie moet als argumenten twee karakters aanvaarden (hoofdletters) en retourneert het karakter dat het resultaat is van de het trigraph-omzettingsprincipe.
2. Schrijf een functie `parseTekst()` die een string aanvaardt als argument en een nieuwe string retourneert waarin alle letters in de inputstring omgezet zijn naar hoofdletters en waaruit alle niet-letters verwijderd zijn.

[8] Idee ontleend aan een oefening van Prof. P. Dawyndt (Programmeren, Faculteit Wetenschappen, UGent).

```
>>> trigraph('C', 'F')
'S'
>>> trigraph('G', 'X')
'W'
>>> parseTekst("Het is mooi weer!")
HETISMOOIWEER
```

Tip: om de positie van een letter te vinden kan je eventueel gebruikmaken van de string `alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"`.

Het tweede principe van de Diana codering maakt gebruik van een **one-time-pad** (een specifiek soort sleutel) en een index. Het one-time-pad is een (gegeven) string van willekeurige hoofdletters. Om een bericht te coderen, plaatst men het onder het one-time-pad startend vanaf de gegeven **index**. Letters die boven elkaar staan vormen koppels die dan gebruikt worden als input voor de trigraph. De letters die daarvan het resultaat zijn vormen samen het gecodeerde bericht.

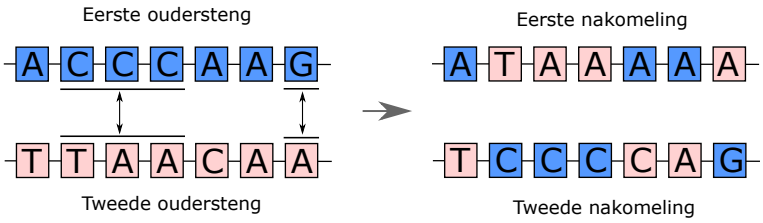
```
index (= 6):    0123456 78...
one-time-pad:   AZZRIRNVHDHGIGODSOZJRHJCHURI...
bericht:              HETISMOOIWEER
                ------------------------------------
gecodeerd:            FAZOAHDFDADHJ
```

3. Schrijf een functie `diana()` die met de parameters `bericht` (het bericht dat moet gecodeerd worden), `onetimepad` (het one-time-pad dat zal gebruikt worden, een voorbeeld one-time-pad is beschikbaar in `otp.txt`) en `index` (zie hierboven).

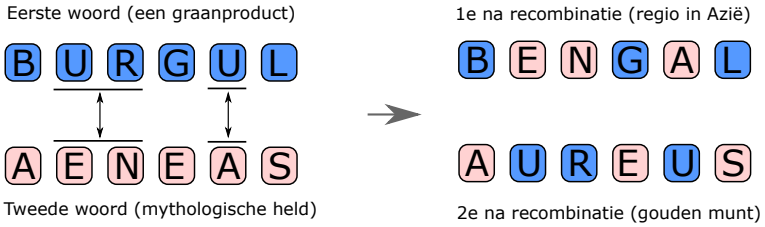
```
>>> import infoFunWP as infoFun
>>> onetimepath = infoFun.stringRead("otp.txt")
>>> bericht = "Het is mooi weer!"
>>> diana(bericht, onetimepath, 6)
'FAZOAHDFDADHJ'
```

Opdracht 6.15 (`recombinatie.py`)

Met recombinatie wordt in de genetica de herschikking van de genetische eigenschappen van een individu aangeduid. Een gevolg hiervan is dat het nageslacht een andere combinatie van genen heeft dan elk van beide ouder-individuen. Vaak is dit een gevolg van (meervoudige) crossing-over tijdens de meiose. Tijdens deze recombinatiefase worden nucleotiden uitgewisseld tussen twee DNA ouder-strengen, om zo twee nieuwe strengen te vormen (zie voorbeeld hieronder, waar de 2e, 3e, 4e en 7e nucleotide worden uitgewisseld).



Het principe van recombinatie komt men ook tegen in de natuurlijke talen. De figuur hieronder toont een voorbeeld waarbij in twee Engelse woorden (BURGUL en AENEAS) de karakters op posities 2, 3 en 5 worden uitgewisseld om twee nieuwe woorden te vormen (BENGAL en AUREUS).

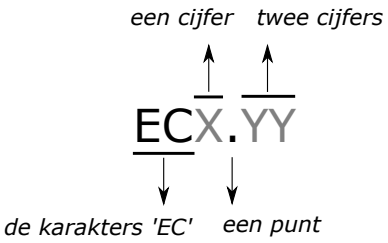


Schrijf een functie `isRecombinatie()` waaraan drie woorden (strings) moeten doorgegeven worden als argumenten. Als het derde woord kan ontstaan zijn na een recombinatie van de eerste twee woorden moet de functie `True` teruggeven, indien het derde woord niet kan ontstaan zijn na een recombinatie van de eerste twee woorden moet de functie `False` teruggeven. **Opmerking**: indien een van de drie woorden een verschillende lengte heeft, kunnen deze woorden nooit leiden tot een geldige recombinatie en moet de functie ook `False` teruggeven.

```
>>> isRecombinatie("burgul", "aeneas", "bengal")  # wel recombinatie
True

>>> isRecombinatie("hans", "piet", "paul")        # geen recombinatie
False

>>> isRecombinatie("ACCA", "CTTG", "ATT")         # derde woord korter
False
```

Opdracht 6.16 (`ECnummer.py`)

Het EC-nummer (enzymcommissienummer) is een numeriek classificatiesysteem voor enzymen dat aangeeft welke reactie een enzym katalyseert. Een EC-nummer bestaat uit de karakters `EC` gevolgd door twee, door een punt gescheiden, gehele getallen. De structuur van een EC-nummer wordt hieronder schematisch voorgesteld.



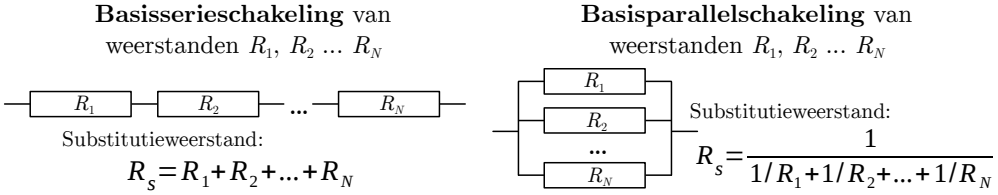
De nummers `EC3.05` en `EC2.13` zijn bijvoorbeeld geldige EC-nummers.

Schrijf een script dat een gebruiker herhaaldelijk vraagt naar een EC-nummer en controleert of het ingegeven nummer een geldig EC-nummer is. Het programma stopt indien `"stop"` (in alle mogelijke schrijfwijzen) ingegeven wordt.

```
Geef een EC-nummer: EC2.13
EC2.13 is een geldig EC-nummer.

Geef een EC-nummer: EC3.05
EC3.05 is een geldig EC-nummer.

Geef een EC-nummer: EC1.1          # laatste getal bevat maar 1 cijfer
EC1.1 is GEEN geldig EC-nummer.

Geef een EC-nummer: EC2.1x         # laatste getal bevat karakter 'x'
EC2.1x is GEEN geldig EC-nummer.

Geef een EC-nummer: 3.05           # EC ontbreekt
3.05 is GEEN geldig EC-nummer.

Geef een EC-nummer: stop
Programma stopt.
```

Opdracht 6.17 (`schakeling.py`)

In een elektrische schakeling kunnen weerstanden in serie en/of parallel geschakeld worden. Voor een dergelijke schakeling kan men de substitutieweerstand R_S berekenen. Een schakeling waarin twee of meerdere weerstanden louter in serie of parallel geschakeld zijn, noemen we basisschakelingen.



We stellen basisserie- en parallelschakelingen voor door waarden van de weerstanden te scheiden door `+` (plusteken) voor serieschakelingen en `//` (dubbele slash) voor parallelschakelingen. Stringvoorstellingen van basisschakelingen met bv. de weerstanden 6.2 Ω, 9.1 Ω en 3.3 Ω worden voorgesteld als:

```
bws = "6.2"            # basis-weerstand (eenvoudigste basisschakeling)
bss = "6.2+9.1+3.3"    # basis-serieschakeling van 3 weerstanden
bps = "6.2//9.1//3.3"  # basis-parallelschakeling van 3 weerstanden
```

In **deze opdracht** gaan we ervan uit dat **enkel geldige** basisschakelingen of weerstanden gebruikt worden (dit hoef je dus **niet te controleren**). Dit wordt hieronder geïllustreerd.

Geldige basisschakelingen:

```
"1.12+2.22"
"1+2+3"
"12.5"
"10.1//5.3//4.3"
```

Ongeldige basisschakelingen (komen niet voor):

```
"6.2+2.1//2.3" # zowel + als //
"1.6+1.1+"     # + op einde
"1.a+1.2"      # ongeldig karakter
"1.1++1.1"     # 2 keer + na elkaar
```

Vul het bestand `schakeling.py` aan:

1. Schrijf een script waarin de gebruiker meermaals gevraagd wordt een geldige basisschakeling of weerstand in te geven. Het programma gaat na of de ingevoerde string een **basisweerstand**, een **basisserieschakeling** of een **basisparallelschakeling** is en toont dit op het scherm. Het programma kan beëindigd worden door `"stop"` in te voeren.

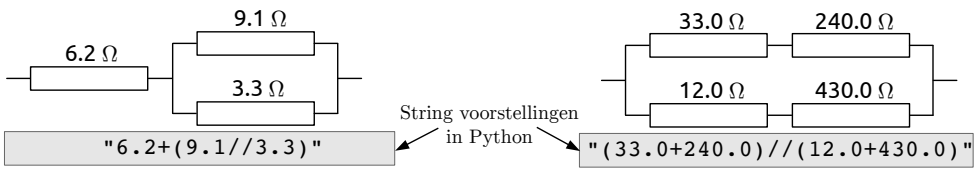
```
>>> Geef een basis-weerstand of schakeling in: 12.1
Dit is een basis-weerstand!

Geef een basis-weerstand of schakeling in: 1.2+1.3+5.1
Dit is een basis-serieschakeling

Geef een basis-weerstand of schakeling in: 1.2//1.3//5.1
Dit is een basis-parallelschakeling

Geef een basis-weerstand of schakeling in: 1.2+1.3
Dit is een basis-serieschakeling

Geef een basis-weerstand of schakeling in: STOP
Programma wordt beeindigd!
```

2. Implementeer een functie `subweerstandBasis()` die een voorstelling (type *string*) van een basisweerstand, basisserie- of parallelschakeling (met een willekeurig aantal weerstandswaarden) als argument aanvaardt en de substitutieweerstand R_s (type *float*) retourneert.

```
>>> subweerstandBasis("6.2+9.1")
15.3                          # is som van 6.2 en 9.1

>>> subweerstandBasis("6.2//9.1//3.3//7.5")
1.4133395275963525            # = 1 / (1/6.2 + 1/9.1 + 1/3.3 + 1/7.5)

>>> subweerstandBasis("620.0")
620.0                         # waarde wordt als float geretourneerd
```

   Indien de stringvoorstelling slechts één weerstandswaarde voorstelt (laatste voorbeeld) moet de *float* versie ervan geretourneerd worden.

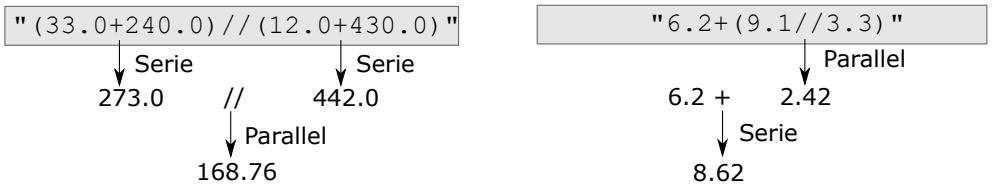
We breiden de mogelijke schakelingen nu uit met **gemengde schakelingen**, waarbij ronde haakjes basisschakelingen van elkaar scheiden zoals hieronder getoond wordt. Merk op dat haakjes daarbij niet genest voorkomen (dus geen haakjes binnen haakjes).



Hieronder volgen nog enkele andere voorbeelden van dergelijke schakelingen:

```
(1.2+1.3+5.1)//(6.3//5.7)
(1.3+1.1)//1.2
(1//2)+(77+112)
```

Door de basisschakelingen die tussen haakjes staan te vervangen door hun substitutieweerstand worden de gemengde schakelingen herleid tot basisschakelingen waarvan men dan finaal de weerstand kan berekenen zoals hieronder getoond:



3. (*) Implementeer een functie `subweerstand()` die als argument een voorstelling (type *string*) van een gemengde schakeling aanvaardt en de substitutieweerstand (type *float*) retourneert.

```
>>> subweerstand("(33.0+240.0)//(12.0+430.0)")
168.76
>>> subweerstand("6.2+(9.1//3.3)")
8.62
```

7

Lijsten

7.1 Container datatypes

Een algemeen kenmerk van containers (objecten van een containerdatatype) is dat ze conceptueel andere objecten *bevatten*. Een container kan men dus beschouwen als een collectie van objecten. Deze objecten noemen we de **elementen** (of items) van de container. Container datatypes verschillen in de manier waarop ze de elementen die ze bevatten structureren (bijvoorbeeld al dan niet geordend), of ze toelaten om bijkomende elementen toe te voegen, het type indexering dat ze toelaten, enz. De vier meest courante container-gegevenstypes in Python zijn:

- `list`: een collectie die **geordend** en **wijzigbaar** (*mutable*) is. Toegang tot de elementen gebeurt o.b.v. een index.
- `tuple`: een collectie die **geordend** en **niet wijzigbaar** (*immutable*) is. Toegang tot de elementen gebeurt o.b.v. een index.
- `dict`: een collectie van *key-value* paren die **niet geordend** maar wel **wijzigbaar** is. Toegang tot de elementen gebeurt o.b.v. een *key*.
- `set`: een collectie die **niet geordend** maar **wel wijzigbaar** is.

In deze cursus wordt in de eerste plaats aandacht besteed aan de container-types `list` (dit hoofdstuk) en `dict` (Hoofdstuk 10).

7.2 Het gegevenstype `list`

Een list kan gecreëerd worden door meerdere variabelen of literals, gescheiden door komma's, te omgeven door vierkante haakjes (`[]`). Dit wordt geïllustreerd in het onderstaande voorbeeld:

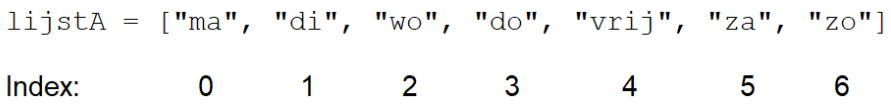
```
>>> lijstA = ["ma", "di", "wo", "do", "vrij", "za", "zo"]
```

```
>>> type(lijstA)
list
>>> lijstA
['ma', 'di', 'wo', 'do', 'vrij', 'za', 'zo']
```

In het bovenstaande voorbeeld zijn de elementen van lijst `lijstA` (verwijzingen naar) strings. De elementen van een lijst kunnen echter van eender welk gegevenstype zijn:

```
>>> lijstB = [12.45, 7.12, 78.1]    # lijst van floats
>>> lijstC = ["WO1", 1914, 9.81]    # elementen van verschillende types
```

De elementen in een lijst zijn geordend. Elk element heeft bijgevolg een **index** die kan gebruikt worden om het element te benaderen. Figuur 7.1 illustreert de link tussen een element en zijn index. Merk op dat, naar analogie met string-indices, het **eerste element index** `0` heeft.



Figuur 7.1: Indices van de elementen in de lijst `lijstA = ["ma", "di", ..., "zo"]`.

Om de elementen te benaderen wordt gebruik gemaakt van **indexering** door middel van de **indexerings-operator** `[]`. Het gebruik van indexering wordt geïllustreerd in het onderstaande voorbeeld, waar het element met index `3` wordt opgevraagd.

```
>>> lijstA = ["ma", "di", "wo", "do", "vrij", "za", "zo"]
>>> dag = lijstA[3]

>>> print(dag)
do                      # dag verwijst naar de string "do"
>>> type(dag)
str                     # het type van dag is str
```

Indexering - voor index binnen bereik van lijst.
Beschouw een `list` object `L` en een geheel getal (`int`) `i`. De expressie

`L[i]`

retourneert:

- **indien** $0 \leq$ `i` $<$ `len(s)` $\rightarrow$ een verwijzing naar het element met index `i`
- **indien** $-$`len(s)` $\leq$ `i` $\leq -1 \rightarrow$ een verwijzing naar het element met index `i + len(s)`
- **in alle andere gevallen** $\rightarrow$ een `IndexError`

In het onderstaande codefragment wordt indexering toegepast om een cijfer te linken aan een dag van de week. De gebruiker wordt gevraagd om een cijfer (1–7) in te voeren, en vervolgens wordt op het scherm getoond met welke dag van de week dit cijfer overeenstemt:

```
dagnummer = int(input("Geef een dagnummer in: "))
dagen = ["ma", "di", "wo", "do", "vrij", "za", "zo"]
dagnaam = dagen[dagnummer-1]
print("De", dagnummer, "e dag is", dagnaam)
```

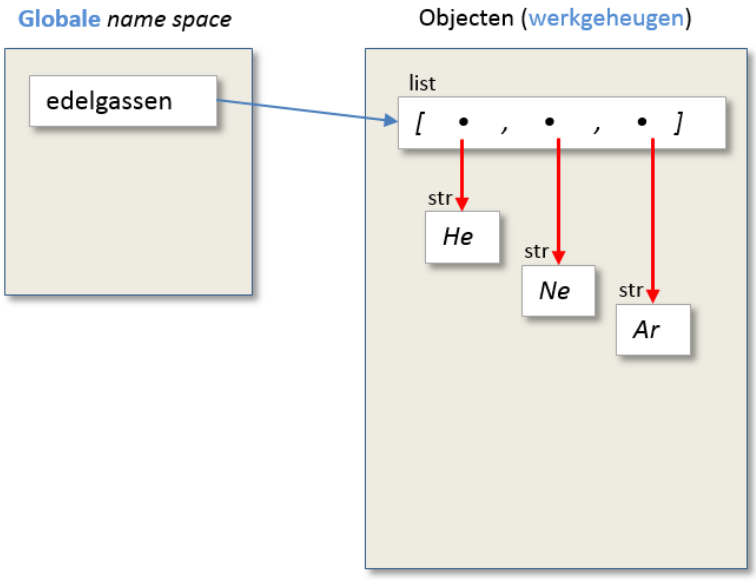
Mogelijke input/output is:

```
Geef een dagnummer in: 5
De 5 e dag is vrij
```

7.2.0.1 Het toestandsdiagram voor lijsten

In het werkgeheugen is de inhoud van een lijst een *geordende collectie van referenties naar objecten*. Het is dus belangrijk om in te zien dat **een `list` object geen andere objecten bevat, maar wel verwijzingen naar andere objecten**. Figuur 7.2 toont het toestandsdiagram nadat de lijst `edelgassen` werd gecreëerd:

```
>>> edelgassen = ["He", "Ne", "Ar"]
```

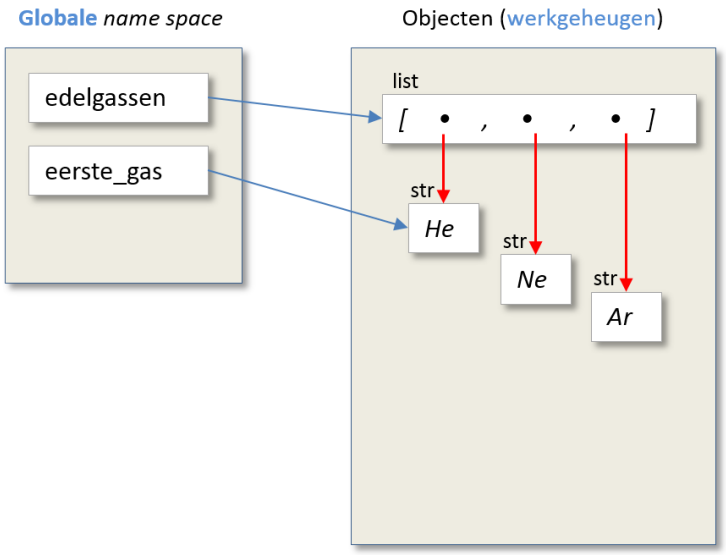


Figuur 7.2: Toestandsdiagram na uitvoeren van het toekenningsstatement `edelgassen = ["He", "Ne", "Ar"]`.

Door gebruik te maken van indexering, kan een element van een lijst ook toegekend worden aan een nieuwe variabele. In de onderstaande instructie wordt een nieuwe variabele `eerste_gas` gecreëerd die verwijst naar het element met index 0 in `edelgassen`:

```
>>> eerste_gas = edelgassen[0]
```

Figuur 7.3 toont het toestandsdiagram na het uitvoeren van dit toekenningsstatement. Merk op dat zowel vanuit de lijst als door de nieuwe variabele verwezen wordt naar de string `"He"`.



Figuur 7.3: Toestandsdiagram na uitvoeren van het toekenningsstatement `eerste_gas = edelgassen[0]`.

Opdracht 7.1 (`chinese_zodiak.py`)

De Chinese zodiak of dierenriem werkt met jaarlijkse tekens die gebaseerd zijn op een 12-jaren cyclus. Elk jaar in deze cyclus wordt voorgesteld door één van de volgende dieren:

0	1	2	3	4	5	6	7	8	9	10	11
aap	haan	hond	varken	rat	os	tijger	konijn	draak	slang	paard	schaap

Het jaartal modulo 12 (rest na deling door 12) bepaalt het zodiakteken (dier): bv. het jaartal 1900 komt overeen met *rat* omdat `1900 % 12 = 4` (cf. 4 is het getal tussen haakjes bij *rat*).
Schrijf een script dat de gebruiker vraagt om een jaartal in te voeren en vervolgens het overeenkomstige Chinese zodiakteken (diernaam) op het scherm weergeeft. Voer daarbij de volgende stappen uit:

- creëer een lijst `zodiak = ["aap", "haan", "hond", ...]`,
- vraag een jaartal aan de gebruiker,
- maak (onder andere) gebruik van indexering om een jaartal te linken aan een dier

Mogelijke input/output is:

```
Geef een jaartal: 1973
Het jaartal 1973 stemt overeen met os.
```

```
Geef een jaartal: 1980
Het jaartal 1980 stemt overeen met aap.
```

7.3 Een element wijzigen

In de voorgaande sectie werd reeds aangehaald dat, via indexering, de elementen van een lijst opgevraagd kunnen worden. Men kan, o.b.v. de index, ook elementen in een lijst wijzigen d.m.v. een toekenningsstatement. Merk daarbij op dat steeds de referentie zal gewijzigd worden. Lijsten zijn het eerste gegevenstype in deze cursus dat wijzigingen toelaat aan objecten nadat ze gecreëerd werden. Naar deze eigenschap wordt verwezen door de term **mutable** (of **wijzigbaar** in het Nederlands).

Hoe elementen kunnen gewijzigd worden o.b.v. hun index, wordt geïllustreerd in het onderstaande codefragment:

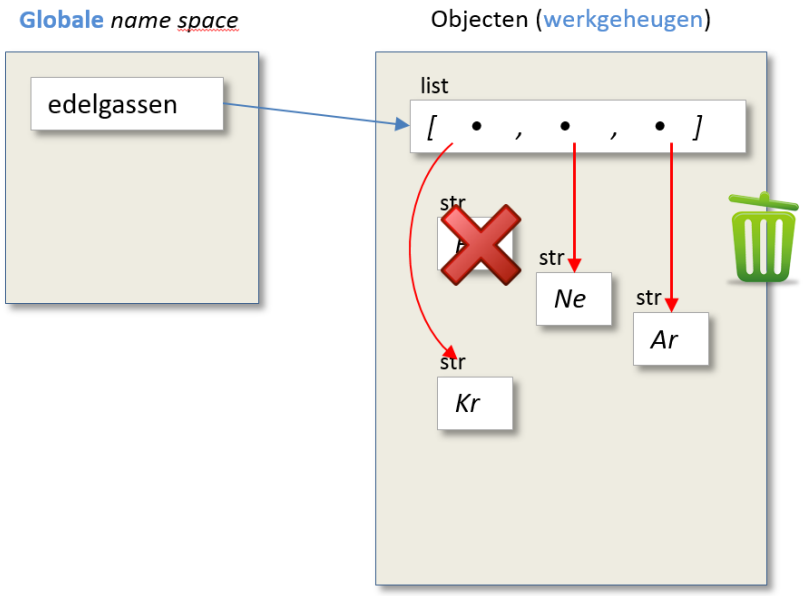
```
>>> edelgassen = ["He", "Ne", "Ar"]

>>> edelgassen[0] = "Kr"        # wijziging element met index 0
>>> edelgassen
['Kr', 'Ne', 'Ar']

>>> edelgassen[0]               # via indexering nieuw element opvragen
'Kr'
```

De invloed van deze wijziging op het toestandsdiagram wordt getoond in Figuur 7.4. Merk op dat op index 0 nu een referentie (of verwijzing) naar een nieuw object `"Kr"` aanwezig is. Omdat er geen enkele verwijzing meer aanwezig is naar `"He"` wordt dit object **verwijderd** uit het werkgeheugen.

In het voorgaande toekenningsstatement werd een referentie gecreëerd naar een nieuw object (voorgesteld door de literal `"Kr"`). Men kan echter ook een variabele gebruiken in het rechter lid van het toekenningsstatement. In dit geval zal het betreffende element gaan verwijzen naar hetzelfde object als die variabele. Dit wordt geïllustreerd in het onderstaande codefragment:

```
>>> edelgassen = ["He", "Ne", "Ar"]
>>> radon = "Rn"

>>> edelgassen[1] = radon
>>> edelgassen
['He', 'Rn', 'Ar']

>>> id(radon) == id(edelgassen[1])
True                        # verwijzen naar zelfde object
```

Algemeen is de syntax om een element te wijzigen o.b.v. zijn index de volgende:



Figuur 7.4: Toestandsdiagram na uitvoeren van het toekenningsstatement `edelgassen[0] = "Kr"`

Wijzigen een element in een lijst o.b.v. de index

Beschouw een `list` object `L`, een geheel getal (`int`) `i` en het toekenningsstatement

`L[i] = w`

- Indien *w* een literal is (bv. `"hallo"` of `1.23`) zal `L[i]` verwijzen naar het (nieuwe) object met waarde *w*.
- Indien *w* een variabele is zal `L[i]` verwijzen naar het object waar ook *w* naar verwijst.

Opdracht 7.2 (`namenlijst_wijzigen.py`)

Het bestand `species.txt` bevat de wetenschappelijke naam van een aantal bacteriën (onder andere `Escherichia coli`). Elke naam staat op een afzonderlijke regel en heeft de volgende vorm:

Genus species

1. Schrijf een script waarin:
   - Het bestand `species.txt` wordt ingelezen met de functie `listRead()` uit de module `infoFun`. Bekijk dit bestand eerst in een tekstverwerker.
   - De resulterende lijst met soortnamen wordt gewijzigd zodat de nieuwe elementen **enkel nog de genusnaam** bevatten: de spatie en de *species* dien je dus te verwijderen.
   - Het resultaat op het scherm getoond wordt (kan eenvoudig met `print()`)

   Het resultaat is:

```
['Bacillus', 'Pseudomonas', 'Escherichia', 'Actinobacter', 'Moraxella',
 'Alkanindiges', 'Perlucidibaca']
```

2. Breid het script uit als volgt:
   - Vraag aan de gebruiker de naam van een genus (uit de lijst).
   - Vraag aan de gebruiker de naam waardoor dit genus vervangen moet worden.
   - Zorg ervoor dat vervolgens het ingegeven genus wordt gezocht in de lijst en wordt vervangen door het nieuwe.
   - Toon de gewijzigde lijst op het scherm.

   Mogelijke input/output is:

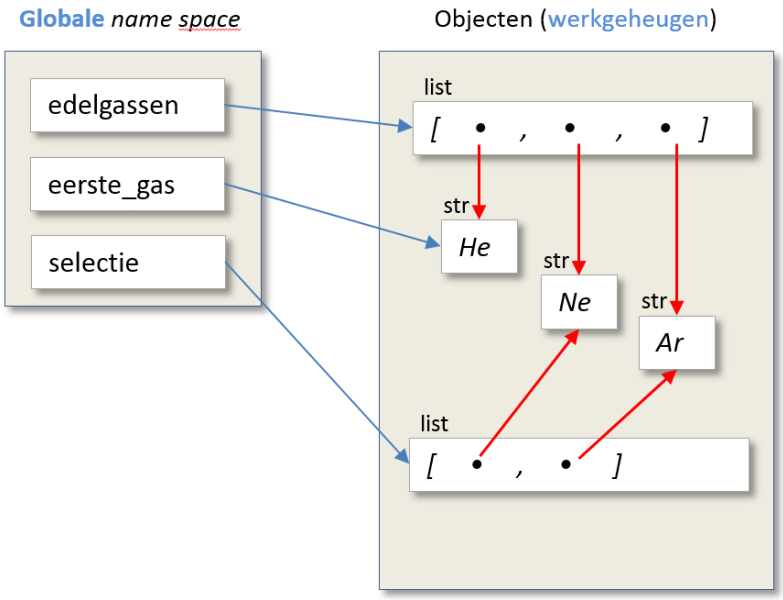
```
Geef de naam van een genus: Moraxella
Geef een nieuwe naam: Listonella

Aangepaste lijst:
['Bacillus', 'Pseudomonas', 'Escherichia', 'Actinobacter', 'Listonella',
'Alkanindiges', 'Perlucidibaca']
```

7.4 List slicing

Wanneer een lijst wordt geïndexeerd met een gegeven index, dan is het resultaat een referentie naar het object met die index. Naar analogie met het slicen van strings (zie Sectie 6.2.2), kan men ook **list-slicing** toepassen. Het resultaat van list-slicing is steeds een nieuwe lijst. Merk hierbij op dat een lijst een opeenvolging van referenties is: **de list-slice kopieert** dus deze **referenties**, **en niet de objecten**. Dit type van kopie noemt men daarom een ***shallow-copy***. De volgende sessie illustreert het gebruik van list-slicing:

```
>>> edelgassen = ["He", "Ne", "Ar"]

>>> eerste_gas = edelgassen[0]
>>> eerste_gas
'He'

>>> selectie = edelgassen[1:3]       # slice bevat indices 1 en 2
>>> selectie
['Ne', 'Ar', 'Kr']
```

Figuur 7.5 toont het toestandsdiagram na het uitvoeren van de slicing instructie. Merk op dat een aantal referenties worden gekopieerd, maar dat geen string-objecten worden gekopieerd.



Figuur 7.5: Toestandsdiagram na uitvoeren van het toekenningsstatement `selectie = edelgassen[1:3]`.

List-slicing kan als volgt worden samengevat.

Slicing - voor indices binnen bereik van de list
Beschouw een `list` object `L` en twee gehele getallen (`int`) `i` (startindex) en `j` (eindindex).
De expressie

`L[i:j]`

retourneert

- indien `i` $<$ `j` $\rightarrow$ een nieuwe lijst met daarin **een kopie** van de referenties met indices `i`, `i+1`, ..., `j-1`,
- indien `i` $\geq$ `j` $\rightarrow$ een **lege list**

Merk op dat, aangezien het resultaat van een slicing expressie steeds een lijst is, je op dit resultaat onmiddellijk een nieuwe slicing operatie kan uitvoeren. Het resultaat van de onderstaande instructies is dan ook hetzelfde:

- Optie 1: `A = edelgassen[1:3]` en vervolgens `A[1]`.
- Optie 2: `edelgassen[1:3][1]`

```
>>> edelgassen = ["He", "Ne", "Ar"]
>>> A = edelgassen[1:3]
>>> A[1]
'Ar'
```

```
>>> edelgassen[1:3][1]
'Ar'
```

Slices van het type `L[i:]`, `L[:j]` en `L[i:j:k]`.
Naar analogie met strings, kunnen ook slices van de vorm `L[i:]`, `L[:j]` en `L[i:j:k]` uitgevoerd worden. Het gebruik van deze slicing types is volledig analoog aan het gebruik ervan bij strings (zie Sectie 6.2.2).

Opdracht 7.3 (`korte_instructies.py`)
Los volgende (korte) opdrachten op:

- Beschouw een lijst `L` die 100 elementen bevat. Creëer een lijst `A` die alle elementen uit `L` bevat met een *even index (0, 2, 4, ...) die kleiner is dan 50.* Vul aan:

```
>>> A = ................................................
```

- Wat is het resultaat van de volgende instructies? Vul aan:

```
>>> L = [10, 20, 30, 40, 50, 60]
>>> L[::2][1]
.............................
```

- Wat is het resultaat van de volgende instructies. Vul aan:

```
>>> L = ["maandag", "dinsdag", "woensdag"]
>>> L[1][:4]
.............................
```

De copy slice `L[:]` en aliasing
Een bijzonder type van slice is de **copy slice** `A = L[:]`. Merk op dat `A` een slice is en dus een kopie is van `L`. Het resultaat is bijgevolg een nieuwe lijst. Na het uitvoeren van het toekenningsstatement `B = L` daarentegen, zullen `B` en `L` naar dezelfde lijst verwijzen, een principe dat men **aliasing** noemt.

Het onderscheid wordt geïllustreerd in het onderstaande codefragment:

```
>>> edelgassen = ["He", "Ne", "Ar"]

>>> edelgassen_kopie = edelgassen[:]   # copy slice
>>> edelgassen_alias = edelgassen      # aliasing

>>> edelgassen_kopie
['He', 'Ne', 'Ar']
>>> edelgassen_alias
['He', 'Ne', 'Ar']
```

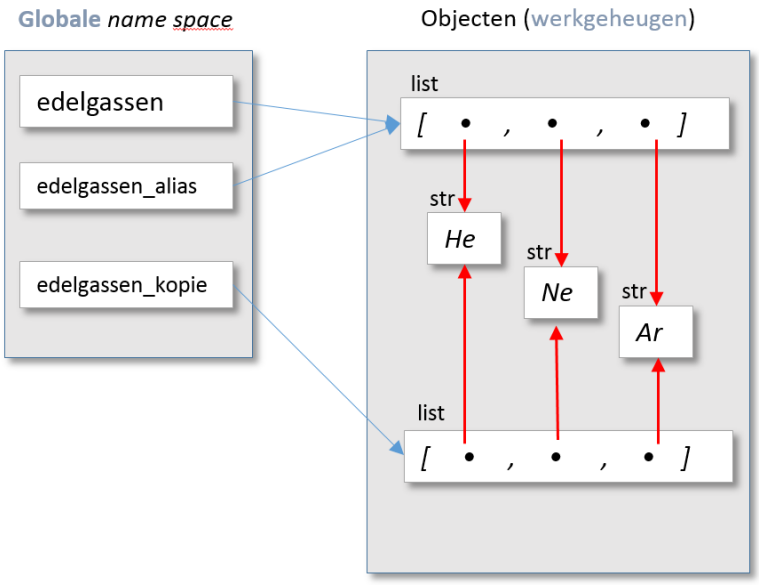
```
>>> id(edelgassen) == id(edelgassen_kopie)
False                                   # verschillende id (nieuw object)

>>> id(edelgassen) == id(edelgassen_alias)
True                                    # zelfde id (zelfde object)
```

Zoals blijkt uit het bovenstaande voorbeeld, is de waarde van de kopie-slice en de alias schijnbaar dezelfde. Merk echter op dat het gaat om **verschillende** objecten (zie ook Figuur 7.6).

Het verschil tussen een shallow copy (copy slice) en aliasing wordt belangrijk wanneer een lijst gewijzigd wordt.

```
>>> edelgassen[0] = "Kr"

>>> edelgassen_kopie
['He', 'Ne', 'Ar']            # copy slice blijft ONGEWIJZIGD

>>> edelgassen_alias
['Kr', 'Ne', 'Ar']            # alias WIJZIGT OOK!
```

Het principe dat hierboven geïllustreerd wordt, is één van de belangrijkste redenen waarom men onderscheid maakt tussen gegevenstypes die **mutable** (bv. `list`) zijn en gegevenstypes die **immutable** zijn (bv. `str`, `int`, `float` en `bool`). Het type `list` is het eerste voorbeeld in deze cursus van een mutable gegevenstype. Objecten van dit type kunnen gewijzigd worden nadat ze werden aangemaakt. Wanneer meerdere variabelen verwijzen naar een object van een mutable type kan dit object dus op verschillende manieren benaderd worden en worden aangepast.



Figuur 7.6: Toestandsdiagram na uitvoeren van het toekenningsstatement `edelgassen[0] = "Kr"`.

Opdracht 7.4 (`wijzig_lokale_maxima.py`)
In de volgende rij van getallen

$$1, \quad 2, \quad \mathbf{5}, \quad 3, \quad 3, \quad 2, \quad 3, \quad 4, \quad 5, \quad \mathbf{6}, \quad 5, \quad 3, \quad 2$$

noemen we 5 en 6 lokale maxima omdat voor beide elementen geldt dat ze groter zijn dan hun linker en rechter buur. Schrijf een script waarin:

- een lijst van getallen wordt aangemaakt (met waarden naar keuze), en
- vervolgens alle lokale maxima in deze lijst automatisch vervangen worden door 0.

```
L = [1, 2, 5, 3, 3, 2, 3, 4, 5, 6, 5, 3, 2]
# vul aan met de nodige instructies
print(L)
```

Je script heeft als output:

```
[1, 2, 0, 3, 3, 2, 3, 4, 5, 0, 5, 3, 2]
```

7.5 Het `del` statement

Het **`del`**-statement kan gebruikt worden om een element uit een lijst te verwijderen. Het verwijderen van elementen gebeurt steeds o.b.v. een index. Merk op dat het **`del`**-statement ook kan gebruikt worden om een variabele uit de *namespace* te verwijderen.

Het `del` statement - verwijderen elementen uit lijst
Beschouw een **`list`** object `L` en een geheel getallen (**`int`**) `i`. Het statement

```
del L[i]
```

verwijdert het element met index `i` uit `L` als $0 \leq$ `i` $\leq$ **`len`**`(L)-1`.

Merk op dat

- men ook meerdere elementen kan verwijderen door gebruik te maken van slicing: zo zullen na het uitvoeren van **`del`** `L[1::2]` alle elementen met een oneven index uit `L` verwijderd zijn.
- negatieve indices ook toegelaten zijn: zo zal **`del`** `L[-1]` het laatste element uit `L` verwijderen

Enkele voorbeelden:

```
>>> edelgassen = ["He", "Ne", "Ar"]
>>> del edelgassen[1]   # verwijder element met index 1

>>> edelgassen
['He', 'Ar']
```

Merk op dat het `del`-statement ook kan gebruikt worden om **variabelen te verwijderen** uit een namespace.

Het `del` statement - verwijderen variabelen uit namespace
Beschouw een variabele `a`. Het statement

`del a`

verwijdert `a` uit de namespace.

Voorbeelden:

```
>>> edelgassen = ["He", "Ne", "Ar"]
>>> del edelgassen     # verwijder variabele
>>> edelgassen
NameError: name 'edelgassen' is not defined

>>> edelgassen = ["He", "Ne", "Ar"]
>>> del edelgassen[:]  # verwijder alle elementen
>>> edelgassen
[]        # lege lijst
```

Merk op dat de statements `del edelgassen` en `del edelgassen[:]` **niet** hetzelfde resultaat opleveren: het eerste verwijdert de **variabele**, het tweede verwijdert **alle elementen** uit de beschouwde lijst.

7.6 Operatoren voor lijsten

Bepaalde operatoren kunnen inwerken op operanden van het type `list`. De belangrijkste operatoren zijn `+` (concatenatie), `*` (multiplicatie), de `in` operator en de relationele operatoren. In de volgende subsecties wordt het gebruik van deze operatoren geïllustreerd.

7.6.1 De operatoren `+` en `*`

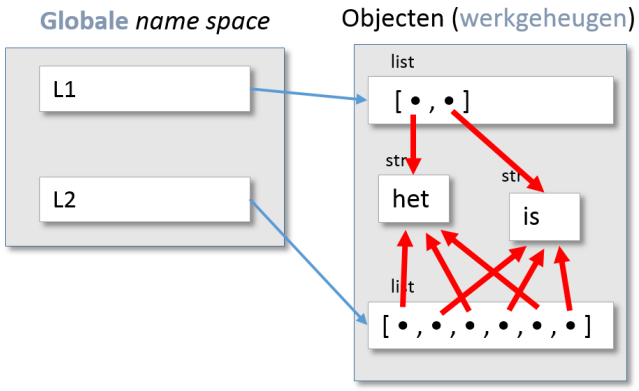
Analoog aan de functie die de operatoren `+` en `*` hebben bij strings, zal `+` twee operanden van het type `list` **concateneren** en zal `*` een lijst **multipliceren**. Merk op dat bewerkingen met deze operatoren steeds een **nieuwe lijst** zullen retourneren, maar de operanden zelf niet wijzigen. Dit wordt geïllustreerd in de onderstaande sessie voor de concatenatie-operator `+`:

```
>>> L1 = ["het", "is"]
>>> L2 = ["mooi", "weer"]

>>> L3 = L1 + L2              # concatenatie
>>> L3
["het", "is", "mooi", "weer"]
```

De onderstaande sessie toont het gebruik van de **duplicatie**-operator `*` voor lijsten:

```
>>> L1 = ["het", "is"]
>>> L2 = L1 * 3               # multiplicatie
>>> L3
["het", "is", "het", "is", "het", "is"]
```

Merk op dat de multiplicatie-operator en de concatenatie-operator **referenties kopiëren**, en niet de objecten zelf. Het toestandsdiagram na het uitvoeren van de multiplicatie in het bovenstaande codefragment wordt getoond in Figuur 7.7.



Figuur 7.7: Toestandsdiagrammen na uitvoeren van `L2 = L1 * 3`.

7.6.2 De `in` operator

De `in` operator kan gebruikt worden om na te gaan of een bepaalde waarde kan teruggevonden worden in een lijst.

De `in` operator

Beschouw een variabele `a` en een lijst `L` met elementen `elem_0`, `elem_1`, `elem_2`, …, `elem_n`.

De logische expressie:

```
a in L
```

retourneert

- `True` als en slechts als **minstens één** van de expressies `a == elem_0`, `a == elem_1`, …,

`a == elem_n` als waarde `True` oplevert.

- `False` indien dit niet het geval is (en dus geen enkele expressie `True` oplevert).

Het gebruik van deze operator wordt hieronder geïllustreerd:

```
>>> "ACG" in ["TCT", "ACG", "ATG", "CTG"]
True         # element met index 1 is "ACG"

>>> "AGG" in ["TCT", "ACG", "ATG", "CTG"]
False        # de string 'AGG' komt niet voor in de lijst
```

Merk op dat men bij strings kan nagaan of een substring (een aaneenschakeling van karakters) deel uitmaakt van een string. Bij lijsten kan men dit **niet** doen met de `in` operator. In het onderstaande voorbeeld komen de strings `'TCT'` en `'ACG'` beide voor in de lijst, maar wordt toch `False` geretourneerd:

```
>>> ["TCT", "ACG"] in ["TCT", "ACG", "ATG", "CTG"]
False          # ["TCT", "ACG"] is geen element van de lijst
```

De reden hiervoor is dat `["TCT", "ACG"]` geen element is van de lijst. In het onderstaande voorbeeld is dit wel het geval:

```
>>> ["TCT", "ACG"] in [ ["TCT", "ACG"], ["ATG", "CTG"] ]
                        # de elementen van de lijst zijn tevens lijsten
True         # ["TCT", "ACG"] is WEL een element van de lijst
```

7.6.3 Relationele operatoren

Het gebruik van de relationele operatoren `==`, `!=`, `<`, `>`, `<=` en `>=` bij lijsten vertoont een grote analogie met het gebruik ervan bij strings. De vergelijkingsoperator `==` gaat na of twee lijsten gelijk zijn, waarbij gelijk wil zeggen dat ze evenveel elementen bevatten en dat de elementen met eenzelfde index gelijk zijn aan elkaar.

De == operator

Beschouw twee lijsten `L1` en `L2` die elk `n` elementen bevatten. De logische expressie

```
L1 == L2
```

retourneert

- `True` indien voor elke `i = 1, ..., n-1` geldt dat `L1[i] == L2[i]`
- `False` in alle andere gevallen

Merk op dat de logische expressie `L1 != L2` **equivalent** is met `not L1 == L2`.

Het gebruik van deze operator wordt hieronder geïllustreerd:

```
>>> [1, 2, 3] == [1, 2, 3]
True   # lijsten zijn gelijk

>>> [1, 2] == [1, 5]
False  # lijsten niet gelijk

>>> [1, 2, 3] == [1, 2]
False  # lijsten zijn niet gelijk (niet even lang)

>>> [1, 2] == ["1", "2"]
False  # int 1 is niet gelijk aan str "1"
```

Ook de < operator voor lijsten is een natuurlijke uitbreiding van de variant bij strings.

De < operator
Beschouw twee lijsten `L1` en `L2` (de lengtes kunnen verschillend zijn) en de kleinste index `i` waarvoor geldt dat `L1[i] != L2[i]`. De logische expressie

`L1 < L2`

retourneert

- `True` als en slechts als voor deze `i` geldt dat `L1[i] < L2[i]`.
- `False` indien dit niet het geval is.

Opmerking: indien het **gegevenstype** van `L1[i]` en `L2[i]` verschilt, dan wordt (bijna) altijd een **foutmelding** opgeworpen.

Merk op dat er mogelijks geen index `i` bestaat waarvoor waarvoor `L1[i] != L2[i]`: we onderscheiden dan de volgende gevallen:

- `L1 == L2`: de expressie `L1 < L2` retourneert `False`: bv. `[1, 2] < [1, 2]`
- `L1 == L2[0:j]`: de expressie `L1 < L2` retourneert `True`: bv. `[1, 2] < [1, 2, 3]`
- `L1[0:j] == L2`: de expressie `L1 < L2` retourneert `False`: bv. `[1, 2, 3] < [1, 2]`

Het gebruik van deze operator wordt hieronder geïllustreerd:

```
>>> [1, 2, 3] < [1, 10, 1]
True   # op index 1 verschillen de lijsten,
       # omdat 2 < 10 wordt True geretourneerd

>>> ["jan", "pol", "piet"] < ["jan", "an"]
False  # omdat "pol" < "an" False retourneert
```

De overige operatoren zijn eenvoudige (syntactische) uitbreidingen van wat reeds gekend is:

- `L1 <= L2` is equivalent aan `L1 < L2 or L1 == L2`
- `L1 > L2` is equivalent aan `not L1 <= L2`
- `L1 >= L2` is equivalent aan `L1 > L2 or L1 == L2`

Opdracht 7.5 (`korte_instructies2.py`)
Los volgende (korte) opdrachten op. Tracht dit eerst te beredeneren en controleer je antwoord in de console:

- Beschouw de lijst `L1 = [1, 2]`. Wat is het resultaat van de volgende instructies? Vul aan:

```
>>> L2 = 3 * L1 + [3]
>>> print(L2)

.................................................

>>> del L2[:3]
>>> print(L2)

.................................................
```

- Wat is het resultaat van de volgende instructies? Vul aan:

```
>>> "ab" in ["ab", "cd"]

...............
>>> ("ab" * 2) in (["ab"] * 4)

...............
```

- Wat is het resultaat van de volgende instructies? Vul aan:

```
>>> ["a", "b", "c"] >= ["a", "k", "f", "s"]

...............
>>> ["a", 1, 2] < ["a", "1", "2"]  # let op de quotes

...............
```

Opdracht 7.6 (`naamlijst.py`)
Beschouw (ter illustratie) de volgende namenlijst:

```
namen_lst = ["Jan", "Pol", "Hans"]
```

In deze opdracht dien je een programma te schrijven dat de gebruiker toelaat om interactief een aantal bewerkingen uit te voeren op deze lijst.

1. De mogelijke bewerkingen worden hieronder opgelijst. Plaats deze bewerkingen onder elkaar in een script:
   - **Toon** de lijst (lijst wordt op scherm geprint). Voorbeeld input/output:

     ```
     Huidige lijst:
     1) Jan  2) Pol  3) Hans
     ```

   - **Voeg** een **naam toe** (gebruiker krijgt de kans om nieuwe naam in te geven, deze wordt achteraan de lijst toegevoegd). Voorbeeld input/output:

     ```
     Geef een nieuwe naam: Joris
     Nieuwe lijst:
     1) Jan  2) Pol  3) Hans  4) Joris
     ```

   - **Verwijder** een **naam** (gebruiker krijgt kans om bestaande naam in te geven, deze wordt verwijderd uit de lijst). Voorbeeld input/output:

     ```
     Geef een bestaande naam: Pol
     Nieuwe lijst:
     1) Jan  2) Hans  3) Joris
     ```

   - **Wijzig** een **naam** (gebruiker krijgt kans om bestaande naam en nieuwe naam in te geven, de bestaande naam wordt vervangen door de nieuwe naam). Voorbeeld input/output:

     ```
     Geef een bestaande naam: Hans
     Geef een nieuwe naam: Gert
     Nieuwe lijst:
     1) Jan  2) Gert  3) Joris
     ```

2. Pas tenslotte je script aan zodat de gebruiker telkens opnieuw een keuze-menu krijgt waarin hij kan meegeven welk type bewerking hij wenst uit te voeren.

   Mogelijke voorbeeld input/output:

   ```
   [1] Toon, [2] Voeg toe, [3] Verwijder, [4] Wijzig, [9] Stop:  1
   ---> 1) Jan  2) Pol  3) Hans
   [1] Toon, [2] Voeg toe, [3] Verwijder, [4] Wijzig, [9] Stop:  2
   ---> Geef een nieuwe naam: Ingrid
   ---> 1) Jan  2) Pol  3) Hans 4) Ingrid
   [1] Toon, [2] Voeg toe, [3] Verwijder, [4] Wijzig, [9] Stop:  4
   ---> Geef een bestaande naam: Hans
   ---> Geef een nieuwe naam: Griet
   ---> 1) Jan  2) Pol  3) Griet 4) Ingrid
   [1] Toon, [2] Voeg toe, [3] Verwijder, [4] Wijzig, [9] Stop:  9
   Programma beeindigd!
   ```

7.7 Typeconversie: de functie `list()`

Objecten die *iterable* zijn kunnen worden geconverteerd naar objecten van het type **`list`**. De gelijknamige functie **`list()`** voorziet deze functionaliteit. Wanneer deze functie gebruikt wordt, zal een lijst gecreëerd worden die de waarden bevat die worden opgeworpen door de iterator van het object dat als argument aan **`list()`** wordt meegegeven:

```
>>> woord_str = "hallo"
>>> woord_lst = list(woord_str)
>>> woord_lst
["h", "a", "l", "l", "o"]

>>> getallen_lst = list(range(1, 10))
>>> getallen_lst
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Het bovenstaande voorbeeld illustreert tevens hoe men efficiënt een lijst van integers kan genereren.

Opdracht 7.7 (`korte_instructies3.py`)
Omschrijf in één zin wat in het volgende codefragment gebeurt met de waarde van de variabele `a`:

```
a = "vandaag is het dinsdag"
a_lst = list(a)
for i in range(len(a_lst)):
    if a_lst[i] in "aeiou":
        a_lst[i] = a_lst[i].upper()
a = "".join(a_lst)
```

Omschrijving:

7.8 Lijstmethoden

Het gegevenstype **`list`** dat lijsten implementeert voorziet, naast de mogelijkheid tot indexeren, nog een aantal andere functionaliteiten die het werken met lijsten vereenvoudigen. Deze functionaliteiten worden aangeboden onder de vorm van **methoden**. Merk op dat (net zoals bij strings) deze methoden vaak functionaliteiten aanbieden die ook d.m.v. een gepaste keuze van indexeringsoperaties (en evt. een aantal **`for`**-lussen) kunnen worden geïmplementeerd. Redenen om toch voor een methode te kiezen zijn:

- gebruiksgemak en tijdsbesparing,
- leesbaarheid van broncode, en
- efficiëntie bij uitvoeren

7.8.1 Beperkt overzicht van lijstmethoden

Tabel 7.1 bevat een overzicht van de methoden van de klasse `list` (zoals beschreven in de officiële documentatie voor Python 3). Deze methoden kunnen worden toegepast op een lijst `L` door gebruik te maken van de volgende syntax:

`L.`*`methodenaam`*`(`*`argument1`*`, `*`argument2`*`, ...)`

In de volgende subsecties worden een aantal van deze methoden meer in detail besproken.

Tabel 7.1: Enkele lijstmethoden.

Lijst-methode	**Beschrijving**
`append(x)`	Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.
`extend(iterable)`	Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.
`insert(i, x)`	Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.
`remove(x)`	Remove the first item from the list whose value is equal to x. It raises a `ValueError` if there is no such item.
`pop([i])`	Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional)
`clear()`	Remove all items from the list. Equivalent to `del a[:]`.
`index(x[, start[, end]])`	Return zero-based index in the list of the first item whose value is equal to x. Raises a `ValueError` if there is no such item. The optional arguments start and end are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the start argument.
`count(x)`	Return the number of times `x` appears in the list.
`sort(key = None, reverse = False)`	Sort the items of the list in place (the arguments can be used for sort customization, see `sorted()` for their explanation).
`reverse()`	Reverse the elements of the list in place. Equivalent to `a[::-1]`.
`copy()`	Return a shallow copy of the list. Equivalent to `a[:]`.

7.8.2 De methodes `count()` en `index()`

De lijstmethodes `count()` en `index()` hebben een functionaliteit die gelijkaardig is aan die van de gelijknamige stringmethodes.

Method help/signature:

`L.count(x) -> int`

Retourneert het aantal keer dat `x` voorkomt in de lijst `L`

De onderstaande console-sessie illustreert het gebruik van de methode `count()`:

```
>>> mijn_lijst = [1, 2, 3, 4, 1, 4, 1, 1]
>>> mijn_lijst.count(1)
4
```

Method help/signature:

`L.index(x, [, start[, end]]) -> int`

Retourneert de laagste index waarop `x` voorkomt in de lijst `L`. Indien `x` niet voorkomt in `L` wordt een `ValueError` opgeworpen. De (optionele) parameters `start` en `end` laten toe om de zoekoperatie te beperken tot elementen waarvan de index tussen `start` (**inclusief**) en `end` (**exclusief**) ligt.

De onderstaande console-sessie illustreert het gebruik van de methode `index()`:

```
>>> mijn_lijst = [1, 2, 3, 4, 1, 4, 1, 1]
>>> mijn_lijst.index(4)
3     # index waarop 4  voor de eerste keer voorkomt

>>> mijn_lijst.index(5)
ValueError: 5 is not in list
```

Opdracht 7.8 (`gebruikIndex.py`)

Beschouw een lijst `L` van integers (zie onderstaande script). Op de tweede regel wordt de gebruiker gevraagd om een geheel getal in te geven. Vul het script aan: gebruik de operator **`in`** en de methode `index()` zodat uit de lijst `L` automatisch alle waarden die gelijk zijn aan het ingegeven getal verwijderd worden.

