Dictaat bij studieonderdeel:

Imperatief programmeren

PO, SE

Python x Meet je leefomgeving

2021-2022

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

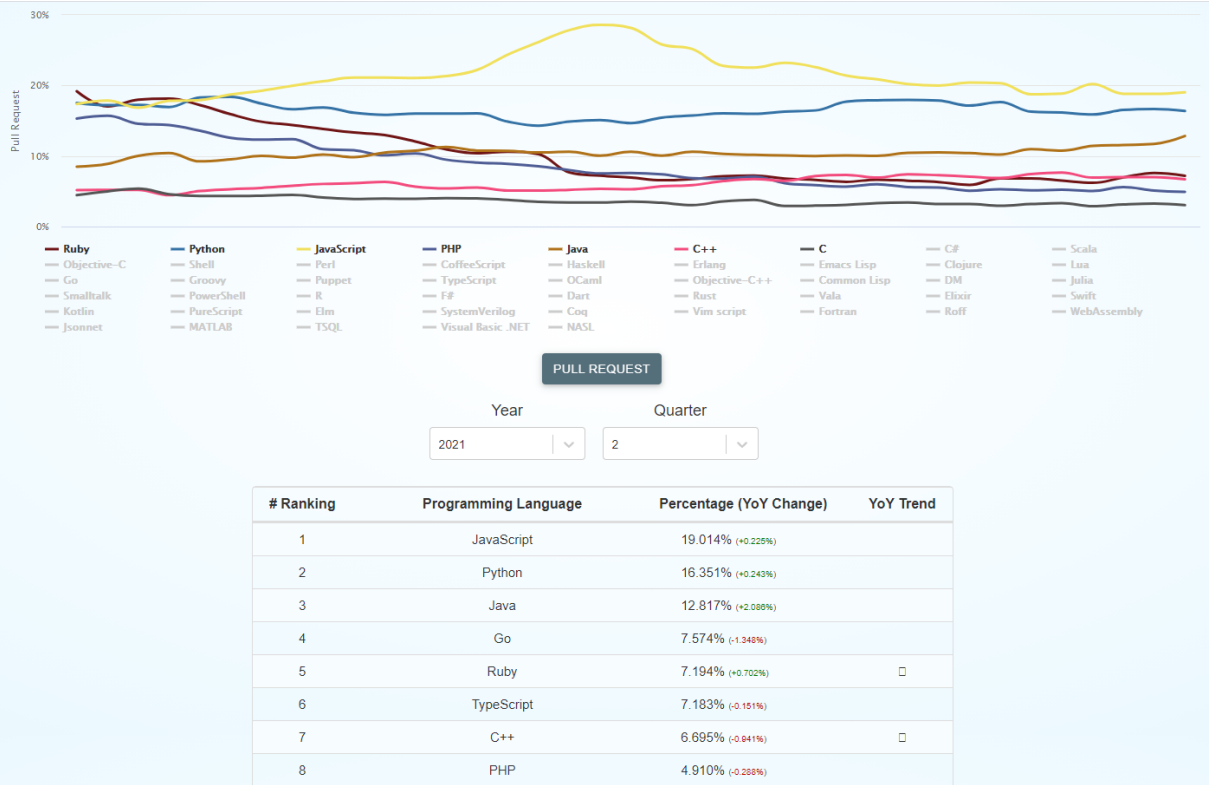
Inhoudsopgave

1 Basisvaardigheden Python

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

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

- Het is gratis en zelf aan te passen.
- Het is eenvoudig om de basis onder de knie te krijgen.
- Er zijn heel veel tutorials en voorbeelden op internet.
- Je kunt er heel veel verschillende dingen mee doen.



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

In deze periode komen de meest nuttige eigenschappen van Python aan de orde. Als je het begint te begrijpen, ga je de lol van programmeren vanzelf inzien. Maar zoals je eerst letters moet leren voordat je kunt lezen, moeten we beginnen met de wat taaiere kost waarvan het nut later duidelijk wordt: hoe maak en bewerk je een script, wat zijn datatypes, functies, variabelen, etc.

Een goede programmeur programmeert gestructureerd en begrijpelijk. Wat moet je hiervoor concreet doen?

- Volg conventies: gebruik variabelen namen die aansluiten bij de inhoud van je programma: een lijst met temperatuurwaarden noem je niet `meuk`.
- Structureer je script: voeg scheidingen toe, definieer *secties* met lege regels en kopjes (zoals je ook in normale tekst doet om hem leesbaar te houden).
- Voeg uitleg toe, vooral wanneer een stuk code moeilijk is. Hoe dit kan worden gedaan wordt later in deze reader behandeld.

Een goede stelregel is: *Hoe moet ik een script schrijven zodat ik het zelf snel weer begrijp als ik het over een jaar open om te gebruiken*? In de beoordeling van deze periode 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

We gebruiken op het Ichthus het programma Spyder om in te programmeren. De interface van Spyder wordt weergegeven in Figuur 2. Deze bestaat standaard uit één window met drie schermen: de Editor, Variable explorer en Console. Elk van deze heeft zijn eigen functie:

- Het scherm links is de Editor. Hierin schrijf je de Python code, ook wel script genoemd, die een berekening of taak uitvoert.
- De Explorer rechtsboven heeft vier tabbladen. Tab 'Files' laat je scripts zien. In de Tab 'Variable Explorer' wordt de waarde van de bestaande variabelen weergegeven nadat je een script hebt uitgevoerd. 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 op te zoeken.
- De Console rechtsonder is het gedeelte waarin informatie over het runnen (of *draaien*) van je script verschijnt. Ook de uitvoer (*output*) komt hier terecht.

Oefenopdracht: Hello World

Typ de volgende regel tekst in de Editor:

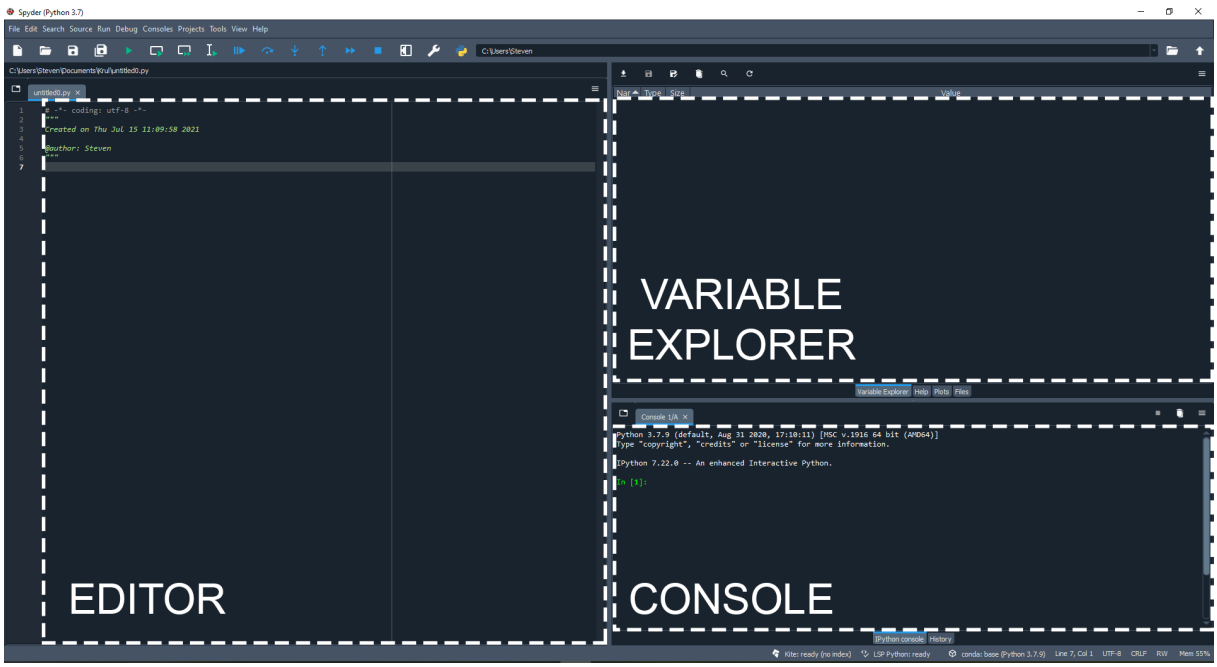
```
print('Hello, world!')
```

Sla je bestanden met een duidelijke naam op in jouw GitHub map. Dit betekent dat de naam van de opgeslagen bestanden duidelijk maakt wat het script doet, of waar het onderdeel van is. Je kunt het beste alleen letters, cijfers en eventueel normale (-) of liggende (_) streepjes (geen spaties!) te gebruiken. Sla je bestand op met de naam `h1_1-helloworld.py` en run het script. Dat doe je door op de groene play-knop (of de F5-toets) in de balk bovenin te klikken. Druk vervolgens eenmalig op Run.

Vraag: Wat is de functie van het commando print? Kijk hiervoor in de console rechtsonderin.

Tijdens de periode zul je regelmatig (voorbeeld)scripts zien in deze bundel. Die kun je het beste openen via de hyperlink erboven. Sla deze als je ze nodig hebt op in je eigen map met bestanden en open deze vanuit Spyder op dezelfde manier waarop je dat in *Word* zou doen[1].

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



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

1.2 Eenvoudige rekenkundige operaties

Met Python kun je makkelijk rekenen. Hiervoor gebruik je normale tekens: optellen (+), aftrekken (−), vermenigvuldigen (∗), delen (/) en (let op!) machtsverheffen (∗∗).

```
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. Dit legt nadruk op de rekenregels. Bijvoorbeeld:

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

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

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

Oefenopdracht: Python als rekenmachine

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

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

De zwaartekracht $g = 9.81\,\mathrm{m/s^2}$. Maak gebruik van bovenstaande formule en vul de getallen in. Let op: komma-getallen noteer je in het Engels met een punt (dus het is echt 9 punt 81).

1.3 Variabelen

1.3.1 Implementatie

Net als wiskunde 'op papier' is het handig om van variabelen als letters gebruik te maken in plaats van de getallen gelijk in te vullen. Dit is ook zo bij programmeren en zorgt voor overzicht als je scripts langer en moeilijker worden.

Bij de vorige oefenopdracht moet je meerdere getallen aanpassen als je het tijdstip verandert. Maar veel handiger en sneller is het gebruik van *variabelen* in Python. Let op de volgorde waarin je de variabelen en de formule in je code opschrijft. Een script wordt (best logisch) van boven naar beneden uitgevoerd. Om berekeningen te kunnen doen met variabelen, moeten deze eerst gedefiniëerd worden. Anders krijg je errors.

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

Tip: Namen voor variabelen kun je zelf kiezen. De enige voorwaarden zijn dat de namen moeten beginnen met een letter, en geen spatie of wiskundig teken mogen bevatten. Kies voor een variabele altijd een naam die past bij de betekenis ervan in jouw script. Noem een lijst van temperaturen bijvoorbeeld `temp` of `temperatuur`.

Oefenopdracht: Gebruik van variabelen

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

1.3.2 Commentaar

Vaak is het voor een buitenstaander een raadsel wat een variabele betekent als hij het script niet kent. Dit is één van de redenen om in een script *commentaar* toe te voegen. Het meest eenvoudig is het om hiervoor de hashtag te gebruiken, want alle tekst op een regel na `#` wordt genegeerd bij het runnen van een script.

De formule voor het vermogen van een apparaat kan bijvoorbeeld op de volgende manier verduidelijkt worden:

```
# vermogen van de wasmachine
U = 230    # spanning van het lichtnet in V
I  = 2     # stroomsterkte over het apparaat in A
P = U*I    # formule voor het vermogen
print(P)
```

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

1.3.3 Datatypen

In alle bovenstaande opdrachten zijn de variabelen hele getallen. Daarnaast kun je gebruik maken van stukken tekst, kommagetallen of `True`/`False`. Dat zijn allemaal *datatypen*. Hieronder de meest gebruikte, met hun officiële naam:

int Gehele getallen: `3 300 20`

In Python (en andere programmeertalen) is **int** het datatype dat gebruikt wordt voor gehele getallen. Zie bovenstaand voorbeeld (`a = 1` etc). Je kunt ook een kommagetal gebruiken, maar let op: dit is niet hetzelfde als afronden op gehele getallen! `int(3.9) = 3` en niet een afgeronde 4.

float Getallen met decimalen: `2.3 4.62 100.00`

Het datatype voor decimale getallen is **float**. Floats maak je door een getal met een decimale punt te maken, zoals: `a = 1.5`.

bool Logische waarde: `True` of `False`

Een **boolean** type, dat we later in het hoofdstuk zullen tegenkomen, is een binair datatype dat twee waarden kan hebben: `True` of `False`.

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

Een **string** is het datatype voor tekst. Deze heb je ongemerkt gebruikt in `'Hello world'`!, met `print()`. Om in Python een `string` te maken gebruik je aanhalingstekens. `a = 'tekst'` en `a = "tekst"` zijn allebei toegestaan.

Je kunt het datatype van een variabele altijd bekijken in de Variable Explorer, als dat nodig is.

1.4 Loops

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

1.4.1 Boolean expressions

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

```
a == b  # test of a gelijk is aan b (let op dubbel = teken!)
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
```

Daarnaast kunnen Boolean expressions ook worden gecombineerd door het gebruik van **and** waarbij de formule alleen **True** zal opleveren als aan beide voorwaarden wordt voldaan; of **or** waarbij de formule **True** zal opleveren als aan minstens één van de twee condities wordt voldaan (beide mag dus ook).

```
a > b and c < d
a > b or c < d
```

1.4.2 While

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

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

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

    t = t + dt
```

Net als eerder definiëren we de benodigde variabelen v_0 en g. Daarnaast maken we een variabele aan voor de tijd t en kiezen we de grootte van de tijdstappen dt die we willen maken voor t. Zolang (*while*) $t \leq 5$ wordt de hoogte y berekend en worden t en y geprint.

Zolang $t \leq 5$ wordt aan de conditie voldaan en wordt de code in de functie van de **while** uitgevoerd. Om door te gaan naar de volgende tijdstap tellen we een stukje tijd dt op bij t.

Net als bij het gebruik van **def** moeten de regels code binnen de **while** *loop* beginnen met een *indent* om aan te geven dat die regels onderdeel zijn van deze *loop*.

Een veelvoorkomende fout bij een **while** is dat de voorwaarde altijd **True** blijft, waardoor het script oneindig doorgaat. Dit gebeurt bijv. als je de regel `t = t + dt` vergeet of verkeerd invoert, of als je de regel wel goed overtypt maar de indent vergeet! Je kunt een script stoppen in Spyder door op de rode knop rechts boven in de Console te drukken.

1.4.3 If

Een tweede soort constructie is de zogenaamde *if - else* constructie. Aan de hand van een voorwaarde (*conditie*) wordt een bepaald deel code wel of niet uitgevoerd. Een variant hierop is de *if - else if - else* constructie, waarbij er meer dan 2 mogelijkheden voorkomen. Je kunt zoveel *elif*'s toevoegen als je wilt.

