Dictaat bij studieonderdeel:

Imperatief programmeren

PO, SE

Python x Meet je leefomgeving

2021-2022

H/V5
Periode 1
Informatica
Auteur: Boonstoppel (bns)
Ichthus College Veenendaal
Met dank aan: Nadine van der Heijden & Peter van Capel

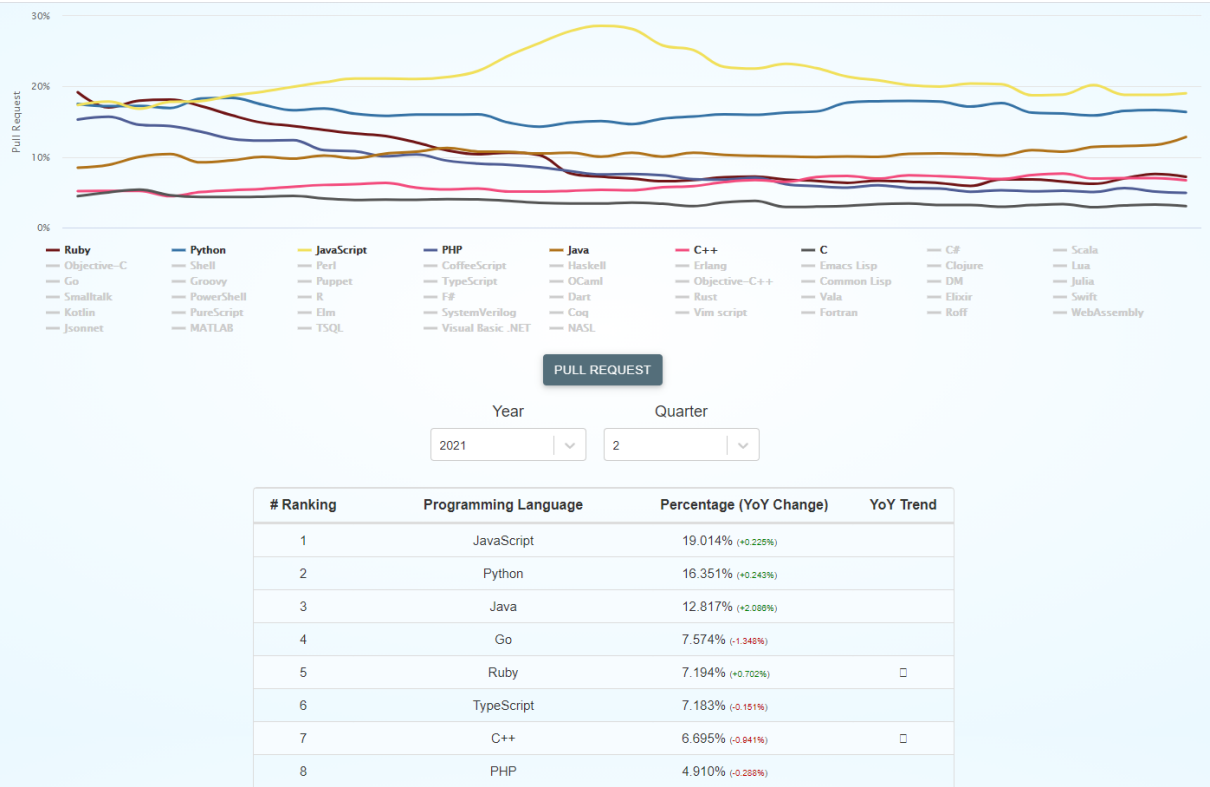
Inhoudsopgave

1 Basisvaardigheden Python

Ons leven bestaat bij de gratie van programmeertalen. Het versimpelt de systemen die we gebruiken, de technieken waarmee we vertrouwd zijn en het feit dat je deze tekst leest bewijst al het nut van programmeertalen.

Een programmeertaal die binnen én buiten de exacte vakken steeds meer gebruikt wordt is Python (www.python.org). Hiervoor zijn een aantal goede redenen:

- Het is gratis en open source. Dit betekent dat iedereen Python kan gebruiken, delen en ontwikkelen.
- Het is eenvoudig om de basis onder de knie te krijgen, zeker in vergelijking met andere populaire talen.
- Het heeft een grote actieve gebruikersgroep. Dit betekent dat er via internet veel hulp, voorbeeldcode, modules en bibliotheken beschikbaar zijn.
- Het is enorm veelzijdig (kan omgaan met code uit 'klassieke' talen als Fortran en C, kan gebruikt worden om apparatuur aan te sturen, ...)



Figuur 1: De meest populaire programmeertalen in de periode 2014-2021. Python is de tweede meest populaire taal, goed voor een zesde van alle (openbare) scripts ter wereld. *Bron: GitHub*

In deze periode komen de meest essentiële eigenschappen van Python aan de orde. Je leert scripts, oftewel korte programma's, te schrijven, eerst met éénvoudige opdrachten (bijvoorbeeld

rekenen, data bewerken, plots maken) en later met combinaties van deze opdrachten tot meervoudige opdrachten. Je kunt je kennis van Python dan zelfs inzetten bij wis-, schei- en natuurkundeverslagen, als de opdracht niet vereist dat je het in Excel o.i.d. moet uitvoeren.

Als je het eenmaal in de vingers hebt, ga je de lol van programmeren vanzelf inzien. Maar zoals je eerst letters moet leren voordat je kunt lezen, moeten we beginnen met de wat taaiere kost waarvan het nut later duidelijk wordt: hoe maak en bewerk je een script, wat zijn datatypes, functies, variabelen, etc.

Tijdens de cursus dien je jezelf ook enkele eigenschappen eigen te maken die een goede programmeur kenmerken. Een verschil met apps zoals je ze op je computer of telefoon draait, is dat er bij (wetenschappelijke) programma's vaak geen gebruikersinterface aanwezig is en dat er zeer beperkte ondersteuning is (redenen hiervoor zijn: geen geld, kleine doelgroep, te ingewikkeld, niet noodzakelijk, programma is een middel en geen doel). De programma's moeten binnen een groep of bedrijf echter wel begrepen, bewerkt en toegepast worden. Dat betekent dat dingen die normaal gesproken onder knoppen of tekstvakken verborgen zitten, in de code zelf aanwezig moet zijn. Het is mogelijk om werkende scripts te schrijven die voor ieder behalve de schrijver niet te gebruiken zijn, maar dit beperkt de bruikbaarheid van jouw harde werk enorm.

Een goede programmeur programmeert daarom niet alleen foutloos, maar ook gestructureerd, modulair (in zelfstandige stukjes) en begrijpelijk. Wat moet hiervoor concreet gebeuren?

- Volg conventies: gebruik variabelen namen die aansluiten bij de inhoud van je programma: een lijst met temperatuurwaarden noem je niet `meuk`.
- Structureer je script: voeg scheidingen toe, definieer *secties* met lege regels en kopjes (zoals je ook in normale tekst doet om hem leesbaar te houden). Breek complexe opdrachten op in meer eenvoudige statements.
- Voeg uitleg toe, vooral wanneer een stuk code zo complex is dat de functie ervan niet direct duidelijk is. Hoe dit kan worden gedaan wordt later in deze cursus behandeld.

Een goede stelregel is: *Hoe moet ik een script schrijven zodat ik het zelf snel weer begrijp als ik het over een jaar open om te gebruiken*?
In de beoordeling van deze cursus is een deel van de punten gereserveerd voor de kwaliteit van de scripts om zo goed programmeren aan te moedigen.

1.1 Python scripts schrijven en uitvoeren in Spyder

Net zoals er meerdere soorten browsers zijn of apps om een route te plannen, zijn er meerdere manieren om Python code te schrijven en uit te voeren. We hebben voor deze cursus gekozen voor het programma Spyder, omdat het gebruikt kan worden voor zowel het schrijven van code als voor het uitvoeren (runnen) en tonen van de resultaten. Andere opties zijn Pycharm of VS Code, maar werken minder intuïtief.

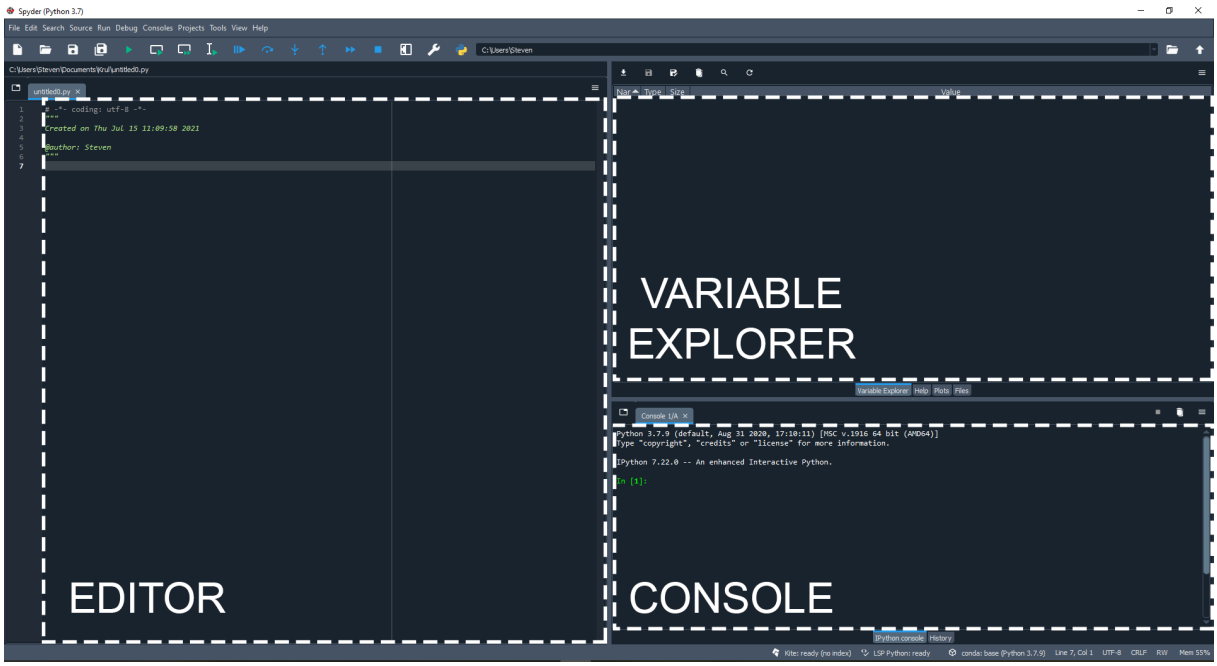
De interface van Spyder wordt weergegeven in Figuur 2. Deze bestaat standaard uit één window met drie schermen, namelijk de Editor, Variable explorer en IPython console. Elk van deze heeft zijn eigen functie:

- Het scherm links is de Editor. Hierin schrijf je de Python code, ook wel script genoemd, die een berekening of taak uitvoert. De editor helpt je om een script te schrijven. Met name

de juiste manier van inspringen bij nieuwe regels (deze *indents* zijn in Python cruciaal) wordt door de editor vrijwel automatisch goed gedaan.

- De Explorer rechtsboven heeft vier tabbladen. Tab 'Files' biedt een makkelijke manier om eerder geschreven scripts op te zoeken en te openen. In de Tab 'Variable Explorer' wordt de waarde van de bestaande variabelen weergegeven nadat je een script hebt uitgevoerd. Je kunt hier alle waarden binnen een variabele zien door die los te openen door er dubbel op te klikken. Alle figuren verschijnen in de Explorer onder de lab 'Plots'. Tot slot geeft de 'Help' tab je een mogelijkheid om snel de eigenschappen van een functie of keyword op te zoeken.

- Het Console rechtsonder is het gedeelte waarin informatie over het runnen (of *draaien* of *evalueren*) van je script verschijnt. Ook de uitvoer (*output*)komt standaard hier terecht.



Figuur 2: De interface van Spyder direct na het opstarten van het programma.

Om een geschreven script te runnen moet het worden opgeslagen in een bestand met de extensie '.py'. Wen jezelf vanaf het begin aan om je verzameling bestanden gestructureerd (in OneDrive!) te houden. Dit betekent dat de naam van de opgeslagen bestanden duidelijk maakt wat het script doet, of waar het onderdeel van uitmaakt. Je doet er goed aan alleen letters, cijfers en eventueel normale (-) of liggende (_) streepjes (en dus geen spaties) te gebruiken. Een voorbeeld is de naam `h1_1-helloworld.py`.

Oefenopdracht: Hello World

Typ de volgende regel tekst in de Editor:

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

De tekst die al in het Editor venster stond mag je weghalen, maar kan ook blijven staan. Hier hoef je je in ieder geval nu niet druk over te maken. De kleur van de diverse woorden/symbolen wordt door Spyder bepaald om zo aan te geven dat ze in Python een bepaalde betekenis hebben. Sla je bestand vervolgens op met de naam `h1_1-helloworld.py` en laat Spyder dit script uitvoeren. Scripts worden in Spyder uitgevoerd door op de groene play-knop (Run file, of de F5-toets) in de balk bovenin te klikken. Als het goed is verschijnt er de eerste keer dat Spyder een script draait nadat het is geopend een scherm waarin bepaalde opties kunnen worden gekozen. Voor nu hoef je daar niets aan te veranderen. De IPython console produceert informatie over welk script wordt gedraaid. Hieronder verschijnt de output van het script. Wat is de functie van het commando print?

Tijdens de cursus zul je regelmatig (voorbeeld)scripts zien in deze bundel. Die zijn ook te vinden op ItsLearning. Sla deze als je ze nodig hebt op in je eigen map met bestanden en open deze vanuit Spyder op dezelfde manier waarop je dat in *Word* zou doen[1].

1.2 Eenvoudige rekenkundige operaties

Veel scripts bevatten simpele rekendingetjes. Hiervoor gebruikt Python de normale tekens: optellen (+), aftrekken (−), vermenigvuldigen (∗), delen (/) en (let op!) machtsverheffen (∗∗). Een andere rekentruc die vaak voorkomt bij programmeren is de *modulus* (%). Dit is eigenlijk de 'rest'-berekening. Bijvoorbeeld bij een digitale klok: als het `19:00` is op de klok dan zeg je: "Het is zeven uur ('s avonds)", want `19 % 12 = 7`. De modulus werkt alleen voor hele getallen.

```
1 + 1 # = 2
2 - 2 # = 0
3*3   # = 9
4/4   # = 1
5**5  # = 3125
```

Tip: Het is onder Python programmeurs gebruikelijk om een spatie te typen voor en na een plus of min-teken, maar niet bij vermenigvuldigen, delen of machtsverheffen. Dit legt nadruk op de rekenregels. Bijvoorbeeld:

```
print(1 + 2 - 3*4**5) # wat komt hieruit?
```

Vraag: Wat is de betekenis van het #-teken? Voer het script uit en je zult er dan wel achter komen.

Vraag: Wat is het nut van `print()`?. Wat gebeurt er als je alleen de som berekent zonder `print()`?

Oefenopdracht: Python als rekenmachine

Bereken de hoogte y van een softbal die vanaf de grond omhoog wordt geslagen met een beginsnelheid van $v_0 = 25\,\mathrm{m/s}$ op $t = 2\,\mathrm{s}$ aan de hand van de formule:

$$y = v_0 t - \frac{1}{2} g t^2 \tag{1}$$

[1] Kopieër **niet** stukjes tekst uit de .pdf file die je nu leest naar het Editor window van Spyder, want dan krijg je vaak verminkte bestanden waaraan je onnodig veel tijd moet besteden om ze te repareren. Typ het over als het kort is, of klik op de hyperlink of ga naar ItsLearning om het hele bestand te vinden.

