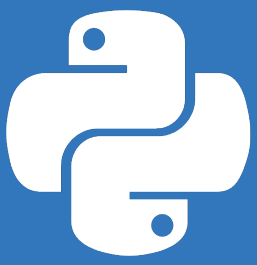


python-cursus.nl

# BASISCURSUS PYTHON

# Inhoudsopgave

# Introductie

Welkom bij de Basiscursus Python. In deze cursus leer je de basisbeginselen van het programmeren met Python. Het doel van deze opleiding is dat je de basisconcepten van het programmeren begrijpt én kunt toepassen in de praktijk.

Na het volgen van de cursus ben je dan ook in staat om zelfstandig (eenvoudige) scripts[1] en (web)applicaties te ontwikkelen met Python. Om dit te bereiken leer je niet alleen de theorie, maar ga je ook veel zelf aan de slag. Door de cursus heen zie je veel voorbeelden, zoals:

```
def greet(name):
    print(f"Hello, {name}!")
```

Het advies is om dergelijke voorbeelden zelf ook uit te proberen en dit te doen door het over te typen (dus niet kopiëren en plakken). Op die manier begrijp en beheers je de stof het snelst.

## Leerdoel

Het hoofddoel van de Basiscursus Python is:

> **Het fundament van Python leren, zodat je een basis hebt waarop je zelf verder kunt bouwen en leren.**

## Hoe lees je dit boek

Dit boek kun je op je eigen tempo doornemen. Er zijn twee manieren om dat te doen.

Je begint bij het begin en leest het hele boek in volgorde. Doe dit als je nog maar net begint met programmeren en ervan houdt om eerst alles te begrijpen voor je iets gaat ontwikkelen.

Houd je er meer van om te leren door te proberen, ga dan naar het laatste hoofdstuk. Dit is een project dat je zelf kunt uitvoeren. Begin eraan en zie hoe ver je komt. Loop je vast of denk je dat iets eenvoudiger moet kunnen, zoek dan het juiste hoofdstuk erbij. Bijvoorbeeld: als je niet weet hoe je een bestand moet openen, lees dan Werken met bestanden.

Verder leer je het best door te doen, dus voer alle voorbeelden ook zeker zelf uit. En belangrijk: je kunt ze het beste overtypen, niet kopiëren! Maak ook alle opdrachtjes om te oefenen. De antwoorden zijn erbij gegeven.

Je kunt het online lezen of downloaden als PDF.

## Conventies

### Code

Code is altijd duidelijk herkenbaar doordat het in een apart blok staat:

```
def greet(name):
    print(f"Hello, {name}!")
```

Soms zie je ook in de lopende tekst code staan, bijvoorbeeld om te verwijzen naar een variabele: `mijn_variabele`. Deze code heeft een ander lettertype, kleur en heeft een licht grijze achtergrond.

### Begrippen

Verder zie je soms een cijfertje achter een woord staan. Daar kun je op klikken voor een korte toelichting op het woord.

## Tot slot

Heb je vragen, opmerkingen, verbeteringen of tips naar aanleiding van dit boek, laat het dan weten!

[1] Een script in Python is een bestand met een reeks commando's die door de Python-interpreter kunnen worden uitgevoerd. Het heeft meestal de bestandsextensie .py.

# Eerste stappen

In dit hoofdstuk zet je de eerste stappen in het programmeren. Na een korte inleiding op wat programmeren nu eigenlijk is, ga je aan de slag met het installeren van Python en een editor. Daarna leer je de basis van Python. Veel onderwerpen komen hier kort aan bod en worden in latere hoofdstukken verder uitgediept.

## Leerdoelen

Aan het eind van dit hoofdstuk:

- Begrijp je wat een programmeur doet
- Begrijp je wat Python is
- Heb je Python geïnstalleerd en verkend
- Begrijp je de structuur van Python
- Kun je werken met getallen
- Begrijp je de gegevenstypen `None` en `bool`
- Kun je werken met tekst
- Begrijp je wat collecties zijn

## Programmeren in Python

In deze paragraaf leer je wat programmeren is, wat Python is en ga je aan de slag met het installeren van Python en software om eenvoudig met Python te kunnen werken. Na deze voorbereidende stappen ga je direct aan de slag met een eerste verkenning van de programmeertaal.

### Wat is programmeren?

Programmeren is het proces van het ontwikkelen van software door instructies te schrijven voor een computer om bepaalde taken uit te voeren. Deze instructies worden geschreven in een specifieke programmeertaal, zoals Python, Java of C++. Het doel van programmeren is om de computer te vertellen wat deze moet doen om problemen op te lossen, gegevens te verwerken, interactieve applicaties te maken en andere nuttige functies uit te voeren die het dagelijks leven en werk vereenvoudigen.

Programmeren is dus het schrijven van instructies voor een computer. Maar, als programmeur ben je niet heel de dag code aan het schrijven. Code schrijven is uiteraard een belangrijk onderdeel en in deze opleiding leer je ook met name dát. Als programmeur ben je echter ook bezig met andere zaken, zoals:

- Het probleem helder krijgen
- Overleggen met collega's, klanten, of andere stakeholders[1] (belanghebbenden)
- Nadenken over de juiste benadering voor een oplossing
- Prioriteren en inschatten hoeveel tijd een bepaalde implementatie gaat kosten
- Code lezen
- Documentatie schrijven
- Andermans code beoordelen

Bepalen wélke code je wanneer schrijft en waarom, is een net zo belangrijk onderdeel van je werk als het daadwerkelijk schrijven van de code. Kijk er dus niet vreemd van op als je een paar uur verder bent en maar enkele regels code hebt geschreven. Vaak is daar heel wat aan vooraf gegaan!

## Wat is Python?

Python is een populaire, krachtige en veelzijdige programmeertaal die eenvoudig te leren en te gebruiken is. Laat je echter niet misleiden door de eenvoud: de taal is zeer capabel!

Het is ontworpen met een focus op leesbaarheid en duidelijkheid van code, wat het schrijven en onderhouden van programma's vergemakkelijkt. Python wordt gebruikt voor een breed scala aan toepassingen, zoals webontwikkeling, data-analyse, kunstmatige intelligentie en automatisering. Het is een open-source[2] taal, wat betekent dat het vrij beschikbaar is en door iedereen kan worden aangepast.

Tot slot is het belangrijk om te weten dat Python een zogenaamde *geïnterpreteerde* taal is. Dat betekent dat de code direct wordt uitgevoerd en niet eerst wordt *gecompileerd* naar een uitvoerbaar bestand (bijvoorbeeld `.exe`). Het voordeel hiervan is dat je snel kunt experimenteren: schrijf een stukje code en voer het direct uit, je hebt direct resultaat. Zeker in de context van een opleiding - waar je nog veel moet leren - is dit handig. Het nadeel is dat het uiteindelijke programma vaak trager is dan gecompileerde talen. Tenzij je met processen te maken krijgt waar snelheid absoluut van belang is, is dit in de praktijk echter geen probleem. Python is nog steeds snel genoeg!

Wist je dat Python oorspronkelijk ontwikkeld is door een Nederlander? Het was Guido van Rossum die in 1989 begon met de ontwikkeling van Python. De naam verwijst naar Monty Python.

## Python installeren

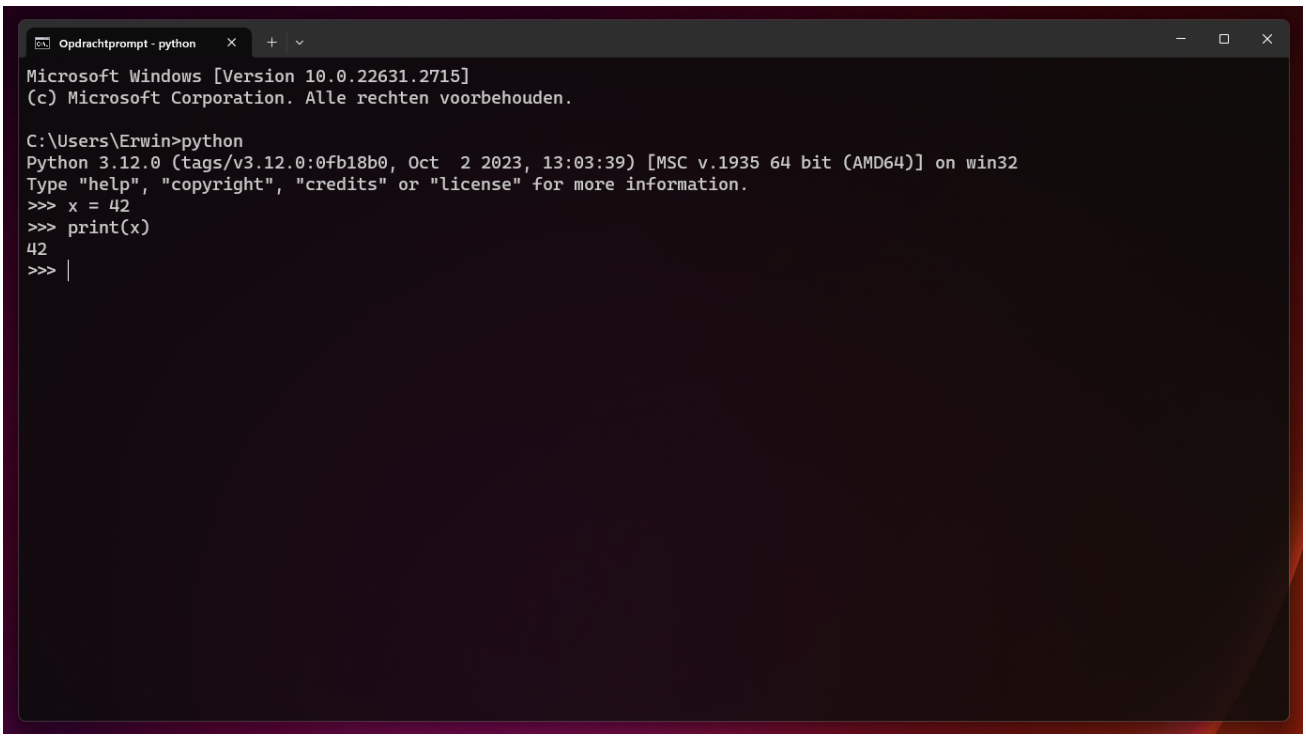
Om te kunnen werken met Python, dient het wel geïnstalleerd te zijn op je computer of laptop. Ga hiervoor naar python.org. Ga daar naar de downloadpagina en kies de laatste versie voor jouw OS. Voor Windows kun je het best de Windows installer (64 bit) kiezen.

Kies bij het installeren in Windows voor [ **Add python.exe to PATH** ].


*De Python-installer in Windows*

## Thonny installeren

Python code kun je in de shell[3] uitvoeren. Je typt in je shell (De opdrachtprompt in Windows, in Linux en MacOS open je de terminal) `python` in, en je kunt aan de slag:



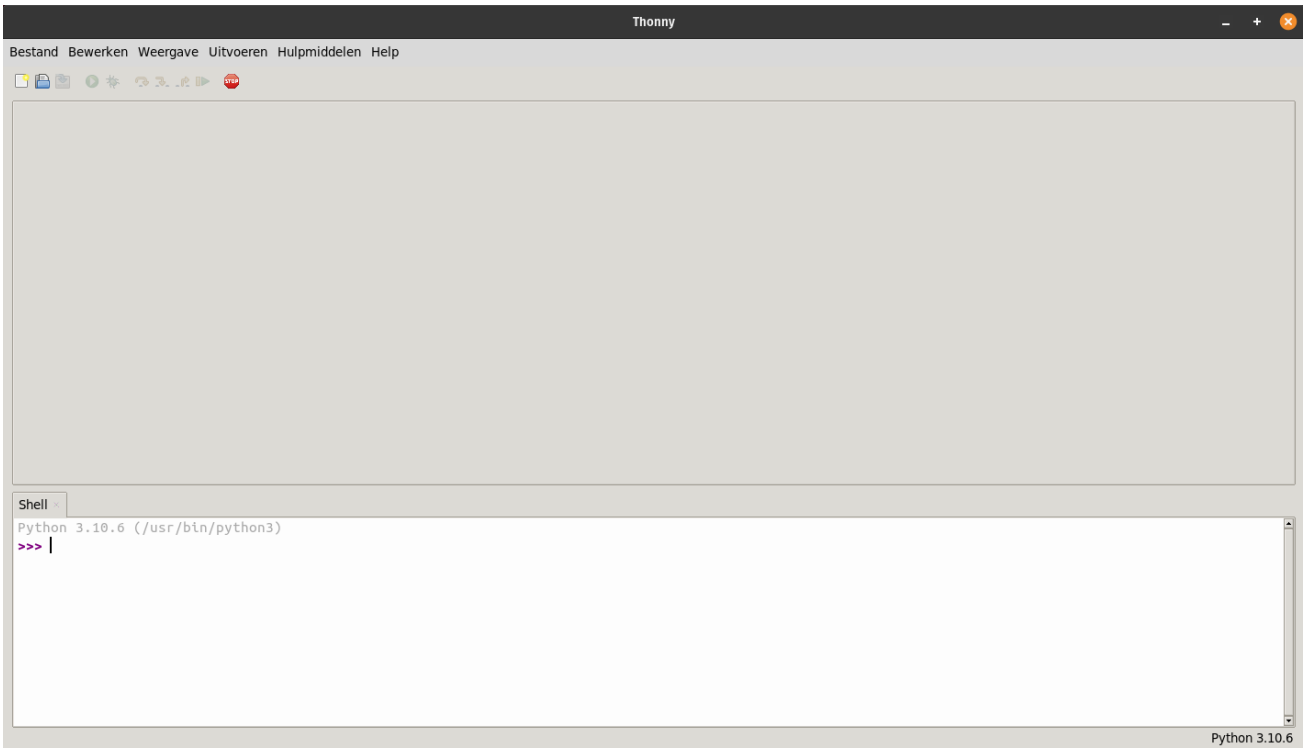

*Python in de opdrachtprompt in Windows*

Dit is handig om snel te experimenteren of te controleren hoe iets ook alweer werkt. Voor iets langere stukjes code wordt dit echter al snel onhandig.

In de praktijk plaats je code dan ook in een speciaal tekstbestand, eindigend op `.py`. Dit bestand voer je vervolgens uit met Python, zodat alle instructies in dat bestand worden uitgevoerd. Het werken met dergelijke bestanden doe je vaak met een zogenaamde Integrated Development Environment (IDE). Dit is software om op een eenvoudige manier met code te werken en bevat vaak vele hulpmiddelen die jouw leven als programmeur eenvoudiger maken.

Veelgebruikte IDEs zoals Pycharm en VSCode kunnen voor een beginnend programmeur nogal overweldigend zijn. In deze opleiding wordt daarom een IDE speciaal voor de beginnende Python programmeur gebruikt: Thonny.

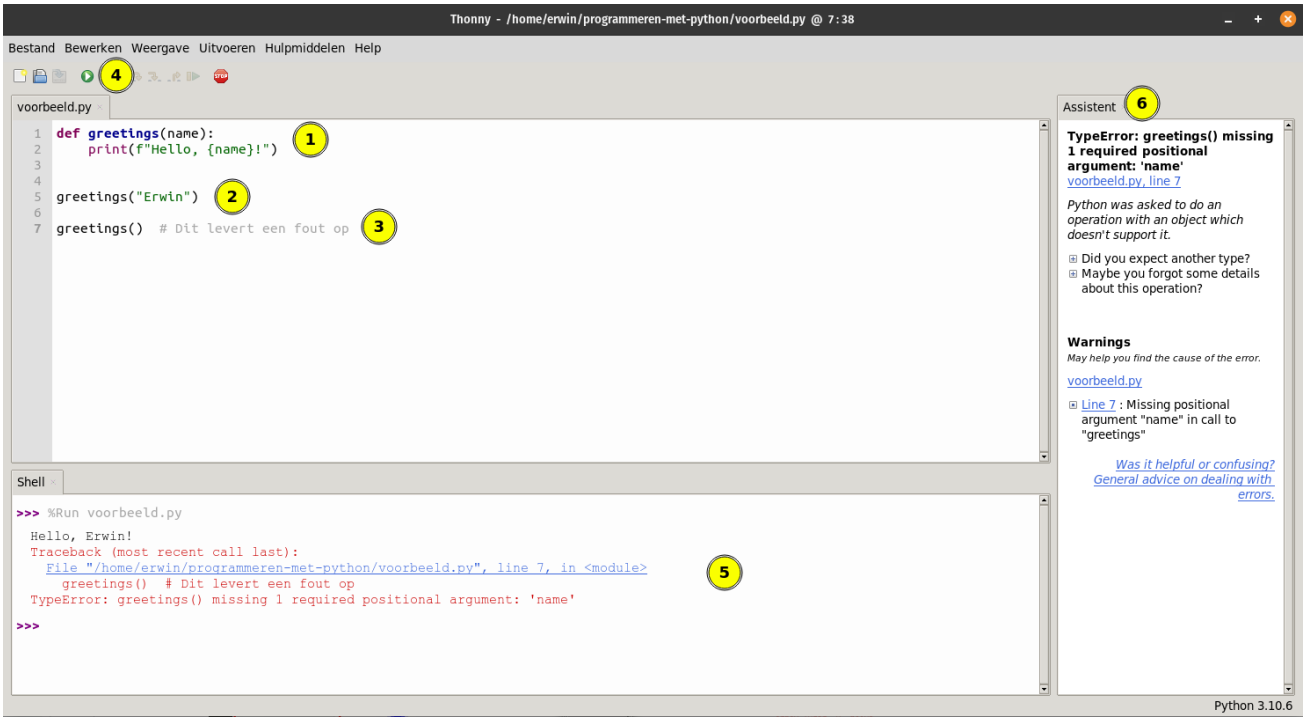
Op de homepage van de website van Thonny kun je Thonny direct downloaden voor Windows, MacOS of Linux. Na het installeren en opstarten zie je het volgende:




*Het beginscherm van Thonny, de IDE voor beginnende Python-programmeurs*

Het onderste veld is een Python-shell, waarin je direct instructies kunt typen en uitvoeren. Het voordeel van Thonny is dat je ook in een bestand kunt werken, dit maakt experimenteren eenvoudiger. Maak een nieuw bestand aan via **Bestand › Nieuw**. Geef het de naam `voorbeeld.py` (`.py` zal automatisch toegevoegd worden) en sla het op een logische plek op.

In dit nieuwe bestand kun je vervolgens instructies plaatsen. Door op [ **F5** ] of op de groene [ **Play** ] knop te drukken, wordt het bestand uitgevoerd in de shell. Zo kun je eenvoudig aanpassingen doen en je bestand opnieuw uitvoeren. Als je een fout maakt, opent Thonny een scherm met tips.




*Thonny in gebruik*

1. Hier is de functie `greetings` gedefinieerd (in Werken met functies leer je wat een functie is).

2. Hier wordt de functie correct aangeroepen.
3. Hier wordt de functie onjuist aangeroepen, bij 6 zie je een toelichting op de fout die dit opwerpt als je het bestand uitvoert.
4. Met de [ **Play** ] knop (of [ **F5** ]) voer je het bestand uit.
5. De shell. Hier kun je zelf Python-code intypen, maar de resultaten (en foutmeldingen) van het uitvoeren van het bestand verschijnen ook hier.
6. Dit scherm verschijnt automatisch wanneer er een fout optreedt. Het bevat handige aanwijzingen over hoe de fout op te lossen.

## Python verkennen

Nu je Python hebt geïnstalleerd en een korte kennismaking hebt gehad met de shell en Thonny, is het tijd om Python verder te verkennen.