Let op: ook hier gebruik je *indents*. In Python wordt *else if* afgekort tot **elif**.

```
if gewicht < 40:              # 0 — 39.99999999 kg
    print("Ondergewicht!")
elif gewicht < 80:            # 40 — 79.99999999 kg
    print("Goed gewicht")
else:                         # 80 kg en meer
    print("Overgewicht!")
```

else heeft *nooit* een voorwaarde! 'Anders' zijn alle overige opties, zonder een voorwaarde. Wil je een voorwaarde gebruiken, dan moet je **elif** gebruiken.

Oefenopdracht: Testen van combinaties van Boolean expressions

- Neem het voorbeeld van de BMI over. Voeg naast gewicht nog een voorwaarde toe: lengte. Je hebt alleen ondergewicht bij 40 kilo als je langer bent dan een bepaald aantal centimeter. Bedenk er zelf maar wat van, maar test je functie ook!

1.4.4 For

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

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

Een handigere manier om dit te doen is met een **for**-loop. Hierbij zit de 'teller' automatisch ingebouwd: de loop wordt herhaald voor elk element in de voorwaarde. Elke **while**-loop is om te schrijven in een **for**-loop, maar soms *voelt* de één logischer, en is die ook makkelijker. Maar gebruik waar mogelijk een **for**-loop. Dit voorbeeld is toch best wat korter en simpeler:

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

Oefenopdracht: for i in range()

- Voer het voorgaande stukje code uit.
  Vraag: welke getallen zitten er verborgen in **range(5)**? En wat als je er **range(10)** van maakt?

Je kunt in de **range**-functie één, twee of drie getallen invullen. Zie onderstaande opties:

```
range(5)          # ziet eruit als [ 0, 1, 2, 3, 4 ]
                  # oftewel: van 0 TOT 5 (begint standaard bij 0)
range(2,8)        # ziet eruit als [ 2, 3, 4, 5, 6, 7 ]
                  # oftewel: van 2 TOT 8
```

```
range(10,40,5)  # ziet eruit als [ 10, 15, 20, 25, 30, 35 ]
                # oftewel: van 10 TOT 40 met stappen van 5
```

1.5 Slotopdrachten

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

1.5.1 Slotopdracht 1: Wortels trekken

Schrijf een Python-script dat aan jou als gebruiker om input vraagt en vervolgens wel of niet een wortel trekt, om daarna weer een nieuw getal te vragen etc.

- Allereerst vragen we de gebruiker om input. Zie daarvoor onderstaande regel. Als je het script uitvoert met deze regel, kun je rechtsonder in de console klikken en achter de tekst een getal invullen. Als je op Enter drukt wordt je getal opgeslagen in de variabele `invoer`.
- Deze input is van het datatype **`str`** (string, ofwel tekst). Van tekst kun je geen wortel trekken, dus moet `invoer` eerst omgezet worden in een getal (**`int`**). Kijk in 1.3.3 of zoek op internet een methode om van de input een **`int`** te maken, en sla dit getal op in een nieuwe variabele.
- Maak vervolgens een **`while`**-loop. Deze loop moet stoppen als het ingevoerde getal gelijk is aan 0.
- Check vervolgens of het getal groter is dan nul of kleiner. Als het opgegeven getal inderdaad positief is wordt de wortel[2] van dat getal geprint. Is het opgegeven getal negatief, dan wordt de wortel niet uitgerekend. Je print dan de tekst `"Dombo: Positief svp!"`.
- Vraag vervolgens (heel beleefd) weer om een positief getal. Gebruik de regels waarmee je voor het eerst om input vraagt hier gewoon opnieuw!

```
invoer = input("Geef een positief getal: ")  # input heeft datatype string
```

1.5.2 Slotopdracht 2: Omrekenen van Fahrenheit naar Celsius

Schrijf een script dat voor de temperaturen van -10 tot en met 100 Fahrenheit met stappen van 5 Fahrenheit de temperatuur in graden Celsius berekent. Print voor elke 5 Fahrenheit de temperatuur in Fahrenheit, de berekende temperatuur in Celsius, en de conclusie of het wel of niet vriest. (Hint: controleer dus of de temperatuur in graden Celsius kleiner of groter dan 0 is!)

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

De conversie tussen Celsius en Fahrenheit is

$$C = \frac{5}{9}(F - 32) \tag{2}$$

2 Modules (1): NumPy

In dit hoofdstuk kijken we naar een essentieel onderdeel van het programmeren in Python: het gebruik van *modules*. Modules bevatten functionaliteit die niet in Python zelf zit maar apart toegevoegd moet worden. Oorspronkelijk kon je met Python niet veel meer dan stukken tekst bewerken. In dit hoofdstuk behandelen we `numpy`, dat het makkelijk maakt om met getallen en data te werken.

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

```
import numpy as np
```

Vanaf dit punt wordt naar `numpy` verwezen als `np`. Om functies uit een module te gebruiken wordt de afkorting van de module als voorvoegsel gebruikt, gevolgd door een punt, en dan het commando(= de naam van de functie) uit de module.
Bijvoorbeeld: `np.sin(X)` `#berekent de sinus van X`.

2.1 Arrays maken

Een *array* (Nederlands: reeks) is een verzameling van getallen. In Python kun je die maken als een officiële Python lijst, maar zo'n lijst is erg onhandig. In plaats daarvan gebruiken we veel liever een numpy array.

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

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

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

Code 2.1: code-inc/w2/np˙vb1.py

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

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

# Om een array te maken met allemaal stapjes tussen twee waarden (grid)
# kunnen deze functies gebruikt worden:
```

```
grid1 = np.arange(0.,5.,0.25)       # 0 TOT 5 met stapjes van 0.25
grid2 = np.linspace(0.,5.,num=20)   # 0 TOT EN MET 5, 20 gelijke afstanden
```

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

Oefenopdracht: Maken en bewerken van numpy arrays

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

2.2 Rekenen met arrays

Rekenen met arrays is super makkelijk. Wanneer twee arrays dezelfde vorm (evenveel getallen) hebben, kun je ze getal voor getal optellen door + te gebruiken. Wanneer deze niet dezelfde vorm hebben krijg je een foutmelding. Alle wiskundige formules (`+,-,/,*,**`) werken op deze manier.

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

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

Oefenopdracht: Maken en bewerken van numpy arrays

b) Maak een array met de naam k met daarin de gehele getallen van 1 tot en met 10. Bereken vervolgens k^k. Print en controleer de resultaten daarvan. Valt je iets op?[3]

2.3 Indexing en slicing

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

Code 2.2: code-inc/w2/np˙vb2.py

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

# Wanneer we een enkel element willen selecteren kan dat als volgt:
```

[3] Als er iets niet klopt, geef het aan bij de docent! Die legt het wel uit :)

```
element1 = test_array[0]          # Het eerste element heeft index '0'.
element7 = test_array[6]          # Dit is dus het getal '7'

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

Oefenopdracht: Maken en bewerken van numpy arrays

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

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

2.4 Allerlei functies

Numpy arrays hebben een aantal ingebouwde functies, om bijvoorbeeld het gemiddelde of de som uit te rekenen. Dat kan door `np.functie(array)` te gebruiken. Daarnaast kun je de lengte en het datatype van een array weergeven.

Code 2.3: code-inc/w2/np˙vb3.py

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

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

# Eigenschappen van een array;
ta_lengte = len(test_array)      # Geeft aantal getallen in array (10,)
ta_dtype  = test_array.dtype     # Geeft datatype in array (int32)
```

Oefenopdracht: Maken en bewerken van numpy arrays

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

2.5 Maskers

In sommige gevallen wil je een lijst getallen ook filteren. Je bent bijvoorbeeld geïnteresseerd in alle leeftijden boven de 18 jaar, of alle leerlingen in de bovenbouw. Dit kan aan de hand van een *masker*. De werking van zo'n masker wordt uitgelegd aan de hand van het onderstaande

voorbeeld[4]. Een masker maak je door een voorwaarde op te geven bij een variabele. Alle plekken in de array die voldoen aan die voorwaarde, geven de waarde **True**, alle anderen geven **False**. Een masker pas je vervolgens toe met de blokhaken `[ ]`.