```
L = [1, 2, 3, 6, 5, 3, 2, 1, 2, 3, 3]
getal = int(input("Geef een getal in: "))
while ..............  :
    del ......................
print(L)
```

Voorbeeld input/output is:

```
Geef een getal in: 3
[1, 2, 6, 5, 2, 1, 2]
```

7.8.3 De methode `append()`

De methode `append()` voegt een element toe achteraan een bestaande lijst en is een **heel vaak gebruikte methode**.

Method help/signature:

`L.append(x) -> None`

Voegt het object `x` toe achteraan de lijst `L`.

Merk op dat deze methode `None` retourneert en dus **geen nieuwe lijst** zal genereren. Hier wordt de lijst `L` waarop de methode inwerkt zelf aangepast. Merk op dat deze methode dus een **bestaande lijst aanpast** (en dus gebruik maakt van het feit dat een `list` *mutable* is). De onderstaande sessie illustreert het gebruik van de methode `append()`:

```
>>> L = ["het", "is", "vandaag"]
>>> L.append("maandag")
>>> print(L)
["het", "is", "vandaag", "maandag"]
```

De return value van `append()` is steeds `None` en kan desgewenst toegekend worden aan een nieuwe variabele (maar dit is vaak niet nodig):

```
>>> L = ["het", "is", "vandaag"]
>>> a = L.append("maandag")
>>> print(L)
["het", "is", "vandaag", "maandag"]
>>> print(a)
None
```

Opdracht 7.9 (`isogram.py`)

De *letterfrequentieverdeling* is een lijst van 26 integers, waarbij elke integer overeenstemt met de frequentie van een karakter uit het alfabet in dit woord.

1. Schrijf een script dat de frequentieverdeling van de letters van een zelf te kiezen string op het scherm print. Voor de string `'wetenschappen'` bekomen we:

```
woord = "wetenschappen"
# frequentieverdeling = .... vul aan
# ........
print(frequentieverdeling)
 [1, 0, 1, 0, 3, 0, 0, 1, 0, 0, 0, 0, 0, 2, 0, 2, 0, ...]
# a  b  c  d  e  f  g  h  i  j  k  l  m  n  o  p  q  ...
```

2. **Uitbreiding 1**: een woord is een **isogram** wanneer het elk karakter hoogstens één keer bevat. `'wetenschappen'` is dus geen isogram omdat bijvoorbeeld het karakter `p` tweemaal voorkomt. `'filmproducent'` is daarentegen wel een isogram. Implementeer een functie `is_isogram()` die een string als input aanvaardt. Deze functie retourneert `True` indien de string een isogram voorstelt en `False` indien dit niet het geval is.

```
>>> is_isogram("filmproducent")
True
>>> is_isogram("wetenschappen")
False
```

3. **Uitbreiding 2**: de *letterfrequentieverdeling* bevat vaak heel veel nullen omdat een woord maar uit een beperkt aantal karakters bestaat. Een alternatieve manier om de frequentieverdeling voor te stellen maakt gebruik van twee lijsten:
   - een eerste lijst bevat de karakters die voorkomen in de string, en
   - een tweede lijst bevat de frequentie van elk karakter.

   Breid je script uit zodat ook op deze manier de letterfrequentie van het woord wordt bepaald.

   Voor `"wetenschappen"` wordt dit:

```
letters = ["w", "e", "t", "n", "s", "c", "h", "a", "p"] # elk letter 1x
f    =    [ 1,   3,   1,   2,   1,   1,   1,   1,   2 ]
```

7.8.4 De methode `extend()`

Als we met de methode `append()` een lijst proberen te appenderen aan een bestaande lijst, dan wordt de toegevoegde lijst als een **sublijst** opgenomen:

```
>>> L1 = ["het", "is", "vandaag"]
>>> L2 = [30, "april"]
>>> L1.append(L2)
>>> L1
['het', 'is', 'vandaag', [30, 'april']]
>>> L1[-1]
[30, 'april']
>>> type(L1[-1])
list
```

Als minstens één element van een lijst zelf een lijst is, spreken we van **geneste lijsten**. Deze worden in detail besproken in Sectie 7.9.

Met de methode `extend()` kan **elk element van een lijst** toe **achteraan** een bestaande lijst **toegevoegd** worden.

Method help/signature:

```
L1.extend(L2) -> None
```

Voegt elk element van de lijst `L2` toe achteraan de lijst `L1`.

```
>>> L1 = ["het", "is", "vandaag"]
>>> L2 = [30, "april"]
>>> L1.extend(L2)
>>> L1
['het', 'is', 'vandaag', 30, 'april']
```

Opmerking: hetzelfde resultaat kan bekomen worden met een `for`-lus en `append()`:

```
for element in L2:
    L1.append(element)
```

Met als resultaat:

```
>>> L1
['het', 'is', 'vandaag', 30, 'april']
```

7.8.5 Overige lijstmethoden

Hierna worden de overige lijstmethoden kort besproken. Deze methoden zorgen er allen voor dat de lijst waarop ze inwerken gewijzigd wordt. Met uitzondering van de methode `pop()` retourneert elk van deze methoden `None`.

De methode `pop()`

Deze methode verwijdert het laatste element uit de *list* en retourneert dit element. De *list* wordt met één element verkort. Indien een index (`pop(i)`) als argument meegegeven wordt, dan wordt het element met deze index verwijderd en geretourneerd:

```
>>> dna_basen = ["A", "C", "G", "T"]
>>> dna_basen.pop()
'T'
>>> print(dna_basen)
['A', 'G', 'C']
>>> dna_basen.pop(1)
```

```
'G'
>>> print(dna_basen)
['A', 'C']
```

De methode `extend(C)`

Deze methode breidt een lijst uit met de elementen in de iterable `C`. De *list* wordt uitgebreid door elk individueel element uit `C` toe te voegen aan de *list*:

```
>>> dna_basen = ["A", "C"]
>>> dna_basen.extend(["G", "T"])
>>> print(dna_basen)
['A', 'C', 'G', 'T']
```

De methode `insert(i, x)`

Deze methode voegt een element in op een bepaalde positie. Het eerste argument `i` is de positie-*index* waar het element `x` moet komen. **De rest van de elementen worden opgeschoven door hun *index* met één te vermeerderen:**

```
>>> dna_basen = ["A", "C", "G", "T"]
>>> dna_basen.insert(2, 'U')
>>> print(dna_basen)
['A', 'C', 'U', 'G', 'T']
```

De methode `remove(x)`

Deze methode verwijdert het eerste element gelijk aan `x` uit de *list*. Er treedt een foutmelding op wanneer het element niet aanwezig is in de *list*. Indien de verwijdering gelukt is, vermindert de lengte van de *list* met één:

```
>>> dna_basen.remove('U')
>>> dna_basen
['A', 'C', 'G', 'T']
>>> print(dna_basen)
['A', 'C', 'G', 'T']

>>> dna_basen.remove("H")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
```

De methode `sort()`

Deze methode rangschikt (sorteert) de elementen in de *list* van klein naar groot. Enkel de volgorde van de elementen in de *list* verandert (tenzij reeds geordend), niet de elementen zelf.

```
>>> dna_basen = ["T", "A", "G", "C"]
>>> dna_basen.sort()
>>> print(dna_basen)
['A', 'C', 'G', 'T']
```

Opmerkingen

1. De methode `sort()` heeft een parameter `reverse` (default is `False`). Wanneer `reverse` de waarde `True` wordt toegekend zal de volgorde worden omgedraaid (van groot naar klein).
2. De orde die gebruikt wordt bij de sortering is dezelfde als diegene die gebruikt wordt door de relationele operatoren. Elementen kunnen bijgevolg pas gesorteerd worden als de relationele operatoren er op kunnen inwerken.
3. Men kan de default orde (bepaald door de relationele operatoren) wijzigen door gebruik te maken van de `key` parameter. Het gebruik van deze parameter valt echter buiten het bestek van deze inleidende cursus.

De methode `reverse()` keert de volgorde van de elementen om in een *list*. Enkel de volgorde van de elementen in de *list* verandert, niet de elementen zelf.

```
>>> dna_basen = ["A", "C", "G", "T"]
>>> dna_basen.reverse()
>>> print(dna_basen)
['T', 'G', 'C', 'A']   # dna_basen = dna_basen[::-1] geeft hetzelfde resultaat
```

7.9 Geneste lijsten

In de voorgaande secties werden voornamelijk lijsten beschouwd waarvan de elementen floats, integers, strings of een combinatie van deze en andere gegevenstypes waren. Er is echter **geen beperking op het gegevenstype van de elementen van een lijst**. In deze sectie bespreken we lijsten waarvan de elementen ook van het type `list` zijn. Men zegt in dit geval dat de lijsten **genest** zijn. Men noemt deze structuur soms ook een **lijst van lijsten** (*list of lists*).

7.9.1 Inleidende voorbeelden

Matrices kunnen in het werkgeheugen voorgesteld worden door geneste lijsten[1]. Hieronder wordt een matrix getoond en daaronder zijn voorstelling als een lijst waarvan de elementen opnieuw lijsten zijn die elk een rij voorstellen van de matrix.

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

Voorstelling als een lijst van lijsten:

```
>>> A = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ]
```

[1] In Hoofdstuk 8 zullen we zien dat er ook nog andere gegevenstypes zoals NumPy arrays bestaan die gebruikt kunnen worden om matrices voor te stellen in het werkgeheugen.

In de bovenstaande sessie verwijst de variabele `A` naar een lijst die drie elementen bevat en waarbij elk element van het type **`list`** is:

```
>>> len(A)
3

>>> type(A[0])
list
```

Om deze elementen te benaderen kan men (de vertrouwde) indexering gebruiken. Zo wordt de laatste rij van de matrix A voorgesteld door het element met index 2 in de lijst `A`, zoals getoond wordt in de onderstaande sessie. Omdat dit element een lijst is, kan in deze opnieuw geïndexeerd worden:

```
>>> laatste_rij = A[2]            # laatste rij van A (index 2)
>>> print(laatste_rij)
[9, 10, 11, 12]

>>> linksonder = laatste_rij[0]   # eerste element in laatste rij
>>> linksonder
9
```

In het bovenstaande voorbeeld wordt het getal 9 dat linksonder in A staat benaderd door eerst de gepaste rij uit `A` te benaderen met indexering en tenslotte uit deze rij het eerste element te benaderen. Men kan echter het gebruik van de variabele `laatste_rij` vermijden door de indexeringsoperatoren onmiddellijk na elkaar te plaatsen[2]. Soms wordt dit *chaining* genoemd:

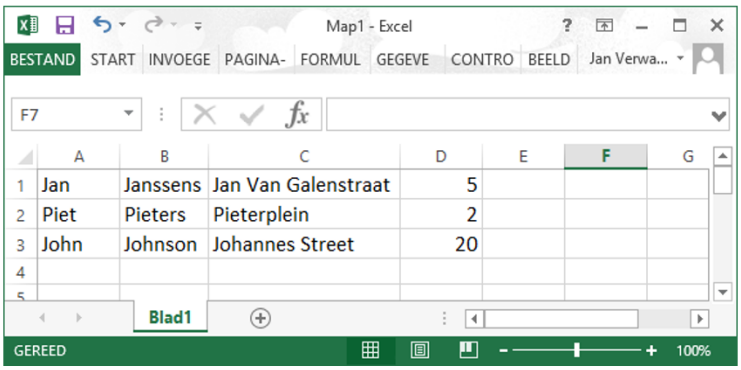
```
>>> A[2][0]
9
```

Dit mechanisme vertoont grote gelijkenissen met de manier waarop matrices geïndexeerd worden in de wiskunde, waarbij $a_{3,1}$ staat voor het element op de derde rij in de tweede kolom in de matrix A. Omdat de indexering in Python start bij 0, wordt dit `A[2][0]`.

Lijsten zijn **heterogene containers**: de gegevenstypes van hun elementen mogen onderling verschillen. In het volgende voorbeeld wordt een kort adresboek samengesteld als een lijst van lijsten[3]. Vaak worden dergelijke gegevens verwerkt in *spreadsheet*-software zoals MS-Excel. Dezelfde gegevens worden daarom hieronder getoond in een spreadsheet.

[2] In sommige programmeertalen wordt dit ook **dubbele indexering** genoemd.

[3] Vaak wordt aangehaald dat de elementen van een lijst bij voorkeur van eenzelfde type zijn. Dit in de eerste plaats om redenen van leesbaarheid en robuustheid tegen logische fouten. In de voorbeelden die hierna volgen volgen we deze richtlijnen **niet**. Deze discussie wordt uitgesteld tot Sectie 7.13.



Deze gegevens kunnen als een lijst van lijsten worden ondergebracht in het werkgegheugen.

```
>>> persoon1 = ["Jan", "Janssens", "Jan Van Galenstraat", 5]
>>> persoon2 = ["Piet", "Pieters", "Pietersplein", 2]
>>> persoon3 = ["John", "Johnson", "Johannes Street", 20]

>>> adressenlijst = [persoon1, persoon2, persoon3]
```

Vervolgens kan men informatie uit deze adressenlijst benaderen via indexering.

```
>>> adressenlijst[1][2]
'Pietersplein'

>>> adressenlijst[2][1:]
['Johnson', 'Johannes Street', 20]
```

Opdracht 7.10 (`adressensom.py`)

Beschouw de variabele `adressenlst` uit het voorgaande voorbeeld. Vul de onderstaande `for`-lussen aan met **één regel code** zodat in beide gevallen de variabele `som` na uitvoeren de som van de drie huisnummers bevat.

```
som = 0
for i in range(len(adressenlijst)):

    ..............................

print(som)
```

```
som = 0
for adres in adressenlijst:

    ..............................

print(som)
```

Opdracht 7.11 (`matrixsom.py`)

Beschouw de variabele `A` (die de matrix A voorstelt) uit het voorgaande voorbeeld:

```
A = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ]
```

Schrijf een script dat de som van alle waarden (alle elementen) in A bepaalt. Je zou 78 moeten bekomen.

7.9.2 Inlezen van gegevens uit `csv` bestanden

Gegevens, zoals de adressenlijst in het voorbeeld hierboven, zijn vaak beschikbaar in bestanden. Deze bestanden zijn vaak tekstbestanden (die je eenvoudig kan bekijken in een text-editor zoals *Notepad*). Indien het gaat om gegevens die gestructureerd worden in een tabelvorm, dan worden vaak de volgende afspraken gevolgd bij de opmaak van deze tekstbestanden:

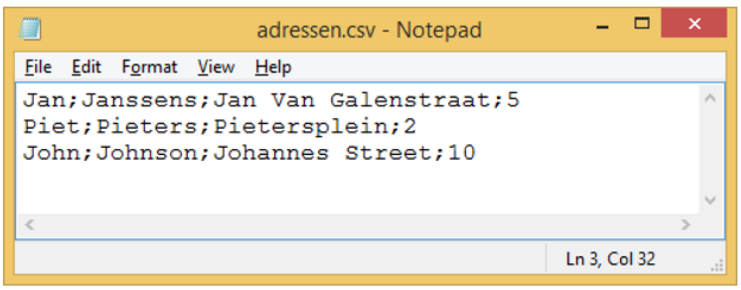
- Elke rij in de tabel bevindt zich op een **afzonderlijke regel** (en eindigt dus met een nieuwelijnkarakter)
- De elementen in eenzelfde rij worden gescheiden door een specifiek **scheidingsteken**. Vaak is dit een puntkomma `';'`, een komma `','` of witruimte (aantal spaties of tab).

Bestanden die deze afspraken volgen, noemt men ***Comma-separated values*** bestanden of kort csv-files. Vaak wordt dit aangegeven door ervoor te zorgen dat de bestandsnaam eindigt op `.csv`, maar dit is niet noodzakelijk.

Tip: bekijk csv-files vóór je ze inleest.
Zorg er steeds voor dat je tekstbestanden **eerst bekijkt in een tekst-editor** zodat je een beeld hebt van de structuur van het bestand. Indien beschikbaar op je systeem is *Notepad++* een heel geschikte tekst-editor (en bovendien gratis). Indien deze niet beschikbaar is, kan eventueel *Wordpad* op een Windows systeem gebruikt worden (is meestal beschikbaar). Notepad (of kladblok) is een alternatief, maar merk op dat *Notepad* niet steeds alle nieuwelijnkarakters toont. Bij recente Windows 10 installaties zou dit wel moeten lukken.

Let op: op een Windows systeem zijn csv-bestanden vaak gelinkt met de applicatie MS-Excel. Door er dubbel op te klikken worden deze vaak automatisch in MS-Excel geopend. Dit is echter **geen** goede werkwijze.

In het onderstaande voorbeeld wordt het bestand `adressen.csv` getoond in *Notepad* (een versie die alle nieuwelijnkarakters correct weergeeft).



Het scheidingsteken (Eng. *separator*) is in dit geval de puntkomma (`;`).

Deze bestanden kunnen worden ingelezen met de functie `listRead()` uit de module `infoFun`. Deze functie retourneert een lijst met daarin drie elementen (drie strings die elk een regel bevatten):

```
>>> adressen = infoFun.listRead("adressen.csv")
>>> print(adressen)
['Jan;Janssens;Jan Van Galenstraat;5', 'Piet;Pieters;Pietersplein;2',
 'John;Johnson;Johannes Street;10']
```

Deze string kan vervolgens worden omgezet in een lijst van lijsten door gebruik te maken van een `for`-lus en de (string)methode `split()` die de elementen splitst o.b.v. het scheidingsteken (hier `';'`) zoals geïllustreerd wordt in het volgende codefragment:

```
import infoFun
adressen = infoFun.listRead("adressen.csv")

adressen_tabel = []
for adres_str in adressen:
   adres_lst = adres_str.split(";")  # splits o.b.v.\ ;
   adressen_tabel.append(adres_lst)  # appendeer lijst aan adressen_tabel
```

Het resultaat van deze code is:

```
>>> adressen_tabel
[['Jan', 'Janssens', 'Jan Van Galenstraat', '5'],
 ['Piet', 'Pieters', 'Pietersplein', '2'],
 ['John', 'Johnson', 'Johannes Street', '10']]
```

De output illustreert dat de variabele `adressen_tabel` verwijst naar een lijst van lijsten. De straat waarin Piet Pieters woont, kan bijvoorbeeld geëxtraheerd worden door gebruik te maken van de (dubbele) indexering `adressen_tabel[1][2]`:

```
>>> adressen_tabel[1][2]
'Pietersplein'
```

Merk op dat de huisnummers in deze voorbeelden nog steeds strings zijn: er staan **aanhalingstekens** rond de huisnummers. Indien men deze wenst te bewaren als integer-objecten, dan kan men het codefragment als volgt aanpassen:

```
adressen = infoFun.listRead("adressen.csv")

adressen_tabel = []
for adres_str in adressen:
   adres_lst = adres_str.split(";")
   adres_lst[3] = int(adres_lst[3])   # converteer huisnummer naar integer
   adressen_tabel.append(adres_lst)
```

Het resultaat van deze code is:

```
>>> adressen_tabel
[['Jan', 'Janssens', 'Jan Van Galenstraat', 5],
 ['Piet', 'Pieters',
 'Pietersplein', 2],
 ['John', 'Johnson', 'Johannes Street', 10]]
```

Merk op dat de aanhalingstekens rond de huisnummers verdwenen zijn, wat aangeeft dat ze als integer objecten worden bewaard in het werkgeheugen.

Oefeningen: Opdrachten 7.19–7.21 zijn toepassingen van de principes die in deze sectie besproken werden. Deze opdrachten werden achteraan toegevoegd bij de **Gemengde opdrachten** van dit hoofdstuk (zie Sectie 7.14).

7.10 De deep copy

In de voorgaande secties werd reeds onderscheid gemaakt tussen

- **aliasing**: twee variabelen die verwijzen naar eenzelfde object, en
- de **copy-slice**: die ervoor zorgt dat een lijst gekopiëerd werd.

Merk op dat de copy-slice een nieuwe lijst genereert, maar niet de elementen van deze lijst kopieert. **Wanneer we met geneste lijsten werken, betekent dit dat een copy-slice er niet noodzakelijk alle gegevens zal kopiëren.** Beschouwen we bijvoorbeeld de geneste lijst `A` (die een matrix voorstelt) en de copy-slice `B`.

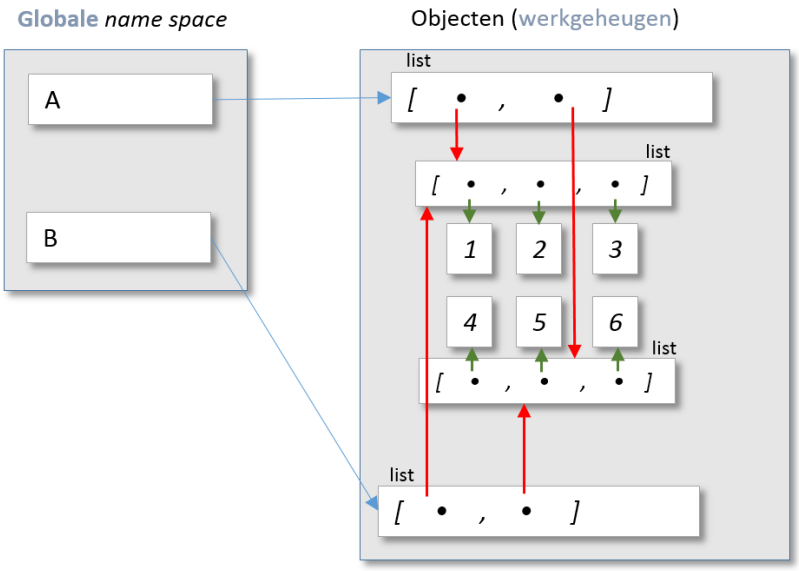
$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

```
>>> A = [ [1, 2, 3], [4, 5, 6] ]
>>> B = A[:]
```

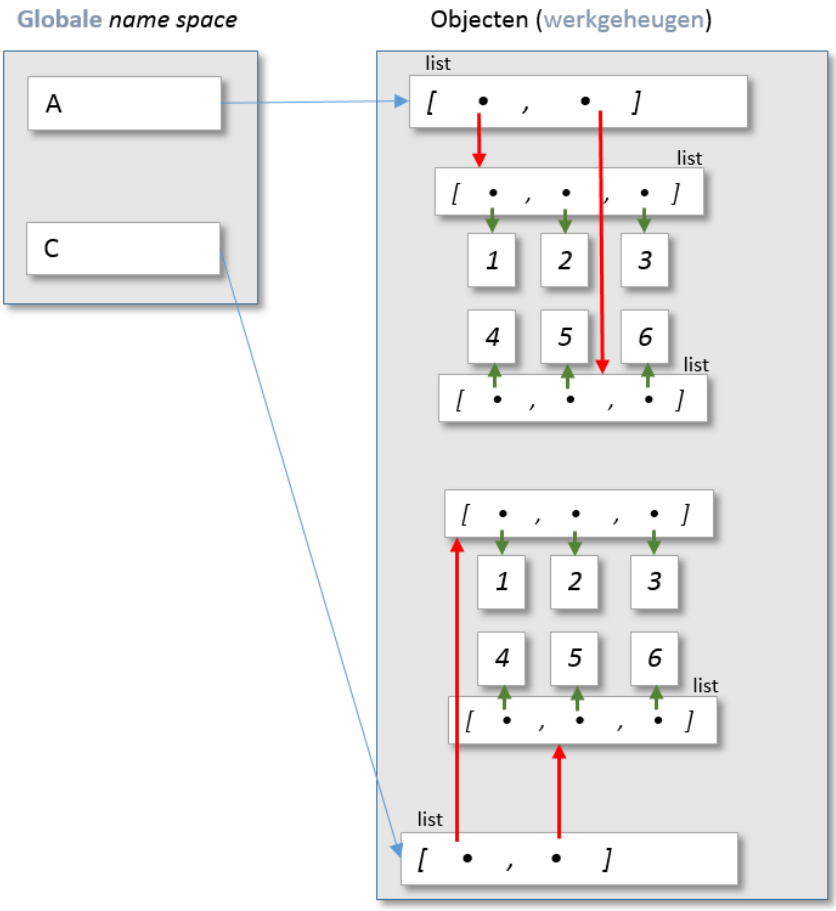
Het toestandsdiagram na het uitvoeren van deze instructies wordt gegeven in Figuur 7.8. Deze figuur toont duidelijk dat het **beide lijsten nog steeds gelinkt** zijn. Daarom noemt men dit type kopie een *shallow copy*. Door een wijziging aan te brengen in `A` kan `B` nog steeds (onbedoeld) wijzigen, zoals geïllustreerd wordt in het onderstaande voorbeeld:

```
>>> B[0][1]
2
>>> A[0][1] = 9
>>> B[0][1]
9               # wijziging ook via B zichtbaar
```

Het omgekeerde is ook waar: een wijziging in `B` zal ook `A` wijzigen.



Figuur 7.8: Toestandsdiagram na uitvoeren van `B = A[:]`.



Figuur 7.9: Toestandsdiagram na uitvoeren van de instructie `C = copy.deepcopy(A)`

De **deep copy** zorgt ervoor dat **alle** gegevens (recursief) gekopieerd worden zodat **beide lijsten volledig onkoppeld** zijn. De deep-copy wordt geïmplementeerd door de functie `deepcopy()` in de module `copy`.

```
>>> import copy
>>> A = [ [1, 2, 3], [4, 5, 6] ]
>>> C = copy.deepcopy(A)
```

Het toestandsdiagram na het uitvoeren van deze operatie wordt gegeven in Figuur 7.9[4].

Wijzigingen in `A` hebben nu **geen invloed meer** op `C` (en omgekeerd):

```
>>> A[0][1] = 90
>>> C[0][1]
2
```

7.11 List comprehensions

List comprehensions zijn bijzondere expressies die toelaten om een lijst te genereren o.b.v. een *iterable*. Beschouw de rij van getallen:

$$1^2, \quad 2^2, \quad 3^2, \quad 4^2, \quad 5^2, \quad \ldots, \quad 99^2\,.$$

Men kan een lijst `L` van deze getallen creëren met de volgende instructies:

```
L = []
for i in range(1, 100):
   L.append(i**2)
```

Door gebruik te maken van *list comprehensions* kan dezelfde lijst als volgt gegenereerd worden:

```
L = [i**2 for i in range(100)]
```

Merk op dat de syntax hier veel korter is.

List comprehensions maken gebruik van de volgende algemene syntax:

`[`*expressie* `for` *variable* `in` *iterable*`]`

Opmerking: men kan in deze syntax ook een `if`-statement opnemen.

Stel dat we een lijst willen met enkel de kwadraten die een veelvoud zijn van 3:

$$9, \quad 36, \quad 81, \quad 144, \quad 225, \quad \ldots, \quad 9801\,.$$

Het volgende codefragment genereert de gevraagde lijst:

[4] Merk op dat, om efficiëntieredenen, kleine integers in het werkgeheugen niet effectief worden gekopieerd. Omdat deze immutable zijn, heeft het al dan niet kopiëren geen invloed op het gebruik (en maakt het dus niet uit of integers al dan niet gekopiëerd worden door de interpreter). Dit is een implementatiedetail dat in deze cursus verder niet behandeld wordt!

```
L = []
for i in range(1, 100):
   if i**2 % 3 == 0:
      L.append(i**2)
```

Gebruikmakende van *list comprehensions* wordt dit:

```
L = [i**2 for i in range(1, 100) if i**2 % 3 == 0]
```

De **veralgemeende** syntax is:

[*expressie* for *variable* in *iterable* if *logischeExpressie*]

7.12 Functies en operatoren voor sequence types

Het gegevenstype `list` is een voorbeeld van een *sequence type*. Ook het type `str` is een sequence type. In de standard library van Python zijn nog een aantal andere sequence types aanwezig die we in deze cursus niet behandelen. Wat al deze sequence types gemeenschappelijk hebben is de mogelijkheid om ze te gebruiken als operanden van `in`, `+` en `*`, ze indexering toelaten en dat ze kunnen dienen als argumenten voor de functies `len()`, `max()` en `min()`. Bovendien implementeren deze gegevenstypes steeds de methodes `index()` en `count()`. Een overzicht wordt gegeven in Tabel 7.2.

Operation	Result
`x in s`	`True` if an item of `s` is equal to `x`, else `False`
`x not in s`	`False` if an item of `s` is equal to `x`, else `True`
`s + t`	the concatenation of `s` and `t`
`s * n or n * s`	equivalent to adding `s` to itself `n` times
`s[i]`	`i`-th item of `s`, origin `0`
`s[i:j]`	slice of `s` from `i` to `j`
`s[i:j:k]`	slice of `s` from `i` to `j` with step `k`
`len(s)`	length of `s`
`min(s)`	smallest item of `s`
`max(s)`	largest item of `s`
`s.index(x[, i[, j]])`	index of the first occurrence of `x` in `s` (at or after index `i` and before index `j`)
`s.count(x)`	total number of occurrences of `x` in `s`

Tabel 7.2: Overzicht van operatoren, functies en methoden die van toepassing zijn op alle sequence types.

7.13 Tuples

Een object van het type `tuple` is een **geordende, indexeerbare collectie** die zeer veel eigenschappen deelt met objecten van het type `list`. Het belangrijkste verschil met het type `list` is dat tuples **immutable** zijn. Dit wil zeggen dat men een tuple niet meer kan wijzigen nadat het werd gecreëerd. Zo kan men indexering wel gebruiken om een element van een tuple te benaderen, maar niet om dit element te wijzigen.

Tuples in deze cursus
Tuples worden in deze cursus in de eerste plaats besproken voor de volledigheid, maar worden maar in beperkt detail beschouwd. Omwille van het feit dat ze immutable zijn, kunnen ze gebruikt worden als *key* in een *dictionary* (zie Hoofdstuk 10), in tegenstelling tot lijsten.

7.13.1 Creëren en indexeren van een tuple

Tuples kan men, analoog aan lijsten, creëren door variabelen of literals na elkaar te plaatsen gescheiden door komma's en omgeven door **ronde** haakjes.
Een voorbeeld:

```
>>> a = (1914, 1918, "Eerste Wereldoorlog")
>>> print(a)
(1914, 1918, 'Eerste Wereldoorlog')

>>> type(a)
tuple
```

Het gebruik van de ronde haakjes is **niet verplicht**. De tuple `a` kan ook als volgt aangemaakt worden:

```
>>> a = 1914, 1918, "Eerste Wereldoorlog"
>>> print(a)
(1914, 1918, 'Eerste Wereldoorlog')
```

Uit dit voorbeeld blijkt duidelijk de analogie met lijsten. Daarnaast kan men ook gebruik maken van **indexering** en **slicing** om elementen uit een tuple op te vragen:

```
>>> a = (1914, 1918, "Eerste Wereldoorlog")
>>> b = a[1]   # element met index 1
>>> b
1918
```

Omdat tuples **immutable** zijn, kunnen ze **niet gewijzigd** worden.

```
>>> a = (1914, 1918, "Eerste Wereldoorlog")
>>> a[1] = 2000
TypeError: 'tuple' object does not support item assignment
```

7.13.2 Typeconversie

De functie `tuple()` laat toe om **elk element van een iterable** te converteren naar een tuple. Het gebruik van deze functie wordt hieronder geïllustreerd:

```
>>> zin = "Hey, alles ok?"
>>> zin_tuple = tuple(zin)
>>> print(zin_tuple)
('H', 'e', 'y', ' ', 'a', 'l', 'l', 'e', 's', ' ', 'o', 'k', '?')

>>> getallen_tuple = tuple(range(1, 5))
>>> print(getallen_tuple)
(1, 2, 3, 4)

>>> L = ["dag", "allemaal"]
>>> L_tuple = tuple(L)
>>> print(L_tuple)
('dag', 'allemaal')
```

Merk ook hier de gelijkenis op met de functie `list()`:

```
>>> zin = "alles ok?"
>>> zin_list = list(zin)
>>> print(zin_list)
['H', 'e', 'y', ' ', 'a', 'l', 'l', 'e', 's', ' ', 'o', 'k', '?']

>>> getallen_list = list(range(1, 5))
>>> print(getallen_list)
[1, 2, 3, 4]
```

7.13.3 Tuple-methoden

Twee frequent gebruyikte tuple-methoden zijn `count()` en `index()`. Het gebruik ervan is analoog aan dat van de gelijknamige methoden bij het type `list`. Merk bovendien op dat het type `tuple` een **sequence type** is. Dit houdt in dat alle operatoren, methoden en functies beschreven in Tabel 7.2 ook toegepast kunnen worden op tuples.

7.13.4 `tuple` versus `list`

Uit de voorgaande secties blijkt dat tuples veel gemeenschappelijk hebben met lijsten, maar dat ze **immutable** zijn. Dit werpt (mogelijks) de vraag op wat het nut is van het gebruik van tuples. Hierna lijsten we de belangrijkste redenen op:

- **Immutable**
  Omdat tuples immutable zijn, 'weet' je als programmeur dat tuples niet (al dan niet per ongeluk) kunnen gewijzigd worden. Dit kan een voordeel hebben bij aliasing. Beschouw de volgende instructies:

```
>>> a = (1, 5, 7)
>>> b = a
```

  De variabelen `a` en `b` verwijzen naar hetzelfde object. Omdat dit object een tuple is, weet men dat dit niet meer kan gewijzigd worden. Indien men als programmeur weet dat een sequentie niet meer gewijzigd hoeft te worden, dan is een tuple een veilige keuze omdat men daar dan ook zeker van is.

- **Dictionaries**
  Omdat tuples immutable zijn, kunnen ze gebruikt worden als *key* in een dictionary. Meer toelichting daarover kan je terugvinden in Hoofdstuk 10.

- **Semantiek**
  De semantiek of *betekenis* die de programmeur geeft aan tuples verschilt vaak van die van lijsten. Vaak zal men lijsten gebruiken als een homogeen collectietype. Vaak houdt dit in dat alle elementen van een lijst hetzelfde gegevenstype hebben (ook al is dat niet noodzakelijk) en bovendien semantisch ook gelijkaardig zijn, zoals bijvoorbeeld een lijst van voornamen.

```
>>> namen = ["Jan", "Pol", "Hans"]
```

  Wanneer men over eenzelfde fysiek object een aantal eigenschappen wenst te bewaren die samen horen (voorbeeld achternaam, voornaam, leeftijd van één persoon) wordt bij voorkeur een tuple gebruikt:

```
>>> persoon = ("Jan", "Janssens", 35)
```

  Dit houdt in dat persoonsgegevens van verschillende personen bij voorkeur worden bewaard als een lijst van tuples:

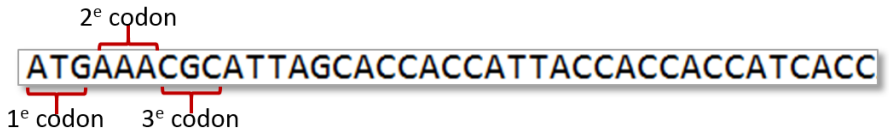
```
>>> persoon1 = ("Jan", "Janssens", 35)
>>> persoon2 = ("Piet", "Pieters", 41)
>>> persoon3 = ("Pol", "Paulissen", 67)
>>> personenlijst = [persoon1, persoon2, persoon3]
```

7.14 Gemengde opdrachten

Opdracht 7.12 (`dna_naar_codonlijst.py`)

Een DNA-sequentie is een opeenvolging van de karakters `'A'`, `'C'`, `'G'` en `'T'`.

1. Implementeer een functie `dna_naar_codonlijst()` die een DNA-sequentie (type *string*, een opeenvolging van karakters `'A'`, `'C'`, `'G'` en `'T'`) als input aanvaardt en een *list* van codons (zie figuur) retourneert.



   De onderstaande sessie illustreert het gebruik van deze functie:

```
>>> codonlijst = dna_naar_codonlijst("AAAGCCTTC")
>>> print(codonlijst)
['AAA', 'GCC', 'TTC']
```

2. **Uitbreiding 1**: indien de lengte van de DNA-string geen veelvoud is van drie, zorg er dan voor dat onvolledige codons (die minder dan 3 karakters tellen) niet worden toegevoegd:

```
>>> codonlijst = dna_naar_codonlijst("AAATTTGG") # 3e codon onvolledig
>>> print(codonlijst)
['AAA', 'TTT']
```

3. **Uitbreiding 2**: voeg een extra parameter `unique` toe in de functiedefinitie met als default waarde `False`. Deze parameter moet toelaten om een **lijst van unieke codons** te retourneren. Elk codon komt dus hoogstens 1 keer voor:
   - Indien `unique = True` wordt een lijst van unieke codons geretourneerd.
   - Indien `unique = False` wordt de oorspronkelijk functionaliteit behouden.

```
>>> codonlijst = dna_naar_codonlijst("AAATTTAAA", unique = False)
>>> print(codonlijst)
['AAA', 'TTT', 'AAA']
>>> codonlijst = dna_naar_codonlijst("AAATTTAAA", unique = True)
>>> print(codonlijst)
['AAA', 'TTT']
```

Opdracht 7.13 (`kloktijd.py`)

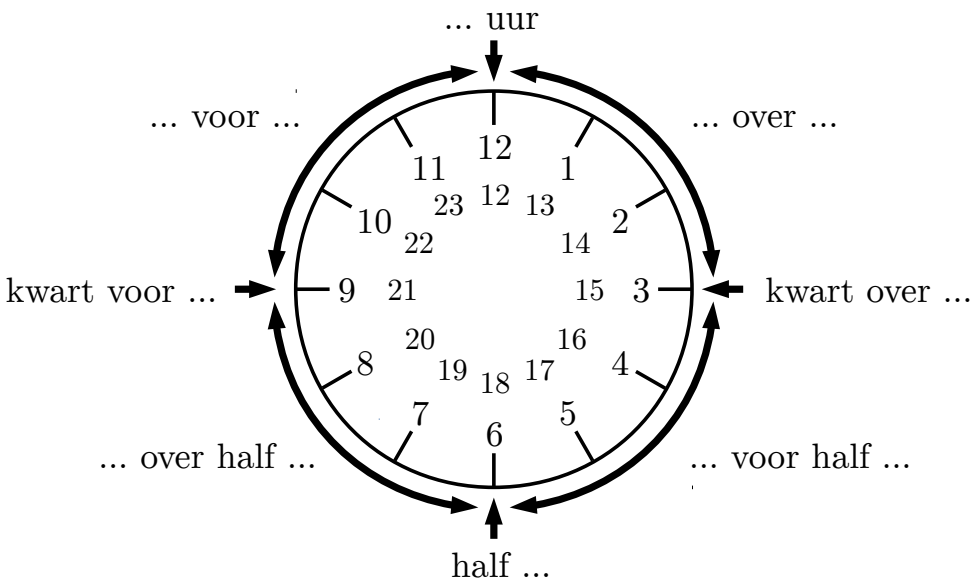
Een tijdstip kan je weergeven in een 24-uursnotatie (`HH:MM`). Hierbij wordt het uur-gedeelte `HH` weergeven door `0`, `1`, `2`, ... `22`, `23` en het minuut-gedeelte `MM` door `00`, `01`, `02`, ..., `58`, `59`. Tussen `HH` en `MM` staat altijd een dubbele punt (`:`). Voorbeelden zijn: `2:35`, `6:15`, `18:44` en `23:05`. Het uur-gedeelte `HH` mag ook steeds voorgesteld worden met twee cijfers, bv. `02:35`, `06:15` en `09:30`.

Opgave - vul het bestand `kloktijd.py` aan

1. Implementeer een script waarin herhaaldelijk aan de gebruiker gevraagd wordt een tijd in 24-uursnotatie in te geven totdat een lege *string* ingegeven wordt. Het script moet nagaan of de ingegeven tijd in 24-uursnotatie al dan niet geldig is en moet een gepaste melding op het scherm weergeven. Indien je dit wenst mag je gebruikmaken van een eigen functie die de geldigheid controleert, maar het **gebruik van een functie is niet verplicht**!

```
Geef een tijd: 2:55
2:55 is een geldige notatie.
Geef een tijd: 03:06
03:06 is een geldige notatie.
Geef een tijd: 16:00
16:00 is een geldige notatie.
Geef een tijd: 24:10
24:10 is GEEN geldige notatie.
Geef een tijd: 58:11
58:11 is GEEN geldige notatie.
Geef een tijd: a4:52
a4:52 is GEEN geldige notatie.
Geef een tijd: 4:5
4:5 is GEEN geldige notatie.
Geef een tijd:
Programma stopt.
```

2. Implementeer een functie `kloktijd()`, waaraan twee *int* argumenten moet meegegeven worden die het uur (`0`, `1`, `2`, ... `22`, `23`) een minuten (`0`, `1`, `2`, ..., `58`, `59`) voorstellen en die een *string* retourneert met de tijd in woorden volgens onderstaande afbeelding.



Tip: om bijvoorbeeld de integer `3` om te zetten naar een string `"drie"` kan je gebruikmaken van een lijst:

```
>>> getallen = ["nul", "een", "twee", "drie", ..., "twaalf"]
>>> getallen[3]
'drie'      # link tussen int en string
>>> getallen.index('drie')
3           # link tussen string en int
```

Het gebruik van de functie `kloktijd()` wordt hieronder geïllustreerd.

```
>>> print(kloktijd(7, 8))
acht over zeven
>>> print(kloktijd(9, 15))
kwart over negen
>>> print(kloktijd(2, 22))
acht voor half drie
>>> print(kloktijd(15, 30))
half vier
>>> print(kloktijd(18, 44))
veertien over half zeven
>>> print(kloktijd(20, 45))
kwart voor negen
>>> print(kloktijd(23, 55))
vijf voor twaalf
```

Opdracht 7.14 (`codontabel.py`)

Een codon is een opeenvolging van drie nucleotiden. Bij de vorming van een eiwit zal tijdens de translatie elk codon worden *vertaald* in een aminozuur. `codontabel.csv` bevat voor 20 aminozuren de volgende informatie: de naam, de *single letter data base code*[5] (SLC) en tenslotte alle codons die vertaald worden naar dit aminozuur. Voer de volgende opdrachten uit:

1. **Implementeer de functie** `lees_codontabel`. Deze functie aanvaardt een bestandsnaam (parameter `bestandsnaam`), zoals bijvoorbeeld het bestand `codontabel.csv`. Deze functie leest de informatie die in dit bestand aanwezig is in, en retourneert ze als een geneste lijst. Je broncode heeft de volgende structuur:

```
#%% functiedefinities
def lees_codontabel(bestandsnaam):
    ... # vul aan
    return codontabel

#%% Instructies
tabel = lees_codontabel("codontabel.csv")
```

De inhoud van de variabele `tabel` is:

[5]Een algemeen aanvaarde afkorting voor het aminzuur.

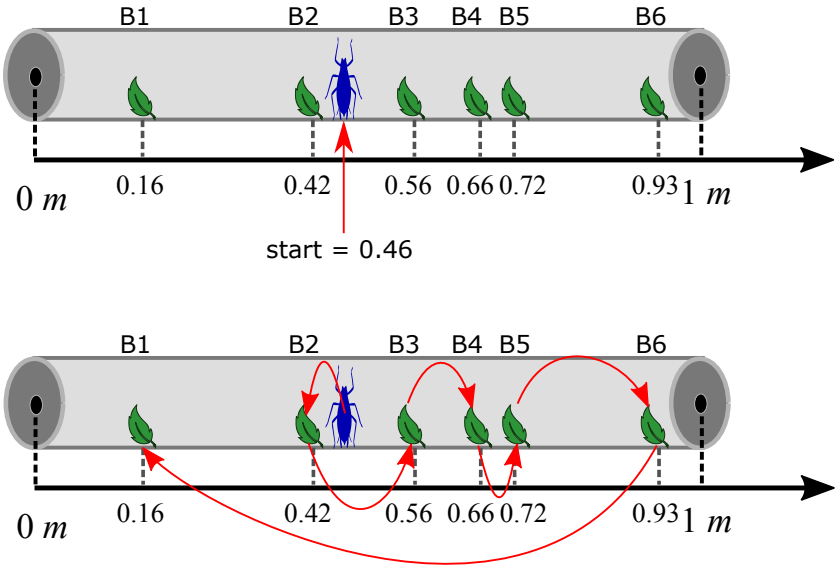
```
>>> print(tabel)
[['Isoleucine', 'I', 'ATT', 'ATC', 'ATA'],
 ['Leucine','L','CTT','CTC','CTA','CTG','TTA','TTG'],
        ... ]
```

2. **Implementeer de functie** `toon_codontabel()`, die een codontabel als input aanvaardt en deze op een gestructureerde manier op het scherm toont (deze functie retourneert niets, maar bevat wel een aantal `print`-statements).
   - Gebruik f-string om de namen en SLC codes mooi uit te lijnen (bv. de expressie `"'{hallo':10s}"` retourneert een string `'hallo     '`, waarin `hallo` **achteraan** wordt aangevuld met spaties zodat deze in totaal 10 karakters telt).
   - Gebruik indien nodig de string methode `join()` (bv. de expressie `"++".join(['a', 'b', "c"])` retourneert de string `'a++b++c'`).

```
>>> tabel = lees_codontabel("codontabel.csv")
>>> toon_codontabel(tabel)

Isoleucine      I      ATT  ATC  ATA
Leucine         L      CTT  CTC  CTA  CTG  TTA  TTG
Valine          V      GTT  GTC  GTA  GTG
...
Lysine          K      AAA  AAG
Arginine        R      CGT  CGC  CGA  CGG  AGA  AGG
Stop codons     Stop   TAA  TAG  TGA
```

Opdracht 7.15 (`bladluizen.py`)

Bladluizen zijn vrij opportunistische insecten die, wanneer ze kunnen kiezen uit verschillende voedselbronnen, vaak kiezen voor de dichtstbijzijnde. Ze zullen deze gebruiken tot deze is uitgeput en zich vervolgens verplaatsen naar de volgende dichtst bijzijnde voedselbron, enz. Dit gaat door tot alle voedselbronnen geconsumeerd zijn. Deze procedure wordt hieronder geïllustreerd. De volgorde waarin de blaadjes worden geconsumeerd is in dit voorbeeld B2, B3, B4, B5, B6, B1.



Voer de volgende opdrachten uit:

1. Schrijf een functie `dichtste_blad()` met de volgende drie parameters:

   - `pos`: een float, de **positie** waarop op bladluis zich bevindt
   - `pos_blaadjes`: een lijst van floats, **plaatsen** waarop de blaadjes zich bevinden
   - `namen_blaadjes`: lijst van strings, de **namen** van de blaadjes in `pos_blaadjes`

   Deze functie retourneert de naam van het blad dat zich het dichtst bij de bladluis bevindt.

```
>>> pos = 0.46
>>> pos_blaadjes = [0.16, 0.42, 0.56, 0.66, 0.72, 0.93]
>>> namen_blaadjes = ["B1", "B2", "B3", "B4", "B5", "B6"]
>>> naam = dichtste_blad(pos, pos_blaadjes, namen_blaadjes)
>>> print(naam)
B2
>>> pos2 = 0.52
>>> pos_blaadjes2 = [0.72, 0.23, 0.65]
>>> namen_blaadjes2 = ["BladA", "BladB", "BladC"]
>>> naam2 = dichtste_blad(pos2, pos_blaadjes2, namen_blaadjes2)
>>> print(naam2)
BladC
```

2. Schrijf een functie `consumptie()`, met dezelfde parameters als `dichtste_blad()`, die de namen van de blaadjes op het scherm print in de volgorde waarin de blaadjes worden geconsumeerd.

```
>>> post = 0.46
>>> pos_blaadjest = [0.16, 0.42, 0.56, 0.66, 0.72, 0.93]
>>> namen_blaadjes = ["B1", "B2", "B3", "B4", "B5", "B6"]
>>> consumptie(pos, pos_blaadjes, namen_blaadjes)
Volgorde van consumptie:
B2, B3, B4, B5, B6, B1
```

```
>>> pos2 = 0.52
>>> pos_blaadjes2 = [0.72, 0.23, 0.65]
>>> namen_blaadjes2 = ["BladA", "BladB", "BladC"]
>>> consumptie(pos2, pos_blaadjes2, namen_blaadjes2)
Volgorde van consumptie:
BladC, BladA, BladB
```

Opdracht 7.16 (`morse.py`)

Morse code is een codering die toelaat om tekstuele informatie (karakters) door te geven o.b.v. een sequentie aan/af tonen, lichtsignalen of clicks op een eenvoudige manier. De Internationale Morse Code codeert de karakters uit het Latijns alfabet, de Arabische cijfers en een aantal leestekens.

Het bestand `"boodschap_morse.txt"` bevat een boodschap in Morse code (je kan dit eenvoudig inlezen met de functie `stringRead()` in de module `infoFun`). Lange tonen worden gecodeerd door min-tekens (`-`) en korte tonen door punten (`.`). Verschillende karakters worden gescheiden door spaties. Woorden worden van elkaar gescheiden door een plusteken (`+`) voorafgegaan en gevolgd door een spatie (dus `" + "`). **Tip:** Bekijk deze tekst eerst eens in Wordpad.

Voer de volgende opdrachten uit:

1. Schrijf een script dat telt hoeveel afzonderlijke woorden voorkomen in `"boodschap_morse.txt"`. **Opmerking**: dit kan zeer kort.
2. Schrijf een script dat (automatisch) de gecodeerde tekst in `"boodschap_morse.txt"` vertaalt en op het scherm toont. Je kan hierbij gebruikmaken van het bestand `"morse_code.txt"`. Dit bestand bevat een digitale versie van de Morse tabel die hierboven getoond wordt, en die je kan inlezen bvb. met `listRead()`.

Opdracht 7.17 (`reservaten.py`)

In het zuiden van Australië zijn er 47 bosreservaten. De CSV-file `bosreservaten.csv` bevat voor elk van deze reservaten de volgende informatie: een nummer (van 1 t.e.m. 47), de naam van het reservaat, de breedteligging (of latitude) en de lengteligging (of longitude). Deze zijn gescheiden door puntkomma's (`';'`). Elk reservaat staat op een afzonderlijke regel en de reservaten zijn alfabetisch gerangschikt. Elk reservaat wordt, met zijn volgnummer, aangeduid op de onderstaande kaart.

De afstand in vogelvlucht tussen twee plaatsen op de wereldbol kan berekend worden o.b.v. de coördinaten (latitude en longitude) van deze plaatsen m.b.v. de *haversine* formule. Deze formule werd reeds geïmplementeerd als functie en is te vinden in het bestand `haversine.py`.

Men wenst achtereenvolgens alle bosreservaten te bezoeken **in alfabetische volgorde**. Men start dus in *'Bagdad'* (nummer 1 op de kaart), gaat dan in vogelvlucht naar *'Boolara'* (nummer 2 op de kaart) enz. tot men uiteindelijk in *'Wombat Flat'* terecht komt (nummer 47). Schrijf een script dat de totale afstand berekent die men op die manier aflegt. Uiteraard kan je in dit script de functie `afstand_lat_long()` oproepen.

Opdracht 7.18 (`lozingen.py`)

Om te voldoen aan de milieuwetgeving moeten bedrijven hun lozingsdebiet regelmatig meten. Een bedrijf meet elk uur het lozingsdebiet (in m^3/uur) met een sensor en bewaart de metingen in tekstbestanden. Ontbrekende waarden, als gevolg van een storing van de sensor, worden in deze bestanden aangeduid door *NaN* (*not a number*). De waarnemingen kunnen voorgesteld worden door een lijst van *strings* (het voorbeeld hieronder is een deel van `lozingen_week1.txt`).

```
>>> infoFun.listRead("lozingen_week1.txt")
['237', 'NaN', '234', '236', '239', '244', 'NaN', '248', ...]
```

Voer de volgende opdrachten uit:

1. NaN's zijn ongewenst en worden vaak vervangen door het gemiddelde van hun onmiddellijke buren; dit zijn de waarden net voor en na de NaN. Dit proces heet imputatie. We gaan er hierbij van uit dat de buren van NaN's steeds getallen zijn (dus dat geen twee NaN's onmiddellijk na elkaar voorkomen) en dat het eerste en het laatste element in de lijst steeds getallen zijn. Schrijf een script waarin het bestand `lozingen_week1.txt` wordt ingelezen, de NaN's in deze lijst *automatisch* worden gedetecteerd en geïmputeerd. Als resultaat wordt een lijst van *floats* gegenereerd (variabele `imputed1`). Een deel van deze lijst wordt hieronder getoond:

```
>>> lozingen1 = infoFun.listRead("lozingen_week1.txt")
>>> lozingen1 =
 ['237', 'NaN', '234', '236', '239', '244', 'NaN', '248', ...]
>>> imputed1
 [237.0, 235.5, 234.0, 236.0, 239.0, 244.0, 246.0, 248.0, ...]
```

2. In de praktijk kunnen uiteraard meerdere NaN's na elkaar voorkomen.

```
>>> lozingen2 = infoFun.listRead("lozingen_week2.txt")
>>> lozingen2
['235', 'NaN', 'NaN', 'NaN', '241', '243', 'NaN', 'NaN', '247', ...]
```

   Ook hier kan men de waarde van een NaN vervangen door het gemiddelde van de meest nabije linker- en rechterbuur die geen NaN zijn. Je mag er hier ook van uit gaan dat het eerste en het laatste element in de lijst steeds getallen zijn. Om dit te kunnen doen, moeten deze buren eenvoudig kunnen bepaald worden. Implementeer de functie `bepaalBuren()` die twee argumenten aanvaardt:

   - Het eerste argument is een lijst zoals `lozingen2`.
   - Het tweede argument is de *index* van een NaN in deze lijst.

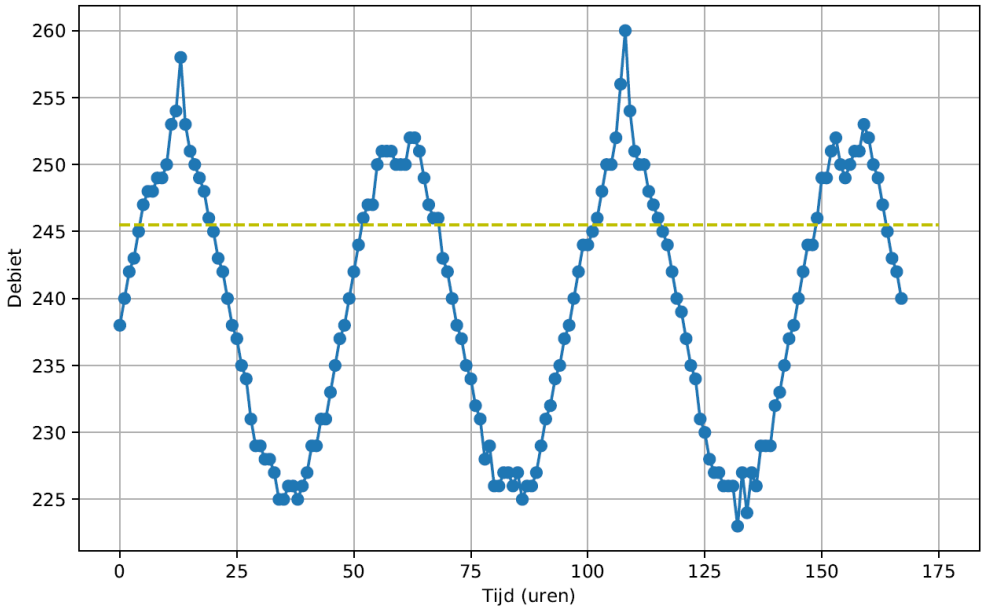
   De functie retourneert de waarde van de meest nabije linker- en rechterbuur (lijst van twee floats) van de NaN waarvan de index werd meegegeven.

```
>>> bepaalBuren(lozingen2, 2)  # linkerbuur is 235, rechterbuur 241
[235.0, 241.0]
>>> bepaalBuren(lozingen2, 3)
[235.0, 241.0]
>>> bepaalBuren(lozingen2, 6)
[243.0, 247.0]
```

3. Implementeer de functie `imputeer()` die een lijst van *strings* met metingen (zoals `lozingen2`) als argument aanvaardt. Deze functie retourneert een lijst van *floats* waarin dezelfde waarden kunnen teruggevonden worden als in de originele lijst, maar waarin alle NaN's vervangen werden door het gemiddelde van hun buren.

```
>>> lozingen2
['235', 'NaN', 'NaN', 'NaN', '241', '243', 'NaN', 'NaN', '247', ...]

>>> imputed2 = imputeer(lozingen2)
>>> imputed2
[235.0, 238.0, 238.0, 238.0, 241.0, 243.0, 245.0, 245.0, 247.0, ...]
```

4. (*) Het debiet mag de drempelwaarde van 245.5 niet te vaak overschrijden. De onderstaande figuur toont de evolutie van het uurlijkse debiet (deze grafiek werd bekomen door de debieten in `lozingen_week3.txt`) te plotten in functie van de tijd. Er komen **geen** NaN's voor.



Uit deze figuur blijkt dat er 4 periodes zijn waarop de drempelwaarde overschreden wordt (een periode is een reeks van opeenvolgende uren waarop de drempelwaarde overschreden wordt).

- Schrijf een script dat automatisch telt hoeveel periodes er zijn waarop de drempelwaarde overschreden wordt. Print dit aantal op het scherm.
- Bepaal voor elke periode tevens automatisch hoeveel uren deze duurt (dit is het aantal metingen dat boven de drempelwaarde ligt).