### Structuur van de code: inspringen

Een belangrijk kenmerk van Python is dat het inspringen van code onderdeel uitmaakt van de regels (syntax[4]) van de taal, dit in tegenstelling tot vele andere programmeertalen.Een eenvoudig voorbeeld om dit te verduidelijken:

```
1 for i in range(5):
2     x = i * 10
3     print(x)
4
5 print("Nieuwe opdracht")
```

In dit voorbeeld zie je twee instructies. De eerste begint op regel één en eindigt op regel drie. De tweede bevindt zich op regel vijf. De eerste instructie lees je als volgt:

- Met `range(5)` genereer je een reeks getallen van 0 tot en met 4
- Met `for` itereer je over deze getallen
- Elke iteratie (herhaling of ronde) sla je het getal op in de variabele `i`. De eerste ronde is `i` dus gelijk aan 0, de tweede ronde is `i` gelijk aan 1, etc.
- Op regel twee vermenigvuldig je `i` met 10 en sla je het resultaat op in de variabele `x`
- Op regel drie druk je `x` af naar je scherm

De eerste regel van de instructie eindigt met een dubbele punt, dit betekent dat daarna een zogenaamd *blok* volgt dat tot deze instructie behoort. Regel twee en drie springen in, waarmee je aangeeft dat ze tot het blok behoren dat begint op regel één. Regel vier is leeg en doet dus niets. Regel vijf springt níet meer in, waarmee dus een nieuwe instructie begint.

Volgens conventie springt elk niveau in met vier spaties. In de meeste IDEs kun je simpelweg op *tab* drukken en zal dit automatisch omgezet worden naar vier spaties. Ook in Thonny is dit het geval.

### Opdracht 1: Inspringen

Kun je voorspellen wat er gebeurt als je de laatste regel wél laat inspringen? Wat gebeurt er als je het twee spaties laat inspringen in plaats van vier?

▼ *Klik om het antwoord te tonen*

Als je de regel wel laat inspringen, zal elke ronde van de `loop` de zin "Nieuwe opdracht" geprint

worden. De regel behoort dan namelijk tot het blok van de `for-loop`. Als je de regel met twee spaties laat inspringen in plaats van vier, zul je een foutmelding krijgen, omdat de mate van inspringen niet overeenkomt met de andere niveaus van inspringen (0 of 4 spaties).

**Structuur van de code: commentaren**

Je zag al dat de voorlaatste regel - een lege regel - niets doet. Blokken worden gedefinieerd op basis van inspringen, lege regels zijn dus niet per se nodig. Voor de leesbaarheid is het wel prettig om af en toe wat witruimte in te voegen.

Niet alleen lege regels worden door Python genegeerd. Je zag het al even voorbijkomen in de afbeeldingen van Thonny: opmerkingen. Zodra je een # plaatst, negeert Python alles erachter, tot aan de volgende regel. Je kunt dit gebruiken om korte toelichtingen voor jezelf of voor je collega's te plaatsen.

```
1  # Deze functie ontvangt één argument: de naam van de persoon (str)
2  def greet(name):
3      """
4      Dit is een docstring, daarover volgt in een later hoofdstuk meer.
5      """
6
7      # print("Hello")
8      print(f"Hello, {name}")  # Print de begroeting met naam
9
10     """
11     Soms heb je iets meer ruimte nodig voor een wat langer
12     commentaar. Je kunt dit doen door te beginnen en
13     eindigen met drie (dubbele) aanhalingstekens.
14
15     Je kunt hierin plaatsen wat je wilt, ook code:
16
17     print(f"Hello, {name}")
18
19     Dit wordt allemaal genegeerd door Python.
20     """
```

In dit voorbeeld zie je de verschillende mogelijkheden om commentaren te plaatsen. Een commentaar hoeft dus niet per se aan het begin van de regel te beginnen. De conventie is dat wanneer je een commentaar áchter een stuk code plaatst - zoals op regel acht - je er twee spaties tussen laat.

Op regel zeven zie je dat er een # voor een instructie is geplaatst. Deze instructie wordt daardoor *niet* door Python uitgevoerd.

# Gegevenstypen

Nu je helder hebt hoe je Python-code structureert, is het tijd om daadwerkelijk kennis te gaan maken met de taal. Programmeren is werken met *gegevens* en in Python heeft elk gegeven een bepaald *type*. Overigens geldt dit voor veel programmeertalen.

**Een gegevenstype bepaalt wat je met het gegeven kunt doen.**

In het dagelijkse leven kom je dit ook tegen. Heb je twee cijfers, dan kun je ze vermenigvuldigen. Heb je twee woorden, dan kan dat niet. Maar de woorden kun je dan wel

in de verleden tijd zetten.

In tegenstelling tot veel andere talen voert Python geen typecontrole uit voordat de code wordt uitgevoerd. Neem bijvoorbeeld onderstaande stukje code:

```
1 x = "test"
2 print(x/2)
```

Voer je dit uit in Thonny, dan zul je een foutmelding krijgen. De variabele `x` is hier van het gegevenstype `str` (*string*, ofwel tekst). Op regel twee probeer je `x` te delen door twee: `x/2`. In Python kun je tekst niet delen, en dus krijg je een foutmelding *bij het uitvoeren van de code*. Dit laatste is belangrijk.

Je kunt de code rustig laten staan in je programma, en Python zal er niet over klagen. Pas wanneer dit stukje code wordt uitgevoerd komt de foutmelding. In veel andere programmeertalen, met name de *gecompileerde* talen, worden dergelijke fouten al opgeworpen vóórdat het programma in gebruik wordt genomen. Zo worden vele bugs[5] al eerder opgemerkt.

De laatste jaren is er wel meer aandacht voor controleren van gegevenstypen en worden hulpmiddelen hiervoor in de taal ingebouwd. In deze opleiding wordt hier verder geen aandacht aan besteed. Wil je hier (later) meer over weten, kijk dan eens naar mypy (www.mypy-lang.org/), een veelgebruikt hulpmiddel bij het controleren van gegevenstypen.

In deze paragraaf leer je over een zestal gegevenstypen:

- int
- float
- bool
- None
- str
- collecties

## Werken met getallen

Tijdens het programmeren zul je veel te maken krijgen met getallen. Python heeft hiervoor verschillende ingebouwde typen, waarvan de belangrijkste `int` en `float` zijn.

### int

Het type `int` (kort voor integer) gebruik je om met gehele getallen te werken. De bekende operatoren[6] zijn van toepassing:

```
1 print(10+5)
2 print(10*5)
3 print(10-5)
4 print(-5-5)
5 print(10**2)
6 print(10/2)
```

Zoals je ziet in de code hierboven zijn integers niet alleen voorbehouden aan positieve getallen.

### Opdracht 2: Operatoren

Kun je achterhalen wat `**` doet op regel vijf? En wat valt je op aan de uitkomst van regel zes?

▼ *Klik om het antwoord te tonen*

Twee asteriksen is de operator om machten te verheffen. `10**2` betekent dus tien tot de macht twee, ofwel 10 in het kwadraat. `10**3` betekent tien tot de macht drie.

Op regel zes valt op dat je twee integers deelt, maar de uitkomst is een `float`. Delen met `/` levert altijd een `float` op.

Een `int` kun je direct noteren, zoals hierboven. Maar soms heb je ook een ander gegevenstype in je code, waarvan je een `int` wilt maken.

```
1 int("5")  # Van str naar int
2 int(3.5)  # Van float naar int
```

### Opdracht 3: int()

Voer bovenstaande eens in Thonny in, en bekijk wat het resultaat is. Wat gebeurt er als je `int("Hallo")` invoert? En wat bij `int(3.6)`?

▼ *Klik om het antwoord te tonen*

`int("5")` levert `5` op. `int(3.5)` levert `3` op. `int("Hallo")` levert een fout op, omdat Python niet weet hoe je van een woord een integer moet maken. `int(3.6)` levert, wellicht onverwacht, `3` op. Als je `int()` gebruikt om van een `float` een integer te maken, neemt hij het gehele getal van voor de decimaal. Er wordt dus niet afgerond.

#### float

Het type `float` (kort voor *floating point*) gebruik je om niet-gehele getallen mee aan te duiden. Een `float` maak je door een getal met een decimaal te noteren. Let hierbij op dat je een punt gebruikt in plaats van een komma, zoals wij gewend zijn. Dus noteer `3.14` en niet `3,14`.

Net als bij `int` kun je de verschillende operatoren gebruiken:

```
1 print(10.5+5)
2 print(10.5*5)
3 print(10.5-5)
4 print(-5.5-5)
5 print(10.5**2)
6 print(10.5/2)
```

En kun je andere gegevenstypen omzetten naar `float`:

```
1 float("5")  # Van str naar float
2 float(3)  # Van int naar float
```

Vaak volstaat het werken met `float` als je werkt met niet-gehele getallen. Maar er gaat een wereld schuil achter wat eigenlijk een *floating point* is. Belangrijk om te onthouden is dat het **niet** een decimaal getal is en dat het niet **precies** is. Kijk maar eens naar de volgende optelsom:

```
1 print(0.1 + 0.1 + 0.1)
2 # 0.30000000000000004
```

Maakt de precisie wel uit, dan kun je beter met `Decimal` werken. Zie de documentatie van Python voor meer.

## Werken met lege waarden (None) en Waar/Niet-waar (bool)

Het werken met getallen zoals hierboven zal redelijk intuïtief aanvoelen, in het dagelijkse leven werk je er geregeld mee. Iets minder intuïtief zijn misschien de gegevenstypen `None` en `bool`.

### Lege waarden

`None` betekent letterlijk *Niets*, en is een zogenaamde `null` waarde, een *lege* waarde. Zie het als een lege snoeppot:

```
1 snoeppot = None  # Een lege snoeppot
2 print(type(snoeppot))
3
4 snoeppot = 100  # Een goedgevulde snoeppot
5 print(type(snoeppot))
6
7 print(snoeppot is None)
```

Je maakt een variabele `snoeppot` en wijst de waarde `None` toe. De waarde van `snoeppot` heeft dus het gegevenstype `None`. Daarna 'vul' je de snoeppot en is het gegevenstype `int` geworden. Met de operator `is` kun je controleren of een object de waarde `None` heeft.

### Opdracht 4: Waar of niet waar?

Voer bovenstaand voorbeeld in Thonny in.

- Wat is het resultaat van de laatste regel?
- Wat is het resultaat als de laatste regel vervangt door `print(snoeppot is not None)`?

▼ *Klik om het antwoord te tonen*

`print(snoeppot is None)` levert `False` op. `snoeppot` is inmiddels niet meer `None`, maar een `int`. `print(snoeppot is not None)` levert `True` op. Met `not` draai je de toets om.

In de code zie je het gebruik van `type()`. Hiermee kun je altijd het type van een object achterhalen. Regel twee geeft als resultaat `<class 'NoneType'>` terug, en regel vijf `<class 'int'>`. Dit lijkt een beetje cryptisch, maar je ziet al snel dat het de gegevenstypen `None` en `int` betreft. Wat een `class` is, leer je in een later hoofdstuk.

`None` is altijd één object in Python. Voer het volgende maar eens uit:

```
x = None
y = None

print(id(None))
print(id(x))
print(id(y))

print(x is None)
```

Met `id()` haal je het unieke ID op van een object (zie het als het unieke adres in het geheugen van de computer).

Met `is` vergelijk je het `id` van twee objecten, en daarom levert `x is None` ook `True` op.

**Waar of niet waar?**

Als je bovenstaande opdracht hebt uitgevoerd kreeg je als het goed is eerst het resultaat `False` en de tweede keer het resultaat `True`. Dit zijn waarden van het type `bool` (kort voor *boolean*). Het type `bool` zul je veel tegenkomen tijdens het programmeren, omdat je het zult gebruiken om het verloop van je programma te bepalen: als x *waar* is, voer dan y uit, als x *niet waar* is, voer dan z uit.

Met de ingebouwde functie `bool()` kun je andere gegevenstypen omzetten naar een `bool`.

```
print(bool(0))  # False
print(bool(1))  # True
print(bool(38))  # True
print(bool(0.0))  # False
print(bool(0.1))  # True

print(bool(""))  # False
print(bool("Test"))  # True
```

Bij getallen zijn alleen 0 (zowel bij `int` als `float`) `False`. Bij tekst zijn enkel lege strings `False`.

### Opdracht 5: "False"

Wat is het resultaat van `bool("False")`?

▼ *Klik om het antwoord te tonen*

`bool("False")` levert `True` op. Alleen een lege `str` zal `False` opleveren.

De `bool` is een subtype van `int`, met maar twee waardes: `True` (1) en `False` (0). Net als bij `None` is van elke waarde ook maar één object.

### Relationele operatoren

Booleans zul je veel gebruiken om het programmaverloop te bepalen. Maar hoe werkt dat? Een belangrijk gegeven hierin is dat *relationele operatoren* een `bool` produceren. Relationele operatoren gebruik je om verschillende objecten met elkaar te vergelijken.

De relationele operatoren zijn:

| | |
|---|---|
| `==` | Zijn twee objecten gelijkwaardig aan elkaar? |
| `!=` | Zijn twee objecten *niet* gelijkwaardig? |
| `<` | Is het ene object kleiner dan het andere? |
| `>` | Is het ene object groter dan het andere? |
| `<=` | Is het ene object kleiner dan of gelijk aan het andere? |
| `>=` | Is het ene object groter dan of gelijk aan het andere? |

In het volgende hoofdstuk leer je uitgebreider over hoe je relationele operatoren gebruikt om het programmaverloop te bepalen, maar hierbij alvast een kort voorbeeldje:

```
1 x = 20  # Wijs de waarde 20 toe aan de variabele x
2
3 if x > 20:
4     print("X is groter dan 20!")
5 else:
6     print("X is 20 of kleiner!")
```

#### Opdracht 6: Als x groter is dan 20

Waarschijnlijk kun je de uitkomst wel voorspellen. Probeer het eens in Thonny om te kijken of je het goed had. Wat gebeurt er als je `>` op regel drie vervangt door `>=`?

▼ *Klik om het antwoord te tonen*

In de oorspronkelijke code is het resultaat "x is 20 of kleiner!". Dit omdat je met `>` toetst of `x` groter is dan 20, wat niet het geval is. Vervang je `>` door `>=` - groter dan of gelijk aan - dan zal het resultaat "x is groter dan 20!" zijn.

Misschien weet je van wiskunde nog wel dat `=` werd gebruikt als symbool voor gelijkheid. Bijvoorbeeld:

```
1 x = y + 5
```

Dit wil zeggen: `x` is gelijk aan `y + 5`. In Python (en vele andere programmeertalen) gebruik je een *dubbel* is-teken (==) om op gelijkheid te toetsen. Het enkele is-teken (=), zoals hierboven, is geen toets op gelijkheid, maar een *toewijzing*. In dit geval wijs je de waarde `y + 5` toe aan de variabele `x`. In het volgende hoofdstuk leer je meer over variabelen.

## Werken met tekst (strings)

In de voorgaande lesstof ben je al een aantal keren het gegevenstype `str` tegengekomen (kort voor *string*). Het (Engelse) woord string komt van het feit dat je meerdere tekens (karakters) aan elkaar 'rijgt'. Het type `str` is het gegevenstype voor tekst, specifiek *Unicode-codepunten*. Unicode is een standaard om karakters in digitale vorm aan te duiden. Zo is is de letter "A" altijd `U+0041` (op Windows, Linux, MacOS). Gelukkig hoef je zelden direct met Unicode te werken, Python handelt dat voor je af.

### Strings maken

Er zijn verschillende manieren om een string te maken. Vaak zul je dit simpelweg doen door aanhalingstekens te zetten rondom de tekst. Dit mogen enkele of dubbele aanhalingstekens zijn, als je maar consequent bent!

```
1 a = "Dit is een string."
2 b = 'Dit is ook een string.'
3 c = "Dit zal niet werken'
4 d = "Maar dit kan wel: 'Hallo, wereld!'."
5 e = 'Je kunt ook cijfers (1) of tekens (&) gebruiken.'
```

Je ziet dat als je een string begint met dubbele aanhalingstekens, je er ook mee moet eindigen. In de string kun je dan wel enkele aanhalingstekens gebruiken, deze zijn dan simpelweg onderdeel van de string.

Een andere manier om een string te maken is met de ingebouwde functie `str()`.

```
1 f = str(1)  # "1"
2 g = str(3.1415)  # "3.1415"
```

#### Opdracht 7: str()

- Wat is het resultaat van de volgende instructie? `str(hallo)`
- Wat is het resultaat van de volgende code? `hallo = 1 print(hallo)`

▼ *Klik om het antwoord te tonen*

`str(hallo)` levert een foutmelding op, omdat de variabele `hallo` niet gedefinieerd is. Definieer je `hallo`, dan print het de waarde van de variabele, in dit geval dus `1`.

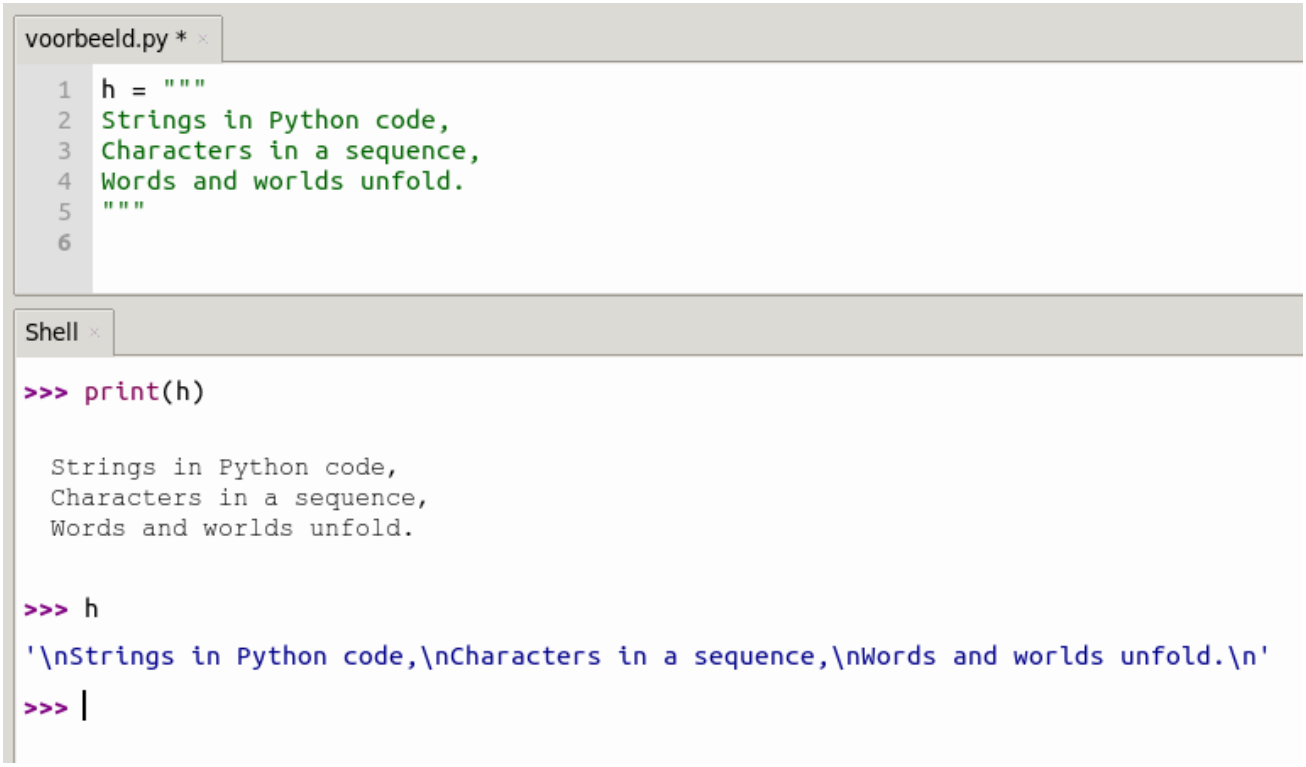
### Meerdere regels

Tot nu toe waren onze teksten vrij kort, maar dit zal in de praktijk niet altijd zo zijn. Om een langere tekst leesbaar te maken, kun je *newlines* (nieuwe regels) gebruiken. Dit kan op twee manieren. De eerste manier is door te werken met een drietal aanhalingstekens (dubbel of enkel):

```
1 gedicht = """
2 Strings in Python code,
3 Characters in a sequence,
4 Words and worlds unfold.
5 """
```

Probeer dit eens in Thonny in te voeren en kijk hoe het wordt weergegeven.