Kies voor de valversnelling g de waarde $9.81\,\mathrm{m/s^2}$. Maak gebruik van het voorbeeld hiervoor.

1.3 Variabelen

1.3.1 Implementatie

Net als wiskunde 'op papier' is het handig om van variabelen gebruik te maken in plaats van de (numerieke) waarden gelijk in te vullen. Dit is niet anders tijdens programmeren en zorgt voor overzicht vooral als je scripts langer en ingewikkelder worden.

Voorbeeld Wanneer we de uitkomst van een formule als $y(x) = ax^2+bx+c$ voor een willekeurige a, b, c, x willen berekenen, kunnen we de getalswaarden van a, b, c, x in de formule elke keer aanpassen en het Python script meerdere keren uitvoeren voor de verschillende uitkomsten. Maar veel handiger en sneller is de volgende manier, waarbij we zowel de parameters a, b, c, variabele x als de uitkomst $y(x)$ definiëren als variabelen in Python. Let op de volgorde waarin je de variabelen en de formule in je code opschrijft. Een script wordt (best logisch) van boven naar beneden uitgevoerd. Om berekeningen te kunnen doen met variabelen, moeten deze eerst bekend zijn (gedefiniëerd worden). Anders zal je script vastlopen en niet het gewenst resultaat opleveren.

```
a = 1
b = 2
c = 3
x = 1
y = a*x**2 + b*x + c
print(y)
```

Tip: Namen voor variabelen kunnen (bijna) vrij gekozen worden. De enige voorwaarden zijn dat de namen moeten beginnen met een letter, geen spatie of wiskundig teken (`+ * /` etc.) mogen bevatten, en niet gelijk mogen zijn aan woorden die in Python al een betekenis hebben, zoals: **`print`**, **`and`**, **`def`**, **`from`**, **`try`**, etc.[2] Kies voor een variabele altijd een naam die past bij de betekenis ervan in jouw script. Een naam met een ingebouwde betekenis in Python, zoals **`print`**, krijgt altijd automatisch een kleurtje, en dan weet je direct dat je dat maar beter niet kunt gebruiken.

Oefenopdracht: Gebruik van variabelen

Bereken opnieuw de hoogte van de bal uit de vorige opdracht. Doe dit nu voor $t = 0, 1, 2, 3, ...$ en geef een grove schatting hoe lang het duurt voordat de bal weer op de grond is. Nu maak je uiteraard gebruik van variabelen zoals hiervoor beschreven.

1.3.2 Naamgeving en commentaar

Gebruikelijk is om namen voor variabelen te kiezen volgens conventies, maar in ieder geval zodanig dat de betekenis ervan duidelijk is. Voor formules overgenomen uit de theorie is het handig om zoveel mogelijk dezelfde symbolen als in de formules te gebruiken. Toch is het vaak voor een ander moeilijk te achterhalen waar symbolen voor staan als een script zonder kennis

[2] Dus namen van variabelen (en later ook van functies) beginnen met een letter, bevatten alleen A-Z, a-z, 0-9 en _ en zijn hoofdlettergevoelig.

gelezen wordt. Dit is één van de rednene om in een script *commentaar* toe te voegen. Het meest eenvoudig is het om hiervoor de hashtag (#) te gebruiken, want alle tekst op een regel na het hashtag (#) wordt genegeerd bij het uitvoeren van een script.

Het script uit de vorige opdracht kan bijvoorbeeld op de volgende manier verduidelijkt worden:

```
# abc-formule voor een parabool
a = 1 # parameter 1, zie beschrijving in reader
b = 2 # parameter 2
c = 3 # parameter 3
x = 1 # variabele
y = a*x**2 + b*x + c # resultaat bij gegeven waarde van x
print(y)
```

Als je in Spyder een nieuw Python script aanmaakt zie je direct nog een andere manier om commentaar te schrijven. Alle tekst tussen """ en de volgende """ is commentaar; deze manier van commentaar aangeven is handig voor grotere stukken tekst.

1.3.3 Datatypen

In alle bovenstaande opdrachten zijn de variabelen getallen. Soms wordt een variabele ook wel een parameter genoemd. In Python zijn variabelen echter veel breder inzetbaar en eigenlijk *objecten* die allerlei soorten informatie kunnen bevatten. Objecten kunnen verwijzen naar enkele getallen, naar lijsten van getallen, naar woorden, naar figuren, naar algoritmes, etc.

Vaak vereisen handelingen dat objecten van een bepaald type zijn om een Python script uit te voeren. De meest gebruikte types komen gedurende deze cursus aan bod.

int Gehele getallen: `3 300 20`

In Python (en andere programmeertalen) is **int** het objecttype dat gebruikt wordt voor gehele getallen. De meest eenvoudige manier om variabelen te definiëren van dit type is door een geheel getal toe te kennen zoals in bovenstaand voorbeeld: `a = 1`. Er kan ook expliciet worden aangegeven dat een variabele als een integer moet aangemerkt door `a = int(x)`. Hierbij kan `x` ook een decimaal getal zijn, maar bij de toekenning worden de getallen achter de komma verwijderd. Let op: dit is niet hetzelfde als afronden op gehele getallen! `int(3.9) = 3` en niet een afgeronde 4.

float Getallen met decimalen: `2.3 4.62 100.00`

Het objecttype voor decimale getallen is **float**. Floats kunnen worden gedefinieerd door een getal toe te kennen met een decimale punt, zoals: `a = 1.5`. Om expliciet aan te geven dat een object `a` een float is, kan gebruik worden gemaakt van `a = float(x)`. De waarde van `x`, wordt omgezet in een float. Door een decimale punt te plaatsen na het getal: `a = 1.` kan je ook meteen aangeven dat een variabele met een gehele waarde toch een float is. `float(1) = 1.00000000` (standaard 8 decimalen nauwkeurig).

bool Logische waarde: `True` of `False`

Een **boolean** object, dat we later in het hoofdstuk zullen tegenkomen, is een binair object dat twee waarden kan hebben, namelijk `True` of `False`. `True` is gelijk aan 1, en `False` is gelijk aan 0. `bool(1)` is dus gelijk aan `True`.

str Geordende reeks karakters (tekst): `'hello''Sam'"2019"`

Een **string** is het objecttype in Python voor tekst. Deze heb je ongemerkt gebruikt in de `'Hello world'`!, met **`print()`**. Om in Python een `string` *object* te maken gebruik je namelijk aanhalingstekens, zowel enkele als dubbele apostrofs mogen gebruikt worden. Zowel `a = 'tekst'` als `a = "tekst"` zijn dus toegestaan.

list Geordende reeks objecten: `[10, "hello",200.3]`

Een **`list`** kan een handig object type zijn om data in bij te houden; doordat het een geordende reeks is kun je met een index een specifiek element uit de reeks opvragen en evt. aanpassen.

Je zult sommige object typen vaker gebruiken dan andere, maar het is goed om te weten dat ze bestaan. Let op de verschillende soorten haakjes en tekens. Met **`type(naam_variabele)`** kun je achterhalen met welk type je te maken hebt. Dit staat ook in de variable explorer.

1.4 Functies

Je script kun je automatiseren door variabelen en de handelingen die je vaak herhaalt, in een *functie* te zetten (een functie te definiëren).

1.4.1 Eigengemaakte functies

Een functie is een Python *object* dat aan de hand van invoervariabelen één of meerdere handelingen kan uitvoeren. Een Python functie is dus veel algemener dan een wiskundige functie zoals *sinus*.

Functies worden vaak gebruikt om te voorkomen dat blokken code meerdere keren in een script herhaald worden. Hiermee zorg je ervoor dat, wanneer er een aanpassing gedaan moet worden, dit maar op één plek hoeft te gebeuren. Ook blijft je script overzichtelijk, ook al maak je hem steeds langer of complexer. Python heeft best veel standaard *functies*, waarvan we er al één hebben gebruikt, namelijk **`print()`**.

We gebruiken de formule $y(x) = ax^2 + bx + c$ om de werking van *functies* uit te leggen. In het script hieronder zie je een voorbeeld van een functie die de waarde van de formule berekent en de uitkomst weergeeft op het scherm met behulp van de functie **`print()`**.

```
def formule(a, b, c, x):
    y = a*x**2 + b*x + c          # bereken de formule
    return y                              # geef de waarde terug

i = 1
j = 2
k = 3
z = 1
waarde = formule(i, j, k, z) # geef de variabelen mee op volgorde
print(waarde)
```

De manier om in Python aan te geven dat er een *functie* moet worden gedefinieerd is door gebruik te maken van het woord **`def`** (afkorting van 'define' of 'definition'), gevolgd door de

naam van de *functie*. Daarna volgen tussen ronde(!) haakjes de variabelen die door de functie moeten worden gebruikt als input. De regel wordt afgesloten met een dubbele punt om aan te geven dat wat er op de volgende regels staat de echte definitie is.

Wat volgt is dus de code van de *functie* zelf. Merk op dat de regels code na het begin van de **def** niet helemaal links beginnen. Dit is belangrijk! De inspringing of *indent* is onderdeel van de syntax van Python. Dit laat Python weten dat de tekst op die regel onderdeel is van de *functie*. Zorg er dus voor dat alle code van de functie dezelfde indent heeft. Dit doe je door steeds een zelfde aantal spaties of tabs in te voeren aan het begin van de regel.[3] De Spyder editor helpt je hierbij enorm, want als je op Enter drukt komt deze indent er automatisch. Al moet je altijd even opletten.

Wanneer de functie definitie klaar is - meestal is dit na een `return` - en je verder wilt gaan met de rest van een script, laat je de indent weg. Dit laat Python weten dat die regel niet bij de functie hoort. Ook hier helpt Spyder je weer, want die laat automatisch de indent weg na een regel met **return**.

Om een functie te gebruiken (ook wel aanroepen) typ je de naam van de functie met de waarden die je als input voor de functie wilt gebruiken. Hierbij hoeven die variabelen niet dezelfde naam te hebben als de variabelen die je gebruikt hebt in de definitie van de functie. Het is de *volgorde* van de variabelen die Python vertelt welke waarde waar moet worden gebruikt. In bovenstaand voorbeeld krijgt parameter a de waarde 1, immers `i = 1`. Parameter b krijgt de waarde 2, want `j = 2`. Na het overtypen (!) van en het uitvoeren van bovenstaand script is het een goed idee om eens te kijken in de **Variable explorer** in Spyder. Hier kun je zien welke variabelen er bestaan, wat het type is en welke waarde ze hebben. De variabele `waarde` heeft als het goed is de waarde 6. Dit getal moet ook afgedrukt zijn in de IPython console vanwege de uitgevoerde `print`.

Een andere manier om waarden toe te kennen aan de variabelen binnen een functie is door duidelijk te maken welke variabele welke waarde krijgt. Dan maakt het niet meer uit in welke volgorde de variabelen een waarde krijgen toegekend. Bijv.

```
# we geven de variabelen mee op een andere volgorde
waarde = formule(x=1, a=3, b=2, c=5)
print('waarde = ', waarde)
```

Je moet in dit geval niet alleen weten dat de functie `formule` vier argumenten verwacht, maar ook wat de naam van die argumenten binnen de functiedefinitie is. Samenvattend:

- Je geeft de argumenten op als een getal: de waarde van a als eerste, etc. Bij deze methode van aanroep maak je gebruik van *positie of volgorde*.
- Ofwel, je specificeert de waarden die a, b, c en x binnen de functie krijgen en dan mag je elke volgorde aanhouden. Bij deze methode maak je gebruik van *namen*.

Oefenopdracht: Controleren van bovenstaande functie

Ga expliciet na dat `formule(3, 2, 5, 1)` en `formule(x=1, a=3, b=2, c=5)` dezelfde waarde opleveren. Welke? En dat een `formule(1, 3, 2, 5)` een andere waarde oplevert. Het is een goede

[3] Het is gebruikelijk om een *indent* te gebruiken van 4 spaties.

zaak om resultaten die makkelijk zelf na te rekenen zijn ook echt te controleren! Alleen op die manier kun je vertrouwen krijgen in de correcte werking van zelf gemaakte scripts.

Vaak weet je van een functie helemaal niet wat de naam van een argument is, en bereken je een functie door elk argument te voorzien van een waarde. Dat werkt prima als het aantal argumenten beperkt blijft. Veel functies die in door professionals geschreven *modules* bestaan maken wel degelijk gebruik van deze namen. Het valt dan meestal helemaal niet op, omdat je denkt dat het nu eenmaal zo hoort als je iets wilt doen met Python. Meer hierover later in de cursus. Vanaf hoofdstuk 2 zul je min of meer ongemerkt ook al deze methode gebruiken.

1.4.2 Lokale vs. globale variabelen

Het bovenstaande voorbeeld toont één waarde op het scherm. Kijk je in het scherm van de **Variable explorer**, dan zie je dat alleen de variabelen `i`, `j`, `k`, `z` en `waarde` een waarde hebben, maar de `a`, `b`, `c`, `x` en `y` komen daarin niet voor. De eerste variabelen zijn globaal (in het hele script) bekend en hebben de getoonde waarde. De andere variabelen zijn alleen bekend binnen de functie `formule` en heten lokale variabelen.

Waarden van lokale variabelen kun je hooguit weergeven door ze te printen binnen een functie, maar dat wordt al snel irritant als zo'n functie duizend keer wordt aangeroepen.

De manier om lokale variabelen binnen een functie daarbuiten bekend te maken is het gebruik van **`return`**. Hiermee geef je aan dat de waarde van een variabele moet worden 'teruggegeven' aan de omgeving die de functie aanroept. In het voorbeeld hieronder geeft de functie `formule` de waarde van `y` terug. In het script stoppen we deze waarde in de variabele `waarde`, die daarna afgedrukt wordt.

```
def formule(a,b,c,x):
    y = a*x**2 + b*x + c
    return y

i = 1
j = 2
k = 3
z = 1

waarde = formule(i,j,k,z)
print(waarde)
```

Wat er binnen een functie gebeurt heeft normaal gesproken dus geen invloed op de rest van je script. Dit is een reden om onderdelen van een groter project op te bouwen met gebruik van functies.

Waarden van globale variabelen kun je wel gebruiken binnen een functie. Je hoeft daarvoor niets speciaals te doen. Maar je mag niet binnen een functie de waarde van een globale variabele veranderen.

Bij het gebruik van veel functies binnen een uitgebreid script is het al snel lastig om unieke korte namen voor variabelen binnen een functie te bedenken en kun je per ongeluk (of expres) meerdere keren dezelfde naam gebruiken. Dit zou er voor kunnen zorgen dat je script verkeerde resultaten oplevert. Noem je variabelen dus niet altijd zomaar *a, b* en *c*, maar geef ze een nuttigere naam waar mogelijk.

1.5 Loops

Binnen Python bestaan manieren om functies of handelingen automatisch en/of heel vaak uit te voeren. We behandelen hier de constructies **while** en **if**.

