Dictaat bij studieonderdeel:

Data-acquisitie & Toegepaste analyse

NS-109B

2022/23

DATA-Py

Julius Instituut
Departement Natuur- en Sterrenkunde
Faculteit Bètawetenschappen
Universiteit Utrecht

Eindredactie: Nadine J. van der Heijden
Dank aan: Fons van Hees en Peter van Capel

Inhoudsopgave

1 Basisvaardigheden Python

Voor een natuurkundige zijn programmeervaardigheden onontbeerlijk. Of het nu gaat om het schrijven van een numerieke benadering van een analytisch onoplosbare differentiaalvergelijking, doorrekening van complexe modellen (van moleculaire interacties tot oceaanstromingen) of het analyseren van meetgegevens, er wordt van je verwacht dat je op eigen kracht code kunt schrijven of code die door anderen geschreven is kunt toepassen om dergelijke taken uit te voeren.

Een programmeertaal die binnen én buiten de natuurkunde 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 heeft een grote actieve gebruikersgroep. Dit betekent dat er via internet veel hulp, voorbeeldcode, modules en bibliotheken beschikbaar zijn.
- Het is enorm veelzijdig; het kan omgaan met code uit 'klassieke' talen als Fortran en C, kan gebruikt worden om apparatuur aan te sturen, of voor machine learning, ...

In het cursusonderdeel DATA-Py komen de essentiële eigenschappen van Python aan de orde. Je leert scripts, oftewel korte programma's, te schrijven, eerst met enkelvoudige opdrachten (bijvoorbeeld lijstbewerkingen, integraties, plotcommando's) en later met combinaties van enkelvoudige opdrachten. In de vakonderdelen DATA-V en DATA-P ga je deze scripts toepassen om gegevens of zelf gemeten data te plotten en analyseren. Later in het curriculum (Statistische Fysica, Kwantummechanica, verscheidene keuzevakken) ga je Python ook gebruiken om theoretische problemen numeriek te analyseren en simulaties te bouwen.

Als je het eenmaal in de vingers hebt, ga je de lol van programmeren en dataverwerking 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 wetenschappelijk programmeur kenmerken. Een verschil met applicaties 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 instituut echter wel begrepen, bewerkt en toegepast worden. Dat betekent dat veel van wat normaal afgedekt wordt door de gebruikersinterface, of via helpfuncties gebeurt, 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 wetenschappelijk programmeur programmeert daarom niet alleen foutloos, maar ook gestructureerd, modulair en begrijpelijk. Wat moet hiervoor concreet gebeuren?

- Volg conventies: gebruik variabelen namen die aansluiten bij symbolen voor natuurkundige grootheden en constanten. Bij de introductie van de verschillende onderdelen van Python zal waar van toepassing gebruikelijke conventies worden benoemd zodat je deze eigen kunt maken.

- Structureer je script: maak headers, definieer *secties* met lege regels en kopjes (zoals je ook in normale tekst doet om hem leesbaar te houden). Breek gecompliceerde opdrachten op in meer eenvoudige statements.

- Voeg begeleidend commentaar toe, vooral wanneer een stuk code zo complex is dat de functie ervan niet direct duidelijk is. Hoe dit kan worden gedaan in Python 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 de cursus DATA-Py 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.[1]

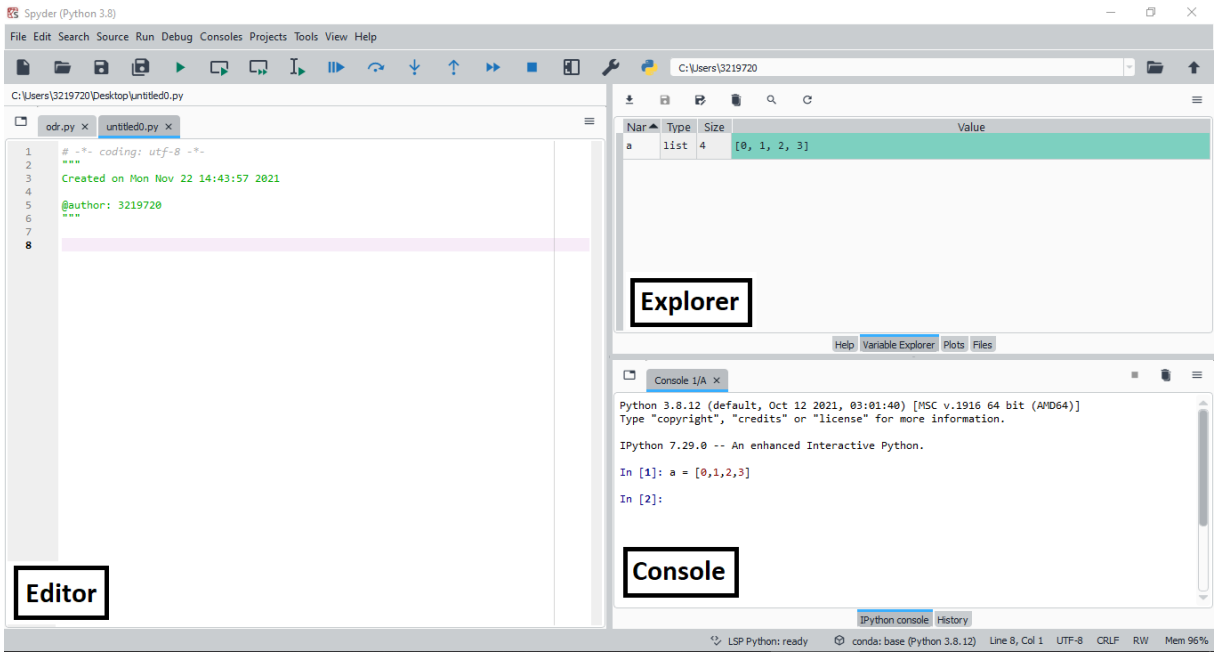
Wanneer je op dit punt Spyder nog niet hebt, verwijzen we je naar de handleiding op Blackboard voor de installatie van Spyder als onderdeel van Anaconda. De interface van Spyder wordt weergegeven in Figuur 1. Deze bestaat standaard uit één window met drie schermen, namelijk de Editor, Explorer en IPython console. Elk van deze heeft zijn eigen functie:

- Het scherm links is de Editor. Eigenlijk is dit niets anders dan een tekstverwerker. 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 een juiste syntax. 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. Daarnaast kan je hier losse regels code ook testen; handig als je bijvoorbeeld wilt checken dat je het juiste element opvraagt, of dat een functie doet wat je verwacht.

[1] Wanneer je al bekend bent met Python en jezelf een andere manier van programmeren hebt aangeleerd, ben je vrij om dat te blijven doen. Ook als je gaandeweg deze cursus een andere manier vindt om met Python te werken die je beter bevalt zou dat voor deze cursus niet uit moeten maken. Toch is een waarschuwing op z'n plaats omdat de docenten en assistenten wellicht niet bekend zijn met andere programma's en je minder goed kunnen helpen als je dan problemen tegenkomt.



Figuur 1: De interface van Spyder direct na het opstarten van het programma. Desgewenst kun je Spyder op Dark mode zetten onder Tools → Preferences → Appearance → Main interface → Interface theme, selecteer Dark en onder Synthax highlighthing theme, selecteer Spyder Dark.

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 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 `datapy-wc1-helloworld.py`.

1.1.1 Opdracht: 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 `datapy-wc1-helloworld.py` en laat vervolgens 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. In Spyder kun je met `#%%` secties maken in je script. Deze secties, genaamd 'cells' zijn dan los te runnen met de knop `Run current cell` (of `SHIFT+F5`) naast de groene 'play' knop bovenin het menu.[2]

[2]Dit knopje, en de knopjes er naast zien er in oudere versies van Spyder net iets anders uit, maar werken hetzelfde en staan wel op dezelfde plek.

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 hoeven hier geen wijzigingen in te worden aangebracht. De IPython console produceert informatie over welk script wordt gedraaid. Hieronder verschijnt de output van het script. Wat doet de functie `print()`?

Tijdens de cursus zul je regelmatig (voorbeeld)scripts zien in deze bundel. Soms staat hier een linkje boven wat je kunt gebruiken om het los te openen en op te slaan. Sla deze als je ze nodig hebt op in jouw eigen bestandsstructuur[3] en open deze vanuit Spyder op dezelfde manier waarop je dat in *MS Word* zou doen. 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, waar over typen vaak sneller is voor korte stukjes.

1.2 Eenvoudige rekenkundige operaties

Veel scripts bevatten basale rekenkundige operaties. Hiervoor gebruikt Python de normale tekens, namelijk: optellen (+), aftrekken (−), vermenigvuldigen (∗), delen (/) en machtsverheffen (∗∗). Een andere rekenkundige operatie die vaak voorkomt bij programmeren is de *modulus* (%). Het eerste getal in de operatie is de *input* en het tweede getal is de *bovengrens*; daarna begin je opnieuw met tellen. 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 gehele getallen.

```
1 + 1
2 - 2
3*3
4/4
5**5
```

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.

Wanneer je een samengestelde rekenkundige bewerking invoert in Python zal deze worden uitgevoerd, tenminste als je het script runt. Echter, dit betekent niet dat de berekende waarde ook op het scherm wordt getoond. Daarvoor wordt het commando **`print`** gebruikt.

Bijvoorbeeld:

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

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

1.2.1 Opdracht: Python als rekenmachine

Bereken de hoogte y van een bal 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:

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

Kies voor de valversnelling g de waarde $9.81\,\mathrm{m/s^2}$.

[3] Rechtsboven in Spyder staat de map waar je huidge script staat opgeslagen en waar ook gezocht wordt naar data die je wilt importeren.

1.3 Variabelen

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.

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 numerieke waarden van a, b, c, x in de formule meerdere keren aanpassen en het Python script meerdere malen uitvoeren voor de verschillende uitkomsten. Veel overzichtelijker is echter de volgende aanpak, 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 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)
```

Namen voor variabelen kunnen (bijna) vrij gekozen worden. De enige voorwaarden zijn dat de namen moeten beginnen met een letter, geen spatie of wiskundige operatie (+ * / etc.) mogen bevatten,[4] en niet gelijk mogen zijn aan woorden die in Python al een betekenis hebben, zoals: **print**, **and**, **def**, **from**, **try**, etc.

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 gebruikte symbolen voor staan als een script zonder context gelezen wordt. Mede daarom kan aan een script *commentaar* worden toegevoegd. 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 bijv. op de volgende manier verrijkt worden:

```
# abc-formule voor een parabool; er is geen achterliggende theorie.
a = 1 # parameter 1, zie beschrijving in reader DATA-Py
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.

[4] 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.

1.3.1 Opdracht: Gebruik van variabelen

Bereken opnieuw de hoogte van de bal uit voorgaande 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.

1.4 Datatypen

In alle bovenstaande 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 operaties 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` (dus zonder de decimale punt). Er kan ook expliciet worden aangegeven dat een variabele als een integer moet aangemerkt door `a = int(x)`. Hierbij kan `x` ook een reëel getal zijn, maar bij de toekenning worden de getallen achter de komma verwijderd. In programmeer-jargon heet dit *casting*. Let op: dit is niet hetzelfde als afronden op gehele getallen!

float Getallen met decimalen: `2.3 4.62 100.00`

Het objecttype voor reële 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 gecast naar (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.

bool Logische waarde: `True` of `False`

Een **boolean** object, dat we later in het hoofdstuk zullen tegenkomen, is een binair object dat slecht twee waarden kan hebben, te weten `True` of `False`. Het type boolean wordt gebruikt in logische operaties.

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.

complex Complex getal met een reëel en imaginair deel.

Complexe getallen hebben het objecttype **complex** in Python. Zulke getallen worden gedefinieerd als bijv. `a = 1. + 2.j`. Er wordt dus een `j` gebruikt in plaats van de in de wiskunde meer gebruikelijke `i`. Ook kun je een complex getal `a` een waarde geven door: `a = complex(r,i)`. Hier is `r` het reële deel en `i` het imaginaire deel van het complexe getal.

Het reële deel van een complex getal `a` is een **`float`** en wordt verkregen met `r = a.real`; het imaginaire deel is eveneens een **`float`** en wordt verkregen met `i = a.imag`.

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.

tuple Geordende onveranderlijke reeks objecten: `(10, "hello", 200.3)`

Let op het verschil tussen een **`tuple`** en een **`list`** is dat een list gewijzigd kan worden. Als je dus per ongeluk de verkeerde haakjes gebruikt (`()` of `[]`) dan kan het zijn dat je script niet doet wat je wilt.

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 of een type wel of niet geordend is. Bij geordende reeksen staat de volgorde van de elementen vast; hier kun je dus met een index een element uit aanroepen. Met **`type`**`(naam_variabele)` kun je achterhalen met welk type je te maken hebt. Dit staat ook in de variable explorer.

1.5 Functies

In een script kan het gebruik van variabelen en de operaties die ermee worden uitgevoerd worden geautomatiseerd om ze later makkelijk vaker te kunnen gebruiken, door het definiëren van een *functie*.

Een functie is een Python *object* dat aan de hand van invoervariabelen één of meerdere bewerkingen kan uitvoeren. Een Python functie is dus veel algemener dan een wiskundige functie zoals bijv. $\sin(x)$.

Python heeft een veelvoud aan standaard *functies*, waarvan we er al een paar hebben gebruikt, zoals **`print`**`()` en **`min`**`()`. Functies zijn te herkennen aan dat ze hun input krijgen binnen ronde haakjes. Binnen python zijn al heel veel functies standaard beschikbaar. Daarnaast kan je met packages nog eens heel veel extra functies importeren. Hieronder staan een aantal handige en veelgebruikte functies die standaard in python zitten:

```
# rekenkundige functies
abs()   # geeft de absolute waarde van de input
min()   # geeft het kleinste element
max()   # geeft het grootste element
sum()   # geeft de som van alle elementen

# type onzet functies
str()   # maakt een str van de input
float() # maakt een float van de input
int()   # maakt een interger van de input

# overige handige functies
len()   # geeft de lengte van de input
print() # print de input in de console
```

```
type()  # geeft het type van de variabele
input() # vraagt om gebruiker input via toetsenbord (bevestig met enter)
sorted() # geeft de gesorteerde input terug in een list

# functies die vooral bij loops handig blijken
range() # geeft een serie getallen
enumerate() # geeft een lijst van tupples met de index en de waarde van de input
```

1.5.1 Opdracht: Data types toewijzen

Door handig gebruik te maken van bestaande functies kun je gemakklijk het laagste of hoogste getal uit een lijst vinden. Print de hoogste en laagste waarde van de volgende lijsten.

Code 1: code-inc/w1/listtypes.py

```
lijst_a = [4,1,3,8,2]
lijst_b = ['4','1','3','8','2','x']
lijst_c = [True,False]
lijst_d = [4.,1.,3.,8.,2.]
lijst_e = 'hello world'
```

Python gebruikt charatcercodes om tekst letter voor letter op alfabetische volgorde te zetten, vandaar dat in een lijst met alleen elementen van het type `str` de `x` als 'hoogste' wordt aangemerkt. Je kunt kijken wat hier achter zit met `ord()`. Porbeer bijvoorbeeld `ord('&')`. Komt het & -teken voor of na het alfabet? En hebben hoofdletters een effect op de alfabetische sortering?

Kijk ook in de Variable Explorer of je kunt vinden om welk type variabele het gaat. Kun je dit ook het type van een variable in een lijst zien? (Hint: ja! dubbelklik op de variabele waar je meer van wilt weten.)

1.5.2 Opdracht: Functies testen

Gebruik de functies hierboven om alle positieve even getallen tot een door de gebruiker getypt getal bij elkaar op te tellen. Print je antwoord zo:

```
De som van de positieve even getallen onder de {gebruiker input}  is {som}.
```

Het script wat je in opdracht 1.5.2 hebt geschreven zou ook zélf een functie kunnen zijn; bijvoorbeeld `even_som()` met als input het maximale getal, en als output de som van alle even getallen tot dat maximum. Nu lijkt dit op het eerste gezicht niet een erg nuttige functie, maar dat je zelf een functie kunt schrijven die je daarna zelf kan gebruiken is wel erg handig. We gaan hier dieper op in in hoofdstuk 5.