```
Er zijn 4 periodes waarin de drempel overschreden wordt.
Periode 1 heeft lengte van 15
Periode 2 heeft lengte van 17
Periode 3 heeft lengte van 14
Periode 4 heeft lengte van 15
```

Opdracht 7.19 (`hoofdsteden_inlezen.py`)

Het bestand `country-capitals.csv` is een csv-bestand dat een lijst bevat van landen, met daarbij de namen van coördinaten van de hoofdsteden van die landen. Tevens wordt de naam van het continent vermeld. Elke regel bevat deze informatie in de volgende volgorde (gescheiden door komma's): landnaam, hoofdstadnaam, latitude, longitude, landcode, continentnaam. Voer de volgende opdrachten uit:

Vóór je begint

- Lees eerst Sectie 7.9.2.

Opgave:

1. Voer eerst de volgende opdrachten uit:
   - Bekijk het betand `country-capitals.csv` in een text-editor.
   - Schrijf een Python script waarin dit bestand wordt ingelezen en in het werkgeheugen wordt ondergebracht als een geneste lijst (lijst van lijsten). Van elk land bevat deze geneste lijst de **landaam**, de **naam van de hoofdstad**, **latitude** en **longitude** (de landcode en continentnaam moeten niet bewaard worden). Hieronder wordt een deel van deze geneste lijst getoond:

```
>>> print(hoofdstaden_tabel)
[['Somaliland', 'Hargeisa', 9.55, 44.05],
 ['South Georgia and ...', 'King Edward Point', -54.283333, -36.5],
 ['French Southern and ...', 'Port-aux-Francais', -49.35, 70.216667],
... ]
```

   - Merk op dat de coördinaten bewaard worden als `float` (en niet als `str`). Pas je implementatie aan indien dit niet het geval zou zijn.
   - Wijzig je implementatie zodat de **landnaam** en **naam van de hoofdstad** van plaats worden gewisseld. Het resultaat wordt:

```
>>> print(hoofdstaden_tabel)
[['Hargeisa', 'Somaliland', 9.55, 44.05],
['King Edward Point', 'South Georgia and ... ', -54.283333, -36.5],
 ['Port-aux-Francais', 'French Southern and ...', -49.35, 70.216667],
... ]
```

   - Sorteer de verschillende elementen o.b.v. de **naam van de hoofdstad** (de eerste hoofdstad word dan Abu Dhabi). Het resultaat wordt:

```
>>> print(hoofdstaden_tabel)
[['Abu Dhabi', 'United Arab Emirates', 24.466666, 54.366667],
 ['Abuja', 'Nigeria', 9.08333333, 7.533333],
 ['Accra', 'Ghana', 5.55, -0.216667],
...]
```

2. **Uitbreiding 1**: wrap de broncode die je schreef in een functie met als naam `lees_gegevens()`. Deze functie aanvaardt een bestandsnaam (vb. `"country-capitals.csv"`) als input en retourneert de gesorteerde hoofdstedenlijst.

```
################## functiedefinities ######################
def lees_gegevens(bestandsnaam):
    ...
    ...
    return hoofsteden
################## instructies       ######################
hoofstedentabel = lees_gegevens("country-capitals.csv")
print(hoofstedentabel)
```

3. **Uitbreiding 2**: implementeer de functie `toon_gegevens()`. Deze functie aanvaardt de hoofdstedentabel uit de voorgaande opdracht, en toont (print) deze op een gestructureerde manier op het scherm. Deze functie retourneert niets (en bevat dus geen `return`-statement). De output ziet er als volgt uit:

```
>>> toon_gegevens(hoofstedentabel)
Abu Dhabi           United Arab Emirates              24.47   54.37
Abuja               Nigeria                            9.08    7.53
...
Yerevan             Armenia                           40.17   44.50
Zagreb              Croatia                           45.80   16.00
```

Opdracht 7.20 (`hoofdsteden.py`)

Deze opdracht bouwt verder op de voorgaande Opdracht 7.19.

1. Schrijf een script waarin:
   - de gebruiker gevraagd wordt om een hoofdstad in te geven,
   - de naam van het bijhorende land en de coördinaten van deze hoofdstad **automatisch** worden getoond op het scherm.

   Voorbeeld input/output is:

```
Geef een hoofdstad (in het Engels): Brussels
Hoofdstad:  Brussels
Land     :  Belgium
Latitude :  50.83
Longitude:  4.33
```

2. **Uitbreiding**: wrap de broncode die je schreef een een functie `zoek_hoofdstad_gegevens()` die een tabel met hoofdsteden en een zoekstad als input aanvaardt en de zoekhoofdstad, land, latitiude en longitude retourneert. Na herschikken zal je script de volgende structuur hebben:

```
#%% functiedefinities
def lees_gegevens(bestandsnaam):
    ... # zie voorgaande opdracht
    return hoofdsteden

def zoek_hoofdstad_gegevens(hoofdstedentabel, zoek_hoofdstad):
    ... # vul zelf aan
    return hoofdstad, land, lat, lon
#%% Instructies
hoofdsteden = lees_gegevens("country-capitals.csv")
zoekstad = input("Geef een hoofdstad (in het Engels): ")
stad, land, lat, lon : zoek_hoofdstad_gegevens(hoofdsteden, zoekstad)
print("Hoofdstad:", stad)
print("Land:", land)
...
```

Opdracht 7.21 (`hoofdsteden_dichtbij.py`)

Deze opdracht bouwt verder op Opdrachten 7.19 en 7.20. Beschouw de latitute en longitue (`lat1`, `lon1`) en (`lat2`, `lon2`) van twee plaatsen op aarde. De (Euclidische) afstand `d` tussen deze twee plaatsen is:

$$d = \sqrt{(\text{lat1} - \text{lat2})^2 + (\text{lon1} - \text{lon2})^2}$$

1. Implementeer een functie `zoek_dichtste_hoofdstad()` die drie inputs aanvaardt: een hoofdstedentabel, de latitude van een plaats op aarde en de longitude van deze plaats. Deze functie retourneert de naam van de hoofdstad die het dichtst in de buurt ligt van deze coördinaten. Je script zal de volgende structuur hebben:

```
#%% functiedefinities
def lees_gegevens(bestandsnaam):
    ... # zie voorgaande opdracht
    return hoofdsteden

def zoek_dichtste_hoofdstad(hoofdstedentabel, latitude, longitude):
    ... # vul zelf aan
    return naam_dichtste_stad
#%% Instructies
hoofdsteden = lees_gegevens("country-capitals.csv")
dichtste_stad = zoek_dichtste_hoofdstad(hoofdsteden, 9.0, 7.4)
print("De dichtste stad is:", dichtste_stad)
```

De output van dit scirpt is:

```
De dichtste stad is: Abuja
```

2. **Uitbreiding**: omdat de aarde bolvormig is, is de Euclidische afstand geen geschikte maat om afstanden te berekenen. De *great-circle distance* is een meer correcte afstandsmaat. Deze afstand kan berekend worden o.b.v. de *haversine* formule. Deze formule werd reeds geïmplementeerd als functie en is te vinden in het bestand `haversine.py`. Pas de functiedefinitie van `zoek_dichtste_hoofdstad()` aan zodat de *haversine* formule gebruikt wordt in plaats van de Euclidische afstand.

8

Homogene arrays

Arrays zijn homogene collecties van elementen (bijvoorbeeld integers) die bewaard worden in een aaneensluitend blok geheugen van vaste grootte. Vaak worden deze data georganiseerd volgens assen. In het geval van 2 assen kan men deze interpreteren als de rijen en kolommen van een array en kan men de array weergeven als een homogene tabel. Matrices zijn voorbeelden van (wiskundige) structuren die zich lenen tot het gebruik van arrays. Arrays zijn built-in gegevenstypes in veel programmeertalen, maar dit is niet het geval in Python. Ze worden er geïmplementeerd door de package `numpy`.

Opmerking: in dit hoofdstuk komt Numpy uitgebreid aan bod. Numpy is opgebouwd rond een beperkt aantal principes. Dit vrij uitgebreide hoofdstuk bevat deze principes en daarnaast tal van illustraties van deze principes en de functionaliteiten die ze bieden. Leg bij het studeren dan ook de nadruk op het beheersen van deze principes en het vlot kunnen opzoeken van informatie in dit hoofdstuk. Het is zinloos om alle functies die in dit hoofdstuk aan bod komen te memoriseren.

8.1 Wat is NumPy?

NumPy (*Numerical Python*) is de basismodule voor *scientific computing* in Python. Het is een Python-*library* waarmee multidimensionale array objecten (en objecten daaruit afgeleid) aangemaakt kunnen worden (bv. vectoren en matrices). NumPy bevat eveneens een assortiment routines voor wiskundige en logische bewerkingen op arrays, vormmanipulatie, sorteren, selecteren, input/output (I/O), discrete Fourier transformaties, basis lineaire algebra, basis statistische operaties, simulaties en nog veel meer.

De kern van de NumPy-module is het `ndarray` gegevenstype. Dit gegevenstype implementeert homogene arrays (bv. *ints*, *floats*, *booleans*), met geoptimaliseerde functionaliteiten.

Rijen van getallen of matrices (zie Figuur 8.1) zullen we vaak in het geheugen onderbrengen als NumPy-arrays.

Figuur 8.1: Een rij van getallen (links) en een matrix van getallen (rechts).

Er zijn verschillende belangrijke verschillen tussen NumPy-arrays en de standaard Python-sequenties (bv. *lists*, *tuples*, *strings*):

- NumPy arrays hebben een **vaste grootte** bij het aanmaken, dit in tegenstelling tot Python lijsten die dynamisch kunnen groeien (cf. het gebruik van de methode `append()`). Het wijzigen van de grootte van een `ndarray` zal automatisch een nieuwe array creëren (nieuwe geheugenplaatsen toewijzen) en de originele array verwijderen (geheugenplaatsen vrijmaken).
- De elementen in een NumPy-array moeten van **hetzelfde gegevenstype** zijn (en elk element in de array moet evenveel geheugen innemen[1]).
- NumPy-arrays vergemakkelijken geavanceerde wiskundige en andere soorten operaties op grote hoeveelheden gegevens. Typisch worden dergelijke bewerkingen efficiënter en met minder code uitgevoerd dan door gebruik te maken van de Python ingebouwde sequenties.
- Een groeiend aantal wetenschappelijke en wiskundige Python-gebaseerde pakketten gebruikt NumPy-arrays. Hoewel deze typisch Python-sequentie-invoer ondersteunen, converteren ze dergelijke invoer naar NumPy-arrays voorafgaand aan de verwerking, en voeren ze vaak NumPy-arrays uit.

Geheugenplaats en snelheid zijn bijzonder belangrijk in wetenschappelijke berekeningen. Veronderstel bijvoorbeeld dat we elk element in een 1-dimensionale reeks willen vermenigvuldigen met het overeenkomstige element in een andere reeks van **dezelfde lengte**. Als de gegevens opgeslagen zijn in twee Python *lists*, `a` en `b` (cf. Figuur 8.1), kunnen we over de elementen itereren, het product berekenen en toevoegen aan een andere *list*:

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

Het bovenstaande levert een correct resultaat, maar als `a` en `b` elk miljoenen getallen bevatten, is deze manier van itereren in Python bijzonder **inefficiënt en tijdrovend**. We kunnen dezelfde taak veel sneller in de programmeertaal C uitvoeren met de volgende code:

```
for (int i = 0; i < rows; i++) {
    c[i] = a[i]*b[i];
}
```

[1]Merk op dat deze beperking omzeild kan worden door een array van objects aan te maken, die dan pointers zal bevatten naar de locatie van deze objecten. Dit wordt verder nog aangehaald, maar valt buiten het bestek van deze syllabus.

Dit bespaart alle overhead die betrokken is bij het interpreteren van de Python-code en het manipuleren van Python-objecten. Helaas is dit ten koste van de eenvoudige Pythoncode. Bovendien neemt het benodigde coderingswerk toe met de dimensionaliteit van de gegevens. Bijvoorbeeld, in het geval van een 2-dimensionale array, wordt de C-code uitgebreid naar:

```
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < cols; j++) {
        c[i][j] = a[i][j]*b[i][j];
    }
}
```

De voordelen van NumPy zijn:

(a) efficiënt **geheugengebruik**,

(b) **eenvoudige codering** en,

(c) een hoge **rekenefficiëntie**.

Element-by-element operaties (**elementsgewijs**) is de "standaard modus" bij `ndarray`'s[2]. Als de getallen in `a` en `b` in Figuur 8.1 bewaard werden in NumPy arrays kan de bewerking in bovenstaande code in NumPy vereenvoudigd worden tot:

```
c = a * b
```

Dit voorbeeld illustreert twee (krachtige) functionaliteiten van NumPy: **vectorisatie** en **elementsgewijze** bewerkingen.

Vectorisatie omvat het ontbreken van een expliciete lus (`for`, `while`, ...), indexering, ... in de broncode. Deze instructies vinden achter de schermen plaats in een geoptimaliseerde, vooraf gecompileerde C-code.

Enkele voordelen van vectorcode zijn:

- Vectorcode is beknopter en gemakkelijker te lezen.
- Minder regels code betekent meestal minder (kans op) fouten.
- De code lijkt meer op standaard wiskundige notatie: het maakt het gemakkelijker om typisch wiskundige constructies correct te coderen.

8.2 De basis van NumPy

NumPy's kernobject is de homogene multidimensionale array (zie: `https://numpy.org/doc/stable/reference/arrays.ndarray.html`). Het is een **tabel van elementen** (meestal getallen), allemaal van **hetzelfde type**, **geïndexeerd door een *tuple* positieve gehele getallen**.

[2]Deze operaties worden eigenlijk uitgevoerd door vooraf gecompileerde C-code)

In NumPy staat het **aantal assen** voor de dimensie(s) van de array:

- een **rij** (vector) met de getallen $[2, 6, -7]$ (zoals bv. `a` en `b` in Figuur 8.1) is 1-dimensionaal en heeft dus één as
- een **matrix** (bv. een 3×4-matrix) daarentegen is 2-dimensionaal en heeft dus 2 assen (bv. de matrix `C` in Figuur 8.1).

De klasse die de homogene multidimensionale array implementeert, is `ndarray`. Deze heeft verschillende **attributen** en **methoden**. Attributen zijn eigenschappen/kenmerken van een instantie van een klasse. Deze kunnen ingesteld of gewijzigd worden m.b.v. parameters die als argumenten worden meegegeven. We zullen dit in dit hoofdstuk illustreren.

Om gebruik te kunnen maken van de functionaliteiten van NumPy moeten we de module `numpy` eerst inladen met `import`. Om niet telkens de naam `numpy` te moeten gebruiken zullen we de module meestal importeren onder de naam `np` met de instructie:

```
>>> import numpy as np
```

8.2.1 Sequenties van getallen creëren: de functie `array()`

De functie `array()` wordt gebruikt om een array te creëren uit een ander object. Vaak zal dit een `list` zijn waarin de elementen worden opgelijst.

De syntax van de functie `array()` is:

```
array( object, dtype = None)
```

De betekenis van de parameters is:

- `object`: een (geneste) sequentie-object, bv. een (al dan niet geneste) lijst of tuple van getallen,
- `dtype`: een gegevenstype (bv. `int` of `float`). De elementen van de gegenereerde array zijn van dit gegevenstype. Indien `None` of niet gespecificeerd wordt het gegevenstype **afgeleid** uit de opgegeven elementen.

De output is een `ndarray`-object.

Het **type** van de resulterende array wordt afgeleid van het type van de elementen die je in `object` meegeeft. Zijn alle elementen *ints* dan zal de resulterende array ook type *int* hebben. Van zodra één element een *float* is, zal de resulterende array ook type *float* hebben.

Om bijvoorbeeld de 1-dimensionale array

$$v = \begin{bmatrix} 2, & 6, & -7 \end{bmatrix}$$

aan te maken, geven we de elementen als een *list* mee aan de functie `array()`:

```
>>> v = np.array( [2, 6, -7] )
>>> print(v)
[ 2  6 -7]

>>> type(v)
numpy.ndarray
```

Een frequent gemaakte fout bestaat erin het oproepen van `ndarray` zonder een *list* (of *tuple*) van elementen als argument. Dit geeft aanleiding tot een `TypeError`:

```
>>> v = np.array(2, 6, -7)      # FOUTIEF opgeroepen: zonder [ ]
Traceback (most recent call last):
  File "<stdin>", line 1, in <cell line: 1>
TypeError: array() takes from 1 to 2 positional arguments but 3
were given
```

Informatie over alle attributen van een `ndarray` vind je terug op `https://numpy.org/doc/stable/reference/arrays.ndarray.html#array-attributes`.

Enkele belangrijke attributen zijn:

- `dtype`: geeft het **gegevenstype** terug
- `ndim`: geeft de **dimensie** weer ("het aantal assen"),
- `shape`: geeft het **aantal rijen en kolommen** weer als *tuple*. Als het aantal assen gelijk is aan 1, dan geeft `shape` een tuple met slechts één element terug,
- `size`: geeft het **totaal aantal elementen** weer.
- `itemsize`: geeft de **lengte van één array element in bytes**. Vermits Numpy arrays **homogeen** zijn, is dit dezelfde voor elk array element.
- `nbytes`: het **aantal bytes** dat gebruikt wordt **in** het **geheugen**.

Deze attributen op van de array `v` kunnen we als volgt opvragen:

```
>>> v = np.array( [2, 6, -7] )
>>> v.dtype
dtype('int32')

>>> v.ndim
1        # 1 dimensie

>>> v.shape
(3,)     # 1-dimensionaal met 3 elementen

>>> v.size
```

```
3        # 3 elementen

>>> v.itemsize
4        # 4 bytes aan geheugen per element

>>> v.nbytes
12       # 4x3 = 12 bytes aan geheugen
```

De output `'int32'` bij `dtype` betekent dat het om een `int` gaat (een **geheel** getal) en dat elke *int* 32 bits[3] inneemt in het geheugen van de computer (32 bits is momenteel nog de standaard).

Met het attribuut `dtype` kan het type van de elementen expliciet meegegeven worden. Enkele mogelijke waarden werden in Tabel 8.1 opgenomen.

Tabel 8.1: Enkele mogelijke waarden voor `dtype`

dtype	**betekenis**
`int`	integer (gehele getallen)
`float`	decimale getallen
`bool`	logische waarden (`True` en `False`)
`str`	strings
`object`	willekeurige objecten

In de volge subsecties worden van elk type enkele voorbeelden gegeven.

8.2.1.1 Een 1-dimensionale array van `int`'s

Gehele getallen worden gezien als gehele getallen:

```
>>> v = np.array( [2, 6, -7], dtype = int)
>>> print(v)
[ 2  6 -7]

>>> v.dtype
dtype('int32')
```

Bij decimale getallen worden **valt het fractionele deel weg** zoals het volgend voorbeeld illustreert:

```
>>> v = np.array( [2.91, 6.3, -7.6], dtype = int)
>>> print(v)
[ 2  6 -7]              # GEEN afronding!

>>> v.dtype
dtype('int32')
```

[3]Vermits 8 bits = 1 byte, betekent dit dus 4 bytes aan geheugenruimte.

Andere integer gegevenstypes

Het is ook mogelijk arrays van andere integer gegevenstypes te creëren. Alle gegevenstypes in Tab. 8.1 zijn mogelijk.

Opgelet: bij getallen die niet in het bereik van het opgegeven gegevenstype vallen, treedt integer ***underflow*** of ***overflow*** op. Bekijken we hiertoe de volgende voorbeelden:

```
>>> v1 = np.array( [400, 5, 6], dtype = "int8")
>>> print(v1)
[-112    5    6]

>>> v2 = np.array( [-239, 5, 6], dtype = "uint8")
>>> print(v2)
[ 17   5   6]
```

In het eerste voorbeeld werd 400 vervangen door -112. Het bereik van het gegevenstype `int8` is het interval $[-128, 127]$. Dit zijn exact 256 mogelijke waarden. Van 400 wordt een geheel aantal keer 256 afgetrokken tot een waarde bekomen wordt die in $[-128, 127]$ ligt. Vermits $400 - 2 \times 256 = -112$ in dit interval ligt, krijgen we het bovenstaande resultaat.

In het tweede voorbeeld gebeurt iets gelijkaardigs: het bereik van het gegevenstype `uint8` is het interval $[0, 255]$. Dit zijn opnieuw exact 256 mogelijke waarden. Bij -239 wordt nu een geheel aantal keer 256 opgeteld tot een waarde bekomen wordt die in $[0, 255]$ ligt. Vermits $-239 + 256 = 17$ in dit interval ligt, krijgen we het bovenstaande resultaat.

Gelijkaardige redeneringen zijn ook van toepassing op de overige integer gegevenstypes.

8.2.1.2 Een 1-dimensionale array van `float`'s

Bij het gebruik van `dtype = float` worden alle getallen weergegeven als decimale getallen:

```
>>> f = np.array( [3.1, 0, 9.81, 5, -1.4], dtype = float)
>>> f.dtype
dtype('float64')

>>> f.shape
(5,)

>>> f.itemsize
8        # 8 bytes aan geheugen per element

>>> f.nbytes
40       # 5x8 = 40 bytes aan geheugen
```

De output `'float64'` bij `dtype` betekent dat het om een `float` gaat (een **decimaal** getal) en dat elke *float* 64 bits inneemt in het geheugen van de computer.

Voor decimale getallen worden standaard 64 bits (= 8 bytes) gebruikt. Wil men een array met enkelvoudige precisie, dan moet men de waarde `"float32"` opgeven[4]:

```
>>> f = np.array( [3.1, 0, 9.81, 5, -1.4], dtype = "float32")
>>> print(f)
[ 3.1   0.    9.81  5.   -1.4 ]

>>> f.dtype
dtype('float32')
```

Overflow en ***underflow***

Het bereik van het gegevenstype `float32` is veel kleiner is dan van het gegevenstype `float64`. Ook nu treedt *overflow* en *underflow* op als er te grote of te kleine getallen gebruikt worden. In tegenstelling tot bij *integer* gegevenstypes wordt nu `-inf` en `inf` gebruikt:

```
>>> f = np.array( [-10**40, 2, 3, 10**40], dtype = "float32")
>>> print(f)
[-inf   2.   3.  inf]
```

Vermits -10^{40} en 10^{40} niet kunnen voorgesteld worden met het gegevenstype `float32` wordt -10^{40} voorgesteld door `-inf` en wordt 10^{40} voorgesteld door `inf`.

8.2.1.3 Een 1-dimensionale array van `bool`'s

Bij het gebruik van **`bool`** wordt een 1 omgezet tot `True` en een 0 tot `False`:

```
>>> b = np.array( [1, 0, 0, 1], dtype = bool)
>>> print(b)
[ True False False  True]

>>> b.dtype
dtype('bool')

>>> b.itemsize
1        # 1 byte aan geheugen per element

>>> b.nbytes
4        # 4x1 = 4 bytes aan geheugen
```

Meer algemeen wordt **elke niet-nul waarde** omgezet tot een `True`:

```
>>> b = np.array( [-1, 0.0, 0, 3.7], dtype = bool)
>>> print(b)
[ True False False  True]
```

[4]Merk op dat er nu dubbele (of enkele) aanhalingstekens **moeten** gebruikt worden.

8.2.1.4 Een 1-dimensionale array van *strings*

Hoewel ongebruikelijk is het mogelijk om ook arrays van strings aan te maken:

```
>>> s1 = np.array(["a", "e", "i", "o", "u"], dtype = str)
>>> print(s1)
['a' 'e' 'i' 'o' 'u']

>>> s1.dtype
dtype('<U1')
```

Hoewel we `str` hebben opgegeven als waarde voor `dtype`, gebruikt Python de notatie `"<U1"`. Hierin staat `U` voor *Unicode* (zie Hoofdstuk 2).

Het cijfer 1 staat voor de lengte van de strings in de array `s1`. **Op basis van de langste string in de array (hier dus 1) bepaalt Python hoeveel geheugen er voor de opslag van één element zal gebruikt worden.** Vermits NumPy per karakter 4 bytes aan geheugenruimte gebruikt, is dit hier $4 \times 1 = 4$ bytes per array element. We kunnen dit als volgt opvragen:

```
>>> s1.itemsize
4     # 4 bytes per element (= 4x1)

>>> s1.nbytes
20    # = 4x5 (er zijn 5 elementen in s1)
```

Het kleiner dan teken `<` staat voor *little-endian*. Onder *endianness* of bytevolgorde verstaat men de manier waarop woorden, die zelf uit meerdere bytes bestaan, in het computergeheugen worden opgeslagen. Het gaat daarbij om de volgorde in het geheugen van de bytes die samen een woord vormen. Wordt de meest significante byte het eerst geschreven, dan spreekt men van *big-endian*, wordt de minst significante byte eerst geschreven, dan spreekt men van *little-endian*.

Opmerking: de strings hoeven niet steeds dezelfde lengte hebben als in het vorige voorbeeld:

```
>>> s2 = np.array(["banaan", "appel", "peer", "citroen"], dtype = str)
>>> print(s2)
['banaan' 'appel' 'peer' 'citroen']

>>> S2.dtype
dtype('<U7')
```

De langste string heeft nu lengte 7. Bijgevolg zullen er in de array `s2` per element $7 \times 4 = 28$ bytes gebruikt worden:

```
>>> s2.itemsize
28    # 28 bytes per element (= 7x4)

>>> s2.nbytes
112   # = 28x4 (er zijn 4 woorden in s2)
```

8.2.1.5 Een 1-dimensionale array van *objecten*

Eenmaal een array is aangemaakt, staat vast hoeveel geheugenruimte er per element zal gebruikt worden. In het geval van een array van strings leidt dit in sommige gevallen tot **afkapping**. Beschouw het volgende voorbeeld waarbij in de array `s2` uit de vorige sectie een van de elementen wordt vervangen door een langere string:

```
>>> s2
array(["banaan", "appel", "peer", "citroen"], dtype = "<U7")
>>> s2[1] = "appelsien"   # vervang 'appel' door 'appelsien'

>>> print(s2)
['banaan' 'appelsi' 'peer' 'citroen']
```

Vermits elk element van de array `S2` een geheugenblok met een lengte van exact 7 bytes inneemt, wordt de string `appelsien` **afgekapt** na het zevende karakter.

Om dergelijke afkappingen te vermijden, is het aangewezen om `object` mee te geven als waarde voor het attribuut `dtype`:

```
>>> s = np.array(["banaan", "appel", "peer", "citroen"], dtype = object)
>>> print(s)

>>> s[1] = "appelsien"
>>> print(s)
['banaan' 'appelsien' 'peer' 'citroen']
```

Opmerking: als we de elementen als een **lijst in een lijst** meegeven, dan krijgen we een 2-dimensionale array:

```
>>> x = np.array( [ [2, 6, -7] ] )   # bemerk de extra haken

>>> x.ndim
2        # 2-dimensionaal

>>> x.shape
(1, 3)   # 1 rij, 3 kolommen
```

Multidimensionale arrays worden **meer in detail besproken in Sectie 8.2.10.**

8.2.2 Sequenties van getallen: de functie `arange()`

In Sectie 4.4.3 hebben we de functie `range()` geïntroduceerd om een *sequentie* van **gehele** getallen in een bepaald interval (*range*) te maken:

```
>>> list(range(10, 50, 5)) % start- en stopwaarde en stapgrootte opgegeven
[10, 15, 20, 25, 30, 35, 40, 45]
```

Niet-gehele argumenten zijn niet toegelaten:

```
>>> range(1, 2.1, 0.2)
TypeError: 'float' object cannot be interpreted as an integer
```

Het NumPy analogon van deze functie is `arange()` aan (zie: `https://numpy.org/doc/stable/reference/generated/numpy.arange.html`). Deze functie geeft een array van *ints* of *floats* terug.

De signatuur van de functie `arange()` is als volgt:

```
arange([start, ]stop, [step, ] dtype = None)
```

Enkel de parameter `stop` is **verplicht**, de parameters tussen rechte haken zijn **optioneel**.

De betekenis van de parameters is:

- `start`: (optioneel) `int` of `float`, **begin**punt van het interval. Indien niet meegegeven wordt gestart bij 0 (de *default* waarde).
- `stop`: `int` of `float`, **eind**punt van het interval (**niet** opgenomen in de output).
- `step`: (optioneel) `int` of `float`, de **stap**grootte. Indien niet meegegeven dan is de stapgrootte 1 (de *default* waarde).
- `dtype`: een gegevenstype (bv. `int` of `float`). De elementen van de gegenereerde array zijn van dit gegevenstype. Indien `None` of niet gespecificeerd wordt het gegevenstype **afgeleid** uit de andere parameters. Van zodra één van de parameters een `float` is, is het gegevenstype ook `float`. Naast `int` en `float` is ook `bool` mogelijk (zie 8.2.10).

De output is een 1-dimensionaal `ndarray`-object van **gelijk gespreide** waarden.

8.2.2.1 Sequenties van `int`'s

- Een 1-dimensionale array met stopwaarde `10`:

  ```
  >>> x = np.arange(10)       % enkel stopwaarde opgegeven: start = 0
  >>> print(x)
  [0 1 2 3 4 5 6 7 8 9]

  >>> x.dtype       % gegevenstype opvragen
  dtype('int32')
  ```

  De array `x` kan ook als volgt gecreëerd worden:

```
>>> x = np.arange(0, 10)    % start- en stopwaarde opgegeven
>>> print(x)
[0 1 2 3 4 5 6 7 8 9]
```

- Een 1-dimensionale array met startwaarde `10`, stopwaarde `30` (de stopwaarde wordt **niet** opgenomen in de array) en met stapgrootte `5`. De waarden in de array zijn `int`'s:

```
>>> y = np.arange(10, 30, 5)
>>> print(y)
[10 15 20 25]
```

- Een 1-dimensionale array met startwaarde `10`, stopwaarde `-12` (**niet** opgenomen in de array) en stapgrootte `-2`. De waarden in de array zijn `int`'s:

```
>>> z = np.arange(10, -12, -2)
>>> print(z)
[ 10   8   6   4   2   0  -2  -4  -6  -8 -10]
```

8.2.2.2 Sequenties van `float`'s

Sequenties van `float`'s kunnen gecreëerd worden door ofwel een `float` als startwaarde, stopwaarde of stapgrootte op te geven, of de parameter `dtype` in te stellen op `float`.

- Een 1-dimensionale array met (default) startwaarde `0`, stopwaarde `10` (niet opgenomen in de array) en (default) stapgrootte `1`. De waarden in de array zijn hier `float`'s:

```
>>> f = np.arange(10, dtype = float)
>>> print(f)
[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
```

  De array `f` kan ook als volgt gecreëerd worden:

```
>>> f = np.arange(10.0)      % float als stopwaarde
```

- Een 1-dimensionale array met startwaarde `-1`, stopwaarde `2` (niet opgenomen in de array) en stapgrootte `0.3`. De waarden in de array zijn floats (omdat de stapgrootte als *float* werd ingegeven):

```
>>> g = np.arange(-1, 2, 0.3)
>>> print(g)
[-1.  -0.7 -0.4 -0.1  0.2  0.5  0.8  1.1  1.4  1.7]
```

8.2.3 Sequenties van getallen: de functie `linspace()`

De functie `linspace()` (zie `https://numpy.org/doc/stable/reference/generated/numpy.linspace.html#numpy.linspace`) creëert een array van gelijk gespreide waarden over een interval.

De signatuur van de functie `linspace()` is:

```
linspace(start, stop, num = 50, endpoint = True, retstep = False, dtype = None)
```

De betekenis van de parameters is:

- `start`: `int` of `float`, **begin**punt van het interval. Indien niet meegegeven wordt gestart bij `0` (de *default* waarde).
- `stop`: `int` of `float`, **eind**punt van het interval. Het eindpunt wordt default **wel** opgenomen in de gegenereerde output (zie item `endpoint`).
- `num`: `int`, geeft aan hoeveel punten de output moet bevatten. De default is 50.
- `endpoint`: `True` of `False`, de default is `True`. Indien niet meegegeven dan wordt de **stopwaarde wel opgenomen**.
- `restep`: `True` of `False`, geeft aan of de stapgrootte moet geretourneerd worden als tweede output.
- `dtype`: `int` of `float`. De elementen van de gegenereerde array zijn van dit gegevenstype. Indien `None` of niet gespecificeerd is het gegevenstype default `float`.

De output is een 1-dimensionaal `ndarray`-object van **gelijk gespreide** waarden. Enkele voorbeelden:

```
>>> x = np.linspace(1, 10, num = 10)
>>> print(x)
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
>>> x.dtype
dtype('float64')                # default gegevenstype is float

>>> y = np.linspace(1, 10, num = 10, dtype = int)
>>> print(y)
[ 1  2  3  4  5  6  7  8  9 10]
>>> y.dtype
dtype('int32')

>>> z = np.linspace(2, 3, num = 5)
>>> print(z)
[ 2.    2.25  2.5   2.75  3.  ]
```

Opmerking: de stapgrootte `step` kan ook berekend worden met de volgende formule:

$$\text{step} = \frac{\text{stop} - \text{start}}{\text{num} - 1}\,.$$

Toegepast op het laatste voorbeeld vinden we `step = (3 - 2)/(5 - 1) = 1/4 = 0.25`, hetgeen overeenstemt met de output hierboven.

8.2.4 De functies `zeros()` en `ones()`

De functie `zeros()` creëert een array gevuld met **nullen**. De functie `ones()` daarentegen creëert een array gevuld met **enen**. Het `dtype` van de elementen is standaard `float64` (*floats*). Het oproepen van deze functies gebeurt als volgt:

```
zeros(shape, dtype = None)
```

```
ones(shape, dtype = None)
```

De parameters zijn:

- `shape`: een `int` of sequentie (*list* of *tuple*) van `int`'s met de dimensies, bv. `5` (1-dimensionaal) of `[3, 5]` (2-dimensionaal) of `(3, 5)` (ook 2-dimensionaal).
- `dtype`: het gegevenstype (bv. `int`, `float` of `bool`). De elementen van de gegenereerde array zijn van dit gegevenstype. Indien `None` is de default `float`.

De output is een `ndarray` object van nullen of enen, met de dimensies opgegeven in `shape`. Bij *default* zijn de elementen *floats*.

Geven we voor de `shape` één `int` mee, dan is de array 1-dimensionaal:

```
>>> z = np.zeros(5)
>>> print(z)
[ 0.  0.  0.  0.  0.]
>>> z.ndim
1

>>> z2 = np.ones(5)
>>> print(z2)
[ 1.  1.  1.  1.  1.]
```

Opmerking: een array met uitsluitend nullen of enen kan ook aangemaakt worden met de functie `array()`:

```
>>> z = np.array([0]*5)
>>> print(z)
[0 0 0 0 0]

>>> z.dtype
dtype('int32')
```

Geven we voor de `shape` de *list* `[3, 5]` mee, dan is de array 2-dimensionaal (zie Sectie 8.2.10) met 3 rijen en 5 kolommen:

```
>>> Z = np.zeros([3, 5])
>>> print(Z)
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]

>>> Z.ndim
2

>>> I = np.ones([3, 5])
>>> print(I)
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

8.2.4.1 Creatie van 1-dimensionale arrays: oefeningen

Opdracht 8.1 (`numpy_arrays_creeren.py`)
Maak de volgende NumPy-arrays aan:

- `[  1   3   5   7 ...  197 199]`
- `[ 50  49  48 ... -48 -49 -50]`
- `[0 1 0 1 ... 0 1 0 1]` (100 elementen)
- `[-1  0  1  0 -1 0 ... -1  0  1  0]` (100 elementen)
- `[-10.  -9.5 -9.  -8.5 -8.  -7.5 ....  9.   9.5 10. ]`
- `[-5.       -4.94949495 -4.8989899 ... -0.05050505 0.  ]` (100 elementen)
-

```
[[1 1 1]
 [1 0 1]
 [1 1 1]]
```

-

```
[[False  True False]
 [ True  True  True]
 [False  True False]]
```

-

```
[[1 1 1 1 1]
 [1 0 0 0 1]
 [1 0 2 0 1]
 [1 0 0 0 1]
 [1 1 1 1 1]]
```

-

```
[[  1    1    1  ...    1    1    1],
 [ 50   49   48  ...  -48  -49  -50]]
```

met shape `(2, 101)`

8.2.5 Indexering, slicing en iteratie bij 1-dimensionale arrays

Indexering, slicing en iteratie bij 1-dimensionale arrays is analoog als bij Pythonsequenties (lijsten, zie Sectie 7.4).

In wat volgt, bespreken we het indexeren, slicen en itereren a.d.h.v. een voorbeeld. Als voorbeeld maken we een array van getallen aan met de functie `arange()`:

```
>>> a = np.arange(1, 20, 2)
>>> print(a)
[ 1  3  5  7  9 11 13 15 17 19]
```

8.2.5.1 Indexering

Het indexeren begint bij index 0 zoals geïllustreerd in Fig. 8.2. Het eerste element van de array `a` heeft index 0, het laatste element heeft index `len(a)-1`.

array **a**	1	3	5	7	9	11	13	15	17	19
index	0	1	2	3	4	5	6	7	8	9

Figuur 8.2: Indexering van een 1D array.

Het element met index `i` in de array `a` kan opgevraagd worden met

```
a[i]
```

waarbij `i = 0, 1, ..., a.shape[0]-1` en `a.shape[0]` de lengte van `a` voorstelt[a].

[a]Voor de lengte van de array werd hier `a.shape[0]` gebruikt maar we konden, vermits het om een 1-dimensionale array gaat, ook `len(a)` gebruikt hebben.

Enkele voorbeelden:

```
>>> a[0]        # eerste element: index 0,
1

>>> a[4]        # vijfde element: index 4
9
```

Negatieve indices zijn ook mogelijk:

`a[-i]`

waarbij `i = 1, ..., a.shape[0]`.

Het resultaat is steeds een **scalair** en dus **geen array**.

Enkele voorbeelden:

```
>>> a[-1]       # laatste element in de array
19

>>> a[-2]       # voorlaatste element in de array
17
```

Opmerking: als de index `i` buiten het bereik zou vallen, wordt een `IndexError` opgeworpen:

```
>>> a[11]
IndexError: index 11 is out of bounds for axis 0 with size 10
```

Dergelijke en andere fouten worden meer in detail behandeld in Sectie 9.9 van het volgende hoofdstuk.

Voor meer gedetailleerde uitleg verwijzen we naar
`https://docs.scipy.org/doc/numpy/reference/arrays.indexing.html`.

8.2.5.2 Slicing

Slicing betekent het selecteren van elementen vanaf een startindex tot een eindindex. Er bestaan meerdere vormen van slicing.

Slices van de vorm `[i:j]`

Slicing gebeurt met dezelfde operator (`[]`) als indexering. In plaats van een enkele, gehele index `i`, wordt nu gebruik gemaakt van een **bereik**:

```
a[ i:j ]
```

waarbij geldt dat `i < j` en `j < a.shape[0]`.

Het resultaat is een array met elementen met indices `i, i+1, i+2, ..., j-1`. Het element met index `j` behoort dus **niet** tot de *slice*.

Een voorbeeld:

```
>>> sel = a[2:7]     # elementen met index 2 tem index 6 (= 7-1)
>>> print(sel)
[ 5  7  9 11 13]
```

Opmerking: als de eindindex `j` groter of gelijk wordt aan de lengte van de array, dan wordt er, in tegenstelling tot bij indexering, **geen `IndexError`** opgeworpen. In plaats daarvan wordt er geselecteerd tot aan het einde van de array:

```
>>> print(a)
[ 1  3  5  7  9 11 13 15 17 19]
>>> sel = a[8:13]
>>> print(sel)
[17 19]
```

Slices van de vorm `[i:]`, `[j:]` en `[:]`

- `a[ i: ]`: slice die start bij positie `i` en eindigt bij het einde van de array
- `a[ :j ]`: slice die start bij positie `0` en eindigt bij positie `j-1`
- `a[ : ]`: begin tot einde (*copy slice*)

Er geldt steeds dat `i, j < a.shape[0]`.

Enkele voorbeelden:

```
>>> sel1 = a[5:]    # elementen met indices 5 tot einde
>>> print(sel1)
[11 13 15 17 19]

>>> sel2 = a[:5]    # elementen met indices 0, tem 4 (= 5 - 1)
>>> print(sel2)
[1 3 5 7 9]
```

```
>>> sel3 = a[:]     # begin tot einde (copy slice)
>>> print(sel3)
[ 1  3  5  7  9 11 13 15 17 19]
```

Slices van de vorm `[i:j:k]`
In het voorgaande werden elementen geselecteerd die aaneensluitend zijn: het verschil tussen opeenvolgende posities was steeds 1. Dit hoeft niet altijd zo te zijn.

Voor een array `a` en drie gehele getallen i (**start**index), j (**eind**index) en k (**stap**grootte) zal de expressie `a[i:j:k]` de elementen selecteren met de volgende indices

`i, i + k, i + 2k, i + 3k, ..., < j`

Het element met index `j` behoort wederom **niet** tot de slice.

Opmerkingen:

- Indien de **startindex afwezig** is, dan wordt deze impliciet vervangen door **nul**.
- Indien de **eindindex afwezig** is, dan wordt deze impliciet vervangen door de **lengte** van de array.
- In de slice `[i:j:k]` kan, op voorwaarde dat `i > j` (of beide afwezig zijn) de **stapgrootte (`k`) negatief zijn**. De array zal dan **van achter naar voor** overlopen worden.

Enkele voorbeelden:

```
>>> print(a)
[ 1  3  5  7  9 11 13 15 17 19]

>>> sel1 = a[2:9:3]  # start = 2 : einde = 9 : stap = 3
>>> print(sel1)
[ 5 11 17]

>>> sel2 = a[0::2]   # alle elementen met even indices
>>> print(sel2)
[ 1  5  9 13 17]

>>> sel3 = a[6:0:-2] # elementen met indices 6, 4 en 2 (0 niet inbegrepen)
>>> print(sel3)
[13  9  5]

>>> sel4 = a[::-1]   # array in omgekeerde volgorde
>>> print(sel4)
[19 17 15 13 11  9  7  5  3  1]
```

8.2.5.3 Fancy indexing

Fancy indexing is conceptueel eenvoudig: het betekent het doorgeven van een lijst met indices om meerdere (al dan niet dezelfde) elementen van een array tegelijk op te vragen.

Indexering van de vorm `[ l ]`
Meerdere elementen (in willekeurige volgorde) van de array `a` kunnen tegelijkertijd geselecteerd door een *list* van indices `l` mee te geven tussen rechte haken:

```
a[ l ]
```

met `l` een lijst met indices is die de waarden

```
-a.shape[0], ..., -1, 0, 1, ..., a.shape[0]-1
```

kunnen aannemen.

Het resultaat is een 1-dimensionale array met de elementen met indices in `l`.

Enkele voorbeelden:

```
>>> a = np.arange(1, 20, 2)
>>> print(a)
[ 1  3  5  7  9 11 13 15 17 19]

>>> a[ [5, 1, 7, 4] ]   # 6de, 2de, 8ste en 5de element
array([11,  3, 15,  9])

>>> a[ [0, -2] ]     # eerste en voorlaatste element
array([ 1, 17])

>>> a[ [1, 1, 1] ]   # 3 keer het tweede element
array([3, 3, 3])
```

8.2.5.4 Itereren over 1-dimensionale arrays

Het itereren over de elementen van een array kan op meerdere manieren:

- **zonder** gebruik van een **index**,
- via **indexering**

Vermits een `ndarray` *iterable* object is, kan deze gebruikt worden in een `for`-statement.

Met behulp van het volgende stukje code kan over elk element van de array `a` geïtereerd worden:

```
a = np.arange(1, 20, 2)
for elem in a:
    print(elem)
```

Het resultaat van de uitvoer van deze code is:

```
1
3
5
...
19
```

De ... wijzen erop dat een deel van de output niet werd overgenomen.

We kunnen ook itereren over elk element van een 1-dimensionale array **via indexering:**

```
for i in range(a.shape[0]):
    print(i, a[i])
```

Het resultaat van de uitvoer van deze code is:

```
0 1
1 3
2 5
...
9 19
```

Opmerking: Voor de lengte van de array werd hier `a.shape[0]` gebruikt, maar we konden ook `len(a)` geschreven hebben: `a` is immers een 1-dimensionale array.

8.2.6 Basisbewerkingen met `ndarray`'s

Bewerkingen kunnen uitgevoerd worden met een array en een scalair, of met meerdere arrays. We illustreren dit a.d.h.v. de volgende 1-dimensionale arrays `a` en `b`:

```
>>> a = np.array([1, 2, 5, 10, 20])
>>> b = np.array([1, 2, 3, 4, 5])
```

8.2.6.1 Rekenkundige bewerkingen met **één** 1-dimensionale array en een scalair

Rekenkundige bewerkingen (`+`, `-`, `*`, `/` en `**`) met een array en een scalair gebeuren **elementsgewijs**: dus op elk element van de array. Elke bewerking retourneert een nieuwe array met eenzelfde *shape* als van de array in de bewerking.

Enkele voorbeelden:

```
>>> a + 4      # bij elk element wordt 4 opgeteld
array([ 5,  6,  9, 14, 24])

>>> a*2        # elk element wordt vermenigvuldigd met 2
array([ 2,  4, 10, 20, 40])

>>> a**2       # elk element wordt verheven tot de 2de macht
array([  1,   4,  25, 100, 400])

>>> 2**a       # 2 wordt verheven tot elk getal in a
array([       2,        4,       32,     1024, 1048576], dtype = int32)
```

8.2.6.2 Rekenkundige bewerkingen met **meerdere 1-dimensionale arrays**

Rekenkundige bewerkingen (`+`, `-`, `*`, `/` en `**`) met meerdere arrays gebeuren eveneens **elementsgewijs**. Een belangrijke voorwaarde opdat de bewerkingen mogelijk zouden zijn, is dat de *shapes* (i.e. de lengtes) **identiek** moeten zijn.

```
>>> a - b
array([ 0,  0,  2,  6, 15])

>>> a/b
array([1.        , 1.        , 1.66666667, 2.5       , 4.        ])

>>> a**b
array([      1,       4,     125,   10000, 3200000])
```

8.2.6.3 Vergelijkingsoperatoren

Bij toepassing van vergelijkingsoperatoren (`<`, `>`, `==`, `<=`, `>=`, en `!=`) op een array en een scalair worden de elementen eveneens **elementsgewijs** vergeleken. De elementen in de geretourneerde array zijn van het gegevenstype `bool`.

Enkele voorbeelden:

```
>>> a > 3   # enkel posities waar elementen > 3 worden True
array([False, False,  True,  True,  True])

>>> b < 5   # enkel posities waar elementen < 5 worden True
array([ True,  True,  True,  True, False])

>>> a == 2  # enkel posities waar elementen == 2 worden True
array([False,  True, False, False, False])
```

Bij toepassing van vergelijkingsoperatoren op meerdere arrays worden de elementen eveneens **elementsgewijs** vergeleken. Een belangrijke voorwaarde opdat de bewerkingen mogelijk zouden zijn, is dat ook nu de *shapes* (i.e. de lengtes) **identiek** moeten zijn.

Enkele voorbeelden:

```
>>> res1 = a > b
>>> print(res1)
[False False  True  True  True]
```

Vermits enkel de laatste drie elementen uit a groter zijn dan die uit b, staat op die plaatsen `True`.

```
>>> res3 = a == b
>>> print(res3)
[ True  True False False False]
```

Vermits de eerste twee elementen uit a gelijk zijn aan die uit b staat op die plaatsen `True`.

8.2.6.4 Logische operatoren

De logische operatoren `&` (and), | (or), $\wedge$ (xor) en $\sim$ (not) worden eveneens elementsgewijs toegepast.

We illustreren dit a.d.h.v. de volgende logische arrays:

```
>>> r = np.array([True, False, True, False, True])
>>> s = np.array([False, True, True, True, False])
```

Enkele voorbeelden:

```
>>> print(r & s)  # True als overeenkomstig element in r en s True is
[False False  True False False]

>>> print(r | s)  # True van zodra 1 overeenkomstig element in r of s True is
[ True  True  True  True  True]

>>> print( r ^ s) # True als precies 1 overeenkomstig element True is
[ True  True False  True  True]

>>> print(~r)     # True wordt False, False wordt True
[False  True False  True False]
```

8.2.6.5 Logische indexering (*logical indexing*)

Met logische indexering kunnen we in een array elementen selecteren o.b.v. een logische array.

Veronderstel bijvoorbeeld dat we in een array `v` de elementen willen selecteren die groter zijn dan 5. Bekijk hiertoe het onderstaande voorbeeld:

```
>>> v = np.array([10, -1, 5, 2, 4, 7, 8, 5, 6])
>>> res = v > 5
>>> print(res)
[ True False False False False  True  True False  True]

>>> groter = v[res]
>>> print(groter)
[10  7  8  6]
```

Merk op dat met `v[res]` enkel die elementen uit `v` worden geselecteerd waarvoor de overeenkomstige elementen uit `res` gelijk zijn aan `True` zijn. Het resultaat is steeds een 1-dimensionale array.

Opmerking: door instructies te **combineren** kan hetzelfde resultaat bekomen worden:

```
>>> groter = v[v > 5]
>>> print(groter)
[10  7  8  6]
```

8.2.6.6 Wijzigen van elementen

Alle `ndarray`'s zijn *mutable*, hetgeen betekent dat de elementen gewijzigd kunnen worden na creatie (cf. *lists*).

We kunnen elk element afzonderlijk wijzigen:

```
>>> a = np.arange(1, 20, 2)
>>> print(a)
[ 1  3  5  7  9 11 13 15 17 19]

>>> a[1] = 57
>>> print(a)
[ 1 57  5  7  9 11 13 15 17 19]   # 3 --> 57

>>> a[4] = a[4]**2
>>> print(a)
[ 1 57  5  7 81 11 13 15 17 19]   # 9 --> 9^2 =  81

>>> a[-2] = 0
>>> print(a)
[ 1 57  5  7 81 11 13 15  0 19]   # 17 --> 0
```

We kunnen ook een selectie van elementen wijzigen in een 1-dimensionale array. Die selectie kunnen we maken met start/eind indices `[i:j]` zoals bij slicing. Opdat deze wijziging zou kunnen uitgevoerd worden, moet:

- de *shape* van de vervangarray overeenstemmen met de *shape* van het geselecteerde deel, of
- de vervangarray uit slechts één element bestaan (laatste voorbeeld hieronder).

```
>>> print(a)
[ 1 57  5  7 81 11 13 15  0 19]

>>> a[0:3] = a[-3:]  # eerste 3 elementen gelijkgesteld aan laatste 3
>>> print(a)
[15  0 19  7 81 11 13 15  0 19]

>>> a[::2] = -1      # elementen met even index --> -1
>>> print(a)
[-1  0 -1  7 -1 11 -1 15 -1 19]
```

8.2.7 View vs. copy

De NumPy array is een datastructuur die uit twee delen bestaat: de aaneengesloten databuffer met de eigenlijke data-elementen en de metadata die informatie over de databuffer bevat. De metadata omvat het gegevenstype en andere belangrijke informatie die helpt bij het manipuleren van de `ndarray`.

Het is mogelijk om rechtstreeks toegang te krijgen tot de interne gegevensbuffer m.b.v. een weergave (***view***) zonder gegevens te kopiëren (***copy***). Dit zorgt voor een goede performantie maar kan ook voor ongewenste problemen zorgen als de gebruiker niet weet hoe dit werkt. Daarom is het belangrijk om het verschil tussen deze twee termen te kennen en te weten welke bewerkingen een kopie retourneren en welke een *view*.

8.2.7.1 View

Door te slicen (zie Sectie 8.2.5.2) ontstaat een view van de originele array, en wijzigt de databuffer niet. Dit betekent dat alle wijzigingen die op een view doorgevoerd worden, de originele array ook wijzigen.

Beschouw een slice van de array `a`:

```
>>> a = np.arange(1, 20, 2)
>>> print(a)
[ 1   3   5   7   9  11  13  15  17  19]
>>> c = a[1:5]
>>> print(c)
[3 5 7 9]
```

Het resultaat van een slice-operatie is **geen nieuwe array** maar een view van de originele array. Wijzigingen in de slice wijzigen dus ook de originele array (en omgekeerd):

```
>>> c[1] = 8
>>> print(c)
[3 8 7 9]                                    # c is gewijzigd
>>> print(a)
[ 1   3   8   7   9  11  13  15  17  19]   # a is OOK gewijzigd
```

8.2.7.2 Copy

Om de slices los te koppelen van het origineel kan een (*shallow*) kopie van de slice gemaakt worden met de functie `array()`:

```
>>> a = np.arange(1, 20, 2)
>>> print(a)
[ 1  3  5  7  9 11 13 15 17 19]
>>> b = np.array(a[1:5])          # maak SHALLOW kopie van de slice
>>> print(b)
[3 5 7 9]
>>> b[0] = 25                     # wijzig element in b
>>> print(b)
[25 3 5 7 9]                      # b is gewijzigd
>>> print(a)
[ 1  3  5  7  9 11 13 15 17 19]   # a is NIET gewijzigd
```

Merk op dat de variabelen `a` en `b` niet langer verwijzen naar **hetzelfde** object. Wijzigingen in de copy array hebben **geen invloed** op de originele array.

Een tweede manier om een *shallow* kopie te maken van een array is gebruik te maken van de functie `copy()`: met `b = np.copy(a)` wordt een kopie gemaakt van `a`.

8.2.8 NumPy Functies

NumPy biedt een uitgebreid aanbod aan wiskundige functies zoals bv. goniometrische, exponentiële, logaritmische, enz. In NumPy worden deze functies "universele functies" (*ufunc*) genoemd (zie `https://numpy.org/doc/stable/reference/ufuncs.html`). Binnen NumPy werken deze functies **elementsgewijs** in op een array.

Een beperkt overzicht is terug te vinden in Tabel 8.2. Voor een volledig overzicht van deze *ufuncs*, zie `https://numpy.org/doc/stable/reference/ufuncs.html#available-ufuncs`.

Tabel 8.2: Overzicht van enkele NumPy functies. De input `x` stelt een array voor.

Wiskundige constanten	
`pi`	het getal π
`e`	Euler's constante, grondtal van de natuurlijke logaritme
Wiskundige operatoren	
`power(x, m)`	machtsverheffing van `x` tot `m` (`m` kan een array zijn, met **dezelfde** *shape* als `x`)
`sqrt(x)`	worteltrekking van `x` (equivalent van `x**0.5`)
*Goniometrische functies en hun inverse (**`x`** in **radialen**)*	
`sin(x)`	de sinus van `x`
`cos(x)`	de cosinus van `x`
`tan(x)`	de tangens van `x`
`arcsin(x)`	de inverse sinus van `x`
`arccos(x)`	de inverse cosinus van `x`
Exponentiële en logaritmische functies	
`exp(x)`	de exponentiële functie van `x` (grondtal e)
`log(x)`	de natuurlijke logaritme met grondtal e (ln) van `x`
`log10(x)`	de tiendelige logaritme ($\log_{10}$) van `x`
Afrondingsfuncties	
`fix(x)`	`x` afronden naar het volgende geheel getal in de richting van 0
`round(x, n)`	`x` afronden tot op `n` cijfers
`remainder(x, a)`	rest na **gehele** deling van `x` door `a` (`a` kan een array zijn, met dezelfde *shape* als `x`)
Aggregatie functies	
`sum(x)`	som van de elementen van `x`
`min(x)`	minimum van de elementen van `x`
`max(x)`	maximum van de elementen van `x`
`mean(x)`	gemiddelde van de elementen van `x`
Zoekfuncties	
`argmin(x)`	index van het minimum van `x`
`argmax(x)`	index van het maximum van `x`
`nonzero(x)`	index van alle niet-nul elementen van `x` (`False` is ook nul)
`unique(x)`	de unieke elementen van `x` (van klein naar groot)
Logische functies	
`any(x)`	geeft `True` terug als minstens één element van `x` verschillend is van 0
`all(x)`	geeft `True` terug als alle elementen van `x` verschillend zijn van 0
Andere functies	
`abs(x)`	absolute waarde van `x`
`sign(x)`	resulteert in `1` voor `x` groter dan nul, `0` voor `x` gelijk aan nul, en `-1` voor `x` kleiner dan nul
Vergelijkingsfuncties	
`array_equal(x, y)`	geeft `True` terug als de arrays `x` en `y` identiek zijn

Al deze functies nemen één of twee `ndarray`'s als input en geven een **nieuwe** (1- of meerdimensionale) `ndarray`, `float` of `bool` terug.