Code 2.4: code-inc/w2/np˙vb4.py

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

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

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

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

Oefenopdracht: Maken en bewerken van numpy arrays

e) Selecteer met behulp van een masker alle oneven getallen uit `test_array` van bovenstaand voorbeeld. Hiervoor kun je goed gebruik maken van de rest-berekening, die je jaren geleden wel eens gezien hebt. Zie dit voorbeeld:

```
# rest(4/2) is 0 (2 past precies 2 keer in 4)
print(4 % 2) # dit print rest(4/2) oftewel 0

# rest(5/2) is 1 (2 past 2 keer in 5, en er blijft 1 over)
print(5 % 2) # dit print rest(5/2), oftewel 1
```

In plaats van een test of de getallen groter zijn dan 5 (zoals in het bestand boven de opdracht), kun je hier dus testen of de rest gelijk is aan 1! Want elk oneven getal gedeeld door 2 levert een rest van 1 op.
Print als laatste ter controle het array met dit masker, zodat je alleen de oneven getallen ziet.

2.6 For-loops

In paragraaf 1.5 is de **for**-loop aan de orde gekomen. Hier komt nu geen nieuwe stof bij, maar in het licht van numpy-array's is het goed om nog een oefenopdracht te maken. Deze helpt ook bij de slotopdracht. Bekijk en run eerst dit voorbeeld:

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

```
data = np.array([1,2,3,4,5,6]) # een test array

# De volgende for-loop print een voor een alle getallen in 'data'
# 'i' is hierbij het getal dat loopt van 0 tot len(data) = 6
for i in range(len(data)):
    print(data[i])
```

Oefenopdracht: for-loops en numpy-arrays

- Maak een array `np.linspace(0,10,num=20)`. Dit zijn dus 20 getallen, oplopend van 0 naar 10.
- Maak vervolgens, net als in het voorbeeld, een `for`-loop met dezelfde print-regel.
- Pas dit vervolgens zo aan, dat je niet elke keer `data[i]` print, maar alle getallen vanaf de eerste plek tot de huidige plek (plek i). De eerste keer print dat alleen lege haken (`[]`), de tweede keer één getal, de derde keer twee, etc.
- Print vervolgens het gemiddelde van die getallen. (De eerste keer is dat dus het gemiddelde van nul getallen: dit levert een waarschuwing op maar geen error. De tweede keer is dat het gemiddelde van één getal, de derde keer van twee getallen, etc.)
- Controleer je loop: het laatste gemiddelde is het gemiddelde van alle getallen behalve de laatste. Dat is dus 4.75.

2.7 Slotopdracht: Voortschrijdend gemiddelde

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

In deze slotopdracht maak je zelf een voortschrijdend gemiddelde.

- Maak een array genaamd `random_array` van 50 random getallen tussen de 0 en 1 (zoek hiervoor in het hoofdstuk).
- Maak een variabele genaamd `periode`, met de waarde 7. Dit is het aantal dagen waarover we het gemiddelde gaan berekenen.
- Maak vervolgens een `for`-loop. Deze start op de dag ná de periode, tot en met de laatste waarde van je array. (Hint: `range(periode,len(random_array))`)
- Bereken per element in de `for`-loop het gemiddelde van de afgelopen 'periode' dagen. Print elke keer het berekende gemiddelde van de afgelopen 7 dagen.
- Wat valt je op als je de periode verhoogt (naar bijvoorbeeld 30)?

3 Modules (2): MATPlotLib

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

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

In dit hoofdstuk staan voor de verschillende mogelijke varianten voorbeelden. Van de variant in paragraaf 3.1 wordt verwacht dat je ze zelf uit je hoofd kunt maken later, maar de varianten in 3.3 en 3.4 kun je uiteraard gewoon elke keer kopiëren bij latere opdrachten.

3.1 Eén grafiek maken

Een grafiek maken in Python hoeft maar een paar regeltjes te kosten. We hebben nodig: een x-as, een formule voor y, en een plaatje om het in te maken. In het onderstaande voorbeeld maken we een array aan voor de x-as: dit is het domein van de grafiek. We kiezen deze van 0 tot 2π. Vervolgens gaan we daar een lijn bij maken, in formulevorm: $y = sin(x)$. Python kent zelf geen sinus, dus moeten we gebruik maken van NumPy om de sinus te kunnen berekenen.

Vervolgens maken we een figuur aan. Dat gebeurt in de regel `fig, ax = plt.subplots()`. `fig` is hierbij de 'omschrijving' van de 'buitenkant' van het plaatje, terwijl `ax` over de 'binnenkant' van het plaatje gaat, oftewel je grafiek. Daarna gaan we de sinus tekenen ofwel *plotten*:

Code 3.1: code-inc/w3/plt˙vb1.py

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

# Maak 2 arrays x en y, waarbij y = sin(x)
x = np.linspace(0,2*np.pi,num=100)
y = np.sin(x)              # Per element in x wordt sin(x) uitgerekend
fig, ax = plt.subplots()   # Maak een nieuw figuur
ax.plot(x,y)               # Plot punten (x,y)

fig, ax = plt.subplots()   # Maak een nieuw figuur
ax.plot(x,y**2)            # Plot punten (x,sin(x)^2)
```

Oefenopdracht: lijnen plotten

a) Kopieer via de hyperlink het bovenstaande voorbeeld en voer het script uit. Let op: er verschijnt rechtsonderin een waarschuwing; deze mag je negeren. Rechtsbovenin zie je twee plaatjes verschijnen.

b) Verander het aantal elementen in `x`. Wat gebeurt er als het aantal elementen in de lijst klein is (bijvoorbeeld 10)?

c) Verander het tweede plaatje zo dat er geen y^2 wordt geplot, maar dat er een cosinus wordt geplot.
Hint: `y2 = np.cos(x)`

d) Maak een extra plaatje door in hetzelfde script nog een keer `fig, ax = plt.subplots()` toe te voegen. Plot in dit plaatje `y` en `y2` tegen elkaar. Dit betekent dat je in plaats van `ax.plot(x,y)` invult: `ax.plot(y,y2)`.

3.2 Grafieken opmaken

Een kale grafiek is niet zo nuttig. Daarom is er ook heel veel opmaak mogelijk in matplotlib. Hieronder zijn een flink aantal opties:

- `ax.set_title("titel")`: voeg een titel toe aan de grafiek.
- `ax.set_xlabel("x-label")`: voeg een titel toe aan de x-as.
- `ax.set_ylabel("y-label")`: voeg een titel toe aan de y-as.
- `ax.set_xlim(x_min, x_max)`: pas het domein van de x-as aan van x_min tot x_max.
- `ax.set_ylim(y_min, y_max)`: pas het bereik van de y-as aan van y_min tot y_max.
- `ax.grid()`: voeg een raster toe aan de grafiek om beter af te kunnen lezen.
- `ax.plot(x,y,label="label")`: voeg een label toe aan de lijn.
- `ax.legend()`: voeg een legenda toe die alle labels toont.
- `ax.set_aspect(1)`: stel de verhouding van de assen in. Als beide assen lopen van 0 tot 1 (of andere dezelfde getallen) krijg je bijvoorbeeld een vierkant plaatje (verhouding 1:1 = 1).

Oefenopdracht: grafieken opmaken

e) We bekijken de volgende simulatie: de Spaanse KNMI heeft sinds 1981 de warmste temperatuur per jaar bijgehouden in Madrid. Een medewerker van het instituut heeft een plotje gemaakt van die data; zie het bestand onder de volgende oefenopgave.
Zijn leidinggevende heeft de volgende eisen gesteld: de figuur moet een algemene titel en twee astitels krijgen. Het domein moet vallen van 1981 tot 2021; het bereik van 40 tot 45 graden Celsius. Daarnaast moeten de meetpunten en de trendlijn aangegeven worden in een legenda, en moet de grafiek goed uit te lezen zijn. Vul het script aan met bovenstaande opties zodat aan de voorwaarden wordt voldaan.

f) Maak de volgende plot: kies t van 0 tot 2π. Gebruik:
`x = 16*np.sin(t)**3`
`y = 13*np.cos(t) - 5*np.cos(2*t) - 2*np.cos(3*t) - np.cos(4*t)`
Zorg voor een goede verhouding tussen de assen.
Tip: maak gebruik van het eerste voorbeeld en pas deze aan.

Code 3.2: code-inc/w3/plt˙vb2.py

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

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