Voor nu is het belangrijk om te weten dat functies hun input in de juiste vorm moeten ontvangen, en dat je moet weten (of achterhalen) welk type output een functie geeft zodat je hiermee verder kunt werken.

1.6 Formatteren van tekst

Zoals eerder genoemd moet de functie **print()** expliciet 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. Merk op dat de tekst nu eigenlijk stukjes **str** *object* zijn, die net zo worden behandeld als vooraf gespecifiëerde variabelen.

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

Een andere manier om alfanumerieke tekst te tonen op je beeldscherm is te bewerkstelligen met behulp van *'f-strings'*. Je hoeft dan veel minder komma's en quotes te gebruiken en hebt controle over de wijze van weergeven (het formatteren). Het is dan niet langer nodig verschillende delen te scheiden met komma's, maar je gebruikt een enkele `string`. Hierin geef je met accolades rond de naam van je variabele aan dat deze geïnterpreteerd moet worden als variabele en niet als onderdeel van de string. Het handige van *f-strings* is dat je dit kunt doen op de plek in de zin waar je wilt.

De manier van schrijven op je scherm wordt normaal gesproken bepaald door het type variabele. Wil je echter dat de gegevens anders worden weergegeven, dan kun je binnen de accolades met een *code* aangeven hoe de uitvoer eruit moet zien. Deze *codes* beginnen met een dubbele punt gevolgd door een aantal leestekens. Zie tabel 1 voor een aantal mogelijkheden.[5]

Tabel 1: Codes voor het weergeven van gegevens.

:d	integer
:e	wetenschappelijke notatie
:f	decimale notatie (met 6 decimalen)
:.xf	decimale notatie met x decimalen
:y.xf	decimale notatie met x decimalen en met minimaal y posities
:g	compacte notatie (Python kiest e of f)
:s	string

1.6.1 Opdracht: Oefenen met het precies formatteren van tekst

[5] In oudere code zie je soms een soortgelijke manier van *formatten*, maar dan met **%** in plaats van **:**. De manier die je hier aan leert hoort bij Python 3. Hoewel de oude versie ook nog werkt zijn de moderne *f-strings* handiger in gebruik.

Code 2: code-inc/w1/fstr.py

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

Verander de wijze 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, wat gebeurt er hierna?). Probeer behalve het aantal decimalen te variëren ook te spelen met het type door t en y af te drukken als string of integer! Noteer wat er dan gebeurt in een *comment* in je script.

1.7 Slicing en indexing

Meestal heb je bij data analyse de hele lijst van data nodig, maar soms kan het handig zijn om een stukje te selecteren (bijvoorbeeld de staart er af te knippen). Ook bijvoorbeeld bij het vinden van piek-posities, of het maken van simulaties is het handig als je één of meerdere elementen kunt selecteren. Dit kan met de index (voor een enkel element) of met slicing (voor een serie, oftewel een slice van elementen).

In geordende lijsten kun je elementen aanroepen met hun index. Let op: python begint met tellen bij `0`, dus het eerste element uit een lijst heeft index `0`.

```
a = [2,3,5,1,6,3]
print(a[2])             # print het element met index 2

b = 'hallo'
print(b[1])             # print element met index 1
```

Bij slicing gebruik je de indexen, maar kun je ook een start, eind en stapgrootte aangeven. Dit wordt in python aangegeven met: `var_name[start:stop:step]`. Een `:` kan ook siginaleren dat je alle elementen van die lijst wilt hebben.

```
c = [0,1,2,3,4,5,6,7,8,9,10]
print(c[3:7])   # is de stop een tot of een t/m?
print(c[::-1])  # wat doet een stap van -1?

d = [[0,1,2],[3,4,5],[6,7,8]]
print(d[0][-1]) # lijsten in lijsten, en een negatieve index
```

1.7.1 Opdracht: index en slice puzzels

```
start_lijst = [0,1,2,3,4,5,6,7,8,9,10]
a = #... schijf hier je slicing/index oplossing
```

(a) Zet je lijst in omgekeerde volgorde in een nieuwe variabele.

(b) Verander de 5 in een 50.

(c) Verwijder alle oneven getallen.

(d) Geef de lijst met het kwadraat van elk getal in de start lijst. Nu is dit nogal omslachtig. In Hoofdstuk 2 zul je een manier leren om opdrachten te herhalen, en in Hoofdstuk **??** leer je met `numpy` dit soort rekenkundige operaties nog makkelijker uit te voeren.

2 Loops

Binnen Python bestaan manieren om functies of operaties automatisch veelvuldig uit te voeren, en om te automatiseren onder welke voorwaarden of hoe vaak iets uitgevoerd of herhaald moet worden.

2.1 Boolean expressions

Het specificeren van voorwaarden gebeurt met het testen van condities. Hebben we het gewenste aantal herhalingen al bereikt? Is deze functie van toepassing? Het antwoord op dit type vragen is altijd 'ja' of 'nee'. De algemene term voor deze conditie is *Boolean expression*. De uitkomst van zo'n conditie heeft slechts twee mogelijkheden, namelijk `True` of `False`. Vaak wordt zo'n conditie 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 conditie. Daarnaast kunnen Boolean expressions ook worden gecombineerd door het gebruik van **`and`** waarbij de expressie alleen `True` zal opleveren als aan beide condities wordt voldaan of **`or`** waarbij de expressie `True` zal opleveren als aan minstens één van de twee condities wordt voldaan.

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

Het is in het algemeen niet nodig om ronde haakjes te gebruiken. Alleen in het laatste voorbeeld is dat wel essentieel. De reden is dat `not a > b or c > d` wordt geïnterpreteerd als `(not a > b) or (c > d)`, en dat is iets heel anders dan `(not (a > b or c > d)`.

2.1.1 Opdracht: 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.
- Laat vervolgens zien dat je complexe getallen niet zomaar kunt ordenen; dus: `3+4j < 4+3j` geeft een foutmelding. Natuurlijk kun je wel een ordening aanbrengen door van twee complexe getallen de reële delen met elkaar te vergelijken, of de absolute waarde.
- Maak tot slot je eigen exclusive or script 'xor' met twee start variabelen en die als resultaat `True` print als één van de argumenten `True` is, en de ander `False`. In alle andere gevallen print je `False`. Test je script door alle mogelijke combinaties uit te proberen. (Dat zijn er maar 4.)

2.2 While

Een zogenaamde **while** loop kan gebruikt worden om een bepaald stuk code te herhalen zolang er aan een vooraf gestelde conditie 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. In termen van vergelijking (1), zolang $t \leq T$. Hoe zo'n loop kan worden geïmplementeerd is te zien in onderstaand voorbeeld.

Code 3: code-inc/w2/while˙ex.py

```
y0 = 0     # beginhoogte
v0 = 25    # beginsnelheid
g = 9.81   # gravitatieconstante
t = 0.     # begintijd
dt = 0.25 # tijdstap

while t <= 5:
    y = y0 + v0*t - 0.5*g*t**2
    print(t, y)

    t = t + dt
```

Net als voorheen definiëren we de benodigde variabelen y_0, v_0 en g. Daarnaast maken we een variabele aan voor de tijd t (de 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*[6] 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 er niet meer aan de conditie `t <= 5` wordt voldaan. De regels code binnen de **while** *loop* moeten beginnen met een *indent* om aan te geven dat die regels onderdeel zijn van één en dezelfde *loop*. In Spyder gaat dit automatisch wanneer je na een regel die begint met **while** een enter typt.

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

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

2.3 If

Een tweede soort constructie is de zogenaamde *if - else* constructie. Aan de hand van een gespecificeerde voorwaarde (*conditie*) wordt een bepaald deel code wel of niet uitgevoerd. Een variant hierop is de *if - else if* constructie, waarbij er meer dan 2 mogelijkheden voorkomen. We zullen deze constructie illustreren aan de hand van een *hat-functie*:

$$N(x) = \begin{cases} 0, & x < 0 \\ x, & 0 \leq x < 1 \\ 2 - x, & 1 \leq x < 2 \\ 0, & x \geq 2 \end{cases} \tag{2}$$

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 Python wordt *else if* afgekort tot **elif**.

Code 4: code-inc/w1/wk1-hat.py

```
x = 0.2
if x < 0:
    print(0)
elif 0 <= x < 1:
    print(x)
elif 1 <= x < 2:
    print(2-x)
elif x >= 2:
    print(0)
```

In dit voorbeeld kan de conditie `0 <= x < 1` net zo goed vervangen worden door `x < 1`, omdat als de situatie `x < 0` al is afgedekt door de voorafgaande **if**. Evenzo kan de regel **elif** `x >= 2:` net zo goed vervangen worden door **else:**, om aan te geven wat er moet gebeuren als alle voorafgaande condities niet wordt voldaan. Door af te sluiten met een **else:** voorkom je dus dat een **if** helemaal niets doet omdat geen enkele conditie van toepassing is.

2.3.1 Opdracht: Gebruik van 'if' en/of 'while'

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

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

  Als het opgegeven getal inderdaad positief is wordt de wortel[7] van dat getal geprint en wordt er om nog een getal gevraagd. Is het opgegeven getal daarentegen negatief dan wordt de wortel niet uitgerekend. Er wordt daarentegen de tekst `"Dombo: Positief svp!"` afgedrukt en vervolgens vraagt het script (heel beleefd) nogmaals om een positief getal. Pas als je de waarde 0 opgeeft stopt het script.

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

2.4 For

Als je een operatie een bepaald 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 *iterable*. Elke **while**-loop is om te schrijven in een **for**-loop, maar soms *voelt* de één natuurlijker dan de ander. 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)
```

In hoofdstuk 3 zul je nog kennis maken met een speciaal soort lijsten, `numpy`-arrays, die uitermate geschikt zijn voor data-analyse. Het rekenen met `numpy`-arrays is uitermate efficiënt (er is weinig code nodig) en snel (er is weinig computertijd nodig).

De algemene Python objecten van het type **list** zijn niet erg geschikt voor data analyse, maar wel voor allerlei administratieve taken. Ze worden vaak gebruikt als een verzameling elementen die kan worden verwerkt in een **for**-loop. De standaard Python lijsten zijn opgebouwd uit een aantal elementen.

Er kan met Python lijsten niet eenvoudig groepsgewijs worden gewerkt. Hierdoor moet je Python lijsten element voor element afwerken.

Het aanroepen van een deel van een list (*slicing*), werkt voor 1 dimensie zoals je gezien hebt, maar bij meerdere dimensies moeten de dimensies 1-voor-1 worden gespecificeerd, bijvoorbeeld: het element $x_{2,3}$ in een matrix moet je opgeven als `x[1][2]`.

Bij de eerder behandelde **while** en **if** constructies werd een handeling herhaald op basis van de waarde van een bepaalde Boolean variabele: `True` of `False`.

for-loops zijn in dit aspect anders, aangezien ze een handeling toepassen op elk element van een lijst, zoals te zien in het eenvoudige voorbeeld hier onder. Bij het aanroepen van de **for**-loop wordt een lokale variabele gedefinieerd die keer op keer de waarde van een element uit een lijst bevat. Hieronder is de variabele `elem`. Wanneer de code van de *loop* is uitgevoerd voor een enkel element, wordt de waarde geüpdatet naar het volgende element. Dit wordt gedaan totdat de gehele lijst, van begin tot eind, is behandeld.

Code 5: 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
```

Natuurlijk kunnen `np.array`'s ook worden gebruikt binnen een **for**-loop. Dan is het bewerken van de elementen, zoals het vermenigvuldigen met 4, ineens wel een normale rekenkundige bewerking.

2.4.1 Opdracht: Spelen met lijsten en printen binnen for-loop

- Maak een lijst l_1 met 100 elementen: de gehele getallen 0 tot en met 99. Vermenigvuldig elk getal in die lijst met 100; dit definieert een 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 matrix 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 maar 3 elementen heeft!

  Noteer de verschillen in de afgedrukte output.

In sommige gevallen wil je zowel de index als de waarde van elk element kunnen gebruiken binnen een **for**-loop. Hieronder staan een aantal verschillende manieren om een **for**-loop te gebruiken. Kijk of je de verschillen begrijpt, en denk bij het gebruik van een **for**-loop na welke versie het handigst is voor die situatie.

Code 6: code-inc/w2/for˙loops.py

```
a = 'hello world!'  # str is ook een iterable, net als een lijst
b = []              # lege lijst
for elem in a:      # gewone for-loop; doe iets met elk element
    b.append(elem)  # plak elk element los aan de lijst b vast

for i in range(len(b)):    # for-loop met alleen indexen
    print(f'index is {i}.')

for index, elem in enumerate(b): # for-loop met enumerate
    print(f'letter {elem} heeft index {index}.')
```

2.4.2 Opdracht: Conversie 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

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

2.5 List comprehension

Een wel heel efficiënte manier van het gebruiken van een **for**-loop in Python is *listcomprehension*. Hierbij staat alles van de **for**-loop in één regel code. Hiermee kunnen de loops in het vorige voorbeeld herschreven worden naar:

```
c = [elem for elem in 'hello world!']
d = [i for i in range(len(c))]
e = [(elem, index) for index, elem in enumerate(c)]
```

Deze manier van loops gebruiken kan regels besparen, maar komt niet altijd de leesbaarheid ten goede. Let ook op dat het resultaat altijd in een lijst staat, omsloten door `[]`-haakjes.

2.5.1 Opdracht: Fibonacci

Een Fibonacci reeks is een reeks waarin elk getal de som is van zijn twee voorgangers. Zie https://en.wikipedia.org/wiki/Fibonacci_number. De Fibonacci reeks is een Fibonacci reeks die begint met de getallen 1 en 1, hoewel sommigen vinden dat de begingetallen 0 en 1 moeten zijn.

Schrijf een script dat aan de hand van twee willekeurige getallen een Fibonacci reeks kan construeren en toont op het beeldscherm. Test de werking van het script met de begingetallen 1, 1 (`integers`), maar ook 1.5 en 2.5 (`floats`).

Stopt je script niet vanzelf? Zoek dan een middel om (in Spyder) je script toch netjes te beëindigen zonder Spyder zelf af te sluiten. Zorg er daarna voor dat je script wel stopt.

2.5.2 Opdracht: 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.

3 Modules

In dit deel 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 was Python tot niet veel meer in staat dan het manipuleren van strings. De twee modules `numpy` en `matplotlib` zijn cruciaal voor het doen van databewerking in de meer algemene zin.

- `numpy` is een module om snel en eenvoudig bewerkingen te doen op numerieke data.
- `matplotlib` is een module om data in beeld te brengen (te "plotten").

Het importeren van een module gebeurt met het commando **`import`**. Het is gebruikelijk om `numpy` af te korten tot `np`. De meest gebruikte submodule van `matplotlib` is `pyplot` en wordt vaak afgekort geïmporteert als `plt`.[8] Modules kunnen geïmporteerd worden onder een alias op de volgende manier:

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

Op deze manier worden `numpy` en `matplotlib.pyplot` geïmporteerd onder de gebruikelijke afkortingen `np` en `plt`. De functionaliteit is verder hetzelfde, maar het is handig, en in deze cursus zelfs verwacht, om de geldende conventies aan te houden. Vanaf dit punt wordt naar `numpy` verwezen als `np`. Om functies uit een module te gebruiken wordt het alias van de module als prefix gebruikt, gevolgd door een punt, en dan het commando(= de naam van de functie) uit de module.