Als je `print(gedicht)` hebt gebruikt, zag je een net stukje tekst, verspreid over meerdere regels. Heb je echter `gedicht` direct in de shell ingetypt, dan zag je een aantal keer `\n` terug in de string. Dit is het *escapeteken* voor een nieuwe regel (*newline*).




*Een haiku met en zonder print()*

Wanneer je de driedubbele aanhalingstekens gebruikt, voegt Python zelf de `\n` toe waar nodig. Dit kun je ook zelf doen:

```
1 i = "Kun je raden\nwaar de newline komt?"
```

Misschien vind je dit wat rommelig en ben je geneigd spaties te zetten rondom de `\n`:

```
1 j = "Kun je raden \n waar de newline komt?"
```

Dat oogt beter, maar probeer eens in Thonny hoe beide gevallen eruit zien als je `print()` gebruikt. Misschien niet wat je wilt!

De `\n` is niet het enige escapeteken. Je kunt bijvoorbeeld ook tabs invoegen (`\t`) of Unicode (`\u0041`). Escapetekens beginnen altijd met een *backslash* (\). Dit kun je ook gebruiken om tekens met betekenis (zoals aanhalingstekens of de backslash zelf) te *escapen*:

```
1 k = "Dit is een \" in een tekst."
2 l = "Dit is een \\ in een tekst."
```

Gebruik steeds `print()` om te kijken hoe het eruit komt te zien.

### Raw string

Soms heb je stukken tekst met veel speciale karakters, die je allemaal wilt *escapen*. Een veel voorkomend geval zijn bestandspaden in Windows. Een bestandspad bestaat onder andere uit backslashes, die je allemaal moet *escapen*. In zulke gevallen kun je ook *raw-strings* gebruiken, door de kleine letter 'r' voor de tekst te zetten:

```
1 m = r"C:\Users\Erwin\Documents\Python"
```

Voer dit in Thonny in, en bekijk het resultaat met en zonder `print()`.

### F-strings

Vaak wil je een tekst opbouwen op basis van gegevens in je programma. Een heel kort voorbeeld zag je al aan het begin van het hoofdstuk:

```
1 def greet(name):
2     print(f"Hello, {name}!")
```

Dit is een zogenaamde functie (waar je later meer over leert). Je kunt de functie aanroepen met een tekst (een naam), en de functie zal vervolgens de naam in de groet invoegen, en dit naar het scherm printen.

De variabele (`name`) wordt gebruikt om een stuk tekst op te bouwen. Dit doe je met een zogenaamde `f-string` (kort voor *formatted string literal*). Je doet dit door een kleine letter `f` voor je zin te plaatsen, en met accolades aan te geven waar de variabele (of elke andere expressie) moet komen.

Er zijn nog andere manieren om teksten te formatteren, maar de `f-strings` zijn flexibel en eenvoudig te lezen.

### Operatoren

Bij de getallen ben je al een aantal operatoren[7] tegengekomen, waarmee je kunt rekenen.

Misschien verrassend, maar een aantal van deze operatoren werken óók bij strings.De belangrijkste zijn de `+` (concatenatie) en `*` (herhaling).

```
1 zeg_eens = 10*"a"
2 woorden = 10*"Python "
3 c_kwadraat = "a-kwadraat " + "plus " + "b-kwadraat"
```

## Collecties

Tot nu toe heb je geleerd over gegevenstypen die op zichzelf staan, zoals `int`, `float`, `None` en `bool`. Het getal `1` is een `int`, maar wat als je een *reeks* aan getallen hebt? Dat zijn collecties. Er zijn een aantal collecties in Python:

- Reeksen
- Sets
- Dictionaries

In deze paragraaf worden ze kort behandeld, in latere hoofdstukken komen ze uitgebreider aan bod.

### Reeksen

Een reeks (*sequence*) is een collectie elementen, waarbij elk element een uniek nummer (integer) heeft, op basis waarvan je het element uit de reeks kunt ophalen.Dat is een hele mond vol!Een voorbeeld maakt het duidelijker:

```
1 mijn_lijst = [1, 5, 3, 2]
2 print(mijn_lijst[2])  # 3
```

In dit voorbeeld zie je een `list` (lijst).Je ziet dat je bij een reeks een bepaald element kunt oproepen met de *index* van dat element door vierkante haakjes met het indexnummer achter de reeks te plaatsen. `mijn_lijst[2]` betekent dus: haal het element met index `2` op uit de reeks `mijn_lijst`.

Als je nog niet eerder hebt geprogrammeerd, had je waarschijnlijk verwacht dat `mijn_lijst[2]` in dit geval `5` zou teruggeven, want die staat op de tweede plaats.Maar het resultaat is `3`.Hoe kan dat?

Indexen in reeksen beginnen in Python (en in vele andere programmeertalen) bij 0, en niet bij 1. Het eerste element in een reeks heeft dus index 0. Het tweede element heeft index 1, enzovoorts.

*Tabel 1. Elementen in de lijst* `mijn_lijst`

| Element | 1 | 5 | 3 | 2 |
|---|---|---|---|---|
| Plaats | 1 | 2 | 3 | 4 |
| Index | 0 | 1 | 2 | 3 |
| Ophalen | mijn_lijst[0] | mijn_lijst[1] | mijn_lijst[2] | mijn_lijst[3] |

Dit benaderen van elementen door indexering geldt voor alle reeksen in Python. Andere operaties die voor alle reeksen gelden zijn:

- Controleren of een element zich in de reeks bevindt met `element in mijn_reeks` (of `not in`).
- Reeksen bij elkaar voegen met `mijn_reeks + mijn_reeks`
- Reeksen vermenigvuldigen met `mijn_reeks * 3`
- Meerdere elementen ophalen op basis van index met `mijn_reeks[2:5]` (*slicing*)
- De lengte van de reeks bepalen met `len(mijn_reeks)`
- Het grootste element uit de reeks halen met `max(mijn_reeks)`
- Het kleinste element uit de reeks halen met `min(mijn_reeks)`
- De index ophalen van een element uit een reeks met `mijn_reeks.index(element)`
- Het aantal keer dat een element voorkomt in een reeks met `mijn_reeks.count(element)`

De twee belangrijkste reeksen zijn de `list` en de `tuple`. Beiden zijn erg vergelijkbaar, het grote verschil is dat de `tuple` niet aanpasbaar is. Het ondersteunt dus alle hierboven beschreven methodes, maar geen methodes om bijvoorbeeld elementen aan de `tuple` toe te voegen.

```
1 mijn_lijst = [1, 5, 3, 2]  # Maak een lijst
2 mijn_lijst.append(7)  # Voeg een element toe
3 mijn_lijst[0] = 9  # Vervang het eerste element
4 print(mijn_lijst)
5
6 mijn_tuple = (1, 5, 3, 2)  # Maak een tuple
```

```
mijn_tuple.append(7)  # Geeft een fout
mijn_tuple[0] = 9  # Geeft een fout
```

Naast `lists` en `tuples` is nóg een type reeks, die ook al aan de orde is geweest. Namelijk de string. De `str` is een speciaal type reeks. Net als de `tuple` is een `str` niet-muteerbaar. Omdat een `str` een reeks is, werkt indexeren hier ook.

```
tekst = "Hello, world"
print(tekst[7])  # "w"
print(tekst[0:5])  # "Hello"
```

De reeksen `list` en `tuple` kunnen allerlei elementen bevatten. De elementen in een reeks hoeven ook niet van hetzelfde type te zijn. Ook kunnen elementen vaker voorkomen. Onderstaande zijn dus geldige lijsten:

```
mijn_lijst = [1, "twee", 3, 1, 2, "drie"]
mijn_tweede_lijst = [mijn_lijst, 4, "vijf", 5, 5]
```

Een element in een lijst kan dus zelf ook weer een lijst zijn!

### Opdracht 8: lijsten

Voer bovenstaande lijsten eens in Thonny in en druk ze af met `print()`. Hoe ziet `mijn_tweede_lijst` eruit?

▼ *Klik om het antwoord te tonen*

Het resultaat is:

```
[[1, 'twee', 3, 1, 2, 'drie'], 4, 'vijf', 5, 5]
```

De gehele eerste lijst wordt dus ook geprint.

**Een reeks is een collectie die meerdere elementen kan bevatten. Ze werken op basis van index.**

Er zijn een aantal methodes die op alle reeksen werken[8], zoals indexering, *slicing* en het ophalen van de lengte van de reeks.

#### Sets

Vergelijkbaar met reeksen zijn `sets`. Het belangrijkste verschil is echter dat een set geen dubbele waarden kan bevatten. Een set maak je aan met accolades (`{1, 2, 3}`) of de ingebouwde functie `set()`. NB: een lege `set` aanmaken kan alleen met `set()`, `{}` gebruik je om een lege `dict` aan te maken (zie hierna).

```
a = {1, 2, 3}
print(a)

b = {1, 1, 1}
print(b)
```

Sets zijn geen reeksen, en elementen ophalen uit een set werkt dan ook anders. In een Werken met sets leer je hier meer over.

### Opdracht 9: uniek

Wat zie je als je de laatste regel uitvoert? Wat gebeurt er als je het volgende invoert?

```
1 mijn_lijst = [1, 1, 2, 2, 3, 3]
2 mijn_set = set(mijn_lijst)
```

▼ *Klik om het antwoord te tonen*

Als je de laatste regel uitvoert, zie je `{1}`, een `set` met één waarde. Een `set` kan enkel unieke waarden bevatten. Alle dubbele waarden worden dus verwijderd.

Voer je de gevraagde code uit, dan zie je dat je met `set()` een `set` van een `list` kunt maken, en dat alle dubbele waarden uit de `list` verwijderd worden. Het resultaat is dus `{1, 2, 3}`.

#### Dictionaries

De laatste type collectie zijn de *mappings*. In deze cursus zul je alleen de woordenboeken gebruiken. In Python noem je dit een `dict`, kort voor *dictionary*. Een `dict` is een collectie waarbij je de elementen niet ophaalt op basis van een index, maar op basis van een sleutel. Je wijst de elementen ook altijd toe aan een sleutel.

Een `dict` maak je door sleutel-waarden paren tussen accolades te plaatsen, en de sleutel en waarden te scheiden door een dubbele punt. Sleutel-waarden paren scheidt je door een komma.

```
1 person = {
2     "name": "Erwin",
3     "role": "Author",
4     "age": 38,
5     "languages": ["Dutch", "Python"]
6 }
```

Het ophalen van een element lijkt hetzelfde als bij reeksen, alleen gebruik je nu de sleutel in plaats van de index (een integer).

```
1 print(person)  # De hele dict
2 print(person["name"])  # De waarde behorende bij sleutel "name"
3 print(person["languages"])  # De waarde behorende bij sleutel "languages"
```

In dit voorbeeld zie je dat de sleutel-waarden paren steeds op een nieuwe regel beginnen. Dit hoeft niet, maar maakt het wel leesbaarder. Hetzelfde principe kun je ook voor reeksen gebruiken:

```
1 a = ["a",
2      "b",
3      "c",
```

```
4     ]
```

Voor een korte lijst voegt dit niet zoveel toe, maar voor langere lijsten (of als de elementen langer zijn) wordt het al snel prettiger om te lezen.

Een `dict` is muteerbaar. Je kunt dus elementen toevoegen, of elementen wijzigen.

```
1 person["languages"] = ["Dutch", "English", "Python"]
2 person["hair"] = "brown"
3 print(person)
```

#### For-lussen

Een kenmerk van alle collecties is, is dat ze itereerbaar[9] zijn. Ofwel: je kunt elk element één voor één uit de collectie ophalen en daar iets mee doen. De meest gebruikte structuur hiervoor is de `for-loop`.

```
1 fruitsoorten = ["appel", "banaan", "druif", "mango", ]
2
3 for fruit in fruitsoorten:
4     print(fruit)
```

Bij `tuples` en `sets` werkt het vergelijkbaar. Woordenboeken werken net iets anders, omdat ze niet op basis van een index werken, maar op basis van sleutels. Itereren met een `for-loop` zal enkel de sleutel teruggeven.

```
1 # Fruitsoorten met het aantal dat je in huis hebt
2 fruitsoorten = {
3     "appel": 10,
4     "banaan": 5,
5     "druif": 20,
6     "mango": 0
7 }
8
9 for fruit in fruitsoorten:
10     print(fruit, fruitsoorten[fruit])
```

Itereren over een `set` is in een belangrijk opzicht anders dan bij een `list` en een `tuple`. Maak je een `list` of een `tuple` aan, dan blijft de volgorde van aanmaken altijd gehandhaafd. Dus maak je een `list` `["a", "b", "c"]`, dan is `"a"` altijd het eerste element.

Maak je echter de `set` `{"a", "b", "c"}` en je roept deze later op, dan is het eerste element misschien wel `"b"`, de volgorde kan dus wijzigen. Itereer je over een set, dan kun je er dus niet van op aan dat de volgorde hetzelfde is als toen je de `set` maakte.

## Opdracht 10: for-loop dict

Kun je voorspellen wat het resultaat is?

▼ *Klik om het antwoord te tonen*

Het resultaat is:

```
appel 10
banaan 5
druif 20
mango 0
```

Met de `for-loop` itereer je over alle sleutels van de `dict`. Die print je eerst af, daarachter gebruik je de sleutel om de waarde op te halen uit de `dict` en ook te printen.

[1] Een stakeholder is een persoon, groep of organisatie die direct of indirect belang heeft bij, invloed heeft op, of wordt beïnvloed door een bepaald project, proces of beslissing.

[2] Open-Source betekent dat de broncode van een softwareproduct vrij beschikbaar is voor het publiek om te bekijken, te wijzigen en te distribueren.

[3] De Python shell is een interactieve omgeving waarin Python code direct kan worden uitgevoerd.

[4] 'Syntax' in de context van programmeren verwijst naar de regels die bepalen hoe programma's in een bepaalde programmeertaal geschreven moeten worden. Het is een set van regels en conventies die bepaalt hoe broncode moet worden geschreven en georganiseerd.

[5] Een bug is een fout in een softwareprogramma die leidt tot onverwacht gedrag, verkeerde resultaten of problemen. Deze fouten kunnen voortkomen uit onjuiste code, onvolledige logica, misinterpretatie van specificaties of onbedoelde interacties tussen verschillende componenten.

[6] Een operator is in Python een woord of symbool dat een operatie uitvoert op één of meer waarden of variabelen. Een voorbeeld is de plus-operator (`+`), welke twee getallen optelt of twee teksten samenvoegt. Een ander voorbeeld zijn de relationele- of vergelijkingsoperators, zoals `<` (kleiner dan) en `==` (gelijk aan). Hiermee vergelijk je twee objecten.

[7] Een operator is in Python een woord of symbool dat een operatie uitvoert op één of meer waarden of variabelen. Een voorbeeld is de plus-operator (`+`), welke twee getallen optelt of twee teksten samenvoegt. Een ander voorbeeld zijn de relationele- of vergelijkingsoperators, zoals `<` (kleiner dan) en `==` (gelijk aan). Hiermee vergelijk je twee objecten.

[8] Eigenlijk niet _alle_ reeksen, er is namelijk ook nog `range()`, die ondersteunt bijvoorbeeld niet het samenvoegen met `+`. Over `range()` leer je later nog meer.

[9] Itereren is het één voor één doorlopen van items in een collectie (zoals een list, tuple, dictionary)

# Programmaverloop en variabelen

## Inleiding

In dit hoofdstuk leer je twee belangrijke concepten binnen het programmeren. Het eerste concept is het **programmaverloop**, ofwel de volgorde waarin instructies in het programma worden uitgevoerd. Die volgorde bepaal je op basis van bepaalde voorwaarden. Simpel gezegd: "Als dit, dan dat." De volgorde kun je ook bepalen op basis van herhaling: voer een instructie uit totdat aan een voorwaarde (niet meer) is voldaan.

Het tweede concept dat je in dit hoofdstuk leert, is de **variabele**. Je bent variabelen al een paar keer tegengekomen. Na het lezen van dit hoofdstuk zul je ze niet alleen gebruiken, maar ook begrijpen.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je hoe je `if`, `elif` en `else` gebruikt om het programmaverloop te bepalen
- Begrijp je hoe je `while` gebruikt om het programmaverloop te bepalen
- Begrijp je wat een variabele is en wat een referentie is
- Begrijp je wat een object is
- Begrijp je hoe muteren van objecten werkt

## Programmaverloop met if/elif/else en while

In een gemiddeld programma worden ontelbaar veel keuzes gemaakt. Als de gebruiker op "Opslaan" klikt, sla dan het bestand op. Als de gebruiker op "Annuleer" klikt, ga dan terug. Laat alle taken in de todo-app zien. Als ze zijn afgevinkt, markeer ze dan doorgestreept. Als ze het label "Urgent" hebben, markeer ze dan rood. Enzovoorts.

In deze paragraaf leer je de `if-elif-else`-statement, waarmee je dergelijke keuzes kunt programmeren. Ook leer je over de `while`-statement, waarmee je code uitvoert zolang een gegeven waar is.

### Als, anders

In de paragraaf Relationele operatoren in Eerste stappen leerde je dat je booleans (`True` en `False`) vaak zult gebruiken om het programmaverloop te beïnvloeden. Je zag daar de volgende code:

```
1 x = 20  # Wijs de waarde 20 toe aan de variabele x
2
3 if x > 20:
4     print("X is groter dan 20!")
5 else:
6     print("X is 20 of kleiner!")
```

Dergelijke `if-else`-clausules zijn één van de manieren om het verloop van je programma te bepalen. Je neemt een expressie[1] - in dit geval `x > 20` - en als die waar is voer je de code in het `if`-blok uit. Is de expressie niet waar, dan ga je verder naar het `else`-blok en voer je die code uit.

De algemene structuur is:

```
if expressie is waar:
```

```
    doe iets
else:
    doe iets anders
```

In dit geval is de expressie `x > 20` *niet* waar. Het blok op regel vier wordt dus niet uitgevoerd. Je komt daarmee in de volgende clausule terecht, op regel vijf. De `else`-clausule betekent simpelweg: "In alle andere gevallen". Hier geldt net als bij de `if`-clausule dat je de regel eindigt met een dubbele punt, en dat het blok op de regels erna uitgevoerd wordt.

In Structuur van de code: inspringen lees je nog eens terug hoe het ook alweer zit met blokken en inspringen.