```
trend = np.sqrt(jaren) - 1                          # trendlijn maken

fig, ax = plt.subplots()                # figuur maken
ax.plot(jaren, temp, 'k.')              # meetpunten plotten
ax.plot(jaren, trend, 'r-')             # trendlijn plotten
```

3.3 Grafieken naast óf onder elkaar

Soms is het fijn om twee of meer plots onder elkaar of naast elkaar te laten zien. In onderstaand voorbeeld worden er 3 grafieken naast elkaar gemaakt in hetzelfde figuur.

In plaats van `plt.subplots()` nu leeg te laten, voegen we in hoeveel grafieken we willen maken. De 1 betekent één plot in de hoogte, 3 in de breedte. Zoals hiervoor wordt de hele figuur beschreven met `fig`. De losse grafieken noemen we `ax1, ax2` en `ax3`. Als je twee grafieken zou willen verander je de 3 in een 2, en moet je `ax3` verwijderen. Als je er vier zou willen, maak je van de 3 een 4 en voeg je een `ax4` toe.

Code 3.3: code-inc/w3/plt˙vb3.py

```
# Begin met een subplots.
# fig bevat de informatie van de gehele figuur.
# ax1 t/m ax3 bevatten informatie over de individuele plots.
fig, (ax1, ax2, ax3) = plt.subplots(1, 3)

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

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

# fig.suptitle() is de titel van de hele figuur.
fig.suptitle("Drie plotjes tegelijk")
```

Oefenopdracht: meer grafieken in een figuur

g) Voer bovenstaand script ook uit en kijk hoe het eruit ziet. Wissel de 1 en 3 eens om en kijk wat er dan gebeurt!

h) Het staat best lelijk dat de getallen op de y-as van het tweede en derde grafiekje overlappen met de vorige. Los dit op door de volgende aanpassing in de eerste regel:
`fig, (ax1, ax2, ax3) = plt.subplots(1, 3, sharex=True, sharey=True)`
Hiermee forceer je dat de x- en y-as van alle grafiekjes automatisch hetzelfde zijn en de getallen dus verdwijnen omdat ze overbodig zijn.

i) Voeg ook aan het tweede en derde grafiekje een grid toe. Geef daarnaast elke plot een label en ook elk figuur een legenda. Let op: je moet elke handeling voor elke grafiek los doen, dus voor `ax1, ax2` en `ax3`.

3.4 Grafieken naast én onder elkaar

Het is uiteraard ook mogelijk om plots onder én naast elkaar te zetten. In het volgende voorbeeld worden 2 bij 2 plots gemaakt. Let op! De plots zitten nu niet meer in de losse namen `ax1, ax2` en `ax3`, maar ze zitten alle vier in de algemene naam `axarr`. Net zoals bij een array een plek opgegeven wordt met bijvoorbeeld `[1]`, kun je een specifiek plot ophalen met de blokhaken. Het eerste getal tussen de blokhaken staat daarbij voor de rij, het tweede getal voor de kolom. Zie het voorbeeld.

Code 3.4: code-inc/w3/plt˙vb4.py

```
# Begin met een subplots.
# sharex=True zorgt dat de x-as gedeeld wordt voor alle grafieken
# sharey=True idem voor de y-as
fig, axarr = plt.subplots(2, 2, sharex=True, sharey=True)

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

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

fig.suptitle("Vier plotjes tegelijk")
```

Oefenopdracht: opmaak van meer grafieken

j) Voeg aan elke grafiek een grid toe. Let op dat je hier niet gebruik maakt van `ax1` etc., maar dat nu vervangt door `axarr[0,0]` etc.

3.5 Slotopdracht: Heel veel plotjes

- Maak een figuur van 2-bij-2 plots waarin net als in bovenstaand voorbeeld `sharex` en `sharey` gebruikt worden.
- Maak een array `x` met een ruim aantal getallen, oplopend van 0 tot en met π (niet random dus!). Hoeveel is genoeg? Denk terug aan de sinus met 10 of 100 getallen.
- Plot vervolgens per plot de volgende lijnen:
  1. $sin(x)$ en $cos(x)$
  2. $sin(x)^2$ en $cos(x)^2$
  3. $x^{0.5}$ en x^2
  4. $1/x$ en $log(x)$

  De logaritme kun je plotten met behulp van `np.log(x)`.
- Zorg ervoor dat het bereik op de x-as (het domein) loopt van 0 tot π voor alle plots.
- Voeg uiteindelijk ook labels en legenda's toe, en maak hem zo netjes mogelijk met de dingen die je in dit hoofdstuk geleerd hebt.

4 Importeren en exporteren

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

4.1 Afbeeldingen exporteren

In het vorige hoofdstuk heb je al geleerd om plaatjes te maken. Figuren staan normaal gesproken alleen in Spyder, en kunnen niet direct gebruikt worden in een verslag, artikel of rapport. Daarvoor moet je de afbeelding eerst opslaan. Hiervoor kan je gebruik maken van `savefig()`.

De standaard locatie waar de afbeelding wordt opgeslagen is de map waar het script is opgeslagen!

De belangrijkste opties die je aan deze functie kunt meegegeven zijn de naam + bestandstype van de afbeelding, en de resolutie. De naam kun je uiteraard vrij kiezen; als bestandstype kan er worden gekozen voor `jpg`, `png` en `pdf` (wij gebruiken hier alleen .png). De resolutie van deze afbeeldingen wordt gespecificeerd in *dots-per-inch* (`dpi`). Een gemiddeld beeldscherm heeft zo'n 140 dpi, maar voor een plaatje is meer dpi zeker fijn.

Met behulp van `figsize=[breedte, hoogte]` kun je instellen hoe groot je plaatje is, en welke verhoudingen hij heeft. Hoe groter de breedte en hoogte, hoe groter het bestand. Deze functie kun je invoegen in `plt.subplots()`.

Code 4.1: code-inc/w4/export˙vb1.py

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-2*np.pi, 2*np.pi, num=1001)
y = np.sin(x)
fig, ax = plt.subplots(figsize=[8,4])
ax.plot(x,y)
fig.savefig('sinus.png', dpi=300)
fig.savefig('sinus2.png', dpi=1200)
```

Oefenopdracht: plots opslaan

- Open het bovenstaande voorbeeld-script `export_vb1.py`. Maak verschillende plaatjes door de volgende instellingen te gebruiken:
  1. Het aantal dots-per-inch. Kies bijv. 300 of 1200.
  2. De grootte van het plaatje. Kies bijv. `[2,1]` of `[20,10]`.
- Open de plaatjes en kijk naar de kwaliteit. Gebruik de zoom-functie! Kijk ook naar de bestandsgrootte (in kilobytes) van de aangemaakte bestanden. Trek je conclusie!
- Sla de figuur van slotopdracht hoofdstuk 3 ook op.