3.1 Arrays en array-manipulatie met numpy

Een ééndimensionale array, een verzameling van getallen van hetzelfde datatype, wordt bijvoorbeeld aangeroepen met `np.array()`. Dit in tegenstelling met Python-lijsten, waar datatypes door elkaar heen gebruikt kunnen worden. Een `np.array()` is zo ingericht dat deze maar één datatype kan bevatten.

Er zijn verschillende manieren om een `np.array()` aan te maken. De meest voor de hand liggende manier is door een bestaande Python-lijst te converteren naar een `np.array()`. Een Python-lijst is te herkennen aan de blokhaken; de elementen van de lijst zijn gescheiden door komma's. In het codeblok hieronder staan een aantal manieren om één-dimensionale arrays aan te maken die ook werken als je nog helemaal geen lijst hebt.

Code 7: code-inc/w2/np˙vb1.py

```
import numpy as np

# De volgende voorbeelden leveren allemaal integer arrays op.
array1a = np.array([0,1,2,3,4,5])  # Vanuit een Python-lijst
array1b = np.array(range(6))       # maak een np.array van een range
array1c = np.arange(6)             # maak direct een np.array
# Merk op dat het resultaat van allen hetzelfde is, maar het laatste
```

[8] Het wordt ten sterkste aangeraden om deze gebruikelijke conventies te volgen

```
# commando kost minder typewerk.

## De volgende voorbeelden leveren allemaal arrays met floats op.
nullen    = np.zeros(6)              # Array gevuld met 6 nullen (flts)
enen      = np.ones(6)               # Array gevuld met 6 enen (flts)
random    = np.random.rand(6)        # Array gevuld met 6 random numbers
                                     # in het bereik [0,1)

# Om arrays met integers te verkrijgen is het nodig om het datatype te
# forceren.
nullen_i  = np.zeros(6,dtype=int)    # Een array gevuld met nullen (ints)
enen_i    = np.ones(6,dtype=int)     # Een array gevuld met enen (ints)

# Om een 'grid' te maken tussen twee waarden
# kunnen deze commando's 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 equidistante getallen
```

Merk op dat `np.arange()` het eindgetal (`5.`) niet meeneemt, `np.linspace()` doet dit wel, dus `grid1` en `grid2` zijn verschillend. Rekenen met arrays is super makkelijk. Wanneer twee arrays dezelfde vorm hebben, kan men ze elementsgewijs (net als vectoren) optellen door `+` te gebruiken. Wanneer deze niet dezelfde vorm hebben krijg je een foutmelding. Alle wiskundige bewerkingen (`+,-,/,*,**`) werken op deze manier.[9]

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 numpy variabele vast te plakken.

Code 8: 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.; merk 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)

# Enkele eigenschappen van een array; merk ontbreken van () op aan het eind
ta_lengte = test_array.shape      # Geeft zgn. tuple met lengte van 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)
```

[9] Merk op dat wanneer je twee arrays met integers op elkaar deelt, het resultaat een array met floats is.

3.1.1 Opdracht: Maken en bewerken van numpy arrays

(a) 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?[10]

(b) 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.

(c) Bereken het gemiddelde en de som van de k^k-array in onderdeel (a).

3.2 Selecteren en maskers

Het is mogelijk om bij het ophalen van elementen uit arrays criteria op te geven, aan de hand van een *masker* en met deze criteria specifieke elementen te wijzigen. De werking van zo'n masker wordt uitgelegd aan de hand van het onderstaande voorbeeld.

Code 9: code-inc/w2/np˙vb4.py

```
import numpy as np
# Gebruik van een test_array om de werking van een masker uit te leggen
test_array = np.array([12, 1, 7, 8, 4, 3])
print(" test_array (origineel) = ", test_array)

# Aanmaken van een masker met dezelfde vorm als test_array
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]
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)
```

Maskers kunnen de waarden `True` en `False` bevatten. Daarom kunnen hierop operaties uitgevoerd worden met de logische functies van numpy zoals:

```
np.logical_and()
np.logical_or()
```

[10] 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.

```
np.logical_not()
np.logical_xor()
```

Code 10: code-inc/w3/masks˙vb1.py

```
## Dit script demonstreert het gebruik van maskers
## in selectie van data. In dit geval de resultaten van een tentamen.

# Lijsten/arrays zelf invullen svp
naam = np.array([]) # naam: Bevat naam van studenten
stno = np.array([]) # stno: Bevat studentnummer studenten.
opg1 = np.array([]) # opg1: punten voor opgave 1 etc.
opg2 = np.array([])
opg3 = np.array([])

## Alle arrays zijn 1-dimensionaal en op dezelfde manier georderd.
## Voor dit tentamen waren 3*10 punten te verdienen. Met 18
## punten heeft de kandidaat een voldoende.
points = opg1 + opg2 + opg3

vol_masker = points >= 18  # vol staat voor voldoende

# Nu kan selectie plaatsvinden
vol_naam = naam[vol_masker]
vol_stno = stno[vol_masker]
vol_cijfer = points[vol_masker]/3

print("Studenten met een voldoende:")
for it in np.arange(len(vol_naam)):
        print("%s (studnr %s ) heeft %s punten" %
       (vol_naam[it],   # afbreken van lange regels
        vol_stno[it],
        vol_cijfer[it]))
# Het % teken wordt hier gebruikt om argument specifiers aan te duiden.
# Dit is een handige manier om meerdere waarden in een string te plakken.
# Je kunt hier zelf mee experimenteren, in hoofdstuk 4 staat meer over het
# opmaken van text binnen Python.
```

3.2.1 Opdracht: Gebruik van maskers

(a) Vul in bovenstaand script zelf namen en getallen in voor tenminste 3 studenten. Zorg er wel voor dat er minimaal één is met minder dan 18 punten, minimaal één met 18 punten of meer, maar wel met onvoldoendes voor één of meerdere opgaven en ook minimaal één met alleen maar voldoendes.

(b) Wijzig de code, zonder dat het resultaat verandert, door bijv. `np.mean()` te gebruiken waardoor je dus niet eerst cijfers bij elkaar hoeft op te tellen om ze daarna weer door 3 (=het aantal opgaven) te moeten delen. Je kunt dan direct selecteren op studenten met

een 6 of hoger. Pas ook het commentaar in je script aan zodat het klopt met de gewijzigde situatie!

(c) Breid het bovenstaande voorbeeld uit zodat studenten een voldoende (een 6 of hoger) moeten halen voor elke opdracht om in een lijstje met ijverige studenten terecht te komen.

3.3 Reshaping

Wanneer men werkt met multidimensionale lijsten kan het handig zijn om de vorm van de array aan te passen. Een eerste stap hierin kan zijn om er een 1D array van te maken met `np.ravel()` door alle elementen achter elkaar te zetten, beginnend bij de laagste index.

Het commando `np.reshape()` ordert de elementen opnieuw en construeert een multidimensionale array met afmetingen afhankelijk van de opgegeven parameters. Bij het aanroepen van `np.reshape()` is het mogelijk om voor 1 dimensie de wildcard -1 op te geven. De afmeting voor deze dimensie wordt dan bepaald uit de andere opgegeven waarden.[11]

3.3.1 Opdracht: sorteren en reshapen

Laat `y = np.arange(30); np.random.shuffle(y)`. Verifieer de syntax en ga na wat de werking van `np.random.shuffle()` is. Waarom werkt `y = np.random.shuffle(np.arange(30))` niet? Schrijf een script wat `y` verandert in een gesorteerde 2 bij 15 array, waar de bovenste rij de 15 oneven getallen gesorteerd bevat en de onderste rij de 15 even getallen. Gebruik `np.sort()` om de lijst weer te sorteren, en maak slim gebruik van `np.reshape()` om de even en oneven getallen te scheiden.

Code 11: code-inc/w3/ravel˙vb1.py

```
import numpy as np
# Gebruik van ravel, reshape en T (=transpose/spiegelen)
a = np.array([[1,3,5], [2,4,6]])
at = a.T                          #  = [[1,2], [3,4], [5,6]]
b = a.ravel()                     #  = [1,3,5,2,4,6]
c = a.T.ravel()                   #  = [1,2,3,4,5,6]
a2 = c.reshape(2,3)               #  = [[1,2,3], [4,5,6]]
```

3.4 Data-visualisatie met matplotlib

In dit deel bekijken we hoe we data kunnen visualiseren. We gebruiken hiervoor het `pyplot` deelpakket uit het pakket `matplotlib` (MATLab Plotting Library). Hieronder een voorbeeld om een sinus in beeld te brengen.

Code 12: code-inc/w2/plt˙vb1.py

```
import numpy as np
import matplotlib.pyplot as plt
## Maak 2 arrays, (x,sin(x))
x = np.linspace(0.,2*np.pi,num=100)
```

[11] Deze commando's werken alleen op `numpy array`'s, niet op objecten van bijvoorbeeld type `list`.

```
y = np.sin(x)                          # Elementsgewijs wordt sin(x) uitgerekend

plt.figure()                           # begin een nieuwe figuur
plt.plot(x,y)                          # Plot punten (x,y)
```

3.4.1 Opdracht: sinus en cosinus plotten

(a) Imiteer het bovenstaande voorbeeld. Varieer het aantal elementen in `x`. Wat gebeurt er wanneer het aantal elementen in de lijst klein is? Voeg een cosinus toe door een `y2` te definiëren en een extra `plt.plot()` toe te voegen.

(b) Maak een tweede plaatje (figuur, door in hetzelfde script een tweede `plt.figure()` toe te voegen) waar `y` en `y2` tegen elkaar uitgezet worden.

(c) De x-as van het eerste plaatje loopt nu tot 7. Pas met `plt.xlim(x_min,x_max)` het bereik van de x-as aan. Doe hetzelfde voor de y-as met `plt.ylim(y_min,y_max)`. Maak het bereik van de x-as 0 tot 2π en voor de y-as van -1.2 tot 1.2.

(d) In het tweede plaatje (wat een cirkel zou moeten tonen) is de aspect ratio (verhouding van lengte x- en y-as) niet vastgesteld. Zet de aspectratio op 1 met `plt.gca().set_aspect('equal')`.

3.5 Opmaak van grafieken

Wanneer twee grafieken in één figuur getekend worden, zoals hiervoor het geval was geeft `pyplot` automatische 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 13: code-inc/w2/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 op eenvoudige manier aan plt.plot() worden toegevoegd.
## Het voorbeeld hieronder produceert een plaatje met lijn en punten.
x = np.linspace(0,2*np.pi)

## Begin aan een nieuw figuur
plt.figure()

## Geef labels op voor dat nieuwe figuur
```

```
plt.xlabel("Waar ik kijk in de golf")
plt.ylabel("Hoeveel de golf op en neer gaat")

plt.plot(x,np.cos(x),'r-')
plt.plot(x,np.cos(x),'k.')
plt.show()
```

3.5.1 Opdracht: Basis rafiek opmaak

Pas het script na opdracht 3.4.1 aan zodat er zowel een lijn als punten geplot worden. Voeg ook as-labels toe.

Soms kan het wenselijk zijn om twee plots boven elkaar of naast elkaar te laten zien, eventueel zodat ze de x- of y-as delen. Om meerdere plots tegelijk te laten zien kan het commando `plt.subplots()` gebruikt worden. `plt.subplots()` retourneert een object dat de hele figuur beschrijft en voor elke grafiek in de figuur (subplot) een apart object, waarop met `axN.plot()` en gelijke commando's figuren getekend kunnen worden. In voorgaande voorbeelden was dit object impliciet gedefiniëerd, maar niet zichtbaar, en werd met `plt.plot()` automatisch in dit object geschreven.

Code 14: code-inc/w2/plt˙vb3.py

```
## Begin met een subplots.
## f 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.
f, (ax1, ax2, ax3) = plt.subplots(3, sharex=True, sharey=True)

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

ax1.plot(x,np.cos(x)**2)
ax2.plot(x,np.sin(x)**2)
ax3.plot(x,np.cos(x)*np.sin(x))

## Dit is finetuning. De eerste regel zet de ruimte tussen de figuren op 0.
## De tweede regel zet de markering langs de x-as uit voor alle plots behalve
## de laatste.
f.subplots_adjust(hspace=0)
plt.xticks()

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

Een tweede voorbeeld waar plotjes in een 2-bij-2 geometrie worden getoond:

Code 15: code-inc/w2/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
f, axarr = plt.subplots(2, 2, sharex='col', sharey='row')

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

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 refereert naar de titel van de gehele figuur. Dit is dus een
## eigenschap van de totale figuur f.
f.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. Dit kan handig zijn wanneer je de markering uit wilt zetten voor alle bovenste plots.

3.5.2 Opdracht: Uitgebreide grafiek opmaak

(e) Maak een geparameteriseerde plot. Kies t in $[0, 2\pi]$. Laat $x = 16\sin^3(t)$ en $y = 13\cos(t) - 5\cos(2t) - 2\cos(3t) - \cos(4t)$. Zorg voor vaste verhouding tussen de assen. Gebruik minstens 20 plotpunten. (TIP: Denk aan de opdracht eerder waar een cirkel geplot werd.)

(f) Maak een grafiek van $f(x) = x\exp(x)$ met de y-as op logschaal. Kies x in het bereik $[0, 5]$. Vergelijk ook met een figuur waar geen log schaal is gebruikt. (TIP: `plt.yscale('log')`).

(g) Maak een figuur met een sinus en cosinus, in het bereik tussen 0 en 2π. Bij het aanroepen van de `plt.plot()`, voeg de optie `label = "Label van grafiek"` toe. Label de sinus "Sinus- en de cosinus "Cosinus". Gebruik vervolgens het commando `plt.legend()` om een legenda te weergeven.

3.6 Multi-dimensionale arrays

Tot nu toe hebben we het enkel gehad over ééndimensionale arrays. Het grafisch weergeven van multidimensionale arrays kan ook interessant zijn. We kijken in het bijzonder naar tweedimesionale arrays. Om een multidimensionale array met enen of nullen te verkrijgen kunnen dezelfde commando's gebruikt worden als eerder. Ook het selecteren van elementen per index gaat op dezelfde manier.

Code 16: code-inc/w2/nd˙vb1.py

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

## Gelijk aan het 1D voorbeeld, hier het 2D voorbeeld
```

```
tweeD_nullen  = np.zeros((6,6))
tweeD_enen    = np.ones((6,6))
tweeD_random  = np.random.rand(6,6)

## Moeilijker is een bereik, er zijn twee manieren om een 2D object
## te maken uit een np.range object.
eenD_array  = np.arange(36)
tweeD_array = eenD_array.reshape(6,6)
## Bovenstaande voorbeeld produceert een array als
## [ [ 0 1 2 3  4  5 ]
##   [ 6 7 8 9 10 11 ] etc.
## Vaak willen we een verzameling x en y coordinaten. Waar de ene array als
## x-coordinaat fungeert en de tweede als y-coordinaat.
x = np.arange(6)
y = np.arange(6)
xv, yv = np.meshgrid(x, y, indexing='xy')
## Dit levert dus:
## x:  [ [ 0 1 2 3 4 5 ]
##       [ 0 1 2 3 4 5 ] etc.
## y:  [ [ 0 0 0 0 0 0 ]
##       [ 1 1 1 1 1 1 ] etc.
## Het indexing keyword is optioneel. De andere optie is 'ij', waardoor
## de indices in beide arrays worden omgedraaid. (In 2D draait dit de rijen
## en kolommen in beide arrays om.)
##
## Gebruik met een functie als volgt:
x = np.linspace(-1.0,1.0)
y = np.linspace(-1.0,1.0)
xv, yv = np.meshgrid(x,y)
hyperbool = xv**2 - yv**2

## Om data te visualiseren gebruiken we plt.imshow()
## De optie "interpolation='none' zorgt ervoor dat pixels niet in elkaar
## verlopen hetgeen ongewenst is voor de meeste data. (Dit is de default)
## Wanneer dit geen probleem is/wenselijk is, kan hier voor een ander type
## interpolatie worden gekozen zoals'bicubic'.
## Extent geeft aan wat er op de x- en y-as moeten komen.
plt.imshow(hyperbool,
           interpolation='none',
           cmap = 'seismic',
           extent = [x.min(),x.max(),y.min(),y.max()])
## Voegt een colorbar toe.
plt.colorbar()
plt.show()
```

3.6.1 Opdracht: 2D arrays plotten

(a) Voer het programma in het bovenstaande voorbeeld uit. Probeer ook `'ij'` indexering, wat is het verschil? (TIP: Variëer om hier achter te komen de afmetingen van x en y. Wanneer x en y hetzelfde zijn, is dit moeilijk te zien. Het is ook mogelijk om een ander aantal elementen te kiezen en gebruik te maken van `.shape`.)