1.5.1 While

Een zogenaamde **while** loop kan gebruikt worden om een bepaald stuk code te herhalen zolang er aan een voorwaarde voldaan wordt. Voor het voorbeeld van de omhoog geslagen bal dat we eerder gebruikt hebben kunnen we de hoogte van de bal berekenen voor de eerste T seconden. Hoe zo'n loop kan worden geïmplementeerd is te zien in onderstaand voorbeeld.

```
v0 = 25   # beginsnelheid
g = 9.81  # gravitatieconstante
t = 0.    # begintijd
dt = 0.25 # tijdstap

while t <= 5:
    y = v0*t - 0.5*g*t**2
    print(t, y)

    t = t + dt
```

Net als eerder definiëren we de benodigde variabelen v_0 en g. Daarnaast maken we een variabele aan voor de tijd t (die een steeds andere waarde zal krijgen maar in het begin de waarde 0 heeft) en kiezen we de grootte van de tijdstappen dt die we willen maken voor t. Zolang (*while*) $t \leq 5$ (we nemen aan dat dit secondes zijn) wordt de hoogte y berekend en worden zowel t als y getoond op het scherm.

Zoals te zien in het voorbeeld kun je dit in Python aangeven door het commando **while** gevolgd door de conditie (`t <= 5`) waaraan moet worden voldaan, afgesloten met een dubbele punt. Zolang $t \leq 5$ wordt aan de conditie voldaan en is het resultaat van de conditie de boolean waarde **True** en de *body*[4] van de **while** wordt dan uitgevoerd.

Om door te gaan naar de volgende tijdstap vernieuwen we de waarde van t door hier dt bij op te tellen. Hiermee wordt doorgegaan totdat de voorwaarde `t <= 5` niet meer klopt. Net als bij het gebruik van **def** moeten de regels code binnen de **while** *loop* beginnen met een *indent* om aan te geven dat die regels onderdeel zijn van één en dezelfde *loop*.

Een veel voorkomende fout bij een **while** is dat de conditie waaraan voldaan moet worden altijd **True** blijft, zodat het script eeuwig doorgaat. Dit gebeurt bijv. als je de regel `t = t + dt` vergeet, of verkeerd invoert (bijv. `t = t - dt`), of als je de regel wel goed overtypt maar de indent vergeet!

Je kunt een script stoppen in Spyder door op de rode stop rechts boven in de Console te drukken, of door `CTRL + C` op je toetsenbord te gebruiken. Als je een script schrijft wat mogelijk lang duurt en je weet niet zeker of het vastgelopen is of dat de computer lang aan het rekenen is kun je zorgen dat je script af en toe iets in de Console print. Als je dan af en toe een nieuwe output ziet verschijnen weet je dat het script nog bezig is. Dit is ook handig om te doen tijdens

[4] De *body* is dus het stuk wat een vast aantal spaties inspringt.

het testen van het script; dan kun je zien tot waar het script normaal functioneert, en of het bijvoorbeeld wel of niet aan een bepaalde plek toe komt.

1.5.2 If

Een tweede soort constructie is de zogenaamde *if - else* constructie. Aan de hand van een voorwaarde (*conditie*) wordt een bepaald deel code wel of niet uitgevoerd. Een variant hierop is de *if - else if - else* constructie, waarbij er meer dan 2 mogelijkheden voorkomen. In woorden ziet dat er zo uit:

$$\text{Je bekijkt het weer : } \begin{cases} \text{als het regent,} & \text{dan paraplu opsteken} \\ \text{als het sneeuwt,} & \text{dan sneeuwballengevecht} \\ \text{als het hagelt,} & \text{dan binnen blijven} \\ \text{anders,} & \text{dan zonnen op het strand} \end{cases} \tag{2}$$

De vertaling van deze functie in Python kan goed worden gedaan door gebruik te maken van de *if - else if* constructie. Merk op dat er hier ook gebruik moet worden gemaakt van *indents*. In onderstaand voorbeeld staat deze constructie binnen de definitie van een *functie* en heeft daarom al een *indent*. Daarnaast moet de code die wordt uitgevoerd wanneer aan een conditie wordt voldaan ook een *indent* hebben, net als bij de **while**. In dit geval is er dus een dubbele *indent* nodig. In Python wordt *else if* afgekort tot **elif**.

```
if regen:
    # paraplu meenemen
elif sneeuw:
    # sneeuwballengevect
elif hagel:
    # binnen blijven
else:
    # zonnen op het strand
```

Uiteraard zul je deze constructie vaker gebruiken voor getallen: bijvoorbeeld bij een BMI-berekening. Als iemand minder dan X aantal kilo weegt, print je 'ondergewicht', tussen X en Y print je 'prima gewicht' en boven Y kilo print je 'overgewicht'.

Oefenopdracht: Gebruik van `if` en/of `while`

- Schrijf een eenvoudig Python-script dat om input vraagt. Gebruik hiertoe: [5]

  ```
  input("Geef een positief getal: ")
  ```

  Als het opgegeven getal inderdaad positief is wordt de wortel[6] van dat getal geprint en wordt er om nog een getal gevraagd. Maak dus gebruik van een loop! Is het opgegeven getal negatief, dan wordt de wortel niet uitgerekend. Je print dan de tekst

[5] De functie **input()** geeft als resultaat een **str** (datatype string). Je kunt met **float()** of **int** een **float** of **int** van de input maken.

[6] Weet je niet hoe je de wortel van een getal moet uitrekenen in Python? Zoek het dan op! Niet in deze reader, maar op internet.

`"Dombo: Positief svp!"` en vervolgens vraagt het script (heel beleefd) weer om een positief getal. Pas als je de waarde 0 opgeeft stopt de loop en dus het script.

1.5.3 Boolean expressions

In de voorbeelden van loops hierboven hebben we voorwaarden gebruikt om delen van scripts wel of niet uit te voeren. De algemene term voor deze voorwaarde is *Boolean expression*. De uitkomst van zo'n conditie heeft twee mogelijkheden, namelijk **`True`** of **`False`**. Vaak wordt zo'n voorwaarde ook wel een test genoemd.

```
a == b  # test of a gelijk is aan b
a != b  # test of a niet gelijk is aan b
a > b   # test of a groter is dan b
a >= b  # test of a groter dan of gelijk is aan b
a < b   # test of a kleiner is dan b
a <= b  # test of a kleiner dan of gelijk is aan b
```

De uitkomst van een Boolean expression kan worden omgedraaid door **`not`** toe te voegen voor de geteste voorwaarde. Daarnaast kunnen Boolean expressions ook worden gecombineerd door het gebruik van **`and`** waarbij de formule alleen **`True`** zal opleveren als aan beide voorwaarden wordt voldaan; of **`or`** waarbij de formule **`True`** zal opleveren als aan minstens één van de twee condities wordt voldaan (beide mag dus ook).

```
not a == b
a > b and c < d
a > b or c < d
```

Oefenopdracht: Testen van combinaties van Boolean expressions

- Laat zien dat de tekst 'abc' kleiner is dan de tekst 'defg'. Met andere woorden: teksten kun je op een alfabetische manier ordenen.
- Maak vervolgens je eigen 'exclusive or' functie 'xor' die twee argumenten verwacht en die als resultaat **`True`** geeft als één van de argumenten True is, en de ander False. In alle andere gevallen is de functiewaarde False. Test je functie door alle mogelijke combinaties uit te proberen. (Dat zijn er maar 4.)

1.5.4 For

Als je een operatie een exact aantal keer wilt uitvoeren kun je dit met een **`while`**- of **`if`**-loop doen waarin je een teller zet. Bijvoorbeeld:

```
i=0
while i < 5:
    print(i)
    i=i+1
```

Een andere manier om dit te doen is met een **for**-loop. Hierbij zit de 'teller' automatisch ingebouwd: de loop wordt herhaald voor elk element in de voorwaarde. Elke **while**-loop is om te schrijven in een **for**-loop, maar soms *voelt* de één logischer, en is die ook makkelijker. Over het algemeen gebruik je een **for**-loop wanneer je van te voren weet hoe vaak de operatie herhaald moet worden en een **while**-loop wanneer het aantal herhalingen af hangt van een conditie.

Bovenstaande **while**-loop is om te schrijven in de volgende **for**-loop:

```
for k in range(5):
    print(k)
```

Oefenopdracht: for k in range()

Voer het voorgaande stukje code uit.

Vraag: welke getallen zitten er verborgen in `range(5)`? En wat als je er `range(10)` van maakt?

In hoofdstuk 4.2 komen **for**-loops en hun toepassingen nog uitgebreider aan bod, dus wees niet bang als dit je nu een klein beetje te moeilijk lijkt.

1.6 Slotopdrachten

1.6.1 Inleiding

Voor alle opdrachten geldt dat je wordt aangemoedigd om op het internet te zoeken in de enorme hoeveelheid voorbeelden, tutorials, vragen en antwoorden over het gebruik van Python. Maak daar vooral gebruik van en leer zo je weg te vinden in het Python doolhof. Voor alle opdrachten geldt ook: maak je eigen script, sla dat op met het nodige commentaar zodat je het later misschien kunt hergebruiken of je medestudenten er blij mee kunt maken.

1.6.2 Opdracht 1: Omrekenen van Fahrenheit naar Celsius

Schrijf een script waarin een reeks van verschillende temperaturen in Fahrenheit gebruikt wordt als invoer om te bepalen of het vriest of niet. Laat voor elke stap de temperatuur in Fahrenheit zien samen met de conclusie of het wel of niet vriest. Test waarden van -10 t/m 100 Fahrenheit in stappen van 5 Fahrenheit. De conversie tussen Celsius en Fahrenheit is

$$F = \frac{9}{5}C + 32. \tag{3}$$

1.6.3 Opdracht 2: Priemgetallen

Schrijf een script dat de priemgetallen binnen een bepaald interval (bijv. alle gehele getallen van 1 tot en met 100) berekent en toont op het beeldscherm. Let op: het getal 1 is alleen deelbaar door 1 en zichzelf, maar is toch geen priemgetal. Dus 2 is het eerste priemgetal. *Tip:* Gebruik de modulus (`%`)!

Doe dit in eerste instantie echt helemaal zelf! Pas als het niet lukt, maar ook als het wel lukt, zoek je daarna op internet naar Python scripts die bruikbaar zijn in deze opdracht.

2 Modules (1): NumPy

In dit hoofdstuk kijken we naar een essentieel onderdeel van het programmeren in Python: het gebruik van *modules*. Modules bevatten functionaliteit die niet in Python zelf zit maar apart toegevoegd moet worden. Oorspronkelijk kon je met Python niet veel meer dan strings (stukken tekst) bewerken. De twee modules `numpy` en `matplotlib` zijn cruciaal voor het doen van databewerking in de meer algemene zin. In dit hoofdstuk behandelen we `numpy`, in het volgende `matplotlib`.

Numpy, een afkorting voor *numeric python*, is zoals de naam een beetje verraadt, erg goed in numerieke berekeningen, oftewel alles wat met alleen getallen te maken heeft. Het is snel, heeft handige functies en vooral héél véél functies. Teveel om allemaal in dit hoofdstuk te behandelen; we kijken hier vooral naar de dingen die relevant zijn voor deze periode. Maar als je meer wilt, kun je altijd terecht op Google. Het gebruik van andere functies dan in het dictaat staan is zeker toegestaan en zelfs wenselijk: als je zelf iets wil maken later in Python is er geen dictaat om jou te helpen, maar Google is er wel.

Het importeren van een module gebeurt (gewoonlijk op de eerste regel(s) van je script) met het commando **`import`**. Het is gebruikelijk om `numpy` af te korten tot `np`; dat typt sneller. Dat ziet er in de praktijk zo uit:

```
import numpy as np
```

Op deze manier wordt `numpy` geïmporteerd onder de gebruikelijke afkorting `np`. Vanaf dit punt wordt naar `numpy` verwezen als `np`. Om functies uit een module te gebruiken wordt de afkorting van de module als prefix (voorvoegsel) gebruikt, gevolgd door een punt, en dan het commando(= de naam van de functie) uit de module.
Bijvoorbeeld: `np.mean(X)` `#berekent het gemiddelde van X`.

2.1 Arrays maken

Een *array* (Nederlands: reeks) is een verzameling van getallen. In Python kun je die maken als een lijst (zie hoofdstuk 1.3.3), maar een lijst is in de praktijk heel erg onhandig. In plaats van een lijst gebruiken we veel liever een numpy array. Hier kan maar één datatype tegelijkertijd in voorkomen: `ints` of `floats`[7].

De duidelijkste manier om een array te maken is met de functie `np.array()`. Als je de getallen van de dobbelsteen in een array wilt zetten, doe je dat bijvoorbeeld zo:

```
getallen_array = np.array([1,2,3,4,5,6])
# let op de extra blokhaken [] tussen de ronde haken ()
```

Je kunt ook een Python list omzetten in een Numpy array:

```
getallen_lijst = [1,2,3,4,5,6]
getallen_array = np.array(getallen_lijst)
```

[7] Het is mogelijk om verschillende datatypes in één numpy-array te zetten, maar dat is in de praktijk nog vervelender dan een Python list.

Kijk eens goed: als je de lijst van getallen invult in `np.array()`, zie je ook direct dat de blokhaken uit het eerste voorbeeld 'nodig' zijn. Stiekem vul je dus een Python list in in de functie `np.array()`.

Handmatig alle getallen invullen is iets waar je als programmeur niet op zit te wachten. Als het ook maar een klein beetje kan, doen we dat het liefst automatisch. In het codeblok hieronder staan een aantal manieren om arrays aan te maken die ook werken als je nog helemaal geen lijst hebt.

Code 2.1: code-inc/w2/np˙vb1.py

```
# De volgende voorbeelden leveren allemaal arrays op met ints:
array1a  = np.array([0,1,2,3,4,5]) # Vanuit een Python-lijst
array1b  = np.arange(6)            # Een bereik, beginnend bij 0
# Let op: het resultaat van beide arrays is hetzelfde, maar de tweede
# functie kost minder typwerk, zeker bij een groter bereik.

## De volgende voorbeelden leveren allemaal arrays met floats op:
nullen   = np.zeros(6)             # Array gevuld met 6 nullen (float)
enen     = np.ones(6)              # Array gevuld met 6 enen   (float)
random   = np.random.rand(6)       # Array gevuld met 6 random getallen
                                   # tussen de 0 en 1

# Om arrays met hele getallen te maken ...
# moet je dat specifiek aangeven met 'dtype=int'
nullen_i = np.zeros(6,dtype=int)   # Array gevuld met 6 nullen (int)
enen_i   = np.ones(6,dtype=int)    # Array gevuld met 6 enen   (int)

# Om een array te maken met allemaal stapjes tussen twee waarden (grid)
# kunnen deze functies gebruikt worden:
grid1 = np.arange(0.,5.,0.25)      # 0 TOT 5 met stapjes van 0.25
grid2 = np.linspace(0.,5.,num=20)  # 0 TOT EN MET 5, 20 gelijke afstanden
```

Merk op dat `np.arange()` het eindgetal (`5.`) niet meeneemt, `np.linspace()` doet dit wel, dus `grid1` en `grid2` zijn verschillend.

Oefenopdracht: Maken en bewerken van numpy arrays

a) Schrijf een script waarmee een array wordt gemaakt met daarin de getallen 1.5 tot en met en 3.0 met stappen van 0.3. Doe dit met zowel `np.arange` als `np.linspace`. Controleer je resultaten met `print()`.