Als/dan statements zijn niet alleen nuttig bij het vergelijken van getallen, maar bij alles wat een `bool` oplevert. Enkele voorbeelden:

```
1 # Controleer of een stuk fruit in de lijst voorkomt met `in`
2 fruit = ['appel', 'banaan', 'sinaasappel']
3 if 'appel' in fruit:
4     print("Appel aanwezig in de lijst van fruit")
5
6
7 # Controleer of twee objecten gelijk zijn met `is`
8 a = [1, 2, 3]
9 b = a
10 if a is b:
11     print("a en b verwijzen naar hetzelfde object")
```

*Else is optioneel*

In de voorbeelden zie je dat je `else` niet altijd hoeft te gebruiken. Hiermee zeg je eigenlijk: "Als waar, doe iets. Zo niet, ga verder met het programma.".

```
1 # `is` gebruik je ook om te controleren of een variable de waarde
  `None` heeft
2 x = 1
3 if x is None:
4     print("x is None")  # Zal niet worden uitgevoerd
5
6 print("x is niet None. Het programma gaat hier verder.")
```

## En, Of

Vaak wil je het programmaverloop bepalen op basis van meerdere voorwaarden, bijvoorbeeld als twee expressies waar zijn. Dit doe je met `and`:

```
1 x = 5
2 y = 10
3 if x > 3 and y > 5:
```

```
4     # Beide zijn waar, dus dit blok wordt uitgevoerd
5     print("x is groter dan 3 EN y is groter dan 5")
6 else:
7     print("Eén van beide expressies is niet waar")
```

Als je `and` gebruikt, zeg je dat *beide* expressies waar moeten zijn om de gehele expressie waar te laten zijn. Dus alleen als `x` groter is dan drie én `y` groter is dan vijf wordt regel vier uitgevoerd. Als één van de twee *niet* waar is, kom je in de `else`-clausule terecht.

Let hierbij op dat Python het zogenaamde *short-circuit* principe hanteert. Als het eerste argument (`x > 3` in dit geval) `False` is, gaat Python direct door naar de `else`-clausule. Het tweede argument (`y > 5`) wordt dan niet meer bekeken.

Het kan ook voorkomen dat je iets wilt doen als van meerdere vergelijkingen er minstens één waar is. Dit doe je met `or`:

```
1 x = 5
2 y = 3
3 if x > 3 or y > 5:
4     # Eén van beide is waar, dus dit blok wordt uitgevoerd
5     print("x is groter dan 3 OF y is groter dan 5")
6 else:
7     print("Beide expressies zijn niet waar")
```

Let op, met `or` test je of *tenminste* één van de opgegeven expressies waar is. Als alle expressies waar zijn, is er dus tenminste één waar, en zal het blok worden uitgevoerd:

```
1 x = 5
2 y = 10
3 if x > 3 or y > 5:
4     # Beide zijn waar, dus dit blok wordt uitgevoerd
5     print("x is groter dan 3 OF y is groter dan 5")
6 else:
7     print("Beide expressies zijn niet waar")
```

Ook hier geldt weer dat het *short-circuit* principe geldt. Als het eerste argument (`x > 3`) `True` is, wordt het blok direct uitgevoerd (één van de twee is immers waar). Het tweede argument (`y > 5`) wordt niet meer bekeken.

### Opdracht 1: Alarm

Schrijf een eenvoudig alarmsysteem voor een ruimte met twee deuren. Neem aan dat het alarm aan staat. Als er een deur open staat, laat dan het alarm afgaan.

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is als volgt:

```
voordeur_open = False
achterdeur_open = True
```

```
if voordeur_open or achterdeur_open:
    print("Alarm! Een deur is open!")
else:
    print("Alles is veilig. Alle deuren zijn gesloten.")
```

### Als, anders-als, anders

Tot nu toe zag je de structuur: "Als waar, doe iets. Anders...". Maar er is nog een mogelijkheid om je programmaverloop te bepalen.

```
1 leeftijd = 83
2 if leeftijd < 4:
3     toegangsprijs = 0
4 elif leeftijd < 65:
5     toegangsprijs = 40
6 else:
7     toegangsprijs = 20
8
9 print(f"Je toegangsprijs is €{toegangsprijs}.")
```

Hier test je eerst of `leeftijd` kleiner is dan vier. Als dat niet zo is, zoals in het voorbeeld, kom je in de volgende clausule. Dit is `elif`, kort voor `else-if`. Hier test je of `leeftijd` kleiner is dan 65. Als dit ook niet waar is, dan ga je naar de `else`-clausule.

#### Opdracht 2: €40

Welke waarde moet je `leeftijd` geven om de toegangsprijs €40 te laten zijn?

▼ *Klik om het antwoord te tonen*

Als je `leeftijd` een waarde vanaf 4 tot 65 (maar geen 65), dan is de toegangsprijs €40.

Je kunt in een `else-elif-else`-blok meerdere `elif`-clausules gebruiken:

```
1 leeftijd = 83
2 if leeftijd < 4:
3     toegangsprijs = 0
4 elif leeftijd < 18:
5     toegangsprijs = 25
6 elif leeftijd < 65:
7     toegangsprijs = 40
8 else:
9     toegangsprijs = 20
10
11 print(f"Je toegangsprijs is €{toegangsprijs}.")
```

Let hierbij op dat de code altijd van boven naar beneden wordt uitgevoerd. De eerste expressie die waar is, wordt uitgevoerd, en de `if-elif-else`-ketting zal daarna zijn afgerond. Dit betekent dat als je meerdere `elif`-statements hebt die waar zijn, alleen de eerste wordt uitgevoerd. Kortom: er wordt altijd maar één blok uitgevoerd uit een `if-elif-else`-ketting. Net als bij een `if-else`-statement, overigens.

```
leeftijd = 17
if leeftijd < 4:
    toegangsprijs = 0
elif leeftijd < 18:  # Waar
    toegangsprijs = 25
elif leeftijd < 65:  # Ook waar, maar nooit uitgevoerd
    toegangsprijs = 40
else:
    toegangsprijs = 20

print(f"Je toegangsprijs is €{toegangsprijs}.")
```

### Opdracht 3: Verkeerslicht

Hoe zou je met `if-elif-else` een verkeerslicht vormgeven? Gebruik de variabele `actie` voor de actie die het voertuig voor het verkeerslicht moet uitvoeren bij elke kleur (rood, oranje, groen).

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is:

```
licht = ""
actie = None

if licht == "rood":
    actie = "stoppen"
elif licht == "oranje":
    actie = "stoppen als het lukt"
elif licht == "groen":
    actie = "doorrijden"
else:
    actie = "wachten op een verkeersleider, de verkeerslichten zijn kapot"

print(f"Het voertuig moet {actie}.")
```

## While

Naast `if-elif-else` is er nog een manier om het programmaverloop te bepalen op basis van expressies die een `bool` opleveren. Met `while` voer je iets uit *zolang iets waar is*.

```
# Eenvoudig voorbeeld
huidig_getal = 1
while huidig_getal <= 10:
```

```
4     print(huidig_getal)
5     huidig_getal += 1
```

Op regel 1 begin je met tellen door de waarde 1 toe te wijzen aan de variabele `huidig_getal`. Op regel 2 stel je in dat de `while-loop` blijft draaien zolang `huidig_getal` 10 of kleiner is. Zodra `huidig_getal` 11 wordt, is de expressie niet meer waar en breekt de *loop* af.

De code in de *loop* print op regel drie het huidige getal. Op regel vier wordt het huidige getal met één opgehoogd (`huidig_getal += 1` is een korte manier om `huidig_getal = huidig_getal + 1` te schrijven). Na regel vier is de *loop* klaar en begint het opnieuw, zolang `huidig_getal <= 10` waar is.

*Loops*

In For-lussen heb je al kort de `for-loop` behandeld. Zowel `for` als `while` gebruik je om een stuk code meerdere malen uit te voeren. Het verschil zit in de manier waarop wordt bepaald hoe vaak de code wordt uitgevoerd.

Bij een `for-loop` wordt de code voor elk element in een reeks uitgevoerd (itereren). Bij een `while-loop` wordt de code uitgevoerd zolang de opgegeven expressie waar is (of je de *loop* met `break` stopt.).

Een `while-loop` gebruik je om continue iets uit te voeren tot een bepaalde conditie is bereikt. Een voorbeeld is om een programma net zo lang te laten draaien totdat de gebruiker vertelt om het te stoppen.

```
1 # User input
2 instructie = "Vertel me iets, en ik herhaal het voor je. " \
3          "Typ 'exit' om te stoppen. "
4 bericht = ""
5
6 while bericht != "exit":
7     bericht = input(instructie)
8     print(bericht, "\n")
```

In bovenstaande code vraag je de gebruiker om input met `input(instructie)` (de instructie is op regel één gedefinieerd). Zolang de gebruiker geen 'exit' typt, zal het programma het bericht printen en opnieuw om input vragen.

### Opdracht 4: Papegaai

Als de gebruiker 'exit' typt, print het programma dit ook voordat het stopt. Hoe zorg je ervoor dat dit niet het geval is?

▼ *Klik om het antwoord te tonen*

Als je 'exit' niet wilt printen, kun je dit voorkomen met een `if`-statement:

```
while bericht != "exit":
    bericht = input(instructie)
    if bericht != "exit":
        print(bericht, "\n")
```

Een alternatieve manier om een *loop* te stoppen is met de `break`-statement.

```
1 while True:
2     bericht = input(instructie)
3     if bericht == "exit":
4         break
5 
6     print(bericht, "\n")
```

Je ziet dat regel een begint met `while True`, ofwel: de conditie is altijd waar en de *loop* zal dus oneindig draaien. Op regel drie en vier zie je: als het bericht 'exit' is, 'breek' dan uit de loop. Regel zes zal in dat geval dus niet uitgevoerd worden.

Het tegenovergestelde van `break` is `continue`. Met `continue` vertel je de code dat je de *huidige* iteratie[2] van de *loop* wilt afbreken en terug wil naar het begin van de *loop*, zonder de code eronder uit te voeren. Waar je met `break` dus de hele *loop* stopt, zorg je er met `continue` voor dat de huidige ronde stopt, maar de *loop* als geheel wel door blijft gaan.

```
1 while True:
2     bericht = input(instructie)
3     if bericht == "Python is stom":
4         continue
5 
6     if bericht == "exit":
7         break
8 
9     print(bericht, "\n")
```

De `break` is verplaatst naar regel zes en zeven. Op regel drie en vier zorg je ervoor dat als een gebruiker 'Python is stom' typt, de *loop* opnieuw begint. De code eronder wordt niet uitgevoerd, en het bericht wordt niet geprint (en terecht!).

De `break` en `continue` statements werken in een `while-loop`, maar ook binnen een `for-loop`.

Mocht je ooit in de situatie komen dat je per ongeluk een oneindige *loop* hebt gemaakt, zonder optie om het te stoppen, dan kun je in de shell op `ctrl` en `c` drukken, het programma stopt dan. Je kunt uiteraard ook simpelweg de shell afsluiten.

### Opdracht 5: Hoogste getal

Schrijf een Python-programma dat de gebruiker vraagt om een getal in te voeren. Het programma moet blijven vragen om getallen tot de gebruiker "stop" intypt. Wanneer "stop" wordt ingevoerd, moet het programma het hoogste ingevoerde getal afdrukken.

▼ *Klik om het antwoord te tonen*

Een voorbeelduitwerking is:

```
hoogste_getal = 0  # Begin bij 0
```

```python
while True:
    # Vraag de gebruiker om een getal en sla het op
    invoer = input("Voer een getal in. Typ 'stop' om te stoppen.\n Invoer: ")

    # Als de invoer 'stop' is, druk het hoogste getal af en stop de while-
loop
    if invoer == "stop":
        print(hoogste_getal)
        break

    # Invoer is altijd tekst, zet om naar int en vergelijk met het huidige
    # hoogste getal. Als de invoer hoger is, sla het op in hoogste_getal
    if int(invoer) > hoogste_getal:
        hoogste_getal = int(invoer)
```

# Variabelen

In voorgaande stof ben je al een aantal keer variabelen tegengekomen, de meeste voorbeelden maken gebruik van één of meer variabelen. Uit de voorbeelden blijkt al wat je met variabelen kunt: je koppelt ze aan waarden om ze later te gebruiken. In deze paragraaf ga je dieper in op variabelen, zodat je goed begrijpt wat een variabele écht is en hoe ze precies werken.

## Labels

In Python wordt data vertegenwoordigd door *objecten*. Elk object heeft een ID, een type en een waarde. Als een object eenmaal is aangemaakt, verandert het ID niet meer. Je kunt het zien als het adres in het geheugen van de computer.

**Alles is een object**

In feite is *alles* een object in Python, dus niet bijvoorbeeld alleen de string "Python", maar ook een functie.

Het ID van een object haal je op met `id()`. Het type haal je op met `type()`. Met `is` vergelijk je of twee objecten hetzelfde ID hebben (en dus naar dezelfde plek in het geheugen verwijzen).

In Python gebruik je variabelen om te verwijzen naar objecten, en daarmee naar de waarde.

Als je het volgende uitvoert: `x = 42` gebeurt er eigenlijk het volgende:

1. Er wordt een **object** aangemaakt in het geheugen met een ID, het type `int` en de waarde 42.
2. Er wordt een **label** (de variabele) aangemaakt, die verwijst naar dit object.

Voer je nu `id(x)` uit, dan vraag je dus feitelijk het ID op van het object waar `x` naar verwijst.

Zie het als een kast met verschillende laden. In elke lade stop je iets, sokken in de eerste, broeken in de tweede, etc. De kast is het geheugen, de lades zijn specifieke locaties in het geheugen en de sokken zijn de objecten (met de waarde 'Blauw' en het type 'Sok').

Om de objecten te gebruiken - je wilt je sokken aantrekken - is het nodig om ze te vinden. Daarom heb je een label op elke la geplakt. Een label toewijzen aan een object doe je met het `=`-teken. In het hoofdstuk Eerste

stappen las je al dat Python een *dynamisch getypeerde* taal is, wat ervoor zorgt dat je bij het toewijzen van een variabele geen type hoeft op te geven.

```
1 sokken = "10 paar sokken"  # str
2 broeken = "3 broeken"  # str
3 geld = 1000  # int
4 lege_lade = None  # Lege lade
```

Het is belangrijk om variabelen zinvolle namen te geven die de opgeslagen gegevens of hun doel in de code beschrijven. Dus de `x`, `y` en `z` die je in de voorbeelden vaak ziet, mógen wel maar zijn in de praktijk niet handig. Er zijn een aantal regels voor naamgeving:

- Variabelen moeten beginnen met een letter (a-z, A-Z) of een underscore (_).
- Ze kunnen bestaan uit letters, cijfers en underscores, maar geen spaties.
- Ze zijn hoofdlettergevoelig, wat betekent dat `leeftijd` en `Leeftijd` twee verschillende variabelen zijn.

Daarnaast zijn er nog conventies - geen harde regels, maar in de praktijk is het wel goed om je eraan te houden:

- Gebruik kleine letters en underscores (ook wel 'snake_case' genoemd). Bijvoorbeeld: `aantal_dagen`, `start_tijd`.
- Gebruik geen gereserveerde *keywords*, zoals `if`, `while`, `class`, en `import`. Gebruik deze woorden niet als variabelenamen om verwarring en fouten te voorkomen.
- Gebruik constanten in hoofdletters. Als je een variabele hebt die een constante waarde vertegenwoordigt en niet mag worden gewijzigd, schrijf dan de variabelenaam in hoofdletters en scheid de woorden met underscores. Bijvoorbeeld: `MAX_AANTAL`.

Naast conventies voor de schrijfwijze van variabelen zijn er vele andere conventies. Eerder las je al over het inspringen met vier spaties, bijvoorbeeld. Al deze conventies lees je terug in 'PEP 8 - Style Guide for Python Code' (peps.python.org/pep-0008/).

Zodra een object een label heeft gekregen - zodra je een variabele hebt aangemaakt dus - kun je het gebruiken in je programma. Dit kan eenmalig, maar ook vaker.

```
1 print(f"In lade 1 vind je {sokken}. In lade 2 vind je {broeken}.")
2 print(f"Maar nu heb ik alleen {sokken} nodig.")  # `sokken` nogmaals gebruikt
```

Als je een variabele hebt toegewezen aan een object, is dit niet voor altijd. Je kunt de verwijzing eenvoudig wijzigen naar een ander object.

```
1 sokken = "8 paar sokken"  # Was "10 paar sokken"
2 print(f"In lade 1 vind je {sokken}. In lade 2 vind je {broeken}.")
```

Het hoeft zelfs niet van hetzelfde type te zijn:

```
1 sokken = 1000  # van str naar int
2 print(f"In lade 1 vind je {sokken} sokken. In lade 2 vind je {broeken}.")
3
4 print(type(sokken))  # <class 'int'>
```

*Objecten, verwijzingen en mutaties*

Als je de code bekijkt dan lijkt het net of je de waarden toewijst aan een variabele, en zo zul je er in de praktijk ook vaak over denken. Toch is het belangrijk om te onthouden dat een variabele een *verwijzing* is *naar* een *object* die een *waarde* bevat. Het object met waarde 1000 is ergens opgeslagen in het geheugen van je computer en vervolgens verwijs je er naar met een label.

Neem bijvoorbeeld een object van het type `str` met de waarde "10 paar sokken". Doe je nu in de code het volgende:

```
1 sokken = "10 paar sokken"  # str
```

Dan maakt Python eerst het *object* "10 paar sokken" van het type `str` aan. Vervolgens maakt Python de *objectreferentie* (label) `sokken` aan en zorgt ervoor dat `sokken` naar het object van type `str` met waarde "10 paar sokken" verwijst. In de praktijk zul je zeggen `sokken` is van het type `str`, maar feitelijk is het *object* waar `sokken` naar verwijst (de zin "10 paar sokken") van het type `str`.

Deze werkwijze heeft als gevolg dat twee variabelen naar hetzelfde object kunnen verwijzen.

```
1 x = 5
2 y = x
3
4 print(x is y)
5 print(id(x))
6 print(id(y))
```

Bij *mutable* variabelen betekent dit daardoor ook dat als je de waarde wijzigt via de ene variabele, de waarde voor de andere variabele ook wijzigt.

```
1 x = [1, 2, 3]
2 y = x
3
4 x[0] = 9
5 print(y)
```

Bekijk de eerste drie segmenten van deze video voor een gevisualiseerde uitleg.

[1] Een expressie is in Python code dat een waarde oplevert. Dus: x > 20 is een expressie (levert `True` of `False` op. Maar een functie kan ook een expressie zijn, net als 1 + 1.

[2] Itereren is het één voor één doorlopen van items in een collectie (zoals een list, tuple, dictionary)

# Werken met lijsten

## Inleiding

Lijsten (`list`) in Python zijn een van de belangrijkste datatypes waarmee je als beginnend programmeur in aanraking zult komen. In Eerste stappen heb je er al kort kennis mee gemaakt. In dit hoofdstuk ga je dieper in op lijsten en leer je er verder mee werken.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je wat een `list` is
- Begrijp je in welke situaties je lijsten toe kunt passen
- Begrijp je hoe je met elementen in een `list` kunt werken

## Lijsten: een inleiding

In deze paragraaf leer je wat een `list` is, wat de specifieke kenmerken van lijsten zijn en hoe lijsten zich onderscheiden van andere **collecties**. Je leert ook in welke situaties je lijsten kunt gebruiken.

In de Reeksen leerde je al dat een `list` een *reeks* is, en dat de volgende methodes beschikbaar zijn:

- Controleren of een element zich in de reeks bevindt met `element in mijn_reeks` (of `not in`).
- Reeksen bij elkaar voegen met `mijn_reeks + mijn_reeks`
- Reeksen vermenigvuldigen met `mijn_reeks * 3`
- Meerdere elementen ophalen op basis van index met `mijn_reeks[2:5]` (*slicing*)
- De lengte van de reeks bepalen met `len(mijn_reeks)`
- Het grootste element uit de reeks halen met `max(mijn_reeks)`
- Het kleinste element uit de reeks halen met `min(mijn_reeks)`
- De index ophalen van een element uit een reeks met `mijn_reeks.index(element)`
- Het aantal keer dat een element voorkomt in een reeks met `mijn_reeks.count(element)`

Verderop leer je hoe je deze methodes toepast.