(b) Laat x en y allebei in het bereik $[-1.0, 1.0]$ liggen. Laat x/y voor dit bereik zien. Merk op dat door de singulariteit dit plaatje erg vertekend is.[12] `plt.clim=(zmin,zmax)` (met gunstige waarden voor `zmin` en `zmax`) om de limiet op de kleurschaal in te stellen, in het `plt.imshow()` commando. Probeer met en zonder interpolatie te visualiseren. Vergeet de colorbar en assen niet!

(c) Laat x en y allebei in het bereik $[-1.0, 1.0]$ liggen. Laat $f(x, y) = \sin^2(4\pi x)y^2$ zien, inclusief colorbar. Neem het gemiddelde langs de x-as voor elke y, gebruik hiervoor `.mean(axis=0)`, gelijk aan `.mean()` eerder. Neem ook het gemiddelde langs de y-as. Laat beide zien in grafieken.

Wanneer data niet equidistant gesampled is kan `plt.imshow()` de data niet correct weergeven. Een voorbeeld is wanneer we een functie parameteriseren op de cirkel. In het geval van 2D geparameteriseerde data kan `plt.pcolormesh()` gebruikt worden.

Code 17: code-inc/w2/nd˙vb2.py

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

## Maak arrays voor r en theta.
th = np.linspace(-0,2*np.pi,num=100)
r  = np.linspace(1E-5,1.0)       # r = 0 is overgeslagen om singulariteiten
                                 # te voorkomen.

## Meshgrid.
rv, thv = np.meshgrid(r,th)

## Genereer functiewaarden in 2d-array 'img'.
## In dit geval de tweede Bessel-functie en een hoekafhankelijke oscillatie.
fr = np.sin(rv)/rv**2 - np.cos(rv)/rv
fth = np.cos(4*thv)
img = fr*fth

## Plot eerst met imshow:
plt.figure("Figuur 1")
plt.imshow(img,
           extent = [r.min(),r.max(),th.min(),th.max()])
plt.colorbar()
plt.axes().set_aspect(1/(2*np.pi)) # Zo wordt 2*pi langs theta-as even
                                   # lang als 1 langs de r-as.
plt.xlabel(r"$r$")
```

[12] Daarnaast valt het je misschien op dat de y-as andersom staat dan je zou verwachten; dit komt omdat bij `plt.imshow()` het coördinaat `[0,0]` (= *origin*) linksboven geplot wordt.

```
plt.ylabel(r"$\theta$")
plt.show()

## Construeer xv en yv uit rv en thv.
xv = rv * np.cos(thv)
yv = rv * np.sin(thv)

## Plot nu de geparameteriseerde plot met plt.pcolormesh()
## Nu wordt de hoekafhankelijkheid 'uitgerold':
plt.figure("Figuur 2")
plt.pcolormesh(xv,yv,img)
plt.colorbar()
plt.axes().set_aspect('equal')
plt.xlabel(r"$x$")
plt.ylabel(r"$y$")
plt.show()
```

Je hebt in de voorbeeld code al gezien dat met behulp van de optie `cmap` de kleuren van de 2D plot gekozen kunne worden. In de natuurkunde worden `cmap = 'Greys'` voor grijstinten (bij absolute waardes), en `cmap = 'seismic'` voor blauw-wit-rood (bij positieve en negatieve waardes) het meest gebruikt. De default (als je geen `cmap` specificeert) is `'viridis'`, die van paars via blauw en groen naar geel gaat.

3.6.2 Opdracht: Berglandschap

In deze opgave introduceren we contourlijnen en contourplots. Contourlijnen zijn wellicht bekend als de hoogtelijnen op een landkaart. De functie die hiervoor gebruikt wordt is `plt.contour(X,Y,Z)`. Deze functie heeft dus dezelfde syntax als `plt.pcolormesh()`. Er zijn verschillende opties om de contourplot te styleren. Zoek op het internet naar voorbeelden.

Voor deze opdracht wordt gebruik gemaakt van een van tevoren geprepareerd script waar jullie jullie bijdrage aan toe kunnen voegen. Het bestand `berglandschap.py` is te vinden op blackboard in de map met codevoorbeelden van deze week. Voeg jullie bijdrage op de daarvoor aangegeven plek toe. Stel hiervoor in dat `use_imshow = 0`, zodat het plaatje niet ook met `plt.imshow()` weergeven wordt.

Om het aantal niveaus van de contour in te stellen bestaat de optie `levels` voor `plt.contour`. Het is mogelijk om het aantal niveaus in te stellen door middel van `levels = numlevels`, waardoor er `numlevels` niveaus worden getekend, evenredig verdeeld over de grootste en kleinste af te beelden waarde.

Vaak is het beter om iets meer controle te hebben, hierom is het ook mogelijk om bijvoorbeeld `levels = np.linspace(-2, 2, num=11)` te kiezen. Hierdoor worden 11 niveaus weergeven verdeeld tussen de z-waarden -2 en 2. Het is mogelijk om een de aangeleverde lijst met niveaus te tweaken. Bijvoorbeeld `levels = np.array([-3,-1,0,-1,-3])` zal lijnen tekenen voor niveaus -3, -1, 0, 1 en 3.

a) Voorzie het originele script, dat je opent in Spyder van comment en pas het bestaande comment aan zodat dit consistent wordt met het uiteindelijke script.

b) De functie `berglandschap1` ga je niet wijzigen. Maar wat is de maximale en minimale waarde die `Z` heeft?

c) Voer de nodige regels code toe die zorgen voor een mooie contourplot, en onderdruk daarbij, zoals gevraagd, het tekenen van het oorspronkelijke berglandschap.

d) Gebruik `plt.figure()` om er voor te zorgen dat de contourplot en de originele plot niet samen in één plaatje verschijnen, maar in twee aparte figuren.

e) Pas de comments zonodig nogmaals aan, en sla jouw versie van `berglandschap.py` op. Sla ook de plaatjes die dit script produceert op met zoiets als `plt.savefig("naam_van_plaatje.png")`.

3.7 Random getallen

Getallen gegenereerd door een (klassieke) computer kunnen nooit geheel willekeurig zijn, omdat de computer op een compleet deterministische manier functioneert. Een ingewikkeld (valt reuze mee) algoritme genereert de getallen sequentieel en baseert het nieuwe willekeurige resultaat op het voorgaand gegenereerd getal. Het numpy pakket maakt standaard gebruik van een Mersennetwister-algoritme, een veelgebruikte pseudo-random number generator. Om de random number generator te initialiseren is er de functie `np.random.seed(int)`. Wanneer de random number generator niet geseed wordt, wordt deze geseed op het moment dat deze voor het eerst aangeroepen wordt. Het grote voordeel (of nadeel!) van het gebruiken van een seed is dat dan 'random getallen' gebruikt kunnen worden die telkens hetzelfde zijn. Dit kan handig zijn bij het testen van code.

In de `np.random` library zijn een groot aantal distributies opgenomen. Een volledige en geüpdatete lijst is te vinden in de documentatie van `numpy` op internet. In deze cursus zullen een aantal van deze verdelingen gebruikt worden. Hieronder een voorbeeld waarin een serie random getallen wordt gegenereerd uit verschillende verdelingen.

Code 18: code-inc/w3/random˙vb1.py

```
## Het volgende script demonstreert een aantal veel gebruikte
## kansverdelingen en het seeden van de random number generator.

np.random.seed(43890) # Seed de RNG met 43890
# Random getallen uit een aantal verdelingen. Uniform [0,1)
unif     = np.random.random()
# Normaalverdeeld gem. = mu, std. = sigma
normaalv = np.random.normal(mu,sigma)
# Poisson verdeeld, lambda = lamb
poissonv = np.random.poisson(lamb)

## Om meerdere samples te nemen (werkt op elke verdeling)
normv10x10 = np.random.normal(mu,sigma,size=[10,10])
# Een andere manier om veel random getallen tegelijk te maken is
normv = np.random.normal(mu,sigma,size=100)
normv = normv.reshape(10,10) # of normv = np.reshape(normv,(10,10))
```

3.7.1 Opdracht: random getallen en pi

Met behulp van uniform verdeelde getallen in het interval $[0, 1)$ kan π bepaald worden. Het algoritme is als volgt:

- Kies twee willekeurige getallen (x, y)
- Bereken $r = \sqrt{x^2 + y^2}$
- Als $r < 1$ ligt het punt binnen een cirkel met straal 1, schrijf dan 1 in lijst, als het punt er buiten ligt schrijf je 0 in de lijst
- De verhouding nullen en enen (Tip: gemiddelde van lijst) moet hetzelfde zijn als de verhouding in oppervlakte van vierkant (1) en kwart-cirkel ($\pi/4$). Bereken zo π.

Implementeer dit algoritme om π te bepalen. Plot het verschil tussen de gevonden waarde en π (`np.pi`) als functie van het aantal gebruikte punten. Gebruik een aantal punten $N = 10^3, 10^4, 10^5, 10^6, 10^7, 10^8$. Plot het verschil met π op een log-log schaal. Herhaal je berekening een aantal maal, door het script met 5 verschillende seeds te runnen. Hierdoor krijg je gevoel voor de spreiding of de onzekerheid in *meetresultaten* waarmee je bij DATA-V uitgebreid zult worden geconfronteerd. In plaats van één punt bij elke waarde van N krijg je zo 5 meetpunten bij elke N, die je dus ook allemaal in één plaatje kunt tonen.

3.7.2 Opdracht: 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: Genereren data

We zullen in deze opdracht onze bananen met behulp van de pseudo-random number generator genereren. De bananen hebben een lengte L, diameter d, kromtestraal R en bruine stukken van het oppervlakte met afmeting A. Kies voor normaalverdeelde statistiek met de parameters $(\mu_L, \sigma_L) = (15.5, 1.0)\,\text{cm}$, $(\mu_d, \sigma_d) = (35.0, 5.0)\,\text{mm}$, $(\mu_R, \sigma_R) = (20.0, 3.0)\,\text{cm}$ en $(\mu_A, \sigma_A) = (2.2, 1.3)\,\text{cm}^2$. Zorg dat de statistiek voor A geen negatieve getallen bevat. Ga ervan uit dat wanneer het oppervlakte A negatief wordt, de banaan geheel vrij is van smetten. Genereer 10^5 samples.

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 $L_{\text{min}} = 14\,\text{cm}$ en de minimale dikte is $d_{\text{min}} = 27\,\text{mm}$. 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. Hoe dit zich uit in de criteria staat 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 er niet binnen een klasse past en dus afgekeurd is.[13] 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	$\leq 1\,\text{cm}^2$	$\leq 2\,\text{cm}^2$	$\leq 4\,\text{cm}^2$

[13] 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.

4 Importeren en exporteren

Voor communicatie en herbruikbaarheid van resultaten zijn de onderliggende gegevens niet alleen beschikbaar in de vorm van een Python *script*. In dit werkcollege worden manieren behandeld om figuren en data als aparte bestanden aan te maken (ook wel *exporteren* genoemd).

Natuurlijk willen we ook gebruik kunnen maken van gegevens van anderen en die binnen onze de Python-omgeving brengen. Dit heet *importeren*.

4.1 Afbeeldingen

In hoofdstuk 3 heb je geleerd om de module `matplotlib` te gebruiken voor het visualiseren van data. Figuren worden standaard alleen getoond binnen de Python omgeving (in de console of in het tabblad 'Plots' in de Explorer van Spyder) en kunnen niet direct gebruikt worden in een verslag, artikel of rapport. Daarvoor moet een afbeelding eerst worden opgeslagen. Hiervoor kan binnen `matplotlib` gebruik worden gemaakt van `plt.savefig()`, waarbij je het gewenste bestandstype zelf kunt opgeven.

De belangrijkste opties die aan deze functie moeten worden meegegeven zijn de naam van de afbeelding, het type (*format*) en de resolutie. Wellicht zou je verwachten dat de grootte van de afbeelding hier ook gespecificeerd kan worden. Echter, dit wordt gedaan op het moment van het aanmaken van het *object* met bijv. `plt.figure(figsize=[8,4])`. Dit bepaalt respectievelijk de horizontale en verticale afmeting in inches.

Zorg ervoor dat de extensie van de bestandsnaam overeenkomt met het gekozen *format*. De default locatie waar de afbeelding wordt opgeslagen is de map waar het script is opgeslagen, of, als je script nog niet is opgeslagen, in de map die je in Spyder rechtsboven geselecteerd hebt. Je kunt afbeeldingen direct op een andere plaats opslaan door in plaats van alleen een bestandsnaam, ook de locatie op te geven. Bijvoorbeeld: `'C://Documenten/onderzoek/project/afbeeldingen/naam.jpg'`.[14] Het nadeel van deze aanpak is dat je script hierdoor veel minder goed bruikbaar wordt door derden zoals jouw compagnon.

Als *format* kan er o.a. worden gekozen voor `jpg`, `png` en `pdf`. Bestanden met de extensie .jpg en .png zijn opgebouwd uit een raster van punten (*pixels*) die elk een gespecificeerde kleur hebben en worden ook wel *rasterafbeeldingen* genoemd. De resolutie van deze afbeeldingen wordt gespecificeerd in *dots-per-inch* (`dpi`). Zulke afbeeldingen kunnen niet onbeperkt worden vergroot, omdat op een gegeven moment de individuele pixels zichtbaar worden. Afbeeldingen van het type .pdf zijn opgebouwd uit formules van eenvoudige meetkundige objecten, zoals punten, lijnen en/of krommen in combinatie met hun onderlinge positie. Ook gewone letters hebben zo'n structuur. Daardoor laten deze zogenaamde *vectorafbeeldingen* zich wel tot elk gewenst formaat vergroten zonder dat er kwaliteitsverlies optreedt.

Code 19: 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)
```

[14] Merk op dat de notatie `'C://'` hier wordt gebruikt. Deze notatie is echter afhankelijk van het operating system (OS) dat je gebruikt. Python laat je gebruik maken van `'/'` voor OS-X, Linux en Windows. Windows gebruikt zelf \ dus let op bij het copy-pasten van file-paths.

```
plt.figure(figsize=[8,4])
plt.plot(x,y)
plt.savefig('sinus.png', format='png', dpi=300)
```

4.1.1 Opdracht: Figuren opslaan

- Open het onderstaande voorbeeld-script `export_vb2.py`. Zoals je ziet zijn er opties, die wel in het vorige voorbeeld stonden, verdwenen zoals `format` en `dpi`. Het opgeven van `format` is overbodig mits je de filenaam maar van de gewenste extensie voorziet. Je kunt met dit voorbeeld ook makkelijk meerdere figuren in één script aanmaken.
- Wijzig 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.

  Sla ook de plaatjes op zonder dat je opties instelt. Aan de filenaam moet je kunnen herkennen wat de waarde is van de diverse instellingen. Je kunt het effect van elke instelling afzonderlijk onderzoeken.
- Kijk grondig naar de kwaliteit van de aangemaakte plaatjes, door ze te openen met de standaard applicatie voor .png of .pdf plaatjes. Gebruik de zoom-functie! Kijk ook naar de bestandsgrootte (in kilobytes) van de aangemaakte bestanden. Bedenk voor welke situaties je welke combinatie van opties zou willen gebruiken.