Tip: Begin met `xmin = 1.5`, `xmax = 3.0` en `dx = 0.3`. Werk daarna alleen met deze variabelen, of andere hulpvariabelen die volgen uit deze 3 variablen.

2.2 Rekenen met arrays

Rekenen met arrays is super makkelijk. Wanneer twee arrays dezelfde vorm (evenveel getallen) hebben, kun je ze getal voor getal optellen door + te gebruiken. Wanneer deze niet dezelfde

vorm hebben krijg je een foutmelding. Alle wiskundige formules (+,−,/,∗,∗∗) werken op deze manier.

Neem het volgende voorbeeld maar over en zie wat er gebeurt:

```
a = np.array([1,2,3,4,5])
b = np.array([6,7,8,9,10])
c = a + b
print("c: ", c)
d = a * b
print("d: ", d)
```

Oefenopdracht: Maken en bewerken van numpy arrays

b) Schrijf een script dat een array maakt met de gehele getallen tot en met 10, beginnend bij 1. Natuurlijk geef je dat array een naam. Bereken met behulp van dit array k^k voor $1 \leq k \leq 10$ en laat het resultaat op het scherm laat zien met het `print()` commando. Bekijk en controleer de resultaten. Valt je iets op?[8]

2.3 Indexing en slicing

Soms is het fijn om delen van een array een aparte naam te geven. Of je wilt een specifiek getal uit een getallenreeks halen. In het codeblok hieronder is te zien hoe je een enkel element uit een array kunt selecteren (*indexing* of een groter deel van de array *slicing*). Door gebruik te maken van *slicing* bekijk je de *oorspronkelijke* data van de array. Wijzig je de data in de slice, dan wijzig je dus ook de data in het oorspronkelijke array!

Code 2.2: code-inc/w2/np˙vb2.py

```
# We maken een test array om als voorbeeld te gebruiken.
test_array = np.arange(1,11)     # Ziet eruit als: [ 1 2 3 4 5 6 7 8 9 10 ]

# Wanneer we een enkel element willen selecteren kan dat als volgt:
element1 = test_array[0]         # Het eerste element heeft index '0'.
element7 = test_array[6]         # Dit heeft dus als resultaat '7'

# Wanneer we een deelverzameling willen selecteren
deel_array1  = test_array[2:5]    # Selecteer element 2 TOT 5 (t/m 4)
deel_array2  = test_array[2:-2]   # Selecteer van 2e t/m 2 na laatste
deel_array3  = test_array[:-2]    # Vanaf begin t/m twee na laatste
deel_array4  = test_array[2:]     # Vanaf tweede

# Het is mogelijk om elk tweede of derde element te selecteren
oneven_arr  = test_array[::2]     # Selecteer elk 2e element, vanaf begin
```

[8] Als je dit met `np.arange()` gedaan hebt worden de elementen in je array opgeslagen als 32-bit integers. Dat betekent dat er 32 éénen of nullen worden gebruikt om het getal binair op te slaan. Het hoogste getal wat in 32 bits past is $2^{32} - 1$, en dat is kleiner dan 10^{10}. Om ook het laatste element in deze reeks goed te krijgen moet je dus meer bits gebruiken; je kunt dit doen door `dtype=np.int64` toe te voegen bij het maken van je numpy array.

```
even_arr    = test_array[1::2]    # Selecteer elk 2e element, vanaf 2e
# Let op de naamgeving: de getallen in even_arr zijn even

# Een kopie maken is speciaal:
kopie_arr = test_array.copy()     # Twee verschillende arrays
same_arr  = test_array            # Twee namen voor hetzelfde array
```

Let dus goed op het verschil tussen de laatste twee handelingen: `kopie_arr` is een kopie van de waarden, terwijl `same_arr` alleen een andere naam is voor `test_array`!

Oefenopdracht: Maken en bewerken van numpy arrays

c) Vraag: wat is de waarde van `test_array[1]`? Let goed op!

Een computer begint bij nul te tellen. Dat betekent dat het eerste getal, wat bij ons altijd plek 1 krijgt (bijvoorbeeld vers 1 van een hoofdstuk in de Bijbel of een psalm), voor de computer op plek 0 staat. En het vijfde getal in een reeks, staat voor de computer op plek 4.

d) Neem de laatste regel (en het bijbehorende `test_array`) over van bovenstaand voorbeeld. Print eerst de laatste getallen van beide arrays (`test_array` en `same_arr`). Voeg vervolgens daaronder deze regel toe: `test_array *= 10`. Daarmee vermenigvuldig je het `test_array` met 10. Print weer beide getallen. Zet daaronder deze regel: `test_array = test_array / 5`, en print weer beide getallen. Wat is het verschil? Vraag het als je het niet snapt!

2.4 Allerlei functies

Numpy arrays hebben een aantal ingebouwde functies, om bijvoorbeeld het gemiddelde of de som uit te rekenen. Ook de lengte of de vorm en datatype worden verkregen door `.functie` aan de naam van zo'n array vast te plakken.

Code 2.3: code-inc/w2/np˙vb3.py

```
# Gebruik een testarray
test_array = np.arange(1,11)      # Ziet eruit als: [ 1 2 3 4 5 6 7 8 9 10 ]

# Berekening van minimum, etc.; let op de verplichte () aan het eind
min_arr = test_array.min()        # Geeft minimum van array (1)
max_arr = test_array.max()        # Geeft maximum van array (10)
sum_arr = test_array.sum()        # Geeft som van array (55)
avg_arr = test_array.mean()       # Geeft gemiddelde van array (5.5)

# Eigenschappen van een array; let op het ontbreken van () aan het eind
ta_lengte = test_array.size       # Geeft aantal getallen in array (10,)
ta_dtype  = test_array.dtype      # Geeft datatype in array (int32)

# Deze manier van aanroepen "functie(argument)" is ook toegestaan
min_arr = np.min(test_array)      # Geeft minimum van array (1)
```

Let op de laatste regel: dit is eigenlijk iets meer werk, maar over het algemeen veel veiliger! Als je bijvoorbeeld de mediaan van een array uit wilt rekenen, kun je *niet* gebruik maken van `test_array.median()`, maar moet je de andere methode gebruiken: `np.median(test_array)`.

In sommige gevallen wil je een lijst getallen ook filteren. Je bent bijvoorbeeld geïnteresseerd in alle leeftijden boven de 18 jaar, of alle mensen in de bovenbouw (leerjaar 4, 5 of 6). Het is hiervoor mogelijk om bij het ophalen van elementen een voorwaarde op te geven, aan de hand van een *masker*. Vervolgens kun je met zo'n *masker* specifieke elementen bijvoorbeeld wijzigen of weggooien. De werking van zo'n masker wordt uitgelegd aan de hand van het onderstaande voorbeeld[9]. Alle plekken in de array die voldoen aan die voorwaarde, geven de waarde **True**, alle anderen geven **False**. Een masker pas je vervolgens toe met de blokhaken `[ ]`.

Code 2.4: code-inc/w2/np˙vb4.py

```
# Gebruik van een test_array
test_array = np.array([12, 1, 7, 8, 4, 3])
print(" test_array (origineel) = ", test_array)

# Aanmaken van een masker met de voorwaarde dat elementen > 5 moeten zijn
masker = test_array > 5      # True als element > 5, anders False
print(" masker =                 ", masker)

# Selecteren van elementen waarvoor het masker True is
g5 = test_array[masker]     # Met de blokhaken [] gebruik je het masker
print(" Elementen die > 5 zijn:  ", g5)

# Aanpassen van de waarde van alle elementen waarvoor het masker True is
test_array[masker] = 0       # Zet alle elementen >5 gelijk aan 0
print(" test_array (nieuw)     = ", test_array)
```

Oefenopdracht: Maken en bewerken van numpy arrays

e) Bereken het gemiddelde van het k^k-array van oefenopgave b).

2.5 For-loops

De structuur van een **for**-loop is altijd:

```
for element in data:
        voer_functie_uit_voor_element
```

Het Python key-woord **in** komt hierin dus altijd voor evenals de `:` aan het eind. Cruciaal is dat `data` elementen bevat. Eén zo'n element kun je noemen zoals je wilt, dus kies een nuttige naam. Kun je geen nuttige naam bedenken, dan is `i` eigenlijk de standaard. Vervolgens voer je berekeningen uit voor elk element. De volgende voorbeeld-code maakt dat duidelijk.

[9] In de .pdf zie je dat diverse Python woorden blauw gekleurd zijn; ook als er in een stukje tekst toevallig een Python woord staat, zoals in masker. Een klein foutje waar ik helaas niet veel aan kan doen.

Code 2.5: code-inc/w2/for˙vb1.py

```
import numpy as np

data = np.array([[1,2,3], [3,4,5]])
print('data = ', data)
print('Gemiddelde van ALLE items = ', np.mean(data))

# Berekening van het gemiddelde van elk item apart gaat met een for-loop:
for item in data:
    print('gemiddelde van', item, ' = ', np.mean(item))

# De volgende for-loop doet hetzelfde maar is wat omslachtiger
for i in range(len(data)):
    print('gemiddelde van data[', i, ']  = ', np.mean(data[i]))

# Als je data eenmaal in een array staat heb je geen for-loops nodig:
print('Kolom gemiddeldes = ', np.mean(data, axis=0))
print('Rij   gemiddeldes = ', np.mean(data, axis=1))

data2 = np.array([1,2,3,4,5,6]) # tweede test array

# De volgende for-loop start pas bij een bepaalde waarde (2)
for i in range(2, len(data)):
    print(data[i])
```

2.6 Slotopdracht: Voortschrijdend gemiddelde

Tijdens de corona pandemie hebben we eindeloos veel grafiekjes gezien van het aantal besmettingen per dag. In het weekend is dat altijd lager, en door de weeks juist weer hoger. Het is daarom nuttiger om naar het gemiddelde van de afgelopen zeven dagen te kijken: dan compenseer je de uitschieters een beetje. Dit gemiddelde noemen we het *voortschrijdend gemiddelde*: voor elke dag bereken je het gemiddelde van de zeven dagen daarvoor. Je kunt over de eerste 7 dagen geen voortschrijdend gemiddelde berekenen: er zijn immers nog geen volledige zeven dagen geweest.

In deze slotopdracht maak je zelf een voortschrijdend gemiddelde.

- Maak een array van 50 random getallen tussen de 0 en 1 (zoek hiervoor in het begin van het hoofdstuk).
- Maak een variabele genaamd `periode`, met de waarde 7. Dit is het aantal dagen waarover we het gemiddelde gaan berekenen.
- Start vervolgens een **for**-loop toe. Deze gaat lopen van de dag ná de periode, tot en met de laatste waarde van je array.
- Bereken per element in de **for**-loop het gemiddelde van de afgelopen 'periode' dagen. Voeg dit gemiddelde toe aan een array met de voortschrijdende gemiddeldes.

- Print na afloop van de loop de array met voortschrijdende gemiddeldes. Wat valt je op als je de periode verhoogt (naar bijvoorbeeld 20)?

3 Modules (2): MATPlotLib

In dit deel bekijken we hoe we data kunnen visualiseren: zichtbaar maken in grafieken. We gebruiken hiervoor het `pyplot` onderdeel uit het pakket `matplotlib` (MATLab Plotting Library). Het importeren werkt vergelijkbaar als bij `numpy`:

```
import matplotlib.pyplot as plt
```

3.1 Grafiek maken

Hieronder een voorbeeld om een sinus in beeld te brengen.

Code 3.1: code-inc/w3/plt˙vb1.py

```
#%% importeer de nodige modules
import numpy as np
import matplotlib.pyplot as plt

#%%
# Maak 2 arrays x en y, waarbij y=sin(x)
x = np.linspace(0.,2*np.pi,num=100)
y = np.sin(x)                          # Per element wordt sin(x) uitgerekend
plt.figure('Fig 1')                    # Maak een nieuw figuur 'Fig 1'
plt.plot(x,y)                          # Plot punten (x,y)

plt.figure('Fig 2')                    # Maak een nieuw figuur 'Fig 2'
plt.plot(x,y**2)                       # Plot sin(x)^2
```

Oefenopdracht

a) Imiteer het bovenstaande voorbeeld. Verander het aantal elementen in `x`. Wat gebeurt er als het aantal elementen in de lijst klein is?

b) Voeg een cosinus toe door een `y2` te definiëren en een extra `plt.plot()` toe te voegen.

c) Maak een extra plaatje (door in hetzelfde script een nieuwe `plt.figure()` toe te voegen) waar `y` en `y2` tegen elkaar uitgezet worden. In plaats van x en y in `plt.plot(x,y)` moet je dus wat anders invoeren.

3.2 Lijnen opmaken

Wanneer twee grafieken in één figuur getekend worden, zoals hiervoor het geval was, geeft `pyplot` automatisch de tweede plot een andere kleur. Om zelf kleur en stijl in te stellen is het mogelijk om een aantal commando's toe te voegen aan `plt.plot()`.

Code 3.2: code-inc/w3/plt˙vb2.py

```
## Hieronder een verzameling van kleuren
# b: blauw                             # m: magenta
```

```
# g: groen                              # y: geel
# r: rood                               # k: zwart
# c: cyaan                              # w: wit

## Het is ook mogelijk om de stijl te varieren. Enkele voorbeelden zijn:
# .:  punten                            # -.: stip-punt lijn
# o:  grote punten                      # : : stippellijn
# -:  lijn                              # --: gestreepte lijn

## Alle opties kunnen aan plt.plot() worden toegevoegd.
## Het voorbeeld hieronder produceert een plaatje met lijn en punten.
x = np.linspace(0,2*np.pi)

plt.figure()                  # nieuwe figuur
plt.plot(x,np.cos(x),'r-')    # cosinus plotten
plt.plot(x,np.cos(x),'k.')    # nog een cosinus eroverheen plotten
plt.show()                    # forceer om weer te geven
```

Oefenopdracht (vervolg)

d) Pas het script van b) aan zodat de sinus als rode lijn en cosinus als magenta punten geplot worden.

e) Bij het aanroepen van de `plt.plot()`, voeg de optie `label = "Label van grafiek"` toe binnen de haakjes van `plt.plot()`. Label de sinus "Sinus" en de cosinus "Cosinus". Gebruik vervolgens het commando `plt.legend()` onder de plot-functies om een legenda weer te geven.

f) De waarden op een bepaalde plek op de lijn zijn niet altijd zo makkelijk af te lezen bovenin of rechts in de figuur. Hier komt `plt.grid()` goed van pas. Hoe zit je figuur er uit als je dit toevoegt?