We beperken ons hier tot het illustreren van enkele van deze functies:

```
>>> x = np.arange(0, 6)
>>> print(x)
[0 1 2 3 4 5]

>>> y = np.round(np.exp(x), 3) # gebruik van de exp() en round() functies
>>> print(y)
[  1.      2.718   7.389  20.086  54.598 148.413]
```

Merk op dat we steeds `np` moeten plaatsen vóór de functienamen.

```
>>> hoeken = np.arange(10, 60, 10)
>>> print(hoeken)
[10 20 30 40 50]

>>> radialen = hoeken*np.pi/180 # omzetten naar radialen
>>> print(radialen)
[0.17453293 0.34906585 0.52359878 0.6981317  0.87266463]

>>> cosinus = np.cos(radialen)
>>> print(cosinus)
[0.98480775 0.93969262 0.8660254  0.76604444 0.64278761]
```

De functies `argmin()` en `argmax()`

```
>>> x = np.array([10, -1, 5, -2, -5, 0, 3, -4, 7, -5])
>>> print(x)
[10 -1  5 -2 -5  0  3 -4  7 -5]

>>> print(np.argmin(x))
4    # index = 4 dus positie = 5
>>> print(np.argmax(x))
0    # index = 0 dus positie = 1
```

Merk op dat, hoewel -5 twee keer voorkomt, de functie `argmin()` de index van het **eerste** minimum terug geeft.

Enkele voorbeelden met `any()` en `all()`:

```
>>> print(x)
[10 -1  5 -2 -5  0  3 -4  7 -5]

>>> np.any(x)
True       # x bevat niet-nul elementen
```

```
>>> np.all(x)
False      # niet elk element in x is verschillend van 0
```

De functies `any()` en `all()` zijn vooral handig bij arrays van `bool`'s:

```
# komt 5 voor in x?
>>> np.any(x == 5)
True

# komt 2 voor in x?
>>> np.any(x == 2)
False
```

Zoals eerder aangegeven, aanvaarden sommige functies twee arrays als input, bv. `remainder()`:

```
>>> v = np.array([10, -1, 5, 2, -4])
>>> print(v)
[10 -1  5  2 -4]

>>> d = np.arange(2, 11, 2)
>>> print(d)
[ 2  4  6  8 10]

>>> rest = np.remainder(v, d)
>>> print(rest)
[0 3 5 2 6]
```

Vermits `-1 = -1×4 + 3` is de rest gelijk aan 3. Idem voor de rest na deling van -4 door 10: `-4 = -1×10 + 6` en is de rest dus gelijk aan 6.

De functie `array_equal()`

Met de functie `array_equal()` kan gecontroleerd worden of twee arrays **identiek** zijn (i.e. **dezelfde dimensies en inhoud** hebben). Enkele voorbeelden:

```
>>> x = np.array([[1, 5, 3, 4, 2]])
>>> y = np.array([[1, 5, 3, 4, 2]])
>>> z = np.array([[1, 5, 3, 7, 0]])
>>> u = np.array([[5, 3, 4]])
>>> np.array_equal(x, y)
True    # dimensies komen overeen, inhoud ook

>>> np.array_equal(x, z)
False   # dimensies komen overeen, inhoud niet (de 7 en 0 in z)

>>> np.array_equal(x, u)
False   # dimensies komen niet overeen (1x5 vs 1x3)
```

De functie `unique()`

Met behulp van de functie `unique()` kunnen de **unieke elementen** in een array bepaald worden:

```
>>> x = np.array([4, 0, -1, 3, 0, 4, 0, -3, -1, 6])
>>> print(x)
[ 4  0 -1  3  0  4  0 -3  -1  6]

>>> uniek = np.unique(x)
>>> uniek
array([-3 -1  0  3  4  6])
```

Merk op dat de functie `unique()` een **van klein naar groot** gesorteerde array teruggeeft.

Werken we met een array van strings, dan retourneert de functie `unique()` een **alfabetisch** gesorteerde array met de unieke strings:

```
>>> s = np.array(["e", "u", "a", "i", "i", "o", "a", "u", "o", "a"])
>>> print(s)
['e' 'u' 'a' 'i' 'i' 'o' 'a' 'u' 'o' 'a']

>>> uniek = np.unique(s)
>>> print(uniek)
['a' 'e' 'i' 'o' 'u']
```

Toepassing: Latijnse vierkanten

Een Latijns vierkant is een vierkant van n rijen en n kolommen, gevuld met n verschillende symbolen waarvan elk precies één keer per rij en ook één keer per kolom voorkomt. De matrices M en N hieronder zijn twee voorbeelden van Latijnse vierkanten, de matrix O echter niet:

$$M = \begin{pmatrix} 1 & 2 \\ 2 & 1 \end{pmatrix} \qquad N = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 2 \end{pmatrix} \qquad O = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \\ 3 & 1 & 4 \end{pmatrix}$$

Opdracht 8.2

Schrijf een functie `isGeldigeRij()` die als argument een $1 \times n$ array van getallen aanvaardt en controleert of elk van de cijfers 1 t.e.m. n precies één keer voorkomen in deze rij. Maak hiervoor gebruik van de functies `unique()` en `array_equal()`.

```
>>> rij1 = np.array([[1, 5, 3, 4, 2]])  # n = 5
>>> isGeldigeRij(rij1)
True      # elk cijfervan 1 t.e.m. 5 komt voor

>>> rij2 = np.array([[3, 1, 4]])        # n = 3
>>> isGeldigeRij(rij2)
False     # 2 komt niet voor, 4 zou er niet mogen zijn
```

De functie `nonzero()`

In bepaalde toepassingen is men geïnteresseerd in de rij- en kolomindices van array-elementen die aan bepaalde voorwaarden voldoen. Men wil bijvoorbeeld de indices bepalen van de niet-nul elementen in een 1-dimensionale array. Een eenvoudige maar omslachtige manier om deze indices te bepalen wordt hieronder gegeven:

```
v = np.array([4, 0, -1, 3, -2, -1, 0, -3, 1, 6])
indices = []
for i in range(len(v)):
    if v[i] != 0:
        indices.append(i)
print(indices)
```

Met als resultaat:

```
[0, 2, 3, 4, 5, 7, 8, 9]
```

Echter, in NumPy is er de functie `nonzero()` beschikbaar. Deze functie aanvaardt een `ndarray` als argument en geeft de indices terug van de niet-nul elementen in de opgegeven array:

```
>>> ind = np.nonzero(v)
>>> ind
(array([0, 2, 3, 4, 5, 7, 8, 9], dtype = int64),)

>>> type(ind)
tuple

>>> len(ind)
1
```

Merk op dat, in het geval van 1-dimensionale arrays, de functie `nonzero()` een **tuple** retourneert met slechts één array (vandaar de **komma**). De array extraheren kan als volgt m.b.v. *tuple unpacking* (zie Sectie 5.5.3):

```
>>> ind, = np.nonzero(v % 2 == 0)
>>> ind
array([0, 2, 3, 4, 5, 7, 8, 9], dtype = int64)

>>> type(ind)
numpy.ndarray

>>> len(ind)
8
```

Door een komma te plaatsen wordt het eerste element van de tuple toegekend aan `ind` en wordt de rest genegeerd. Hetzelfde resultaat bekomen we met indexering:

```
>>> res = np.nonzero(v % 2 == 0)
>>> ind = res[0]
>>> ind
array([0, 2, 3, 4, 5, 7, 8, 9], dtype = int64)
```

De functie `nonzero()` kan ook in combinatie met logische expressies gebruikt worden. In het volgende voorbeeld worden de indices van de **even**[5] **getallen** uit `v` bepaald:

```
>>> ind2, = np.nonzero(v % 2 == 0)
>>> ind2
array([0, 1, 4, 6, 9], dtype = int64)
```

Het resultaat van de expressie `v % 2 == 0` is een logische array die dezelfde lengte heeft als `v` en uitsluiten `True` en/of `False` bevat. Elke `True` wordt door `nonzero()` gezien als een niet-nul waarde, elke `False` daarentegen wordt gezien als 0. Met de gevonden indices kunnen we vervolgens:

- de **even getallen** uit de array **selecteren**:

```
>>> even = v[ind2]    # equivalent met v[v % 2 == 0]
>>> print(even)
[ 4  0 -2  0  6]
```

- het **aantal even getallen** in de array bepalen:

```
>>> n = len(ind2)     # equivalent met len(v[v % 2 == 0])
>>> n
5
```

Opmerking: de functie `nonzero()` aanvaardt ook lists en tuples als argument:

```
>>> lijst = [4, 0, -1, 3, -3, -1, 0, -3, 1, 3]  # lijstversie van de array v
>>> indices, = np.nonzero(lijst)
>>> indices
array([0, 2, 3, 4, 5, 7, 8, 9], dtype = int64)
```

In Sectie 8.2.15 zullen we voorbeelden zien met 2-dimensionale arrays.

8.2.9 Vectorisatie: voorbeelden

In Sectie 8.2 werd vectorisatie geïntroduceerd. Hier illustreren we hoe we, gebruikmakende van NumPy, vectorisatie kunnen toepassen.

[5]Het getal 0 wordt als even beschouwd.

8.2.9.1 Voorbeeld 1: priemgetallen

In Opdracht 4.14 (Hoofdstuk 4) werd gevraagd om een programma te schrijven die nagaat of een getal al dan niet een priemgetal voorstelt. Om te testen of een getal `n` een priemgetal is, kan voor `i = 2, ..., int(n**(1/2))` bepaald worden of `i` een deler is. Indien geen deler gevonden wordt, kan besloten worden dat n een priemgetal is. Voorbeelden van priemgetallen zijn: $2, 3, 5, 7, 11, 13, ...$

Een mogelijke oplossing, **zonder gebruik van NumPy**, is de volgende:

```
n = int(input("Geef een natuurlijk getal: "))
i = 1
while i < int(n**(1/2)):
    i = i + 1  # 2 is de eerste deler die gecheckt wordt
    if n % i == 0:
        print(n, "is geen priemgetal,", i, "is de kleinste deler.")
        break
else: # wordt uitgevoerd als while-lus volledig doorlopen wordt!
    print(n, "is wel een priemgetal.")
```

Maken we gebruik van de functionaliteiten (elementsgewijze berekeningen, logische indexering, ...) van NumPy, en willen we enkel nagaan of `n` een priemgetal is of niet, dan krijgen we de volgende implementatie zonder `while`-lus en indexering:

```
1 n = int(input("Geef een natuurlijk getal: "))
2 delers = np.arange(2, int(n**(1/2)) + 1) # +1 om eindpunt te includeren
3 rest = n % delers
4 resultaat = np.all(rest != 0)
```

De code kan verder ingekort worden door instructies samen te nemen (nesten):

- op regel 4 vervangen we in het rechterlid de variabele `rest` door het rechterlid van regel 3,
- daarna vervangen we de variabele `delers` door het rechterlid van regel 2.

We krijgen dan de volgende heel compacte code:

```
1 n = int(input("Geef een natuurlijk getal: "))
2 resultaat = np.all(n % np.arange(2, int(n**(1/2)) + 1) != 0)
```

8.2.9.2 Voorbeeld 2: perfecte getallen

In Opdracht 4.10 (Hoofdstuk 4) werd gevraagd om een programma te schrijven die nagaat of een getal al dan niet een perfect getal voorstelt: een natuurlijk getal waarvoor de som van zijn gehele delers, met uitzondering van het getal zelf, gelijk is aan het natuurlijk getal. Voorbeelden van perfecte getallen zijn: $6 (= 1 + 2 + 3)$, $28 (= 1 + 2 + 4 + 7 + 14)$, 496 en 8128.

Een mogelijke oplossing, zonder gebruik van NumPy, is de volgende:

```
getal = int(input("Geef een natuurlijk getal: "))
som = 1
for i in range(2, getal):
    if getal % i == 0:
        som = som + i
if som == getal:
    print(getal, "is een perfect getal.")
else:
    print(getal, "is GEEN perfect getal.")
```

Maken we gebruik van NumPy, dan krijgen we de volgende implementatie:

```
1 n = int(input("Geef een natuurlijk getal: "))
2 delers = np.arange(1, n)
3 rest = n % delers
4 somDelers = np.sum(delers[rest == 0])
5 resultaat = (somDelers == n)
```

De code kan ook nu verder ingekort worden door instructies te nesten:

```
1 n = int(input("Geef een natuurlijk getal: "))
2 delers = np.arange(1, n)
3 resultaat = (np.sum(delers[n % delers == 0]) == n)
```

8.2.9.3 Vectorisatie: oefeningen

Opdracht 8.3

In Opdracht 5.3 (Hoofdstuk 5) werd gevraagd om twee functies `gemiddelde()` en `std()` te schrijven die respectievelijk het gemiddelde en de standaarddeviatie van de getallen uit het bestand `data_waarden.txt` berekenen.

Het gemiddelde van deze getallen kan berekend worden met de volgende formule:

$$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

waarbij $x_1, \ldots, x_n$ de waarden voorstellen.

De standaarddeviatie kan berekend worden met de volgende formule:

$$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}\,.$$

Maak gebruik van de functionaliteiten van NumPy om de volgende opdrachten uit te voeren:

1. Lees de waarden uit het bestand `data_waarden.txt` in als een array van floats met de functie `loadtxt()`[6] uit de module `numpy`:

[6]Deze functie wordt in detail besproken in Sectie 8.2.16.1.

```
waarden = np.loadtxt("data_waarden.txt")
```

2. Schrijf een functie `gemiddelde()` die een array van floats (zoals de variabele `waarden`) als argument aanvaardt en het gemiddelde `gem` van deze waarden retourneert (3 cijfers na de komma):

```
>>> gem = gemiddelde([7, 5, 3, 9])
>>> gem
6.0
```

3. Schrijf een functie `std()` die een array van floats (zoals de variabele `waarden`) als argument aanvaardt en de standaarddeviate `stdev` van deze waarden retourneert (3 cijfers na de komma). Maak gebruik van de functie `gemiddelde()`:

```
>>> stddev = std([7, 5, 3, 9])
>>> stddev
2.582
```

Indien je deze functies correct geïmplementeerd hebt, bekom je de volgende output:

```
>>> import numpy as np
>>> waarden = np.loadtxt("data_waarden.txt")
>>> gem = gemiddelde(waarden)
>>> stdev = std(waarden)
>>> print(gem)
13.623
>>> print(stdev)
2.079
```

8.2.10 Multidimensionale arrays

Om een 2-dimensionale `ndarray` aan te maken, geven we een lijst mee met de rijen als sublijsten (*list* van *lists*) aan de functie `array()`. De 3×3-matrix M

$$M = \begin{pmatrix} 9 & 8 & 7 \\ 4 & 5 & 6 \\ 3 & 2 & 1 \end{pmatrix}$$

kan als volgt gecreëerd worden als een 2-dimensionale array:

```
>>> M = np.array( [[9, 8, 7], [4, 5, 6], [3, 2, 1]] )
>>> M
array([[9, 8, 7],
       [4, 5, 6],
       [3, 2, 1]])
```

Net zoals bij 1-dimensionale arrays kunnen we ook nu de attributen `dtype`, `ndim`, `shape`, `size`, `itemsize` en `nbytes` opvragen:

```
>>> M.ndim
2        # 2 dimensies

>>> M.shape
(3, 3)   # 3 rijen en 3 kolommen

>>> M.size
9        # 3x3 = 9 elementen
```

Met het attribuut `dtype` kan het type van de elementen expliciet meegegeven worden. Enkele mogelijke waarden voor dit attribuut werden in Tabel 8.1 opgenomen.

In hetgeen volgt, worden van elk type enkele voorbeelden gegeven.

Een 2-dimensionale array van *ints*

```
>>> I = np.array( [[1, 2], [2, 3], [3, 4]], dtype = int )
>>> print(I)
[[1 2]
 [2 3]
 [3 4]]

>>> I.dtype
dtype('int32')
```

Een 2-dimensionale array van *floats*

```
>>> F = np.array( [[1, 2], [2, 3], [3, 4]], dtype = float)
>>> print(F)
[[ 1.  2.]
 [ 2.  3.]
 [ 3.  4.]]

>>> F.dtype
dtype('float64')
```

Een 2-dimensionale array van *booleans*

```
>>> B = np.array([[1, 0, 1], [0, 1, 0]], dtype = bool)
>>> print(B)
[[ True False  True]
 [False  True False]]

>>> B.dtype
dtype('bool')
```

8.2.10.1 3-dimensionale arrays

Voor de meeste toepassingen in deze cursus volstaan 1- en 2-dimensionale arrays. Er bestaan echter ook 3-dimensionale arrays. Een typisch voorbeeld hiervan zijn kleurafbeeldingen. Deze worden voorgesteld door een 3-dimensionale array waarbij elk kleurkanaal (rood, groen of blauw) een 2-dimensionale array voorstelt.

In NumPy kunnen 3-dimensionale arrays op verschillende manieren gecreëerd worden: bijvoorbeeld met de functie `array()`, en de functies `zeros()` en/of `ones()`.

Creatie van een n-dimensionale array met de functie `array()`

Het volstaat om een *list* met 2-dimensionale arrays mee te geven als input:

```
>>> D = np.array([ [[1, 2], [3, 4]], [[5, 6], [7, 8]], [[9, 10], [11, 12]] ])
>>> print(D)
[[[ 1  2]
  [ 3  4]]

 [[ 5  6]
  [ 7  8]]

 [[ 9 10]
  [11 12]]]
```

De verschillende 2-dimensionale submatrices kunnen als volgt geselecteerd worden:

```
>>> print(D[0,:,:])   # eerste 2x2 submatrix
[[1 2]
 [3 4]]
```

Merk op dat met de **eerste** index de submatrices worden geïndexeerd.

Creatie van een 3-dimensionale array met de functie `zeros()`

Als we aan de functie `zeros()` een triplet meegeven als `shape`, dan wordt het eerste element van het triplet beschouwd als de derde dimensie. Een 3-dimensionale array met drie 2×2 arrays met uitsluitend nullen kan als volgt aangemaakt worden:

```
>>> Z = np.zeros((3, 2, 2), dtype = int)
>>> print(Z)
[[[0 0]
  [0 0]]

 [[0 0]
  [0 0]]

 [[0 0]
  [0 0]]]
```

8.2.11 Indexering, slicing en iteratie bij 2-dimensionale arrays

Voor meer gedetailleerde uitleg verwijzen we naar
`https://docs.scipy.org/doc/numpy/reference/arrays.nditer.html` voor multidimensionale arrays.

In wat volgt, bespreken we het indexeren, slicen en itereren a.d.h.v. een voorbeeld.

Als voorbeeld maken we een 2-dimensionale array van getallen aan met de functie `arange()`:

```
>>> M = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
>>> print(M)
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
>>> print(M.shape[0])   # aantal rijen
4
>>> print(M.shape[1])   # aantal kolommen
3
```

8.2.11.1 Indexering

De indexering bij 2-dimensionale arrays is gelijkaardig aan de indexering bij 1-dimensionale arrays, maar gebeurt met 2 indices gescheiden door komma's.

Elk element van de array `M` kan afzonderlijk opgevraagd worden met

`M[i,j]`

waarbij

- `i = -M.shape[0], ..., 0, 1, ..., M.shape[0]-1`, en
- `j = -M.shape[1], ..., 0, 1, ..., M.shape[1]-1`.

Enkele voorbeelden:

```
>>> M[0,2]   # element op rij-index 0, kolom-index 2
3
>>> M[1,1]   # element op rij-index 1, kolom-index 1
5
>>> M[-2,2]  # element op rij-index -2, kolom-index 2 (-2: voorlaatste rij)
9
```

8.2.11.2 Slicen

Ook het slicen van 2-dimensionale arrays is gelijkaardig aan het 1-dimensionaal geval, maar nu met start- en eindindices voor zowel de rijen als de kolommen.

Om de verschillende vormen van slicing te illustereren maken we de volgende 2-dimensionale array `A` aan:

$$\texttt{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

```
>>> A = np.array( [[1, 2, 3, 4], [5, 6, 7, 8], \
... [9, 10, 11, 12], [13, 14, 15, 16]])
>>> print(A)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
>>> A.shape[0]
4
>>> A.shape[1]
4
```

Slices van de vorm `[r1:r2, k1:k2]`

`A[ r1:r2, k1:k2 ]`

waarbij

- `r1` de startrij-index is
- `r2` de index is van de eindrij + 1
- `k1` de startkolom-index is
- `k2` de index is van de eindkolom + 1

Er moet gelden dat

- `r1 < r2`,
- `k1 < k2`,
- `r2 < A.shape[0]`, en
- `k2 < A.shape[1]`.

Opmerking: de elementen van de rij met index `r2` en van de kolom met index `k2` behoren dus **niet** bij de selectie.

We selecteren uit `A` de eerste 3 rijen en eerste 2 kolommen:

```
>>> A[0:3, 0:2]
>>> print(B)
[[ 1  2]
 [ 5  6]
 [ 9 10]]
```

Slices van de vorm `[r1:r2:stapr, k1:k2:stapk]`
De geselecteerde elementen hoeven **niet aaneensluitend** te zijn.

Voor een 2-dimensionale array `A` zal de expressie

`A[r1:r2:r, k1:k2:k]`

de elementen selecteren met de volgende **rij**indices:

`r1, r1 + r, r1 + 2r, r1 + 3r, ..., < r2`

en de volgende **kolom**indices:

`k1, k1 + k, k1 + 2k, k1 + 3k, ..., < k2`

Opmerkingen:

- Indien `r1` afwezig is, dan wordt deze impliciet vervangen door nul.
- Indien `r2` afwezig is, dan wordt deze impliciet vervangen door `A.shape[0]`.
- Indien `k1` afwezig is, dan wordt deze impliciet vervangen door nul.
- Indien `k2` afwezig is, dan wordt deze impliciet vervangen door `A.shape[1]`.
- Op voorwaarde dat `r1 > r2` (of beide afwezig zijn) en dat `k1 > k2` (of beide afwezig zijn) kunnen de **stapgroottes `r` en `k` negatief zijn**.

Enkele voorbeelden:

```
# rij-indices 0 t.e.m. 3 in stappen van 2
# kolom-indices 0 t.e.m. 2 in stappen van 2
>>> print(A[0:4:2, 0:3:2])
[[ 1  3]
 [ 9 11]]

# rij-index 1 t.e.m. laatste rij-index
# kolom-index 0 t.e.m. kolom-index 2
>>> print(A[1:, :3])
[[ 5  6  7]
```

```
 [ 9 10 11]
 [13 14 15]]

# rijen in omgekeerde volgorde doorlopen
>>> print(A[::-1, ::1])
[[13 14 15 16]
 [ 9 10 11 12]
 [ 5  6  7  8]
 [ 1  2  3  4]]
```

8.2.11.3 Fancy indexing

Fancy indexing kan ook gebruikt worden bij 2-dimensionale arrays.

Indexering van de vorm `[ k, l ]`
Meerdere elementen (in willekeurige volgorde) van de 2-dimensionale array `A` kunnen tegelijkertijd geselecteerd door een *list* van indices mee te geven tussen rechte haken:

`A[ k, l ]`

met `k` een **lijst** met indices is die de waarden

`-A.shape[0], ..., -1, 0, 1, ..., A.shape[0]-1`

kunnen aannemen, en met `l` een **lijst** met indices is die de waarden

`-A.shape[1], ..., -1, 0, 1, ..., A.shape[1]-1`

kunnen aannemen. Dergelijke lijsten kunnen uit één element of meerdere (**al dan niet gelijke**) indices bestaan.

Het resultaat is een array met de elementen met indices in `k` en `l`.

Enkele voorbeelden:

```
>>> A = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], \
[13, 14, 15, 16]])
>>> print(A)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

>>> A[ [3, 1], [0, 2] ]        # elementen A[3,0] en A[1,2]
array([13,  7])
```

```
>>> A[ [0, 1, 2], [2, 2, 2] ]  # eerste 3 elementen van 3e kolom
array([ 3,  7, 11])
```

Opmerking: *fancy indexing* kan ook in combinatie met *slicing* gebruikt worden:

```
>>> A[ :, [-1, 0] ]     # laatste en eerste kolom
array([[ 4,  1],
       [ 8,  5],
       [12,  9],
       [16, 13]])
```

8.2.11.4 Itereren over 2-dimensionale arrays

Het itereren over de elementen van een 2-dimensionale array kan, net zoals bij 1-dimensionale arrays, op meerdere manieren:

- **zonder** gebruik van een **index**,
- via **indexering**

Stel dat we uit `A` alle 2×2 matrices van aaneensluitende rijen en kolommen willen selecteren, en op het scherm printen door middel van iteratie.

De matrix `A`, de eerste drie en de laatste 2×2 matrices van `A` zijn:

$$\texttt{A} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix} \quad \begin{pmatrix} 2 & 3 \\ 6 & 7 \end{pmatrix} \quad \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix} \quad \ldots \quad \begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix}$$

Het volgende stukje code realiseert dit:

```
for i in range(A.shape[0]-1):
    for j in range(A.shape[1]-1):
        print(A[i:i+2, j:j+2])
```

Het resultaat van de uitvoer van deze code is:

```
[[1 2]        # 1ste 2x2 submatrix
 [5 6]]
[[2 3]        # 2e 2x2 submatrix
 [6 7]]
[[3 4]        # 3e 2x2 submatrix
 [7 8]]
[[ 5  6]
 [ 9 10]]
[[ 6  7]
```

```
 [10 11]]
[[ 7  8]
 [11 12]]
[[ 9 10]
 [13 14]]
[[10 11]      # 8ste 2x2 submatrix
 [14 15]]
[[11 12]      # 9e en laatste 2x2 submatrix
 [15 16]]
```

Merk op dat het stel `i,j` telkens de **linker bovenhoek** voorstelt van de selectie `A[i:i+2, j:j+2]`. Belangrijk is dat onze selectie `A[i:i+2, j:j+2]` met de indices `i`, `i+2`, `j` en `j+2` niet buiten de grenzen treedt van de rij- en kolomindices van onze matrix A:

- met `for i in range(A.shape[0]-1)` gaat de rij-index van `0` **t.e.m.** `A.shape[0]-2`,
- met `for j in range(A.shape[1]-1)` gaat de kolom-index van `0` **t.e.m.** `A.shape[1]-2`,

zodat `A[i:i+2, j:j+2]` steeds een 2×2 matrix is die binnen de grenzen van de matrix A valt.

Opdracht 8.4 (`array_slicing1.py`)
Vervang de grenzen van de geneste `for`-lus hierboven als volgt:

```
for i in range(A.shape[0]):
    for j in range(A.shape[1]):
        print(A[i:i+2, j:j+2])
```

en voer de code opnieuw uit. Je zou het volgende moeten bekomen:

```
[[0 1]
 [4 5]]
[[1 2]
 [5 6]]
[[2 3]
 [6 7]]
[[3]      # 2x1 matrix
 [7]]
...
[[14 15]] # 1x2 matrix
[[15]]    # 1x1 matrix
```

We stellen vast dat er nu ook 1×2, 2×1 en 1×1 matrices worden geselecteerd. Als `i = 0` en `j = 3`, zal `A[i:i+2,j:j+2]` gelijk zijn aan `A[0:2,3:5]`. In de slice `3:5` wordt er geselecteerd vanaf kolom-index 3 t.e.m. kolomindex 4, maar deze laatste valt buiten het bereik $[0, 3]$. Net zoals bij 1-dimensionale arrays wordt er ook hier **geen** `IndexError` opgeworpen[7], maar wordt er **geselecteerd tot waar mogelijk** is:

[7] Zie Sectie 8.2.5.2

```
>>> sel = A[0:2,3:5]      # i = 0 en j = A.shape[1]-1
>>> print(sel)
[[3]
 [7]]
```

Gelijkaardige redeneringen gelden voor het geval `i = A.shape[0]-1, j = 0` en voor het geval `i = A.shape[0]-1, j = A.shape[1]-1`:

```
>>> sel1 = A[3:5,0:2]      # i = A.shape[0]-1 en j = 0
>>> print(sel1)
[[12 13]]

>>> sel2 = A[3:5,3:5]      # i = A.shape[0]-1 en j = A.shape[1]-1
>>> print(sel2)
[[15]]
```

Opdracht 8.5 (`array_slicing2.py`)

Maak een array van de 5×6-matrix

$$\mathtt{A} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 & 16 & 17 \\ 18 & 19 & 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 & 28 & 29 \end{pmatrix}$$

1. Schrijf een geneste `for`-lus die alle 3×3-matrices toont van aaneensluitende rijen en kolommen. Gebruik daarvoor rij- en kolom-indices `i,j` die de **linker bovenhoek** voorstellen van de geselecteerde 3×3-matrices. **Tip**: maak gebruik van de code uit Opdracht 8.4 en pas deze aan. Volgend resultaat moet je bekomen:

   ```
   [[ 0  1  2]   # 1ste 3x3 submatrix
    [ 6  7  8]
    [12 13 14]]
   [[ 1  2  3]   # 2e 3x3 submatrix
    [ 7  8  9]
    [13 14 15]]
   ...
   [[15 16 17]   # 12e en laatste 3x3 submatrix
    [21 22 23]
    [27 28 29]]
   ```

2. Schrijf een geneste `for`-lus die alle 3×3-matrices toont van aaneensluitende rijen en kolommen. Gebruik daarvoor rij- en kolom-indices `i,j` die het **midden** voorstellen van de geselecteerde 3×3-matrices. Je zou hetzelfde resultaat moeten bekomen als in de vorige opdracht.

   Tip: indien `i,j` de middenindices voorstellen van een 3×3-matrix binnen `A`, dan wordt deze 3×3 matrix geselecteerd met `A[i-1:i+2,j-1:j+2]`.

```
[[ 0  1  2]   # 1ste 3x3 submatrix
 [ 6  7  8]
 [12 13 14]]
[[ 1  2  3]   # 2e 3x3 submatrix
 [ 7  8  9]
 [13 14 15]]
...
[[15 16 17]   # 12e en laatste 3x3 submatrix
 [21 22 23]
 [27 28 29]]
```

3. We willen nu het selecteren van submatrices veralgemenen tot $m \times n$ matrices. Schrijf een kort script dat de gebruiker vraagt om twee dimensies (rij- en kolomdimensies) in te geven (gescheiden door een spatie) en vervolgens alle mogelijke aaneensluitende submatrices (met de ingegeven rij- en kolomdimensies) uit matrix `A` toont. Indien één of beide ingegeven dimensies groter is dan overeenkomstige dimensie(s) van `A`, dan moet een gepaste melding verschijnen. Het volgende illustreert de werking van je script:

```
>>> Geef rij- en kolomdimensies: 1 4
[[0 1 2 3]]
[[1 2 3 4]]
[[2 3 4 5]]
[[6 7 8 9]]
...
[[26 27 28 29]]

>>> Geef rij- en kolomdimensies: 3 2
[[ 0  1]
 [ 6  7]
 [12 13]]
[[ 1  2]
 [ 7  8]
 [13 14]]
...
[[16 17]
 [22 23]
 [28 29]]

>>> Geef rij- en kolomdimensies: 6 1
Een of meerdere dimensies te groot!
```

8.2.12 Vormmanipulatie van arrays

Een array heeft een vorm (*shape*) die bepaald wordt door het aantal elementen langs elke as:

```
>>> print(A)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
>>> A.shape
(4, 4)
```

De *shape* van `A` is `(4, 4)` wat neerkomt op 4 rijen en 4 kolommen.

De vorm van een array kan op verschillende manieren aangepast worden. We beperken ons hier tot:

- **omvormen** (reshapen) met `reshape()`,
- **transponeren** met `T`,
- **ontrafelen** met `ravel()`,
- **concateneren** met `concatenate()`, en
- **randen** toevoegen.

Deze drie methoden retourneren nieuwe gewijzigde array's, maar wijzigen de oorspronkelijke array niet. In wat volgt, beperken we ons tot 2-dimensionale arrays.

8.2.12.1 Omvormen met `reshape()`

Met de functie `reshape()` kunnen de dimensies van een array aangepast worden.

De signatuur is

```
reshape(a, newshape, order = "C")
```

De betekenis van de parameters is:

- `a`: de array die moet omgevormd worden,
- `newshape`: een *tuple* of *list* met de nieuwe dimensies. Als één van de dimensies als -1 wordt opgegeven, wordt de andere dimensie bepaalt o.b.v. de andere dimensie en het aantal elementen van de array (i.e. de *size*),
- `order`: een string met één karakter (`"C"`, `"F"` of `"A"`) die bepaalt hoe de nieuwe array moet opgevuld worden. De default is `"C"` (rij per rij).

De functie retourneert een array met de nieuwe dimensies.

```
>>> A2 = np.reshape(A, (2, 8))
>>> print(A2)
[[ 1  2  3  4  5  6  7  8]
```

```
 [ 9 10 11 12 13 14 15 16]]

>>> A3 = np.reshape(A, (4, 3))
Traceback (most recent call last):
    A3 = np.reshape(A, (4, 3))
ValueError: cannot reshape array of size 16 into shape (4,3)
```

Merk op dat:

- de nieuwe array **rij per rij** wordt opgevuld, en
- het product van argumenten van `reshape()` moet overeenstemmen met de *size* van de array (het aantal elementen).

Indien we -1 als één van de dimensies opgegen, wordt de andere dimensie bepaald door de overige dimensie en de *size* van `A`. Ook nu moet het product van de dimensies gelijk zijn aan de *size*:

```
>>> A4 = np.reshape(A, (2, -1))
>>> print(A4)
[[ 1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15 16]]

>>> A5 = np.reshape(A, (-1,3))
Traceback (most recent call last):
    A5 = np.reshape(A, (-1,3))
ValueError: cannot reshape array of size 16 into shape (3)
```

In het tweede voorbeeld is het onmogelijk om een array met 16 elementen om te vormen naar een nieuwe array met 3 kolommen. Vandaar de foutmelding.

Opmerking: vaak wordt de functie `reshape()` **in combinatie** gebruikt met functie `arange()`.

De matrix `A` kan bijvoorbeeld ook als volgt aangemaakt worden:

```
>>> A = np.arange(1, 17).reshape((4, 4))
>>> print(A)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]
```

Opmerking: de functie `resize()` is gelijkaardig aan `reshape()` maar `resize()` **wijzigt de oorspronkelijke array**:

```
>>> A.resize(2, 8)
>>> print(A)
[[ 1  2  3  4  5  6  7  8]
 [ 9 10 11 12 13 14 15 16]]
```

Voor meer informatie over de functie `reshape()`, zie:
`https://numpy.org/doc/stable/reference/generated/numpy.reshape.html`.

8.2.12.2 Transponeren met `T`

```
>>> print(A.T)
[[ 1  5  9 13]
 [ 2  6 10 14]
 [ 3  7 11 15]
 [ 4  8 12 16]]
```

Voor meer informatie over `T`, zie:
`https://numpy.org/doc/stable/reference/generated/numpy.ndarray.T.html`.

8.2.12.3 Ontrafelen met `ravel()`

De functie `ravel()` zal alle elementen van een array achter elkaar plaatsen in een nieuwe, 1-dimensionale array.

De signatuur is:

```
ravel(a, order = "C")
```

De betekenis van de parameters is:

- `a`: de array die ontrafeld moet worden, en
- `order`: een string met één karakter (`"C"`, `"F"`, `"A"` of `"K"`) die bepaalt hoe de array moet doorlopen worden (rij per rij, of kolom per kolom). De default is `"C"` (rij per rij).

De functie retourneert een 1-dimensionale array:

```
>>> B = np.ravel(A)
>>> print(B)
[ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16]
>>> B.shape
(16,)          # 1-dimensionaal

>>> np.ravel(A, order = "F")  # kolom per kolom
[ 1  5  9 13  2  6 10 14  3  7 11 15  4  8 12 16]
```

Voor meer informatie over `ravel`, zie:
`https://numpy.org/doc/stable/reference/generated/numpy.ravel.html`.

Opdracht 8.6 (`numpy_array_reshape.py`)
Maak de volgende numpy-arrays aan met behulp van `reshape()`. In elk van de gevallen is het niet de bedoeling van letterlijk alle getallen over te typen maar eerst een 1-dimensionale array aan te maken o.b.v. de instructies uit Sectie 8.2.

-
```
>>> A = np.arange(..., ..., ...).reshape((..., ...))
>>> print(A)
[[ 1  3  5  7  9 11 13]
 [15 17 19 21 23 25 27]
 [29 31 33 35 37 39 41]
 [43 45 47 49 51 53 55]]
```

-
```
>>> B = np.arange(..., ..., ...).reshape((..., ...))
>>> print(B)
[[ 2  4  6  8 10 12]
 [14 16 18 20 22 24]
 [26 28 30 32 34 36]
 [38 40 42 44 46 48]
 [50 52 54 56 58 60]]
```

-
```
>>> C = np.arange(..., ..., ...).reshape((..., ...))
>>> print(C)
[[-60 -55 -50 -45 -40]
 [-35 -30 -25 -20 -15]
 [-10  -5   0   5  10]
 [ 15  20  25  30  35]
 [ 40  45  50  55  60]]
```

-
```
>>> D = np.array(......).reshape((..., ...))
>>> print(D)
[[1 0 1 0 1]
 [0 1 0 1 0]
 [1 0 1 0 1]
 [0 1 0 1 0]
 [1 0 1 0 1]]
```

8.2.12.4 Concateneren met `concatenate()`

In sommige toepassingen worden arrays met al dan niet verschillende dimensies horizontaal of vertikaal samengevoegd (geconcateneerd) tot één array. Dit samenvoegen kan met de functie `concatenate()`.

De signatuur van de functie `concatenate()` is:

```
concatenate((a1, a2, ...), axis = 0, dtype = None)
```

De betekenis van de parameters is:

- `(a1, a2, ...)`: een tuple met arrays,
- `axis`: geeft aan volgens welke dimensie (as) er geconcatenereerd moet worden. De waarde 0 komt overeen met de kolommen (vertikaal), de waarde 1 met de rijen (horizontaal),
- `dtype`: het gegevenstype van de resulterende array (zie Tabel 8.1)

De functie retourneert een `ndarray` met de elementen uit de samengevoegde arrays.

Concatenatie zal enkel mogelijk zijn als de **dimensies compatibel** zijn:

- **1-dimensionale arrays** kunnen **uitsluitend horizontaal** geconcateneerd worden. Vermits dergelijke arrays slechts één as hebben, kan enkel volgens deze as geconcateneerd worden,
- bij **vertikale** (**onder** elkaar) concatenatie van 2-dimensionale arrays moet het aantal **kolommen** identiek zijn,
- bij **horizontale** (**naast** elkaar) concatenatie van 2-dimensionale arrays moet het aantal **rijen** identiek zijn.

We zullen het gebruik van de functie `concatenate()` illustreren a.d.h.v. de volgende 1-en 2-dimensionale en arrays:

$$a = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \qquad b = \begin{pmatrix} 4 & 5 & 6 \end{pmatrix} \qquad c = \begin{pmatrix} 7 & 8 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} \qquad C = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

We maken deze arrays eerst aan:

```
>>> a = np.array([1, 2, 3])
>>> print(a)
[1 2 3]
>>> b = np.array([4, 5, 6])
>>> print(b)
[4 5 6]
>>> c = np.array([7, 8])
>>> print(c)
[7 8]
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(A)
[[1 2 3]
 [4 5 6]]
>>> B = np.array([[7, 8], [9, 10]])
>>> print(B)
```

```
[[ 7  8]
 [ 9 10]]
>>> C = np.array([[1, 2], [3, 4], [5, 6]])
>>> print(C)
 [[1 2]
 [3 4]
 [5 6]]
```

Concatenatie van 1-dimensionale arrays

Enkele voorbeelden:

```
>>> res1 = np.concatenate((a, b))
>>> print(res1)
[1 2 3 4 5 6]

>>> res2 = np.concatenate((a, c))
>>> print(res2)
[1 2 3 7 8]
```

Merk op dat, hoewel `axis` default op 0 staat, de 1-dimensionale arrays `a` en `c` horizontaal geconcateneerd worden. Willen we deze arrays vertikaal concateneren, dan moeten we er 1×3 arrays van maken door extra haakjes te gebruiken:

```
>>> a = np.array([ [1, 2, 3] ]) # extra []
>>> print(a)
[[1 2 3]]
>>> b = np.array([ [4, 5, 6] ]) # extra []
>>> print(b)
[[4 5 6]]
>>> c = np.array([ [7, 8] ])    # extra []
>>> print(c)
[[7 8]]

>>> res3 = np.concatenate((a, b))           # onder elkaar
>>> print(res3)
[[1 2 3]
 [4 5 6]]

>>> res4 = np.concatenate((a, b), axis = 1) # naast elkaar
>>> print(res4)
[[1 2 3 4 5 6]]

>>> res5 = np.concatenate((a, c), axis = 1) # naast elkaar
>>> print(res5)
[[1 2 3 7 8]]
```

Concatenatie van 2-dimensionale arrays

Enkele voorbeelden:

```
>>> res = np.concatenate((A, B), axis = 1) # naast elkaar
>>> print(res)
[[ 1  2  3  7  8]
 [ 4  5  6  9 10]]

>>> res = np.concatenate((A, B), axis = 0) # onder elkaar
ValueError: all the input array dimensions for the concatenation axis must
match exactly, but along dimension 1, the array at index 0 has size 3 and
the array at index 1 has size 2
```

Vermits `A` een 2×3 array en `B` een 2×2 array is, kunnen deze arrays niet vertikaal geconcateneerd worden (`B` heeft slechts 2 kolommen, geen 3).

Concatenatie van 1-dimensionale en 2-dimensionale arrays

In sommige toepassingen moet onderaan een 2-dimensionale array een rij, of rechts een kolom toegevoegd worden. Stel bijvoorbeeld dat we de 2-dimensionale matrix `A` en de 1-dimensionale matrix `a` vertikaal willen concateneren:

```
>>> a = np.array([1, 2, 3])      # 1-dimensionale array
>>> res = np.concatenate((A, a))
ValueError: all the input arrays must have same number of dimensions,
but the array at index 0 has 2 dimension(s) and the array at index 1
has 1 dimension(s)
```

Hoewel het aantal kolommen identiek is, wordt er een `ValueError` opgeroepen omdat `A` twee assen (2 dimensies) en `a` slechts één as heeft. Dir probleem kan eenvoudig opgelost worden m.b.v. de functie `reshape()`:

```
>>> a = np.array([1, 2, 3])      # 1-dimensionale array
>>> res = np.concatenate((A, np.reshape(a, (1, 3))))
>>> print(res)
[[1 2 3]
 [4 5 6]
 [1 2 3]]
```

8.2.12.5 Randen toevoegen

In bepaalde toepassingen worden aan 2-dimensionale arrys één of meer randen (met nullen of enen) toegevoegd. Een typisch voorbeeld is om ervoor te zorgen dat elk element van de beschouwde array 8 buren heeft. Het toevoegen van een rand kan als volgt:

1. creëer (met `zeros()` of `ones()`) een nieuwe array met **2 extra rijen en 2 extra kolommen**, en

2. vul het **binnenste** van de nieuwe array op met de elementen van de oorspronkelijke array.

Een voorbeeld:

```
>>> print(A)
[[1 2 3]
 [4 5 6]]
>>> m, n = A.shape
>>> A2 = np.zeros((m+2, n+2), dtype = int)
>>> print(A2)
[[0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]
 [0 0 0 0 0]]

 >>> A2[1:m+1, 1:n+1] = A  # m: index van de voorlaatste rij
 >>> print(A2)             # n: index van de voorlaatste kolom
 [[0 0 0 0 0]
  [0 1 2 3 0]
  [0 4 5 6 0]
  [0 0 0 0 0]]
```

8.2.13 Basisbewerkingen op 2-dimensionale arrays

Alle bewerkingen (`+`, `-`, `*`, `/` en `**`), vergelijkingsoperatoren (`<`, `>`, `==`, `<=`, `>=`, en `!=`) en logische operatoren (`&`, `|`, $\wedge$ en $\sim$) die aan bod besproken werden in Sectie 8.2.6 zijn ook toepasbaar op 2-dimensionale arrays. Het resultaat is steeds een array met dezelfde dimensies (*shape*) als waarop de bewerking(en) uitgevoerd werden.

8.2.13.1 Logische indexering (*logical indexing*)

Logische indexering is ook bij 2-dimensionale arrays mogelijk.

Veronderstel bijvoorbeeld dat we in de 2-dimensionale array

$$M = \begin{pmatrix} 2 & 3 & 6 & 4 & 8 \\ 5 & 2 & 4 & 6 & 2 \\ 7 & 2 & 2 & 5 & 3 \\ 6 & 2 & 6 & 8 & 9 \\ 5 & 5 & 0 & 3 & 7 \end{pmatrix}$$

de elementen willen selecteren die groter of gelijk zijn aan 3. Bekijk hiertoe het onderstaande voorbeeld:

```
>>> M = np.array([[2, 3, 6, 4, 8], [5, 2, 4, 6, 2], [7, 2, 2, 5, 3],
                  [6, 2, 6, 8, 9], [5, 5, 0, 3, 7]])
```

```
>>> print(M)
[[2 3 6 4 8]
 [5 2 4 6 2]
 [7 2 2 5 3]
 [6 2 6 8 9]
 [5 5 0 3 7]]

>>> res = M >= 3
>>> print(res)
[[False  True  True  True  True]
 [ True False  True  True False]
 [ True False False  True  True]
 [ True False  True  True  True]
 [ True  True False  True  True]]
>>> N = M[res]
>>> print(N)
[3 6 4 8 5 4 6 7 5 3 6 6 8 9 5 5 3 7]
```

Merk op dat de elementen die voldoen aan de opgegeven voorwaarde **rij per rij** uit `M` gehaald worden.

Beide instructies kunnen **gecombineerd** worden:

```
>>> N = M[M >= 3]
>>> print(N)
[3 6 4 8 5 4 6 7 5 3 6 6 8 9 5 5 3 7]
```

Opdracht 8.7 (`omringd.py`)

Lees uit het bestand `omringd.txt` de matrix `A` als volgt in met de functie `loadtxt()`:

```
>>> A = np.loadtxt("omringd.txt", dtype = int)
```

De matrix A bevat enkel de getallen `0`, `1` en `2` zoals hieronder te zien is:

```
0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 2 1 1 0 0 0 0
0 0 0 1 1 1 0 0 0 0 0 0 0 0 1 1 1 0 0 0
0 0 0 1 2 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0
0 0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 0 0 0 0
0 1 1 0 0 0 0 0 0 0 0 1 2 1 0 0 0 0 0 0
1 1 1 1 0 0 0 1 1 1 0 1 0 1 0 0 0 0 0 0
1 2 1 1 1 0 0 1 0 1 0 0 0 0 1 1 1 0 0 0
1 1 1 0 0 0 0 1 1 1 0 0 0 0 1 2 1 0 0 0
0 1 1 0 1 1 1 0 0 0 0 0 0 1 1 1 1 0 0 0
0 0 0 0 1 0 1 0 0 1 0 0 0 0 2 1 0 0 0 0
0 0 0 0 1 1 1 0 0 0 2 1 0 0 1 1 0 0 0 0
0 0 0 1 0 0 1 0 1 1 1 1 0 0 1 0 0 0 0 0
```

```
0 0 1 2 1 1 1 2 0 0 0 0 0 1 2 0 0 0 0 0
0 0 0 1 0 0 0 1 1 0 0 0 0 1 1 1 0 0 0 0
0 0 0 0 0 0 0 1 1 1 1 1 1 0 0 0 0 0 0 0
0 1 1 1 1 0 0 0 2 1 1 2 1 0 1 1 1 1 0 0
1 2 0 0 0 0 0 1 1 1 1 1 1 0 1 2 2 1 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0
0 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Bepaalde van de getallen `2` zijn deels of volledig omringd door de buurgetallen `1`. De bedoeling van deze oefening is te bepalen hoeveel van de getallen `2` omringd zijn door minstens 7 buren die gelijk zijn aan `1`. Een voorbeeld van een dergelijk getal is het getal 2 met rij-index 3 en kolom-index 4. We veronderstellen dat de getallen `2` zich **niet op de rand** van de matrix `A` bevinden.

Volg het volgende stappenplan:

- Maak de onderstaande logische matrix `M` aan (dit kan op meerdere manieren):

```
[[ True  True  True]
 [ True False  True]
 [ True  True  True]]
```

- Itereer met een geneste `for`-lus over de matrix `A` (gebruik index `i` als rij-index en index `j` als kolom-index).
- Ga na welke elementen `A[i, j]` gelijk zijn aan `2`.
- Voor de elementen `A[i, j]` die gelijk zijn aan `2`, gebruik array-slicing om de 3×3 submatrix te selecteren met `2` als centraal element. Ken deze submatrix toe aan de variabele `S`.
- Gebruik de matrix `M` om via logische indexering de buren te bepalen van `2`.
- Ga na of de som van de buren gelijk is aan 7. Hou dit bij met een teller.
- Controleer je uitkomst via visuele inspectie van de matrix `A`. Je zou 8 moeten bekomen.

8.2.13.2 Wijzigen van elementen

We kunnen één element of een selectie van elementen wijzigen in een array. Die selectie kunnen we maken met start/eind rij/kolom indices `[r1:r2, k1:k2]` zoals bij slicing.

Opdat de wijziging zou kunnen uitgevoerd worden, moet(cf. Sectie 8.2.6.6):

- de *shape* van de vervangarray overeenstemmen met de *shape* van het geselecteerde deel, of
- de vervangarray uit slechts één element bestaan (laatste voorbeelden).

Bekijk de onderstaande voorbeelden:

```
>>> A = np.arange(1, 13).reshape((3, 4))  # aanmaak van de array A
>>> print(A)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

>>> A[2,1] = 0          # element op rij 3 en kolom 2 wijzigen tot 0
>>> print(A)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9  0 11 12]]

>>> A[0,:] = 3*A[0,:]   # 1ste rij verdrievoudigen
>>> print(A)
[[ 3  6  9 12]
 [ 5  6  7  8]
 [ 9  0 11 12]]

>>> B = np.array( [[10, 20], [20, 30]] )  # aanmaak van de array B
>>> print(B)
[[10 20]
 [20 30]]

>>> A[0:2,0:2] = B      # vervang deel van A door B
>>> print(A)
[[10 20  9 12]
 [20 30  7  8]
 [ 9  0 11 12]]

>>> A[1,:] = 0          # vervang 2e rij door 0'en
>>> print(A)
[[10 20  9 12]
 [ 0  0  0  0]
 [ 9  0 11 12]]

>>> A[1:3,2:4] = -1     # 2x2 array rechtsonder wordt -1
>>> print(A)            # A[1:3,2:4] is equivalent met A[1:,2:]
[[10 20  9 12]
 [ 0  0 -1 -1]
 [ 9  0 -1 -1]]
```

Opdracht 8.8

Maak de volgende twee matrices `A` en `B` aan:

$$A = \begin{pmatrix} 0 & 2 & 5 & 7 & 4 & 9 & 11 & 9 \\ 2 & 47 & 8 & 4 & 1 & 2 & 7 & 5 \\ 8 & 2 & 9 & 4 & 12 & 2 & 8 & 4 \\ 8 & 7 & 2 & 7 & 5 & 7 & 2 & 11 \end{pmatrix} \qquad B = \begin{pmatrix} 7 & 2 & 4 & 8 & 1 & 1 & 1 & 12 \\ 2 & 55 & 8 & 4 & 8 & 2 & 2 & 5 \end{pmatrix}$$

Voer de volgende opdachten uit **zonder** gebruik te maken van **lussen**:

- Toon de arrays op het scherm met het `print`-statement. Volgend resultaat moet je zien:

```
[[ 0  2  5  7  4  9 11  9]
 [ 2 47  8  4  1  2  7  5]
 [ 8  2  9  4 12  2  8  4]
 [ 8  7  2  7  5  7  2 11]]
```

```
[[ 7  2  4  8  1  1  1 12]
 [ 2 55  8  4  8  2  2  5]]
```

- Stel het element in A op de tweede rij (rij-index 1), derde kolom (kolom-index 2) gelijk aan de waarde 100. Toon de array A op het scherm. Volgend resultaat moet je zien:

```
[[  0   2   5   7   4   9  11   9]
 [  2  47 100   4   1   2   7   5]
 [  8   2   9   4  12   2   8   4]
 [  8   7   2   7   5   7   2  11]]
```

- Vervang de eerste twee rijen van A door de waarden in B. Toon de array A op het scherm. Volgend resultaat moet je zien:

```
[[  7   2   4   8   1   1   1  12]
 [  2  55   8   4   8   2   2   5]
 [  8   2   9   4  12   2   8   4]
 [  8   7   2   7   5   7   2  11]]
```

- Extraheer alle elementen uit A die even rij- en kolomindices hebben. Probeer dit zowel met als zonder gebruik te maken van lussen. Volgend resultaat moet je zien:

```
[7, 4, 1, 1, 8, 9, 12, 8]
```

8.2.14 Broadcasting

Tot hiertoe werden bewerkingen uitgevoerd op een array en een getal, of arrays met identieke dimensies. Dit hoeft echter niet altijd zo te zijn.

Broadcasting is een verzameling van regels voor het toepassen van rekenkundige bewerkingen op arrays met verschillende dimensies. De arrays (niet noodzakelijk allemaal) worden *uitgebreid* zodat de dimensies compatibel worden.

Een specifiek geval van broadcasting kwam reeds aan bod in Sectie 8.2.6: bij het optellen van een array en een scalair[8]. Bij het optellen van een scalair `g` bij een array `v` wordt `g` **uitgebreid** (*gestretched*) tot een array die dezelfde dimensies heeft als de array `v`. Dit wordt geïllustreerd in Fig. 8.3 waar `v = [1, 2, 3]` en `g = 5`.

[8] Analoog voor de aftrekking, vermenigvuldiging, deling, machtsverheffing, enz.

Figuur 8.3: Broadcasting bij 1-dimensionale arrays: horizontale uitbreiding.