Code 20: 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.
```

- Zoek een eerder gemaakt script waarin je een plot hebt gemaakt. Exporteer een plaatje dat door het script wordt aangemaakt zodat je het makkelijk kunt opnemen in een rapport geschreven in LaTeX. Beoordeel nogmaals de .pdf en de .png bestanden die met dat script worden aangemaakt op hun bruikbaarheid binnen een rapport dat je schrijft in LaTeX.[15]

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 of ander experiment) vereist wat meer werk.

4.2 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 voor een bepaalde mate van inzichtelijkheid zorgt. Hoewel het een eenvoudige manier is, wordt dit binnen de (exacte) wetenschap veel gebruikt.

Structuur aanbrengen in een bestand is essentieel. Dit begint bij het ordenen van de gegevens langs in rijen en kolommen overeenkomstig met de structuur van je data. Daarnaast is het ook belangrijk om extra informatie (ook wel *metadata*) aan het bestand toe te voegen. Dit omvat namen van variabelen, maar ook meetomstandigheden en andere informatie die een gebruiker van het bestand nodig heeft om de gegevens te interpreteren en/of te verifiëren. Alleen hiermee zijn de gegevens in het bestand bruikbaar voor een ander en/of op een later tijdstip.

Voor het aanmaken van bestanden kan goed gebruik worden gemaakt van de ingebouwde *functies* `open()` en `write()`. De werking van deze functies staat dicht bij het fysiek bijhouden van bijvoorbeeld een labjournaal. De functie `open()` creëert een *object* dat aangeeft in welk bestand de gegevens moeten worden opgeslagen, terwijl met de *functie* `write()` het daadwerkelijk wegschrijven 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* werkt overigens (nagenoeg) hetzelfde als `print()`.

Merk op dat de *functie* `write()` onderdeel is van het *object* dat is aangemaakt met `open()`. Dit is een typische structuur die in Python veel gebruik wordt en wordt *object-georiënteerd programmeren* genoemd. Binnen deze manier van werken worden *objecten* aangemaakt die automatisch bepaalde functionaliteiten en functies hebben. De manier om deze functionaliteiten aan te spreken is door het object aan te wijzen, gevolgd door de naam van de functie die gebruikt moet worden. In onderstaand voorbeeld staat: `file.write('blabla '.format(x,y,z))`. Hier zie je weer een andere manier van formatten. Bij deze manier gebruik je `.format()` om na je `str` de variabelen aan te roepen. Deze *'.format-methode'* kan handig zijn bij het exporteren van data

[15] Voor meer informatie kijk op https://www.a-eskwadraat.nl/Vereniging/Commissies/hektex/

naar een file omdat de variabelen overzichtelijk aan het einde staan. Je kunt echter ook gewoon de *f-string*-methode gebruiken. In het voorbeeld is '**file**' de (willekeurige) naam van het object en '`write()`' de functie die aangeroepen wordt. Tussen de ronde haakje van de `write` zien we weer een dergelijke structuur: eerst een string, gevolgd door een punt en de functie **format** die aangeeft hoe die string (inclusief alles erin) geformatteerd dient te worden

Code 21: 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 in het bestand geschreven mag worden
mijn_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
mijn_file.write('# Dit is de header (eerste regel) \n')
mijn_file.write('# Schaal parameter horizontaal:\t {:5.2f}\n'.format(par1))
mijn_file.write('# Schaal parameter verticaal  :\t {:5.2f}\n'.format(par2))
mijn_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)):
   mijn_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.

mijn_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 tab. Dat is het `\t` symbool. Dit is een conventie die we in deze cursus aanhouden.[16] Het maakt het makkelijker om elkaars databestanden zo weer in te lezen en de inhoud ervan correct te interpreteren.

4.2.1 Opdracht: Wegschrijven van data naar file

Genereer vier arrays van 101 elementen. Kies x in $[-\pi/2, \pi/2]$ en laat $y = \sin(x)$ en $z = \cos(x)$ en bereken $t = y/z$. Dat is dus $\tan(x)$. Gebruik het bovenstaande voorbeeld-script als uitgangspunt om deze arrays te exporteren. Voorzie het bestand van een header die uit één regel tekst bestaat. Sla alle getallen op in wetenschappelijke notatie. Inspecteeer, door in het weggeschreven bestand te kijken, hoe de getallen waarvan je weet dat ze 0 zijn (of juist $\pm\infty$) zijn opgeslagen. Noteer je bevindingen.

4.3 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 wederom met generieke Python functies zoals `read()` of `readlines()`. Er zijn talloze functies/methoden die zich bezighouden met dergelijke I/O (input/output) aspecten.

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 tab's. Het scheidingskarakter heet de *delimiter*. We spreken af dat dit het `'\t'`-karakter is. Met `np.genfromtxt()` kun je de delimeter specificeren. Andere veelvoorkomende delimiters zijn: `','` en `';'`. De *default* waarde van de delimiter (als die niet gespecificeerd wordt, of bij `delimiter=None`) 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*.

5. De inhoud van zo'n commentaarregel bevat mogelijk belangrijke informatie (metadata). Zoals de numerieke waarde van één parameter. De waarde van die parameter is dan het laatste element op zo'n commentaarregel.

[16] Een komma als scheidingssymbool tussen de diverse items zou ook kunnen, en wordt in de praktijk vaak gebruikt.

Het doel is om bestanden die aan deze eisen voldoen te kunnen importeren en bovenal correct te kunnen interpreteren in Python. Concreet betekent dit dat na het inlezen van het bestand een `np.array` wordt aangemaakt dat de gewenste afmetingen (`shape`) heeft. Tevens moet er een parameter-lijst worden aangemaakt die de numerieke waarde van de parameters bevat die in het bestand voorkomen. Dit kan een `numpy`-array zijn, maar het is ook mogelijk dat een aantal variabelen (de parameters), zoals `par0` en `par1`, de gegeven waardes krijgen.

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. De werking van al die *lees-functies* zijn dus het digitale equivalent van het lezen van bijvoorbeeld een labjournaal: regel voor regel en woord voor woord.

Code 22: code-inc/w4/import˙vb1.py

```
"""
Script om alle items in een bestand als string te interpreteren
Zo'n string bevat geen spaties of tab's.
Maak zelf je eigen oefen—bestand aan.
"""
f =  open("vb1.txt", "r")
block = f.readlines()

data = []   #  een lege Python lijst
for line in block:
    data.append( line.split() )
# check data na runnen script
```

Opmerking 1: In dit voorbeeld wordt een bestand ingelezen als een reeks elementen van het type `str`. De Python lijst `data` bevat na inlezen evenveel elementen als dat er regels in het bestand zijn. Elk element van die lijst kan uit één of meerdere elementen (eventueel zelfs 0) bestaan. Dat zijn namelijk de *woorden* (strings) die op een regel staan.

Opmerking 2: Er is een een extra stuk code nodig om Python te laten weten dat een ingelezen `string` een getal is. In onderstaand code fragment wordt een handige functie gedefinieerd die daarvoor bruikbaar is.[17]

Normaal is regel `b = float(a)` genoeg om een string `a` te casten naar een float `b`. Maar als `a` helemaal geen getal voorstelt dan geeft Python een foutmelding en loopt het hele script vast. Dit vastlopen willen we juist voorkomen, en dat wordt netjes opgelost door de functie `is_number()`.

Code 23: code-inc/w4/is˙number.py

```
def is_number(s):
    try:
        float(s)
        return True
    except ValueError:
        return False
```

[17] Het zelf maken van functies komt aan bod in hoofdstuk 5. Kijk of je deze gegeven functie toch al kunt gebruiken zonder helemaal te begrijpen wat er gebeurt door die bovenaan je script te kopiëren.

Het expliciet openen van een file, en het lezen van alle data (met `readlines()`) en het daarna uitpluizen van wat er zoal op elke regel staat is lang niet altijd nodig. 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 volgens onze conventies commentaar. Het weergeven van bijvoorbeeld twee kolommen uit de ingelezen data in een xy-plot stelt dan echt weinig meer voor.

Code 24: 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.figure()
plt.plot(x,y)
plt.show()
```

4.3.1 Opdracht: Importeren en bewerken

- 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 interpreteren.

- Gebruik de drie bovenstaande voorbeelden om tot één script te komen dat dit kan doen:

  1. Gebruik `np.genfromtxt` met altijd de tab als `delimiter`. Dit heeft tot doel om de meetgegevens in het bestand te lezen en die in een `numpy`-array om te zetten.
  2. Voeg de definitie van de `is_number` functie toe direct na het importeren van de modules. Je mag die functie eventueel ook zelf *importeren* met `import` en zo meerdere `.py`-bestanden aan elkaar koppelen.
  3. Voeg ook het script toe dat het bestand opent en leest met `readlines()`. Dit lezen, eigenlijk het opnieuw lezen, van het hele bestand heeft alleen maar tot doel om de parameters op te sporen.

4. Maak op de juiste positie een lege Python lijst `para`. Betekenis: nog geen parameters gevonden; de parameter-lijst is leeg.
5. In een **for**-loop kijk je of het eerste symbool in `line` gelijk is aan het karakter `'#'`. Indien ja, dan heb je een commentaarregel gevonden, indien nee, dan ga je gewoon verder met de volgende regel (dat doet de **for**-loop vanzelf).
6. Nu je dus een commentaarregel hebt gevonden is de volgende stap om de woorden op die regel te analyseren. Dat doe je met `words = line.split(sep='\t')`. Het specificeren van de tab als separator is nodig omdat we ons aan de gemaakte afspraken willen houden. Specificeer je geen separator, dan gaat het voor dit bestand wellicht net zo goed. Alles dat op een spatie lijkt wordt er dan uitgegooid.
7. Vanwege afspraak 5 hoeven we alleen het laatste 'woord' op elke regel te analyseren. Dat is dus `words[-1]`.
8. Nu komt de `is_number` functie van pas. Als deze functie `True` geeft als functiewaarde, dan kan het argument (dat is `words[-1]`), geïnterpreteerd worden als een getal. Deze als-dan suggereert natuurlijk het gebruik van een **if**.
9. Zet het betreffende woord met `float()` daadwerkelijk om in een getal. Dat doe je in de *body* van de **if**.
10. Gebruik tot slot `para.append()` om dit getal toe te voegen aan de Python lijst `para` met parameters.

Deze 10 stappen lijken heel veel werk, maar het leidt tot maar een paar regels Python-code.

- Print de lijst met parameters, en controleer of daar voor dit bestand inderdaad de getallen 50.0 en 0.12 in staan.
- Gebruik deze parameters tot slot om de x en y-waarden te schalen, omdat dat nu eenmaal in de beschrijving stond. Maak een plaatje van deze geschaalde sinus-functie en exporteer deze figuur.

4.4 Extra uitdaging

Nu heb je de belangrijkste basisvaardigheden voorbij zien komen. In deze cursus ligt de nadruk op een programmeertoepassing in data anlyse, omdat dat voor natuurkundigen erg van pas komt. Programmeren is echter veel breder dan alleen data analyse. In de cursus statistische fysica ga je een simulatie maken, en bij numerieke methode leer je hier nog meer over.

Er zijn talloze leuke programmeerpuzzels te vinden op internet. Als je van puzzels houdt, en een extra programmeeruitdaging zoekt kijk dan eens hier:

- https://projecteuler.net/about heeft heel veel puzzels die je o.a. met behulp van scripts of programmeren zou kunnen oplossen. De puzzels hebben meestal een wiskundige achtergrond.
- https://adventofcode.com/ begint elk jaar op 1 december met een serie programmeerpuzzels. Puzzels van voorgaande jaren zijn ook nog beschikbaar. Bij deze puzzels gebruik je een persoonlijk gegenereerde input (die je dus moet zien te importeren), en als controle geef je een antwoord wat uniek bij jouw input hoort. Deze puzzels hebben meestal een focus in optimalisatie. Let op: de puzzels van de eerste paar dagen zijn meestal goed te doen, maar worden al snel moeilijk!

5 Functies

Functies worden gebruikt om te voorkomen dat blokken code meerdere malen in een script herhaald worden. Hiermee bewerkstellig je dat, wanneer er een aanpassing gedaan moet worden, dit slechts op één plaats hoeft te gebeuren. Hierdoor helpt het je ook om het overzicht te behouden binnen een script als deze toeneemt in lengte en complexiteit.

Het importeren van packages is eigenlijk het importeren van een groep gerelateerde functies. Je hebt hier bijvoorbeeld al kennis mee gemaakt met `plt.plot(x,y,'k+')` of `plt.imshow(data, cmap='jet')`. Hierin zijn `y` en `data` argumenten, `x` en `'k+'` optionele argumenten en `cmap='jet'` een keyword argument.

In dit hoofdstuk leer je hoe je zelf functies kunt schrijven en gebruiken. Dit zal je ook meer inzicht geven in hoe je geïmporteerde functies gebruikt en hoe je de daarbij horende documentatie interpreteert.[18]

5.1 Definieer een functie

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

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

i = 1
j = 2
k = 3
z = 1
waarde = polynoom(i, j, k, z)    # hier gebruiken we positional arguments
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 daadwerkelijke definitie is.

Wat volgt is dus de code van de *functie* zelf. Merk op dat de regels code na het begin van de **`def`** net als bij loops niet geheel 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 kan worden bereikt door steeds een zelfde aantal spaties of tabs in te voeren aan het begin van de regel.[19] De Spyder editor helpt je hierbij enorm, want na een enter is de indent meestal gelijk aan die van de vorige regel. Precies zoals Python verwacht.

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

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

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, de technische term hiervoor is *call*) 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** window in Spyder. Hier kun je zien welke variabelen er zoal 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 het iPython console vanwege de uitgevoerde `print`.

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

```
waarde = polynoom(x=1, a=3, b=2, c=5) # hier gebruiken we keyword arguments
print('waarde = ', waarde)
```

Je moet in dit geval niet alleen weten dat de functie `polynoom` 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 *positional arguments*.
- 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 *keyword arguments*.

5.1.1 Opdracht: Controleren van bovenstaande functie

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

Vaak weet je van een functie helemaal niet wat de naam (het keyword) 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 derden geschreven *modules* zoals `numpy` en `matplotlib` bestaan maken wel degelijk gebruik van *keyword arguments*. 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 de 2^e week zul je min of meer ongemerkt ook al *keyword arguments* gebruiken.

5.2 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 bekend en hebben de getoonde waarde. De andere variabelen zijn alleen bekend binnen de functie `polynoom` en heten lokale variabelen.

Waarden van lokale variabelen kun je hooguit weergeven door ze te printen binnen een functie, maar dat wordt al snel hoogst irritant als zo'n functie duizenden malen wordt aangeroepen. Bij programmeren is het vaak de bedoeling dat je taken automatiseert en herhaald, weliswaar met andere waarden voor de argumenten, uitvoert.

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 `polynoom` de waarde van `y` terug. In het script stoppen we deze waarde in de variabele `waarde`, welke daarna afgedrukt wordt.

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

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

waarde = polynoom(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.

Wanneer je toch wilt dat binnen een functie de waarde van een globale variabele moet kunnen veranderen in een andere waarde dan kan dit door gebruik te maken van het statement **`global`**. Zulke functies beïnvloeden dus hun omgeving, en zeker voor beginnende programmeurs is het een slechte gewoonte om hier gebruik van te maken. Zie voorbeeld hieronder.