3.3 Assen opmaken

Een grafiek zonder tekst is (bijna) helemaal nutteloos. Zien we de windsnelheid, of de temperatuur, en dan in graden Celsius of Farenheit? En gaat het over de uren op een dag of over de maandnummers? Cruciale info, die aanwezig moet zijn in een grafiek. Uiteraard helpt `pyplot` ons hier ook weer een handje met een aantal nuttige functies. Zie onderstaand voorbeeld:

Code 3.3: code-inc/w3/plt˙vb5.py

```
jaren = np.arange(1981, 2022)                          # bereik x-as
temp = np.sqrt(jaren) - 2*np.random.rand(len(jaren))   # meetpunten maken
trend = np.sqrt(jaren) - 1                             # trendlijn maken

plt.figure()                                           # figuur maken
plt.plot(jaren, temp,'k.', label='Meting')             # meetpunten plotten
plt.plot(jaren, trend, 'r-', label='Trend')            # trendlijn plotten
```

```
plt.xlabel('Jaren')                                    # x–as label
plt.ylabel('$T$ ($^\circ$C)')                          # y–as label
plt.suptitle("Hoogst gemeten temperatuur in Madrid")   # titel
plt.legend()                                           # legenda
plt.grid()                                             # lijnen
plt.show()                                             # figuur weergeven
```

Oefenopdracht (vervolg)

g) De lijn in het plaatje van het voorbeeld toont gaten aan de linker- en rechterkant. Pas met `plt.xlim(x_min,x_max)` het bereik van de x-as aan. Maak het bereik van de x-as 1981 tot 2021. Je kunt het bereik op de y-as ook op dezelfde manier aanpassen. Zet het bereik van de y-as van 30 tot en met 50.

h) Probeer ook eens een titel toe te voegen met `plt.title()`. Wat is het verschil met `plt.suptitle()`?

i) In het plaatje van c) (wat een cirkel zou moeten tonen) is de verhouding van x- en y-as niet ingesteld. Zet de aspectratio (verhouding) op 1 met `plt.axes().set_aspect('equal')`. In plaats van `'equal'` kun je ook `1` invullen (verhouding 1:1).

j) Maak de volgende plot: kies t van 0 tot 2π. Laat $x = 16\sin^3(t)$ en $y = 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t)$. Zorg voor een goede verhouding tussen de assen.

3.4 Meer grafieken in één figuur

Soms is het fijn om twee plots boven elkaar of naast elkaar te laten zien, eventueel zodat ze de x- of y-as delen voor het overzicht (ze hebben dan hetzelfde bereik). Om meerdere plots tegelijk te laten zien kan de functie `plt.subplots()` gebruikt worden. `plt.subplots()` maakt een object dat de hele figuur beschrijft en voor elke grafiek in de figuur (subplot) een apart object, waarop met `axN.plot()` en gelijke functies figuren getekend kunnen worden. Hier is de naam `ax` niet verplicht, maar heel veel voorbeelden op internet werken met deze naam, dus doen we dat hier ook.

Code 3.4: code-inc/w3/plt˙vb3.py

```
# Begin met een subplots.
# fig bevat de informatie van de gehele figuur.
# ax1 t/m ax3 bevatten informatie over de individuele plots.
# sharex en sharey zorgen ervoor dat zowel x– als y–as gedeeld zijn.
fig, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=True)

x = np.linspace(0,2*np.pi)      # startwaarden

ax1.plot(x,np.cos(x)**2)        # plot sin(x)^2 in het eerste figuur
ax2.plot(x,np.sin(x)**2)        # plot cos(x)^2 in het tweede figuur
ax3.plot(x,np.cos(x)*np.sin(x)) # plot sin(x)*cos(x) in het derde figuur
ax1.grid()                      # lijnen toevoegen in het eerste plaatje
```

```
# suptitle verwijst naar de titel van de gehele figuur. Dit is dus een
# eigenschap van de totale figuur f.
fig.suptitle("Drie plotjes tegelijk")
plt.show()
```

Een tweede voorbeeld waar plotjes in een 2-bij-2 vorm worden getoond.

Code 3.5: code-inc/w3/plt˙vb4.py

```
# Begin met een subplots.
# sharex='col' zorgt dat de x—coordinaat gedeeld wordt over de kolommen
# sharey='row' idem voor rijen en de y—coordinaat
fig, axarr = plt.subplots(2, 2, sharex='col', sharey='row')

x = np.linspace(0,2*np.pi)                          # startwaarden

axarr[0,0].plot(x,np.cos(x)**2)                     # linksboven
axarr[0,1].plot(x,np.sin(x)**2)                     # rechtsboven
axarr[1,0].plot(x,np.cos(x)*np.sin(x))              # linksonder
axarr[1,1].plot(x,np.cos(x)**2*np.sin(x)**2)        # rechtsonder

# suptitle verwijst naar de titel van de gehele figuur. Dit is dus een
# eigenschap van de totale figuur fig.
fig.suptitle("Vier plotjes tegelijk")
plt.show()
```

Merk op dat gebruik wordt gemaakt van een 2-bij-2 array om de informatie voor de individuele plots op te slaan. De eerste index (het eerste getal) tussen de blokhaken staat daarbij voor de x-as, het tweede getal voor de y-as. Let op: Python begint linksbovenin met `[0,0]`, en heeft `[0,1]` linksonderin staan. De y-as loopt dus naar beneden in plaats van naar boven. De x-as 'klopt' wel.

Oefenopdracht (vervolg)

k) Maak een figuur met twee subplots: plot in de ene een sinus, en in de andere een cosinus. Kies de x-as tussen 0 en 2π met behulp van `axN.xlim(x_min, x_max)`. Voeg in allebei de plots de legenda toe. HINT: vervang `plt` nu door `ax1` en `ax2`.

3.5 Slotopdracht: Heel veel plotjes

Maak een figuur van 2-bij-2 plots waarin net als in het voorbeeld met de drie plotjes de x-markering weggelaten is voor figuren die niet onderaan staan. Laat ook de y-markering weg voor figuren die niet helemaal links staan. De plots moeten er als volgt uit zien: voor elke plot is het bereik voor x gelijk aan $[0, 1]$. In elke plot moet x^α komen staan. In linksboven voor $\alpha = 0.125, 0.25, 0.5, 1$, in rechtsboven $\alpha = 1, 1.5, 2$, in linksonder $\alpha = 1, 2, 3, 4$ en rechtsonder $\alpha = 0.1, 1, 10$. Je kunt dit met de hand, regel voor regel doen, maar de uitdaging van deze opdracht zit hem vooral in het zo compact en efficiënt mogelijk maken van je script. Voeg achteraf ook labels en legenda toe en maak de figuur met behulp van de geleerde dingen in dit hoofdstuk zo netjes mogelijk.

4 Importeren en exporteren

Voor het verwerken van data ben je eigenlijk altijd afhankelijk van losse bestanden die alle gegevens bevatten. Niemand gaat voor de lol duizenden getalletjes met de hand invoeren: dan kun je net zo goed Excel gebruiken. Ook wil je graag je gemaakte figuren en resultaten opslaan om ergens in te voegen: anders kun je net zo goed maar wat met de hand tekenen of berekenen. Maar daar doen we het niet voor: we gaan handelingen automatiseren door bestanden te *importeren*, en vervolgens resultaten te *exporteren*. Dat gaan we dit hoofdstuk behandelen.

4.1 Afbeeldingen

In het vorige hoofdstuk heb je al geleerd om de module `matplotlib` te gebruiken voor het visualiseren van data. Figuren worden standaard alleen getoond binnen de Python omgeving en kunnen niet direct gebruikt worden in een verslag, artikel of rapport. Daarvoor moet je de afbeelding eerst opslaan. Hiervoor kan je gebruik maken van `plt.savefig()`.

De belangrijkste opties die aan deze functie moeten worden meegegeven zijn de naam van de afbeelding en de resolutie. De standaard locatie waar de afbeelding wordt opgeslagen is de map waar het script is opgeslagen.

Als bestandstype kan er worden gekozen voor `jpg`, `png` en `pdf`. De resolutie van deze afbeeldingen wordt gespecificeerd in *dots-per-inch* (`dpi`). Afbeeldingen van het type .jpg en .png kun je niet zo ver inzoomen, omdat op een gegeven moment de pixels zichtbaar worden. Afbeeldingen van het type .pdf zijn opgebouwd uit formules zoals punten, lijnen en/of krommen. Ook gewone letters hebben zo'n structuur. Daardoor kun je deze zogenaamde *vectorafbeeldingen* zo ver inzoomen als je zou willen zonder dat het onscherp wordt.

Met behulp van `plt.figure(figsize=[breedte, lengte])` kun je instellen hoe groot je plaatje is, en welke verhoudingen hij heeft. Dit heeft ook effect op de bestandsgrootte van je afbeelding: hoe groter de breedte en lengte, hoe groter het bestand.

Code 4.1: code-inc/w4/export˙vb1.py

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-2*np.pi, 2*np.pi, num=1001)
y = np.sin(x)
plt.figure(figsize=[8,4])
plt.plot(x,y)
plt.savefig('sinus.png', format='png', dpi=300)
```

Oefenopdracht

- Open het onderstaande voorbeeld-script `export_vb2.py`. Je kunt met dit voorbeeld ook makkelijk meerdere figuren in één script aanmaken.
- Verander het script zodat het een aantal plaatjes van alleen de sinus-functie maakt. Om niet steeds hetzelfde plaatje te krijgen varieer je met de volgende instellingen:

1. De extensie van het weggeschreven bestand. Kies: `png` of `pdf`. Hierin is al voorzien in het voorbeeld door tweemaal een `plt.savefig` te doen.
2. De grootte van het plaatje. Kies bijv. `[2,1]` of `[20,10]`.
3. Het aantal dots-per-inch. Kies bijv. 300 of 1200. (Zie het vorige voorbeeld!)

- Kijk grondig naar de kwaliteit van de aangemaakte plaatjes, door ze te openen met de standaard app voor .png of .pdf plaatjes. Gebruik de zoom-functie! Kijk ook naar de bestandsgrootte (in kilobytes) van de aangemaakte bestanden. Trek je conclusie!

Code 4.2: code-inc/w4/export˙vb2.py

```
import numpy as np
import matplotlib.pyplot as plt

plt.figure(1) # begin met opbouwen van Figuur 1, default afmeting

x = np.linspace(-2*np.pi, 2*np.pi)
y = np.sin(x)
plt.plot(x, y, color = 'red')  # 1 curve is getekend

# nu een gladdere curve tekenen door meer (dan 50) punten te kiezen
x = np.linspace(-2*np.pi, 2*np.pi, num=200)
y = np.sin(x)

# de breedte van de te tekenen curve moet ook worden aangepast
# anders zie je nog geen verschillen met de rode lijn
plt.plot(x, y, linewidth=0.1, color='black')

plt.savefig('sinus.png')      # sla verschillende bestandstypen
plt.savefig('sinus.pdf')      # op om te kunnen vergelijken

plt.figure(2, figsize=[8,4])  # begin met opbouwen van Figuur 2
                              # met opgegeven afmeting in inches
plt.plot(x, np.cos(x))
plt.savefig('cosinus.pdf')    # ... en weer een figuur opgeslagen.
```

Zoals je merkt is exporteren van afbeeldingen eenvoudig. Het exporteren van gegevens (meestal zijn dat numerieke data die te maken hebben met de analyse van een experiment) kost wat meer werk.

Voordat we hier mee aan de slag gaan, worden nog twee bouwstenen aangereikt: **for**-loops en het *formatteren* oftewel opmaken van tekst.

4.2 For-loops

Eerder zijn we uitgebreid ingegaan op een speciaal soort lijsten, `numpy`-arrays, omdat deze heel erg geschikt zijn voor data-analyse. Het rekenen met `numpy`-arrays is heel erg efficiënt (er is weinig code nodig) en snel (er is weinig computertijd nodig). Je kunt over het algemeen veel beter een array gebruiken dan een Python list. Run het volgende voorbeeld maar eens:

Code 4.3: code-inc/w4/forloop˙vb1.py

```
lijst = ["pa", "ma", 1, [2,3]]
# met deze lijst kun je niet rekenen
for elem in lijst:
    print(4*elem)
# ... maar wel grappige output produceren
```

Omdat een Python lijst uit allemaal verschillende datatypen kan bestaan, werkt dat in een **for**-loop totaal niet goed. Daar heb je het liefst dat alles op dezelfde manier werkt, zodat je op elk element dezelfde berekening of handeling kunt doen. Zo kun je de gehele lijst, van begin tot eind, op dezelfde manier bewerken.

Oefenopdracht: Spelen met lijsten en printen binnen for-loop

- Maak een `numpy` lijst l_1 met 100 elementen: de gehele getallen 0 tot en met 99. Vermenigvuldig elk getal in die lijst met 100; noem deze lijst l_2.
- Maak een nieuwe lijst $l_3 = l_2 - l_1^2$. Sorteer de elementen in l_3.
- Zorg ervoor dat de 9 grootste elementen van l_3 in aflopende volgorde in een lijst l_4 terechtkomen (die dus 9 elementen bevat).
- Maak nu een nieuwe lijst l_5 die deze 9 waarden bevat, alleen nu geordend als een 3 bij 3 array. Dit is dus een twee-dimensionaal array. Het eerste element bevat de 3 grootste getallen, etc.
  - Druk nu de gehele lijst l_5 met één print af.
  - Druk ook elk element van de lijst l_5 af in een **for**-loop over alle elementen van l_5. Let er hierbij op dat l_5 stiekem maar 3 elementen heeft (met elk weer 3 elementen)!

  Noteer de verschillen in de afgedrukte output.

4.3 Formatteren van tekst

Zoals eerder genoemd moet de functie **print()** uitdrukkelijk worden aangeroepen om uitkomsten op het beeldscherm te tonen. Meestal wil je alleen aan het eind van je script weten en zien wat er berekend is, maar tijdens de ontwikkeling van je script is een **print()** op een aantal plekken best handig. In het begin hebben we losse waarden op het beeldscherm getoond: vaak dus één getal per regel, maar er kunnen ook meerdere berekende waarden in een keer op het scherm worden getoond met één **print** statement. Dit kan worden bereikt door meerdere variabelen op te geven gescheiden door komma's. Ook het gebruik van de puntkomma om meerdere variabelen in je Python script in één regel een waarde te geven kan soms handig zijn.

```
a = 1; b = 2; c = 3
print('a =', a, 'b=', b, 'c=', c)
```

Het kan handig zijn tekst toe te voegen, zodat berekende waarden meer voor zichzelf spreken, zoals in onderstaand voorbeeld.