### Een `list` maken

Een `list` kun je op verschillende manieren maken. Meestal zul je de vierkante haken gebruiken (`[]`) of de ingebouwde functie `list()`. Je kunt een `list` leeg beginnen, maar ook direct items toevoegen:

```
1 # Lijsten maken
2 lege_lijst_1 = []
3 lege_lijst_2 = list()
4 lijst_met_items = [1, 2, 3, ]
```

Items in een `list` scheidt je door een komma.

*Komma's*

Je zult vaak zien dat er achter het laatste item van een `list` ook een komma wordt geplaatst. Dit is niet verplicht. De laatst komma behoed je echter wel voor een mogelijke fout. Stel dat je een initiële versie van een `list` hebt, en je wilt later (handmatig) een item toevoegen:

```
# Initiële lijst
mijn_lijst = [
    "appel",
    "banaan",
    "peer"
]

# Aangepaste lijst, komma vergeten
mijn_lijst = [
    "appel",
    "banaan",
    "peer"
    "kiwi"
]

print(mijn_lijst)
# ['appel', 'banaan', 'peerkiwi']
```

Print `mijn_lijst` maar eens, en bekijk het resultaat. Door in de initiële `list` achter elk item een komma te plaatsen, voorkom je dit foutje.

## Kenmerken van lijsten

De belangrijkste kenmerken van een `list` leerde je al in Reeksen:

- Een `list` is een vorm van een **reeks**, ofwel een collectie elementen.
- Je haalt elementen op basis van een **index** op uit een `list`.
- De index begint bij 0, het eerste element haal je dus zo op: `mijn_lijst[0]`.
- Een `list` is **muteerbaar** (aanpasbaar). Je kunt elementen toevoegen, aanpassen en verwijderen.
- De elementen in een `list` hoeven niet van hetzelfde type te zijn.

### Opdracht 1: min, max, len

Schrijf code dat het kleinste getal, het grootste getal en de lengte van de onderstaande `list` print.

```
mijn_lijst = [1, 100, 1000, 2, ]
```

▼ *Klik om het antwoord te tonen*

```
print(min(mijn_lijst))  # 1
print(max(mijn_lijst))  # 1000
print(len(mijn_lijst))  # 4
```

### Wanneer gebruik je een lijst?

Nu je weet wat de belangrijkste kenmerken van een `list` zijn rijst de vraag: wanneer gebruik je een `list`?

Aangezien een `list` (meestal) meerdere elementen bevat, is het een logisch gegevenstype om te gebruiken op het moment dat je bepaalde handelingen op verschillende elementen wilt toepassen. Meestal zul je dit dan doen in een **iteratie**[1], bijvoorbeeld als je elke actieve gebruiker van je systeem een e-mail wilt sturen:

```
1 # Dummy code
2 for user in users:
3     # Mail elke user indien actief
4     if user.is_active:
5         send_mail(user)
```

Vaak gaat de afweging tussen een `list` en `tuple`. Beiden zijn reeksen met elementen waar je iets mee wilt doen. *Maak* je een collectie op basis van gegevens, dan valt de `tuple` af, omdat deze niet-aanpasbaar is. Bijvoorbeeld:

```
 1 # Collectie maken (list)
 2 users = []  # Lege lijst
 3
 4 # Open een CSV bestand
 5 with open("users.csv") as csvfile:
 6     # Gebruik de csv module om het bestand in te lezen
 7     csv_reader = csv.DictReader(csvfile)
 8
 9     # Voor elke regel in het CSV-bestand, haal de naam op uit de kolom
10     # "Name" en voeg dit toe aan de list users
11     for row in csv_reader:
12         users.append(row["Name"])
```

In bovenstaande (vereenvoudigd) voorbeeld lees je een CSV-bestand in, itereer je over elke rij en voeg je de naam die je in het CVS-bestand vindt toe aan de `list` `users`, die je op regel één hebt aangemaakt. Een resultaat ziet er als volgt uit:

```
1 print(users)
2 # ["Piet", "Marie", "Ali"]
```

Het geeft niet als je niet elke stap begrijpt, als maar duidelijk is dat je een `list` *opbouwt*. In een `for-loop` voeg je steeds een element toe aan een `list`. Met een `tuple` zal dit niet werken.

Je verkiest een `tuple` boven een `list` als de reeks niet zal wijzigen. Een `tuple` verbruikt minder geheugen en is hierdoor efficiënter. Je ziet dit vaak met korte reeksen die binnen het programma worden rondgegeven, bijvoorbeeld de coördinaten van een plaats: (long, lat).

#### Opdracht 2: list opbouwen

Neem de volgende `list`:

```
1 mijn_lijst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Schrijf een stukje code waarin je begint met een lege `list` en deze met een `for-loop` vult met het kwadraat van elk van de elementen in bovenstaande `list`.

▼ *Klik om het antwoord te tonen*

```
1 kwadraten = []
2 for getal in mijn_lijst:
3     kwadraten.append(getal ** 2)
4
5 print(kwadraten)
```

# Werken met elementen in een lijst

In deze paragraaf leer je werken met de elementen in een `list`: ophalen, aanpassen en verwijderen.

## Elementen uit een lijst ophalen

Een veelvoorkomende handeling is het ophalen van één of meerdere elementen uit een `list`.

### Eén element ophalen

In hoofdstuk 1 in de paragraaf Collecties heb je al gezien hoe je een enkel element ophaalt uit een `list`, op basis van de index:

```
1 eerste_item = lijst_met_items[0]
2 tweede_item = lijst_met_items[1]
```

Met de nul-gebaseerde indexering haal je het eerste element op met `[0]`, het tweede element met `[1]`, etc.

Indexering kan vanaf het begin van de `list`, maar ook vanaf het *einde*. In plaats van positieve integers gebruik je dan negatieve integers. Het laatste element (dus het eerste gezien vanaf het eind) begint met index -1:

```
1 lijst_met_items = ["a", "b", "c", ]
2 laatste_item = lijst_met_items[-1]  # "c"
```

Het een-na-laatste element haal je op met `[-2]`, het twee-na-laatste element met `[-3]`, etc. Waar indexering vanaf het begin nul-gebaseerd is, kun je indexering vanaf het einde beschouwen als een-gebaseerd (met een min-teken ervoor).

*Tabel 2. Elementen in de lijst* `mijn_lijst`

| Element | 1 | 5 | 3 | 2 |
|---|---|---|---|---|
| Plaats vanaf het eind | 4 | 3 | 2 | 1 |
| Index | -4 | -3 | -2 | -1 |
| Ophalen | mijn_lijst[-4] | mijn_lijst[-3] | mijn_lijst[-2] | mijn_lijst[-1] |

#### Slicen

Soms wil je niet één, maar meerdere elementen tegelijk ophalen uit een `list`. Dit kan met **slicen**. De notatie hiervoor is `[1:3]`.

```
1 mijn_lijst = [1, "b", "drie", 4, "V", ]
2 mijn_slice = mijn_lijst[1:3]  # ["b", "drie"]
```

Het resultaat is een nieuwe `list`. Let op dat de eerst index *vanaf* is en de tweede *tot* (en níet tot en met). Met `[1:3]` haal je dus elementen met index één en twee op, maar niet het element met index drie.

Zowel de index voor als na de dubbele punt is optioneel. Laat je de eerste index weg, dan begin je bij het begin (index 0). Laat je de laatste index weg, dan *slice* je tot het einde.

```
1 mijn_lijst = [1, "b", "drie", 4, "V", ]
2 mijn_slice = mijn_lijst[:3]  # [1, "b", "drie"]
3 mijn_slice = mijn_lijst[3:]  # [4, "V"]
```

Omdat beide indexen optioneel zijn, kun je ze ook beide weglaten:

```
1 mijn_lijst = [1, "b", "drie", 4, "V", ]
2 mijn_slice = mijn_lijst[:]  # [1, "b", "drie", 4, "V"]
```

Je maakt dan een *slice* vanaf het begin, tot het einde. Feitelijk maak je dan dus een kopie van de eerste `list`.

Een *slice* werkt ook met negatieve indexering.

##### Opdracht 3: negatieve slice

Neem onderstaande `list`:

```
1 mijn_lijst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Haal hier een nieuwe `list` uit met de waarde `[4, 5, 6, 7]`. Gebruik hiervoor *negatieve* indexering in een *slice*.

▼ *Klik om het antwoord te tonen*

```
1 mijn_slice = mijn_lijst[-7:-3]
2 print(mijn_slice)
```

Let bij het *slicen* dus op dat je negatieve indexering gebruikt, maar de *slice* nog steeds van links naar rechts werkt. Misschien had je verwacht `[-3:-7]` te moeten doen, maar dit werkt niet.

### Controleren of een element zich in een lijst bevindt

Soms wil je eerst weten of een element zich in een `list` bevindt. De meest handige manier om dit te doen is

met `in`, of als je wilt weten of een element zich *niet* in een `list` bevindt met `not in`:

```
1 mijn_lijst = ["a", "a", "b", "b", "c", ]
2 print("a" in mijn_lijst)  # True
3 print("d" in mijn_lijst)  # False
4 print("e" not in mijn_lijst)  # True
```

Andere manieren om te controleren of een element zich in een `list` bevindt zijn `.index()` en `.count()`. Met `.index()` geef je een waarde op, en krijg je de index van het *eerste* element met die waarde terug. Met `.count()` geef je een waarde op en krijg je terug hoeveel elementen met die waarde zich in de `list` bevinden.

```
1 mijn_lijst = ["a", "a", "b", "b", "c", ]
2
3 # Met index()
4 print(mijn_lijst.index("a"))  # 0
5 print(mijn_lijst.index("d"))  # Fout
6
7 # Met count()
8 print(mijn_lijst.count("b"))  # 2
9 print(mijn_lijst.count("d"))  # 0
```

Hoe je de fout op regel vijf verwerkt leer je het hoofdstuk Werken met uitzonderingen. Het is afhankelijk van de context welk instrument je gebruikt. Soms is `in` voldoende, maar zoals uit bovenstaande voorbeelden blijkt weet je dan nog *niet* dat de waarde `a` zich vaker in de `list` bevindt.

### Opdracht 4: index

Neem de volgende `list`:

```
1 mijn_lijst = ["a", "b", "c", ]
```

Schrijf code waarmee je de index van een element ophaalt. Zorg ervoor dat als je de index van een niet-bestaand element wilt ophalen, er geen fout ontstaat.

▼ *Klik om het antwoord te tonen*

```
1 te_zoeken = "b"  # Probeer ook eens met "d"
2
3 if te_zoeken in mijn_lijst:
4     print(mijn_lijst.index(te_zoeken))
5 else:
6     print("Element niet in de lijst")
```

## Itereren langs elementen in een lijst

Het itereren langs elementen in een `list` heb je al meerdere malen voorbij zien komen in voorgaande

hoofdstukken. Met een `for-loop` ga je elk element in een `list` langs en kun je daar iets mee doen. Wát je ermee doet is aan jou, het blok in een `for-loop` kan net zo lang of kort zijn als nodig is.

Een `for`-statement heeft de volgende opbouw:

```
1 for <naam> in <lijst>:
2     blok uitgevoerd voor elk element <naam>
```

Elke ronde wordt er een element uit de `list` gehaald en toegewezen aan de variabele `<naam>`. In de rest van de `for-loop` kun je die variabelen dan ook gebruiken zoals elke variabele.

Hoe je de variabele noemt, maakt in principe niet uit. Soms zie je dat het heel algemeen blijft (`i`, `item`, `x`), maar handiger is het om het iets beschrijvender te maken. Bevat je `list` boektitels, dan is `book` logisch. Bevat de `list` gebruikers, dan noem je het `user`, etc.

Een veelgehoorde uitdrukking in het programmeren is dat code vaker wordt gelezen dan geschreven. Jij en je collega's lezen code om te begrijpen wat er gaande is, wat een functie ook alweer doet, etc.

Daarom is het belangrijk om variabelen, functies, klassen en namen in een `for-loop` duidelijke namen te geven. Elke naam geeft inzicht in wat er gaande is.

Dit geldt ook voor projecten waar je in je eentje aan werkt. Over zes maanden ben je blij als je oude code duidelijk is omschreven!

Soms wil je niet alleen dat het element aan de variabele wordt gekoppeld, maar wil je ook de index van de elementen bijhouden. Dit doe je met `enumerate`. Optioneel geef je op waar de index moet starten.

```
1 users = ["marie", "john", "ali", ]
2
3 # Index start bij 0
4 for index, user in enumerate(users):
5     print(f"Index {index}: {user}")
6
7 # Index start bij 1
8 for ronde, user in enumerate(users, 1):
9     print(f"Ronde {ronde}: {user}")
```

In plaats van één naam op te geven in de `for`-statement, geef je er twee op gescheiden door een komma. Verder roep je de functie `enumerate` aan met de `list` als argument (en optioneel de start van de index).

Je ziet dat het ook hier weer niet uitmaakt hoé je de variabelen noemt. In het eerste geval is het `index, user` en in het tweede geval `ronde, user`.

## Elementen in een lijst sorteren en omkeren

Een `list` kun je zowel sorteren als omkeren.

### Sorteren

Met de `.sort()` methode sorteer je de elementen in een `list`. Standaard is de sortering van laag naar hoog. Je kunt `.sort(reverse=True)` gebruiken om de sortering om te draaien.

```
1 mijn_lijst = ["b", "a", "d", "c", ]
2 mijn_lijst.sort()
3 print(mijn_lijst)  # ["a", "b", "c", "d"]
```

Dit sorteren past de bestaande `list` aan, er wordt dus geen nieuw object aangemaakt. Het sorteren werkt alleen als de elementen in de `list` met elkaar vergeleken kunnen worden.

```
1 mijn_lijst = ["a", 2, "b", 3, ]
2 mijn_lijst.sort()  # Fout
```

Voer je deze code uit, dan krijg je een foutmelding die aangeeft dat het sorteren niet lukt.

Tot slot kun je opgeven hoe je wilt sorteren, door een functie mee te geven aan `.sort()` die de sleutel ophaalt uit elk element en dat gebruikt om te sorteren. Je kunt bijvoorbeeld de ingebouwde functie `len()` gebruikte om de lengte van een `str` te bepalen.

```
1 print(len("test"))  # 4
2
3 # Gebruik len als sleutel om te sorteren
4 mijn_lijst = ["aaa", "bb", "c", ]
5 mijn_lijst.sort(key=len)
6 print(mijn_lijst)  # ["c", "bb", "aaa"]
```

Met `sort(key=len)` zorg je ervoor dat eerst `len()` op elk element wordt uitgevoerd, dit levert voor elk element een waarde op (3, 2 en 1 in dit voorbeeld) en deze waarden worden gebruikt om de `list` te sorteren.

In plaats van een `list` *in-place* te sorteren, kun je ook de ingebouwde functie `sorted()` gebruiken. Dit doet hetzelfde als `sort()`, maar levert een nieuwe `list` op. Het voordeel van `sorted` is dat het niet alleen op `list` werkt, maar bij alle *iterables*, zoals een `tuple`.

#### Omkeren

Naast een `list` sorteren, kun je een `list` ook omkeren. Dit doe je met de methode `.reverse()` of de ingebouwde functie `reversed()`.

```
1 mijn_lijst = [1, 2, 3, 4, ]
2 mijn_lijst.reverse()
3 print(mijn_lijst)  # [4, 3, 2, 1]
```

Net als bij `.sort()` levert `.reverse()` geen nieuw object op. Met de ingebouwde functie `reversed()` bereik je dit wel.

### Elementen aan een lijst toevoegen

Een veel voorkomende taak bij een `list` is het toevoegen van elementen. Hiervoor zijn drie methodes beschikbaar.

- `.append()`

- `.insert()`
- `.extend()`

#### append

Met `.append()` voeg je een element toe aan het einde van de `list`. Je kunt één element tegelijk toevoegen. Bijvoorbeeld:

```
1 mijn_lijst = []
2 mijn_lijst.append("a")
3 print(mijn_lijst)  # ["a"]
```

Je kunt elk type element toevoegen, let hierbij wel op dat als je bijvoorbeeld een `list` of `tuple` toevoegt, het één element aan het einde van de oorspronkelijke `list` wordt:

```
1 mijn_lijst = ["a", ]
2 mijn_lijst.append([1, 2, 3])  # Voeg een lijst toe
3 print(mijn_lijst)  # ["a", [1, 2, 3]]
```

#### insert

Met `.insert()` voeg je een enkele item toe *in* de `list`. Je geeft het element dat je wilt toevoegen door, samen met de index waar het element geplaatst moet worden. De elementen erna zullen allemaal één index opschuiven.

```
1 mijn_lijst = ["a", "c", "d", ]
2 mijn_lijst.insert(1, "b")  # Voeg 'b' toe op index 1
3 print(mijn_lijst)  # ["a", "b", "c", "d"]
```

Net als bij `.append()` kun je bijvoorbeeld een `list` toevoegen, maar wordt dit één element in de oorspronkelijke `list`.

#### extend

Met `.extend()` voeg je alle elementen uit een *iterable* (zoals een `list` of een `tuple`) toe aan het einde van de huidige `list`:

```
1 mijn_lijst = ["a", "b", "c", ]
2 mijn_tuple = (1, 2, 3, )
3 mijn_lijst.extend(mijn_tuple)
4 print(mijn_lijst)  # ["a", "b", "c", 1, 2, 3]
```

### Elementen uit een lijst verwijderen

Net als je op verschillende manieren elementen aan een `list` kunt toevoegen, zijn er ook verschillende manieren om één of meerdere elementen te verwijderen.

#### remove

Met de `.remove()` methode verwijder je een element op basis van de waarde. Het *eerste* element in de `list` met de opgegeven waarde wordt verwijderd (de rest dus niet, als er meerde elementen met die waarde zijn). Wordt de waarde niet gevonden, dan ontstaat een fout.

```
1 mijn_lijst = [1, 2, 3, 2, ]
2 mijn_lijst.remove(2)  # Verwijder element met waarde 2
3 print(mijn_lijst)  # [1, 3, 2]
4 mijn_lijst.remove(0)  # Fout
```

#### pop

Met de `.pop()` methode verwijder je een element op basis van de index. Het element wordt teruggegeven, zodat je het kunt opslaan in een variabele.

```
1 mijn_lijst = [1, 2, 3]
2 element = mijn_lijst.pop(1)  # pop element met index 1
3 print(element)  # 2
4 print(mijn_lijst)  # [1, 3]
```

Geef je geen index op, dan verwijder je het laatste element in de `list`.

#### del

Met het ingebouwde *keyword* `del` verwijder je één of meerdere elementen uit een `list`, op basis van een index of een *slice*.

```
1 mijn_lijst = [1, 2, 3, 4, 5]
2 del mijn_lijst[1]  # Verwijder element met index 1
3 print(mijn_lijst)  # [1, 3, 4, 5]
4 del mijn_lijst[1:3]  # Verwijder elementen met index 1 tot 3
5 print(mijn_lijst)  # [1, 5]
```

### Opdracht 5: in, uit

Neem de volgende `list`:

```
1 mijn_lijst = ["a", "b", "c", ]
```

Voer de volgende acties uit:

- Voeg elementen "d", "e" en "f" toe aan het einde van de `list`
- Verwijder het element "c" en voeg het aan het einde van de `list` toe
- Verwijder de elementen "d", "e" en "f"

Als alles goed is gegaan, heb je weer de oorspronkelijke `list`.