```
>>> v = np.array([1, 2, 3])
>>> g = 5
>>> v + g
array([6, 7, 8])
```

Het broadcasten volgt een aantal **regels**. Bij het inwerken op twee arrays vergelijkt NumPy hun *shape* (dimensies) elementsgewijs. Het begint met de laatste (d.w.z. meest rechtse) dimensie en werkt zo naar voren (naar links). Twee dimensies zijn **compatibel** als

1. ze gelijk zijn, of
2. één van beide gelijk is aan 1.

Als niet aan deze voorwaarden is voldaan, zijn de dimensies niet compatibel en wordt een foutmelding gegenereerd.

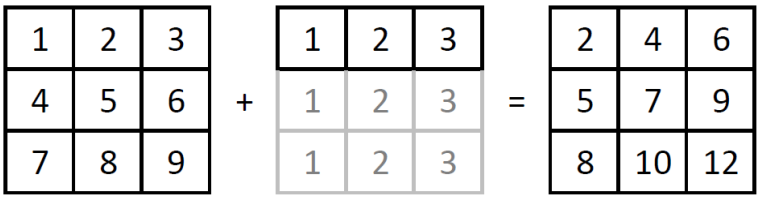
De arrays hoeven niet hetzelfde aantal dimensies te hebben. De resulterende array heeft hetzelfde aantal dimensies als de array met het grootste aantal dimensies. De grootte van elke dimensie zal het maximum zijn van de overeenkomstige dimensie van de arrays. Ontbrekende dimensies worden verondersteld waarde 1 te hebben. Enkele voorbeelden zullen dit principe verduidelijken.

Voorbeeld 1

Veronderstel dat we bij een 3×3 array `A` een 1-dimensionale array `v` met 3 elementen willen optellen:

$$\texttt{A} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad en \qquad \texttt{v} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix}.$$

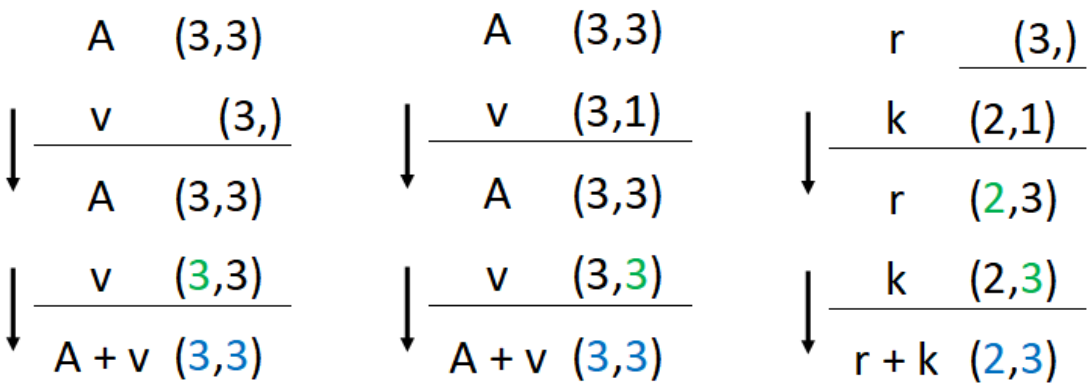
```
>>> A = np.arange(1, 10).reshape((3,3))
>>> A.shape
(3,3)       # 2-dimensionale array A
>>> v = np.array([1, 2, 3])
>>> v.shape
(3,)        # 1-dimensionale array v
>>> som = A + v
>>> (som
array([[ 2,  4,  6],
       [ 5,  7,  9],
       [ 8, 10, 12]])
>>> som.shape
(3,3)
```



Figuur 8.4: Broadcasting bij de optelling van een 2-D en een 1-D array: **vertikale** uitbreiding.

De array `v` zal, ten gevolge van *broadcasting*, **vertikaal** worden uitgebreid tot een 3×3 matrix en elementsgewijs opgeteld worden bij `A`. Dit wordt geïllustreerd in Fig. 8.4.

Vermits `A` de meeste (nl. 2) dimensies heeft, zal de resulterende array ook 2 dimensies hebben. De grootte van de rechtse dimensie zal gelijk zijn aan 3 omdat de grootste overeenkomende dimensie gelijk is aan 3. Idem voor de linkse dimensie: bij `A` is deze gelijk aan 3, en bij `v` ontbreekt deze deze dimensie en wordt dus verondersteld gelijk te zijn aan 1. Het maximum is dus 3. Figuur 8.5(a) illustreert dit.

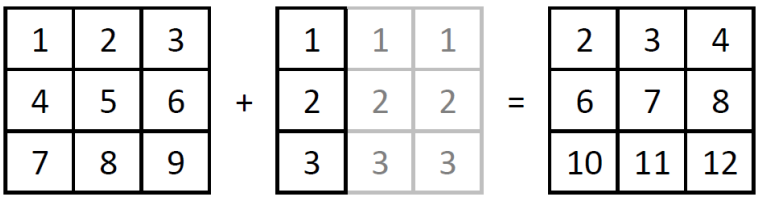


(a) 2-D plus 1-D (b) 2-D plus 2-D (c) 1-D plus 2-D

Figuur 8.5: Toepassing van de broadcasting regels.

Voorbeeld 2

Als de array `v` een 3×1 array zou zijn, dan wordt deze **horizontaal** uitgebreid. Dit wordt geïllustreerd in Fig. 8.6.



Figuur 8.6: Broadcasting bij 2-dimensionale arrays: **horizontale** uitbreiding.

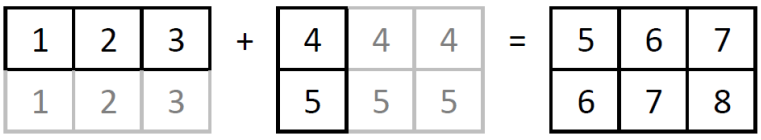
```
>>> v = np.array([1, 2, 3]).reshape((3,1))
>>> print(v)
```

```
[[1]
 [2]
 [3]]
>>> v.shape
(3,1)       # 2-D array
>>> A + v
array([[ 2,  3,  4],
       [ 6,  7,  8],
       [10, 11, 12]])
```

Ook nu zal de de resulterende array 2 dimensies hebben. De grootte van de rechtse dimensie zal gelijk zijn aan 3 omdat de grootste overeenkomende dimensie gelijk is aan 3: bij `A` is deze gelijk aan 3, bij `v` is deze gelijk aan 1. Idem voor de linkse dimensie: bij `A` is deze gelijk aan 3, en bij `v` is deze ook gelijk aan 3. Het maximum is dus wederom 3. Figuur 8.5(b) illustreert dit.

Voorbeeld 3

Als de ene array een $1 \times n$ rijvector `r` en de andere een $m \times 1$ kolomvector `k` is, worden beide arrays uitgebreid tot $m \times n$ arrays. Dit wordt geïllustreerd in Fig. 8.7.



Figuur 8.7: Horizontale **én** vertikale uitbreiding.

```
>>> r = np.array([1, 2, 3])
>>> print(r)
[1 2 3]
>>> r.shape
(3,)       # 1-D array
>>> k = np.array([4, 5]).reshape((2,1))
>>> print(k)
[[4]
 [5]]
>>> k.shape
(2,1)

>>> r + k
array([[5, 6, 7],
       [6, 7, 8]])
```

De rechtse dimensie van `r` is 3, die van `k` is 1. Het maxiumum is dus 3. De linkse dimensie van `r` ontbreekt en wordt dus verondersteld gelijk te zijn aan 1, dat van `k` is 2. Het maximum is dus 2. Bijgevolg zal de resulterende array een 2×3 array zijn. Figuur 8.5(c) illustreert dit.

8.2.15 NumPy Functies

Alle functies (goniometrische, exponentiële, logaritmische, enz.) in Tabel 8.2 aanvaarden ook 2-dimensionale arrays als input.

We beperken ons hier tot het illustreren van enkele van die functies met 2-dimensionale arrays. Bekijk de onderstaande voorbeelden:

```
>>> A = np.array([[18, 18, 15, 4, 13], [6,  6, 4, 10, 19], \
[ 7, 5, 12, 15, 8], [17, 10, 3, 12, 6]])
>>> print(A)
[[18 18 15  4 13]
 [ 6  6  4 10 19]
 [ 7  5 12 15  8]
 [17 10  3 12  6]]

>>> print(np.max(A))
19        # maximum van de volledige array

>>> print(np.mean(A))
10.4      # gemiddelde van de volledige array
```

Willen we het minimum, het maximum, de som, ..., **rij per rij** of **kolom per kolom** berekenen, dan kunnen we met het attribuut `axis` aangeven volgens welke dimensie (as) er gewerkt moet worden. De waarde 0 komt, bij 2-dimensionale arrays, overeen met de kolommen, de waarde 1 met de rijen:

```
# minimum per KOLOM
>>> print(np.min(A, axis = 0))
[6 5 3 4 6]       # er zijn 5 kolommen

# som per RIJ
>>> print(np.sum(A, axis = 1))
[68 45 47 48]     # er zijn 4 rijen
```

Enkele interessante functies m.b.t. 2-dimensionale arrays zijn:

- `fliplr()`: keert de volgorde van de **kolommen** om,
- `flipud()`: keert de volgorde van de **rijen** om,
- `diag()`: retourneert de **diagonaal**elementen

Bij elk van deze functies blijft de **oorspronkelijke array ongewijzigd**.

Enkele voorbeelden:

```
>>> print(np.fliplr(A))
[[13  4 15 18 18]
 [19 10  4  6  6]
 [ 8 15 12  5  7]
 [ 6 12  3 10 17]]

>>> print(np.flipud(A))
[[17 10  3 12  6]
 [ 7  5 12 15  8]
 [ 6  6  4 10 19]
 [18 18 15  4 13]]
```

De functie `diag()` retourneert ofwel een 1-dimensionale array met de diagonaalelementen terug, ofwel, indien een 1-dimensionale array werd meegegeven een 2-dimensionale array met de opgegeven arrayelementen op de diagonaal:

```
>>> M = np.arange(1, 17).reshape((4, 4))
>>> print(M)
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]
 [13 14 15 16]]

>>> print(np.diag(M))   # extraheer diagonaalelementen
[ 1  6 11 16]

>>> d = np.arange(1, 6)
>>> print(d)
[1 2 3 4 5]

>>> print(np.diag(d))   # maak een array met d op de diagonaal
[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]
```

Opdracht 8.9

We werken hier verder met de matrices bekomen in Opdracht 8.8. Op het einde van Opdracht 8.8 ziet de matrix `A` er als volgt uit:

```
>>> print(A)
[[  7   2   4   8   1   1   1  12]
 [  2  55   8   4   8   2   2   5]
 [  8   2   9   4  12   2   8   4]
 [  8   7   2   7   5   7   2  11]]
```

Voer de volgende opdachten uit:

- Bepaal hoeveel waarden in `A` groter zijn dan 10. Volgend resultaat moet je bekomen:

```
4               # aantal waarden groter dan 10
[12 55 12 11]   # een lijst met de waarden groter dan 10
```

- Vervang alle elementen in `A` die groter zijn dan 10 door 1. Volgend resultaat moet je bekomen:

```
[[7 2 4 8 1 1 1 1]
 [2 1 8 4 8 2 2 5]
 [8 2 9 4 1 2 8 4]
 [8 7 2 7 5 7 2 1]]
```

- We werken hier verder met de array:

```
[[7 2 4 8 1 1 1 1]
 [2 1 8 4 8 2 2 5]
 [8 2 9 4 1 2 8 4]
 [8 7 2 7 5 7 2 1]]
```

  Beschouw de cel met rij-index 2 en kolom-index 3. Extraheer alle buurcellen van deze cel en controleer hoeveel van deze buurcellen de waarde 2 bevatten. Gebruik hiervoor o.a. de functie **`sum()`**. Volgend resultaat moet je bekomen:

```
4               # cel met rij-index 2 en kolom-index 3

# array van alle buurcellen van cel met rij-index 2 en kolom-index 3
[8 4 8 9 1 2 7 5]

1               # aantal buurcellen gelijk aan 2
```

- Gebruik controlestructuren om het vorige toe te passen op elke cel (met uitzondering van de cellen op de rand) in de matrix. Bepaal telkens het aantal buren dat gelijk is aan 2 en voeg dit aantal toe aan een *list*. De volgende *list* moet je bekomen:

```
[3, 2, 0, 2, 2, 2, 2, 2, 1, 2, 3, 4]
```

De functie `array_equal()`

De functie `array_equal()` aanvaardt ook multidimensionale arrays als argument:

```
>>> X = np.array([[1, 2, 3], [4, 5, 6]])
>>> Y = np.array([[1, 2, 3], [4, 5, 6]])
>>> Z = np.array([[1, 2, 3], [7, 8, 9]])
>>> U = np.array([[1, 2], [3, 4]])
```

```
>>> np.array_equal(X, Y)
True    # dimensies komen overeen, inhoud ook

>>> np.array_equal(X, Z)
False   # dimensies komen overeen, inhoud niet (2e rij van Z)

>>> np.array_equal(X, U)
False   # dimensies komen niet overeen (2x3 vs 2x2)
```

De functie `unique()`

De functie `unique()` aanvaardt ook multidimensionale arrays als argument:

```
>>> R = np.array([[1, -3, 0, 0, -1, -4], [0, 0, 0, -2, 4, 0], \
                  [1, 0, 0, 0, -5, 0]])
>>> print(R)
[[ 1 -3  0  0 -1 -4]
 [ 0  0  0 -2  4  0]
 [ 1  0  0  0 -5  0]]

>>> uniek = np.unique(R)
>>> uniek
array([-5, -4, -3, -2, -1,  0,  1,  4])
```

De functie `nonzero()`

In Sectie 8.2.8 hebben we gebruik gemaakt van de functie `nonzero()` om in een 1-dimensionale array de indices van de niet-nul elementen te bepalen. Ook deze functie aanvaardt multidimensionale arrays als argument:

```
>>> indices = np.nonzero(R)  # aanmaak array R: zie vorige pagina
>>> indices
(array([0, 0, 0, 0, 1, 1, 2, 2], dtype = int64),
 array([0, 1, 4, 5, 3, 4, 0, 4], dtype = int64))
```

Merk op dat de functie `nonzero()` nu een tuple retourneert met 2 arrays, één voor elke dimensie.

De rij- en kolomindices kunnen op een eenvoudige manier in afzonderlijke variabelen bewaard worden m.b.v. *tuple unpacking* (zie Sectie 5.5.3):

```
>>> rij, kol = np.nonzero(R)
>>> print(rij)
[0 0 0 0 1 1 2 2]

>>> print(kol)
[0 1 4 5 3 4 0 4]
```

Merk op dat de array **rij per rij** doorlopen wordt (cf. Sectie 8.2.12.1 waar de nieuwe array default rij per rij opgevuld wordt). Zo komt de positie `(rij[0], kol[0]) = (0, 0)` overeen met de 1 op de eerste rij in de array `R`, terwijl de positie `(rij[6], kol[6]) = (2, 0)` overeen met de 1 die zich op de laatste rij bevindt (en dus niet met -4 op de eerste rij).

Opdracht 8.10 (`controleer.py`)

Schrijf een functie `controleer()` die een 2-dimensionale array `M` en een getal `g` aanvaardt en nagaat of het opgegeven getal **minstens 1 keer voorkomt in elke rij** van de opgegeven array. Je functie retourneert `True` als dit het geval is, en `False` indien niet. Er zijn verschillende oplossingen mogelijk maar hier **moet** je gebruik maken van de functie `nonzero()`. Onderstaande sessie illustreert het gebruik van de functie `controleer()`:

```
>>> M = np.array([[1, 2, 3, 4, 5], [2, -5, 0, 6, 7], [4, 3, 0, 2, -1]])
>>> print(M)
[[ 1  2  3  4  2]
 [ 2 -5  0  6  0]
 [ 4  3  0  2 -1]]

>>> res = controleer(M, 2)
>>> res
True        # 2 komt voor op elke rij

>>> res2 = controleer(M, 0)
>>> res2
False       # 0 komt NIET voor in de eerste rij
```

Tip: de functies `unique()` en `array_equal()` (zie Sectie 8.2.8) kunnen hier handig zijn. De implementatie kan bovendien **zonder** `for`-lus(sen).

8.2.16 Input/output van/naar bestanden met `loadtxt()` en `savetxt()`

8.2.16.1 Inlezen van numerieke tekstbestanden met `loadtxt()`

Het inlezen van arrays kan op vele verschillende manieren. We beperken ons hier tot de functie `loadtxt()` die numerieke tekstbestanden inleest. We illustreren het inlezen met enkele nuttige opties. Voor meer gedetailleerde informatie omtrent de functie `loadtxt()` verwijzen we naar: `https://numpy.org/doc/stable/reference/generated/numpy.loadtxt.html`.

De signatuur van de functie `loadtxt()` is:

```
loadtxt(fname, dtype = None, comments = "#", delimiter = None,
        skiprows = 0, ndmin = 0)
```

De betekenis van de parameters is:

- `fname`: een string met de **naam** van het in te lezen **tekstbestand**,
- `dtype`: het **gegevenstype** van de resulterende **array** (default `float`),
- `comments`: een string met het teken waarmee **commentaarlijnen** beginnen, default is `"#"`,
- `delimiter`: een string met het **scheidingsteken**, default is witruimte (één of meer spaties),
- `skiprows`: een `int` die aangeeft hoeveel rijen **overgeslagen** moeten worden, default is 0,
- `ndmin`: een `int` die aangeeft hoeveel **dimensies** (assen) de resulterende array moet hebben, default is 0.

De functie retourneert een `ndarray` met de gegevens in het ingelezen tekstbestand. Het gegevenstype wordt bepaald door de parameter `dtype`.

Enkele voorbeelden:

- In het tekstbestand `matrixA.txt` zijn de gehele getallen (*int*'s) gescheiden door een spatie en staan in rijen onder elkaar:

```
0 2 5 7 4 9 11 9
2 47 8 4 1 2 7 5
8 2 9 4 12 2 8 4
8 7 2 7 5 7 2 11
```

  De functie `loadtxt` leest standaard *floats* in gescheiden door spaties:

```
>>> A = np.loadtxt("matrixA.txt")
>>> print(A)
[[  0.   2.   5.   7.   4.   9.  11.   9.]
 [  2.  47.   8.   4.   1.   2.   7.   5.]
 [  8.   2.   9.   4.  12.   2.   8.   4.]
 [  8.   7.   2.   7.   5.   7.   2.  11.]]
```

  Willen we `int`'s behouden, dan moeten we het dtype expliciet specifiëren met de optie: `dtype = int`:

```
>>> A = np.loadtxt("matrixA.txt", dtype = int)
>>> print(A)
[[ 0  2  5  7  4  9 11  9]
 [ 2 47  8  4  1  2  7  5]
 [ 8  2  9  4 12  2  8  4]
 [ 8  7  2  7  5  7  2 11]]
```

- In het bestand `matrixD.txt` staat er bovenaan het bestand informatie (bv. regels tekst), zijn de data van elkaar gescheiden door `";"` in plaats van spatie, en er staat een commentaarregel (voorafgegaan door een procentteken `%`):

```
Matrix met willekeurige getallen
343;218;523;86;635;313;864;370;163;255;109;141;65;251;526;537
467;738;842;867;529;983;727;278;805;512;947;38;835;20;498;905
611;42;763;470;928;452;598;241;358;54;803;524;88;924;516;544
% laatste rij getallen
888;523;868;75;173;84;432;445;981;919;440;19;88;499;772;54
```

Hier wensen we de **eerste regel niet** in te lezen. We kunnen dit aangeven met de optie: `skiprows = 1`. Vermits in dit bestand commentaarlijnen met een `%` beginnen, dienen we dit ook op te geven als waarde voor `comments`:

```
>>> D = np.loadtxt("matrixD.txt", dtype = int, delimiter = ";",\
... skiprows = 1, comments = "%")
>>> print(D)
[[343 218 523  86 635 313 864 370 163 255 109 141  65 251 526 537]
 [467 738 842 867 529 983 727 278 805 512 947  38 835  20 498 905]
 [611  42 763 470 928 452 598 241 358  54 803 524  88 924 516 544]
 [888 523 868  75 173  84 432 445 981 919 440  19  88 499 772  54]]
```

- Indien alle getalwaarden in een tekstbestand op één regel staan (zoals in `matrixE.txt`), of alle getalwaarden staan in één kolom onder elkaar, dan zal `loadtxt()` deze standaard als een 1-dimensionale array inlezen:

```
>>> E = np.loadtxt("matrixE.txt", delimiter = ";")
>>> print(E)
[ 0.564912  0.863341  0.266266  0.650384  0.25515   0.224277  0.995094
  0.761347  0.908963  0.04458   0.49773   0.729493  0.956066  0.487019
  0.245407  0.314358  0.666645  0.551497  0.732919  0.403817]
>>> E.shape
(20,)      # 1-dimensionale array E
```

Bemerk de enkelvoudige vierkante haakjes aan het begin (`[`) en einde (`]`) van de array.

Wens je de matrix toch als 2-dimensionale array in te lezen dan moet je de optie `ndmin = 2` meegeven:

```
>>> E = np.loadtxt("matrixE.txt", delimiter = ";", ndmin = 2)
>>> print(E)
[[ 0.564912  0.863341  0.266266  0.650384  0.25515   0.224277
   0.995094  0.761347  0.908963  0.04458   0.49773   0.729493
   0.956066  0.487019  0.245407  0.314358  0.666645  0.551497
   0.732919  0.403817]]
>>> E.shape
(1, 20)    # 2-dimensionale array E
```

Bemerk de dubbele vierkante haakjes aan het begin (`[[`) en einde (`]]`) van de array.

8.2.16.2 Wegschrijven van numerieke tekstbestanden met `savetxt()`

Ook het wegschrijven van array's kan op vele verschillende manieren. We beperken ons hier tot de functie `savetxt()` die numerieke tekstbestanden wegschrijft. We illustreren het wegschrijven met enkele nuttige opties. Voor meer gedetailleerde informatie omtrent de functie `savetxt()` verwijzen we naar: `https://numpy.org/doc/stable/reference/generated/numpy.savetxt.html`.

De signatuur van de functie `savetxt()` is:

```
savetxt(fname, X, fmt = "%.18e", delimiter = " ", newline = "\n")
```

De betekenis van de parameters is:

- `fname`: een string met de naam van het tekstbestand waarnaar weggeschreven moet worden,
- `X`: een 1- of 2-dimensionale `ndarray` met de data,
- `fmt`: een string of een lijst van strings waarbij elke string het formaat beschrijft, default is `"%.18e"` (wetenschappelijke, i.e. exponentiële, notatie waarbij 18 cijfers na de komma worden gebruikt),
- `delimiter`: een string met het scheidingsteken, default is witruimte (1 spatie),
- `newline`: een string die aangeeft hoe de lijnen van elkaar moeten gescheiden is, default is `"\n"` (nieuwe-lijn-karakter).

Het resultaat is een tekstbestand waarin de data weggeschreven zijn.

De belangrijkste opties van de functie `savetxt()` worden geïllustreerd a.d.h.v. de array `A`:

```
A = np.array([[ 1,  3,  4, 18,  7],
              [19,  8, 13, 14,  3],
              [12, 12, 16, 11, 16],
              [ 8,  8,  1,  3,  4],
              [14, 13,  9,  6, 19]])
```

- Indien we de matrix `A` wegschrijven zonder opties dan moeten we enkel een bestandsnaam en de arraynaam meegeven:

  ```
  >>> np.savetxt("matrix01.txt", A)
  ```

  De inhoud van het bestand `matrix01.txt` ziet eruit als:

  ```
  1.00000000000000000e+00 ... 7.00000000000000000e+00
  1.90000000000000000e+01 ... 3.00000000000000000e+00
  1.20000000000000000e+01 ... 1.60000000000000000e+01
  8.00000000000000000e+00 ... 4.00000000000000000e+00
  1.40000000000000000e+01 ... 1.90000000000000000e+01
  ```

Standaard worden de getalwaarden gescheiden door een spatie en wordt de wetenschappelijke (exponentiële) notatie gebruikt bij het wegschrijven. De structuur, i.e. aantal rijen en kolommen van de array, wordt bewaard.

- Indien we de getalwaarden in een andere notatie wensen (bv. in decimale notatie maar zonder exponent, met een beperkt aantal cijfers na de komma of als integer) dan kunnen we een zgn. *formatter* gebruiken en deze meegeven als `fmt = ...`. Bekijk de volgende mogelijkheden:

  – Formaat `fmt = "%5.2f"`: schrijf alle waarden weg als *float*, voorzie daarvoor 5 plaatsen en 2 cijfers na de "komma". Bemerk dat er standaard nog steeds een spatie tussen de getallen komt.

  ```
  np.savetxt("matrix03.txt", A, fmt = "%5.2f")
  ```

  levert:

  ```
   1.00  3.00  4.00 18.00  7.00
  19.00  8.00 13.00 14.00  3.00
  12.00 12.00 16.00 11.00 16.00
   8.00  8.00  1.00  3.00  4.00
  14.00 13.00  9.00  6.00 19.00
  ```

  – Formaat `fmt = "%3d"`: schrijf alle waarden weg als *int*, voorzie daarvoor 3 plaatsen. Bemerk dat er standaard nog steeds een spatie tussen de getallen komt. De instructie

  ```
  np.savetxt("matrix04.txt", A, fmt = "%3d")
  ```

  levert het volgende resultaat op:

  ```
   1   3   4  18   7
  19   8  13  14   3
  12  12  16  11  16
   8   8   1   3   4
  14  13   9   6  19
  ```

Opdracht 8.11 (`matrix_bewerking.py`)

Het bestand `oefenmatrix.txt` bevat een matrix waarvan de gehele getallen op elke rij gescheiden zijn door puntkomma's.

1. Lees de matrix uit het bestand `oefenmatrix.txt` in met de functie `loadtxt()` en ken deze toe aan de variabele `A`. Zorg ervoor dat de getalwaarden in `A` *ints* zijn. De matrix `A` zou er als volgt moeten uitzien:

```
>>> print(A)
[[17 37 38 46 44 15 34  1 28 19 27 40 22  4 17 36]
 [ 2  7 26 34 45 28 39 15 41  4 48  3 42 31 43 24]
 [ 6  3 33 17  8  8 24 37 12  3  3  8 30  6 21  4]
```

```
 [28 35 18 25 27 11 33  2 32 26 19 13  5 45 17 23]
 [38 15 37 44 20 47 13 42 45 39 11  8  4 34 28 32]
 [30  5  9  8 32 34 40 22 24 13 26  7 48 13 24 23]
 [27 16  9 32 46 23 27 41 48 34  9 36  4 28 10 28]
 [48 33 13 19 29 27 16  8 16 24  9 20 43 39 39 48]
 [27 43 40  7  5 45 15 10  8 30 24 20 13  6 19 31]]
```

2. Maak een kopie `B` van de matrix `A` door middel van de functie `array()`.

3. Gebruik een geneste **`for`**-lus structuur (itereer over de rijen en de kolommen) om elk oneven getal in `B` te vermenigvuldigen met -1. De matrix `B` zou er dan als volgt moeten uitzien:

```
>>> print(B)
[[-17 -37  38  46  44 -15  34  -1  28 -19 -27  40  22   4 -17  36]
 [  2  -7  26  34 -45  28 -39 -15 -41   4  48  -3  42 -31 -43  24]
 [  6  -3 -33 -17   8   8  24 -37  12  -3  -3   8  30   6 -21   4]
 [ 28 -35  18 -25 -27 -11 -33   2  32  26 -19 -13  -5 -45 -17 -23]
 [ 38 -15 -37  44  20 -47 -13  42 -45 -39 -11   8   4  34  28  32]
 [ 30  -5  -9   8  32  34  40  22  24 -13  26  -7  48 -13  24 -23]
 [-27  16  -9  32  46 -23 -27 -41  48  34  -9  36   4  28  10  28]
 [ 48 -33 -13 -19 -29 -27  16   8  16  24  -9  20 -43 -39 -39  48]
 [-27 -43  40  -7  -5 -45 -15  10   8  30  24  20 -13   6 -19 -31]]
```

4. Maak nu de som van `A` en `B` en ken deze toe aan de variabele `C`. De matrix `C` zou er dan als volgt moeten uitzien:

```
>>> print(C)
[[ 0  0 76 92 88  0 68  0 56  0  0 80 44  8  0 72]
 [ 4  0 52 68  0 56  0  0  0  8 96  0 84  0  0 48]
 [12  0  0  0 16 16 48  0 24  0  0 16 60 12  0  8]
 [56  0 36  0  0  0  0  4 64 52  0  0  0  0  0  0]
 [76  0  0 88 40  0  0 84  0  0  0 16  8 68 56 64]
 [60  0  0 16 64 68 80 44 48  0 52  0 96  0 48  0]
 [ 0 32  0 64 92  0  0  0 96 68  0 72  8 56 20 56]
 [96  0  0  0  0  0 32 16 32 48  0 40  0  0  0 96]
 [ 0  0 80  0  0  0  0 20 16 60 48 40  0 12  0  0]]
```

   De matrix `C` bevat nullen op plaatsen waar in `A` oneven getallen staan en de andere, even getallen, werden verdubbeld.

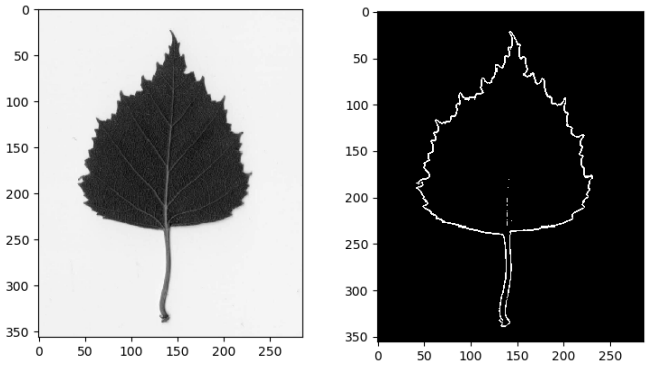
5. Tracht hetzelfde resultaat te bekomen voor de matrix `C` maar nu zonder geneste **`for`**-lus structuur. Dit kan met hoogstens twee korte instructieregels!

8.3 Gemengde opdrachten

Opdracht 8.12 (`detecteer_rand.py`)

Een grijswaardenfoto kan in het werkgeheugen bewaard worden een `numpy` array `M` waarin elk element een pixel voorstelt met als waarde (een getal tussen 0 en 1) de grijswaarde van de pixel (lage waarden zijn donkere pixels, hoge waarden stellen lichtere pixels voor). De onderstaande interactieve sessie leest een foto in als een `numpy` array `M` en visualiseert deze array (Figuur 8.8(a)).

```
>>> import matplotlib.pyplot as plt
>>> M = np.loadtxt("berk.txt")
>>> plt.imshow(M, cmap = "gray") # Figuur 2.1 (a)
>>> M[50, 250]
0.953 # HOGE waarde want zeer LICHTE pixel
>>> M[200, 100]
0.271 # LAGE waarde want zeer DONKERE pixel
```



(a) Origineel (b) Randen

Figuur 8.8: Resultaat van de toepassing van de gradiëntmethode.

De rand van een object wordt gekenmerkt door een overgang van donker naar licht (of omgekeerd). Men kan dus zeggen dat een pixel op de rand ligt indien de waarde van zijn linkerbuur sterk afwijkt van de waarde van zijn rechterbuur (hetzelfde geldt voor boven- en onderburen). De gradiënt van een pixel (i, j) is een maat voor deze afwijkingen en wordt als volgt berekend:

$$grad(i,j) = \left(\frac{M(i,j+1) - M(i,j-1)}{2}\right)^2 + \left(\frac{M(i+1,j) - M(i-1,j)}{2}\right)^2$$

Men besluit dat een pixel een randpixel is indien $grad(i, j) > 0.015$. Aan de zijkanten van de foto is de gradiënt nul. Schrijf een functie `detecteerRand()` die als argumenten een grijswaardenmatrix `M` aanvaardt, en die als output een randenmatrix `randen` retourneert. Dit is een boolean array met dezelfde dimensies als `M` waarin randpixels de waarde `True` hebben.

```
>>> M = np.loadtxt("berk.txt")
>>> randen = detecteerRand(M)
>>> plt.imshow(randen, cmap = "gray")  # Figuur 2.1 (b)
```

Opdracht 8.13 (`is_naburig.py`)

Beschouw een matrix `M` met **enkel** de getallen 1 t.e.m. 9 als matrixelementen:

```
>>> M
array([[2, 9, 3, 2, 4],
       [8, 6, 4, 8, 5],
       [5, 7, 1, 6, 4],
       [9, 8, 9, 5, 1]])
```

Beschouw een matrixelement `M(i,j)` dat **niet op de rand** gelegen is. Een dergelijk matrixelement heeft steeds acht buren. Als `M(i,j) > 1`, dan zeggen we dat het matrixelement **naburig** is als elk getal van 1 t.e.m. `M(i,j) - 1` voorkomt als één van de acht buren. Als `M(i,j) = 1`, dan is het matrixelement bij definitie naburig.

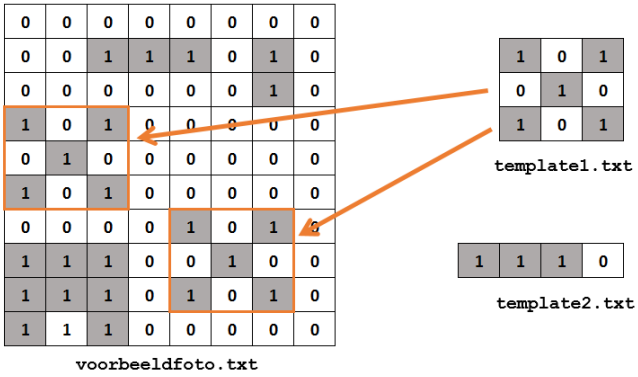
- **Voorbeeld 1**: Het matrixelement `M(1,1) = 6` is naburig omdat de getallen 1, 2, 3, 4 en 5 elk minstens één keer als buur voorkomen.
- **Voorbeeld 2**: Het matrixelement `M(1,2) = 4` is naburig omdat de getallen 1, 2 en 3 elk minstens één keer als buur voorkomen.
- **Voorbeeld 3**: Het matrixelement `M(2,3) = 6` is niet naburig daar de getallen 2 en 3 niet voorkomen als buren.
- **Voorbeeld 4**: Het matrixelement `M(2,2) = 1` is per definitie naburig.

Schrijf een functie `isNaburig()` die als inputs een matrix `M` met gehele getallen en een rij- en een kolomindex heeft. Je functie controleert of het matrixelement met de opgegeven rij- en kolomindex naburig is, en retourneert `True` indien dit het geval is (en `False` indien dit niet zo is).

```
>>> isNaburig(M, 2, 2)
True

>>> isNaburig(M, 2, 3)
False
```

Opdracht 8.14 (`matchtemplate.py`)

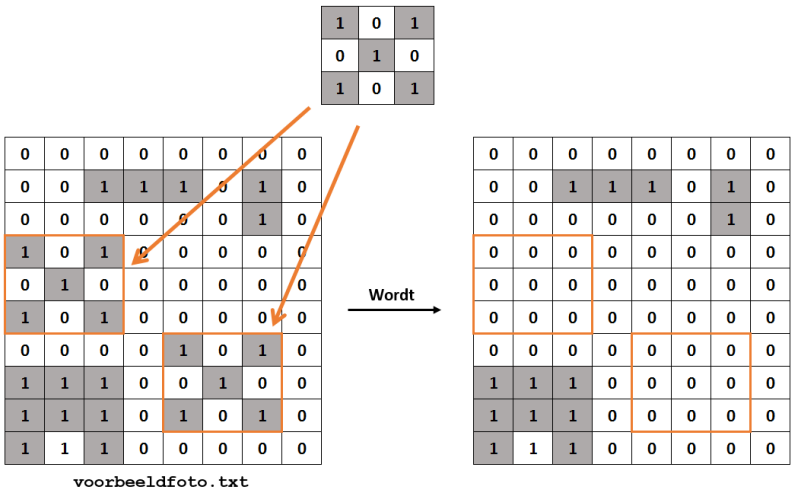
Template matching is een belangrijke techniek binnen de digitale beeldverwerking waarbij men in een zwart-wit foto (een `ndarray` in Python) op zoek gaat naar kleine stukjes die overeenkomen met een template.



In de bovenstaande zwart-wit foto komt `template1` exact twee keer voor. De template `template2` komt exact drie keer voor. De bestanden `voorbeeldfoto.txt`, `template1.txt` en `template2.txt` zijn beschikbaar in de map `bestanden_oefeningenlessen`. Je kan ze gebruiken om je implementatie te testen.

1. Schrijf een functie `matchTemplate()` die twee strings als input aanvaardt. Het eerste argument is de bestandsnaam van een foto/matrix en het tweede argument is de bestandsnaam van een template. Zorg dat deze functie:
   (i) Beide matrices inleest als numpy arrays (gebruik `loadtxt()`).
   (ii) Telt hoeveel keer de template voorkomt en dit aantal op het scherm print (je functie hoeft niets te retourneren).

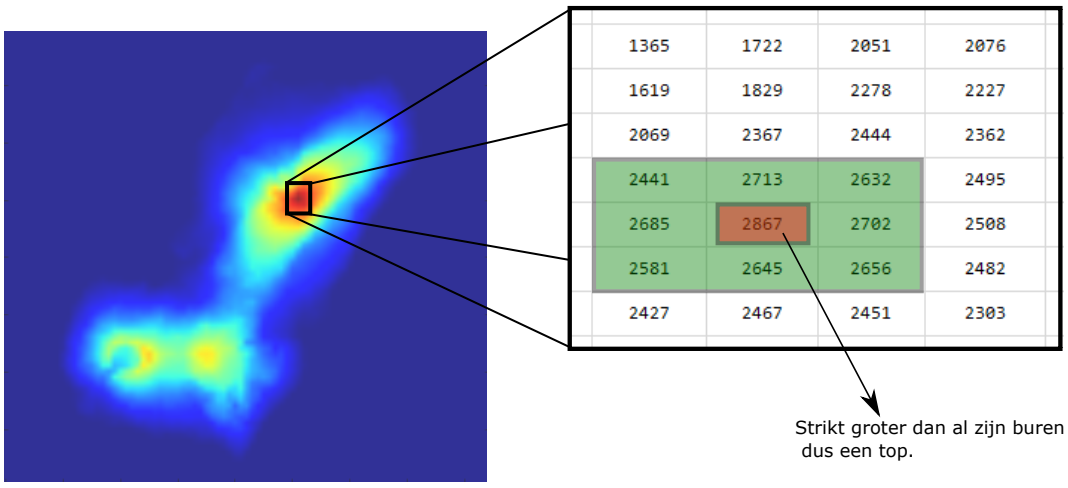
2. Breid de functie `matchTemplate()` uit zodat ze bovendien een numpy array retourneert met dezelfde dimensies als de ingelezen foto, maar waaruit de stukken die matchen de met de template vervangen worden door nullen. We zeggen dat de originele foto gefilterd werd.



```
>>> gefilterd = matchTemplate("voorbeeldfoto.txt", "template1.txt")
Er werden 2 matches gevonden.
>>> print(gefilterd)
[[ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  1.  1.  1.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  1.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.  0.  0.]
 [ 1.  1.  1.  0.  0.  0.  0.  0.]
 [ 1.  0.  1.  0.  0.  0.  0.  0.]]
>>> gefilterd = matchTemplate("voorbeeldfoto.txt", "template2.txt")
Er werden 3 matches gevonden.
```

Opdracht 8.15 (`dem_zoek_toppen.py`)

Een digitaal hoogtemodel (DEM) is een rasterbeeld van een deel van de wereld, voorgesteld door een matrix, waarin elke pixel een hoogte (in meter) boven de zeespiegel voorstelt. De bestanden `Bioko.txt` en `Saotome.txt` bevatten digitale hoogtemodellen van de respectievelijke eilanden Bioko en São Tomé (centraal West-Afrika). Je kan deze kaarten eenvoudig inlezen met de `loadtxt()`-functie uit de `numpy`-module.



Digitale hoogtemodellen kunnen gebruikt worden om de locaties van berg- en heuveltoppen te bepalen. Een top kan je definiëren als een plaats waarvoor geldt dat de hoogte ervan groter is dan de hoogtes van alle omringende plaatsen (zoals aangegeven in de figuur hierboven).

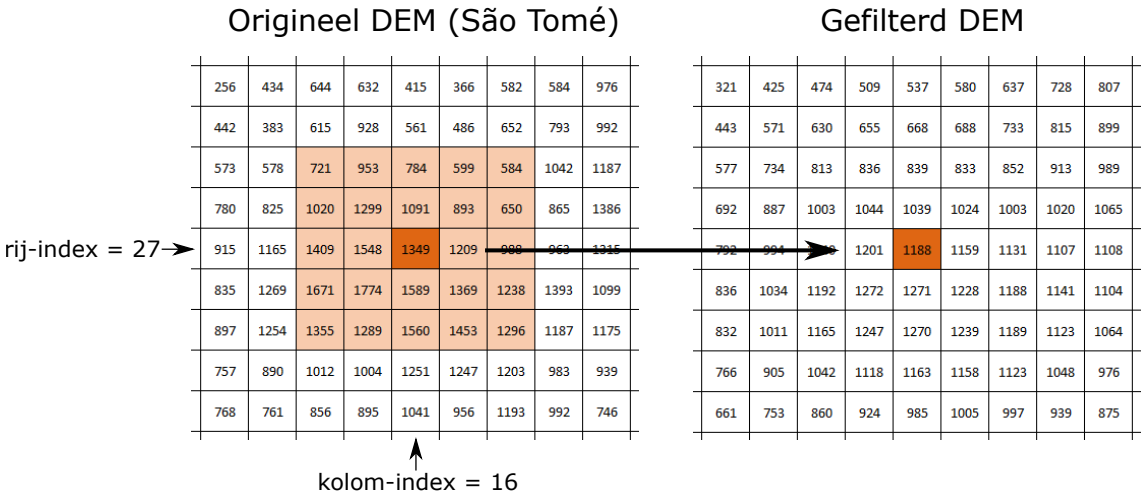
1. Schrijf een functie `zoekToppen()` die een digital hoogtemodel (een numpy array) aanvaardt als argument. Zorg dat deze functie:

   (i) De rij- en kolomindices bepaalt van alle berg- en heuveltoppen in het digitaal hoogtemodel.

(ii) Deze rij- en kolomindices retourneert als een lijst van tuples.

```
>>> import numpy as np
>>> dem_bioko = np.loadtxt("Bioko.txt")
>>> locaties_toppen_lst = zoekToppen(dem_bioko)
>>> print(locaties_toppen_lst)
[(30, 51), (39, 31), (48, 43), (55, 19), (57, 25), (57, 35)]
```

2. Kleine glooiingen in het landschap kunnen ervoor zorgen dat de functie `zoekToppen()` vrij veel bergtoppen terugvindt. Om dit aantal te reduceren en ervoor te zorgen dat enkel belangrijke toppen teruggevonden worden kan men het beeld gaan filteren met een gemiddelde-filter. Daarbij wordt een nieuw digitaal hoogtemodel gebouwd (nieuwe array) waarin de hoogte van een pixel bepaald wordt door het gemiddelde te berekenen van zijn buurpixels (inclusief zichzelf) in een 5×5 venster, zoals aangegeven in de figuur hieronder.



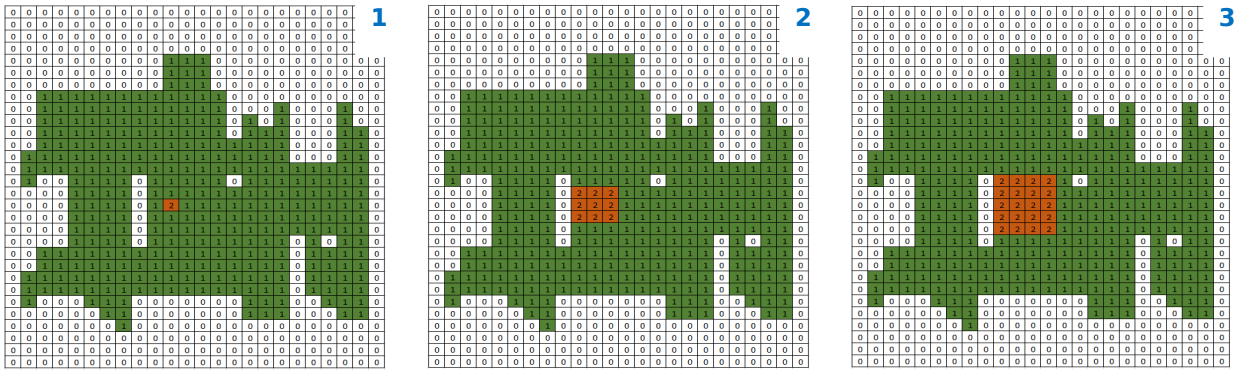
Schrijf een functie `gemiddeldeFilter()` die een digitaal hoogtemodel (een numpy array) aanvaardt als argument. Zorg dat deze functie een nieuwe array retourneert die het resultaat is van het toepassen van de gemiddelde-filter op dit DEM. Pixels die aan de rand van het DEM liggen alsook zee-pixels hoef je niet te filteren (deze blijven hun oorspronkelijke waarde behouden).

```
>>> dem_saotome = np.loadtxt("Saotome.txt")
>>> dem_gefilterd = gemiddeldeFilter(dem_saotome)
>>> print(len(zoekToppen(dem_saotome)))
27
>>> print(len(zoekToppen(dem_gefilterd)))
5
```

Opdracht 8.16 (`bosbrandsimulator.py`)

Wanneer in een natuurgebied een brand ontstaat (en er is geen/weinig wind), dan zal deze brand zich vaak concentrisch uitbreiden.

We maken in deze opdracht een eenvoudige bosbrandsimulator. Het bestand `bos.txt` bevat een kaart van een bosgebied. De kaart is gecodeerd als een matrix, met de volgende legende: `0` = geen bos, `1` = bos, `2` = brandhaard.



Deze kaart stelt een toestand voor op dag 1, cellen die grenzen aan een brandhaard zullen op dag 2 zelf ook wijzigen in een brandhaard (1 → 2). Hetzelfde gebeurt op dag 3, 4, ...

1. Lees het `bos.txt` bestand in met de NumPy functie `loadtxt`. Dit is de toestand op tijdstip 1.
2. Visualiseer de toestand op tijdstip 1 door de NumPy array die je in (1) bekomt te visualiseren met de functie `imshow()` (zie hieronder voor een voorbeeld van het gebruik van deze functie).
3. Schrijf een script dat de toestand (dus de matrix) berekent op dag 5.
4. Maak een visuele voorstelling de toestand van het gebied (water/bos/afgebrand) m.b.v. de functie `imshow()`.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> A = np.array( [[1, 2, 3, 4, 5, 6, 7], \
                   [3, 4, 5, 6, 7, 1, 2], \
                   [7, 6, 5, 4, 3, 2, 1]] )
>>> plt.imshow(A, interpolation = "none")
```

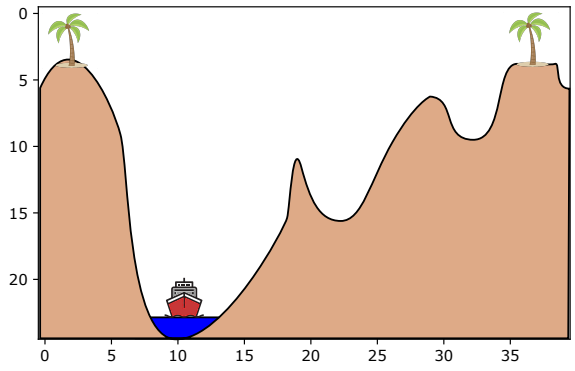
Opdracht 8.17 (`overstromingsvoorspeller.py`)

Het bestand `DEM_europa.txt` bevat een digitaal hoogtemodel van Europa (een matrix waarin de waarde van elke cel overeenstemt met een gemeten hoogte). Het zeeniveau heeft de waarde `0`.

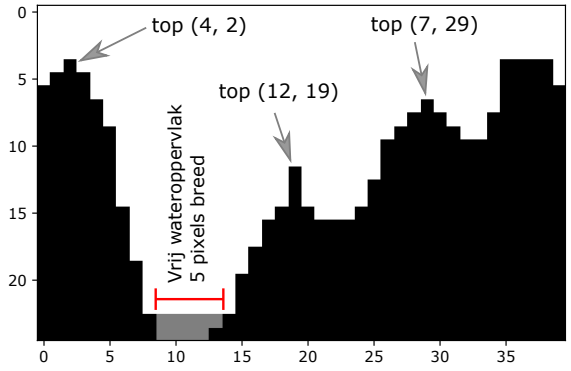
1. Lees deze kaart in met de NumPy functie `loadtxt()`.
2. Visualiseer deze kaart door de bekomen NumPy array in (1) te plotten met de functie `imshow`.
3. Maak een array van *booleans* aan (zelfde dimensies als de kaart), waarin alle cellen die zeegebied voorstellen de waarde `True` krijgen en alle andere cellen de waarde `False`.
4. Visualiseer de array die je bekomt in (2) met de functie `imshow()`.
5. Gebruik deze kaart om te voorspellen welke delen van Europa zullen overstromen wanneer de zeespiegel 3 m stijgt.
   (a) Het resultaat dient een NumPy array (een matrix) te zijn met dezelfde dimensies als de originele kaart. Overstroomde gebieden krijgen de waarde `True`, niet-overstroomde gebieden krijgen de waarde `False`.
   (b) Eenvoudige manier: selecteer alle cellen waarvoor de waarde lager is dan 3. Visualiseer ook het resultaat.
6. Het resultaat uit (5) is een figuur met daarop het (iets kleinere) Europa na de stijging van de zeespiegel. Maak nu een gelijkaardige figuur waarop enkel de gebieden te zien zijn die onder water zullen staan na de stijging met 3 m, maar die droog waren voor de stijging (op deze figuur zullen waarschijnlijk vooral de kuststreken naar voor komen). Doe dit:
   (a) Door gebruik te maken van 2 geneste `for`-lussen.
   (b) Zonder lussen te gebruiken. **Tip**: gebruik de arrays die je bekomt in deelopdrachten (3) en (5).
7. Lage gebieden die ingesloten zijn door hoger gelegen gebieden zullen mogelijks niet overstromen door een zeespiegelstijging, zorg dat deze gebieden niet als overstroomd worden beschouwd. Visualiseer ook het resultaat. Tip: Gebruik het principe van het bosbrandmodel om de zee te laten “groeien”.

Opdracht 8.18 (`vallei.py`)

De figuur hieronder toont de doorsnede van de vallei van een rivier. Het assenstelsel komt hierbij overeen met de rij/kolomnummering in `numpy`. We onderscheiden lucht (waarde 0), water (waarde 1) en land (waarde 2) in deze doorsnede (voorbeeld `vallei1.txt`)



Grafische voorstelling valleidoorsnede



vallei1.txt

De figuur rechts kan je bekomen op basis van de volgende instructies.

```
import numpy as np
import matplotlib.pyplot as plt
doorsnede1 = np.loadtxt("vallei1.txt")
plt.imshow(doorsnede1, cmap="gray_r")
```

1. Implementeer de functie `vrijWateroppervlak()`, die een doorsnede (`numpy` array) aanvaardt als input, en de breedte retourneert van het deel van het water dat aan lucht grenst, m.a.w. het aantal water-pixels (1) dat bovenaan grenst aan de lucht (0). Bekijk de bovenstaande figuur voor een voorbeeld.

   ```
   >>> doorsnede1 = np.loadtxt("vallei1.txt")
   >>> vrijWateroppervlak(doorsnede1)
   5
   ```

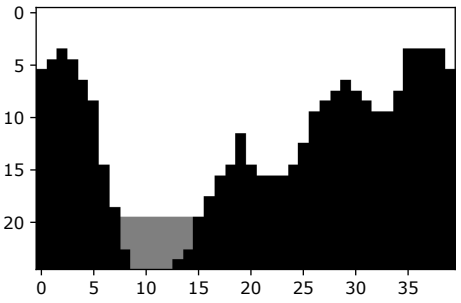
2. Implementeer de functie `toppen`, die een doorsnede (`numpy` array) aanvaardt als input, en een `list` van tuples retourneert met daarin de rij- en kolomindices van de heuveltoppen in de doorsnede. Een heuveltop is landpixel waarvoor geldt dat zijn linker-, rechter-, boven- en onderbuur lager gelegen zijn dan hijzelf. Een plateau (zoals uiterst rechts in het voorbeeld) is geen top.

   ```
   >>> doorsnede1 = np.loadtxt("vallei1.txt")
   >>> toppen(doorsnede1)
   [(4, 2), (12, 19), (7, 29)]
   ```

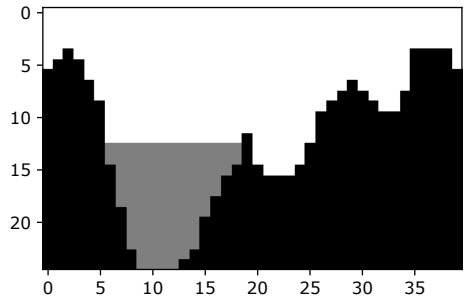
3. (*) Implementeer de functie `verhoogWaterpeil()`, die op basis van een gegeven doorsnede berekent wat er gebeurt wanneer het waterpeil in de rivier stijgt. Hou er rekening mee dat water niet doorheen landpixels kan vloeien. Deze functie heeft twee parameters:

- `doorsnede`: een `numpy` array die een doorsnede voorstelt (vb. deze in `vallei1.txt`)
- `waterpeil`: een integer (een rij-index) die aangeeft tot en met welke rij het waterniveau stijgt. Er moet steeds een stijging zijn tegenover het originele beeld. Het waterpeil moet niet kunnen dalen.

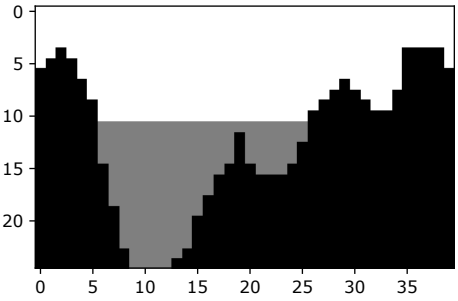
Deze functie retourneert een `numpy` array die de toestand voorstelt na de stijging. De onderstaande figuren tonen enkele voorbeelden van stijgingen.



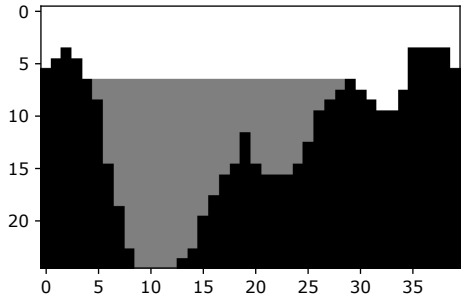

```
plt.imshow(verhoogWaterpeil(doorsnede1, 20))
```



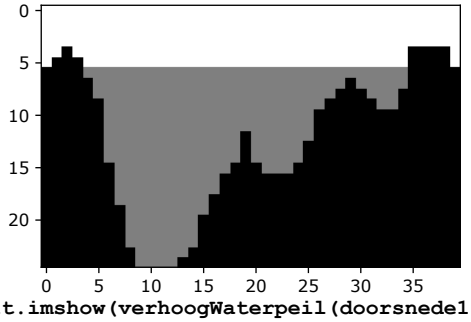

```
plt.imshow(verhoogWaterpeil(doorsnede1, 13))
```



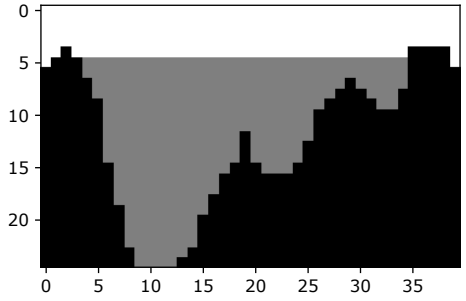

```
plt.imshow(verhoogWaterpeil(doorsnede1, 11))
```




```
plt.imshow(verhoogWaterpeil(doorsnede1, 7))
```




```
plt.imshow(verhoogWaterpeil(doorsnede1, 6))
```




```
plt.imshow(verhoogWaterpeil(doorsnede1, 5))
```

Opdracht 8.19 (`cijferreeks.py`)

Een woordzoeker is een raster van letters waarin een aantal woorden verborgen zitten. Een getallenzoeker volgt een gelijkaardig principe, maar maakt gebruik van cijfers tussen `0` en `9`, waarin een aantal cijferreeksen verborgen zijn. Zo komt de cijferreeks `[3, 7, 2]` tweemaal horizontaal voor in de matrix `M` (éénmaal in de 1ste rij en éénmaal in de 3de rij). De cijferreeks `[3, 5]` komt éénmaal verticaal voor in de matrix `P` (in de 4de kolom) maar niet in de matrix `M`.

$$M = \begin{bmatrix} 8 & 1 & 1 & 3 & 7 & 2 \\ 9 & 1 & 5 & 1 & 6 & 8 \\ 3 & 7 & 2 & 6 & 2 & 5 \\ 0 & 0 & 6 & 3 & 8 & 1 \\ 5 & 4 & 2 & 2 & 5 & 1 \end{bmatrix}, \qquad P = \begin{bmatrix} 0 & 1 & 8 & 4 & 1 & 2 & 0 & 4 \\ 1 & 0 & 7 & 5 & 7 & 7 & 8 & 8 \\ 8 & 5 & 2 & 3 & 4 & 1 & 2 & 5 \\ 4 & 5 & 7 & 5 & 0 & 7 & 8 & 7 \end{bmatrix}$$

1. Implementeer een functie `zoekCijferreeks()` die een cijferreeks (type *list*), een matrix van cijfers (2D-`ndarray`) en een oriëntatieletter (type *string*) `h` (voor **h**orizontaal) of `v` (voor **v**erticaal) als argumenten aanvaardt. De functie retourneert het aantal keer (type *int*) dat de cijferreeks in de opgegeven volgorde horizontaal (van links naar rechts) of verticaal (van boven naar onder) voorkomt in de matrix van cijfers.

```
>>> M = np.array([[8, 1, 1, 3, 7, 2],[9, 1, 5, 1, 6, 8],
                  [3, 7, 2, 6, 2, 5],[0, 0, 6, 3, 8, 1],
                  [5, 4, 2, 2, 5, 1]])
>>> P = np.array([[0, 1, 8, 4, 1, 2, 0, 4],[1, 0, 7, 5, 7, 7, 8, 8],
                  [8, 5, 2, 3, 4, 1, 2, 5],[4, 5, 7, 5, 0, 7, 8, 7]])
>>> zoekCijferreeks([3, 7, 2], M, "h")
2
>>> zoekCijferreeks([3, 5], P, "v")
1
>>> zoekCijferreeks([3, 5], M, "v")
0
```

2. (*) Cijferreeksen kunnen ook op een diagonaal (van linksboven naar rechtsonder) of op een anti-diagonaal (van rechtsboven naar linksonder) voorkomen. Zo komt de cijferreeks `[8, 5]` driemaal voor in matrix `P` op diagonalen, en de cijferreeks `[2, 0, 5]` éénmaal in de matrix `M` op een anti-diagonaal.
   Schrijf nu een functie `zoekCijferreeks2()` zodat die het aantal cijferreeksen op de **d**iagonaal (oriëntatieletter `d`) en **a**nti-diagonaal (oriëntatieletter `a`) kan bepalen (bekijk ook de tips onder het voorbeeld).

```
>>> zoekCijferreeks2([8, 5], P, "d")
3
>>> zoekCijferreeks2([2, 0, 5], M, "a")
1
```

```
>>> zoekCijferreeks2([1, 2, 3, 4], M, "d")
0
>>> zoekCijferreeks2([0, 7, 4, 5], P, "a")
1
```

De volgende tips kunnen daarbij helpen (je bent uiteraard niet verplicht om deze tips te gebruiken):

- De functie `diag()` aanvaardt een 2D-array (een matrix) als argument en retourneert een 1D-array met daarin enkel de diagonaalelementen zoals hieronder getoond wordt:

```
>>> A = np.array([[1, 2], [3, 4]])
>>> A
array([[1, 2],
       [3, 4]])

>>> np.diag(A)
array([1, 4])
```

- De anti-diagonaalelementen kan je eenvoudig extraheren door de `diag()` te combineren met de functie `fliplr()` (bekijk eventueel de help-pagina van deze functie, zie ook Sectie 8.2.15).