```
print('Na ', t, 'seconden, is de bal', y, 'meter hoog.')
```

Een andere manier om tekst te tonen op je beeldscherm kan met behulp van **format**. Je hoeft dan veel minder komma's en quotes te gebruiken en hebt er meer controle over. Het is dan niet langer nodig verschillende stukken te scheiden met komma's, maar je gebruikt een enkele `string`. Hierin geef je met accolades ({}) aan waar de waarden van variabelen kunnen worden ingevuld en sluit je af met de functie format, waarin je de namen van de variabelen aangeeft.

In de accolades kun je met een code aangeven hoe je tekst er uit komt te zien. Deze *codes* beginnen met een dubbele punt gevolgd door een aantal letters. Zie tabel 1 voor een aantal mogelijkheden.

Tabel 1: Codes voor het weergeven van gegevens.

:d	integer	`500`
:f	decimale notatie (met 6 decimalen)	`500.000000`
:.xf	decimale notatie met x decimalen	`500.00` ($x = 2$)
:e	wetenschap. notatie (met 6 decimalen)	`5.000000e+02`
:.xe	wetenschap. notatie met x decimalen	`5.0e+02` ($x = 1$)
:s	string	`'5'`

Oefenopdracht: Oefenen met het precies formatteren van tekst

- Verander de manier waarop de getallen t en y worden geformatteerd. Hoeveel correcte decimalen van `np.pi` kun je zo op je scherm te zien krijgen? (Antwoord: 15). Probeer behalve het aantal decimalen te variëren ook te spelen met het type door t en y af te drukken als string of integer! Let op wat er dan gebeurt. Je kunt gebruik maken van onderstaand voorbeeld:

```
t = np.pi
y = 22/7
print('Na {} seconden is de bal {} meter hoog'.format(t,y))
print('Na {:.2f} seconden is de bal {:.2f} meter hoog'.format(t,y))
```

4.4 Exporteren

Naast afbeeldingen is het noodzakelijk om de achterliggende gegevens te bewaren voor later gebruik. Hiervoor zijn talloze types bestanden beschikbaar, elk toegespitst op een bepaald soort gegevens. Hoewel er erg geavanceerde bestandstypes bestaan die weinig ruimte in beslag nemen of erg snel werken met bepaalde programma's, zullen we ons in deze cursus beperken tot tekstbestanden (.txt of .csv). Deze bestanden zijn leesbaar voor de mens, wat uiteraard fijn is.

Structuur aanbrengen in een bestand is noodzakelijk. Dit begint bij het ordenen van je data. Daarnaast is het ook belangrijk om extra informatie (ook wel *metadata*) aan het bestand toe te voegen. Denk aan namen van variabelen, maar ook meetomstandigheden, de datum en andere informatie die een gebruiker van het bestand nodig heeft om de gegevens te verwerken.

Voor het aanmaken van bestanden kan goed gebruik worden gemaakt van de ingebouwde *functies* **open()** en **write()**, net zoals jij een Word document opent en gaat typen. De functie **open()** creëert een *object* dat aangeeft in welk bestand de gegevens moeten worden opgeslagen, terwijl met de *functie* **write()** het schrijven van de gegevens gebeurt. Net zoals de fysieke tegenhanger, pen en papier, gebeurt dit woord voor woord, van boven naar beneden en van links naar rechts. Deze functie **write()** werkt overigens (nagenoeg) hetzelfde als **print()**.

In onderstaand voorbeeld staat: **file.write('blabla'.format(x,y,z))**. Hierin is '**file**' de (willekeurige) naam van het object en '**write()**' de functie die aangeroepen wordt. Tussen de ronde haakje van de **write** zien we eigenlijk hetzelfde als in hoofdstuk 4.3 bij **print()**.

Code 4.4: code-inc/w4/write˙vb1.py

```
""" Data wegschrijven in een bestand inclusief een header
"""
import numpy as np

# Specificeren van object in Python dat het nieuwe bestand aanwijst
# 'w' staat voor 'write' en geeft aan dat het bestand geschreven mag worden
file = open('vb1.txt', 'w')

# Aanmaken van numpy arrays om weg te schrijven naar bestand
x = np.linspace(-2*np.pi, 2*np.pi, num=20)
y = np.sin(x)

# Metadata
col0 =  '     i'  # een onbelangrijke string; alleen voor leesbaarheid
col1 =  '      x' # een onbelangrijke string; alleen voor leesbaarheid
col2 =  ' sin(x)' # een onbelangrijke string; alleen voor leesbaarheid
par1 = 50         # een waarde / parameter 1
par2 = 0.12       # een waarde / parameter 2

# Schrijven van header, herkenbaar aan # op eerste positie van elke regel
file.write('# Dit is de header (eerste regel) \n')
file.write('# Schaal parameter horizontaal:\t {:5.2f}\n'.format(par1))
file.write('# Schaal parameter verticaal  :\t {:5.2f}\n'.format(par2))
file.write('#{:s} \t {:s} \t {:s}\n'.format(col0, col1, col2))
# de opgegeven namen worden weggeschreven als een character-string met
# daartussen steeds een \t (tab) voorafgegaan/gevolgd door een spatie

# Schrijven van (meet)gegevens:
for i in range(len(x)):
   file.write('{:7d} \t {:7.4f} \t {:7.4f} \n'.format(i, x[i], y[i]))
# op iedere regel staat nu: een geheel getal op de eerste 7 posities,
# spatie, tab, spatie, decimaal getal met 4 decimalen op 7 posities,
# spatie, tab, spatie, decimaal getal met 4 decimalen op 7 posities,
# spatie, en de regel wordt afgesloten met een \n (new-line symbool).
# MERK OP: de laatste  regel is een lege regel.

file.close()
```

Opmerking 1: Om een nieuwe regel te beginnen wordt het regeleinde `\n` symbool opgegeven.

Opmerking 2: Tussen de verschillende items die worden weggeschreven, schrijven we altijd een komma. Dit is een conventie die we in deze cursus aanhouden[10]. Het maakt het makkelijker om elkaars databestanden zo weer in te lezen **en** de inhoud ervan correct te verwerken.

Oefenopdracht: Wegschrijven van data naar file

Maak 4 arrays van 100 elementen. Het eerste array is van het datatype `int` (hiervoor kun je bij het maken van de array de tekst `dtype=int` invoegen tussen de haakjes), en bestaat uit de gehele getallen 0 t/m 99. Het tweede array bestaat uit honderd random getallen tussen de 0 en 1. Het derde array is het eerste array gedeeld door het tweede array. Het vierde array is 10 tot de macht van het eerste array. Hierin komen dus hele grote getallen te staan. Je kunt gebruik maken van bovenstaand voorbeeld, maar je moet daarin wel redelijk wat aanpassen. Let vooral erg goed op de formattering van de tekst!

4.5 Importeren

Net als bij het exporteren van gegevens voor later gebruik, moet er bij importeren een vertaalslag worden gemaakt om de gegevens uit een bestand om te zetten zodat deze bruikbaar zijn binnen Python. Dit kan bijvoorbeeld met standaard Python functies zoals `read()` of `readlines()`.

In deze cursus focussen we echter op bestanden die we zelf aan de hand van een meting hebben aangemaakt. Als je zo'n bestand correct wilt kunnen inlezen dan moet je van tevoren daarover duidelijke afspraken maken. Afspraken die bij het natuurkundepracticum zijn gemaakt:

1. Het bestand bestaat uit regels (lines). Elke regel bevat of numerieke data of het is een commentaarregel. Zoals in elk tekst bestand zijn regels van elkaar gescheiden door het newline-karakter: '`\n`'.

2. Elke regel bestaat uit één of meerdere elementen die van elkaar gescheiden door komma's. Het scheidingsteken heet de *delimiter*. We spreken af dat dit het `''`,-karakter is. Met `np.genfromtxt` kun je de delimiter specificeren. Andere veelvoorkomende delimiters zijn: `';'` en `'t'`. Het standaardteken van de delimiter (als die niet gespecificeerd wordt) is om de kolommen te splitsen bij elke aangesloten witte ruimte; dit kan dus een (serie) spatie(s) of een tab zijn.

3. Op elke regel met numerieke data staan evenveel getallen; lege posities zijn dus niet wenselijk.

4. Een commentaarregel is herkenbaar aan het `#` teken op de eerste positie. Precies zoals in een Python script. Commentaarregels aan het begin van een bestand heten ook wel de *header*.

Het doel is om bestanden die aan deze eisen voldoen te kunnen importeren en vooral correct te kunnen verwerken in Python. Concreet betekent dit dat na het inlezen van het bestand een `np.array` wordt aangemaakt van de data. Daarnaast is het mogelijk dat een aantal variabelen

[10] Een tab als scheidingssymbool tussen de items zou ook kunnen, en wordt in de praktijk ook vaak gebruikt. Het symbool daarvoor is '\t'.

(de parameters), zoals `par0` en `par1`, de waardes krijgen uit de *header* moeten krijgen, maar dat doen we wel voorlopig wel handmatig.

Je kunt een bestand natuurlijk regel voor regel, woord voor woord, letter na letter inlezen, maar het is natuurlijk een stuk handiger als er functies beschikbaar zijn die een heleboel van dat puzzelwerk voor je uit handen nemen. En gelukkig bestaan die ook. Het inlezen van bestanden die numerieke data bevatten in de vorm van een array (rijen en kolommen) kan heel makkelijk gebeuren met de `numpy` functie `genfromtxt()`. Deze functie negeert regels die beginnen met een `#` en dat komt goed uit, want dat is meestal toch vooral commentaar. Het weergeven van bijvoorbeeld twee kolommen uit de ingelezen data in een xy-plot stelt dan echt weinig meer voor. Zie volgend voorbeeld.

Code 4.5: code-inc/w4/genfromtxt˙vb1.py

```
'''
Voorbeeld: het gebruik van genfromtxt
'''
import numpy as np
import matplotlib.pyplot as plt

# de naam van het bestand
dir = 'http://nspracticum.science.uu.nl/DATA2020/DATA-Py/Databestanden/'
filename = dir+'vb1.txt'  # het 'optellen' van strings

# lees de data in dit tab-gescheiden bestand
data = np.genfromtxt(filename, delimiter='\t')

# als het goed is kun je zien dat data een array met floats is
# en de size ervan is (20,3); zie in Spyder bij je Variable explorer
# of bekijk het resultaat van onderstaande print
print('size of data = ', np.shape(data))

# plot de ingelezen data; kolom 0 wordt niet gebruikt
(x, y) = (data[:,1], data[:,2])
plt.plot(x,y)

plt.show()
```

4.6 Slotopdracht: De Europese bananeninspectie

In de eindopdracht van deze week passen we het geleerde toe op de strenge Europese regelgeving voor bananen. Eerst genereren we een dataset, deze bevat parameters zoals de diameter, lengte en kromming van bananen. Vervolgens passen we verschillende criteria op de data toe en toetsen we de bananen aan Europese regelgeving. Ook toetsen we het mythische 'kromte-criterium' voor bananen en bekijken hoe dit invloed zou hebben op de verdeling van de bananen over verschillende klassen.

1: Importeren van data

Bekijk het volgende voorbeeld-bestand: Voorbeeld-0.

Sla een kopie van dit bestand op in dezelfde map waar ook het Python-script staat dat dit bestand moet gaan lezen en bewerken. Gebruik `np.genfromtxt` met altijd de tab als `delimiter`, zodat je de meetgegevens in het bestand kunt inlezen en die in een `numpy`-array om kunt zetten.

2: Testen op dikte

We zullen de officiële criteria van de Europese Commissie aanhouden zoals vastgesteld in *Commission Regulation (EC) No 2257/94*. Deze richtlijnen stellen dat de minimale lengte van een banaan $L_{\min} = 14\,\text{cm}$ en de minimale dikte is $d_{\min} = 27\,\text{mm}$. We passen nu nog geen criteria toe op de kromtestraal R of de smetten op het oppervlak A.

Voor bananen uit bepaalde Europese grondgebieden (Madeira, de Azoren, de Algarve, Kreta en Laconië) geldt vanwege klimaatfactoren het criterium op de lengte niet, maar mogen deze toch op de Europese markt verkocht worden. Ervan uitgaande dat de dataset gegenereerd bij Opdracht 1 betrekking heeft op bananen uit één van deze regio's, bereken het percentage bananen wat zou worden afgekeurd.

3: Lengte en dikte

Na onderzoek blijkt dat een slimme bananenhandelaar uit Portugal heeft geprobeerd om een partij bananen uit India als bananen afkomstig uit de Algarve te verkopen om zo de Europese regelgeving omtrent de lengte van een banaan te omzeilen. Toets de partij bananen nu niet enkel op dikte, maar ook op lengte. Hoeveel procent van de bananen wordt nu afgekeurd?

4: Klasseindeling

Binnen de Europese richtlijnen worden bananen in Klassen ingedeeld. De "Extra" klasse bevat bananen van "superieure kwaliteit". Klasse I bananen is de standaardklasse en Klasse II bevat bananen met lelijke vorm. De bijbehorende criteria staan in de tabel hieronder.

Deel de bananen aan de hand van de criteria in bovenstaande tabel in in de verschillende klassen. Rapporteer voor de gegenereerde dataset hoeveel procent van de bananen in de Extra klasse, hoeveel in klasse I en hoeveel in klasse II worden ingedeeld, rapporteer ook hoeveel procent van de bananen afgekeurd is.[11] Controleer of de bepaalde percentages optellen tot 100%.

Tabel 2: Criteria gesteld aan Bananen in de Europese Unie, per Klasse.

	Extra	**I**	**II**
d	$\geq 27\,\text{mm}$	$\geq 27\,\text{mm}$	$\geq 27\,\text{mm}$
L	$\geq 14\,\text{cm}$	$\geq 14\,\text{cm}$	$\geq 14\,\text{cm}$
R	$1.25L \leq R \leq 1.3L$	$1.2L \leq R \leq 1.4L$	geen
A	$\leq 1\,\text{cm}^2$	$\leq 2\,\text{cm}^2$	$\leq 4\,\text{cm}^2$

[11] Het correcte antwoord ligt rond: 1.52±0.04% Extra, 12.84±0.10% Klasse I, 66.50±0.15% Klass II en 19.13±0.13% afgekeurd.

5 Functies

In hoofdstuk 1 hebben we al kort gekeken naar functies. In dit hoofdstuk gaan we hier dieper op in, want het is heel erg fijn om functies te gebruiken. Een functie is een stuk code dat eerder gemaakt is en een bepaalde taak uitvoert. Het resultaat van een functie, bijv. de sinus-functie, wordt bepaald door de waarden die je meegeeft als argument(en). In Python kun je met **def** heel makkelijk zelf allerlei functies maken en gebruiken. Er zijn meerdere mogelijkheden:

- Functies kunnen nul of meer argumenten hebben.
- Soms hoef je argumenten niet op te geven en wordt een standaardwaarde (*defaultwaarde*) gebruikt.
- Soms kom je functies tegen met argumenten die een naam/label hebben: de *keyword* argumenten.
- Soms kom je functies tegen die een wisselend aantal argumenten gebruiken.

Je hebt hier bijvoorbeeld al kennis mee gemaakt met `plt.plot(x,y,'k+',label='sinus')`. Hierin is `y` een argument, `x` en `'k+'` een optioneel argument en `label='sinus'` een keyword argument (omdat je `label=...` gebruikt).

In dit hoofdstuk leer je hoe je zelf functies kunt schrijven en gebruiken. Dit is ook nuttig als je op internet informatie over functies opzoekt (documentatie).[12]

Let op: een functie met nul argumenten zal altijd hetzelfde doen. Denk bijvoorbeeld aan `plt.show()`: er staat nooit iets tussen de haakjes en deze functie zal dan ook altijd precies hetzelfde doen.

5.1 Constructie van functies

Informatie wordt naar een functie gestuurd in de vorm van argumenten. Een serie argumenten wordt opgegeven en de volgorde van de argumenten bepaalt welk argument welk is.