Gebruik je een globale variabele met de naam `a` die dus ook als lokale variabele `a` bestaat binnen een functie dan is er in wezen geen probleem. De lokale `a` beïnvloedt de globale `a` niet. Ter illustratie:

```
a = 20          # globale variabele a
b = -2          # globale variabele b
def f1(x):
    a = 21      # dit is een lokale variabele
    c = a*x + b # lokale a (waarde 21) en globale b (waarde -2)
```

```
    return c

def f2(x):
    global a
    a = 21      # de globale variabele a krijgt nu waarde 21
    c = a*x + b # zowel a als b zijn globale, maar c is lokaal
    return c

x = f1(3);
print(x) # geeft 21*3-2 = 61
print(a) # geeft 20
y = f2(3);
print(y) # geeft 21*3-2 = 61
print(a) # geeft 21 want f2 heeft de globale a gewijzigd.
```

Bij het gebruik van vele 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 express) meerdere keren dezelfde naam gebruiken. Dit zou er voor kunnen zorgen dat je script verkeerde resultaten oplevert. Gelukkig is Python zo opgebouwd dat globale variabelen worden verborgen door lokale variabelen met dezelfde naam.

5.2.1 Opdracht: Begrip van variabelen binnen en buiten een functie

Beschouw het volgende Python script:

```
def polynoom(a,b,c,x):
    print('binnen functie geldt a = ', a)
    return a*x**2 + b*x + c

f = polynoom(x=3.14, a=1, b=1, c=1)
print('buiten functie geldt a = ', a)
```

Wat gebeurt er als je dit script uitvoert? Had je dit 'resultaat' in Spyder (dus niet door naar dit script te kijken op een vel papier of in een pdf bestand op je scherm) ook al kunnen zien aankomen zonder dat je het script uitvoerde? Voeg de regel `a = 3.14` toe direct na de functiedefinitie. Run het script opnieuw en ga na of je de output ervan nu wel begrijpt.

5.3 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. Elk aantal argumenten is mogelijk, dus ook nul argumenten. Zo'n functie met nul argumenten zal dus altijd hetzelfde doen omdat er geen argument is om het gedrag te beïnvloeden.

```
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.

- Functies kunnen nul of meer argumenten hebben.
- Soms hoef je argumenten niet op te geven en wordt een *defaultwaarde* aangenomen.
- Soms kom je functies tegen met argumenten die een label hebben: de *keyword* argumenten.
- Soms kom je functies tegen die een variabel aantal argumenten hebben.

5.4 Extra argumenten

Een functie kan een *variabel* aantal argumenten mee krijgen. Die voer je in door `f(a,b,*args)`, waar `*args` een willekeurig aantal argumenten kan zijn. Bijvoorbeeld sommatie 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
```

5.4.1 Opdracht: 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 de *index* om losse argumenten te maken.)

Een handige toepassing is wanneer een functie een andere functie moet aanroepen. Een functie die op een andere willekeurige functie werkt, waarbij deze laatste functie een aantal vaste parameters heeft (zoals bijvoorbeeld in minimalisatieproblemen). Om dit te illustreren hieronder een voorbeeld, waarin de functie `my_plot` met soms 4 en soms met 5 argumenten moet worden aangeroepen om de gewenste plaatjes te krijgen.

Code 25: code-inc/w5/voorbeeld5˙2.py

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

# Sinus
def g(x,amp,freq,offset):
    return offset + amp*np.sin(freq*x)

## Simpele plotfunctie
# plot functie "func" over range "x"
# extra argumenten worden doorgegeven met *args
def my_plot(x,func,*args):
    plt.plot(x,func(x,*args))
    plt.show()

## Gebruik:
x_arr = np.linspace(-5.,5.)
```

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*.

Opdracht: 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 een tweedegraads polynoom) en plot deze functie met de `my_plot` functie van hierboven.

5.5 Keyword argumenten

Een uitbreiding op de *extra arguments* in `*args` zijn er de *keyword arguments*. Deze werken net als `*args`, maar naast het argument geef je ook een label (een *keyword*) mee. Een voorbeeld waarin alle drie de types argumenten voorkomen staat hieronder:

Code 26: code-inc/w5/voorbeeld5˙3.py

```
def f(farg, *args, **kwargs):
    print("Het formele argument is", farg)
    print("De extra argumenten zijn:")
    for arg in args:
        print(arg)
    print("De keyword argumenten zijn:")
    for key in kwargs:
        print("  de key",key,"met argument",kwargs[key])
        if key == 'my_name':
            print("Mijn naam is", kwargs[key])

## Aanroepen van bovenstaande functie:
f("Aap", "Noot", "Mies", arg4="Wim", my_name="Zus")
# in bovenstaande aanroep zijn:
# Formeel argument   "Aap"
# Extra   argumenten "Noot" en "Mies"
# Keyword argumenten "Wim" bij keyword arg4, en "Zus" bij my_name
```

Zo'n keyword argument heeft natuurlijk alleen maar zin als je aan zo'n opgegeven *keyword* een bepaalde actie verbindt. In het vorige voorbeeld gebeurt dat alleen maar bij het keyword `my_name`. Keyword argumenten worden vaak optionele argumenten genoemd.

Een zinvol voorbeeld van het gebruik van een keyword is bijvoorbeeld `num` in de functie `np.linspace`. Als je dit keyword argument niet gebruikt, dan wordt een default waarde gebruikt; in dit geval is dat `num=50`. In de functiedefinitie van `np.linspace` moet die waarde dus gegeven zijn. Je kunt dit op de domme manier uitvinden door `num` weg te laten en te tellen hoeveel (en welke) punten er worden gegenereerd door `np.linspace(0,1)`.

5.5.1 Opdracht: Keyword argumenten

Bestudeer het bovenstaande voorbeeld. Kijk wat er gebeurt wanneer je extra argumenten toevoegt/verwijdert. Is het mogelijk geen keyword argumenten te geven? Is het mogelijk om geen extra argumenten te geven?

5.6 Default waardes

Bij het gebruik van keyword argumenten kan het handig zijn om meteen een *defaultwaarde* mee te geven. Dit is wat er als argument ingevuld wordt als het keyword niet gebruikt wordt.

In het voorbeeld hieronder is een sinusfunctie gemaakt met de defaultwaardes zó gekozen dat als je alleen `x` in vult er de sinus van de eenheidscirkel uit komt. Je kunt echter ook meer of minder andere argumenten mee geven.

Code 27: code-inc/w5/voorbeeld5˙6.py

```
def sin_func(x, A=1, P=2*np.pi, phase=0, v=0):
    # x : invoer van x-waarde(s)
    # A : amplitude, default is 1
    # P : periode, default is 2 pi
```

```
    # phase : fase-verschuiving naar links, default is 0
    # v : vertikale verschuiving, default is 0
    return A*np.sin((2*np.pi/P)*(x + phase)) + v
```

5.6.1 Opdracht: Default waardes

Plot de uitkomsten van bovenstaande functie met in elk geval de volgende y-waardes allemaal samen in één plot.

```
y1 = sin_func(x)
y2 = sin_func(x, P=np.pi)
y3 = sin_func(x, 0.5, v=1, phase=np.pi/2)
```

Gebruik voor x minimaal 2 periodes. Gedragen de variabelen zich als keyword argumenten of als positionele argumenten nadat je bij de definitie van de functie een defaultwaarde hebt toegekend?

5.7 Eigen functies importeren

Het importeren van externe pakketten kan erg handig zijn, zoals je al gezien hebt bij het gebruik van `numpy` en `matplotlib`. Op deze manier kan je ook functies importeren die je zelf geschreven hebt. Hieronder zie je een simpel voorbeeld van een script met een functie. Dit script doet zelf niks; de functie wordt niet aangeroepen, dus het runnen van dit script geeft geen output.

Code 28: code-inc/w5/mypolyimport.py

```
def poly(x,a,b,c):
    return a*x**2 + b*x + c
```

In het script hieronder zie je hoe deze zelfgeschreven functie geïmporteerd en gebruikt wordt. Net als bij `numpy` geven we hier een afkorting aan om deze later te gebruiken.

Code 29: code-inc/w5/voorbeeld5˙5.py

```
import mypolyimport as mp
import numpy as np

x = np.arange(0,11,1)
y = mp.poly(x,4,3,2)
```

5.7.1 Opdracht: Plotten

In deze opdracht schrijven we een functie om het plotten van data te vergemakkelijken. Met `matplotlib` is het plotten van functies een operatie die over verschillende regels verdeeld wordt. Het instellen van de figuur moet na het aanroepen van de plot gebeuren en neemt soms veel regels in beslag. Voor de meeste toepassingen zijn een aantal standaard-functies genoeg.

Schrijf een functie die (x, y)-data plot en de assen van labels voorziet. Dit zijn de minimale eisen aan een figuur in het vak DATA. Geef als optionele argumenten de fout in x en y, en zorg dat

er gebruik gemaakt wordt van `plt.errorbar()` wanneer deze waarden ingesteld zijn. Zorg ook dat het mogelijk is om de plot-range in te stellen. Ook kun je dingen als opmaak van de plot toevoegen, of eventueel een mogelijkheid om de figuur weg te schrijven naar een bestand. Kijk uitgebreid naar voorbeelden uit Week 2 en dit hoofdstuk en zoek vooral op internet, bijvoorbeeld in de documentatie van `matplotlib`.

De vorm van de functie wordt dus:
`my_plot(x, y, xlabel, ylabel, xerror=None,yerror=None, xrange=None,yrange=None)`.

Hoewel dit geen inleveropdracht is, is een goed geschreven functie `my_plot` handig en herbruikbaar in de rest van de cursus. Start met het programmeren zodat de functie werkt als je de vier verplichte formele argumenten meegeeft. Het toevoegen van de keyworded argumenten doe je stap voor stap, en controleer vooral steeds of je functie het ook nog doet zonder de keyworded argumenten.

5.7.2 Opdracht: Machtreeksen

In deze opdracht programmeren we de functie:

$$y(x) = y_0 + \sum_{n=0}^{N} A_n (x - x_0)^n$$

Schrijf een functie die als argumenten x, de coëfficienten A_n en als optioneel (ofwel *keyword*) argument de offsets y_0 en x_0 accepteert. De reeks moet zo lang zijn als het aantal coëfficienten A_n dat je opgeeft. Wanneer x_0 en y_0 niet ingevuld zijn, moet de functie ervan uitgaan dat deze 0 zijn. Dat heet de *defaultwaarde*. Deze opdracht is op eigen manier in te vullen, maar zorg dat je hem goed begrijpt want de inleveropgave bouwt op dit principe voort.

De functie moet de volgende argumenten accepteren:

```
x         # array met x-waarden
An        # array met coefficienten A_n

Optioneel
x0        # x-offset, defaultwaarde 0
y0        # y-offset, defaultwaarde 0
```

En de functie moet als *output* een array geven met de waarden voor $y(x)$ teruggeven (gebruik `return`), dat dus dezelfde dimensie heeft als input argument `x`.

6 Groepsopdracht: Functies en het Gibbs-fenomeen

In deze week is er tijd om samen met je groepje aan de groepsopdracht te werken. Bekijk eerst opdracht 5.7.2, deze lijkt in vorm erg op de inleveropgave. In de inleveropgave van deze week bestuderen we het Gibbs-fenomeen. Je kunt ***1 punt*** verdienen met een overzichtelijk en goed leesbaar script.

6.1 Fourierreeksen *3.5 punten*

Schrijf een Python functie die voor een gegeven t de volgende functiewaarde als output geeft:

$$y(t) = \sum_{n=0}^{N} A_n \sin(n \times 2\pi f_0(t - t_0) + \varphi_n).$$

Bovenstaande functie is een Fourierreeks: een som van golffuncties. Door de coëfficiënten A_n, en de andere parameters (f_0, t_0 en φ_n) juist te kiezen kan je de vorm van elke periodieke functie hiermee benaderen.

Net als bij de opgave over machtenreeksen bepaalt het aantal coëfficiënten A_n dat je opgeeft hoe groot N is. De verplichte argumenten zijn t, de lijst met coëfficiënten A_n en de frequentie f_0.[20] De optionele argumenten zijn de offset t_0 en de lijst met fase-getallen φ_n. Merk op dat wanneer φ_n ingesteld is, de lengte van de lijst met φ_n en A_n hetzelfde moet zijn. Dit dien je te controleren binnen je functie. Als de lijst met φ_n ontbreekt mag je veronderstellen dat alle φ's gelijk zijn aan 0.

Als φ_n en A_n niet dezelfde lengte hebben print je functie de foutmelding: `'phi_n en A_n zijn niet even lang!'`.

6.2 Testfuncties *2 punten*

Het Gibbs-fenomeen is het verschijnsel dat wanneer een discontinue functie ontbonden wordt over sinussen (wat continue functies zijn), de reeks van sinussen nooit exact convergeert naar de discontinue functie die gerepresenteerd wordt. Om dit te demonstreren vergelijken we de driehoeksgolf $D(t)$ met een zaagtand $Z(t)$[21]. Deze functies worden gegeven door:

$$D(t) = 4\left\| f_0 t + \frac{1}{4} - \left\lfloor f_0 t + \frac{3}{4} \right\rfloor \right\| - 1$$

$$Z(t) = f_0 t - \lfloor f_0 t \rfloor - \frac{1}{2}$$

De uitdrukking $\lfloor x \rfloor$ staat voor naar beneden afronden (Engels: *floor*) en wordt geïmplementeerd in Python met `np.floor()`. de uitdrukking $\|x\|$ betekent: de absolute waarde nemen.

- Implementeer deze functies in Python. Hoe denk je om te gaan met de grootheid f_0? Dit is een frequentie, die je natuurlijk ook wilt kunnen variëren.
- Laat de zaagtand, driehoeksgolf en $S(t) = \sin(2\pi f_0 t)$ in één figuur zien. Kies $f_0 = 2$.

[20] Een eerste test is dus dat de functie bij gebruik van dit statement het correcte antwoord print: 2. `print(fourierreeks([0.25], [1,2], 1))`.

[21] Beide functies zijn zo gedefinieerd dat het tijdgemiddelde gelijk is aan 0

6.3 Convergentie *3 punten*

We nemen vanaf nu voor het gemak aan dat $f_0 = 1$. De coëfficiënten A_n (zoals hiervoor beschreven) in de Fourierreeks van de driehoeksgolf (D_n) en zaagtand (Z_n) worden gegeven door:

$$D_n = \frac{8}{\pi^2} \begin{cases} 0 & n \text{ even} \\ \frac{(-1)^{(n-1)/2}}{n^2} & n \text{ oneven} \end{cases},$$

$$Z_0 = 0 \qquad Z_{n>0} = -\frac{1}{n\pi}.$$

Gebruik bovenstaande coëfficiënten om de convergentie van de Fourierreeks naar de originele functie te onderzoeken.

- Plot behalve de originele functie ook de benadering ervan met reeksen bestaande uit een verschillend aantal termen. Kies 1000 punten voor t om de functie te representeren. Je moet kunnen zien dat alle functies die je plot periodiek zijn (net zoals een sinus-functie) met een periode 1, ongeacht het aantal termen dat je beschouwt. Al die 1000 punten verdeel je dus bij voorkeur uniform over één periode.
- Kijk ook naar hoe het totale kwadratisch verschil verandert wanneer er meer termen meegenomen worden. Verwar hierbij niet het aantal termen N en het aantal punten in t. Met het totale kwadratisch verschil wordt bedoeld $\sum_n (D(t_n) - D_{\text{reeks}}(t_n))^2$, en gelijk voor $Z(t)$.
- Plot dit verschil op een log-schaal (langs de y-as). Langs de x-as staat het aantal termen uitgezet, N.

6.4 Conclusies *0.5 punt*

Beschrijf en vergelijk het convergentiegedrag van de driehoeksgolf en de zaagtand. (Je kunt dit onderaan je script in de vorm van comments doen.) Besteed aandacht aan de volgende punten:

- Hoe groot is het gemiddelde kwadratische verschil in beide situaties? Dat is namelijk een grootheid die niet afhangt van het aantal punten dat je kiest.
- Hoeveel termen zijn (in beide gevallen) nodig om convergentie te bereiken? Je moet dus aangeven wanneer je vindt dat convergentie is bereikt.
- Convergeren beide functies volledig? Ofwel, gaat het gemiddelde kwadratisch verschil naar 0? Houd er hier rekening mee dat resolutie in de tijd wordt bepaald door het aantal punten dat je kiest, en hierdoor je boven een aantal termen de gevoeligheid voor fluctuaties kwijt bent.

7 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[22] makkelijker en sneller te maken.

7.1 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 TA 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 rand
```

Richtlijnen voor het schrijven van goed commentaar:

- Gebruik *natuurlijke taal*, d.w.z. gebruik zinnen en schrijf voluit. Gebruik geen python code in comments,[23] en definieer (óók voor de hand liggende) symbolen als F, m, g, en a.
- 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.