4.2 Bestanden 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. Daarvoor zijn allerlei instellingen, maar deze periode zien de bestanden er allemaal hetzelfde uit:

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 een enter, of in programmeertaal: '`\n`'.
2. Elke regel bestaat uit één of meerdere getallen die van elkaar gescheiden door komma's. Het scheidingsteken heet de *delimiter*.
3. Een commentaarregel is herkenbaar aan het `#` teken op de eerste positie. Precies zoals in een Python script. Commentaarregels aan het begin van een bestand heten ook wel de *header*.

Het inlezen van bestanden die numerieke data bevatten in de vorm van een array (rijen en kolommen) kan heel makkelijk gebeuren met de `numpy` functie `genfromtxt()`. Deze functie negeert regels die beginnen met een `#` en dat komt goed uit, want dat is meestal toch vooral commentaar. Daarnaast kun je de *delimiter* specificeren, hier de komma dus. Het weergeven van bijvoorbeeld twee kolommen uit de ingelezen data in een xy-plot stelt dan weinig meer voor. Zie volgend voorbeeld.

Code 4.2: code-inc/w4/genfromtxt˙vb1.py

```
import numpy as np

# de naam van het bestand
file = 'vb1.txt'

# lees de data van dit komma-gescheiden bestand
data = np.genfromtxt(file, delimiter=',')

# als het goed is kun je zien dat data een array met floats is
# en de size ervan is (20,3); zie de 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 = data[:,1]   # selecteer kolom 1
y = data[:,2]   # selecteer kolom 2
```

Oefenopdracht: plotten van geïmporteerde data

- Open bovenstaand script. Open ook deze link. Kopieer de inhoud ervan naar het programma Notepad, en sla het bestand op onder de naam `vb1.txt`, *in dezelfde map als je script*! Run daarna het script, bekijk de werking en check in de Variable Explorer hoe de data eruitziet. Plot vervolgens x en y zoals je in het vorige hoofdstuk hebt gedaan.

4.3 Slotopdracht: De weerstations van de KNMI

In deze slotopdracht gaan we aan de slag met (gesimuleerde) meetdata van de KNMI. Het KNMI heeft in totaal 48 meetstations verspreid over Nederland. Elke paar seconden doet zo'n meetstation een meting en rapporteert de waarden naar de servers van de KNMI[5]. Op basis hiervan worden allerlei voorspellingen gedaan, zoals het mooie overzicht dat Buienradar altijd maakt. De voorspellingen zijn niet altijd zo accuraat, maar daar kijken we hier niet naar: we zijn geïnteresseerd in de gemeten data.

Elk meetstation rapporteert in deze opdracht gemiddeld elke minuut de temperatuur, maar soms duurt het wat langer, soms wat korter, en soms is het internet wat trager dan anders. Dit heeft tot gevolg dat de gemeten waarden elke keer in een andere volgorde binnenkomen op de servers van de KNMI.

Aan jou de taak om de gemeten temperaturen te importeren, te sorteren, te visualiseren en als laatste te exporteren. Zo combineren we alle voorgaande kennis, en bereid je je al goed voor op de eindopdracht die hier erg op lijkt.

1: Importeren van data

Bekijk het volgende bestand: data-h4. Kopieer net zoals in de oefenopdracht de inhoud (je kunt een van de knoppen rechtsbovenin gebruiken om alles te kopiëren) en sla het (via Notepad weer) op onder de naam `data-h4.txt` (wederom in dezelfde map als je script).

Gebruik `np.genfromtxt` met de komma als delimiter, zodat je de meetgegevens in het bestand kunt inlezen.

2: Sorteren op meetstation

De data bestaat uit twee kolommen: de eerste bevat het nummer van het meetstation, de tweede de gemeten temperatuur. Elke vier opeenvolgende metingen zijn altijd van de vier verschillende weerstations: het kan niet voorkomen dat er eentje ontbreekt. Alleen de volgorde verschilt. Je wordt aangeraden om gebruik te maken van `np.argsort()`. Hieronder een voorbeeld:

```
# nummers van de meetstations
stations = np.array([1,4,3,2])

# de temperaturen die bij de stations horen, op dezelfde volgorde
temperaturen = np.array([20.0, 20.5, 22.6, 21.2])

# [0 3 2 1] is de goede volgorde: het eerste getal staat goed,
# het tweede getal staat achteraan, etc.
volgorde = np.argsort(stations)

# sorteer de temperaturen met behulp van de volgorde
temperaturen = temperaturen[volgorde]
```

Dit stukje code sorteert vier waarden op basis van de nummers van de weerstations. In het bestand staan er echter bijna 2000 keer 4 metingen onder elkaar. Je zult dus elke keer de

[5] Bron: KNMI

temperaturen moeten sorteren in blokjes van vier, en dan doorgaan naar de volgende vier. Denk hiervoor terug aan de slotopdracht van hoofdstuk 2: hoe selecteer je elke keer vier elementen?

Let op: nu schuif je niet telkens één plekje op, maar vier plekken. Maak daarom gebruik van de volgende regel:

```
for i in range(0, len(data), 4)
```

Zo maak je elke keer een stap van 4!

3: Reshapen van data

Je data is nu gesorteerd, maar alles staat op één rij achter elkaar. Om de data te kunnen plotten, moeten we nu de data zo aanpassen, dat je één hele rij of kolom kunt selecteren waar alle temperaturen van één weerstation staan. Daarvoor kun je goed gebruik maken van `np.reshape(rijen, kolommen)`:

```
temperaturen = temperaturen.reshape(int(len(temperaturen)/4), 4)
```

Dit is een best complexe regel: we zeggen dat de vorm van `temperaturen` zó moet zijn dat er 4 kolommen naast elkaar staan, en het aantal rijen is het totaal aantal meetpunten gedeeld door 4, omdat er over 4 meetstations gemeten wordt. Per weerstation zijn er dus `len(temperaturen) / 4` meetpunten. En omdat het aantal rijen een heel getal moet zijn, zeggen we specifiek dat het een `int` moet zijn.

4: Plotten per weerstation

Maak nu per weerstation een plot van de temperatuur. Gebruik als x een array van 0 tot 24 (uur), met het aantal meetpunten van één weerstation als het aantal getallen in dat array (hint: maak dus gebruik van `np.linspace()`!). Gebruik als y uiteraard een kolom uit `temperaturen`.

Je mag zelf kiezen of je vier losse figuren maakt, of een 2-bij-2 subplot pakt van het vorige hoofdstuk. Zorg in ieder geval dat je al je figuren opslaat!

5 Functies

Je script kun je automatiseren door variabelen en de handelingen die je vaak herhaalt, in een *functie* te zetten (een functie te definiëren). Een functie is een stukje tekst dat aan de hand van input één of meerdere handelingen kan doen.

Functies worden vaak gebruikt om te voorkomen dat blokken code meerdere keren in een script herhaald worden. Hiermee zorg je ervoor dat, wanneer er een aanpassing gedaan moet worden, dit maar op één plek hoeft te gebeuren. Ook blijft je script overzichtelijk, ook al maak je hem steeds langer of complexer.