```
def f(a,b):
    return a + 2*b

print(f(1,2)) # 5
print(f(2,1)) # 4
```

Het is ook mogelijk om argumenten te labelen, dan maakt de volgorde niet uit:

```
print(f(a=1,b=2)) # 5
print(f(b=1,a=2)) # 4
print(f(b=2,a=1)) # 5
```

Als sommige argumenten wel, en andere niet gelabeld zijn, dan moeten de **niet** gelabelde argumenten eerst komen, en er mag geen conflict optreden. De volgende twee functie aanroepen geven dan ook een foutmelding: `f(a=1, 2)` en `f(1, a=2)`, terwijl `f(1,b=2)` wel correct is.

[12] Kijk bijvoorbeeld op https://matplotlib.org/api/_as_gen/matplotlib.pyplot.plot.html voor de documentatie over de functie `plt.plot()`.

5.2 Extra argumenten

Een functie kan een *wisselend* aantal argumenten mee krijgen. Die voer je in door bijvoorbeeld `f(a,b,*args)`, waar `*args` (van *arguments*) een willekeurig aantal argumenten kan zijn. Dat maakt het sterretje ervoor duidelijk. Bijvoorbeeld het optellen van een willekeurig aantal elementen:

```
def mijn_som(*args):
    som = 0
    for arg in args:
        som += arg
    return som

print( mijn_som(1,2,3) )      #  6
print( mijn_som(2,1,3,4,5) ) # 15
print( mijn_som() )           #  0
```

Dit is natuurlijk een erg droog voorbeeld. Let erop dat je op deze functie niet kunt gebruiken om bijv. de elementen in een lijst op te tellen. Dus

```
mijn_lijst= [1,2,3]
print( mijn_som(mijn_lijst) )     # geeft foutmelding
```

Oefenopdracht: Extra argumenten

(a) Test de functie `mijn_som`. Krijg je de voorspelde uitkomsten?

(b) Probeer ook het voorbeeld met `mijn_lijst`. Hoe kun je tóch de lijst gebruiken? (*Hint:* maak gebruik van een **for**-loop om losse argumenten te maken.)

Een handige toepassing is wanneer een functie een andere functie moet aanroepen. Ter verduidelijking hieronder een voorbeeld, waarin de functie `my_plot` met soms 4 en anders 5 argumenten moet worden aangeroepen om de gewenste plaatjes te krijgen.

Code 5.1: code-inc/w5/voorbeeld5˙2.py

```
# Rechte lijn
def f(x,a,b):
    return a + b*x

# Sinus
def g(x,a,b,c):
    return a + b*np.sin(c*x)

## Simpele plotfunctie:
# plot "functie" op het bereik "x"
# extra argumenten worden doorgegeven met *args
def my_plot(x,functie,*args):
    plt.plot(x,functie(x,*args))
    plt.show()
```

```
# Bereik:
x_arr = np.linspace(-5.,5., num=100)
```

Het runnen van dit script 'doet' niks (geeft geen output). Na het runnen zijn de functies en variabelen in het script wel actief in de *console*.

Oefenopdracht: Extra argumenten (vervolg)

(c) Laad en run de voorbeeldcode met de functie `my_plot`. Test nu één voor één de volgende vier regels code: (*Hint:* type deze regels in de *command line* van de Console in Spyder; waar normaal je geprinte text verschijnt.)

```
my_plot(x_arr,f,1,2)
my_plot(x_arr,g,1,2)
my_plot(x_arr,f,1,2,3)
my_plot(x_arr,g,1,2,3)
```

Welke werken er wel en welke niet? Begrijp je waarom?

(d) Maak ook een eigen functie met 4 argumenten (wees creatief, gebruik bijvoorbeeld machten, wortels en sinussen) en plot deze functie met de `my_plot` functie van hierboven.

5.3 Defaultwaarde

Sommige argumenten worden maar zelden gebruikt of juist altijd op dezelfde manier. Een plot van temperatuurverloop bijvoorbeeld is eigenlijk altijd prettiger te lezen mét gridlijnen. Maar het is wel fijn als je dit soms kunt aanpassen (in dit geval het grid onderdrukken). Daarvoor zijn defaultwaarden erg praktisch: je kunt een argument standaard een waarde toekennen in een functie, tenzij je het anders opgeeft. Kijk maar eens naar het volgende voorbeeld:

Code 5.2: code-inc/w5/voorbeeld5˙3.py

```
## Simpele plotfunctie:
# plot een rechte lijn op het bereik "x"
# voor een grid maken we gebruik van een defaultwaarde
def my_plot(x,a,b,*args,grid=True):
    y = a + b*x
    plt.plot(x,y)
    if grid:
        plt.grid()
    plt.show()

# Bereik:
x_arr = np.linspace(-5.,5., num=100)
```

Oefenopdracht: Defaultwaarde

1. Herhaal de opdracht van de vorige paragraaf, maar nu met de volgende regels:

```
my_plot(x_arr,1,2)
my_plot(x_arr,1,2,True)
my_plot(x_arr,1,2,False)
```

   Wat gebeurt er?

5.4 Slotopdracht: Eigen plotfunctie

Als je met een bak data aan het werk bent, maak je vaak plots die (bijna) helemaal hetzelfde zijn, op de lijnen na. Daarom is het best fijn om een standaardfunctie te hebben die je in elke opdracht opnieuw kunt gebruiken, en met behulp van argumenten aan te passen is naar hoe de specifieke opdracht. In deze opdracht heb je alle vrijheid, op één regel na: je `my_plot`-functie begint met deze regel:

```
def my_plot(x,functie,*args,style='k-',label=None,
            xmin=min(x),xmax=max(x),ymin=None,ymax=None,
            xlabel=None,ylabel=None,titel=None,
            legenda=False,grid=True,show=True):
```

Hier staat `x` voor het bereik op de x-as, `functie` voor de functie die op de y-as geplot wordt, `*args` voor de argumenten die die functie nodig heeft. Vervolgens komen de keyword-arguments: `style` staat voor de kleur en vorm van de lijn, `label` staat voor het label van je lijn. `xmin, xmax, ymin` en `ymax` staan voor het bereik op de x- en y-as. `xlabel, ylabel` en `titel` spreken voor zich; `legenda` voor `plt.legend()`, `grid` voor `plt.grid()` en `show` betekent `plt.show()`.

Als *proof-of-concept* moet je vervolgens het volgende data-bestand downloaden en importeren, om vervolgens elke kolom te plotten met behulp van jouw plot-functie. Plot ook in elke figuur het bijbehorend voortschrijdend gemiddelde over de afgelopen 10 waarden. Hiervoor kun je je functie van de slotopdracht van hoofdstuk 2 hergebruiken. (Hoe goed was je commentaar destijds? Begrijp je jouw script van toen nu nog steeds?)

6 PLUS: Gebruiksvriendelijke code

Tot nu toe heb je gewerkt met kant-en-klare scripts, en heb je ook soms eigen code helemaal zelf geschreven. Je hebt waarschijnlijk gemerkt dat je veel van die code (soms met enige aanpassingen) kunt hergebruiken in volgende opdrachten. In dit hoofdstuk leer je hoe je dit nog makkelijker en efficiënter kunt doen; programmeren is tenslotte bedoeld om je leven[13] makkelijker en sneller te maken.

6.1 Commentaar schrijven

Meestal ben je zelf de gebruiker van de code die je schrijft, maar je moet dit toch leesbaar maken voor anderen. Niet alleen de docent moet het kunnen volgen om een cijfer te kunnen geven, maar als je code leert schrijven zonder commentaar dan is jouw code meteen compleet onbruikbaar voor anderen. Vuistregel bij het schrijven van commentaar is dat je wilt dat wanneer je over 2 jaar de code weer opent weinig moeite hebt om te achterhalen wat er waar gebeurt, en hoe je de code opnieuw kunt gebruiken.

De hoeveelheid commentaar is een persoonlijke keuze, maar met te weinig commentaar maak je je code onleesbaar en daarmee compleet onbruikbaar in de toekomst. Goed commentaar schrijven kost tijd, dus houd daar rekening mee tijdens het programmeren - het schrijven van goed commentaar is onderdeel van het schrijven van code, en een script is niet af voordat er goed commentaar bij staat. Het is handig om commentaar gaandeweg te schrijven, en niet pas op het eind. Dit maakt het oplossen van problemen makkelijker en is ook handig voor jezelf, want je weet juist op het moment dat je de code voor het eerst schrijft het best wat het doel van dat stukje is. Veel commentaar is niet per se goed commentaar. Zorg dat je commentaar ook echt iets zegt en helpt te begrijpen waarom een stuk of regel code gebruikt wordt:

```
# slecht commentaar:
x = x + 1          # verhoog x
# nuttig commentaar:
x = x + 1          # compenseer voor index 0
```

Richtlijnen voor het schrijven van goed commentaar:

- Gebruik *natuurlijke taal*, d.w.z. gebruik zinnen en schrijf voluit. Gebruik geen Python code in comments,[14] en definieer (óók voor de hand liggende) symbolen als F, g en T.
- Zet commentaar bij het toekennen van numerieke waardes; zeg bij welke grootheid (evt. met symbool tussen haakjes) ze horen en welke eenheid ze hebben.

```
g = 9.81        # gravitatie constante (g) in m/s^2
v0 = 25         # snelheid (v) bij t=0 in m/s
n = 100         # aantal punten voor berekening
```

- Wanneer je `print()` of `input()` gebruikt, print dan niet alleen de waarde of variabele waarin je geïnteresseerd bent, maar print ook een beschrijving met context over die waarde of een specifieke instructie voor wat de input betekent.

[13] Data analyse binnen deze cursus, maar ook alles wat je in de toekomst met programmeren zult doen
[14] Tenzij het een stukje code is wat tijdelijk niet gebruikt wordt.

- Elke functie heeft minimaal een beschrijving van wat die doet, wat de input en output is en wat de eventuele `*args` zijn.
- Bovenaan het script staat een algemene beschrijving van het script en voor welke toepassingen deze geschikt is.
- Blokken commentaar (met `'''text'''`) worden gebruikt om lange stukken commentaar te geven, bijvoorbeeld bij de uitleg van een functie, of bovenaan het script.
- In-line commentaar (met `# text`) staat precies op de plek waar het relevant is, en staat bij voorkeur uitgelijnd rechts van de code om de leesbaarheid te verbeteren.

Dan een aantal richtlijnen die niet per se met commentaar te maken hebben, maar die wel de leesbaarheid van je script sterk kunnen vergroten:

- Maak de regels niet te lang. In Spyder (en de meeste andere editors) kun je een verticale lijn laten weergeven na een specifiek aantal karakters.[15] Zorg dat regels code en commentaar niet (veel) langer worden dan tot die lijn. Bij Python is de conventie om deze na 79 karakters te zetten. Let op dat je door syntax niet zomaar een nieuwe regel kunt beginnen; dit heeft namelijk een betekenis binnen Python. Door gebruik van de juiste *indentatie* kun je toch aan de 79-karakter-limiet voldoen.
- Gebruik `#%%` om *cells* te maken die los te runnen zijn. Let op: bij goede modulaire code (waarbij de acties in functies staan) is dit juist niet altijd handig. Tijdens het ontwikkelen van de code kunnen cells wel uitermate handig zijn, zodat je bijvoorbeeld een tijdrovend deel van je code kunt overslaan (bijvoorbeeld het genereren of analyseren van data) terwijl je door werkt aan een ander stuk (bijvoorbeeld het plotten van data).
- Schrijf bovenaan elke cell wat er in die cell gebeurd (welke 'titel' je de cell zou geven).
- Gebruik scheidingsmarkeringen tussen lange stukken code. Een lege regel tussen het importeren van packages en de start van de code, en een lege regel tussen het einde van een loop en het vervolg van de (niet geïndenteerde) code, of een stuk code gescheiden door een regel `#----------` geven het script een overzichtelijkere uitstraling en maken het daarmee beter leesbaar.
- Verkies leesbaarheid over snelheid.

```
# dit is goed leesbaar en makkelijk aan te passen:
q=1
w=2
e=3
r=4

q,w,e,r = 1,2,3,4 # kan ook, maar is erg onoverzichtelijk en foutgevoelig
```

[15] Het is je vast al opgevallen dat bij programmeren standaard een font gekozen wordt waarbij alle letters even breed zijn, zoals `Courier New`.

6.2 Efficiëntie verbeteren

De meest gebruiksvriendelijke code is natuurlijk ook snel, je wilt liever een paar seconden wachten op je resultaten dan een paar minuten. Binnen de programmeerwereld staat Python bekend als een langzame taal. De snelheid van code hangt af van (a) de efficiëntie van de code zelf, (b) de snelheid waarmee het script wordt omgezet in machine code, en (c) de snelheid van computer processing unit (CPU).[16] Python is een langzame taal omdat de vertaling tussen het script en de instructies die naar de CPU gaan (in éénen en nullen) gebeurt tijdens het runnen van het script; dit maakt Python een *interpreting language*. Bij een *compiling language* zoals C of C++ moet je een script eerst *compilen* voordat het naar de CPU gestuurd kan worden. Dit compilen kost tijd, maar het daadwerkelijke runnen van de code gaat dan veel vlotter.

Het voordeel van een *scripting language* zoals Python is dat het makkelijker te leren is. Daarnaast kun je door de directe interpretatie snel en makkelijk kleine dingen veranderen in de code en meteen het effect daarvan zien. *"Python was not made to be fast, but to make developers fast."* - Sebastian Witowski (Software Engineer bij CERN)

Er zijn manieren om je Python code om te zetten in C-gecompileerde code, en zelfs om je Python code op meerdere processors tegelijk (parallel) te laten uitvoeren.[17] Meer over deze methodes zul je vanzelf tegen komen als je meer complexe simulaties of berekeningen gaat doen tijdens je studie of onderzoek. Deze manier van optimalisatie is echter geen onderdeel van deze beginnerscursus.

Over de snelheid van je CPU heb je ook niet direct invloed; behalve door het kopen van een nieuwe computer, snellere CPU, of gebruik maken van een externe CPU op een computercluster. Om je code sneller te laten uitvoeren kijken we in deze sectie dus alleen naar hoe we het script zélf efficiënter kunnen maken.

6.2.1 Efficiëntie meten

De makkelijkste manier om de snelheid van je code te meten is door de tijd te noteren aan het begin van je code, en nadat je code klaar is, en dan het verschil te nemen. Dit kan intern met het `time`.

Code 6.1: code-inc/w6/time˙vb1.py

```
# vind alle even getallen uit een random lijst op twee manieren
import numpy as np
import time

n = 1000000                                    # aantal random getallen
random_nrs = np.random.randint(100,size=n)     # n random int tussen 0 en 99

start_tijd1 = time.time()                      # start de tijd voor methode 1
even_nrs1 = []                                 # lege lijst voor even getallen
```

[16] Hier gebruiken we 'script' en 'code' als synoniemen met de betekenis: de text die jij schrijft of gebruikt om een geprogrammeerde taak uit te (laten) voeren. In werkelijkheid is een script maar een deel van de code, want er zitten veel (basis) instructies achter de schermen, die wel deel zijn van de code, maar die je niet terug ziet in je script.

[17] Normaal gebruikt Python maar 1 CPU-core tegelijk; ook als jouw computer dus 4 cores heeft gebruikt Python er maar 1, en voert dus alle instructies in serie uit.