[22] data analyse binnen deze cursus, maar ook alles wat je in de toekomst met programmeren zult doen
[23] 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` en ∗∗`kwargs` 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.[24] Zorg dat regels code en commentaar niet (veel) langer worden dan tot die lijn. Bij Python is de conventie[25] 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 # dit werkt ook, maar is erg onoverzichtelijk en foutgevoelig
```

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

[25] Gebruik de PEP8 Style Guide: https://www.python.org/dev/peps/pep-0008

7.2 Handigheden in Spyder

Spyder heeft een aantal shortcuts en handige ingebouwde functionaliteit waar je gebruik van kunt maken bij o.a. het schrijven van commentaar.

Voor het beschrijven van functies kun je gebruik maken van *docstrings*. Dit is een gestandaardiseerde weergave van informatie over functies. Je kunt in Spyder makkelijk een docstring beginnen door rechts te klikken op de functie, en daar 'Generate docstring' te selecteren. Let op dat je niet alleen de betekenis van de input en output specificeert, maar ook beschrijft wat de functie doet en waar deze voor gebruikt kan worden. De informatie uit de docstring is nu ook te zien als 'pop-up' als je de naam van de functie typt, zoals je al gewend bent bij functies uit geïnporteerde modules.

Handige key-board short-cuts in Spyder:

- `CTRL + C`: stopt het script, bijvoorbeeld omdat die vastgelopen is omdat die bijvoorbeeld in een slecht gemaakte loop vast zit.
- `F5`: run het hele script.
- `CTRL + enter`: run de huidige cell.
- `SHIFT + enter`: run de huidige cell, en stap naar de volgende cell.
- `CTRL + 1`: zet een `#` voor deze regel (of haal die juist weg).
- `TAB`: spring 1 level meer in. Dit werkt ook als je een aantal regels geselecteerd hebt, je hoeft dit dus niet regel voor regel te doen.
- `SHIFT + TAB`: omgekeerde van `TAB`: 1 level minder inspringen.

7.3 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).[26] 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 het de code gaan dan veel vlotter.

Het voordeel van een *scripting language* zoals Python is dat de leercurve minder steil is; het is makkelijker te leren en intuïtiever. 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)

[26] 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.

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.[27] 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 uitsluitend naar hoe we het script zélf efficiënter kunnen maken.

7.4 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 30: 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 = []                               # maak lege lijst voor even getallen
for element in random_nrs:                   # loop door de hele lijst
    if element % 2 == 0:                     # als een getal even is (% 2 = 0)
        even_nrs1.append(element)            # voeg het getal toe aan de lijst 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   # maak een 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.

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

7.4.1 Opdracht: Testen welke implementatie het snelst is

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

(b) Kijk in de Variable explorer, of typ `type(naam_variabele)` rechtstreeks in de console 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 (door er een loop omheen te zetten). De runtime van beide methodes is elke keer een beetje anders. Bereken de gemiddelde runtime (met onzekerheid) voor beide methodes.

Bovenstaande methode werkt minder goed voor het testen bij het optimaliseren van specifieke processen binnen een code. Hiervoor bestaat het package `timeit`. Deze module is vooral bedoeld voor het onderzoeken van de relatieve snelheid van verschillende aanpakken, en niet voor het timen van een groot stuk code als geheel.

Binnen de module timeit zitten veel functies, maar hier gebruiken we een simpele short-cut om de functionaliteit aan te roepen. Het %-teken is een IPython Magic Command (IMC), omdat we geen IPython gebruiken binnen Spyder besteden we geen tijd aan hoe je dit breder kunt toepassen. Het is echter wel de snelste en makkelijkste manier om gebruik te maken van de module timeit, en het werkt ook binnen Spyder.

We gebruiken hetzelfde voorbeeld als hierboven: vind alle even getallen op twee manieren.

Ga naar deze link voor de voorbeeldcode.[28]

Het resultaat wordt geprint in de Console. Bij de IMC methode van timeit moet elke `%timeit` in een eigen cell staan, of los worden aangeroepen in de console zelf. Je kunt de output op deze manier dus niet zelf printen, maar je kunt wel makkelijk en snel meten hoeveel tijd een bepaalde cell kost.

Je kunt `timeit` ook nog een aantal opties meegeven zoals je in dit voorbeeld kunt zien.

De module `timeit` heeft veel functies die handig kunnen zijn bij het exact timen van stukjes code, maar deze zijn wat ingrijpender in de code, en worden hier niet behandeld.[29]

Opdracht: Timing methodes vergelijken

(a) Ga na of beide timing methodes (ongeveer) hetzelfde antwoord geven. Let op wat je definieert als 'hetzelfde aantal handelingen'. In dit geval bijvoorbeeld het vinden van alle even getallen uit totaal x random getallen. Waar denk je dat een eventueel verschil vandaan komt?

(b) Gebruik `timeit`, de voorbeelduitwerking en je eigen uitwerking van oefenopdracht 3.3.1 om het snelste algoritme/script te vinden om de opdracht uit te voeren.

[28] Door het gebruik van % in de python code en het feit dat % in LaTeXgebruikt wordt om commentaar aan te duiden kan deze code niet weergegeven worden binnen het dictaat.

[29] Een uitgewerkt voorbeeld is wel te vinden op: https://nspracticum.science.uu.nl/DATA2021/DATA-Py/OpgBundel/code-inc/w6/timevb2.py

7.5 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.

7.6 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 (evt. van alle kolommen of rijen in x, met gebruik van `**kwarg axis`. Idem voor `min()` en `max()`.

  - `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 if 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 timeit: 0.40 sec
def compute_squares():
    return [i**2 for i in range(1000)]
# 1000x gemeten met timeit: 0.31 sec
```

Code 31: code-inc/w6/time˙vb3.py

```
x = 'awesome'
print('hello ' + x + ' world')      # oudste notatie
print('hello %s world' % x)         # verouderde notatie (python versie 2.7)
print('hello {} world'.format(x))   # nieuwe notatie (python versie 3.0)
print(f'hello {x} world')           # nieuwste notatie (python versie 3.6)
```

7.6.1 Opdracht: timeit gebruiken

(a) Gebruik `timeit` om de relatieve snelheid te bepalen van de oudere en nieuwe manieren van string-formatting in het voorbeeld hierboven.

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

7.6.2 Opdracht: Verbeter het script voor het vinden van pi

Kijk terug naar de code die je geschreven hebt voor opdracht 3.7.1 (het vinden van π m.b.v. random getallen). Gebruik je opgedane kennis over het maken van functies, het optimaliseren van code, en het maken van mooie plots om dit script te verbeteren. Let in elk geval op de volgende punten:

- Genereer de random getallen niet één voor één.
- Bereken en plot het absolute verschil tussen π en de gevonden waardes voor π als functie van N (aantal random getallen) voor N tussen 10 en 10^8. (Hint: gebruik `np.logspace()`.) Voor het testen kun je beter eerst tot bijvoorbeeld 10^6 punten gebruiken.
- Vind het gemiddelde absolute verschil (met onzekerheid) voor bovenstaande punten. (Hint: herhaal het bestaande script een aantal keer om een idee te krijgen van de onzekerheid.)
- Plot alle gevonden absolute verschillen samen met de gemiddeldes en onzekerheden in een log-log plot.
- Zorg dat de plot wordt opgeslagen.
- Zorg dat de data van het absolute gemiddelde verschil, de onzekerheid daarin, en de bijbehorende waarde van N worden opgeslagen in een `.txt`-bestand, met een header met daarin het aantal herhalingen. (Volg de afspraken die we over bestanden gemaakt hebben in 4.3.)
- Time de totale runtijd van het script. (Hint: Voor het timen van lange stukken code is `time.time()` de handigste methode.)
- Zorg dat alle in te stellen getallen bovenaan staan (zoals het aantal herhalingen om het gemiddelde en de onzekerheid uit te bepalen, de minimale en maximale waarde van N en het aantal stappen in N).
- Voorzie het script van goed commentaar.

Lukt het om voor $N = 10$ tot $N = 10^8$, met 10 herhalingen voor het vinden van het gemiddelde je code sneller te maken dan 2 minuten (inclusief plotten en data en plot opslaan[30])?

Extra uitdaging voor gevorderden: bereken π één keer voor alleen $N = 10^8$, dus zonder het gemiddelde en onzekerheid te berekenen, en zonder data te plotten of op te slaan. Kan jouw script dit binnen 6 seconden?

7.7 Jouw code

Je zult merken dat wanneer je vaker python nodig hebt je vaak dezelfde dingen wilt doen, zoals data importeren, plotten. En zelfs vergelijkbare 'problemen' tegen komt. Het is handig om jezelf aan te leren al je code goed te orderen en documenteren zodat je die (delen van) kunt hergebruiken.

7.7.1 Opdracht: Voorbereiden op inleveren

Verzamel code uit eerdere opgaven, structureer deze zodat je weet wat waar staat en je er makkelijk bij kunt. Zorg dat je in elk geval voor de volgende situaties een werkend en goed leesbaar voorbeeld script voor jezelf hebt gemaakt:

- Maak een simpele plot uit x en y data. Zorg voor labels op de assen, en dat je weet welke andere as-opties beschikbaar zijn en hoe je die moet gebruiken, zoals `xscale`, `xlim`, `title`, `savefig`, `legend` en `fmt`.
- Een script wat een goed opgemaakte figuur met meerdere plots kan maken. Gebruik `plt.figure`, `plt.subplots`, en zorg dat je de figuur de gewenste lay-out kunt geven. (Dit script is vooral als naslagwerk voor jezelf; dus zorg dat je er goede comments bij schrijft.)
- Importeren van data die voldoet aan de afspraken voor lay-out en header uit 4.3.
- Exporteer data in een vorm die voldoet aan de afspraken.
- Een functie die $d * n$ random punten genereert en hier een array van maakt. Als output geeft de functie een array van d kolommen en n rijen. Als optionele argumenten moet de verdeling kunnen worden ingevuld (`binomial`, `normal` of `uniform`), en de minimale en maximale waardes. Bij de default waarde (als er geen `*args, **kwargs` zijn ingevuld) geeft de functie uniform verdeelde random coördinaten in het interval $[0, 1)$.

[30] Denk na over waar je de meeste tijd kunt winnen, is dit in het plotten, het wegschrijven van de data, of in het berekenen van pi? Je kunt `time.time()` op verschillende plekken in je code zetten.

8 Groepsopdracht: Data analyse van A tot Z

In deze week is er tijd om samen met je groepje aan de groepsopdracht te werken. Deze groepsopdracht lijkt qua vorm precies op de individuele eindopdracht voor de tentamenweek. Deze opdracht is daarom te beschouwen als een goede voorbereiding en oefententamen.

De opdracht en bijbehorende data die gebruikt dient te worden komt in een Assignment op Blackboard te staan.

De opdracht bestaat uit twee delen, die je beiden met je werkgroepje samen maakt:

1. Het eerste deel is een programmeeropdracht waarbij jullie voorbeelddata inlezen en verwerken. Je hebt hier ongeveer een week de tijd voor. Het script wat jullie hier schrijven heb je nodig voor het tweede deel. Schrijf het script aan de hand van de opdrachten van het eerste deel. Zorg dat dit af is voor jullie aan het live-deel beginnen.

2. Het tweede deel omvat de interpretatie aan de hand van een aantal vragen. Voor het live-deel krijgen jullie een nieuwe dataset die dezelfde vorm heeft als de test dataset. Over deze nieuwe dataset beantwoorden jullie een aantal vragen.

9 De proef van Millikan

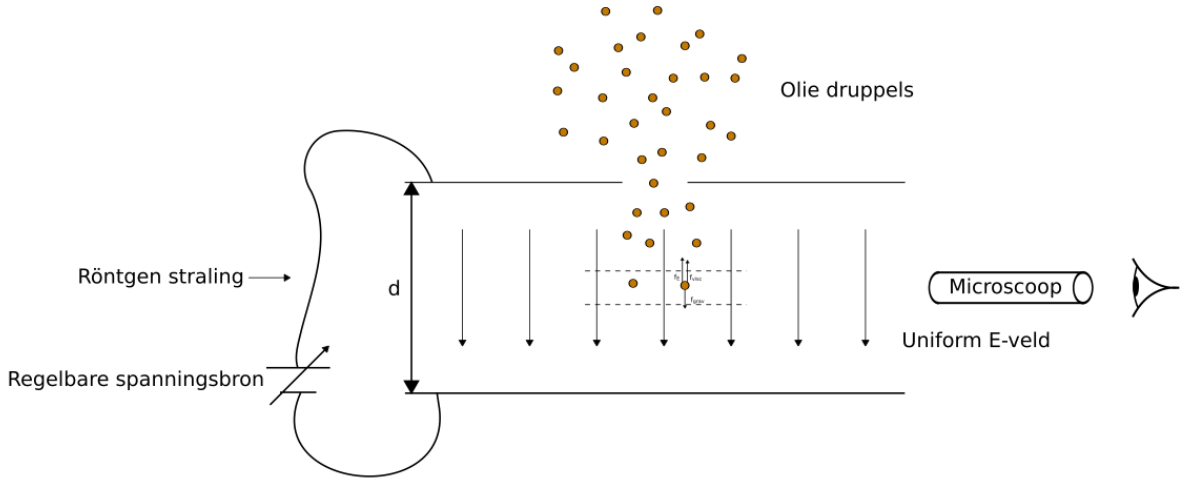
De opgave in dit hoofdstuk is **geen** inleveropgave, maar wel een goede voorbereiding op de eindopdracht. De opdracht en data bij de individuele eindopdracht komen/staan los op Blackboard bij de Assignment.

De structuur van deze opgave, en van de groepsinleveropgave en de individuele eindopdracht is ongeveer hetzelfde: In de opgave lees je een bestand met data in. De structuur van zulke bestanden is steeds hetzelfde. Je doet iets met die data zoals sorteren, filteren en allerlei rekenkundige bewerkingen uitvoeren. Daarna maak je ook een grafische weergave van die data. Er is een stukje theorie die ook aanleiding geeft tot (model) gegevens, die je ook kunt plotten. Je drukt rekenresultaten af. Je maakt eventueel bestanden aan met plaatjes of met tekst die je helpen om het onderliggende fysische systeem beter te begrijpen of te kunnen onderzoeken.

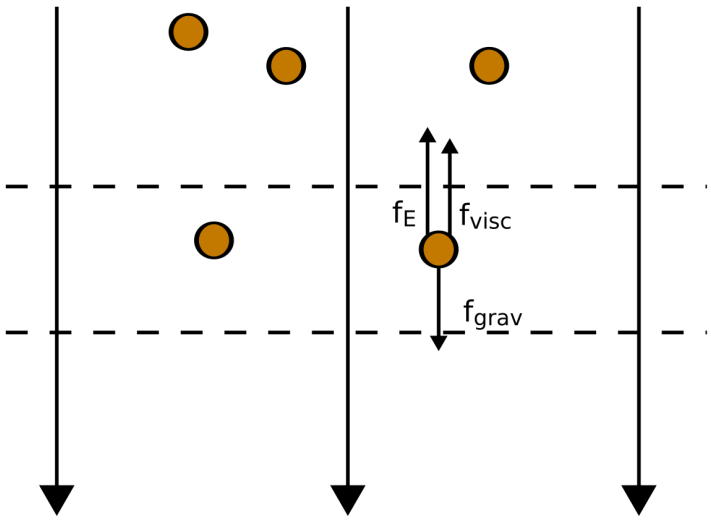
9.1 Achtergrond

Robert Millikan was een Amerikaans fysicus die inmiddels meer dan 100 jaar geleden al in staat was om met zeer grote precisie de lading van een elektron te bepalen. Zijn opstelling is in Figuur 2 schematisch weergegeven. Deze bestond uit twee elektrisch geladen metalen platen. In de bovenste plaat zit een gat waardoor een zeer fijne nevel van oliedruppeltjes kon worden gespoten. De ruimte tussen de twee platen werd zijdelings verlicht. Met een microscoop kon het gedrag van de oliedruppeltjes worden geobserveerd. Niet alle druppels vielen even snel omlaag, en sommigen leken zelfs omhoog te bewegen. Door de wrijving met de lucht of door röntgenstraling van buiten waren de druppeltjes namelijk elektrisch geladen.