5.1 Constructie van functies

We gebruiken de formule $y = ax^2 + bx + c$ om de werking van *functies* uit te leggen. In het script hieronder zie je een voorbeeld van een functie die de waarde van de formule bereken.

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

Een Python functie begin je met het woord **def** ('definition'), gevolgd door de naam van de *functie*. Daarna volgen tussen ronde haakjes de variabelen die door de functie moeten worden gebruikt als input en waar je in de formule mee rekent (bijvoorbeeld a, b, c en x in bovenstaand voorbeeld). De regel wordt afgesloten met een dubbele punt om aan te geven dat wat er op de volgende regels staat de echte inhoud is.

Daarna komt de code van de *functie* zelf. Let op dat de regels binnen een definitie een indent nodig hebben, net als bij de loops en if/else-statements.

Als laatste *kun* je de definitie afsluiten met een **return**, maar dat kan verschillen. **return** geeft de waarde terug die je erachter zet, in bovenstaand voorbeeld is dat de waarde van y. Als je wilt weten wat de uitkomst is, willen we de waarde van y graag 'terugkrijgen' en gebruik je **return**. Maar als je in een functie alleen zet dat er een lijn geplot moet worden in een grafiek, hoef je geen getal terug te krijgen o.i.d., en sla je deze regel over.

5.2 Gebruik van functies

Om een functie te gebruiken typ je de naam van de functie, met de waarden die je als input wilt gebruiken. Daarbij zijn er twee smaken:

1. Je geeft de argumenten op volgorde mee: de waarde van a als eerste, etc.
2. Of: je geeft specifiek de labels a, b, c en x aan en dan mag je elke volgorde aanhouden. Dit heet het gebruik maken van *keyword* argumenten (sleutelwoord).

Je kunt beide opties ook mengen tot een derde smaak:

3. Eerst komen de argumenten op volgorde, daarna *keyword* argumenten.

```
# we geven de variabelen mee op de normale volgorde:
waarde = formule(3, 2, 5, 1)

# we maken gebruik van keyword argumenten (specifieke volgorde):
waarde = formule(x=1, a=3, b=2, c=5)

# we mengen:
waarde = formule(3, 2, x=1, c=5)
```

Oefenopdracht: Controleren van bovenstaande functie

- Controleer dat `formule(3, 2, 5, 1)` en `formule(x=1, a=3, b=2, c=5)` dezelfde waarde opleveren. Welke? En dat een `formule(1, 3, 2, 5)` een andere waarde oplevert. (Maak gebruik van `print(waarde)` uiteraard.)
- Test ook `formule(x=5, 1, 3, 2)` en `formule(3, 2, x=1, c=5)`. Welke optie werkt er wel en welke niet?

5.3 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 `a`, `b`, `c` en `x` daar niet voorkomen. Deze variabelen zijn alleen bekend binnen de functie `formule` en heten lokale variabelen.

Waarden van lokale variabelen kun je hooguit weergeven door ze te printen binnen een functie, maar dat wordt al snel irritant als zo'n functie duizend keer wordt aangeroepen. Je kunt ze verder niet gebruiken *buiten* de functie. Waarden van globale variabelen kun je wel gebruiken binnen een functie. Maar je kunt niet binnen een functie de waarde van een globale variabele veranderen. **Tip:** zorg ervoor dat de namen van variabelen in een functie andere namen hebben dan de globale variabelen. Zo voorkom je verwarring.

5.4 Defaultwaarde

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

Code 5.1: code-inc/w5/voorbeeld5˙1.py

```
fig, ax = plt.subplots()

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

```
    if grid == True:
        ax.grid()

x_arr = np.linspace(-5.,5., num=100)     # bereik x-as
```

Oefenopdracht: Defaultwaarde

- Test bovenstaand script met de volgende regels:

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

  Wat gebeurt er? Wat is de invloed van `True` en `False`?

5.5 Slotopdracht: Eigen plotfunctie

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