▼ *Klik om het antwoord te tonen*

```python
# Voeg de elementen "d", "e" en "f" toe
mijn_lijst.extend(["d", "e", "f"])
print(mijn_lijst)

# Verwijder het element "c" en plaats het terug aan het eind
c = mijn_lijst.pop(2)
mijn_lijst.append(c)
print(mijn_lijst)

# Verwijder nu de elementen "d", "e" en "f"
del mijn_lijst[2:5]
print(mijn_lijst)
```

[1] Itereren is het één voor één doorlopen van items in een collectie (zoals een list, tuple, dictionary)

# Werken met tuples

## Inleiding

In voorgaande hoofdstuk leerde je over de `list`. In dit hoofdstuk leer je over een vergelijkbare collectie: de `tuple`. Een `tuple` is zeer vergelijkbaar met een `list`, behalve dat een `tuple` niet aanpasbaar (**mutable**) is. Dit zorgt ervoor dat er bijvoorbeeld geen methodes zijn om elementen toe te voegen of te verwijderen.

Om herhaling te voorkomen, leer je in dit hoofdstuk met name hoe de `tuple` afwijkt van de `list`. Overeenkomstige handelingen (zoals `for-loops`) kun je eventueel teruglezen in eerdere hoofdstukken.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je wat een tuple is
- Begrijp je in welke situatie je tuples
- Begrijp je hoe je met elementen in een tuple kunt werken

## Tuples: een inleiding

Een `tuple` is een *onveranderlijke* collectie, in veel opzichten vergelijkbaar met een `list`. In deze paragraaf leer je wat de kenmerken van een `tuple` zijn en wanneer je het in kunt zetten.

### Kenmerken van tuples

Een `tuple` maak je op verschillende manieren:

```
1 lege_tuple_1 = ()
2 lege_tuple_2 = tuple()
3 tuple_met_items = ("a", "b", 1, 2, )
4 list_to_tuple = tuple([1, 2, 3, 4, ])
```

Een `tuple` bestaat altijd uit één of meerdere elementen geplaatst tussen haakjes (`()`), gescheiden door een komma. De elementen kunnen verschillende typen gegevens zijn, inclusief andere `tuples`, lijsten en woordenboeken.

Let op dat als je een `tuple` wilt maken met maar één element, je niet simpelweg het element tussen haakjes kunt plaatsen. Python zal die haakjes dan namelijk negeren. Om dit te omzeilen plaats je een komma achter het element:

```
1 enkel_item = ("a")
2 print(enkel_item)  # a
3
4 enkel_item = ("a", )
5 print(enkel_item)  # ("a", )
6
7 # Je kunt ook de haakjes weglaten
8 enkel_item = "a",  # Let op de komma
9 print(enkel_item)
```

De belangrijkste kenmerken van een `tuple` zijn:

- Een `tuple` is een type collectie en specifieker een **reeks**.
- Je haalt waarden op basis van de **index** op.
- Een `tuple` is *niet* muteerbaar.
- De elementen hoeven niet van hetzelfde **type** te zijn.

Omdat een `tuple` net als een `list` een reeks is, zijn alle acties die bij een `list` mogelijk zijn, ook mogelijk bij een `tuple`:

- Controleren of het element `i` in de reeks aanwezig is met `i in x`
- Ze samenvoegen met `x + y`
- Ze herhalen met `x * 3`
- Het aantal elementen ophalen met `len(x)`
- Het kleinste en grootste element ophalen met `min(x)` en `max(x)`
- Het aantal elementen tellen met `x.count()`
- De index van element `i` verkrijgen met `x.index(i)`

### Wanneer gebruik je tuples

Een `tuple` is erg vergelijkbaar met een `list`, behalve dat een `tuple` niet aanpasbaar is. Hierdoor is het minder geschikt om te gebruiken om een lijst met elementen op te bouwen, gebruik daarvoor een `list`.

Een `tuple` is vooral geschikt voor gegevens die tijdens het verloop van het programma éénmaal worden aangemaakt en vervolgens vaker worden gebruikt. Als je weet dat de reeks niet wijzigt tijdens het programma, is de `tuple` namelijk efficiënter in gebruik dan een `list`. Een voorbeeld is het definiëren van een reeks toegestane keuzes in een bepaald formulier:

```
1 OPTIONS = (
2     (1, "Altijd"),
3     (2, "Regelmatig"),
4     (3, "Soms"),
5     (4, "Zelden"),
6     (5, "Nooit"),
7 )
```

Zoals je verderop zult zien, is de `tuple` ook erg handig om een beperkte reeks gegevens in het programma door te geven.

## Werken met elementen in een tuple

Omdat werken met elementen in een `tuple` vrijwel hetzelfde is als werken met elementen in een `list`, lees je hier alleen de zaken die anders werken. Voor de rest lees je het best nogmaals het hoofdstuk over lijsten.

### Sorteren en omkeren

Omdat een `tuple` niet aanpasbaar is, heeft het geen eigen methodes om te sorteren en om te keren (`.sort()` en `.reverse()`). Wel is het mogelijk de ingebouwde functies `sorted` en `reversed` te gebruiken. Beiden leveren echter een nieuwe `list` op, en geen `tuple`. Uiteraard kun je met `tuple()` wel weer een `tuple` van de `list`

maken.

```
1 mijn_tuple = ("b", "c", "a", )
2 gesorteerd = tuple(sorted(mijn_tuple))
3
4 print(gesorteerd)
5 # ('a', 'b', 'c')
```

## Elementen toevoegen en verwijderen

Omdat een `tuple` niet aanpasbaar is, kun je geen elementen toevoegen of verwijderen.Wil je dit wel doen, dan dien je een nieuw object aan te maken.Bijvoorbeeld:

```
 1 mijn_tuple = (1, 2, 3, )
 2 print(id(mijn_tuple))  # Een bepaald ID
 3
 4 mijn_tuple = (1, 2, 3, 4)
 5 print(id(mijn_tuple))  # Een ander ID
 6
 7 # Of:
 8 mijn_tuple = (1, 2, 3, )
 9 nieuw_element = 4
10
11 mijn_tuple = mijn_tuple + (nieuw_element, )
12 print(mijn_tuple)
13 # (1, 2, 3, 4)
```

Let er in het laatste geval op dat de `+`-operator alleen gebruikt kan worden tussen twee `tuples`, daarom is van de variabele `nieuw_element` eerst een `tuple` gemaakt (op regel 11).

## Uitpakken

Een nog niet genoemd kenmerk van een `tuple` is dat de haakjes optioneel zijn:

```
1 tuple_1 = (1, 2, 3, 4, 5, )
2 tuple_2 = 1, 2, 3, 4, 5,
3
4 print(tuple_1)
5 # (1, 2, 3, 4, 5)
6
7 print(tuple_2)
8 # (1, 2, 3, 4, 5)
```

Dat de haakjes worden weggelaten zul je vaak zien bij een manier van werken die specifiek is voor `tuples`: **uitpakken** (*unpacking*).

Uitpakken is het toewijzen van de elementen uit een `tuple` aan variabelen, in één regel:

```python
a, b = (1, 2)
print(a)  # 1
print(b)  # 2
```

Op regel één zie je dat een `tuple` met twee elementen wordt toegewezen aan *twee* variabelen (gescheiden door een komma). Het eerste element zal worden toegewezen aan de eerste variabele (`a`) en het tweede element aan de tweede variabele (`b`). Dit is niet beperkt tot twee elementen/variabelen.

Omdat je de haakjes bij een `tupel` mag weglaten, levert onderstaande hetzelfde resultaat op:

```python
a, b = 1, 2
print(a)  # 1
print(b)  # 2
```

Dit uitpakken is met name interessant bij het gebruik van functies, waarover je meer leert in het hoofdstuk Werken met functies. Voor nu is het belangrijk om te begrijpen dat een functie een waarde *retourneert* (teruggeeft) met `return`. Datgene wat wordt geretourneerd door een functie, kun je toewijzen aan een variabele.

```python
def mijn_functie():
    # Deze functie geeft altijd "a" terug
    return "a"

# Wijs de uitkomst van de functie toe aan een variabele
a = mijn_functie()
print(a)  # a
```

Een functie kan echter ook meerdere waarden teruggeven, feitelijk een `tuple` zonder haakjes. En deze `tuple` kun je weer uitpakken:

```python
def mijn_functie():
    # Deze functie geeft altijd "a", "b" terug
    return "a", "b"

# Wijs de uitkomst van de functie met uitpakken toe aan twee variabelen
a, b = mijn_functie()
print(a)  # "a"
print(b)  # "b"
```

*Uitpakken*

Je kunt een langere `tuple` ook uitpakken en (bijvoorbeeld) alleen de eerste twee waarden gebruiken:

```python
mijn_tuple = (1, 2, 3, 4, 5)

a, b, *rest = mijn_tuple  # Let op de *
```

```
4
5 print(a)  # 1
6 print(b)  # 2
7 print(rest)  # [3, 4, 5]
```

In Python is het de gewoonte om een variabele die je verder niet meer gebruikt de naam _ te geven (een _underscore_):

```
1 a, b, *_ = mijn_tuple
```

### Opdracht 1: coördinaten

Neem onderstaande `tuple` met lengte- en breedtegraden:

```
1 coordinaten = (
2     (52.3702, 4.8952),
3     (51.9225, 4.47917),
4     (52.0907, 5.12142),
5     (50.8503, 4.3517),
6     (51.2093, 3.2247)
7 )
```

Van elk paar is het eerste getal de breedtegraad en het tweede getal de lengtegraad. Schrijf code dat voor elk paar de volgende zin print:

*"Breedtegraad: <getal>, Lengtegraad: <getal>"*

▼ *Klik om het antwoord te tonen*

```
1 # Uitwerking zonder uitpakken
2 for coord in coordinaten:
3     print(f"Breedtegraad: {coord[0]}, Lengtegraad: {coord[1]}")
4
5 # Uitwerking met uitpakken
6 for breedte, lengte in coordinaten:
7     print(f"Breedtegraad: {breedte}, Lengtegraad: {lengte}")
```

Net al een `list` kun je langs een `tuple` itereren. De eerste uitwerking is wellicht de meest intuïtieve: elke *loop* neem je het coördinatenpaar (een `tuple`) en haal je de elementen op met respectievelijk `coord[0]` en `coord[1]`.

De tweede oplossing maakt gebruik van *uitpakken*. In elke *loop* pak je de `tuple` uit in de variabelen `breedte` en `lengte`, zodat je deze direct in je `f-string` kunt gebruiken.

Overigens werkt uitpakken niet alleen bij `tuples`, maar bij alle *iterables*. Wel zie je het vaak bij functies die een *tuple* (zonder haakjes) teruggeven.

# Werken met sets

## Inleiding

In voorgaande hoofdstukken leerde je over de lists en de tuples. In dit hoofdstuk leer je over de `set`. Ook de `set` is een collectie, maar heeft een heel andere functie dan de `list` en de `tuple`.

Het belangrijkste kenmerk van een `set` is dat het enkel unieke elementen bevat. Verder gebruik je `sets` met name om twee groepen elementen met elkaar te vergelijken (welk element is onderdeel van beide groepen, bijvoorbeeld).

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je wat een set is
- Begrijp je in welke situatie je sets toe kunt passen
- Begrijp je hoe je met elementen in een set kunt werken

## Sets: een inleiding

In veel opzichten lijkt ook een `set` op een `list`. Het is een type collectie dat verschillende elementen kan bevatten, waar je langs kunt itereren en elementen aan kunt toevoegen of uit kunt halen.

Er zijn echter ook grote verschillen. In deze paragraaf leer je over de unieke eigenschappen van een `set` en wanneer je een `set` kunt gebruiken.

### Kenmerken van sets

Een `set` maak je op verschillende manieren:

```
1 lege_set = set()
2 set_met_items = {"a", "b", "c", }
3
4 mijn_list = [1, 2, 3, ]
5 mijn_set = set(mijn_list)
```

Een lege `set` maak je altijd met de `set()`-constructor. Omdat je een `set` met elementen aanmaakt met accolades, verwacht je misschien dat je een lege `set` ook met accolades aan kunt maken (`{}`), maar dat is al gereserveerd om een lege `dict` aan te maken!

Een `set` bestaat altijd uit één of meer elementen tussen accolades, gescheiden door een komma.

De belangrijkste kenmerken van een `set` zijn:

- Een `set` is een type collectie.
- Een `set` zelf is muteerbaar, maar de elementen in een `set` mogen dat *niet* zijn.
- Elementen in een `set` mogen van verschillende typen zijn (als ze maar niet muteerbaar zijn).
- Elementen in een `set` zijn altijd *uniek*.
- De volgorde van elementen in een `set` staat niet vast.

*Hashable*

Net als de bij de sleutels van een `dict`, gaat het er bij een `set` om dat de elementen *hashable* zijn. De meeste ingebouwde types in Python die muteerbaar zijn, zijn niet *hashable*. In de praktijk is het echter wel mogelijk om een datatype te hebben die veranderlijk is, maar toch *hashable*. Voor nu is het voldoende om er vanuit te gaan dat een element van een set onveranderlijk moet zijn.

Op Wikipedia lees je wat een *hash* is: nl.wikipedia.org/wiki/Hashfunctie

Omdat een `set` een collectie is, kun je er langs itereren. Verder zijn andere acties van collecties ook mogelijk:

- Controleren of het element `i` in de `set` aanwezig is met `i in mijn_set`.
- Het aantal elementen ophalen met `len(mijn_set)`.
- Het kleinste en grootste element ophalen met `min(mijn_set)` en `max(mijn_set)`.

Indexering werkt niet, door de ongeordende structuur (de volgorde van elementen staat niet vast).

*Geordend en ongeordend*

Hoe zit dat met *geordende* en *ongeordende* datatypen? Een `set` is *ongeordend*. De term doet vermoeden dat het iets met een gesorteerde volgorde te maken heeft, maar niets is minder waar. Dat een `set` ongeordend is, wil zeggen dat de volgorde van de elementen niet vast staat.

Stel je maakt een `set` aan met drie elementen, in de volgende volgorde: `{2, 3, 1}`.

Als de elementen nu in een `for-loop` gaat printen, kan het best zijn dat eerst de `3` wordt geprint, dan de `1` en dan de `2`.

Kortom: de volgorde van aanmaken zegt niets over de volgorde van opslaan. Vandaar dat indexering niet werkt. En zoals je verderop ziet, heeft ook sorteren en omkeren door dit gegeven geen zin.

Daarnaast zijn er nog een aantal acties die speciaal aan `sets` zijn voorbehouden, hierover lees je in de volgende paragraaf.

## Wanneer gebruik je sets

Een `set` gebruik je vrijwel altijd om twee redenen:

- Je wilt enkel met unieke waarden werken.
- Je wilt verschillende groepen elementen met elkaar vergelijken.

Om een voorbeeld bij dit laatste te geven, stel dat je twee `sets` met personen hebt, en je wilt weten welke persoon in beide `sets` aanwezig is:

```
1 set_a = {"Marie", "John", "Erwin", }
2 set_b = {"Ali", "John", }
3
4 in_beide = set_a.intersection(set_b)
5 print(in_beide)  # {"John"}
```

In de volgende paragraaf leer je meer van dergelijke berekeningen te maken.

# Werken met elementen in een set

Een `set` wijkt in meer opzichten af van een `list` dan een `tuple` doet. In deze paragraaf leer je hoe te werken met elementen in een `set`, waarbij weer de nadruk ligt op handelingen die anders werken dan bij een `list`.

## Elementen uit een set ophalen

De `set` is niet erg geschikt om met individuele elementen te werken. Een element uit een `set` verkrijgen is dan ook niet eenvoudig. Indexering werkt bijvoorbeeld niet, door de ongeordende structuur. Je kunt `.pop()` gebruiken om een willekeurig element uit de set te verkrijgen (en te verwijderen).

Wil je met één element uit een `set` werken, dan kun je het beste langs alle elementen itereren, tot je het gewenste element bereikt hebt.

## Sorteren en omkeren

Net als bij `tuples` heeft een `set` ook geen `.sort()` en `.reverse()` methodes. In dit geval niet omdat een `set` niet aanpasbaar is (want dat is het wel), maar omdat een `set` *ongeordend* is (zie kader eerder). Je kunt een `set` wel sorteren (met `sorted`), maar het zal de volgorde niet behouden. Wil je dit toch, dan dien je er (bijvoorbeeld) een `list` of een `tuple` van te maken.

## Elementen aan een set toevoegen

Om een enkel element aan een `set` toe te voegen gebruik je de `.add()` methode. Om meerdere elementen uit een *iterable* toe te voegen, gebruik je `.update()` (net als bij een `list`).

```
1 mijn_set = {1, 2, 3, }
2 mijn_set.add(4)
3
4 print(mijn_set)
5 # {1, 2, 3, 4}
6
7 mijn_list = [5, 6, 7, ]
8 mijn_set.update(mijn_list)
9
10 print(mijn_set)
11 # {1, 2, 3, 4, 5, 6, 7}
```

Let op dat in beide gevallen *geen* foutmelding wordt gegeven als je een element toevoegt dat al in de `set` aanwezig is.

## Elementen uit een set verwijderen

Om een element uit een `set` te verwijderen, gebruik je `.remove()` of `.discard()`. In beide gevallen geef je de waarde van het element op dat je wilt verwijderen. Het verschil tussen beiden is dat als het betreffende element niet bestaat, `.remove()` een foutmelding geeft en `.discard()` niet.

```
1 mijn_set = {1, 2, 3, 4, }
2 mijn_set.remove(1)
3
4 print(mijn_set)
```

```
# {2, 3, 4}

mijn_set.remove(5)  # Fout
mijn_set.discard(5)  # Geen fout
mijn_set.discard(2)

print(mijn_set)
# {3, 4}
```

Welke methode je gebruikt hangt af van de context. In veel gevallen wil je weten dat je een element probeert te verwijderen die er niet is en wil je dat afhandelen (hoe je dat doet, leer je in het hoofdstuk Werken met uitzonderingen). In die gevallen gebruik je `.remove()`. Het kan echter zijn dat het geen probleem is dat je programma zonder melding doorloopt als je een element wilt verwijderen dat niet bestaat. In dat geval kun je `.discard()` gebruiken.

Tot slot kun je - net als bij een `list` of `dict` - de methode `.pop()` gebruiken. De werking is wel anders.

- Bij een `list` verwijdert `.pop()` het laatste element. Je kunt het ook een index meegeven om het element op die index te verwijderen.
- Bij een `dict` dien je altijd een sleutel mee te geven aan `.pop()`. De waarde behorende bij die sleutel zal verwijderd worden.
- Bij een `set` kun je geen waarde aan `.pop()` meegeven. Het verwijdert een willekeurig element uit de `set`.

In alle gevallen wordt het verwijderde element ook teruggegeven.

**Opdracht 1: `.pop()`**

Leg in je eigen woorden uit waarom `.pop()` niet het laatste element van een `set` geeft zoals bij een `list`.

▼ *Klik om het antwoord te tonen*

Een `set` is een *ongeordende* datastructuur. Per definitie is er daarom niet zoiets als een laatste (of eerste) element. `.pop()` kan daarom ook niet een laatste element verwijderen en teruggeven, omdat het concept *laatste* in een `set` niet van toepassing is.

## Set vergelijkingen

Eén van de interessantere mogelijkheden van `sets` is het vergelijken van de elementen in verschillende `sets`. Uitgaande van `set_1` en `set_2`, kun je snel nagaan:

- Welke elementen in `set_1` OF in `set_2` (of beide) aanwezig zijn met `.union`.
- Welke elementen in `set_1` EN in `set_2` aanwezig zijn met `.intersection`.
- Welke elementen in `set_1` maar NIET in `set_2` aanwezig zijn met `.difference`.
- Welke elementen in `set_1` OF in `set_2`, maar NIET in beide aanwezig zijn met `.symmetric_difference`.