9

File input/output en Exception handling

9.1 Inleiding

In Sectie 4.5 maakten we een eerste keer kennis met bestanden en bestandslocaties. In hetgeen volgt, bekijken en bespreken we bestanden meer in detail.

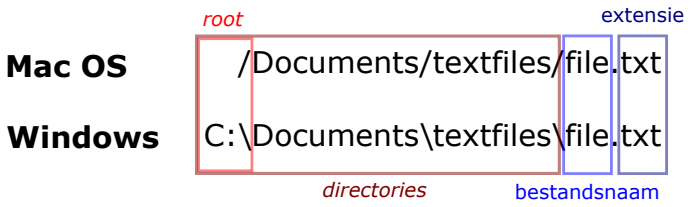
9.2 Omschrijving van een bestand (*file*)

Een bestand is een geordende verzameling van gegevens in elektronische vorm die door een apparaat (in dit geval een computer) onder een naam kan worden aangesproken en behandeld. Bestanden bevinden zich meestal op een opslagmedium zoals een harde schijf, cd-rom of USB-stick. Bestanden kunnen op verschillende manieren worden gegroepeerd in bestandstypes. In deze cursus onderscheiden we maar twee types bestanden: **tekst**bestanden en **binaire** bestanden. Met **tekstbestanden** kwamen we reeds vele malen in aanraking (zie Hoofdstuk 4); de gegevens in deze bestanden zijn *human-readable* en worden veelal gecodeerd als ASCII of Unicode (bv. UTF-8) tekst. Tekstbestanden zijn leesbaar wanneer ze geopend worden in een teksteditor of webbrowser. Bestanden die andere coderingsschema's gebruiken (bv. Word bestanden of foto-bestanden zoals PNG-bestanden) waardoor ze niet zo eenvoudig door een mens te lezen zijn, worden **binaire bestanden** genoemd. Voor het vervolg van dit hoofdstuk spitsen we ons enkel toe op tekstbestanden.

9.3 De bestandslocatie en de `os` module

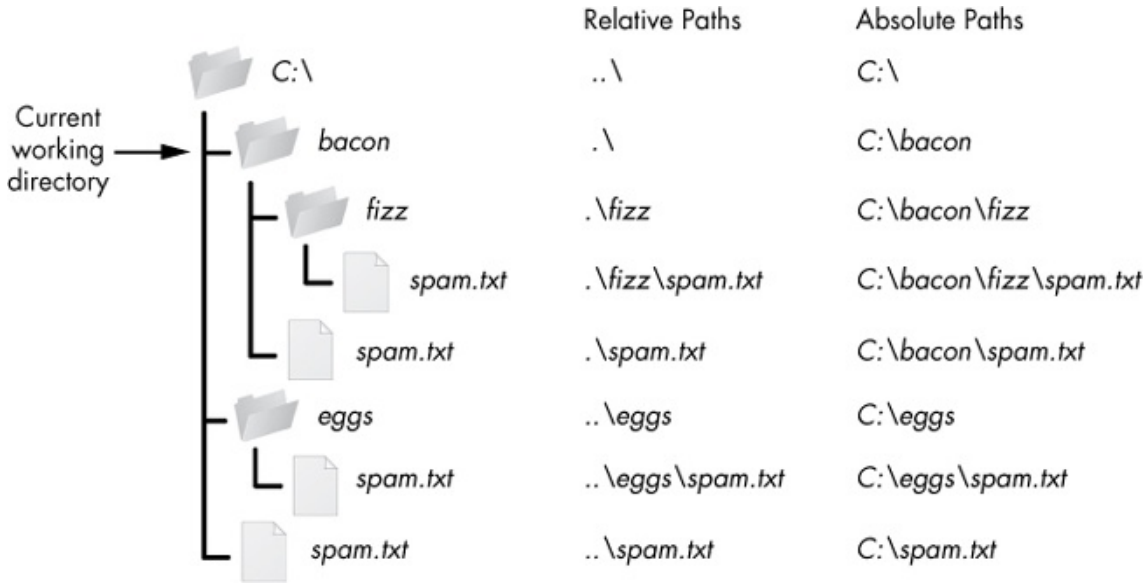
Vooraleer we een bestand kunnen gebruiken in Python, moet we eerst weten waar het zich bevindt. "Waar" slaat dan niet enkel op het opslagmedium (meestal is dit gewoon de harde schijf van de computer waarop we aan het werken zijn), maar vooral op de **locatie** in het **bestandssysteem** van desbetreffend opslagmedium. Deze locatie wordt voor elk bestand aangegeven met een uniek ***path***, dat aangeeft wat de **bestandsnaam** is en in welke ***directory*** het bestand zich bevindt. Hoe dit

path precies wordt opgebouwd, hangt af van het besturingssysteem waarmee je werkt (Figuur 9.1).



Figuur 9.1: Opbouw van een *path* voor een Windows en MacOS besturingssysteem.

De bestandslocatie kan ook relatief t.o.v. een bepaalde directory worden aangegeven. In dat geval spreken we van een ***relative path***: een *path* zoals in Figuur 9.1 wordt het ***absolute path*** genoemd. In de praktijk wordt bijvoorbeeld vaak het *path* gebruikt, relatief t.o.v. de huidige werkdirectory (***working directory***) zoals geïllustreerd in Figuur 9.2.



Figuur 9.2: Illustratie van relative vs. absolute path op een Windows machine.

De module `os` bevat een aantal functionaliteiten om vlot met bestandslocaties aan de slag te kunnen. Tabel 9.1 geeft een overzicht van `os`-functies die we verder in deze cursus zullen gebruiken en Fragment 9.1 illustreert een aantal van deze.

Tabel 9.1: Veelgebruikte functies uit de `os` module.

Functie	Toelichting
`os.getcwd()`	retourneert het *path* van de huidige werkdirectory
`os.chdir()`	verandert de huidige werkdirectory
`os.listdir()`	retourneetr lijst met de namen van alle items in een directory
`os.mkdir()`	maakt een nieuwe directory aan
`os.path.join()`	voegt strings samen tot een *path*, cf. het besturingssysteem

Fragment 9.1: Illustratie van het gebruik van enkele functies van de `os` module op een Windows machine

```
>>> import os
>>> os.mkdir("foo")            # maak directory "foo" aan

>>> os.path.join("foo", "bar") # maak een path-string
'foo\bar'

>>> os.mkdir(pad)              # maak subdirectory

>>> os.listdir("foo")          # geef inhoud van "foo"
['bar']

>>> os.mkdir("foo")            # fout indien directory al bestaat
FileExistsError: [WinError 183]
Cannot create a file when that file already exists: 'foo'
```

9.4 Inlezen van tekstbestanden

Om toegang te krijgen tot een bestand in Python moet je een **connectie** maken tussen de Python sessie en het bestand op de harde schijf. Via deze connectie kunnen data uitgewisseld worden tussen het medium waarop het bestand zich bevindt en het programma.
Die zogenaamde connectie wordt voorgesteld door een bestandsobject (*file object*) in Python dat gecreëerd wordt wanneer een connectie opgesteld wordt. Dit bestandsobject gebruikt het geheugen van de computer als een soort buffer om data in op te slaan wanneer data uitgewisseld worden tussen de harde schijf en het programma. De functie `open()` creëert zo'n connectie en retourneert het file object dat deze connectie voorstelt. Alle volgende acties worden uitgevoerd via dit bestandsobject (ook wel *file descriptor* of *stream* genoemd). Een bestand kan geopend worden voor het *inlezen* of *wegschrijven* van data (of beide) via dit bestandsobject.

9.4.1 Bestanden uit de huidige *working directory* inlezen

Ter illustratie maken we een tekstbestandje in een teksteditor met de naam `temp.txt` en de volgende inhoud:

```
Eerste regel
Tweede regel
Derde regel
Vierde regel
[BLANCO LIJN]
```

Opmerking: de tekst `[BLANCO LIJN]` moet je **niet** in het tekstbestandje schrijven maar stelt een blanco lijn voor. Dit doe je door op **Enter** te drukken op het einde van `Vierde regel`. Op die manier hebben we op het einde van elke regel een *end-of-line* karakter `"\n"` (**niet zichtbaar** in de teksteditor).
Stel de werkmap (*working directory*) in op de directory waar `temp.txt` staat (met `os.chdir()`, zie Sectie 9.3). We zullen dit tekstbestandje openen en de inhoud ervan regel per regel inlezen en weergeven op het scherm. Voer daarvoor de code uit van het volgende Fragment 9.2:

Fragment 9.2: `lees_bestand.py`

```
afile = open("temp.txt", "r")
for line in afile:
    print(line, end = "")
afile.close()
```

We bekijken deze instructies één voor één in meer detail:

- Met `open("temp.txt", "r")` wordt een connectie gelegd tussen de data in `temp.txt` en de Python sessie. Deze functie neemt twee argumenten: (1) de bestandsnaam (hier: `"temp.txt"`), en (2) een code (de "mode") die de aard van de connectie aangeeft (hier: `"r"` voor *read-only*). Het bestandsobject voor deze connectie wordt geretourneerd in `afile`.
- Met de `for`-lus `for line in afile:` wordt elke regel één voor één ingelezen. **Merk op dat elke regel wordt ingelezen als *string*!** De afzonderlijke regels worden achtereenvolgens toegekend aan de variabele `line`.
- Het statement `print(line, end = "")` geeft elke regel weer op het scherm.
- Met de instructie `afile.close()` wordt de connectie tussen het bestand en ons programma verbroken.

Doordat de mode-optie `"r"` gekozen werd in de functie `open()`, kunnen we dus het bestand in dit geval **enkel lezen** (we kunnen er niets in schrijven mocht we dit willen proberen). Als output (op het scherm) zou je het volgende moeten krijgen:

```
Eerste regel
Tweede regel
Derde regel
Vierde regel
[BLANCO LIJN]
```

met onderaan een blanco lijn.
Merk op dat `end = ""` aanwezig is als optie in het **`print`**-statement. Toch wordt elke regel in het bestand op een afzonderlijke regel weergegeven! Dit komt omdat in het bestand op elke regel reeds een *end-of-line* karakter `"\n"` aanwezig is (omdat je op **Enter** drukte na elke regel tekst).

Opdracht 9.1

We proberen enkele varianten op het **`print`**-statement uit Fragment 9.2.

- Neem de instructies in Fragment 9.2 op in een script en vervang daarin het **`print`**-statement door `print(line)` (dus zonder optie `end = ""`) en voer je script opnieuw uit. Is het resultaat zoals je verwachtte?
- Ga, ter opfrissing van je geheugen, na wat de *string*-methode `strip()` doet, in het bijzonder met de *end-of-line* karakter `"\n"` in een *string*. Vervang vervolgens in de code uit het vorige puntje nu `print(line)` door `print(line.strip())` en voer je programma opnieuw uit. Werd de *end-of-line* karakter `"\n"` verwijderd in elke regel voordat je het **`print`**-statement oproept?

Opdracht 9.2 (`amino_slc.py`)

Schrijf een programma waarmee je de inhoud van het bestand `amino_slc.csv` inleest en **tijdens het inlezen** de volgende tabel op het scherm weergeeft:

```
SLC | Aminozuur
--------------------
I   | Isoleucine
L   | Leucine
V   | Valine
F   | Phenylalanine
M   | Methionine
...
R   | Arginine
```

Elke regel in het bestand `amino_slc.csv` bestaat uit de naam van een aminozuur en de SLC (*single letter code*) gescheiden door een komma (bv. `Methionine,M`). Baseer je op Fragment 9.2 om de inhoud van het bestand `amino_slc.csv` in te lezen. Je kan de methoden `strip()` (Sectie 6.4.2.3) en `split()` (Sectie 6.4.3) gebruiken om je ingelezen data te verwerken.

Zoals we reeds opmerkten wordt elke regel in het codefragment 9.2 ingelezen als *string*.

Getalwaarden (int of float) als data

Indien je data uit getalwaarden bestaan en je wenst bewerkingen uit te voeren met deze getalwaarden dan zal je een **conversie** moeten toepassen **naar** een `int` of `float`.

Een voorbeeld

We maken in een teksteditor een tekstbestand `data.txt` aan met de volgende inhoud:

```
8.1
9
4.6
5.2
7
```

In Fragment 9.3 wordt het bestand `data.txt` ingelezen en wordt de som berekend van al de getallen.

Fragment 9.3: `data_som.py`

```
1 afile = open("data.txt", "r")
2 som = 0
3 for getal_str in afile:
4     som = som + float(getal_str)
5 afile.close()
6 print("De som is: ", som)
```

In `data.txt` bestaat elke regel uit één getal. Bij het inlezen met `for getal_str in afile:` wordt elke afzonderlijke regel geplaatst in `getal_str` (cf. het getal als *string*). Om de som te berekenen moeten we `getal_str` omzetten naar een *float*, zoals in: `som = som + float(getal_str)`.

Opdracht 9.3 (`rainfall.py`)

Meteorologen gebruiken een regenmeter (pluviometer) om de hoeveelheid neerslag gedurende een bepaalde periode op te vangen en te meten. Hierbij wordt de hoeveelheid neerslag aangegeven in millimeters (mm). Eén millimeter komt overeen met 1 liter water per vierkante meter. In het bestand `rainfall_brussels.txt` staan gemiddelde hoeveelheden neerslag per maand voor een locatie in Brussel. Schrijf een programma dat de inhoud van het bestand `rainfall_brussels.txt` inleest en **tijdens het inlezen** de volgende tabel op het scherm weergeeft:

```
Maand Hoogte (mm)
Jan    81.7
Feb    51.2
Mar    80.6
Apr    52.6
May    74.0
Jun    74.3
Jul    58.4
Aug    42.2
Sep    69.2
Oct    85.0
Nov    61.3
Dec    68.3
Jaar  798.8
```

Opmerkingen/tips:

- De eerste regel in het bestand `rainfall_brussels.txt` is informatief. **Hoe zou je deze regel kunnen overslaan?**
- Vanaf de tweede regel staat telkens de afkorting van de maand en de gemiddelde hoeveelheid neerslag.
- Lees zowel de maand en de hoeveelheid in om de bovenstaande tabel te maken.
- Hou ook de totale hoeveelheid neerslag bij om deze op het einde in de tabel weer te geven.

9.4.2 Bestanden uit een andere directory inlezen

In de praktijk zullen we vaak bestanden uit een andere directory dan de huidige working directory willen inlezen. Dit kan eenvoudig door in plaats van de bestandsnaam, het volledige *path* als argument mee te geven aan de functie `open()`. Dit wordt geïllustreerd in Fragment 9.4, voor het geval `temp.txt` in de directory `Documents` staat op een computer met een Windows besturingssysteem.

Fragment 9.4: `lees_bestand.py`

```
1 afile = open("C:\\Documents\\temp.txt", "r")
2 for line in afile:
3         print(line, end = "")
4 afile.close()
```

Het gebruik van het absolute *path* heeft als nadeel dat gebruikers van de code het in te lezen bestand op exact dezelfde locatie in hun bestandssysteem moeten hebben staan, wat uitwisselen van code bemoeilijkt. Bovendien kunnen absolute paden al snel **erg lange strings** worden wat de code minder leesbaar maakt. Vandaar dat het vaak eenvoudiger is om een *path* relatief t.o.v. de huidige working directory te gebruiken.

Opdracht 9.4 (`ph_metingen.py`)
De directory `pH_metingen` bevat een reeks metingen van een pH-sensor, genomen gedurende een dag. Elke meting werd door de sensor opgeslagen als een afzonderlijk tekstbestand met als naam het tijdstip waarop gemeten werd. Schrijf een script waarin je de volgende zaken uitvoert:

1. Stel de directory `bestanden_oefeningen` als werkmap (working directory) in. Gebruik hiertoe de functies van de `os` module uit Tabel 9.1.
2. Lees de metingen in uit de tekstbestanden in de directory `pH_metingen`.
3. Toon onderstaande tabel op het scherm.

Tip: de rij met koppeltekens kan je eenvoudig printen met `print("-"*37)`.

```
locatie:    IO-exceptions\pH metingen
bestand                          waarde
-------------------------------------
00h06m.txt                        7.037
01h30m.txt                        6.931
03h08m.txt                        7.055
...
21h40m.txt                        7.050
21h55m.txt                        6.631
22h30m.txt                        6.972
-------------------------------------
gemiddelde pH                     6.966
```

9.5 Wegschrijven van tekstbestanden

Voor het wegschrijven van bestanden moet je eerst, net zoals bij het inlezen, een bestandsobject aanmaken dat instaat voor de connectie tussen het bestand op de harde schijf en je programma. Dit doe je met de functie `open()` met de mode-optie `"w"` (voor *write-only*). Indien je een bestand dat niet bestaat opent met de mode-optie `"w"`, dan zal dit bestand aangemaakt worden. **Pas op**: indien je een bestand opent met de mode-optie `"w"` dat **wel** bestaat, dan zal dit bestand **overschreven** worden! Later zullen we andere mode-opties tegenkomen.
Eénmaal een bestand geopend is, kan je ernaar wegschrijven door `file = afile`[1] toe te voegen als argument in het `print`-statement. In het volgende code fragment worden de getallen 1 tot en met 10 in een bestand weggeschreven voorafgegaan door een informatieve regel.

Fragment 9.5: `schrijf_bestand.py`

```
1 een_file = open("temp2.txt", "w")
2 print("De getallen van 1 t.e.m. 10:", file = een_file)
3 for i in range(1, 11):
4     print(i, file = een_file)
5 een_file.close()
```

We bekijken de verschillende instructies één voor één meer in detail:

- De instructie `open("temp2.txt", "w")` stelt een connectie op tussen `temp2.txt` (in de huidige werkdirectory) en je programmaatje. Doordat de mode-optie `"w"` gekozen werd in de functie `open()`, kan je enkel data in het bestand wegschrijven. Het bestandsobject voor deze connectie wordt geretourneerd in `een_file`.
- Bij het eerste `print`-statement wordt de *string* `"De getallen van 1 t.e.m. 10:"` in het bestand `temp2.txt` met bestandsobject `een_file` weggeschreven. Dit gebeurt omwille van het argument `file = een_file` bij het `print`-statement.

[1] In de verondestelling dat `afile` het bestandsobject voor de connectie naar het bestand bevat.

- Binnen de *for*-lus wordt met met `print(i, file = een_file)` elk *integer* getal `i` weggeschreven in `temp2.txt` met als bestandsobject `een_file`. Merk op dat elk *integer* getal `i` eerst **impliciet** omgezet wordt naar een *string* binnen het `print`-statement!
- Met `een_file.close()` wordt de connectie tussen het bestand en ons programma verbroken.

In het bestand `temp2.txt` zou je het volgende moeten zien:

```
De getallen van 1 t.e.m. 10:
1
2
3
4
5
6
7
8
9
10
[BLANCO LIJN]
```

Net zoals bij het inlezen van bestanden, kunnen tekstbestanden ook worden weggeschreven naar een andere directory dan de huidige werkdirectory. Dit kan weer door een (absoluut of relatief) *path* te gebruiken als argument van `open()`. **Let op**: alle *directories* in het *path* moeten al bestaan voordat het tekstbestand kan worden geopend, anders krijg je een foutmelding zoals getoond in onderstaand fragment:

```
>>> een_file = open("temp_dir\\temp2.txt", "w")
FileNotFoundError: [Errno 2] No such file or directory:
"temp_dir\\temp2.txt"
```

Opdracht 9.5

We proberen enkele varianten op het `print`-statement uit Fragment 9.5.

- Zorg dat het bestand `temp2.txt` in de directory `tekstbestand_voorbeeld` komt te staan. Gebruik hiertoe de os-module (Tabel 9.1).
- Zorg ervoor door gebruik te maken van *string*-concatenatie dat in het bestand `temp2.txt` vóór elk *integer* getal de *string* `Getal:` staat zoals bv.: `Getal: 5` i.p.v. van enkel `5`.
- Tracht hetzelfde resultaat te bekomen als in het vorige puntje, maar nu met een f-string binnenin het `print`-statement.

Opdracht 9.6 (`leerling_scores.py`)
Schrijf een script dat herhaaldelijk vraagt naar de (voor)naam van een leerling en zijn score. De naam en de score moeten weggeschreven worden in het bestand `leerling_scores.txt` en ziet er bv. als volgt uit:

```
Naam                 Punten
----------------------------
Tom                     6.5
Viviane                 8.3
Leon                    2.6
Helena                  9.8
```

Schrijf de naam en de score systematisch (dus één voor één) weg naar het bestand. Pas wanneer een lege *string* als naam ingegeven wordt, moet je programma stoppen met vragen naar de naam en score, en moet het bestand gesloten worden.

9.6 Inlezen en wegschrijven van tekstbestanden in een programma

We zullen nu de vorige concepten verenigen in een eenvoudig programmaatje dat 2D-cartesische coördinaten omzet in poolcoördinaten. Dit programma opent twee bestanden simultaan: één om te lezen (uit `2d_coord_c.txt`), en een ander om te schrijven (naar `2d_coord_cp.txt`). De 2D-cartesische coördinaten staan in het bestand `2d_coord_c.txt`, zoals hieronder:

```
  5.247   8.058
 -8.410   7.948
  1.688  -2.885
 -0.874  -4.060
```

Op elke regel staat een x- en y-coördinaat gescheiden door één of meerdere spaties. Het programma leest regel per regel uit `2d_coord_c.txt` en zet de coördinaten om naar *floats*. Met deze *float* coördinaten (x, y) worden de poolcoördinaten (r, θ) berekend. Hierbij moet de hoek θ van onze poolcoördinaten in het interval $[-180°, +180°]$ liggen. Merk op dat bij de berekening van $arctan(y/x)$ er in theorie twee oplossingen zijn voor θ. Om de juiste oplossing te kiezen moet je kijken naar de tekens van x en y, dit gebeurt met de `if`-statements. In het bestand `2d_coord_cp.txt` worden nu zowel de cartesische als de poolcoördinaten weggeschreven op één regel voorzien van `x =`, `y =`, `r =` en `theta =` bij de juiste getallen.

Fragment 9.6: `coordinaten_omzetten.py`

```
1 import math
2
3 inp_file = open("2d_coord_c.txt", "r")
4 out_file = open("2d_coord_cp.txt", "w")
```

```
5
6  for line_str in inp_file:
7      x_str, y_str = line_str.split()
8      x = float(x_str)
9      y = float(y_str)
10     r = math.sqrt(x**2 + y**2)
11     t = math.degrees(math.atan(y/x))
12     if x < 0:
13         if y > 0:
14             t += 180.0
15         else:
16             t -= 180.0
17     if x == 0:
18         if y > 0:
19             t = 90.0
20         else:
21             t = -90.0
22     print(f"x={x:7.3f}   y={y:7.3f}   r={r:7.3f}   theta={t:7.1f}",\
23           file = out_file)
24
25 out_file.close()
26 inp_file.close()
```

De inhoud van het bestand `2d_coord_cp.txt` zou er als volgt moeten uitzien:

```
x=  5.247   y=  8.058   r=  9.616   theta=   56.9
x= -8.410   y=  7.948   r= 11.571   theta=  136.6
x=  1.688   y= -2.885   r=  3.343   theta=  -59.7
x= -0.874   y= -4.060   r=  4.153   theta= -102.1
```

Opdracht 9.7 (`woorden_omkeren.py`)

Schrijf een programma dat woorden inleest die allen op een afzonderlijke regel staan in het bestand `woorden.txt`, al deze woorden omkeert, en wegschrijft in het bestand `woorden_omgekeerd.txt`. Als bijvoorbeeld `woorden.txt` de volgende woorden bevat

```
uurwerk
straat
geodriehoek
lepel
aardvark
optimalisatie
```

dan moet, na het uitvoeren van je programma, het bestand `woorden_omgekeerd.txt` het volgende bevatten:

```
krewruu
taarts
```

```
keoheirdoeg
lepel
kravdraa
eitasilamitpo
```

Je programma moet tevens het oorspronkelijke en omgekeerde woord op het scherm weergeven zoals:

```
woord: uurwerk          omgekeerd: krewruu
woord: straat           omgekeerd: taarts
woord: geodriehoek      omgekeerd: keoheirdoeg
woord: lepel            omgekeerd: lepel
woord: aardvark         omgekeerd: kravdraa
woord: optimalisatie    omgekeerd: eitasilamitpo
```

Bij het weergeven van de woorden op het scherm mag je ervan uitgaan dat de woorden niet langer zijn dan 15 karakters.

Opdracht 9.8 (`palindromen_zoeken.py`)

Een palindroom (of keerwoord) is een woord waarin de letters symmetrisch gerangschikt zijn, zodanig dat het woord van achter naar voren gelezen hetzelfde is als van voor naar achter. In het bestand `nl_dic.txt` staan heel wat Nederlandse woorden onder elkaar getabeleerd, in toenemend aantal letters per woord.

Schrijf een programma dat deze Nederlandse woorden inleest, nagaat welke woorden palindromen zijn, en vervolgens deze palindromen wegschrijft in het bestand `nl_palindromen.txt`. Geef de gevonden palindromen ook weer op het scherm. Je programmaatje moet eveneens bijhouden hoeveel woorden werden ingelezen en hoeveel palindromen er gevonden werden. Deze aantallen moeten op het scherm weergegeven worden.

Let op, bepaalde woorden beginnen met een hoofdletter, je moet er dus voor zorgen dat je hoofdletterongevoelig werkt. Een mogelijk output op het scherm is:

```
ede
aha
ara
bob
...
serres
staats
stoots
meeneem
rotator
parterretrap
Totaal aantal woorden:        225311
Aantal gevonden palindromen:  84
```

9.7 Bestanden aanmaken (en overschrijven)

Wanneer je een bestand tracht te openen om te **lezen** en dat bestand bestaat niet op de harde schijf, dan zal een foutmelding optreden. Indien het bestand wel bestaat op de harde schijf (in de juiste directory!), dan zal een connectie gecreëerd worden en zullen we de data kunnen inlezen.
Wanneer je een bestand tracht te openen om te **schrijven** en dat bestand bestaat niet op de harde schijf, dan zal het bestand aangemaakt worden (default in de werkdirectory). Indien het bestand wel bestaat, dan zal dit bestand verwijderd worden, d.w.z. de inhoud zal gewist worden, en alle nieuwe data zal er standaard in de plaats komen.
Wanneer je enkel een bestandsnaam meegeeft, zal je programma standaard in de werkdirectory inlezen of wegschrijven. Wanneer je een absoluut of relatief *path* meegeeft, moeten alle directoeries in het path reeds bestaan, anders krijg je een foutmelding.
De modes waarmee een bestand geopend kan worden en hun effect wanneer het bestand al dan niet bestaat staan getabeleerd in Tabel 9.2.

Tabel 9.2: Modes voor het openen van een bestand.

Mode	Hoe geopend	Bestand bestaat	Bestand bestaat niet
`"r"`	enkel lezen	opent het bestand	fout
`"w"`	enkel schrijven	wist de inhoud	creëert en opent nieuw bestand
`"a"`	enkel schrijven	inhoud ongewijzigd, nieuwe data aan einde toegevoegd	creëert en opent nieuw bestand
`"r+"`	lezen en schrijven	leest en overschrijft vanaf het begin van het bestand	fout
`"w+"`	lezen en schrijven	wist de inhoud	creëert en opent nieuw bestand
`"a+"`	lezen en schrijven	inhoud ongewijzigd, lezen kan mits gebruik van `fseek` en schrijven aan einde	creëert en opent nieuw bestand

Opdracht 9.9 (`telefoonnummers_toevoegen.py`)
Schrijf een programma dat:

1. bestaande namen en telefoonnummers kan weergeven op het scherm die ingelezen werden uit het bestand `telefoonnummers.txt`, en
2. nieuwe namen en telefoonnummers (aan het einde) toevoegen in ditzelfde bestand.

In het bestand `telefoonnummers.txt` moeten de naam en telefoonnummer op één regel geschreven worden, gescheiden door een komma. Je programmaatje moet telkens een menuutje weergeven met de keuzes, zoals bv.:

```
[1] Telefoonnummers tonen
[2] Naam en telefoonnummer toevoegen
[9] Afsluiten
Maak je keuze:
```

- Bij keuze 1 moeten de namen en telefoonnummers weergegeven worden op het scherm die reeds in het bestand `telefoonnummers.txt` staan.
- Bij keuze 2 moet gevraagd worden naar een naam en telefoonnummer en moet deze gegevens weggeschreven worden in `telefoonnummers.txt` na alle andere bestaande gegevens in het bestand.
- Bij keuze 9 moet het programma afsluiten.
- Bij elke andere keuze moet een foutmelding weergegeven worden op het scherm.

In volgend voorbeeld wordt de werking van het programmaatje geïllustreerd.

```
[1] Telefoonnummers tonen
[2] Naam en telefoonnummer toevoegen
[9] Afsluiten
Maak je keuze: 2
--------------------------
Geef naam: Karin
Geef tel.: 056338871
[1] Telefoonnummers tonen
[2] Naam en telefoonnummer toevoegen
[9] Afsluiten
Maak je keuze: 2
--------------------------
Geef naam: Tony
Geef tel.: 0475285410
[1] Telefoonnummers tonen
[2] Naam en telefoonnummer toevoegen
[9] Afsluiten
Maak je keuze: 1
--------------------------
Naam:  Karin
Tel:   056338871
--------------------------
Naam:  Tony
Tel:   0475285410
--------------------------
[1] Telefoonnummers tonen
[2] Naam en telefoonnummer toevoegen
[9] Afsluiten
Maak je keuze: 7
--------------------------
Foute keuze!
[1] Telefoonnummers tonen
[2] Naam en telefoonnummer toevoegen
[9] Afsluiten
Maak je keuze: 9
--------------------------
Programma stopt.
```

De inhoud van het bestand `telefoonnummers.txt` is nu:

```
Karin,056338871
Tony,0475285410
```

Je mag ervan uitgaan dat je geen gegevens zal tonen (keuze 1) indien het bestand `telefoonnummers.txt` in het begin leeg is.

9.8 Andere methoden voor bestandstoegang

Naast de toegangsmethoden die hiervoor besproken werden, bestaan nog tal van andere functies, waaronder `readline()`, `readlines()`, `read()`, `write()`, `writelines()` de belangrijkste zijn.

- `readline()`: retourneert de huidige regel uit een tekstbestand als *string*
- `readlines()`: leest de volledige inhoud van het bestand en plaatst elke aparte regel als een element in een *list*
- `read()`: leest de volledige inhoud van het bestand in als één string
- `write(...)`: schrijft een string weg naar een bestand
- `writelines(...)`: schrijft een lijst van strings weg naar een bestand

Een uitgebreide beschrijving van deze methoden valt buiten het bestek van deze cursus.

9.9 Foutenbehandeling

Fouten kunnen optreden in programma's. We kunnen ze indelen in twee grote categorieën: *syntax* fouten en *runtime* fouten.
***Syntax* fouten** zijn fouten die optreden wanneer de code niet voldoet aan de Python codeerregels. Als voorbeeld zien we hieronder in een interactieve sessie de output wanneer we een : vergeten na een `for`-statement en wanneer we een aanhalingsteken van een *string* vergeten binnen een `print`-statement.

```
for i in range(5)
    print(i)

  File "<ipython-input-3-94439a011d38>", line 1
    for i in range(5)
                     ^
SyntaxError: invalid syntax
```

```
print('Geen aanhalingsteken op het einde)
  File "<ipython-input-6-0ee5d8e906d7>", line 1
    print('Geen aanhalingsteken op het einde)
                                              ^
SyntaxError: EOL while scanning string literal
```

***Runtime* fouten** zijn fouten die optreden, niet door een verkeerde *syntax* maar tijdens het uitvoeren van de code. We kunnen bijvoorbeeld een syntactisch juist Python programma schrijven dat tracht een geheel getal te delen door 0. Geen enkele Python regel weerhoudt ons van zo'n programma te schrijven maar de uitvoering ervan zal tot een fout leiden.

```
a = 7
b = 0
c = a / b
Traceback (most recent call last):

  File "<ipython-input-12-575d20dc0029>", line 3, in <module>
    c = a / b

ZeroDivisionError: division by zero
```

Zoals het bovenstaande voorbeeld illustreert, zal de Python *interpreter* in dit geval een ***exception*** opwerpen. **Een *exception* is een signaal dat aangeeft dat er een fout (of iets ongewoons) is opgetreden**. Wanneer een *exception* wordt opgeworpen, wordt de uitvoering van het programma onderbroken. De *interpreter* zal eerst deze *exception* afhandelen. De programmeur kan deze afhandeling zelf voorzien (zie verder), maar als dit niet het geval is zal het programma onmiddellijk beëindigd worden. Het afhandelen van de *exception* bestaat dan eenvoudigweg uit het printen van een foutboodschap op het scherm en een onmiddellijke beëindiging van de uitvoering.

Opdracht 9.10

Bekijk de onderstaande code en tracht voor jezelf uit te maken wat de code doet.

```
getallen = [3, 5, 9, 17, 33]
for i in range(len(getallen)):
    verschil = getallen[i+1] - getallen[i]
    print(verschil)
```

Voer deze code uit. Welke foutmelding krijg je en waarom? Met welke naam wordt deze fout aangeduid? Is dit een *syntax* of *runtime* fout? Pas de code (lichtjes) aan zodat de fout niet meer optreedt.

Opdracht 9.11

Bekijk de onderstaande code en tracht voor jezelf uit te maken wat de code doet.

```
getal = '5'
print("Het kwadraat is: ", getal**2)
```

1. Voer deze code uit. Welke foutmelding krijg je en waarom? Met welke naam wordt deze fout aangeduid? Is dit een *syntax* of *runtime* fout? Pas de code (lichtjes) aan zodat de fout niet meer optreedt.
2. Wanneer je de code hebt verbeterd, voer deze code opnieuw uit en geef als input geen geheel getal maar een *string*. Welke foutmelding krijg je en waarom? Met welke naam wordt deze fout aangeduid? Is dit een *syntax* of *runtime* fout?

Tot nu toe hebben we de *runtime* fouten grotendeels genegeerd. Bijvoorbeeld: indien een gebruiker een bestand tracht te open (om te lezen) dat niet bestaat dan zal het programma een *exception* opwerpen, abrupt stoppen met uitvoeren en er verschijnt een foutboodschap op het scherm. In de meeste gevallen is dit geen hulpvaardige oplossing en wensen we dit soort van *exceptions* af te handelen zonder dat het programma stopt. Door de *exception* te behandelen, kan duidelijk gemaakt worden wat er gaande is en eventueel een nieuwe kans geboden worden aan de gebruiker om een bestaand bestand te openen.

In Fragment 9.7 wordt bijvoorbeeld getracht een onbestaand bestand te openen.

Fragment 9.7: Script waarin onbestaand bestand wordt opgeroepen.

```
1 print("bestand openen...")
2 bestand = open("blablabla.txt", "r")
3 print("bestand lezen...")
4 inhoud = bestand.read()
5 print("bestand sluiten...")
6 bestand.close()
```

Wanneer we Fragment 9.7 uitvoeren, krijgen we:

```
bestand openen...
Traceback (most recent call last):

  File "<ipython-input-38-c93e418845a7>", line 2, in <module>
    bestand = open("blablabla.txt", "r")

FileNotFoundError: [Errno 2] No such file or directory: 'blablabla.txt'
```

De *exception* die wordt opgeworpen is een `FileNotFoundError` waaruit kan afgeleid worden dat het bestand `blablabla.txt` **niet bestaat in de directory waar de code wordt uitgevoerd**. Merk op dat de **rest van de instructies (bestand lezen en sluiten) niet meer uitgevoerd** worden.

Tot nu toe hebben we de volgende *exceptions* ontmoet:

- `ZeroDivisionError`: Deze exception wordt opgeworpen bij deling door nul. Bv. `5 / 0`
- `IndexError`: Deze exception wordt opgeworpen wanneer men indexeert met een index die buiten het bereik van de collectie valt. Bv. `"Hallo"[15]`
- `TypeError`: Deze exception wordt opgeworpen bij het toepassen van een operatie of functie op een object van een ongepast *type*. Bv. `1 + "a"`
- `ValueError`: Deze exception wordt opgeworpen wanneer men een functie of operatie toepast op een object waarvan het type correct is, maar de waarde niet. Bv. `int("hallo")`
- `FileNotFoundError`: Deze exception wordt opgeworpen wanneer men een bestand tracht in te lezen dat niet kan teruggevonden worden.

Opmerking: code met (opzettelijk) een specifieke fout uitvoeren in een interactieve sessie is een gemakkelijke manier om uit te maken welke *exception* Python zal opwerpen. Er zijn heel wat type *exceptions* die in Python kunnen optreden. Vaak voorkomende *exceptions* worden best gememoriseerd, maar minder voorkomende kan je eventueel zelf uitlokken m.b.v. een voorbeeldje.

9.9.1 De `try-except` constructie

Python voorziet een constructie met een zgn. **`try-except`** die de programmeur toelaat een *exception* op te vangen en aan de programmeur de keuze geeft om deze fout te handelen. Deze constructie heeft twee delen: een **`try`**-deel en één of meerdere **`except`**-delen. Hun functies zijn de volgende:

- Het **`try`**-deel bevat code waarover we (als programmeurs) wensen te waken op mogelijke *runtime* fouten. Met andere woorden, we zijn bezorgd over een deel van onze code die een *exception* kan opwerpen.
- Elk **`except`**-deel wordt geassocieerd met een bepaalde *exception* en een code gedeelte dat zal uitgevoerd worden als deze bepaalde fout optreedt.

In het algemeen de constructie ziet er als volgt uit:

```
try:
    # deel met code om over te waken
except ParticularErrorName1:
    # deel met code om een ParticularErrorName1 te behandelen
except ParticularErrorName2:
    # deel met code om een ParticularErrorName2 te behandelen
...
except ParticularErrorNameX:
    # deel met code om een ParticularErrorNameX te behandelen
```

Hoe weten we nu over welke stukken code we moeten waken? Een eerste mooi voorbeeld: plaatsen waar we van de gebruiker input verwachten. We weten dat fouten zullen optreden indien de gebruiker een foutieve of ongewenste input verstrekt.

In het volgende stukje code wordt aan de gebruiker gevraagd om een geheel getal in te geven. Indien we daadwerkelijk een geheel getal (*integer*) ingeven, dan zal

1. de functie `input()` dit getal als *string* retourneren,
2. de `int`-functie het getal als *string* omzetten naar een *int*, en
3. het `print`-statement zal dit *int* getal op het scherm weergeven.

```
getal_str = input("Geef een geheel getal: ")     # (1)
getal = int(getal_str)                           # (2)
print("Het getal is: ", getal)                   # (3)
print("Programma stopt.")
```

Als we nu als gebruiker geen geheel getal, maar bv. een *string* ingeven, dan zal Python een `ValueError` opwerpen op het moment dat de functie `int()` de ingegeven *string* tracht om te zetten naar een `int`:

```
Geef een geheel getal: drie
Traceback (most recent call last):

  File "<ipython-input-9-e8b5968584e0>", line 2, in <module>
    getal = int(getal_str)

ValueError: invalid literal for int() with base 10: 'drie'
```

Om zo'n onverwachte (en ongewenste) fout te vermijden, moeten we de `try-except` constructie gebruiken. Het gevaar dat hier kan optreden bij het uitvoeren van het sciptje is een verkeerde input door de gebruiker. Het gevoelige stukje code is de instructie `getal = int(getal_str)` (omzetting van de input van de gebruiker naar een `int`). Daarom zullen we de instructie

`getal = int(getal_str)` in het `try` deel plaatsen. Bovendien plaatsen we ook de instructie `print("Het getal is: ", getal)` in het `try`-deel (zie even later waarom). In het `except`-deel zullen we hier een `print`-statement plaatsen die ons waarschuwt dat een fout optrad bij de omzetting. Het geheel wordt samengevat in Fragment 9.8:

Fragment 9.8: `geheel_getal_omzetten.py`

```
1 getal_str = input("Geef een geheel getal: ")
2 try:
3     getal = int(getal_str)
4     print("Het getal is: ", getal)
5 except ValueError:
6     print("Fout bij het converteren naar een int.")
7 print("Programma stopt.")
```

Als we nu een *string* ingeven, dan zal de optredende fout bij de omzetting opgevangen worden door het `except`-deel van de constructie en de waarschuwing weergegeven worden. Erna doet het programma gewoon verder:

```
Geef een geheel getal: 4
Het getal is:  4
Programma stopt.
```

```
Geef een geheel getal: vier
Fout bij het converteren naar een int.
Programma stopt.
```

Opdracht 9.12

Open een nieuw *interactive window* zodat al je voorgaande gedefinieerde variabelen gewist zijn! Plaats nu de instructie `print("Het getal is: ", getal)` na het `except` deel zoals in:

```
getal_str = input("Geef een geheel getal: ")
try:
    getal = int(getal_str)
except ValueError:
    print("Fout bij het converteren naar een int.")
print("Het getal is: ", getal)
print("Programma stopt.")
```

Voer deze code uit met een foutieve input. Welke foutmelding krijg je nu? Leg uit waarom je die fout krijgt. Met welke naam wordt deze fout aangeduid?

9.9.2 Exception voorbeelden

We breiden eerst het vorige uit. Het programma in Fragment 9.8 zal na de `try-except` constructie zijn instructies verder zetten ook indien de gebruiker een foutieve input meegaf. In dit laatste geval zullen we de variabele `getal` niet kunnen gebruiken in een eventueel vervolg van het programma omdat in dit geval `getal` niet gedefinieerd werd. Met een `while`-lus kunnen we ervoor zorgen dat ons programma blijft vragen naar een geheel getal totdat de gebruiker een juiste input meegeeft. Bekijk Fragment 9.9:

Fragment 9.9: `geheel_getal_while.py`

```
while True:
    getal_str = input("Geef een geheel getal: ")
    try:
        getal = int(getal_str)
        break
    except ValueError:
        print("Fout bij het converteren naar een int.")
print("Het getal is: ", getal)
```

Indien een geheel getal ingegeven wordt door de gebruiker, dan zal binnen de `try`-suite dit getal foutloos omgezet worden van een *string* naar een *int* en zal onmiddellijk daarna het `break`-statement uitgevoerd worden waardoor de `while`-lus beëindigd wordt[a]. Indien **geen** geheel getal ingegeven wordt, dan zal binnen de `try`-suite het statement `getal = int(getal_str)` een `ValueError` opwerpen. Deze `ValueError` wordt opgevangen door het `except`-statement en de bijhorende `except`-suite wordt uitgevoerd. Merk hier op dat het `break`-statement **niet** uitgevoerd zal worden!

```
Geef een geheel getal: vijf
Fout bij het converteren naar een int.

Geef een geheel getal: a
Fout bij het converteren naar een int.

Geef een geheel getal: @#('
Fout bij het converteren naar een int.

Geef een geheel getal: 8
Het getal is:  8
```

In Fragment 9.10 wordt gewaakt of een bestand met variabelenaam `bestandsnaam` al dan niet geopend kan worden. Indien bv. het bestand niet bestaat in de huidige directory of we hebben geen rechten om het bestand te lezen (of schrijven), of het bestand is reeds geopend door een ander programma dat niet toelaat dat het bestand een tweede maal geopend wordt, dan zal de functie `open()` een `OSError` opwerpen.

Fragment 9.10: bestand_openen_os_error.py

```
bestandsnaam = input("Geef een bestandsnaam: ")
try:
    bestand = open(bestandsnaam, 'r')
        for line in bestand:
            print(line, end='')
        bestand.close()
except OSError:
    print(f"Het bestand {bestandsnaam:s} werd niet gevonden.")
print("Programma stopt.")
```

We voeren Fragment 9.10 uit en geven eens een bestandsnaam mee van een bestaand bestand en eens een bestandsnaam van een onbestaand bestand.

De output voor een bestaand bestand:

```
Geef een bestandsnaam: temp.txt
Eerste regel
Tweede regel
Derde regel
Vierde regel
Programma stopt.
```

De output voor een onbestaand bestand:

```
Geef een bestandsnaam: blablabla.txt
Het bestand blablabla.txt werd niet gevonden.
Programma stopt.
```

Opdracht 9.13 (deling_exceptions.py)

Schrijf een script dat de gebruiker achtereenvolgens vraagt een teller en een noemer in te geven als reële getallen. Het script moet de deling uitvoeren en vervolgens zowel teller, noemer als resulaat op het scherm weergeven. Je code moet zowel waken over de omzetting van de input *strings* naar *floats*, als over delen door 0. Je zal daarvoor twee **except**-delen nodig hebben. Gebruik de volgende constructie:

```
try:
    # vraag naar twee floats: teller en noemer
    # voer deling uit
except ValueError:
    # indien fout optreedt bij omzetting naar float
except ZeroDivisionError:
    # indien deling door nul
```

Baseer je verder op de volgende voorbeelden om je script te schrijven:

```
Geef de teller: 8.3
Geef de noemer: 2.7
 8.30 gedeeld door  2.70 geeft  3.07
```

```
Geef de teller: 14.52
Geef de noemer: drie
Fout bij het converteren naar een float.
```

```
Geef de teller: 44.12
Geef de noemer: 0
Fout ten gevolge van deling door 0.
```

Opdracht 9.14

Schrijf een script dat de gebruiker achtereenvolgens vraagt naar een bestandsnaam en een regelnummer. Het script moet vervolgens de regel weergeven op het scherm dat overeenkomt met het opgegeven regelnummer. Je code moet zowel waken over het openen van het bestand als over de omzetting van de regelnummer via input als *string* naar *int*. Je zal daarvoor twee `except`-delen nodig hebben. Gebruik de volgende constructie:

```
try:
    # vraag naar bestandsnaam en regelnummer
    # open bestand en zet regelnummer om naar int
    # code om regel op regelnummer te zoeken en te tonen
except OSError:
    # indien fout optreedt bij het openen van bestand
except ValueError:
    # indien fout optreedt bij omzetting naar int
```

Baseer je verder op de volgende voorbeelden om je script te schrijven:

```
Geef bestandsnaam: blinde_liefde.txt
Regelnummer: 5
Regel 5 in bestand blinde_liefde.txt is: Want Annelies is blind...
Programma stopt.
```

```
Geef bestandsnaam: liefde.txt
Het bestand liefde.txt werd niet gevonden.
Programma stopt.
```

```
Geef bestandsnaam: blinde_liefde.txt
Regelnummer: zeven
Regelnummer zeven is niet geldig.
Programma stopt.
```

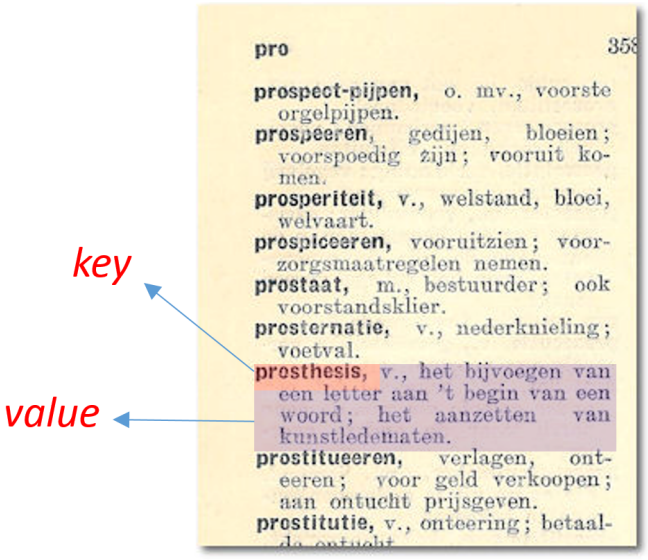
[a] Merk op de het gebruik van **break** vaak als een “slechte” programmeerstijl aanzien wordt. Je kan dit hier vermijden door een functie te schrijven en gebruik te maken van het **return**-statement

10

Dictionaries

10.1 Key-value paren

Het datatype *dictionary* (gegevenstype `dict`) is, net als het gegevenstype `list`, een container-datatype. Dictionaries zijn een voorbeeld van *mapping types*, hetgeen wil zeggen dat men hun elementen niet indexeert met getallen, maar met *keys*. Een *key* kan een object van eender welk immutable gegevenstype zijn (vaak strings, integers of tuples). Elk element van de dictionary heeft een unieke *key*. Een element en zijn key noemt men vaak een *key-value paar*. Een dictionary kan men dus beschouwen als een verzameling van *key-value* paren. Een klassiek voorbeeld van een verzameling key-value paren is een verklarend woordenboek (zie Figuur 10.1). Hierin vormen een woord en de verklaring van dat woord een *key-value* paar.



Figuur 10.1: Uittreksel uit een woordenboek als een voorbeeld van een verzameling *key-value* paren.

10.2 Creëren en indexeren van een `dict`

Als voorbeeld beschouwen we een (hypothetisch) telefoonboek:

Naam	Telefoonnummer
"Paul"	"09/123 45 67"
"John"	"09/987 65 43"
"Richard"	"09/135 79 13"
"George"	"09/999 99 99"

Deze naam-telefoonnummer koppels kunnen als key-value paren (waarbij de naam als key gebruikt wordt en het telefoonnummer als value) toegevoegd worden aan een object van het type **`dict`** zoals getoond wordt in de volgende interactieve sessie:

```
>>> beatles_tel = {"Paul": "09/123 45 67", "John": "09/987 65 43",
   "Richard": "09/135 79 13", "George": "09/999 99 99"}

>>> type(beatles_tel)
dict

>>> beatles_tel
{'George': '09/999 99 99',
 'John': '09/987 65 43',
 'Paul': '09/123 45 67',
 'Richard': '09/135 79 13'}
```

Het voorgaande codefragment illustreert dat een dict-object kan worden gecreëerd door meerdere key-value paren te scheiden door komma's en deze te omgeven door accolades `{}`. De key en zijn value worden daarbij gescheiden door een dubbelpunt (`:`).

Syntax voor het creëren van een `dict`
De expressie

```
D = { key1: value1, key2: value2, key3: value3, ... }
```

creëert een dictionary `D` met key-value paren `key1-value1`, `key2-value2`, ….

Merk op dat **elke key hoogstens één keer** mag voorkomen in eenzelfde dictionary. Komt, bij de creatie van een dictionary, een *key* meerdere keren voor, dan wordt **enkel de laatste** (en de bijhorende *value*) overgehouden.

Uit het voorbeeld blijkt ook dat de **volgorde van de elementen waarin ze werden gedefinieerd niet bewaard** werd. Een dictionary is dan ook **geen sequence type**. **Men kan dus niet spreken van de positie van een key-value paar in de dictionary.**

Na creatie kan men een value opvragen door te **indexeren** met de bijhorende **key**:

```
>>> beatles_tel = {"Paul": "09/123 45 67", "John": "09/987 65 43",
   "Richard": "09/135 79 13", "George": "09/999 99 99"}

>>> beatles_tel["John"]
"09/987 65 43"

>>> type(beatles_tel["John"])
str
```

Indexeren van een `dict`

Beschouw een dict `D` en een key-value paar met key `k` en value `v`. De expressie:

`D[k]`

zal

- de value `v` retourneren indien de key `k` voorkomt in de dict, en
- een `KeyError` opwerpen indien `k` **niet** voorkomt als key.

Enkele voorbeelden:

```
>>> beatles_tel["John"]
"09/987 65 43"

>>> beatles_tel["Neil"]
KeyError: 'Neil'
```

Opdracht 10.1 (`mendelejev_kort.py`)

Voer volgende opdrachten uit:

1. Creëer een dictionary (variabele mendelejev) waarin de key-value paren gevormd worden door het symbool van een chemisch element (type **`str`**) als key en zijn molaire massa (type **`float`**) als value. Beperk je daarbij tot de onderstaande tabel:

Symbool	Molaire massa
`"Na"`	22.99
`"O"`	16.00
`"H"`	1.01

```
>>> mendelejev = {.....................................................}
```

2. Wat zullen de volgende expressies retourneren?

```
>>> 2 * mendelejev["Na"] + mendelejev["O"]

....
>>> mendelejev[1]

....
>>> mendelejev[22.99]

....
```

10.2.1 Lists als values

Tot hiertoe werden strings, integers of floats gebruikt als values. Er is echter **geen beperking op het gegevenstype** dat gebruikt kan worden als value. Dit betekent dat **ook lists als value** gebruikt kunnen worden.

Een voorbeeld:

```
>>> modellen = {"Audi": ["A4", "A5", "A6", "A7", "A8"], "BMW": ["1-serie",
"2-serie", "2-serie Tourer", "3-serie"], "Cupra":["Formentor"]}

>>> modellen
{'Audi': ['A4', 'A5', 'A6', 'A7', 'A8'],
 'BMW': ['1-serie', '2-serie', '2-serie Tourer', '3-serie'],
 'Cupra': ['Formentor']}
```

10.3 Wijzigen van een `dict`

Dictionaries zijn **mutable** en kunnen dus gewijzigd worden nadat ze werden gecreëerd. De belangrijste wijzigingen die men kan aanbrengen zijn:

- **Toevoegen** van een nieuw key-value paar.
- **Wijzigen** van een bestaand key-value paar.
- **Verwijderen** van een key-value paar.

10.3.1 Toevoegen/wijzigen key-value paar in een `dict`

De syntax die toelaat een key-value paar toe te voegen is dezelfde als diegene die gebruikt wordt om een bestaand paar te wijzigen.

Beschouw een dictionary D, een object k (van een immutable datatype) en een object v. Het toekenningsstatement

```
D[k] = v
```

zal de volgende wijziging aanbrengen aan D:

- indien k nog niet voorkomt als *key* in D, dan wordt k toegevoegd als nieuwe key aan D met bijhorende value v,
- indien k reeds voorkomt als key in D, dan wordt de bijhorende value **vervangen** door v.

Het toevoegen en wijzigen van key-value paren wordt geïllustreerd in de onderstaande sessie:

```
>>> draagtijd = {"muis" : 20, "rat" : 22, "kat" : 60}
>>> draagtijd["leeuw"] = 100    # toevoegen van "leeuw" -> 100
>>> draagtijd
{'kat': 60, 'leeuw': 100, 'muis': 20, 'rat': 22}

>>> draagtijd["muis"] = '?'     # wijzigen van 20 -> '?'
{'kat': 60, 'leeuw': 100, 'muis': '?', 'rat': 22}
```

10.3.2 Verwijderen van een key-value paar

Om een key-value paar te **verwijderen** uit een dictionary kan gebruik gemaakt worden van het **del**-statement.

Verwijderen van een key-value paar uit een dict
Beschouw een dictionary D, en een key k van deze dictionary. Het statement

```
del D[k]
```

zal

- de key k en de bijhorende value verwijderen uit de dictionary,
- een KeyError opwerpen indien k **niet** voorkomt als key in D.