De spanning over de metalen platen werd zo ingesteld, dat één druppel op zijn plaats bleef zweven. Deze spanning werd afgelezen. Als we voor het moment aannemen dat de straal van een druppel bekend is, kunnen we zo de lading op die druppel bepalen.



Figuur 2: Schematische weergave van de opstelling van Millikan. De twee condensatorplaten zijn van elkaar gescheiden door isolerend materiaal (lucht). Over de platen wordt een regelbare spanning aangelegd, die door de voltmeter kan worden gemeten. De twee stippellijnen zijn hulplijnen om de snelheid van een druppel te bepalen.



Figuur 3: Een detail van de figuur hierboven, met de elektrische kracht f_{E}, zwaartekracht (f_{grav}) en wrijvingskracht (f_{visc}) op een oliedruppel weergegeven.

In Figuur 3 vind je een schets van de krachtenbalans. Wanneer een oliedruppel stilhangt, wordt de zwaartekracht f_{grav} opgeheven door de elektrische kracht (f_{E}), oftewel

$$f_{\mathrm{E}} = f_{\mathrm{grav}} \tag{4}$$

Vervolgens werd het spanningsverschil opgeheven; het druppeltje viel naar beneden. Door de, voor zo'n klein druppeltje, relatief zeer hoge wrijvingskracht was de valbeweging niet eenparig versneld, maar verliep al na korte tijd met constante snelheid. Wanneer de druppel met constante snelheid beweegt weten we dat de zwaartekracht f_{grav} opgeheven wordt door de wrijvingskracht f_{visc}):

$$f_{\mathrm{visc}} = f_{\mathrm{grav}} \tag{5}$$

De valsnelheid kon worden gemeten met behulp van de twee meetlijnen in het blikveld van de microscoop en de valtijd. Uit deze valsnelheid kan de straal van een druppeltje nauwkeurig worden bepaald (zie verder bij Opdracht 9.2.2).

9.2 Opdrachten

De onderstaande waarden zijn experimenteel bepaald voor een zwevend oliedruppeltje. Deze mogen in je Python-script als parameters worden gebruikt tenzij anders wordt gevraagd.

9.2.1 Verkenning

- Wat is de massa van een (rond) oliedruppeltje? Het volume van een bolletje met straal r is $4/3\pi r^3$, en de massa m is het product van het volume en de dichtheid ρ.

ρ	dichtheid olie	890	kg/m^3
r	straal oliedruppel	0.535×10^{-6}	m
d	afstand tussen de platen	4.46×10^{-3}	m
V	spanning over de platen	52	V
g	valversnelling	9.81	m/s^2
η	wrijvingsconstante	17.1×10^{-6}	kg/m/s

- Geef de formule voor de gravitatiekracht f_{grav} op een druppeltje met massa m. Geef ook de formule voor de elektrische kracht f_{E} op een druppeltje (tussen de platen) met lading q. (Dit is een combinatie van q, V en d). Ga voor jezelf na dat newton en coulomb $\times$ volt / meter dezelfde samengestelde SI-eenheden zijn.

- Wat is de relatie tussen de lading q van het oliedruppeltje en de vijf bovengenoemde grootheden r, ρ, g, V en d als de druppel zweeft? Alleen als de twee hiervoor genoemde krachten f_{E} en f_{grav} even groot zijn kunnen ze elkaar opheffen en kan zo'n druppel blijven zweven.

  Opmerking: Deze twee krachten moeten natuurlijk wel een tegengestelde richting hebben. De richting van de gravitatiekracht kun je niet vrij kiezen, maar je kunt de condensator altijd zo draaien dat de elektrische kracht op een geladen druppel omhoog is gericht. Is de elektrische kracht voor een bepaalde druppel omlaag gericht dan draai je de condensator gewoon om. De elektrische kracht is dan wel omhoog gericht.

- Wat is de waarde van de elementaire lading e, als we aannemen dat er drie elementaire zulke ladingen op het betreffende druppeltje "zitten"?

- Schrijf een (eenvoudige) functie `millikan1` die als invoervariabelen de straal r van een oliedruppel (in meter) en het spanningsverschil V tussen de platen (in V) heeft; als uitvoer geeft deze functie de lading van het oliedruppeltje (in C) . De drie overige grootheden ρ, d en g zijn de optionele argumenten. De default waarde is gelijk aan de waarde die in de tabel hiervoor is gegeven.

  Controleer of jouw functieaanroep `millikan1(0.535E-6, 52)` het antwoord uit de vorige onderdeel weet te reproduceren.

9.2.2 Viscositeit en de grensvalsnelheid

Omdat het moeilijk is om de straal r van elk oliedruppeltje nauwkeurig direct te meten, is de truc bedacht door Millikan om elk druppeltje dan maar te laten vallen. Na het uitzetten van de condensator (f_{E} wordt dan gelijk aan 0) bereikt de druppel vanwege wrijving al snel een grensvalsnelheid v. Je kunt die snelheid meten door de tijd te meten tussen het moment waarop zo'n druppel een start- en een finishlijn onder je microscoop (zie Figuur 3) passeert.

De straal r en de grensvalsnelheid v zijn op de volgende wijze aan elkaar gekoppeld (deze relatie wordt hier niet afgeleid):

$$v = \frac{2\rho g}{9\eta} r^2 \tag{6}$$

Hierin zijn ρ en g als voorheen gedefinieerd, en η is een uit andere experimenten bepaalde wrijvingsconstante van lucht. De numerieke waarde is in de voorgaande table gegeven. Een vergelijking voor de wrijvingskracht f_{visc} is dus niet nodig.

Is de druppel de finishlijn gepasseerd dan wordt er een spanning over de condensator gezet, zodanig dat de druppel tot boven de startlijn stijgt. Daarna wordt (opnieuw) de spanning gemeten die nodig is om hem daar een tijdje te laten zweven. Vervolgens kan de spanning er weer afgehaald worden, en de snelheidsmeting onder de microscoop opnieuw uitgevoerd worden, etc. Dat hele proces kan voor alle oliedruppels in het beeldveld van de microscoop worden herhaald. Op deze manier kunnen dus veel metingen worden verzameld.

- Hierboven is de relatie gegeven tussen de grensvalsnelheid v en de straal r van een oliedruppel. Met weinig rekenwerk kun je die formule omschrijven als $r = \cdots$.
- Schrijf een variant op de functie `millikan1` en noem die gewoon `millikan`. Deze functie heeft als invoervariabelen de grensvalsnelheid v van een oliedruppel en het spanningsverschil V tussen de platen. Als uitvoer geeft deze functie wederom de lading van het oliedruppeltje. De vier overige grootheden ρ, d, g en η zijn de optionele argumenten. De default waarde is gelijk aan de waarde die in de tabel hiervoor is gegeven. In plaats van het elimineren van r ten gunste van v, is het raadzaam om juist een *lokale* variabele r te definiëren binnen je functie `millikan` die zijn waarde krijgt volgens aan de zojuist gevonden formule.

9.2.3 Meetgegevens inlezen, functie gebruiken, en grafische weergave

We hebben nu een functie `millikan` gemaakt die de lading van een oliedruppeltje kan bepalen aan de hand van de grensvalsnelheid v van het druppeltje en de spanning V die nodig is om het druppeltje te laten zweven tussen twee condensatorplaten. In het bestand http://nspracticum.science.uu.nl/DATA2018-2019/DATA-Py/Databestanden/Millikan.dat zijn 1000 (gesimuleerde) metingen opgeslagen. Elke meting bestaat uit een waarneming van het spanningsverschil V en de grensvalsnelheid v (beide in SI eenheden). Met behulp van deze data kan nu bepaald worden wat de quantumlading van één enkel elektron is.

- Lees het bestand in, zodat je een `numpy`-array met afmeting (1000,2) krijgt.
- Er zijn 1000 metingen, elke meting bestaat uit 2 getallen: de spanning V en de snelheid v. Inspecteer de data door deze te plotten met `plt.scatter`.[31] Kies zelf geschikte waarden voor het bereik langs de x- en y-as, als de automatische keuze je niet bevalt. De set punten met $V = 9999$ hoeft niet zichtbaar te zijn. Als het goed is zie je een paar puntenwolken; misschien hebben de punten in zo'n wolk wel iets gemeenschappelijk?
- Noem de eerste kolom van de meetdata `V` (of `spanning`) en de tweede kolom `v` (of `snelheid`). Bereken nu met één regel Python code (`q = millikan(V,v)`) de lading die hoort bij elk meetpunt.
- Als je een file inleest moet je weten wat de betekenis van de diverse items is. Als het goed is bemerk je dat 53 meetpunten liggen bij $V = 9999$. Die 9999 is een *tag* om aan te geven dat er geen potentiaalinstelling te vinden was waarmee het druppeltje stilgezet kon

[31] Of met bijvoorbeeld `plt.plot(x,y,'o'`, zodat je punten als losse punten plot.

worden; zoiets staat in de heading van het bestand. Het is daarom veilig om te stellen dat een druppel in zo'n geval géén lading heeft.

- Wijzig de al berekende array `q` zodat voor meetpunten met $V = 9999$ de waarde $q = 0$ wordt ingevuld ongeacht de waarde van v. Gebruik een masker, zie ook Hoofdstuk 3.

- Plot de lijst met gevonden waarden voor de lading met `plt.hist`. Om af te komen van factoren 10^{-19} langs de x-as deel je alle ladingen q door een *schatting* van de eenheidslading. Kies voor die schatting de waarde $q_e = 1.6 \times 10^{-19}$C. Zorg ervoor, door de optionele variabelen `bins` en `range` geschikt te kiezen, dat $q/q_e = 0$ (of een ander geheel getal) altijd in het **midden** van een bin terecht komt, en niet juist op de grens!

9.2.4 Berekening van de eenheidslading

In het histogram dat hiervoor is getekend zie je een aantal pieken, die corresponderen met n maal de eenheidslading (gequantiseerd). De eerste piek is weinig bruikbaar omdat je uit 0 keer de eenheidslading niet kunt achterhalen hoe groot die eenheidslading is. De pieken met één en 2 eenheidsladingen zijn heel mooi van elkaar gescheiden. Deze twee pieken gebruik je om berekeningen te doen. De overige pieken zijn minder bruikbaar omdat ze niet zo goed gescheiden zijn, en het dus niet *zeker* is of zo'n meetpunt correspondeert met bijvoorbeeld 3 of met 4 eenheidsladingen.

Ter illustratie in Figuur 4 is een simulatie uitgevoerd, waarbij van elke druppel bekend is hoeveel eenheidsladingen deze bevat.

Door een verstoring aan te brengen, waardoor q/q_e niet langer heeltallig is, wordt het experiment nagebootst; niet alle druppels zijn exact even groot! In de figuur kun je duidelijk zien dat ondanks de verstoring alle groene punten met 2 eenheidsladingen netjes binnen het gebied $1.5 < q/q_e < 2.5$ liggen, maar ook is te zien dat er één punt is met 4 eenheidsladingen, waarvoor de verstoorde waarde ruim onder de 3.5 terecht komt. Doordat er hier sprake is van een simulatie is bekend dat zo'n punt hoort bij de set punten met 4 eenheidsladingen. Maar als die informatie er niet is, dan is het heel verleidelijk om zo'n meting maar te rangschikken bij de punten met 3 eenheidsladingen. Hierdoor wordt de analyse onbetrouwbaar.

- Selecteer uit de berekende set getallen q/q_e die waarden die liggen tussen 0.5 en 1.5. Deze grenzen zijn niet heel erg belangrijk, maar wat wel belangrijk is dat je op deze manier alle meetpunten van de tweede piek (de eerste piek bevat de metingen met $q = 0$) gebruikt en geen andere! Gebruik hiervoor wederom een masker.

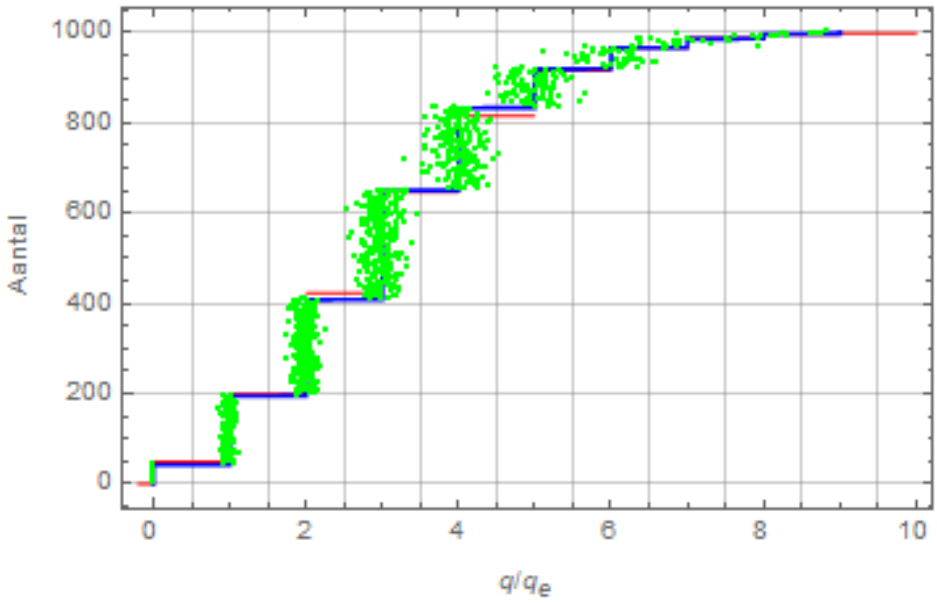
- Bereken van deze geselecteerde set getallen het gemiddelde en gebruik dit om een nieuwe schatting te maken voor q_e. Dus beter dan de initiële schatting $q_e = 1.6 \times 10^{-19}$C.

- Doe hetzelfde voor de set getallen met $1.5 < q/q_e < 2.5$.

- Combineer beide schattingen voor q_e (dus die met één en die met 2 eenheidsladingen) om zo te komen tot een zo goed mogelijke schatting (waarde plus onzekerheid) van de elementaire lading (in C).

Het feit dat het gemiddelde aantal eenheidsladingen per oliedruppel niet exact gelijk is aan 3, zoals in Figuur 4 wordt aangenomen, heeft geen invloed op de analyse. Ook is het voor de

analyse niet erg belangrijk dat dit aantal verdeeld is volgens een Poisson distributie. Random verdelingsfuncties (zowel discrete als continue) komen ter sprake in het laatste hoofdstuk van deze cursus.



Figuur 4: Simulatie van de resultaten van Millikan's experiment. In het rood de CDF, de cumulatieve distributie functie (maal 1000) behorend bij de Poisson verdeling met $n = 3$ eenheidsladingen. In het blauw een simulatie van de lading op 1000 druppels, die een Poissonverdeling met $n = 3$ hebben, gesorteerd op grootte. In het groen: de gehele waarde van q/q_e is verstoord door vermenigvuldiging met een normaal verdeeld random getal (met gemiddelde $\mu = 1$ en standaarddeviatie $\sigma = 0.05$).