Om de set-vergelijkingen te illustreren maak je de volgende drie `sets` aan:

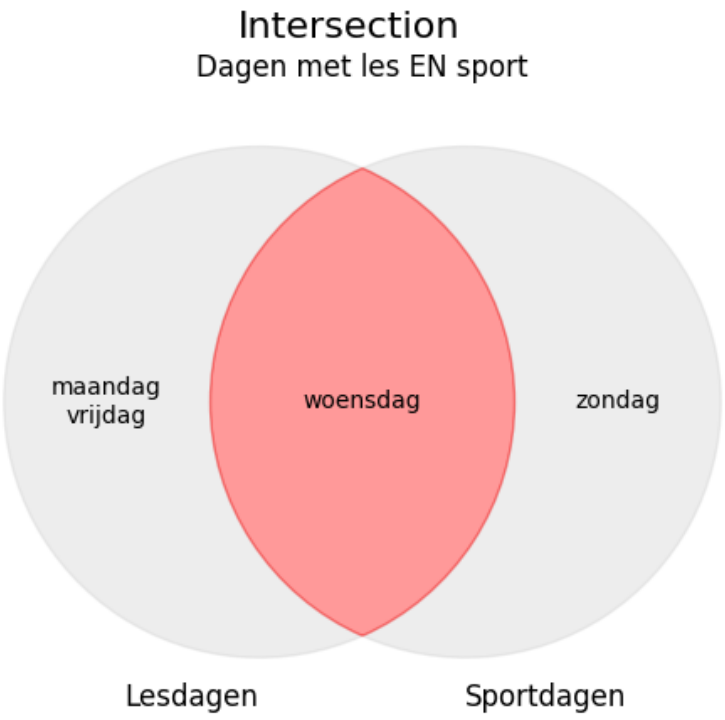
```
weekdagen = {
    "maandag",
    "dinsdag",
```

```
    "woensdag",
    "donderdag",
    "vrijdag",
    "zaterdag",
    "zondag"
}
lesdagen = {"maandag", "woensdag", "vrijdag"}
sportdagen = {"woensdag", "zondag"}
```

Allereerst zie je een `set` met alle dagen van de week. De andere twee `sets` zijn de dagen waarop je les hebt en waarop je sport.

*Venn*

Als je met `sets` werkt kan het handig zijn om één en ander te visualiseren. De meest gebruikte manier hiervoor is om met **Venn-diagrammen** te werken. Elke `set` wordt hierbij vertegenwoordigd door een cirkel, de cirkels kunnen elkaar deels overlappen. Bijvoorbeeld:




In elke cirkel noteer je de elementen, de elementen die in beide `sets` voorkomen noteer je in het overlappende deel.

Er zijn ook online tools die dit automatisch voor je doen, zie bijvoorbeeld: goodcalculators.com/venn-diagram-maker/

## Union

Om nu na te gaan op welke dagen je *iets* hebt (sport, les, of beide) gebruik je `.union`:

```
# Union: dagen met les, sport of beide
bezette_dagen = lesdagen.union(sportdagen)
```

```
print(bezette_dagen)
# {'maandag', 'vrijdag', 'zondag', 'woensdag'}
```


### Opdracht 2: Woensdag

Leg in je eigen woorden uit hoe het kan dat woensdag maar eenmaal in het resultaat voorkomt, terwijl je op woensdag sport én lessen volgt.

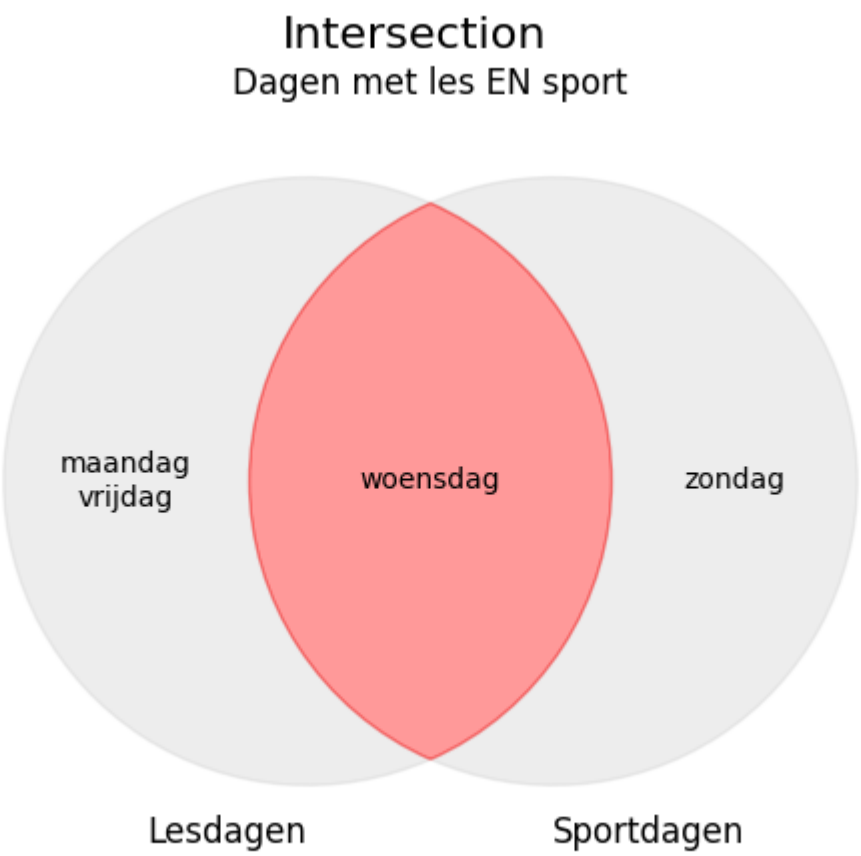
▼ *Klik om het antwoord te tonen*

Een `set` kan geen dubbele waarden bevatten. Dus alhoewel woensdag tweemaal voorkomt, wordt deze waarde ontdubbelt.

## Intersection

Om na te gaan wanneer je zowel sport als les hebt, gebruik je `.intersection`:

```
# Intersection: dagen met les én sport
drukke_dagen = lesdagen.intersection(sportdagen)
print(drukke_dagen)
# {'woensdag'}
```




### Opdracht 3: Omdraaien

Wat is het resultaat van `drukke_dagen = sportdagen.intersection(lesdagen)` (lesdagen en sportdagen zijn omgedraaid)? Leg uit hoe dat kan.

▼ *Klik om het antwoord te tonen*

Het resultaat is hetzelfde, namelijk `{'woensdag'}`. Omdat je met `.intersection` controleert welke elementen in zowel sportdagen als lesdagen voorkomen, maakt de volgorde niet uit. Dit werkt overigens net zo bij `.union`, en `symmetric_difference`. Een dergelijke bewerking - dat de volgorde niet uitmaakt - noem je *commutatief*.

## Difference

Om na te gaan op welke dagen je vrij hebt (dus geen sport en geen les), ga je het verschil na tussen de `weekdagen` en de `bezette_dagen`:

```
1 # Difference: dagen in weekdagen, maar niet in bezette dagen
2 vrije_dagen = weekdagen.difference(bezette_dagen)
3 print(vrije_dagen)
4 # {'dinsdag', 'donderdag', 'zaterdag'}
```

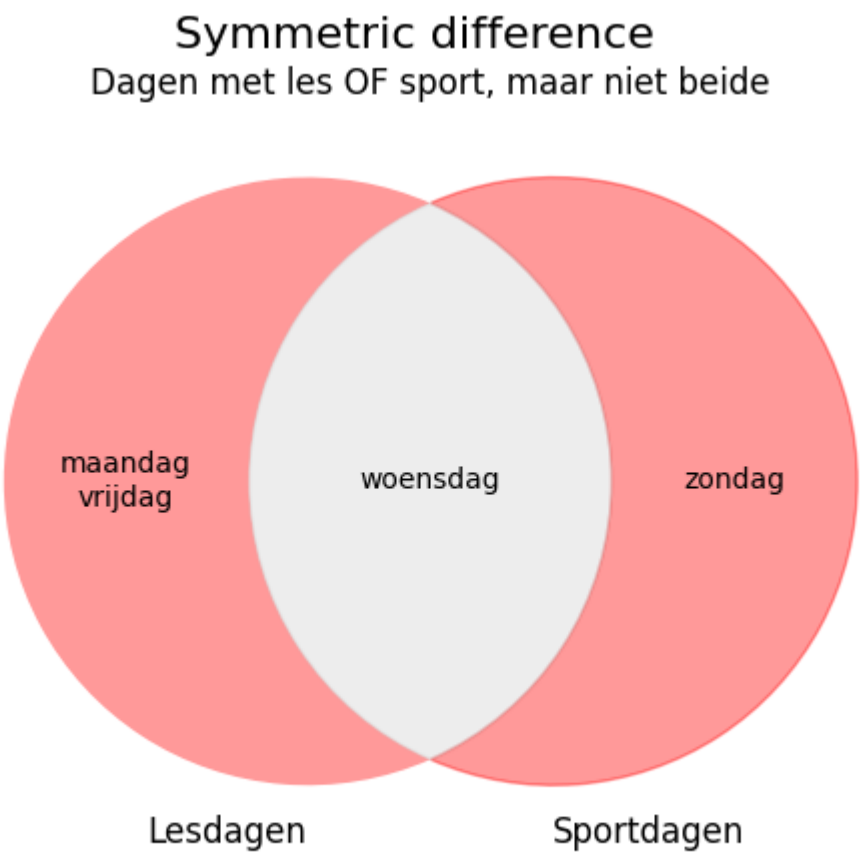

Je bekijkt zo de weekdagen die *niet* in `bezette_dagen` aanwezig zijn.

Als je het omdraait, dan bekijk je de bezette dagen die niet in `weekdagen` aanwezig zijn. Dat levert een lege `set` op, omdat alle dagen in `bezette_dagen` ook in `weekdagen` aanwezig is.

### Symmetric Difference

Tot slot kun je nog nagaan op welke dag je maar één iets hebt (of les, of sport, maar niet beide):

```
1 # Symmetric Difference: dagen in lesdagen of sportdagen, maar niet in beide
2 een_ding = lesdagen.symmetric_difference(sportdagen)
3 print(een_ding)
4 # {'maandag', 'zondag', 'vrijdag'}
```




### Opdracht 4: Alternatief

Hoe kun je met een set-vergelijking nog meer nagaan op welke dagen je maar één activiteit hebt?

▼ *Klik om het antwoord te tonen*

Dit kan als volgt: `een_activiteit = bezette_dagen.difference(drukke_dagen)`. Op `bezette_dagen` heb je één of twee activiteiten, op `drukke_dagen` heb je twee activiteiten. De dagen die in `bezette_dagen` voorkomen, maar niet in `drukke_dagen`, zijn dus de dagen met één activiteit.

## Subset, superset, disjoint

Daarnaast zijn er methodes om na te gaan of `set_1`:

- Een subset is van `set_2` (alle elementen van `set_1` zijn ook aanwezig in `set_2`).
- Een superset is van `set_2` (alle elementen van `set_2` zijn ook aanwezig in `set_1`).
- Er geen enkele overlap is (geen enkel element van `set_1` bevindt zich in `set_2`).

Neem de volgende, iets aangepaste `sets`:

```
1 weekdagen = {
2     "maandag",
3     "dinsdag",
4     "woensdag",
```

```
    "donderdag",
    "vrijdag",
}
weekend = {"zaterdag", "zondag",}
lesdagen = {"maandag", "woensdag", "vrijdag"}
sportdagen = {"woensdag", "zondag"}
```

Je controleert eenvoudig of alle lesdagen onderdeel zijn van weekdagen met `.issubset`. Hetzelfde doe je voor de sportdagen:

```
lessen_in_week = lesdagen.issubset(weekdagen)
print(lessen_in_week)  # True

sport_in_week = sportdagen.issubset(weekdagen)
print(sport_in_week)  # False
```

Andersom kun je controleren of alle dagen van de week de lesdagen omvatten:

```
week_lessen = weekdagen.issuperset(lesdagen)
print(week_lessen)  # True
```

En tot slot kun je nagaan of twee `sets` totaal geen overlap hebben, zoals doordeweeks en het weekend:

```
geen_overlap = weekdagen.isdisjoint(weekend)
print(geen_overlap)  # True
```

*Operatoren*

In plaats van bijvoorbeeld `set_a.union(set_b)` kun je ook operatoren gebruiken. Voor `union` gebruik je `|`: `set_a | set_b`.

- union: `|`
- intersection: `&`
- difference: `-`
- symmetric_difference: `^`

Zie docs.python.org/3.11/library/stdtypes.html#set voor alle operatoren en hun gebruik.

# Werken met dicts

## Inleiding

Woordenboeken zijn een veelgebruikte type collectie binnen Python. In Dictionaries zag je ze al even voorbij komen, in dit hoofdstuk leer je er uitgebreider mee werken.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je wat een woordenboek is
- Begrijp je in welke situatie je woordenboeken toe kunt passen
- Begrijp je hoe je met elementen in een woordenboek kunt werken

## Woordenboeken: een inleiding

In een (echt) woordenboek zoek je een bepaald woord op om daar de definitie of vertaling van te leren. In Python werken woordenboeken net zo. Een woordenboek, *dictionary* in het Engels en in Python afgekort tot `dict`, is een type collectie waarbij je elementen niet opzoekt op basis van **index** (zoals bij lijsten), maar op basis van een **sleutel**. Bijvoorbeeld:

```
1 nederlands_engels = {
2     "programmeertaal": "programming language",
3     "opleiding": "course",
4     "leraar": "teacher",
5 }
```

In dit woordenboek zie je drie elementen, elk bestaande uit een sleutel (het Nederlandse woord) en de waarde (de Engelse vertaling). Dergelijke elementen worden vaak *key-value pairs* genoemd. De sleutels en waarden bestaan in dit voorbeeld uit tekst, maar een `dict` is hier zeker niet toe beperkt.

In deze paragraaf leer je wat de kenmerken van woordenboeken zijn en in welke situaties je ze kunt gebruiken.

### Kenmerken van woordenboeken

Een `dict` kun je op verschillende manieren maken:

```
1 lege_dict_1 = {}
2 lege_dict_2 = dict()
3 dict_met_items = {
4     1: "a",
5     2: "b",
6 }
```

Een *key-value pair* bestaat altijd uit een sleutel en een waarde, gescheiden door een dubbele punt (`:`). Meerdere *key-value pairs* in een `dict` scheidt je met een komma. Net als bij een `list` kun je achter het laatste element een komma plaatsen, maar dit is niet verplicht.

Je kunt ook een `dict` maken van een `list` (of `tuple`) van *key-value pairs*. In onderstaand voorbeeld bevat

`mijn_lijst` drie `tuples`, welke allemaal een *key-value pair* in de `dict` zullen worden.

```
mijn_lijst = [
    ("a", 1),
    ("b", 2),
    ("c", 3),
]

mijn_dict = dict(mijn_lijst)
print(mijn_dict)
# {'a': 1, 'b': 2, 'c': 3}
```

De belangrijkste kenmerken van een `dict` zijn:

- Een `dict` is een type collectie.
- Je haalt waarden op basis van sleutels op uit een `dict`.
- Een `dict` is **muteerbaar** (aanpasbaar). Je kunt elementen toevoegen, aanpassen en verwijderen.
- De elementen hoeven niet van hetzelfde type te zijn. Dit geldt voor zowel de sleutels als de waarden.

Een aantal handelingen die je bij lijsten tegenkwam zijn ook bij een `dict` mogelijk, maar niet alle. Je kunt:

- Controleren of het element `i` in de `dict` aanwezig is met `i in mijn_dict`.
- Het aantal elementen ophalen met `len(mijn_dict)`.
- Het kleinste en grootste element ophalen met `min(mijn_dict)` en `max(mijn_dict)`.

Verder zijn er nog specifieke kenmerken voor de sleutels en de waarden.

#### Sleutels

De sleutels in een `dict` moeten *uniek* te zijn. Het volgende heeft dust geen zin:

```
dubbel = {
    "a": 1,
    "a": 2,
}
```

Zoals je verderop zult zien, haal je waarden uit een `dict` op basis van de sleutel op. Met dubbele sleutels weet Python niet welke waarde het terug moet geven. Je kunt de `dict` wel op deze manier aanmaken, maar enkel de laatste `a` zal opgeslagen worden.

Daarnaast moeten sleutels een onveranderlijk (*immutable*) type zijn. Vaak zul je tekst of integers als sleutel gebruiken, maar `floats` en `tuples` zijn ook toegestaan.

Voor nu is het voldoende om over sleutels te denken als uniek en onveranderlijk. Maar feitelijk ligt het genuanceerder, het gaat er namelijk om dat sleutels *hashable* moeten zijn. In de praktijk betekent dit dat een veranderlijke type wel als sleutel gebruikt kan worden, als het maar elke keer dezelfde *hash* oplevert.

Een *hash* functie is een functie dat data van arbitraire grootte omzet naar data met een vaste grootte.

> De *hash* van het woord "Python" is:
> 18885f27b5af9012df19e496460f9294d5ab76128824c6f993787004f6d9a7db
>
> De *hash* van de zin "Python is super" is:
> 33b9132f634040992a01e4c77a462813cc624868c428c7cbfdb89c7a3d4ff397

**Waarden**

Waarden hoeven niet uniek te zijn binnen een `dict` en mogen ook veranderlijk zijn. Het is dus mogelijk om bijvoorbeeld een `list` of een andere `dict` als waarde te hebben:

```
werknemers = {
    "Ali": {
        "leeftijd": 30,
        "rol": "Manager",
        "salaris": 5000
    },
    "Marie": {
        "leeftijd": 25,
        "rol": "Ontwikkelaar",
        "salaris": 4000
    },
    "John": {
        "leeftijd": 35,
        "rol": "Analist",
        "salaris": 4500
    }
}
```

Hier zie je een `dict` met als sleutels de naam van de werknemers, en als waarden een nieuwe `dict`. Bijvoorbeeld: `Ali` is een sleutel, met de `dict` `{"leeftijd: 30, ⋯}` als waarde. Dit gegeven, dat de waardes van een `dict` zelf ook weer bestaan uit een `dict`, heet **nesting**. Dit is overigens niet beperkt tot één laag:

```
nested_dict = {
    "fruit": {
        "appel": {
            "kleur": "rood",
            "smaak": "zoet"
        },
        "banaan": {
            "kleur": "geel",
            "smaak": "zoet"
        }
    },
    "groente": {
        "wortel": {
            "kleur": "oranje",
            "smaak": "hartig"
        },
```

```
        "sla": {
            "kleur": "groen",
            "smaak": "knapperig"
        }
    }
}
```

Zolang je laptop of computer genoeg geheugen heeft, is er geen beperking aan het aantal lagen dat je kunt *nesten*.

De waarden van een `dict` kunnen van elk type zijn.

Let op de leesbaarheid van `dicts` in het algemeen en bij *geneste* `dicts` in het bijzonder. Gebruik inspringingen om duidelijk te maken tot welk niveau elke laag hoort.

### Wanneer woordenboeken te gebruiken

Een `dict` is erg geschikt om te werken met data dat verschillende eigenschappen heeft. In Wanneer gebruik je een lijst? zag je het voorbeeld van een `list` voorbij komen met een aantal namen. Dit zou je nu kunnen uitbreiden naar een `list` met `dicts`. Bijvoorbeeld:

```
# Als lijst, alleen de naam
gebruikers = ["John", "Ali", "Marie", ]

# Als dict, met meer informatie
gebruikers = [
    {"naam": "John", "leeftijd": 30},
    {"naam": "Ali", "leeftijd": 30},
    {"naam": "Marie", "leeftijd": 25},
]
```

## Werken met elementen in een woordenboek

Zoals inmiddels duidelijk is, bestaat een `dict` uit *key-value pairs*. Dit feit bepaalt de werking van hoe je met elementen in een `dict` werkt. Je zult namelijk vaak de keuze moeten maken of je wilt werken met de sleutels, de waarden of beiden.