```
def mijn_plot(x,y,i,j,label=None,xlabel=None,ylabel=None,
              xmin=min(x),xmax=max(x),ymin=None,ymax=None,
              legenda=False,grid=True):
```

Hier staat `x` voor het array voor de x-as, en `y` voor de lijn die je wilt plotten. `i` en `j` staan hier voor de subplot waarin je wilt plotten (denk bijvoorbeeld aan `axarr[i,j]`). Vervolgens komen de keyword-arguments: `label` staat voor het label van je lijn, `xlabel, ylabel` spreken voor zich. `xmin, xmax, ymin` en `ymax` staan voor het bereik op de x- en y-as.; `legenda` voor `ax.legend()` en `grid` voor `ax.grid()`.
Tip: maak gebruik van het voorbeeld in paragraaf 5.4 en breid deze uit voor alle opties in de `mijn_plot`-regel. Kijk terug naar paragraaf 3.2 als je niet alle opties meer helder hebt.

Als *proof-of-concept* pak je je slotopdracht van hoofdstuk 4. Maak gebruik van jouw functie in deze slotopdracht om de data van elk weerstation te plotten.

6 Meet je leefomgeving

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

De sensorkastjes zijn op zichzelf best 'dom': het enige wat ze doen is de sensoren uitlezen en de data versturen. Aan jou nu de taak om van deze data chocola te gaan maken: hoe was de temperatuur de afgelopen weken, hoeveel fijnstof is er rond jouw huis, en hoeveel lawaai maken jouw buren nou werkelijk?

De eerste vijf hoofdstukken hebben we geleerd hoe Python werkt, hoe je `numpy` en `matplotlib` kan gebruiken, data kan importeren en figuren kunt opslaan. Dat gaat nu allemaal samenkomen in de eindopdracht.

Dit hoofdstuk bevat geen nieuwe stof: in principe kun je alles wat je tot nu toe geleerd hebt combineren om de eindopdracht te maken. De informatie in dit hoofdstuk geeft jou vooral de nodige kennis om te weten wat voor data je nou werkelijk bekijkt en wat voor plaatjes je moet maken, en een enkele truc die specifiek voor dit project fijn is om te weten.

6.1 De sensoren

Het kastje meet een flink aantal dingen. We behandelen de sensoren hier één voor één met de bijbehorende informatie zoals eenheden (nuttig voor bijvoorbeeld de y-as!).

6.1.1 ADC: accuspanning

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

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

6.1.2 BME280: temperatuur, luchtdruk en relatieve luchtvochtigheid

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

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

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

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

6.1.3 MAX4466: geluidssterkte

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

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

6.1.4 TSL2591: lichtintensiteit

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

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

6.1.5 VEML6070: UV-intensiteit

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

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

6.1.6 CJMCU-811: VOC en CO_2-gehalte

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

7) VOC-gehalte in de lucht. VOC's zijn Vluchtige Organische Stoffen (*Volatile Organic Compounds*), gemeten in 'parts per billion' (ppb). Ze zijn verantwoordelijk voor smog, verzuring en hebben een (kleine) invloed op klimaatverandering. Ze worden geproduceerd door sommige planten en bomen, maar ook door mensen bij het gebruik van bijvoorbeeld verf en spuitbussen. Tot 250 ppb is een gezond gehalte; bij een aanhoudend niveau van 250-2000 ppb gedurende een aantal dagen is het opletten, en een gehalte van meer dan 2000 ppb is erg schadelijk voor de gezondheid.[6]

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

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

6.1.7 SDS011: fijnstof

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

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

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

6.1.8 Ublox NEO-6M GPS6MU2: GPS

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

11) GPS-locatie, in NB-OL coördinaten. Coördinaten kun je echter niet zo makkelijk in een grafiekje weergeven, dus laten we dat hier achterwege. Mocht je de uitdaging aan willen gaan om GPS op een kaartje weer te geven is dat mogelijk, maar het valt wel buiten het doel van deze reader.

6.2 Handige informatie

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

np.argsort()

De data van het sensorkastje is ongesorteerd. De meetwaarden worden allemaal in één keer verzonden en afhankelijk van het bereik, de kracht van de antenne en andere factoren komen de data allemaal net niet tegelijkertijd aan, en telkens in een andere volgorde. Om te sorteren kun je handig gebruik maken van `np.argsort()`; zie de slotopdracht van hoofdstuk 4.

Er zijn elf channels: 0 tot en met 10. Elke elf waarden kun je sorteren met behulp van deze nummers. *Hint: je kunt ook een deel van de data sorteren door bijvoorbeeld het volgende in te vullen:* `np.argsort(data[1:4])`.

6.3 De eindopdracht

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

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

Voltooi je deze opdracht, dan scoor je daarmee een 8. Verwerk je de volgende punten ook, dan scoor je daarmee 1 extra punt. Steek je extra veel moeite in efficiëntie, commentaar en/of opmaak, kun je daar 1 punt extra voor scoren.

De exacte puntenverdeling en de onderdelen die aanwezig moeten zijn staan vermeld in de rubric op ItsLearning: kijk deze nog eens door voor je begint en af en toe terwijl je bezig bent.

Extra opgaven:

- De UV-index wordt vermeld als *maat*, zonder eenheid. Je kunt er in de praktijk dus eigenlijk weinig mee. De fabrikant heeft echter wel de matchende UV-*levels* bij vermeld. De plot van de UV-waarden is veel nuttiger als je deze levels of zones dus ook aangeeft. Maak daarbij gebruik van de volgende verdeling:

  1) "LOW": [0, 560],
  2) "MODERATE": [561, 1120],
  3) "HIGH": [1121, 1494],
  4) "VERY HIGH": [1495, 2054],
  5) "EXTREME": [2055, 9999]

  Maak zelf een bewuste keuze: welke regio's geef je aan? Ga je door tot 9999? En geef je ook de namen weer, of alleen met kleuren de zones? En als lijnen, of als gekleurde vlakken? Maak hierbij gebruik van internet om het mooi te maken!

- De CJMCU-811-sensor is eigenlijk een beetje nep. Hij meet inderdaad correct het VOC-gehalte in de lucht, maar eigenlijk helemaal geen CO_2. De sensor neemt aan dat alle VOC's van mensen afkomstig zijn, en berekent dus hoeveel mensen er aanwezig zouden zijn volgens de VOC-waarde. Vervolgens neemt hij de gemiddelde CO_2-productie van een mens, om te berekenen hoeveel CO_2 er misschien dan in de lucht zit. Maak het inzichtelijk dat de sensor je hier voor de nep houdt door beide lijnen in dezelfde plot (in een nieuw figuur) te zetten. Helemaal mooi is het als je een tweede y-as (aan de rechterkant) weet toe te voegen waardoor je de lijnen kunt schalen.

7 BONUS

7.1 Voormalige slotopdracht H4: De Europese bananeninspectie

We passen de stof van de hoofdstukken 1 tot en met 4 toe op de strenge Europese regelgeving voor bananen.

1: Importeren van data

Bekijk het volgende bestand: data-h7.

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

2: Testen op dikte

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

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

3: Lengte en dikte

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

4: Klasseindeling

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

Deel de bananen aan de hand van de criteria in bovenstaande tabel in in de verschillende klassen. Rapporteer voor de gegenereerde dataset hoeveel procent van de bananen in de Extra klasse, hoeveel in klasse I en hoeveel in klasse II worden ingedeeld, rapporteer ook hoeveel procent van de bananen afgekeurd is.[7] Controleer of de bepaalde percentages optellen tot 100%.

[7] 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.

Tabel 1: 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	$\leq 1\,\text{cm}^2$	$\leq 2\,\text{cm}^2$	$\leq 4\,\text{cm}^2$

7.2 Commentaar schrijven

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

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

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

Richtlijnen voor het schrijven van goed commentaar:

- Gebruik *natuurlijke taal*, d.w.z. gebruik zinnen en schrijf voluit. Gebruik geen Python code in comments,[8] en definieer (óók voor de hand liggende) symbolen als F, g en T.
- Zet commentaar bij het toekennen van numerieke waardes; zeg bij welke grootheid (evt. met symbool tussen haakjes) ze horen en welke eenheid ze hebben.

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

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

[8] Tenzij het een stukje code is wat tijdelijk niet gebruikt wordt.

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

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

- Maak de regels niet te lang. In Spyder (en de meeste andere editors) kun je een verticale lijn laten weergeven na een specifiek aantal karakters.[9] Zorg dat regels code en commentaar niet (veel) langer worden dan tot die lijn. Bij Python is de conventie om deze na 79 karakters te zetten. Let op dat je door syntax niet zomaar een nieuwe regel kunt beginnen; dit heeft namelijk een betekenis binnen Python. Door gebruik van de juiste *indentatie* kun je toch aan de 79-karakter-limiet voldoen.
- Gebruik `#%%` om *cells* te maken die los te runnen zijn. Let op: bij goede modulaire code (waarbij de acties in functies staan) is dit juist niet altijd handig. Tijdens het ontwikkelen van de code kunnen cells wel uitermate handig zijn, zodat je bijvoorbeeld een tijdrovend deel van je code kunt overslaan (bijvoorbeeld het genereren of analyseren van data) terwijl je door werkt aan een ander stuk (bijvoorbeeld het plotten van data).
- Schrijf bovenaan elke cell wat er in die cell gebeurd (welke 'titel' je de cell zou geven).
- Gebruik scheidingsmarkeringen tussen lange stukken code. Een lege regel tussen het importeren van packages en de start van de code, en een lege regel tussen het einde van een loop en het vervolg van de (niet geïndenteerde) code, of een stuk code gescheiden door een regel `#----------` geven het script een overzichtelijkere uitstraling en maken het daarmee beter leesbaar.
- Verkies leesbaarheid over snelheid.

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

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

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

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).[10] Python is een langzame taal omdat de vertaling tussen het script en de instructies die naar de CPU gaan (in éénen en nullen) gebeurt tijdens het runnen van het script; dit maakt Python een *interpreting language*. Bij een *compiling language* zoals C of C++ moet je een script eerst *compilen* voordat het naar de CPU gestuurd kan worden. Dit compilen kost tijd, maar het daadwerkelijke runnen van de code gaat dan veel vlotter.

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

Er zijn manieren om je Python code om te zetten in C-gecompileerde code, en zelfs om je Python code op meerdere processors tegelijk (parallel) te laten uitvoeren.[11] Meer over deze methodes zul je vanzelf tegen komen als je meer complexe simulaties of berekeningen gaat doen tijdens je studie of onderzoek. Deze manier van optimalisatie is echter geen onderdeel van deze beginnerscursus.

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

7.3.1 Efficiëntie meten

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

Code 7.1: code-inc/w6/time˙vb1.py

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

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

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

[10] 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.

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

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

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

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

Oefenopdracht

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

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

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

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

7.3.2 Wanneer optimaliseren?

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

Wanneer is optimaliseren dan een goed idee? Vuistregel is:

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

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

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

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

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

7.3.3 Standaard snelle oplossingen

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

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

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

Voorbeelden van snellere en langzamere code:

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

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

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

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

Oefenopdracht: efficiëntie

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