Het gebruik van **del** wordt geïllustreerd in de onderstaande sessie:

```
>>> leeftijden = {"Hans": 15, "Griet": 17, "Wolf": 20}

>>> del leeftijden["Griet"]
>>> leeftijden
{'Hans': 15, 'Wolf': 20}

>>> del leeftijden["Heks"]
KeyError: 'Heks'
```

10.3.3 Stapsgewijs aanvullen van een dictionary.

In veel toepassingen worden dictionaries stapsgewijs aangevuld. Vaak start men daarbij van een **lege dictionary**.

Een lege dictionary kan eenvoudig gecreëerd worden met de literal {}:

```
D = {}
```

Vervolgens worden stelselmatig key-value paren toegevoegd aan de dictionary:

```
cijfers = {}          # lege dictionary
cijfers[1] = "een"    # toevoegen key 1 met value "een"
cijfers[2] = "twee"
cijfers[5] = "vijf"
print(cijfers)
```

Met als resultaat:

```
{1: 'een', 2: 'twee', 5: 'vijf'}
```

Opdracht 10.2 (leeftijden_dict.py)

Implementeer een script dat de gebruiker meermaals vraagt om een naam en een leeftijd in te voeren (van elkaar gescheiden door een **spatie**). Vervolgens worden deze naam en leeftijd toegevoegd als een key-value paar aan een dictionary `leeftijden`. De naam is de key (type `str`) en de bijhorende leeftijd de value (type `int`). De gebruiker kan aangeven dat het programma beëindigd moet worden door `"stop"` in te geven. Vul daartoe het onderstaande fragment aan:

```
.........................
naam_leeftijd = input("Geef naam en leeftijd in: ")
while naam_leeftijd != "stop":
   .........................
   .........................
   .........................
print(leeftijden)
```

Voorbeeld input/output:

```
>>> Geef naam en leeftijd in: Hans 15
>>> Geef naam en leeftijd in: Griet 17
>>> Geef naam en leeftijd in: Wolf 20
>>> Geef naam en leeftijd in: stop
{'Hans': 15, 'Griet': 17, 'Wolf': 20}
```

Opdracht 10.3 (codontabel_dict.py)

In Opdracht 7.14 werd de functie `lees_codontabel()` geïmplementeerd die het bestand `codontabel.csv` inleest en retourneert als een lijst van lijsten zoals hieronder geïllustreerd wordt:

```
>>> lees_codontabel("codontabel.csv")
[['Isoleucine', 'I', 'ATT', 'ATC', 'ATA'],
 ['Leucine','L','CTT','CTC','CTA','CTG','TTA','TTG'],
  ... ]
```

1. In dit voorbeeld is `I` de *single letter code* (SLC) van de codons `"ATT"`, `"ATC"` en `"ATA"`. Implementeer de functie `maak_codondict()` die deze tabel als input aanvaardt en een dictionary retourneert waarin elk codon dat voorkomt in de tabel een *key* is en de bijhorende *value* de SLC die bij dat codon hoort. Het onderstaande codefragment illustreert het gebruik van deze functie.

   ```
   >>> codonTabel = lees_codontabel("codontabel.csv")
   >>> codonDict = maak_codondict(codonTabel)
   >>> print(codonDict)
   {'ATT': 'I', 'ATC': 'I', 'ATA': 'I', 'CTT': 'L', 'CTC': 'L', ...}
   ```

2. **Uitbreiding**: gebruik de dictionary die je bekomt om een DNA-string om te zetten in een string van SLCs door de string **codon per coden** te vertalen. Het volgende codefragment kan je daartoe aanvullen:

   ```
   dnastring = "ATCATCCTCCTC"
   slcstring = ""
   for ............ in ................... :
      ..............................
      ..............................
      ..............................
   ```

   Het resultaat is:

   ```
   >>> print(slcstring)
   'IILL'    # "ATC" --> "I", "ATC" --> "I", "CTC" --> "L", "CTC" --> "L"
   ```

10.4 De `in` operator

De `in` operator kan gebruikt worden om na te gaan of een bepaalde waarde aanwezig is als **key** in een dictionary.

De `in` operator.
Beschouw een dictionary `D` en een object `k`. De logische expressie

`k in D`

retourneert

- `True` als `k` een key is in `D`,
- `False` als `k` niet voorkomt als key in `D`.

Het gebruik van deze operator wordt geïllustreerd in het onderstaande codefragment:

```
>>> leeftijden = {"Hans": 15, "Griet": 17, "Wolf": 20}
>>> "Griet" in leeftijden    # KEY
True
>>> "Jan" in leeftijden      # KEY
False
>>> 20 in leeftijden         # VALUE 20
False
```

Merk op dat de operator `in` **niet** kan gebruikt worden om na te gaan of een bepaalde **value** aanwezig is in een dictionary.

Opdracht 10.4 (`gebruik_in_dict.py`)
In het onderstaande codefragment wordt de informatie die in de lijst `woorden` aanwezig is, gebruikt om een dictionary op te bouwen.

```
woorden = ["rood", "groen", "rood", "blauw", "blauw", "blauw", "geel"]
mijn_dict = {}
for woord in woorden:
   if woord in mijn_dict:
      mijn_dict[woord] = mijn_dict[woord] + 1
   else:
      mijn_dict[woord] = 1
```

1. Interpreteer bovenstaande codefragment en omschrijf in één zin wat de inhoud zal zijn van `mijn_dict` nadat dit fragment werd uitgevoerd. Doe dit eerst zonder de code uit te voeren en bevestig nadien indien nodig je antwoord door deze code uit te voeren.

2. **Uitbreiding**: het onderstaande codefragment vertoont enkele gelijkenissen met het voorgaande fragment. Interpreteer ook dit codefragment en omschrijf in één zin wat de inhoud van `mijn_dict` zal zijn nadat dit fragment werd uitgevoerd (doe dit eerst zonder de broncode in te voeren in Python).

```
woorden = ["rood", "groen", "rood", "blauw", "blauw", "blauw", "geel"]
mijn_dict = {}
for i in range(len(woorden)):
   if woorden[i] in mijn_dict:
      mijn_dict[woorden[i]].append(i)
   else:
      mijn_dict[woorden[i]] = [i]
```

Opmerking: in toepassingen waarbij de values meerdere waarden kunnen bevatten (zoals lijsten, zie Sectie 10.2.1), volstaat de werkwijze beschreven in Opdracht 10.2 om een dictionary stapsgewijs op te bouwen niet.

Veronderstel bijvoorbeeld dat we de dictionary `modellen` uit Sectie 10.2.1 stapsgewijs willen opbouwen door een merk en een model (bv. `Audi A4`) in te lezen.

We willen, vertrekkende van `modellen = {}`, het volgende als resultaat:

```
>>> modellen
{'Audi': ['A4', 'A5', 'A6', 'A7', 'A8'],
 'BMW': ['1-serie', '2-serie', '2-serie Tourer', '3-serie'],
 'Cupra': ['Formentor']}
```

Proberen we dit te realiseren met:

```
modellen = {}
merk_model = input("Geef een merk en model: ")
while merk_model != "stop":
    merk, model = merk_model.split(" ")
    modellen[merk] = model
    merk_model = input("Geef een merk en model: ")

print(modellen)
```

Dan krijgen we bv. als resultaat:

```
Geef een merk en model: Audi A4
Geef een merk en model: Audi A5
Geef een merk en model: Audi A6
Geef een merk en model: BMW 1-serie
Geef een merk en model: BMW 2-serie
Geef een merk en model: Cupra Formentor
Geef een merk en model: stop
{'Audi': 'A6', 'BMW': '2-serie', 'Cupra': 'Formentor'}
```

Vaststellingen:

- De values zitten **niet** in een list. Dit is logisch daar we nergens een lijst hebben gedefinieerd.
- Per merk werd slechts één model opgenomen. Dit is het gevolg van de instructie `modellen[merk] = model`. Immers, de syntax die toelaat een key-value paar toe te voegen is dezelfde als diegene die gebruikt wordt om een bestaand paar te wijzigen (Sectie 10.3.1).

De oplossing bestaat er in om eerst te controleren of het opgegeven merk (de *key*) voorkomt (met de operator `in`), en als:

1. de key voorkomt het model (de *value*) te **appenderen** aan de bestaande lijst met modellen met

```
modellen[merk].append(model)
```

2. de key niet voorkomt het model als **een lijst met één element** toe te voegen als value met

```
modellen[merk] = [model]       # bemerk de [ en ]
```

Met deze aanpassingen bekomen we de volgende code:

```
modellen = {}
merk_model = input("Geef een merk en model: ")
while merk_model != "stop":
    merk, model = merk_model.split(" ")
    if merk in modellen:  # als merk WEL voorkomt
        modellen[merk].append(model)
    else:                 # als merk NIET voorkomt
        modellen[merk] = [model]
    merk_model = input("Geef een merk en model: ")
print(modellen)
```

Met het gewenste resultaat:

```
Geef een merk en model: Audi A4
Geef een merk en model: Audi A5
Geef een merk en model: Audi A6
Geef een merk en model: BMW 1-serie
Geef een merk en model: BMW 2-serie
Geef een merk en model: Cupra Formentor
Geef een merk en model: stop
{'Audi': ['A4', 'A5', 'A6'], 'BMW': ['1-serie', '2-serie'],
'Cupra': ['Formentor']}
```

Opdracht 10.5 (`continenten.py`)
Het bestand `country-capitals.csv` werd reeds beschreven in Opdracht 7.19. Van elk land ter wereld bevat dit bestand de naam, hoofdstad, latitute, longitude, afkorting landnaam en tenslotte de naam van het continent waarop dit land gelegen is. Deze informatie wordt, per land, gescheiden door komma's. Voor België is dit bijvoorbeeld:

```
Belgium,Brussels,50.833333333333336,4.333333,BE,Europe
```

Gebruik de functie `listRead()` in de module `infoFun` om dit bestand in te lezen. Creëer o.b.v. de resulterende lijst een dictionary `continenten_dict` die als keys de namen van de continenten bevat die voorkomen in het bestand (zonder deze expliciet zelf in te typen) en als bijhorende values voor elk continent een lijst van landen bevat die op dit continent liggen. Een deel van deze dictionary wordt hieronder getoond:

```
>>> continenten_dict
{'South America': ['Argentina','Bolivia','Brazil', 'Chile', ...],
'Asia': ['Palestine','Afghanistan','Bahrain','Bangladesh', ...],
 ...  }
```

10.5 Dictionaries zijn iterable

Men kan een dictionary converteren naar een iterator met de functie `iter()`. Het resultaat is een iterator die itereert over de keys die voorkomen in de dictionary (en dus **niet** over de values). Het gebruik van deze iteratoren wordt geïllustreerd in het onderstaande codefragment. **Opmerking**: over de volgorde waarin deze keys overlopen worden hebben we als programmeur geen controle!

```
>>> leeftijden = {"Hans": 15, "Griet": 17, "Wolf": 20}
>>> leeftijden_iter = iter(leeftijden)

>>> next(leeftijden_iter)
'Hans'

>>> next(leeftijden_iter)
'Griet'
```

Omdat dictionaries iterable zijn, kan men ze gebruiken in een `for`-statement. Merk op dat ook hier wordt geïtereerd over de keys van de dictionaries. Wenst men de values te overlopen, dan kan dit enkel via indexering. Dit wordt geïllustreerd in het onderstaande codefragment.

```
leeftijden = {"Hans": 15, "Griet": 17, "Wolf": 20}
for name in leeftijden:
   print("Naam:", name, "Leeftijd:", leeftijden[name])
```

De output van dit fragment is:

```
Naam: Hans Leeftijd: 15
Naam: Griet Leeftijd: 17
Naam: Wolf Leeftijd: 20
```

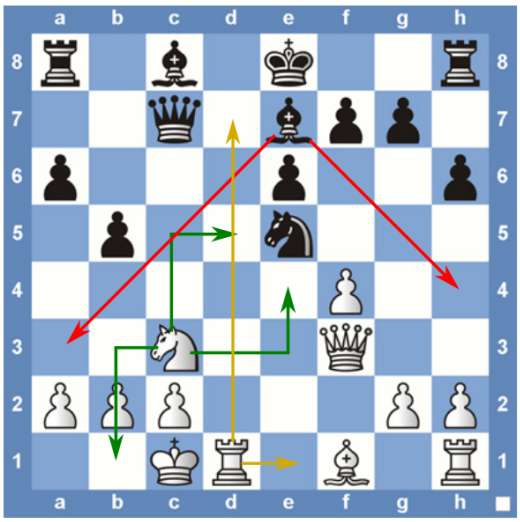
10.6 Tuples als key

In de voorgaande voorbeelden werden vaakt strings gebruikt als keys. Men kan echter objecten van eender welk immutable gegevenstype gebruiken als key. In deze sectie wordt bijzondere aandacht besteed aan keys van het type `tuple`. **Ter herhaling**: een tuple is een **immutable** sequentie (zie onderstaand voorbeeld).

```
>>> vb = (1, "a")
>>> type(vb)
tuple

>>> vb
(1, 'a')

>>> vb[1]
'a'
```

Beschouw ter illustratie een schaakbord. De positie van de schaakstukken kan worden bewaard in een dictionary. De keys in deze dictionary zijn de tuples bestaande uit de rij (cijfers 1–8) en de kolom (karakters `a`–`h`) waarop een schaakstuk zich kan bevinden.

```
1 schaakbord = {(7, "f") : "pion",
2   (5, "e") : "paard",
3   (8, "a") : "toren", ...}
```



Op basis van deze dictionary kan vervolgens bekeken worden welk schaakstuk er op een bepaalde plaats staat.

Opdracht 10.6 (`schaken.py`)

Beschouw het schaakbord uit de voorgaande figuur. De huidige positie van de zwarte schaakstukken en hun type zijn beschikbaar in het bestand `schaakbord_zwart.txt`.

1. Implementeer de functie `lees_schaakbord()` die een bestandsnaam (zoals `schaakbord_zwart.txt`) als input aanvaardt. De posities van de schaakstukken worden ingelezen en bewaard als key-value paren in de dictionary (positie is key en type schaakstuk is value):

```
>>> schaakbord = lees_schaakbord("schaakbord_zwart.txt")
>>> schaakbord
{(7, "f"): "pion", (5, "e"): "paard", (8, "a"): "toren", ...}
```

2. Implementeer de functie `verplaats_schaakstuk()` die toelaat om de positie van een schaakstuk in de dictionary aan te passen. Deze functie aanvaard een schaakbord-dictionary als input, samen met de huidige en nieuwe coördinaten van het te verplaatsen schaakstuk:

```
1 def verplaats_schaakstuk(schaakbord, rij, kol, rij_nieuw, kol_nieuw):
2     ...
```

   Deze functie zal:

   - **controleren** of de key (rij, kol) voorkomt in het schaakbord,
   - indien **ja**: het schaakstuk **verplaatsen** (de bestaande positie (key) verwijderen en een nieuwe toevoegen),
   - indien **nee**: **melden** dat op die plaats geen schaakstuk staat.

```
>>> schaakbord = lees_schaakbord("schaakbord_zwart.txt")
>>> schaakbord
{(7, "f"): "pion", (5, "e"): "paard", (8, "a"): "toren", ...}

>>> verplaats_schaakstuk(schaakbord, 5, "e", 4, "g")
>>> schaakbord
{(7, "f"): "pion", (4, "g"): "paard", (8, "a"): "toren", ...}
```

3. **Uitbreiding**: zoals de pijlen op de figuur aangeven, is de verplaatsing van schaakstukken onderhevig aan bepaalde regels. Pas de functie `lees_schaakbord()` aan zodat enkel verplaatsingen die toegelaten zijn kunnen worden uitgevoerd. Ongeldige verplaatsingen leiden tot een gepaste foutboodschap. Doe dit voor
   (a) de toren,
   (b) het paard
   (c) de loper.

```
>>> schaakbord = lees_schaakbord("schaakbord_zwart.txt")
>>> schaakbord
{(7, "f"): "pion", (5, "e"): "paard", (8, "a"): "toren", ...}
```

```
>>> verplaats_schaakstuk(schaakbord, 8, "a", 7, "b")
Ongeldige zet! Toren enkel horizontaal of vertikaal verplaatsen !!

>>> schaakbord
{(7, "f"): "pion", (5, "e"): "paard", (8, "a"): "toren", ...}
```

10.7 De functie `dict()`

De functie **`dict()`** aanvaardt een iterable als input en converteert deze naar een object van het type **`dict`**. Merk op dat de elementen die deze iterable opwerpt op hun beurt *tuple unpacking* moeten toelaten zodat ze kunnen ontbonden worden in twee elementen. Een typerend voorbeeld is een geneste lijst.

```
>>> leeftijden_lst = [["Hans", 15], ["Griet", 17], ["Wolf", 20]]
>>> leeftijden_dict = dict(leeftijden_lst)
>>> leeftijden_dict
{'Griet': 17, 'Hans': 15, 'Wolf': 20}
```

De functie **`dict()`** werd hier voor de volledigheid besproken, maar wordt verder in deze cursus **niet** gebruikt.

10.8 Dictionary methoden

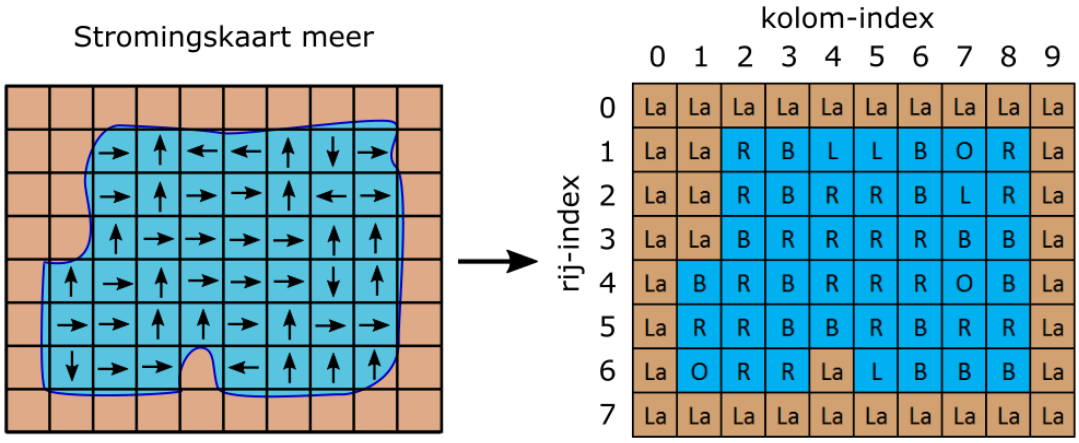
Net als de **`str`** en **`list`** klassen, voorziet ook de **`dict`** klasse een aantal methoden die de programmeur toelaten om dictionaries te manipuleren. Merk op dat (binnen deze cursus) nagenoeg alle functionaliteiten die voorzien worden door deze methoden vrij eenvoudig kunnen worden geïmplementeerd door gebruik te maken van *key*-indexering. De volgende tabel werd dan ook in de eerste plaats ter volledigheid toegevoegd, maar wordt binnen deze cursus niet gebruikt.

dict-methode	**Beschrijving**
`clear()`	Removes all the elements from the dictionary.
`copy()`	Returns a copy of the dictionary.
`fromkeys()`	Returns a dictionary with the specified keys and values.
`get()`	Returns the value of the specified key.
`items()`	Returns a list containing a tuple for each key value pair.
`keys()`	Returns a list containing the dictionary's keys.
`pop()`	Removes the element with the specified key.
`popitem()`	Removes the last inserted key-value pair.
`setdefault()`	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value.
`update()`	Updates the dictionary with the specified key-value pairs.
`values()`	Returns a list of all the values in the dictionary.

10.9 Gemengde opdrachten

Opdracht 10.7 (`stroming.py`)

Het is algemeen gekend dat in oceanen een aantal belangrijke zeestromingen voorkomen. Ook in grote meren komen dergelijke stromingen voor. De kaart hieronder toont een meer waarbij de pijlen de stromingsrichting voorstellen. Deze kaart kan voorgesteld worden als een matrix, waarin elke cel een pixel voorstelt en landpixels de waarde `'La'` hebben. De stromingsrichtingen (rechts, links, boven en onder) worden voorgesteld door R (→), L (←), B (↑), O (↓).



Merk op dat elk element van deze kaart-matrix wordt gekoppeld aan een rij- en kolomindex die ook de ruimtelijke positie van de pixels aanduiden. De matrixvoorstelling van de kaart in het voorbeeld is beschikbaar in het tekstbestand `kaart1.txt`. Dit bestand volgt het csv-principe waarin elke rij op een afzonderlijke regel staat en de waarden gescheiden worden door puntkomma's.

1. Schrijf een functie `lees_kaart()` die als input de bestandsnaam (*string*) van een kaartbestand aanvaardt (zoals bv. `'kaart1.txt'`). Deze functie leest de inhoud van het bestand in en retourneert de ingelezen matrix als een lijst van lijsten met daarin de ingelezen karakters.

```
>>> kaart_matrix = lees_kaart("kaart1.txt")
>>> print(kaart_matrix)
[['La', 'La', 'La', ...,'La', 'La', 'La', 'La'],
 ['La', 'La', 'R', ..., 'B', 'O', 'R', 'La'],
 ['La', 'La', 'R', ...,'L', 'R', 'La'],
 ['La', 'La', 'B', ..., 'B', 'B', 'La'],
  ...
  ]
>>> print(kaart_matrix[3][5])  # rij-index 3 en kolom-index 5
R
```

2. Implementeer een functie `kaart_naar_dictionary()` die als input een kaart aanvaardt (de lijst van lijsten zoals `kaart_matrix` hiervoor). Deze functie retourneert een dictionary waarin elk element in de kaart-matrix wordt voorgesteld door een key-value paar:

- De key bevat de rij- en kolomindices van het beschouwde element (als *tuple* van *integers* of als *string*, dit mag je zelf kiezen)
- De value hangt af van de waarde van het beschouwde element:
  - Stelt het element een landpixel voor, dan is de value de *string* `'Land'`.
  - Stelt het element een waterpixel voor, dan is de value het koppel gevormd door de rij- en kolomindices van het buur-element in het verlengde van de stroomrichting (als *tuple* van *integers* of als *string*).

Deze dictionary bevat onder andere de volgende key-value paren:

```
(3, 1): 'Land'   # element met rij-index 3 en kolom-index 1
                 # is landpixel

(5, 1): (5, 2)   # element met rij-index 5 en kolom-index 1
                 # heeft stroomrichting rechts (rij-indices gelijk)
                 # en de rechterbuur heeft indices (5, 2)

(4, 3): (3, 3)   # element met rij-index 4 en kolom-index 3
                 # heeft stroomrichting boven (kolom-indices gelijk)
                 # en de bovenbuur heeft indices (3, 3)
```

```
>>> kaart_dict = kaart_naar_dictionary(kaart_matrix)
>>> print(kaart_dict)
{(3,1): 'Land', (5,1): (5,2), (4,3): (3,3),
(4, 7): (5,7), (6,4): 'Land', ...}
```

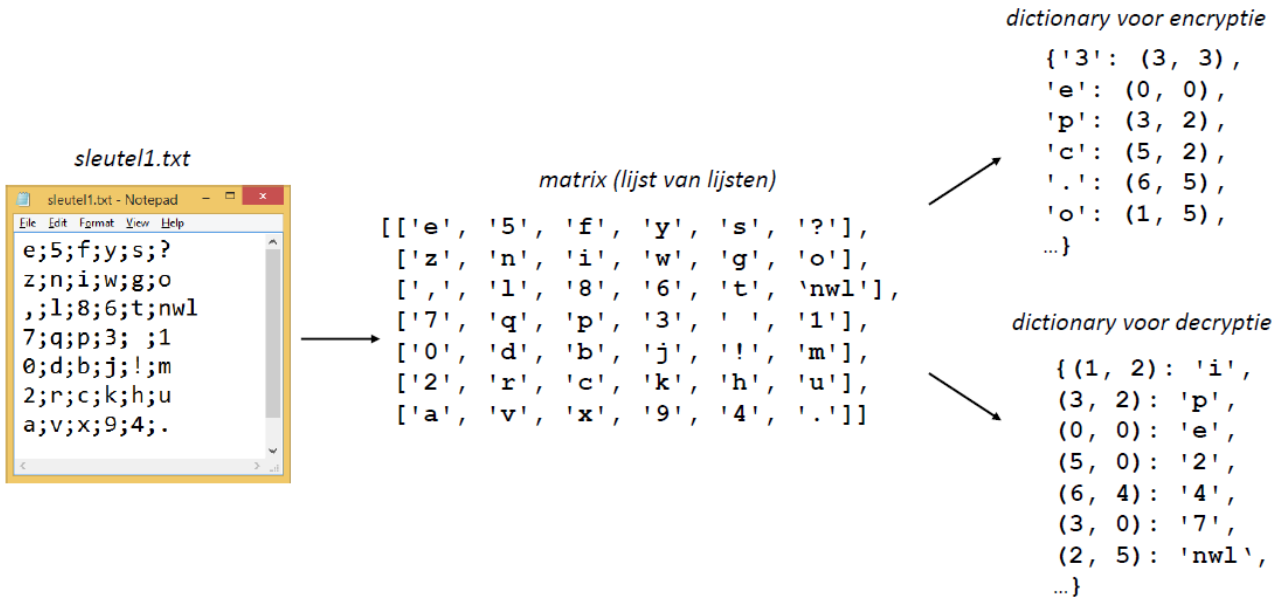
3. (*) Implementeer een functie `aanspoelen()`. Deze functie aanvaardt **drie** argumenten:
   (1) De rij-index (`r`) van een element in de kaart-matrix.
   (2) De kolom-index (`k`) van een element in de kaart-matrix.
   (3) Een kaart-dictionary die de vorm heeft zoals beschreven in de opgave hiervoor.

   Deze functie retourneert de rij- en kolomindex van de landpixel waarop een voorwerp zal *aanspoelen* wanneer het in het water valt op positie `(r, k)` en de stroomrichting volgt tot voor het eerst een landpixel bereikt wordt.

```
>>> locatie = aanspoelen(4, 6, kaart_dict)
>>> print(locatie)
(5, 9)   # dit zijn de rij- en kolomindex van de
         # landpixel waar het voorwerp zal aanspoelen
>>> locatie = aanspoelen(5, 1, kaart_dict)
>>> print(locatie)
(0, 6)
```

Opdracht 10.8 (`versleuteling.py`)

Encryptie is het coderen of versleutelen van gegevens (vaak tekst) o.b.v. een bepaald algoritme. Deze versleutelde gegevens kunnen nadien weer gedecrypteerd (ontcijferd of gedecodeerd) worden zodat men de originele informatie weer terugkrijgt. Dit proces wordt decryptie genoemd. Om de encryptie/decryptie te kunnen uitvoeren heeft men een sleutel nodig. De sleutel die we hier gebruiken is een matrix van 42 karakters (alfabet, cijfers, leestekens) met 7 rijen en 6 kolommen. Elk karakter wordt door deze matrix afgebeeld op een koppel gevormd door zijn rij- en kolomindex, zoals geïllustreerd in de onderstaande figuur. In dit voorbeeld wordt bijvoorbeeld het karakter `p` voorgesteld door het koppel `(3, 2)` omdat de rij-index van `p` in de matrix 3 is en de kolomindex 2. Bovendien is `nwl` het symbool voor een nieuwelijnkarakter.



Deze afbeelding kan men gebruiken om een tekst*string* te encrypteren als een *lijst* van koppels (*tuples*), of een lijst van koppels te decrypteren naar een tekst*string*.

1. Schrijf een functie `lees_sleutel()` die als input de bestandsnaam (*string*) van een sleutelbestand aanvaardt (zoals bv. `sleutel1.txt`). Deze functie leest de inhoud van het bestand in en retourneert de ingelezen matrix als een lijst van lijsten met daarin de ingelezen karakters:

```
>>> sleutel_lst = lees_sleutel("sleutel1.txt")
>>> print(sleutel_lst)
[['e', '5', 'f', 'y', 's', '?'],
 ['z', 'n', 'i', 'w', 'g', 'o'],
 [',', 'l', '8', '6', 't', 'nwl'],
  ...
 ]
```

2. Schrijf een functie `sleutel_naar_dictionary()` die als input een sleutel aanvaardt (de lijst van lijsten zoals `sleutel_lst` hiervoor). Deze functie retourneert een dictionary met als *keys* de rij/kolom *tuple*s en als *values* de bijhorende karakters:

```
>>> sleutel_lst = lees_sleutel("sleutel1.txt")
>>> sleutel_dict = sleutel_naar_dictionary(sleutel_lst)
>>> print(sleutel_dict)
{(1, 2): 'i', (3, 2): 'p', (0, 0): 'e', (5, 0): '2',
 (6, 4): '4', (3, 0): '7', (2, 5): 'nwl', ...}
```

3. Schrijf een script dat de versleutelde tekst in het bestand `geheim.txt` inleest, decrypteert en de gedecrypteerde tekst naar het scherm print als een mooi doorlopende tekst (uiteraard wordt 'nwl' daarbij een nieuwelijnkarakter). **Tip**: kijk eerst eens naar dit bestand in Wordpad.

4. (*) In de map `sleutels` zitten 1000 bestanden die elk een andere sleutel bevatten: `sleutel1.txt`, …, `sleutel1000.txt`. Bovendien zit in deze map een bestand `grootgeheim.txt` met daarin een versleutelde Nederlandstalige tekst. Eén van deze 1000 sleutels werd gebruikt voor de versleuteling, maar je weet niet dewelke.

   Schrijf een script dat (automatisch) bepaalt:

   (i) welke sleutel de correcte is, en

   (ii) de gedecrypteerde tekst naar het scherm print.

   Maak hiervoor gebruik van het **volgende principe**: in Nederlandstalige teksten komen de lidwoorden 'de', 'het' en 'een' heel vaak voor. De correcte sleutel zal dus diegene zijn waarvoor deze lidwoorden na decryptie het vaakst voorkomen.

Opdracht 10.9 (`isotopen.py`)

Het **atoomnummer** van een atoom geeft het aantal protonen in de kern weer en bepaalt ook zijn chemisch symbool (en omgekeerd). De atoomsoort kan je dus aangeven door het symbool (bv. *Cu*) of door zijn atoomnummer (bv. 29).

Vul het bestand `isotopen.py` aan

1. Omdat er een 1-op-1 verband bestaat tussen het symbool en zijn atoomnummer, kan je deze link eenvoudig bewaren in een dictionary. Het bestand `atoomnummers.csv` bevat alle chemische symbolen met hun atoomnummers. Schrijf een script waarin je dit bestand inleest en de informatie gebruikt om twee dictionaries te maken:

   - De dictionary `symNaarAtoomNr` met als keys de symbolen (type *str*) en als value het bijhorende atoomnummer (type *int*). Een deel van deze dictionary wordt hieronder getoond:

     ```
     >>> symNaarAtoomNr
     {'H': 1, 'He': 2, 'Li': 3, 'Be': 4, ..., 'Uuo': 118}
     ```

   - De dictionary `atoomNrNaarSym` met als keys de atoomnummer (type *int*) en als value het bijhorende symbool (type *str*). Een deel van deze dictionary wordt hieronder getoond:

     ```
     >>> atoomNrNaarSym
     {1: 'H', 2: 'He', 3: 'Li', 4: 'Be', ..., 118: 'Uuo'}
     ```

2. Door het aantal protonen (of m.a.w. het atoomnummer) te verhogen of te verlagen wijzigt dus ook het chemisch symbool van een atoom. Schrijf een script waarin je de gebruiker vraagt om een chemisch symbool in te geven. Vervolgens vraag je hoeveel protonen moeten worden toegevoegd (aan te geven met +) of verwijderd (aan te geven met -). Het resulterende symbool moet op het scherm getoond worden. Maak o.a. gebruik van de dictionaries die je hiervoor aanmaakte.

   Merk op: Indien een ongeldig atoomnummer bekomen wordt of een onbekend symbool ingegeven wordt verschijnt een gepaste foutmelding:

```
Geef symbool in: Cu                          # Cu heeft 29 protonen
Geef verhoging/verlaging van protonen: -3    # 3 protonen verwijderen
Resulterende atoom is Fe                     # Fe heeft 26 protonen
```

```
Geef symbool in: O                           # O heeft 8 protonen
Geef verhoging/verlaging van protonen: +2    # 2 protonen toevoegen
Resulterende atoom is Ne                     # Ne heeft 10 protonen
```

```
Geef een symbool in: He                      # He heeft 2 protonen
Geef verhoging/verlaging van protonen: -9    # 9 protonen verwijderen
Deze wijziging van protonen is niet toegelaten! # kan uiteraard niet
```

```
Geef symbool in: Ha                          # Ha is geen symbool
Het symbool dat je ingaf is niet geldig!
```

   Deze vragen moeten niet herhaald te worden, je hoeft geen gebruik te maken van een while-lus met stop-criterium.

De functie `leesIsotopen()` (reeds beschikbaar in `isotopen.py`) kan gebruikt worden om het bestand `isotopen.data` in te lezen. Deze functie retourneert een dictionary waarin de keys isotoopstrings zijn en de bijhorende value een lijst met daarin de toegelaten types verval bij dit isotoop.

```
>>> isotopen_dct = leesIsotopen("isotopen.data")
>>> print(isotopen_dct)
{'H-3': ['b-'],
 'Be-7': ['b+'],
 ...
 'Os-165': ['a', 'b+'],
 ...
}
```

3. (*) Het **massagetal** is de som van het aantal protonen en het aantal neutronen. Bepaalde atoomsoorten hebben meerdere isotopen, die verschillen in het aantal neutronen en dus ook een verschillend massagetal hebben. We noteren een atoom en zijn massagetal als een *isotoopstring* door zijn symbool, gevolgd door een koppelteken en het massagetal. Bv. fluor met massagetal 17 wordt: `"F-17"` en fluor met massagetal 18 wordt `"F-18"`. Het atoomnummer is voor beide uiteraard hetzelfde (in dit geval 9).

   Radioactief verval is een proces waarbij het aantal protonen en/of neutronen van een atoom wijzigt. Drie vormen van radioactief verval worden hieronder getoond:

Type verval	Afkorting	Invloed op atoomnr.	Invloed op massagetal	Voorbeeld
β^--verval	`b-`	+1	geen invloed	`H-3` $\rightarrow$ `He-3`
β^+-verval	`b+`	-1	geen invloed	`Be-7` $\rightarrow$ `Li-7`
α-verval	`a`	-2	-4	`Os-165` $\rightarrow$ `W-161`

   Schrijf een Python script dat o.b.v. `isotopen_dct` een nieuwe dictionary `verval_dct` genereert met daarin dezelfde keys als die in `isotopen_dct`, met als bijhorende value een lijst van *isotoopstrings* die bekomen worden na de toegelaten types verval. Een deel van deze dictionary wordt hieronder getoond:

```
>>> print(verval_dct)
{'H-3': ['He-3'],                # want b- verval voor H-3  -> He-3
 'Be-7': ['Li-7'],               # want b+ verval voor Be-7 -> Li-7
 ...
 'Os-165': ['W-161', 'Re-165'], # want a -> W-161 en b+ -> 'Re-165'
 ...
}
```

11

Datavisualisatie met Matplotlib

11.1 Inleiding

In de wetenschappelijke wereld wordt Python vaak gebruikt als een tool voor de analyse van data. De mogelijkheden die Python biedt om data te visualiseren (of plotten) zijn daarbij heel belangrijk. Er zijn verschillende bibliotheken of modules beschikbaar die toelaten om 2D plots zoals scatter-plots (spreidingsdiagrammen), bar-plots, pie-charts, ... te genereren[1]. Van deze bibliotheken is Matplotlib (`https://matplotlib.org/`) een van de meest populaire en best gedocumenteerde[2]. In dit hoofdstuk lichten we daarom het (basis)gebruik van deze module toe.

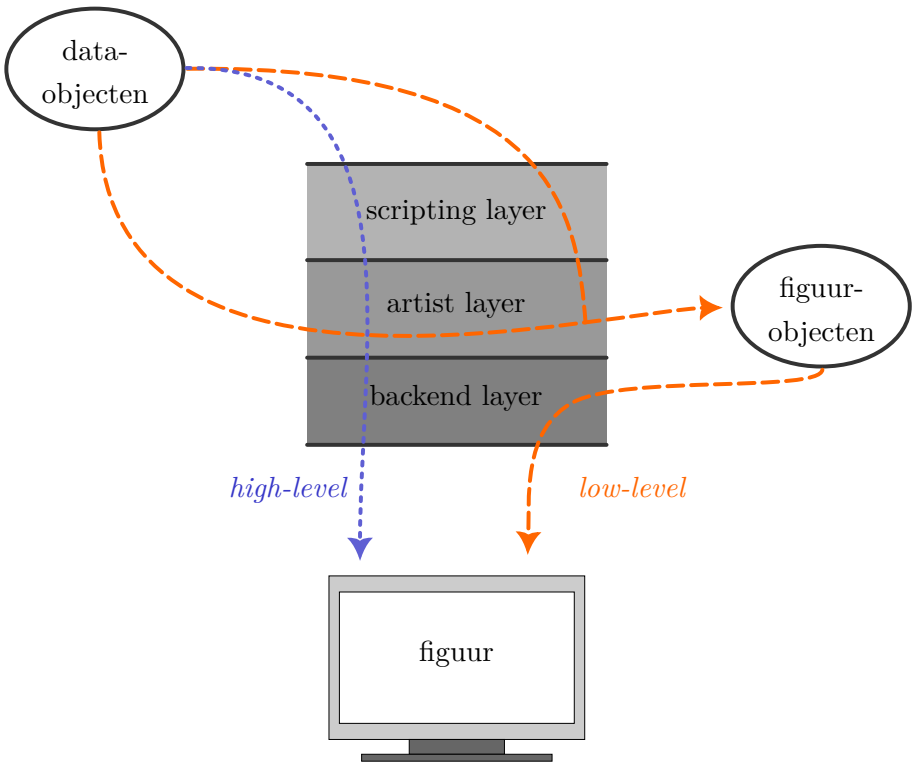
11.2 Matplotlib architectuur

De Matplotlib bibliotheek bestaat uit tientallen klassen en submodules. Om doeltreffend met Matplotlib aan de slag te kunnen, is het van belang een globaal inzicht te hebben in de achterliggende architectuur. Deze kan worden voorgesteld door drie lagen, waarvan een hogere laag de lagere aanstuurt (Figuur 11.1). Hierbij zullen we meestal gebruik maken van de *scripting layer*, die gebundeld is in de submodule `pyplot`. Deze submodule bevat tal van functies om snel figuren te maken van data-objecten. Deze functies zullen figuur-objecten aanmaken uit de onderliggende *artist layer*. Deze *artist layer* kan ook direct aangestuurd worden om een figuur samen te stellen. Zo kunnen we twee manieren onderscheiden om figuren te genereren en manipuleren met Matplotlib.

1. De **`pyplot` interface** is de *high-level* en meest eenvoudige manier van werken. Deze zullen we introduceren in de volgende sectie (Sectie 11.3).
2. Via de *low-level* **object-georiënteerde interface** kunnen we elk afzonderlijk element van de figuur aanpassen. Deze manier is veel flexibeler, maar vergt meer code. Dit zullen we hier niet behandelen.

[1] https://wiki.python.org/moin/NumericAndScientific/Plotting

[2] De 'officiële' documentatie van Matplotlib kan soms vrij uitdagend zijn voor beginnende programmeurs. Een aantal geschikte zoektermen ingeven in je favoriete zoekmachine is vaak een goed alternatief.



Figuur 11.1: Schematische voorstelling van de Matplotlib architectuur en de twee manieren om figuren te genereren en manipuleren. De blauwe stippenlijn en oranje streepjeslijnen stellen respectievelijk de *high-level* en *low-level* manier van werken voor om een data-object om te zetten naar een figuur.

Ten slotte vormt de *backend layer* de onderste laag, die precies bepaalt hoe een figuur wordt voorgesteld op het scherm. De naam slaat op het gegeven dat de *rendering*, het omzetten van de figuurobjecten naar pixels op het scherm/in een bestand, achter de schermen plaatsvindt. Hoe dit precies gebeurt is vaak minder belangrijk, maar beperkt inzicht in de *backend layer* is wel van belang wanneer we bijvoorbeeld willen

- dat de figuur in een apart (interactief) venster verschijnt,
- willen controleren of en hoe lang een figuur getoond wordt op het scherm.

Beide zullen belangrijk zijn wanneer we een animatie willen maken. Om na te gaan welke backend gebruikt wordt, kan je volgende instructies ingeven:

```
>>> import matplotlib
>>> matplotlib.rcParams["backend"]
```

Wanneer je een specifieke backend wil gebruiken, moet je dit aangeven **vóór je de eerste figuur maakt**. Hiervoor kan je de functie `matplotlib.use()` gebruiken die je dan bovenaan het script en/of in het begin van een interactieve sessie oproept.

We onderscheiden twee types backend: de **statische** back-ends en **interactieve** back-ends. Statische back-ends tonen de figuur als een foto in een interactieve sessie of bewaren deze in een (png-) bestand. Interactieve back-ends genereren figuren in een afzonderlijk figuurvenster en maken interactie met de figuur mogelijk.

De `QtAgg` backend een populaire backend voor interactieve figuren. Je activeert deze als volgt:

```
>>> matplotlib.use("qtagg")
```

11.3 Pyplot interface

Om gebruik te kunnen maken van de `pyplot` interface zullen we in de praktijk steeds volgende import statements opnemen (bovenaan) in ons script[3]. De toelichtingen die volgen gaan tevens uit van het gebruik van een *interactieve* Python-sessie:

```
1 import matplotlib
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 matplotlib.use("qtagg")
6 plt.ion()
```

Merk daarbij het volgende op:

- Regel 5 geeft aan dat een interactieve back-end gebruikt wordt.
- Regel 6 configureert matplotlib voor optimale interactie (`ion` staat voor *interactive on*). Dit zorgt er onder andere voor dat actieve figuurvensters automatisch gerenderd worden of geüpdatet worden na elke plot-instructie. Als deze regel niet wordt meegegeven, kan de instructie `plt.show()` gebruikt worden. Deze functie zal alle actieve figuren renderen/updaten.

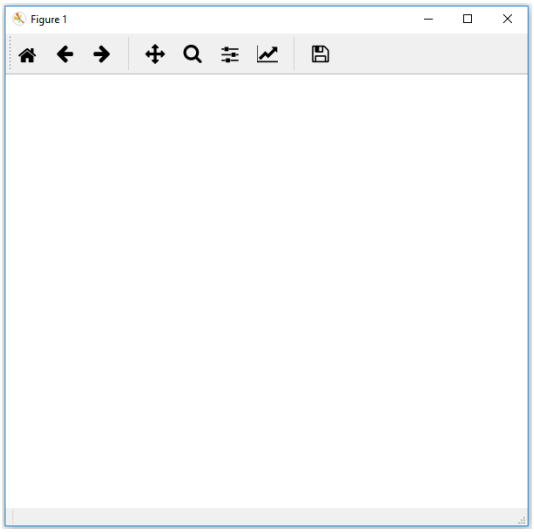
NumPy is niet noodzakelijk, maar zal meestal gebruikt worden om de data-objecten (vectoren, matrices) voor te stellen waarvan we figuren willen maken. De functies in de `pyplot` interface gaan steeds uit van een huidige figuur (*current figure*) en coördinatenstelsel (*current axes*). Met de functie `figure()` wordt een lege figuur aangemaakt, die meteen beschouwd wordt als de **actieve** figuur. Alle grafische instructies die daarna volgen, zullen inwerken op deze *current figure*. We kunnen aan deze functie ook een geheel getal meegeven als (optioneel) argument, waarna de figuur dit nummer toegewezen krijgt. Zo geeft volgend fragment Figuur 11.2 als resultaat, indien gebruik gemaakt wordt van de interactieve `QtAgg` backend.

Fragment 11.1: `lege_figuur.py`

```
3 plt.figure(1)
4 # plt.show()  # enkel indien plt.ion() niet werd gebruikt
```

Opmerking: werken in niet-interactieve sessies
Wanneer gewerkt wordt in niet-interactieve sessies (een script wordt dan uitgevoerd via een terminal), is de bovenstaande werkwijze nog steeds van toepassing. Echter, de figuurvensters zoals in

[3]We gaan er in de volgende codefragmenten steeds van uit dat dit de vijf twee regels van het script zijn



Figuur 11.2: Resultaat Fragment 11.1.

Fig. 11.1 zullen maar **zeer kort zichtbaar** zijn en vervolgens verdwijnen (vaak zo kortstondig dat ze amper waarneembaar zijn). Als een Python-script volledig is uitgevoerd, zal het *process* dat werd opgestart immers beëindigd worden met het sluiten van alle vensters tot gevolg. Dit automatisch afsluiten verhinderen kan eenvoudig door op het einde van het script om gebruikersinput te vragen. Het process wordt dan pas afgesloten nadat de gebruiker input heeft gegeven. Dit kan bijvoorbeeld door de volgende instructie achteraan te plaatsen:

```
input("Druk enter om af te sluiten.")
```

Zoals hiervoor vermeld, gaan we er in dit hoofdstuk echter vanuit dat er steeds gewerkt wordt in een interactieve sessie en is deze instructie bijgevolg **overbodig**.

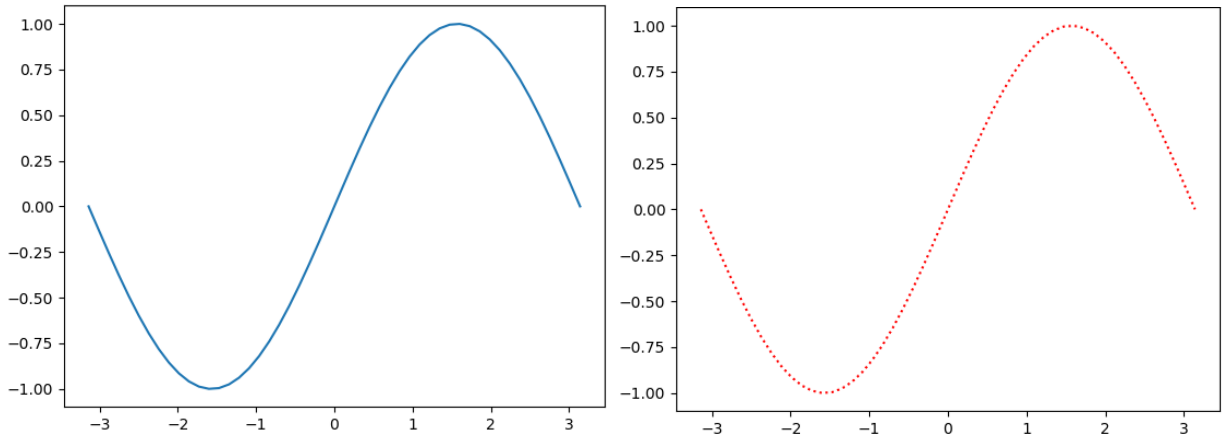
11.3.1 Het maken van een grafiek

11.3.1.1 De functie `plot()`

De functie `plot()` kan gebruikt worden om punten met opgegeven coördinaten in een XY-assenstelsel te plotten en (eventueel) te verbinden. Deze functie kan als volgt (zie Fragment 11.2) gebruikt worden om de grafiek te plotten van bijvoorbeeld de functie $f : x \rightarrow \sin(x)$ voor $x \in [-\pi, \pi]$. Het resultaat van de uitvoer van dit stukje code wordt weergegeven in Figuur 11.3(a).

Fragment 11.2: `plot_sinus.py`

```
1 x = np.linspace(-np.pi, np.pi)  # maak een vector x met 50 x-waarden
2                                 # in het interval [-pi, pi]
3 y = np.sin(x)                   # bereken de functiewaarden
4 plt.figure()                    # maak een leeg figuurvenster (optioneel)
5 plt.plot(x, y)                  # teken de grafiek
```



(a) Blauwe volle lijn

(b) Rode stippelijn met rondjes

Figuur 11.3: Gebruik van de functie `plot`.

Op de eerste twee regels van codefragement 11.2 worden de numpy arrays `x` en `y` aangemaakt. Bij het gebruik van `matplotlib` wordt vaak gewerkt met `numpy` arrays, maar dit is niet noodzakelijk. Zo aanvaardt de functie `plot()` ook lijsten van getallen (floats of integers). De enige vereiste is dat de argumenten van de functie `plot()` array-like moeten zijn: ze moeten m.a.w. converteerbaar zijn naar `numpy` arrays. Ten slotte worden met de functie `plot()` de y-coördinaten tegenover de x-coördinaten geplot. Vermits we geen lijntype of lijnkleur hebben opgegeven, wordt standaard een blauwe volle lijn gebruikt.

Willen we een lijntype, een lijnkleur en/of een puntmarkering specificeren, dan moeten we één string `stijlopties` als derde argument meegeven aan de functie `plot()`. Zo zal de combinatie `"r:"` een plot met een rode stippellijn maken. Er zijn talloze combinaties van lijntype, lijnkleur en/of puntmarkering mogelijk. Een beperkt overzicht van de beschikbare stijlopties wordt weergegeven in Tabel 11.1. Raadpleeg de documentatie voor meer stijlopties.

Tabel 11.1: Enkele stijlopties van de functie `plot`.

Letter	Kleur	Symbool	Lijntype	Symbool	Puntmarkering
`b`	blauw	`-`	volle lijn	`+`	plusteken
`c`	cyaan	`--`	streeplijn	`o`	open rondje
`y`	geel	`:`	stippellijn	`*`	sterretje
`g`	groen	`-.`	streep-punt	`x`	x-teken
`m`	magenta			`.`	punt
`r`	rood			`^`	driehoek
`w`	wit			`s`	vierkantje
`k`	zwart			`d`	ruit

Opdracht 11.1 (`kiemplantje.py`)

De groei van een kiemplantje over een periode van 12 dagen wordt gegeven in onderstaande tabel:

dag	1	2	3	4	5	6	7	8	9	10	11	12
lengte [mm]	0	0	0	0	1	4	6	10	16	20	27	33

Maak een script `kiemplantje.py` aan. Geef de dagen en de lengte in als lijsten (lists). Plot de lengte van het kiemplantje over de groeiperiode met **zwarte vierkantjes**.

11.3.1.2 Tekst toevoegen

Zoals blijkt uit Figuur 11.3 is er geen titel, zijn de assen niet benoemd, is er geen legende, ... Kortom, deze figuur kan wat meer opmaak gebruiken.

Een titel voor de figuur en labels voor de X- en Y-assen kunnen toegevoegd worden door gebruik te maken van respectievelijk de functies `title()`, `xlabel()` en `ylabel()`:

Fragment 11.3: `plot_sinus.py` (vervolg)

```
6 plt.title("De sinusfunctie")
7 plt.xlabel("x")
8 plt.ylabel("sin(x)")
```

Het resultaat van deze instructies wordt weergegeven in Figuur 11.4(a).

Om een legende toe te voegen wordt de functie `legend()` gebruikt:

Fragment 11.4: `plot_sinus.py` (vervolg)

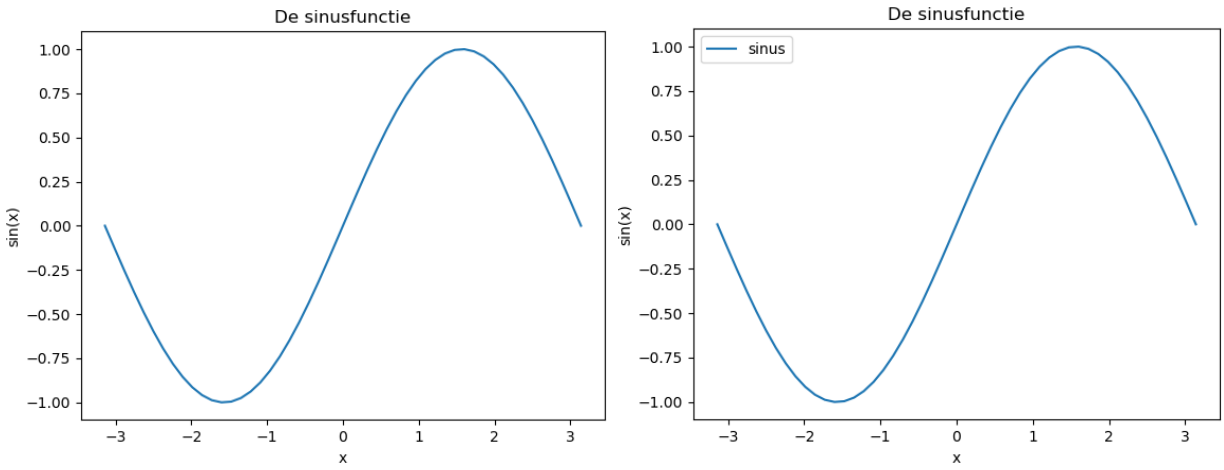
```
9 plt.legend(["sinus"])
```

Merk op dat er rechte haken worden gebruikt: de **labels moeten als een lijst (*list*) meegegeven worden**. De positie van de legende kan gekozen worden door de locatie (*location*) te specificeren met behulp van het sleutelargument (*keyword*) `loc`. Om de legende bijvoorbeeld linksboven te plaatsen, moeten we `loc = "upper left"` als tweede argument meegeven aan `legend()` (Fragment 11.5). Het resultaat van deze instructie wordt weergegeven in Figuur 11.4(b).

Fragment 11.5: `plot_sinus.py` (vervolg)

```
10 plt.legend(["sinus"], loc = "upper left")
```

Een overzicht van enkele mogelijke locaties voor de legende wordt samengevat in Tabel 11.2. Merk op dat je de locatie ook kunt aanduiden met een cijfer.



(a) Toevoegen van een titel en labels

(b) Toevoegen van een legende

Figuur 11.4: De geannoteerde plot van de functie $sin(x)$.

Tabel 11.2: De betekenis van de input `loc` bij de functie `legend()`.

Locatie	**Positie van de legende**	**Locatiecode**
`"best"`	binnen het assenstelsel, door een minimaal aantal punten te verbergen (**default**)	0
`"upper right"`	in de **rechterboven**hoek van het assenstelsel	1
`"upper center"`	**boven**aan **gecentreerd**	9
`"upper left"`	in de **linkerboven**hoek	2
`"lower right"`	in de **rechterbenedenhoek**	4
`"lower center"`	**onder**aan **gecentreerd**	8
`"lower left"`	in de **linkeronder**hoek	3

Opdracht 11.2 (`kiemplantje.py`)

Geef de plot uit Opdracht 11.1 een passende titel, benoem de assen en plaats een legende **rechtsonder**. Vul hiertoe je script `kiemplantje.py` verder aan.

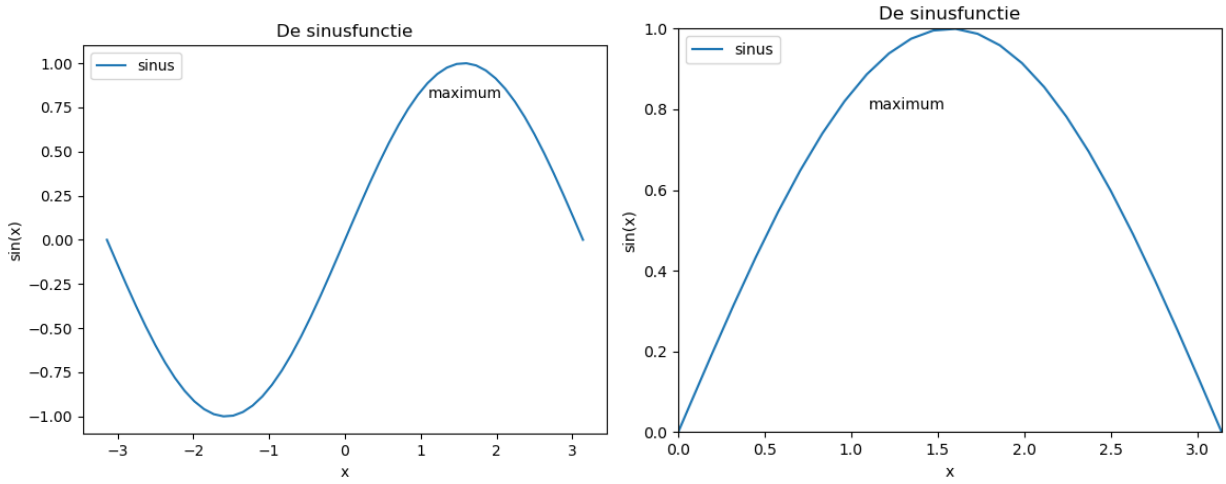
Met de functie `text()` kan tekst op een opgegeven locatie in het huidige assenstelsel geplaatst worden. Willen we bijvoorbeeld de tekst `"maximum"` op het punt met coördinaten `(1.1, 0.8)` plaatsen, dan kan dit als volgt:

Fragment 11.6: `plot_sinus.py` (vervolg)

```
11 plt.text(1.1, 0.8, "maximum")
```

Het resultaat wordt weergegeven in Figuur 11.5(a).