```
for element in random_nrs:                    # loop door de hele lijst
    if element % 2 == 0:                      # als een getal even is
        even_nrs1.append(element)             # voeg het toe aan even_nrs1
eind_tijd1 = time.time()                      # stop de tijd voor methode 1
verstreken_tijd1 = eind_tijd1-start_tijd1     # bereken de verstreken tijd
print(f'methode 1 duurt: {verstreken_tijd1:.4f} sec')

start_tijd2 = time.time()                     # start de tijd voor methode 2
masker_even_getallen = random_nrs % 2 == 0    # masker voor even getallen
even_nrs2 = random_nrs[masker_even_getallen] # gebruik het masker
eind_tijd2 = time.time()                      # stop de tijd voor methode 2
verstreken_tijd2 = eind_tijd2-start_tijd2     # bereken de verstreken tijd
print(f'methode 2 duurt: {verstreken_tijd2:.4f} sec')
```

Deze methode werkt prima als je de totale run-tijd van je script wilt bepalen. Het is altijd handig als de gebruiker weet hoe lang een script ongeveer runt; als je dit gemeten hebt, zet dit dan ook in de beschrijving bovenaan het script.

Oefenopdracht

(a) Run bovenstaande code. Welke methode is het snelst?

(b) Kijk in de Variable explorer, of typ `type(naam_variabele)` rechtstreeks in de console (rechtsonder, na de input promt `In [getal]:`) om de types van de output van beide methodes te vergelijken; deze zijn verschillend.

Zet nu het resultaat van de snelste methode om in hetzelfde type als het langzaamste resultaat. (Hint: gebruik `a = list(a)` of `b = np.array(b)`.) Als je specifiek dit type (`list` of `np.array`) resultaat wilt, is dezelfde methode dan nog steeds het snelst?

(c) Run het script nu een aantal keer (je kunt dit automatiseren door er een loop omheen te zetten!). De runtime van beide methodes is elke keer een beetje anders. Bereken de gemiddelde runtime voor beide methodes.

6.2.2 Wanneer optimaliseren?

Optimaliseren door gebruik te maken van het soort testen die hierboven beschreven worden kost vrijwel altijd meer tijd dan dat het oplevert. Het optimaliseren als doel op zich kan een leuke puzzel zijn, maar is meestal bijzaak.

Wanneer is optimaliseren dan een goed idee? Vuistregel is:

> *"First make it work. Then make it right. Then make it fast"* - Kent Beck

Het belangrijkste is dat je code werkt; dat je code doet wat je wilt dat die doet (niet alleen geen errors geeft, maar ook dat het fysisch correct geïmplementeerd is) en dat de aannames en input correct zijn. Daarna kun je je code robuuster maken; bijvoorbeeld opvangen wat er gebeurd als er verkeerde input gegeven wordt, of als ergens onverhoopt een negatief getal uit komt terwijl dat fysisch niet kan. Hier valt ook onder dat je zorgt dat je code toekomst-bestendig is, dus dat er goed commentaar bij staat. Pas daarna, als allerlaatst, kun je de snelheid van je code proberen te verbeteren.

Voor een goede optimalisatie moet er eerst onderzocht worden waar de meeste tijd verloren gaat. De meeste tijdswinst kan vaak gewonnen worden op de knelpunten (*bottle neck*) die nu het meest tijd kosten. Binnen programmeren wordt het zoeken naar optimalisatie-knelpunten *profileren* genoemd. Er zijn verschillende functies en packages beschikbaar die helpen met het profileren van code; veel gebruikt zijn `cProfile` (vaak in combinatie met `pstats`) en `line_profiler`. Het gebruik van deze packages ligt buiten de leerdoelen van deze cursus; we geven hier slechts vast een paar handvatten en vuistregels.

Bij het zoeken naar *bottlenecks* kan het zijn dat niet CPU, maar bijvoorbeeld het lezen/schrijven van/naar geheugen (*disk I/O*) het meest tijd kost. Daarnaast is optimalisatie niet altijd gericht op de code het snelst uitvoeren, soms is het belangrijker dat er weinig werkgeheugen (RAM), hardeschijfruimte, netwerkverkeer, of zelfs energie gebruikt wordt. Optimalisatie in tijdsefficiëntie zijn dan niet altijd gewenst.

In sommige gevallen is optimaal gebruik van CPU wel belangrijk, bijvoorbeeld wanneer een model of berekening zo groot is dat deze op een extern CPU cluster uitgevoerd moet worden. In dat geval wil je wel zorgen dat je zo optimaal mogelijk gebruik maakt van de CPU-tijd die je toegewezen krijgt.

6.2.3 Standaard snelle oplossingen

Met het idee dat optimaliseren onnodig veel tijd kost in het achterhoofd; hier toch een aantal vuistregels die je aan kunt houden om de code die je schrijft in Python meteen wat sneller te maken:

- Leesbaarheid gaat altijd boven snelheid.
- Gebruik ingebouwde functies (dit kan alleen als je weet dat ze bestaan, dus als je iets nieuws wilt doen, kijk altijd even of er al een functie voor bestaat)
  - `len(x)` geeft de lengte van een variabele, dit is sneller dan zelf de lengte tellen in een loop
  - `np.mean(x)` geeft een gemiddelde van alle waardes in x.
  - `map(function,iterable)` voert de functie uit op elk element in de lijst (iterable). Dit is doorgaans sneller dan een loop.
  - Voor een uitgebreide lijst aan standaard functies zie: https://docs.python.org/3/library/functions.html
- Loops zijn langzaam; als je een loop kunt vervangen voor een bestaande functie of een direct statement is dit sneller.
- Gebruik numpy arrays voor berekeningen op het gehele array; hiermee kun je vaak loops ontwijken.
- Doe rekenkundige operaties binnen een functie i.p.v. een simpele versie van die functie meerdere malen aan te roepen. (*Let op:* dit gaat soms ten koste van de modulariteit, omdat dit je functie minder veelzijdig maakt.)
- Als iets is in minder regels (actieve) code kan, is het vaak het snelst om dat te doen. (*Let op:* dit gaat soms ten koste van de leesbaarheid, en dat is ongewenst; als je code slecht leesbaar is kost het je altijd meer tijd om die weer te ontcijferen, dan het je oplevert als die een paar milliseconden sneller runt.)

- Gebruik de nieuwste versie van python, en de nieuwste formats/technieken/functies.
- Tussentijds printen of wegschrijven naar file kost tijd. Tijdens de test-fase is het printen van tussentijdse resultaten erg nuttig; dit maakt het makkelijker om een mogelijke fout op te sporen. Als een stuk code eenmaal correct werkt is het een goed idee om die vele prints te deactiveren (door ze weg te halen of er tijdelijk een `#` voor te zetten). Bij lange scripts kan het tóch handig zijn om af en toe een print te laten staan (of toe te voegen) zodat de gebruiker weet waar het script mee bezig is en niet is vastgelopen.

Voorbeelden van snellere en langzamere code:

```
#%% check op True/False
if variable == True: #35.8ns
if variable is True: #28.7ns
if variable:         #20.6ns

if variable == False: #35.1ns
if variable is False: #26.9ns
if not variable:      #19.8ns
# Is dit voor jou de tijdswinst waard?
```

```
# 1000 operaties en 1 functie
def square(number):
    return number**2
squares = [square(i) for i in range(1000)]
# 1000x gemeten met time: 0.40 sec

def compute_squares():
    return [i**2 for i in range(1000)]
# 1000x gemeten met time: 0.31 sec
```

Oefenopdracht: efficiëntie

(a) Wat denk je dat sneller is: `a = np.arange(100)` of `b = [*range(100)]`? Test dit met `time`

6.3 Slotopdracht: ???

7 Meet je leefomgeving

Dit hoofdstuk is de klapper op de vuurpijl. De afgelopen weken heb je een kastje vol met sensoren mee naar huis gehad (tenzij je te ver weg woont). Dit kastje heeft lange tijd data verzameld van jouw leefomgeving, en het is tijd dat we die data gaan verwerken. Niet toevallig dat we Python combineren met het Meet je leefomgeving-project.

De sensorkastjes zijn op zichzelf best 'dom': het enige wat ze doen is de sensoren uitlezen en de data versturen. Het online dashboard waar alle waarden uit te lezen zijn, is ook niet al te slim: hij laat alleen op een leuke manier zien wat de huidige meetwaarden zijn. Maar aan jou nu de taak om van deze data chocola te gaan maken: hoe was de temperatuur de afgelopen weken, hoeveel fijnstof is er rond jouw huis, en hoeveel lawaai maken jouw buren nou werkelijk?

De eerste vijf hoofdstukken hebben we geleerd hoe Python werkt, hoe je `numpy` en `matplotlib` kan gebruiken, data kan importeren, bewerken en opslaan. Dat gaat nu allemaal samenkomen in de eindopdracht. Tot en met de vrijdag voor de toetsweek gaan we hiermee aan de slag.

Dit hoofdstuk bevat eigenlijk geen nieuwe stof meer: in principe kun je alles wat je tot nu toe geleerd hebt combineren om de eindopdracht te maken. De informatie in dit hoofdstuk geeft jou vooral de nodige kennis om te weten wat voor data je nou werkelijk bekijkt en wat voor plaatjes je moet maken, en een enkele truc die specifiek voor dit project fijn is om te weten. Lees dit globaal door voor je aan de slag gaat, maar gebruik dit vooral terwijl je bezig bent zodat je weet waar je naar kijkt.

7.1 De sensoren

Het kastje meet een flink aantal dingen. We behandelen de sensoren hier één voor één met de bijbehorende informatie zoals eenheden.

7.1.1 ADC: accuspanning

Het meest cruciale onderdeel is de accu. Hiervoor maken we geen gebruik van een fysieke sensor, maar van de ingebouwde

0) ADC: een Analog to Digital Converter. Een analoog signaal (de stroom) komt binnen bij de microcontroller, die een digitaal signaal berekent in Volt, die de spanning van de batterij aangeeft. Deze ligt typisch tussen de 4.7 en 3.4 V. Een vol opgeladen batterij levert 4.7 à 4.8V en naarmate die leegloopt gaat de spanning terug naar 3.4V waarna de batterij 'afslaat'.

7.1.2 BME280: temperatuur, luchtdruk en relatieve luchtvochtigheid

Deze sensor zit flink verstopt in het kastje: het is de middelste van de paarse printplaatjes onder aan de voorkant. Deze sensor meet drie dingen:

1) Temperatuur, in graden Celsius (°C). Als het goed is gaat de temperatuur uiteraard elke dag op en neer.

2) Luchtdruk, in hectopascal (hPa); de volledige naam is barometrische luchtdruk. Deze heeft meestal een waarde net iets boven de 1000 hPa, en kan door de dag of week heen wat op en neer gaan, maar niet zo hard als de temperatuur.

3) Relatieve luchtvochtigheid, in procenten (%). De luchtvochtigheid verschilt met het weer, en hangt een beetje af van de luchtdruk. Dit gaat dus ook wat op en neer, net als de luchtdruk. Een logische waarde hiervoor is rond de 50%.

7.1.3 MAX4466: geluidssterkte

Deze chip zit links onderaan en heeft een gaatje ervoor aan de voorkant. Dat is voor een goede meting wel nodig, want deze chip meet:

4) Geluidssterkte, in decibel (dB). De chip zit wat naar achteren in het kastje zodat hopelijk de wind niet al teveel herrie maakt, maar het lawaai nog wel goed te meten is. Maar als het stormt zal dat waarschijnlijk best wat storing opleveren in je meting: goed om rekening mee te houden! Er is niet zomaar een gemiddeld waarde, maar het zal ergens tussen de 20 en 80 decibel liggen waarschijnlijk.

7.1.4 TSL2591: lichtintensiteit

Was het de afgelopen week groeizaam weer of ontbrak de zon? Voor een boer en onderzoekers een relevante vraag, en zodoende is deze blauwe chip linksbovenop toegevoegd, en die meet:

5) Lichtintensiteit, in Lux (lx). Deze waarde kan erg hard heen en weer gaan: in een klaslokaal is de lichtintensiteit een paar honderd lux, buiten op een bewolkte dag ongeveer 1000 lux, in daglicht 10 tot 20 kilolux (klx) en in volle zon loopt het zelfs op tot 100 klx. Maar: deze chip zit onder wat donker plastic, dus een zonnige dag betekent hier niet direct 100 klx.

7.1.5 VEML6070: UV-intensiteit

Gebroederlijk naast de lichtsensor zit een kleine paarse chip. Deze meet:

6) UV-intensiteit. Deze heeft geen logische grootheid en eenheid, maar daarover later meer. Een logische waarde voor deze sensor is tussen de 0 en 1000. Ook deze sensor heeft 'last' van de donkere plastic cover.

7.1.6 CJMCU-811: VOC en CO_2-gehalte

Dit is de rechtse van de sensoren onder aan de voorkant met zijn eigen opening, en meet de luchtkwaliteit met betrekking tot:

7) VOC-gehalte in de lucht. VOC's zijn Vluchtige Organische Stoffen (*Volatile Organic Compounds*), gemeten in 'parts per billion' (ppb). Ze zijn verantwoordelijk voor smog, verzuring en hebben een (kleine) invloed op klimaatverandering. Ze worden geproduceerd door sommige planten en bomen, maar ook door mensen bij het gebruik van bijvoorbeeld verf en spuitbussen. Tot 250 ppb is een gezond gehalte; bij een aanhoudend niveau van 250-2000

ppb gedurende een aantal dagen is het opletten, en een gehalte van meer dan 2000 ppb is erg schadelijk voor de gezondheid.[18]

8) CO_2-gehalte in de lucht. Dit wordt gemeten in 'parts per million' (ppm): het aantal deeltjes dat per miljoen deeltjes in de lucht voorkomt. Gemiddeld lag dit in 2019 volgens Buienradar op 417 ppm. Uiteraard de grootste veroorzaker van het versterkte broeikaseffect. Hierover later ook nog een kanttekening.

7.1.7 SDS011: fijnstof

De grote metalen sensor onderin heeft ook een opening aan de voorkant, en een ventilator die je soms aan hoort slaan. Deze meet ook luchtkwaliteit, maar dan met betrekking tot fijnstof, in twee categoriën:

9) $PM_{2,5}$ deeltjes (*particulate matter*): deeltjes met een diameter van minder dan 2,5 μm (micrometer). Dat is het formaat van rookdeeltjes, of de diameter van een gemiddelde bacterie. De maximale toegestane waarde volgens de Wereldgezondheidorganisatie voor deze deeltjes is 10 $\mu g/m^3$. Dat is dan ook direct de eenheid waarin dit gemeten wordt.

10) PM_{10} deeltjes: deeltjes met een diameter van minder dan 10 μm. Dat is kleiner of gelijk aan het formaat van een mistdruppeltje. De maximaal toegestaan concentratie in de lucht volgens de WHO is 20 $\mu g/m^3$.

7.1.8 Ublox NEO-6M GPS6MU2: GPS

Niet altijd is duidelijk waar precies je sensorkastje zich bevindt, of soms wil je een leuk overzichtje hebben van alle locaties. Daarom is ook de grote blauwe module met het metaal-roze blok bijgevoegd. Deze meet:

11) GPS-locatie, in NB-OL coördinaten. Veenendaal en daarboven ligt in noorderbreedte 52, onder Veenendaal 51; van Utrecht tot voorbij Arnhem zitten we in oosterlengte 5. Binnenshuis of onder een dikker dak is er soms weinig bereik, en deze sensor moet vaak ook 'opstarten' voordat hij de locatie goed kan bepalen.

7.2 Handige informatie

Lees eerst de volgende paragraaf over de eindopdracht, en kom vervolgens hier terug voor hints over het opzetten van je script.

7.3 De eindopdracht

Na het harde(?) werken om 5 (of 6) hoofdstukken onder de knie te krijgen, ben je eindelijk klaar om de eindopdracht te maken. Deze opdracht luidt als volgt:

Verwerk de data van jouw meetkastje tot drie figuren met subplots gesorteerd per categorie: één figuur met alle data van de afgelopen maand, één figuur met alle data van de afgelopen week, en één figuur met de data van een dag naar keuze uit de afgelopen maand.

[18] Bron: https://www.airthings.com/what-is-voc