Soms willen we ook de tekst op onze figuur (in titels, aslabels, legendes,...) mooi opmaken. Daarbij kunnen we gebruik maken van TeX markup, wat wordt aangegeven door twee dollartekens in de string `"$...$"`. Met bijvoorbeeld `"$_{...}$"` en `"$^{...}$"` geven we aan om de tekst tussen accolades in resp. sub- en superscript te plaatsen. We zullen de TeX markup gebruiken in sommige voorbeelden, maar beschouwen dit als **uitbreidingsleerstof**.



(a) Gebruik van `text()`

(b) Wijziging van het bereik van de assen

Figuur 11.5: Gebruik van de functies `text()` en `axis()`.

11.3.1.3 Bereik van de assen

Het bereik van beide assen kan tegelijkertijd aangepast worden met de functie `axis()`. Deze functie verwacht een lijst met 4 elementen: de grenzen van de X- en Y-assen. In CodeFragment 11.7 beperken we ons bijvoorbeeld tot het gebied $[0, \pi] \times [0, 1]$. Het resultaat wordt weergegeven in Figuur 11.5(b).

Fragment 11.7: `plot_sinus.py` (vervolg)

```
12 plt.axis([0, np.pi, 0, 1])
```

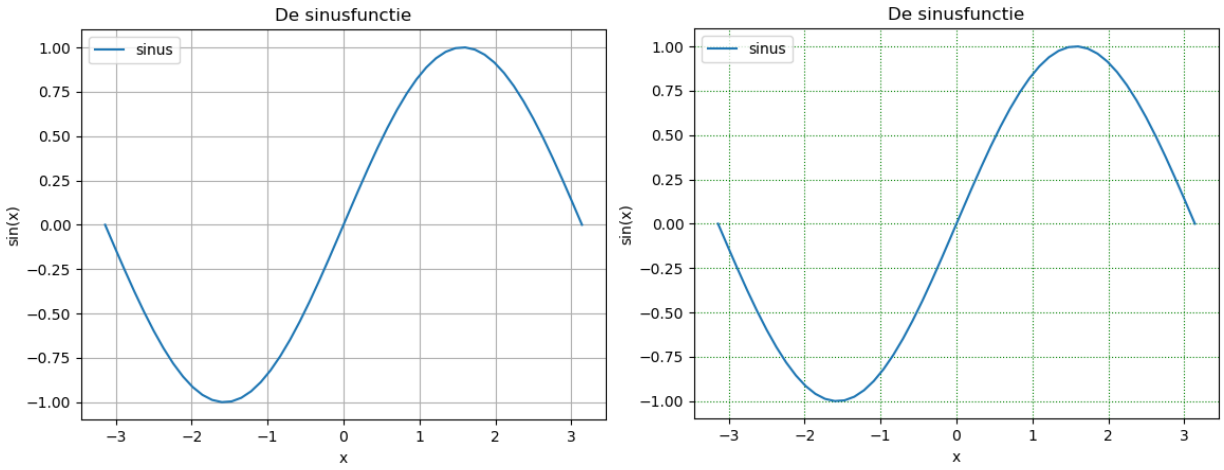
Met de functies `xlim()` en `ylim()` kan het bereik van resp. de X-as en de Y-as afzonderlijk ingesteld worden. Met onderstaande instructies bekomen we hetzelfde resultaat als in Figuur 11.5(b):

Fragment 11.8: `plot_sinus.py` (vervolg)

```
13 plt.xlim(0, np.pi)
14 plt.ylim(0, 1)
```

11.3.1.4 Gridlijnen

Met de instructie `grid()` of `grid(True)` worden de gridlijnen van het assenstelsel weergegeven met volle grijze lijnen. Het visualiseren van deze gridlijnen kan nuttig zijn bij het bepalen van de bijhorende functiewaarden bij een gegeven waarde op de X-as. Met de instructie `grid(False)` kunnen de gridlijnen terug onzichtbaar gemaakt worden. De lijnkleur en -stijl kunnen respectievelijk aangepast worden met de opties `color` en `linestyle`. Het gebruik van de functie `grid()` wordt geïllustreerd in Figuur 11.6.



(a) Default grid: met `plt.grid()`

(b) Aangepast grid: groene stippellijnen met `plt.grid(color = "g", linestyle = ":")`

Figuur 11.6: Gebruik van de functie `grid()`.

Opdracht 11.3 (`kiemplantje.py`)

Stel de eindpunten van de Y-as uit de plot in Opdracht 11.2 zo in dat enkel de lengtes vanaf 1 weergegeven worden. Vul hiertoe je script `kiemplantje.py` verder aan.

Opdracht 11.4 (`sinExp.py`)

Maak een script `sinExp.py` aan waarin je de grafiek van de functie $f : x \to e^{1.2\,x} \sin(10\,x + 5)$ tekent over het interval $[0, 5]$ met een **magenta puntstreeplijn**. Gebruik voldoende punten zodat de curve een mooi glad verloop vertoont. Benoem de assen, geef de grafiek een titel en plaats de legende **linksonder**.

11.3.2 Opslaan en sluiten van figuren

Een figuur kan opgeslagen worden met de functie `savefig()`. Het volstaat een bestandsnaam (met een extensie zoals `png`, `jpeg`, …) op te geven waarna de huidige actieve figuur zal worden bewaard in de huidige werkmap. Er kan ook een *path* worden meegegeven (absoluut of relatief t.o.v. de huidige werkmap). Zo kan het resultaat van Opdracht 11.3 als `png` worden opgeslagen in de subdirectory `figuren` met de instructie

```
>>> plt.savefig("figuren/kiemplantje.png")
```

Bij gebruik van een interactieve backend zoals `QtAgg` kan je figuren ook opslaan via het menu in het grafisch pop-up venster. Bij zo een backend is het ook van belang om figuren weer te sluiten. Als er meerdere figuren open staan, kunnen deze allemaal tegelijkertijd gesloten worden met de functie `close()`. Het volstaat om de string `"all"` mee te geven als argument:

```
>>> plt.close("all")   # sluit alle openstaande figuren
```

11.3.3 Samenvoegen en vergelijken van plots

11.3.3.1 Overlay plots

Wanneer twee of meer grafieken met elkaar vergeleken moeten worden, is het handig wanneer ze in eenzelfde assenstelsel geplot worden. Dit noemen we *overlay* plots. Een eerste manier om deze plots te maken is door herhaaldelijk de functie `plot()` op te roepen:

```
plt.plot(x1, y1, stijlopties1)
plt.plot(x2, y2, stijlopties2)
...
```

Nieuwe plots worden toegevoegd aan de bestaande plots, m.a.w. oude plots worden **niet overschreven** met nieuwe plots. Het script in Fragment 11.9 illustreert deze werkwijze voor het creëren van een overlay plot van de grafiek van $f_1 : x \to \sin(x)$, $f_2 : x \to x$ en $f_3 : x \to x - \frac{x^3}{3!} + \frac{x^5}{5!}$ (f_1 en f_2 zijn benaderingen). Het resultaat wordt weergegeven in Figuur 11.7.

Fragment 11.9: `overlayplot.py`

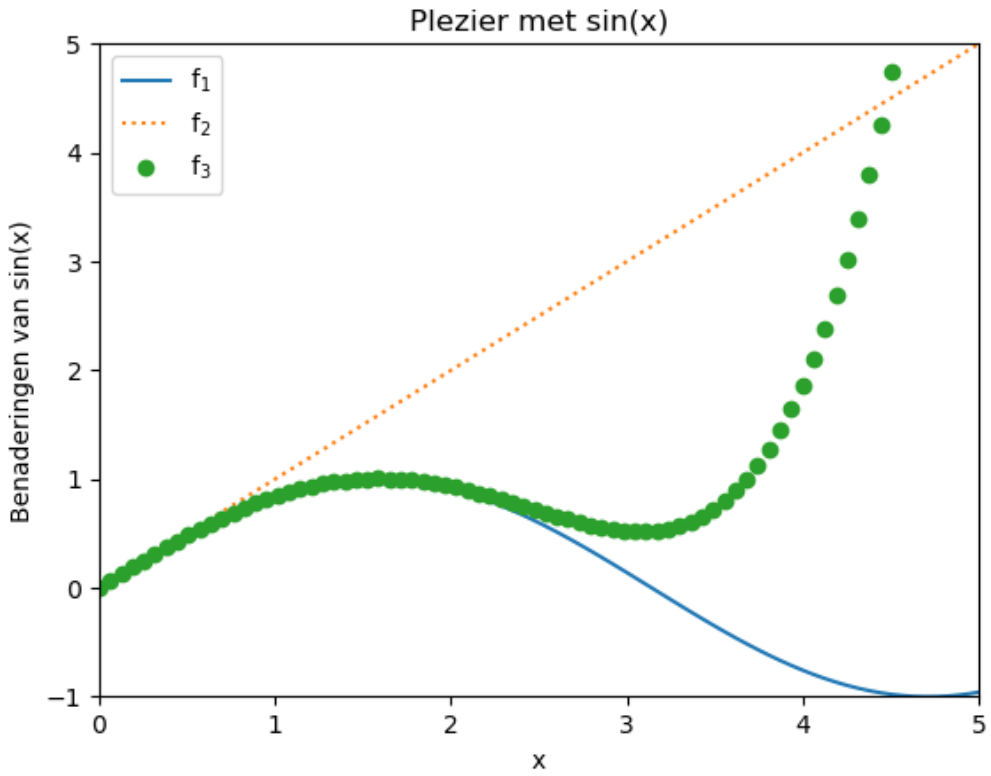
```
1  plt.figure()
2  x = np.linspace(0, 2*np.pi, 100)        # vector x met 100 punten
3  y1 = np.sin(x)                          # bereken y1
4  y2 = x                                  # bereken y2
5  y3 = x - (x**3)/6 + (x**5)/120          # bereken y3
6  plt.plot(x, y1)                         # plot f1 met volle lijn,
7  plt.plot(x, y2, ":")                    # plot f2 met stippellijn en
8  plt.plot(x, y3, "o")                    # plot f3 met bolletjes
9  plt.axis([0, 5, -1, 5])                 # zoom in tot nieuwe eindptn
10 plt.xlabel("x")                         # plaats label voor X-as
11 plt.ylabel("Benaderingen van sin(x)")   # plaats label voor Y-as
12 plt.title("Plezier met sin(x)")         # plaats titel
13 plt.legend(["f$_1$", "f$_2$", "f$_3$"]) # plaats legende
```

Als de data voor de verschillende grafieken tegelijkertijd beschikbaar zijn, kan met de functie `plot()` een overlay plot gemaakt worden door meerdere inputs mee te geven:

```
plt.plot(x1, y1, stijlopties1, x2, y2, stijlopties2, ...)
```

De plotinstructies op regels 8–10 in Fragment 11.9 kunnen bijgevolg vervangen worden door 1 één instructie:

```
plt.plot(x, y1, x, y2, ':', x, y3, 'o')
```



Figuur 11.7: Resultaat van Fragment 11.9: voorbeeld van een overlay plot van de functies $f_1 : x \to \sin(x)$, $f_2 : x \to x$ en $f_3 : x \to x - \frac{x^3}{3!} + \frac{x^5}{5!}$.

Opdracht 11.5 (`mijnOverlayplot.py`)

Beschouw de functies f, g en h over het interval $[0, 3.5]$:

$$\begin{cases} f : x \to 0.1\,x^2 \\ g : x \to \cos^2(x) \\ h : x \to e^{-0.3\,x} \end{cases}$$

Maak een script `mijnOverlayplot.py` aan en voer de volgende opdrachten uit:

1. Definieer de 3 functies f, g en h als 3 **lambda-functies** (zie Sectie 5.6) `f`, `g` en `h`.
2. Plot f, g en h in eenzelfde assenstelsel. Zorg ervoor dat elke plot een andere kleur en lijntype krijgt (vrij te kiezen).
3. Zorg voor een legende op een geschikte plaats.
4. Sla de figuur op als `mijnOverlayplot.png`.

11.3.3.2 Subplots

Om meerdere plots naast en/of onder elkaar te plaatsen in eenzelfde figuurvenster kan men gebruik maken van de functie `subplot()`. Deze functie wordt als volgt opgeroepen:

```
plt.subplot(m, n, p)
```

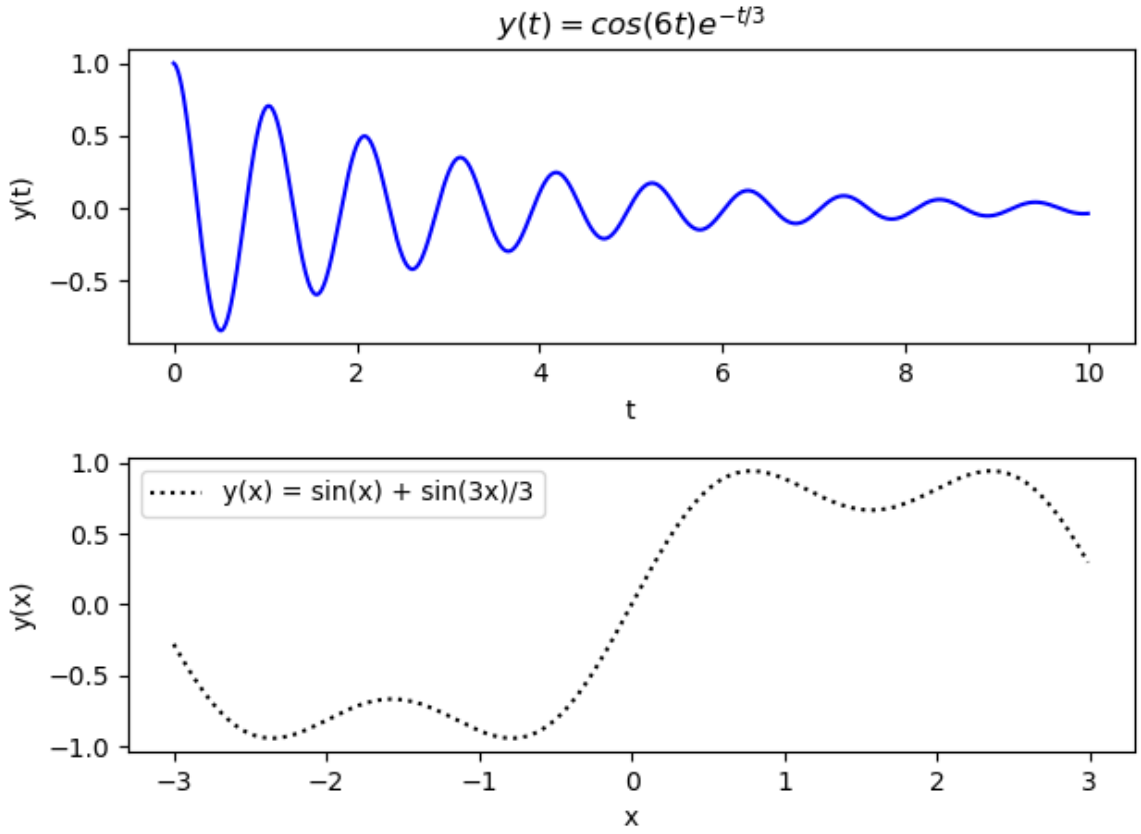
Deze functie verdeelt het grafisch venster in `m` bij `n` rechthoekige panelen die elk een eigen assenstelsel hebben en stelt het `p`-de paneel in als het huidige paneel zodat de volgende functieoproep naar een plotfunctie zoals `plot()` in dit paneel zal plotten. De panelen worden **rij per rij genummerd** en **kolom per kolom**, startend links bovenaan.

Zo verdeelt het script `subplots.py` in Fragment 11.10 het grafisch venster in twee panelen en zet $f_1 : t \to \cos(6\,t)\,e^{-t/3}$ versus t uit in het bovenste paneel met een blauwe volle lijn, en $f_2 : x \to \sin(x) + \frac{\sin(3\,x)}{3}$ versus x in het onderste paneel met een zwarte stippellijn (zie Figuur 11.9).

Overlappende subplots
In Fragment 11.10 wordt gebruik gemaakt van de functie `tight_layout()` op. Deze zorgt ervoor dat de subplots elkaar **niet** (gedeeltelijk) overlappen.

Fragment 11.10: `subplots.py`

```
plt.figure()
plt.subplot(2, 1, 1)                        # eerste subplot
t = np.arange(0, 10, 0.01)
y1 = np.cos(6*t)*np.exp(-t/3)
plt.plot(t, y1, "b-")
plt.title("$y(t) = cos(6t)e^{-t/3}$")
plt.xlabel("t")
plt.ylabel("y(t)")

plt.subplot(2, 1, 2)                        # tweede subplot
x = np.arange(-3, 3, 0.01)
y2 = np.sin(x) + np.sin(3*x)/3
plt.plot(x, y2, "k:")
plt.legend(["y(x) = sin(x) + sin(3x)/3"])
plt.xlabel("x")
plt.ylabel("y(x)")

plt.tight_layout()  # maak subplots passend
```



Figuur 11.8: Resultaat van Fragment 11.10: voorbeeld van een subplot van $f_1 : t \to \cos(6\,t)e^{-t/3}$ (boven) en $f_2 : x \to \sin(x) \; + \; \frac{\sin(3\,x)}{3}$ (onder).

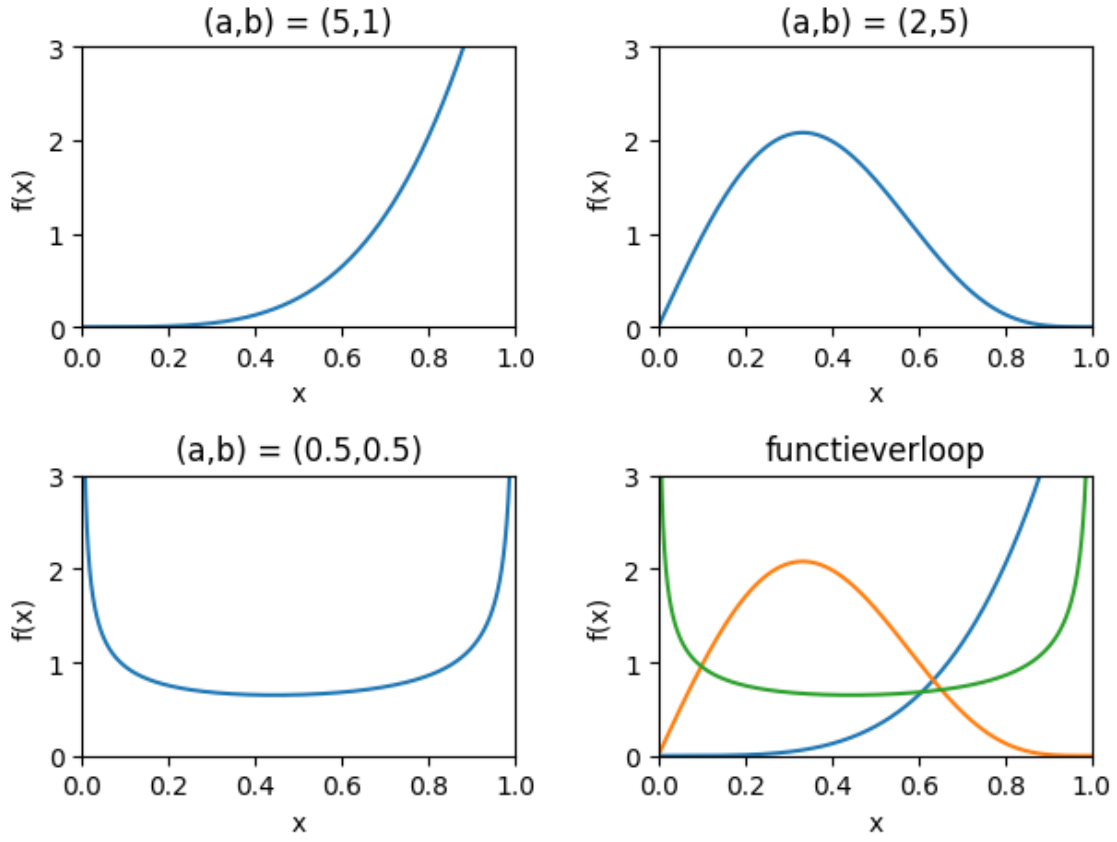
Opdracht 11.6 (`mijnSubplot.py`)

Gegeven de functie

$$f : x \to a\,b\,x^{a-1}(1-x^a)^{b-1}\,,$$

met $x \in [0,1]$ en a en b twee parameters.

1. Maak een script `mijnSubplot.py` aan.
2. Definieer de functie f als een **lambda-functie** `f` met a en b als extra parameters.
3. Verdeel het grafisch venster in 4 panelen (2 rijen, 2 kolommen).
4. Teken de functie f over het interval $[0,1]$ met $a = 5$ en $b = 1$ in het eerste paneel. Gebruik voldoende punten. Plaats een titel met daarin de waarden van a en b, benoem de assen, en stel het bereik van de Y-as in op $[0,3]$.
5. Teken de functie f nu met $a = 2$ en $b = 5$ in het tweede paneel, en met $a = 0.5$ en $b = 0.5$ in het derde paneel. Plaats ook hier een titel met daarin de waarden van a en b, benoem de assen, en stel het bereik van de Y-as in op $[0,3]$.
6. Maak een overlay plot met de drie plots samen in het vierde en laatste paneel. Plaats opnieuw een titel, benoem de assen, en stel het bereik van de Y-as in op $[0,3]$.
7. Sla de figuur op als `mijnSubplot.png`. Het resultaat wordt getoond in Figuur 11.9.

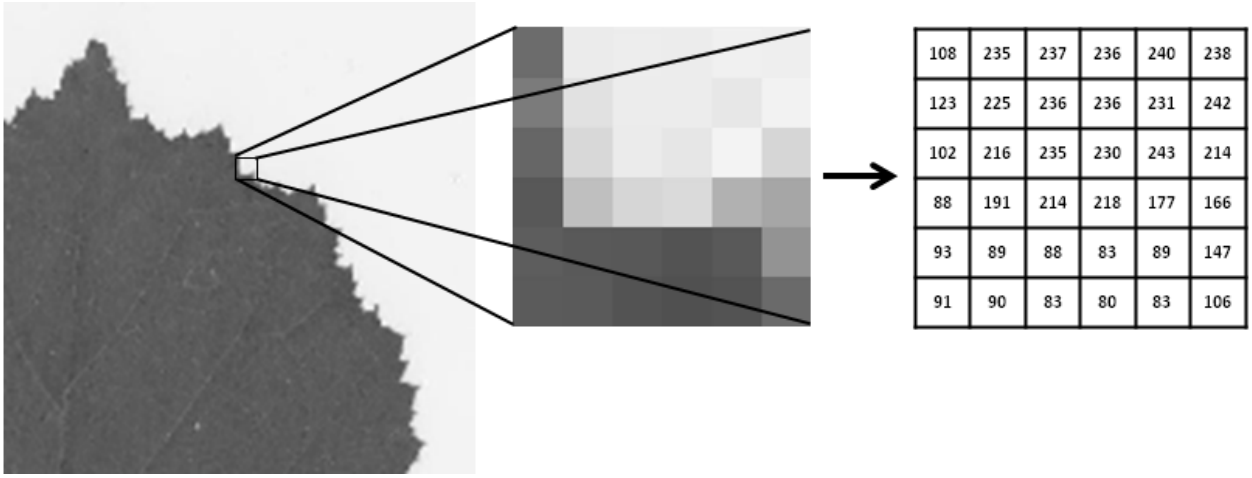


Figuur 11.9: Resultaat van Opdracht 11.6 (kleur en lijntype kunnen anders zijn).

11.3.4 Weergave van afbeeldingen

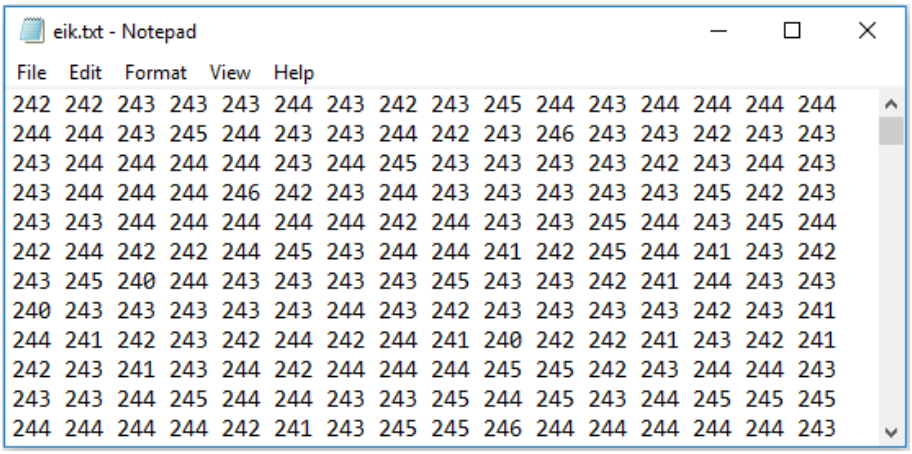
11.3.4.1 De functie `imshow()`

Een afbeelding kan op een eenvoudige manier voorgesteld worden door een matrix die iedere pixel van het beeld voorstelt aan de hand van zijn kleurcode. Een typisch voorbeeld is een grijswaarden-afbeelding waarbij de pixels voorgesteld worden door een grijswaarde tussen 0 (zwart) en 255 (wit). Donkere pixels krijgen dus een lagere waarde toegekend dan lichtere pixels. Dit wordt geïllustreerd voor een blad van een hazelaar in Figuur 11.10. Het gebiedje binnen het vierkant wordt daarbij uitvergroot en de overeenkomstige grijswaarden worden getoond in matrixvorm.



Figuur 11.10: Voorstelling van pixels door grijswaarden.

Dergelijke matrices met grijswaarden kunnen eenvoudig worden opgeslagen in tekstbestanden (die de extensie `.txt` hebben). In Figuur 11.11 wordt een deel van de inhoud van het tekstbestand `eik.txt` weergegeven.



Figuur 11.11: Inhoud van het bestand `eik.txt`.

Zoals gezien in Sectie 8.2.16 kunnen dergelijke tekstbestanden bestanden eenvoudig ingelezen worden met de `numpy`-functie `loadtxt()`:

```
>> np.loadtxt("eik.txt")
array([[242, 242, 243, ..., 244, 243, 245],
       [244, 242, 244, ..., 244, 243, 244],
        ...,
       [241, 241, 242, ..., 242, 243, 244]])
```

Met behulp van de functie `imshow()` uit de submodule `pyplot` kunnen `numpy` arrays gevisualiseerd worden. Deze functie vereist een `numpy` array als verplicht eerste argument. Optioneel kan je een kleurenpalet (*color map*) specificeren met het argument `cmap`. Deze parameter geeft aan hoe een waarde in de matrix vertaald wordt in een kleur. Wordt de parameter `cmap` niet opgegeven, dan wordt het kleurenpalet `viridis` gebruikt. In het geval van grijswaardenafbeeldingen is het gebruik van het kleurenpalet `gray` aangewezen. Fragment 11.11 illustreert het gebruik van de functie `imshow()` en het resultaat wordt weergegeven in Figuur 11.12.

Fragment 11.11: `eik.py`

```
1 eik = np.loadtxt("eik.txt")
2 plt.figure(1)
3 plt.imshow(eik)                # default kleurenpalet viridis
4
5 plt.figure(2)
6 plt.imshow(eik, cmap = "gray") # zwart-wit weergave
```

11.3.4.2 De functie `imread()`

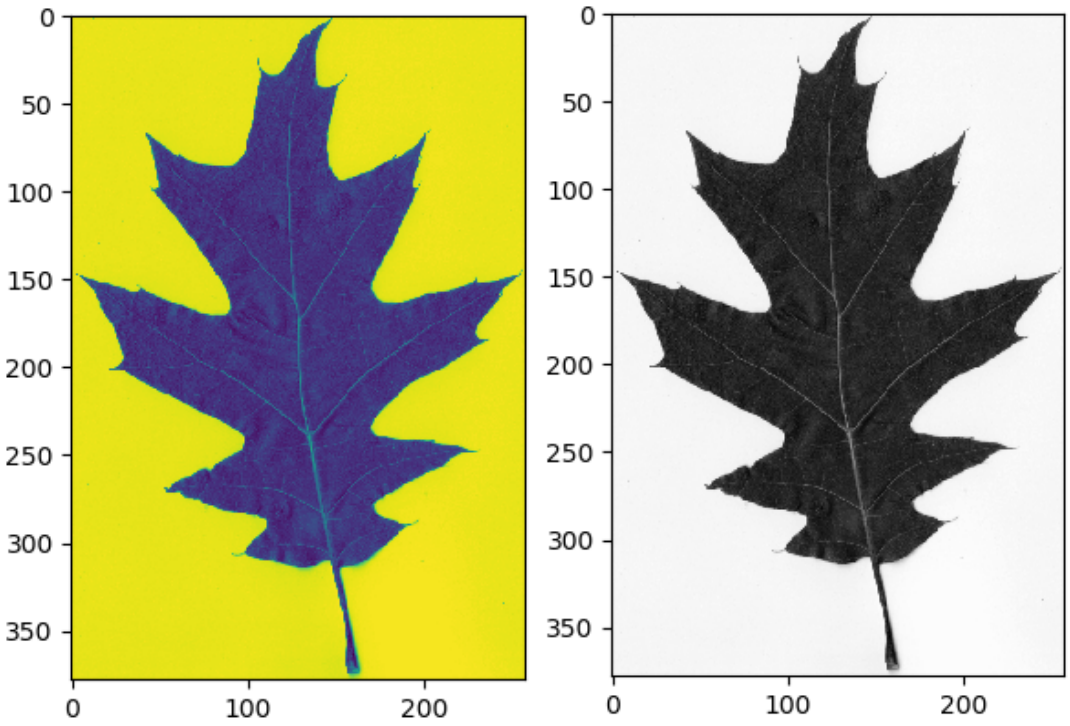
In de praktijk worden afbeeldingen opgeslagen als `.png`, `.jpg`, `.jpeg`, `.gif`, ... bestanden. Dergelijke bestanden kunnen niet ingelezen worden met de functie `loadtxt()` uit de module `numpy`. De submodule `pyplot` bevat echter de functie `imread()` die ons toelaat afbeeldingen in te lezen als `numpy` array [4]. Het volstaat de naam van de in te lezen afbeelding mee te geven als eerste verplicht argument, waarbij `imread()` het bestandsformaat afleidt o.b.v. de extensie in de bestandsnaam. Optioneel kan het bestandsformaat ook worden meegegeven met het *keyword*-argument **`format`** (**niet** te verwarren met de functie **`format()`** uit Hoofdstuk 4).

Het visualiseren van afbeeldingen kan uiteraard op de gebruikelijke manier met de functie `imshow()` uit de submodule `pyplot`. Dit wordt geillustreerd in codefragment 11.12 voor het bestand `dier.png` (Figuur 11.13).

Fragment 11.12: `dier.py`

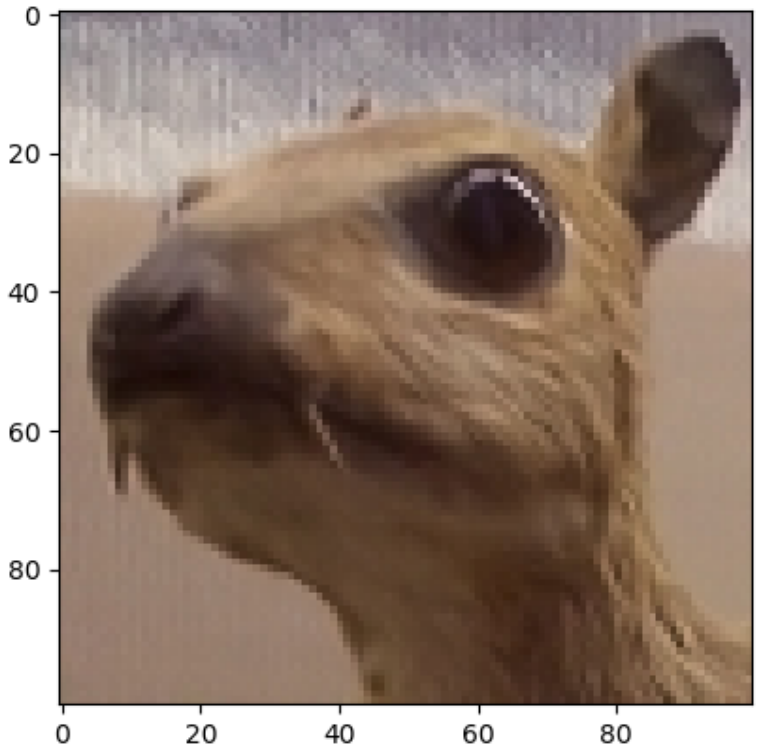
```
1 dier = plt.imread("dier.png") # inlezen met imread
2
3 plt.figure()
4 plt.imshow(dier)
```

[4] De functie `imread()` retourneert een `numpy` array waarvan de dimensies afhangen van het ingelezen bestand. Wordt een grijswaardenafbeelding ingelezen, dan retourneert `imread()` een 2-dimensionale `numpy` array, bij een kleurenafbeelding is dit een 3-dimensionale array.



(a) Default kleurenpallet `viridis` (b) Kleurenpallet `gray`

Figuur 11.12: Gebruik van de functie `imshow()` (codefragment 11.11).



Figuur 11.13: Resultaat van codefragment 11.12.

11.3.5 Overige plotfuncties

Er bestaan in Python nog vele andere functies voor het maken van tweedimensionale plots. In deze sectie zullen we enkele vaak gebruikte plotfuncties verder toelichten.

11.3.5.1 Taartdiagrammen

Met de functie `pie()` kunnen taartdiagrammen (*pie charts*) gemaakt worden. Deze functie verwacht als eerste verplicht argument een lijst of `numpy` array met de aantallen (of percentages) die voorgesteld moeten worden. Daarnaast kunnen nog een aantal optionele argumenten worden meegegeven, zoals:

- `labels`: een lijst met labels die als tekst bij de taartstukken moeten geplaatst worden
- `colors`: een lijst met kleuren om de taartstukken mee op te vullen.

Indien ze worden meegegeven, moeten deze lijsten **even lang** zijn als het eerste argument.

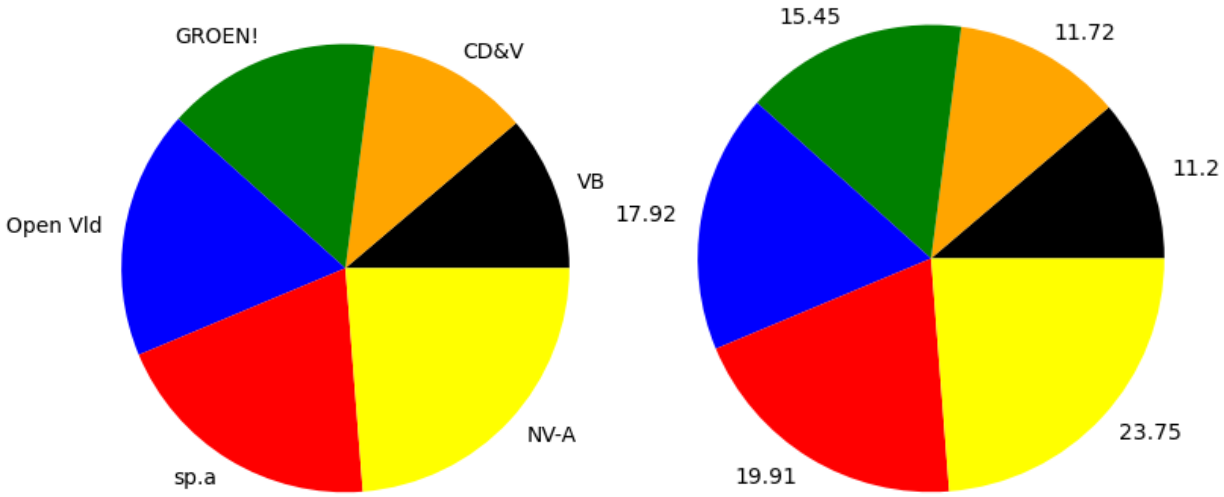
Deze functie kunnen we illustreren a.d.h.v. de uitslagen van de federale verkiezingen van 2010 voor de stad Gent. Met de functie `pie()` genereren we twee taartdiagrammen: de eerste met de partijnamen als labels (Figuur 11.14(a)), de tweede met de percentages als labels (Figuur 11.14(b)). Bovendien wordt aan iedere partij vooraf een kleur toegekend.

Fragment 11.13: `verkiezingsuitslagen.py`

```
1 percentages = [11.2, 11.72, 15.45, 17.92, 19.91, 23.75]
2 partijen = ["VB", "CD&V", "GROEN!", "Open Vld", "sp.a", "NV-A"]
3 kleuren =  ["black", "orange", "green", "blue", "red", "yellow"]
4
5 plt.figure()
6 plt.pie(percentages, labels = partijen, colors = kleuren)
7
8 plt.figure()
9 plt.pie(percentages, labels = percentages, colors = kleuren)
```

Tabel 11.3: Verkiezingsuitslag Gent.

partij	% van de stemmen
VB	11.25
CD&V	11.72
GROEN!	15.45
Open Vld	17.92
sp.a	19.91
NV-A	23.75



(a) Partijnamen als labels (b) Percentages als labels

Figuur 11.14: Resultaat van fragment 11.13.

11.3.5.2 Staafdiagrammen

Met de functie `bar()` kunnen staafdiagrammen (*bar charts*) gemaakt worden. Deze functie verwacht twee verplichte argumenten: het eerste argument bevat de x-coördinaten van de linkerbenedenhoek van de staven (bars), en het tweede argument bevat de hoogtes van deze staven. Een eenvoudige manier om de x-coördinaten te definiëren is het creëren van een lijst met de getallen 1 t.e.m. n met n het aantal staven.

Fragment 11.14 maakt een barplot van de data in Tabel 11.3.

Fragment 11.14: `verkiezingsuitslagen.py` (vervolg)

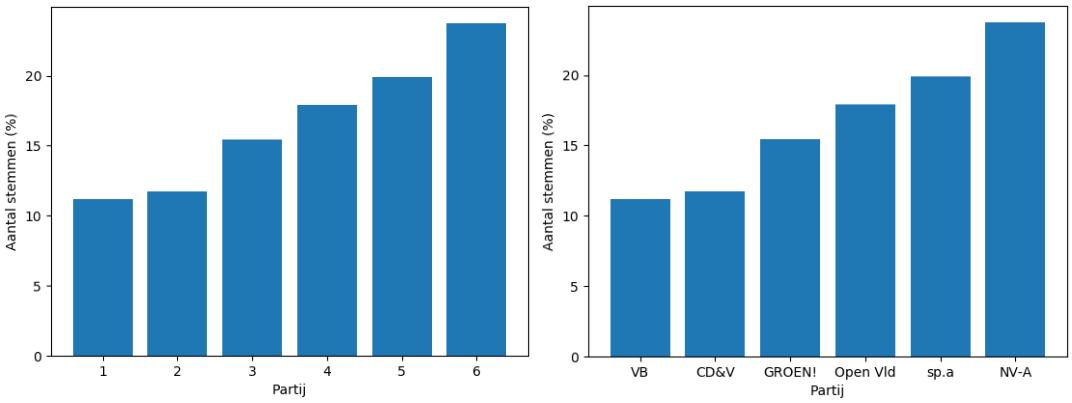
```
11 plt.figure()
12 x = range(1, len(percentages) + 1)  # x = 1, 2, ..., 6
13 plt.bar(x, percentages)
14 plt.xlabel("Partij")
15 plt.ylabel("Aantal stemmen (%)")
```

Het resultaat wordt weergegeven in Figuur 11.15(a).

Merk op dat in Figuur 11.15(a) de staven genummerd zijn. We kunnen deze cijfers vervangen door de partijnamen door gebruik te maken van de optie `tick_label`:

Fragment 11.15: `verkiezingsuitslagen.py` (vervolg)

```
11 plt.figure()
12 plt.bar(x, percentages, tick_label = partijen) # namen als labels
13 plt.xlabel("Partij")
14 plt.ylabel("Aantal stemmen (%)")
```



(a) Nummers als labels

(b) Partijnamen als labels

Figuur 11.15: Resultaat van Fragmenten 11.14 (a) en 11.15 (b).

Het resultaat wordt weergegeven in Figuur 11.15(b).

Opdracht 11.7 (`telKlinkers.py`)

Het tekstbestand `dna.txt` bevat twee paragrafen tekst met uitleg over dna. De bedoeling is om te tellen hoeveel keer elke klinker voorkomt in deze tekst en deze aantallen weer te geven in een **taart**diagram en een **staaf**diagram. We maken hierbij **geen onderscheid** tussen kleine letters en hoofdletters. Maak hiertoe een script `telKlinkers.py` aan waarin je de volgende deelopdrachten uitvoert:

1. Lees de inhoud van `dna.txt` in met het volgend stukje code [5]:

```
f = open("dna.txt", "r", encoding = "iso-8859-15")
dna = f.read().lower()  # omzetten naar kleine letters
f.close()
```

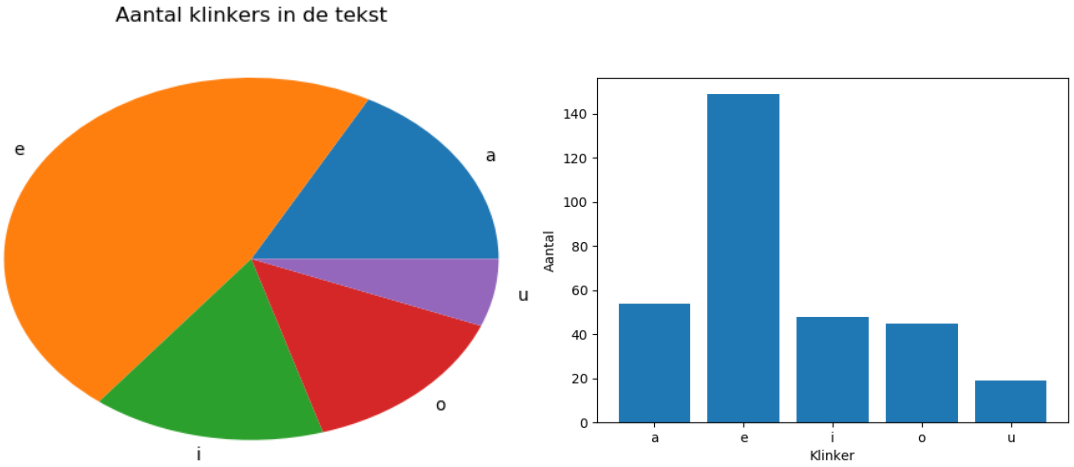
Met als resultaat **één lange string met uitsluitend kleine letters**:

```
>>> dna
'het dna (of desoxyribonucleïnezuur) is de belangrijkste drager van
erfelijke informatie in alle levende organismen. een
dna-molecule bestaat uit twee lange strengen van nucleotiden, die zich
samen buigen tot een dubbele helix zoals afgebeeld in fig. 3.1.
...
de opeenvolging van nucleotiden in één enkele streng wordt een
dna-sequentie genoemd.'
```

2. Tel het aantal klinkers en bewaar deze in een lijst of array.

[5] Het argument `encoding` zorgt ervoor dat dit specifiek bestand correct wordt ingelezen op alle besturingssystemen.

3. Plot deze aantallen in een **taart**diagram in een eerste figuur (cf. Fig. 11.16(a)).
4. Plot deze aantallen in een **staaf**diagram in een tweede figuur (cf. Fig. 11.16(b)).

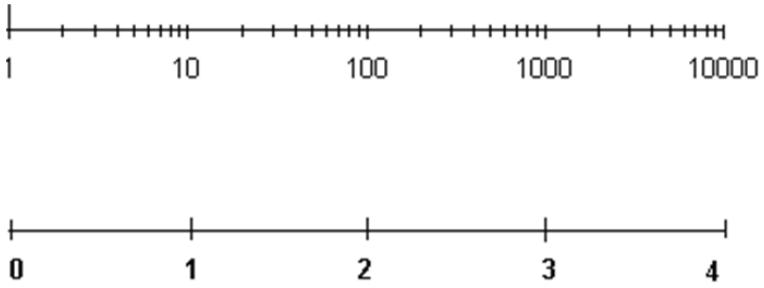


(a) Taartdiagram (b) Staafdiagram

Figuur 11.16: Resultaat van Opdracht 11.7.

11.3.5.3 Plot met logaritmische schaal

Python bevat drie plotfuncties die gebruik maken van assen met een logaritmische schaal: de functies `semilogx()`, `semilogy()` en `loglog()`. Bij het gebruik van een logaritmische schaal is de afstand tussen opeenvolgende aswaarden, in tegenstelling tot een lineaire schaal, **niet constant**: deze neemt af naarmate men de volgende macht van 10 nadert (zie Figuur 11.17).



Figuur 11.17: Verschil tussen een lineaire en een logaritmische schaal.

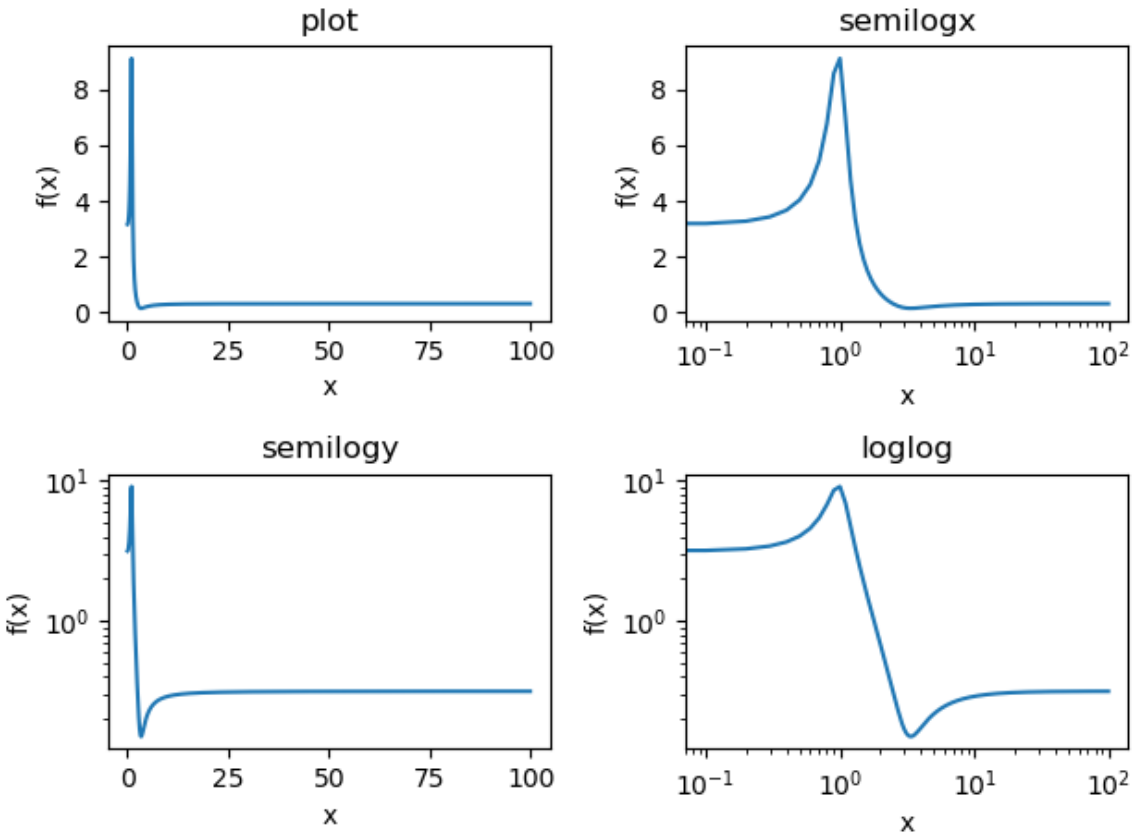
Beschouw bijvoorbeeld de volgende functie f over het interval $[0, 100]$:

$$f : x \rightarrow \sqrt{\frac{10\,(1\ -\ 0.1\,x^2)^2\ +\ 0.2\,x^2}{(1\ -\ x^2)^2\ +\ 0.1\,x^2}}\,. \tag{11.1}$$

De grafiek van deze functie vertoont een maximum bij $x \approx 0.97$ ($f(x) \approx 9.28$) en een minimum bij $x \approx 3.37$ ($f(x) \approx 0.15$). Het script in Fragment 11.16 verdeelt het grafisch venster in 2×2 panelen. In het eerste paneel wordt de functie `plot()`, in het tweede paneel de functie `semilogx()`, in het derde de functie `semilogy()` en in het laatste paneel de functie `loglog()` gebruikt om f te plotten. Het resultaat van de uitvoer van het script wordt weergegeven in Figuur 11.18.

Fragment 11.16: `logplots.py`

```
x = np.arange(0, 100, 0.1)
y = np.sqrt((10*(1 - 0.1*x**2)**2 + 0.2*x**2)/((1 - x**2)**2 + 0.1*x**2))

plt.figure()
plt.subplot(2, 2, 1)
plt.plot(x, y)
plt.title("plot"), plt.xlabel("x"), plt.ylabel("f(x)")

plt.subplot(2, 2, 2)
plt.semilogx(x, y)
plt.title("semilogx"), plt.xlabel("x"), plt.ylabel("f(x)")

plt.subplot(2, 2, 3)
plt.semilogy(x, y)
plt.title("semilogy"), plt.xlabel("x"), plt.ylabel("f(x)")

plt.subplot(2, 2, 4)
plt.loglog(x, y)
plt.title("loglog"), plt.xlabel("x"), plt.ylabel("f(x)")
plt.tight_layout()  # maak subplots passend
```



Figuur 11.18: Gebruik van `semilogx()`, `semilogy()` en `loglog()` om de functie f gedefinieerd in (11.1) te plotten.

Uit Figuur 11.18 blijkt duidelijk dat

- in de eerste subplot de grafiek van f niet goed wordt weergegeven: de punten waar f een maximum en minimum bereikt, zijn moeilijk te bepalen,
- in de tweede subplot is niet duidelijk waar het minimum ligt, en
- in de derde subplot is het moeilijk te bepalen waar het maximum ligt.

Enkel in de laatste subplot, waar beide assen op logaritmische schaal worden weergegeven, zijn alle details van de grafiek van f duidelijk te onderscheiden. Het is dan ook aan te raden om, indien het verloop van de te plotten functie niet gekend is, de grafiek van de functie op een assenstelsel met geen, één of twee logaritmische assen te plotten en vervolgens de beste voorstellingswijze te kiezen.

Opdracht 11.8 (`mijnLogplot.py`)
Gegeven de functie f over het interval $[0.01,\ 2.5]$:

$$f : x \to \frac{e^x}{(x-1)^2}\,.$$

Maak een script `mijnLogplot.py` aan waarin je de grafiek van f plot in een venster dat verdeeld werd in 2×2 subplots met `plot()`, `semilogx()`, `semilogy()` en `loglog()`. Welke plot is te verkiezen?

11.3.5.4 Plot met poolcoördinaten

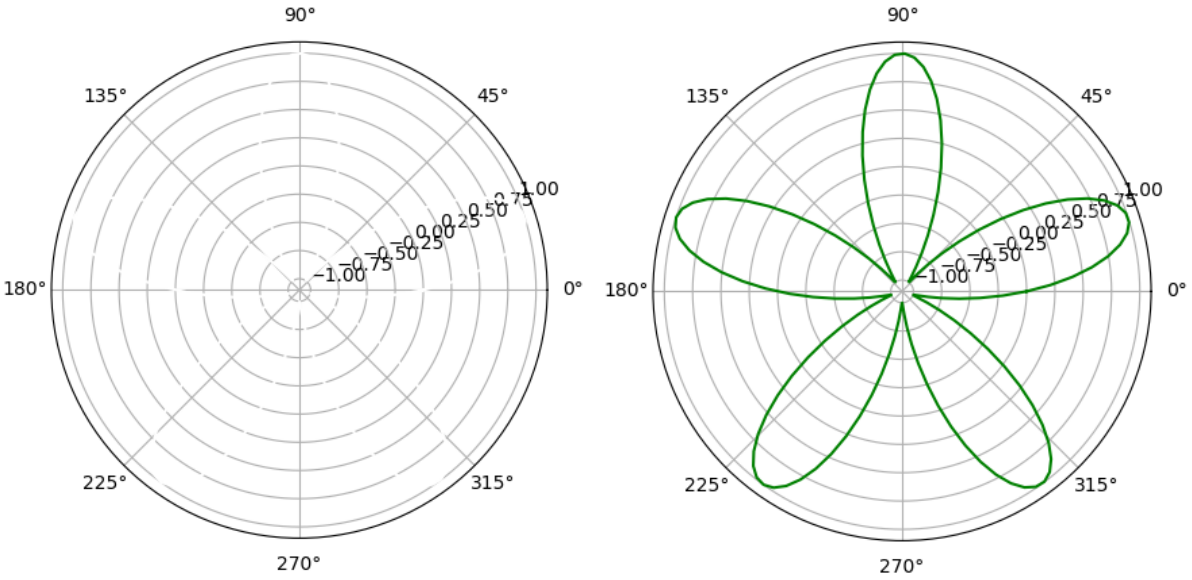
De laatste plotfunctie die we zullen bespreken is de functie `polar()`. Deze functie kan gebruikt worden indien men over de poolvergelijking van een kromme beschikt. De inputs van deze functie zijn de poolcoördinaten r en θ die respectievelijk de afstand van een punt P van de kromme tot de oorsprong en de hoek tussen de positieve X-as en het lijnstuk OP voorstellen. Voor een cirkel is r gelijk aan de straal en $0 \leq \theta \leq 2\,\pi$. Eventuele stijlopties kunnen als derde input meegegeven worden zoals bij de functie `plot()`. Wordt de functie `polar()` opgeroepen, dan wordt eerst een polair grid getekend (zie Figuur 11.19(a)). Vervolgens wordt de te plotten kromme op dat grid getekend.

Een mooi voorbeeld is de Rhodonea kromme. Deze heeft de volgende poolvergelijking:

$$r(\theta) = a\,\sin(k\,\theta)\,,$$

met k een positieve gehele parameter die het aantal *bladeren*[6] van de kromme bepaalt, a een positieve constante die de lengte van de bladeren bepaalt, en $0 \leq \theta \leq 2\,\pi$. Figuur 11.19(b) geeft het resultaat weer voor $a = 1$ en $k = 5$ (zie Fragment 11.17).

[6] Als k even is, zijn er $2k$ bladeren, als k oneven is, zijn er k bladeren.



(a) Het polair grid

(b) De Rhodonea kromme met $a = 1$ en $k = 5$

Figuur 11.19: Gebruik van de functie `polar()`.

Fragment 11.17: `rhodonea.py`

```
a = 1
k = 5
theta = np.linspace(0, 2*np.pi, 150)
r = a*np.sin(k*theta)
plt.figure()
plt.polar(theta, r, "g")
```

Opdracht 11.9 (`hartkromme.py`)

De hartkromme heeft de volgende poolvergelijking:

$$r(\theta) = a - a\,\sin(\theta) + \sin(\theta)\,\frac{\sqrt{|\cos(\theta)|}}{b\ +\ \sin(\theta)}\,,$$

met $a = 2$, $b = 1.4$ en $0 \leq \theta \leq 2\,\pi$. Hierin staat $|\ |$ voor de absolute waarde (functie `abs()`). Maak een script `hartkromme.py` aan waarin je de hartkromme tekent. Experimenteer met verschillende waarden voor a en b.