Een `dict` heeft dan ook drie belangrijke methodes die je veel zult tegenkomen:

- `.keys()`
- `.values()`
- `.items()`

Een voorbeeld maakt duidelijk wat elke methode doet:

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
```

```
}

print(mijn_dict.keys())
# dict_keys(['naam', 'beroep', 'leeftijd'])

print(mijn_dict.values())
# dict_values(['Erwin', 'Auteur', 38])

print(mijn_dict.items())
# dict_items([('naam', 'Erwin'), ('beroep', 'Auteur'), ('leeftijd', 38)])
```

Elke methode levert een **iterable** op, je kunt het zien als een speciaal soort lijst. Je kunt er dus ook langs itereren:

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

for key in mijn_dict.keys():
    print(key)

# naam
# beroep
# leeftijd
```

De `.items()` methode levert een lijst op waarbij elk element een `tuple` is van de sleutel en bijbehorende waarde, zodat je met beiden kunt werken. Een voorbeeld hiervan zie je in Itereren langs een woordenboek als je leert een `for-loop` te gebruiken met een `dict`.

De `.keys()` methode zul je in de praktijk zelden zien, omdat dit als de standaard wordt beschouwd. Laat je `.keys()` weg, dan zul je de acties uitvoeren op de sleutels van de `dict`. Bovenstaand voorbeeld kun je dus als volgt herschrijven:

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

for key in mijn_dict:  # Let op, geen .keys()!
    print(key)

# naam
# beroep
# leeftijd
```

## Kenmerken van een `dict` ophalen

Met `min()`, `max()` en `len()` haal je enkele kenmerken op van een `dict`. Met `len()` haal je het aantal elementen op. Voor `len()` maakt het niet uit of je dit uitvoert op `mijn_dict.keys()`, `mijn_dict.values()` of `mijn_dict.items()`, het resultaat is logischerwijs altijd hetzelfde.

Met `min()` en `max()` haal je de kleinste en grootste sleutel of waarde op:

```
1 mijn_dict = {
2     "a": 100,
3     "b": 10,
4     "c": 1,
5 }
6
7 print(len(mijn_dict))  # 3
8
9 print(min(mijn_dict))  # a
10 print(max(mijn_dict))  # c
11
12 print(min(mijn_dict.values()))  # 1
13 print(max(mijn_dict.values()))  # 100
```

### Opdracht 1: min / max

Neem onderstaande `dict`. Wat is het resultaat van regel 7? Wat verwacht je dat er gebeurt als je de laatste twee regels uitvoert?

```
1 mijn_dict = {
2     "a": 100,
3     "b": 10,
4     "c": 1,
5 }
6
7 print(max(mijn_dict.items()))
8
9 mijn_dict["a"] = "honderd"
10 print(max(mijn_dict.values()))
```

▼ *Klik om het antwoord te tonen*

Het resultaat van regel 7 is een `tuple`: `('c', 1)`. `max()` kijkt standaard naar het eerste element in de `tuple`, en van `a', 'b' en `c`, is `c` het grootst.

Voer je de laatste twee regels uit dan zul je een foutmelding krijgen. Python weet namelijk niet hoe het integers met tekst moet vergelijken.

## Elementen uit een `dict` ophalen

Je haalt de *waarden* uit een `dict` op op basis van de *sleutel*. Dit kun je op twee manieren doen. De eerste lijkt op

hoe je een element uit een `list` ophaalt:

```
1 mijn_dict = {
2     "naam": "Erwin",
3     "beroep": "Auteur",
4     "leeftijd": 38,
5 }
6
7 naam = mijn_dict["naam"]
8 print(naam)  # Erwin
```

Net als bij lijsten gebruik je haakjes (`[]`), maar in plaats van een index gebruik je de sleutel, `naam` in dit geval. Als de sleutel niet bestaat, krijg je een fout. Probeer maar eens om `naam` te veranderen in `Naam`, en kijk wat er gebeurt.

Om dit te voorkomen is er nog een andere manier om waarden op te halen, met de `.get()` methode:

```
 1 mijn_dict = {
 2     "naam": "Erwin",
 3     "beroep": "Auteur",
 4     "leeftijd": 38,
 5 }
 6
 7 naam = mijn_dict.get("naam")
 8 print(naam)  # Erwin
 9
10 hobby = mijn_dict.get("hobby")
11 print(hobby)  # None
12
13 default = mijn_dict.get("hobby", "Onbekend")
14 print(default)  # Onbekend
```

In plaats van de haakjes te gebruiken, gebruik je de `.get()` methode. Bestaat de sleutel niet, dan zal `.get()` standaard `None` teruggeven. Dit kun je echter aanpassen door een standaardwaarde op te geven, zoals op regel 9.

De `.get()` methode is handig om in `if`-statements te gebruiken:

```
1 mijn_dict = {
2     "naam": "Erwin",
3     "beroep": "Auteur",
4     "leeftijd": 38,
5 }
6
7 hobby = mijn_dict.get("hobby")
8 if not hobby:
9     print("Bezoek onze website voor inspiratie!")
```

Eerder zag je dat een `dict` meerdere lagen diep kan zijn. Het ophalen van waarden uit de diepere lagen werkt hetzelfde als het ophalen van de eerste laag:

```
werknemers = {
    "Ali": {
        "leeftijd": 30,
        "rol": "Manager",
        "salaris": 5000
    },
    "Marie": {
        "leeftijd": 25,
        "rol": "Ontwikkelaar",
        "salaris": 4000
    },
    "John": {
        "leeftijd": 35,
        "rol": "Analist",
        "salaris": 4500
    }
}

rol_ali = werknemers["Ali"]["rol"]
print(rol_ali)  # Manager
```

Je verwacht misschien dat je dit ook kunt doen met meerdere aanroepen van `.get()`. En dit werkt ook, maar levert alsnog fouten op als de eerste sleutel niet bestaat:

```
werknemers = {
    "Ali": {
        "leeftijd": 30,
        "rol": "Manager",
        "salaris": 5000
    }
}

rol_erwin = werknemers.get("Erwin").get("rol")
# Sleutel 'Erwin' bestaat niet
# Fout: AttributeError
```

Omdat de sleutel `Erwin` niet bestaat, wordt er `None` teruggegeven. En `None` heeft geen methode `.get()`, waardoor een fout ontstaat. Er zijn manieren om dit op te lossen, één hiervan zul je leren in Werken met uitzonderingen, waar je leert hoe je fouten af kunt handelen.

### Opdracht 2: get it?

Schrijf code dat onderstaande `dict` neemt, en de naam van de werknemer en zijn salaris print. Als de werknemer niet bestaat, print dan "Werknemer niet gevonden." Als het salaris niet wordt gevonden, print dan "Het salaris van <werknemer> is niet bekend".

```python
werknemers = {
    "Ali": {
        "leeftijd": 30,
        "rol": "Manager",
        "salaris": 5000
    },
    "Marie": {
        "leeftijd": 25,
        "rol": "Ontwikkelaar",
        "salaris": 4000
    },
    "John": {
        "leeftijd": 35,
        "rol": "Analist",
    }
}
```

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is:

```python
te_zoeken = "Ali"
werknemer = werknemers.get(te_zoeken)
if werknemer:
    salaris = werknemer.get("salaris")
    if salaris:
        print(f"Het salaris van {te_zoeken} is {salaris}.")
    else:
        print(f"Het salaris van {te_zoeken} is niet bekend.")
else:
    print("Werknemer niet gevonden.")

# Een alternatieve manier, iets korter:
te_zoeken = "Ali"
werknemer = werknemers.get(te_zoeken)
if werknemer and werknemer.get("salaris"):
    print(f"Het salaris van {te_zoeken} is {werknemer.get('salaris')}.")
elif werknemer:
    print(f"Het salaris van {te_zoeken} is niet bekend.")
else:
    print("Werknemer niet gevonden.")
```

## Controleren of een element zich in een woordenboek bevindt

Net als bij een `list`, kun je `in` (en `not in`) gebruiken om te controleren of een element zich in een `dict` bevindt. Hierbij gaat het echter niet om de *key-value pair*, maar controleer je op de sleutel óf de waarde.

Gebruik je `in`, dan controleer je standaard op de sleutels:

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

print("naam" in mijn_dict)  # True, "naam" is een sleutel
print("Erwin" in mijn_dict)  # False, "Erwin" is géén sleutel
```

Om te controleren of een *waarde* zich in een `dict` bevindt, voeg je `.values()` toe aan de `dict`:

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

print("naam" in mijn_dict.values())  # False, "naam" is geen waarde
print("Erwin" in mijn_dict.values())  # True, "Erwin" is een waarde
```

### Itereren langs een woordenboek

Omdat een `dict` een collectie is, kun je `for` gebruiken om langs de `dict` te itereren. Bekijk wat er gebeurt als je dit doet:

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

for element in mijn_dict:
    print(element)

# naam
# beroep
# leeftijd
```

Je ziet dat elke *sleutel* wordt afgedrukt. Net als bij `in`, kun je `.values()` gebruiken om de *waarden* op te halen:

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

```

```python
for element in mijn_dict.values():
    print(element)

# Erwin
# Auteur
# 38
```

Als je `for` gebruikt zonder `.values()`, itereer je over de sleutels. Je kunt dan ook dit schrijven:

```python
for element in mijn_dict.keys():
    print(element)
```

Maar je mag `.keys()` dus ook weglaten. Hetzelfde geldt bij `in` en andere acties. Het uitgangspunt is dat je acties uitvoert op de sleutels, daarom mag je `.keys()` weglaten.

In de meeste gevallen zul je echter willen itereren langs de sleutels én de waarden. Gebruik hiervoor `.items()`:

```python
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

for key, value in mijn_dict.items():
    print(f"{key}: {value}")

# naam: Erwin
# beroep: Auteur
# leeftijd: 38
```

Je ziet dat je ook twee variabelen in de `for-loop` gebruikt. In het voorbeeld heten ze `key` en `value`, en dit zul je vaak zien. Maar je mag hier kiezen wat je wilt, bijvoorbeeld `kenmerk` en `waarde`.

### Opdracht 3: Werknemers

Neem weer onderstaande `dict` met werknemersgegevens. Schrijf code dat voor elke werknemer alle gegevens per werknemer print.

```python
# Gegevens
werknemers = {
    "Ali": {
        "leeftijd": 30,
        "rol": "Manager",
        "salaris": 5000
    },
    "Marie": {
```

```
        "leeftijd": 25,
        "rol": "Ontwikkelaar",
        "salaris": 4000
    },
    "John": {
        "leeftijd": 35,
        "rol": "Analist",
        "salaris": 4500
    }
}

# Gewenst resultaat
# Ali
# ---
# Leeftijd: 30
# Rol: Manager
# Salaris: 5000
#
# Marie
# -----
# Leeftijd: 25
# Rol: Ontwikkelaar
# Salaris: 4000
#
# John
# ----
# Leeftijd: 35
# Rol: Analist
# Salaris: 4500
```

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is:

```
for werknemer, gegevens in werknemers.items():
    print(f"{werknemer}")
    print("-"*len(werknemer))
    print(f"Leeftijd: {gegevens.get('leeftijd')}")
    print(f"Rol: {gegevens.get('rol')}")
    print(f"Salaris: {gegevens.get('salaris')}")
    print()
```

## Elementen wijzigen of toevoegen

Tot dusver heb je steeds een `dict` gemaakt met een aantal *key-value pairs*. Een dict is echter aanpasbaar, en het is dus ook mogelijk om later elementen toe te voegen.

De meest eenvoudige manier is als volgt:

```python
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

mijn_dict["hobby"] = "Schrijven"
print(mijn_dict["hobby"])  # Schrijven
```

Je voegt tussen haakjes de sleutel toe en wijst vervolgens de waarde toe aan die sleutel.

Gebruik je een bestaande sleutel, dan zal de oude waarde overschreven worden:

```python
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
    "hobby": "Schrijven",
}

mijn_dict["hobby"] = "Lezen"
print(mijn_dict["hobby"])  # Lezen
```

Een tweede manier om elementen aan een `dict` toe te voegen is met de `.update()` methode. Hiermee voeg je echter niet één *key-value pair* toe, maar voeg je twee `dicts` samen, of voeg je een andere *iterable* met *key-value pairs* (zoals een `list`) toe aan de `dict`.

```python
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

tweede_dict = {
    "hobby": "Schrijven",
    "taal": "Python",
}

mijn_dict.update(tweede_dict)
print(mijn_dict)
# {
#   'naam': 'Erwin',
#   'beroep': 'Auteur',
#   'leeftijd': 38,
#   'hobby': 'Schrijven',
#   'taal': 'Python'
# }
```

```
21
22 mijn_list = [("lengte", 1.78), ("favoriet_dier", "Cobra"), ]
23 mijn_dict.update(mijn_list)
24 print(mijn_dict)
25 # {
26 #   'beroep': 'Auteur',
27 #   'favoriet_dier': 'Cobra',
28 #   'hobby': 'Schrijven',
29 #   'leeftijd': 38,
30 #   'lengte': 1.78,
31 #   'naam': 'Erwin',
32 #   'taal': 'Python'
33 # }
```

Als je `dict` steeds uitgebreider wordt, zal het steeds minder leesbaar worden als je simpelweg `print()` gebruikt. Alles zal namelijk op één regel worden geprint.

Je kunt hiervoor `pprint` gebruiken. Voeg bovenaan je bestand het volgende toe:

```
from pprint import pprint
```

Druk nu de `dict` af met: `pprint(mijn_dict)` en zie het verschil.

In een later hoofdstuk leer je meer over `imports`.

### Opdracht 4: Geneste toevoeging

Neem onderstaande `dict` en voeg de volgende zaken toe:

- Een sleutel `kinderen`, met als waarde een `list` met twee kinderen.
- Elk kind heeft weer de volgende gegevens:
  - Leeftijd
  - Naam
- Een sleutel `moeder`, met dezelfde gegevens als `vader`.

```
1 gezin = {
2     "vader": {
3         "naam": "John",
4         "leeftijd": 38,
5     }
6 }
```

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is:

```
1 from pprint import pprint
2
```

```
kinderen = [
    {"naam": "Lotte", "leeftijd": 3},
    {"naam": "Jaap", "leeftijd": 6},
]

moeder = {"naam": "Liesbeth", "leeftijd": 39}

gezin["kinderen"]= kinderen
gezin["moeder"] = moeder

pprint(gezin)
```

In de alternatieve oplossing werkt `if werknemer and werknemer.get("salaris"):` omdat als `werknemer` niet bestaat, de rest van de `if`-statement niet meer geëvalueerd wordt.

### Elementen verwijderen

Net als bij een `list`, kun je bij een `dict` ook `del` gebruiken om elementen te verwijderen.

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

del mijn_dict["leeftijd"]
print(mijn_dict)
# {'naam': 'Erwin', 'beroep': 'Auteur'}
```

Ook `.pop()` kun je bij een `dict` gebruiken, en net als bij een `list` verwijdert je hiermee het element én geef je het terug:

```
mijn_dict = {
    "naam": "Erwin",
    "beroep": "Auteur",
    "leeftijd": 38,
}

naam = mijn_dict.pop("naam")
print(naam)  # Erwin
print(mijn_dict)
# {'beroep': 'Auteur', 'leeftijd': 38}
```

In tegenstelling tot `.pop()` bij een `list`, geef je de sleutel op, niet een index. Gebruik je `.pop()` zonder index bij een `list`, dan zal het het laatste element teruggeven. Bij een `dict` moet je `.pop()` altijd een waarde meegeven.

## Elementen sorteren en omkeren

Een `dict` kun je sorteren en omkeren, net als je in het vorige hoofdstuk leerde bij lijsten. In tegenstelling tot bij een `list`, heeft `dict` geen eigen methodes `.sort()` en `.reverse()`. Je zult dus de ingebouwde functies `sorted()` en `reversed()` moeten gebruiken.

In Python hebben `dicts` pas vanaf versie 3.7 een vaste volgorde. Dit betekent dat de volgorde van de elementen bij het aanmaken van de `dict` ook de volgorde is als je er met een `for-loop` overheen itereert of als je het print.

Dit had ook gevolgen voor het sorteren: je kon wel sorteren en het opslaan in een (nieuwe) `dict`, maar omdat de volgorde niet behouden bleef, kon je de sortering ook weer verliezen.

Loop je tegen problemen met sorteren en omkeren aan, controleer dan of je Python versie 3.7 of hoger is. Typ `python --version` in je shell om de versie te verkrijgen.

### Sorteren

Je kunt een `dict` sorteren op basis van zowel de sleutels als de waarden. Sorteren op waarden is iets complexer en laten we hier buiten beschouwing. Om te sorteren op basis van de sleutel gebruik je de ingebouwde functie `sorted()`. Echter, `sorted()` geeft altijd een `list` terug:

```
1 personen = {
2     "Marie": 36,
3     "Ali": 35,
4     "John": 38,
5 }
6
7 alfabetisch = sorted(personen)
8 # ["Ali", "John", "Marie"]
```

Dit kan nuttig zijn, maar hiermee is de `dict` zelf niet gesorteerd. Om dit te bereiken gebruik je de `.items()` methode, die je ook hebt gebruikt in de `for-loop`. Met `.items()` maak je een *iterable* van *key-value pairs*:

```
1 personen = {
2     "Marie": 36,
3     "Ali": 35,
4     "John": 38,
5 }
6
7 print(personen.items())
8 # dict_items([('Marie', 36), ('Ali', 35), ('John', 38)])
```

Omdat `sorted()` met *iterables* werkt en `.items()` een *iterable* oplevert, kun je beiden combineren om te sorteren:

```
1 personen = {
2     "Marie": 36,
3     "Ali": 35,
4     "John": 38,
```

```
}

alfabetisch = sorted(personen.items())
print(alfabetisch)
# [('Ali', 35), ('John', 38), ('Marie', 36)]

alfabetisch = dict(alfabetisch)
print(alfabetisch)
# {'Ali': 35, 'John': 38, 'Marie': 36}
```

Omdat `sorted()` een `list` oplevert, maak je er met `dict()` weer een `dict` van. Dit kan overigens ook in één regel:

```
alfabetisch = dict(sorted(personen.items()))
```

#### Omkeren

Een `dict` keer je om met `reversed()` en werkt vergelijkbaar als `sorted()`:

```
personen = {
    "Marie": 36,
    "Ali": 35,
    "John": 38,
}

omgekeerd = dict(reversed(personen.items()))
print(omgekeerd)
# {'John': 38, 'Ali': 35, 'Marie': 36}
```

# Werken met functies

## Inleiding

Tot dusver heb je de code simpelweg in één bestand getypt en dat bestand direct uitgevoerd (vanuit Thonny). Voor kleine stukjes code werkt dit prima, maar zodra je uitgebreidere programma's gaat schrijven wordt dit al snel onoverzichtelijk. Ook gebruik je dezelfde code vaak meermaals, die code op twee of meer plaatsen herhalen is inefficiënt en foutgevoelig.

In dit hoofdstuk leer je daarom eerst over *functies*. Een functie is een herbruikbaar stuk code, idealiter met één specifieke taak. Deze functie kun je vervolgens vaker aanroepen. Bijvoorbeeld:

```
1 namen = ["Ali", "Marie", "John", ]
2
3
4 # Definieer de functie `groet`
5 def groet(naam):
6     print(f"Hallo, {naam}!")
7
8
9 # Roep `groet` aan voor elke naam in `namen`
10 for naam in namen:
11     groet(naam=naam)
12
13 # Hallo, Ali!
14 # Hallo, Marie!
15 # Hallo, John!
```

Nu is dit een eenvoudig voorbeeld en is de winst die je behaalt ten opzichte van drie keer een `print`-statement uitschrijven wellicht niet zo groot. Maar stel je voor dat het een complexe berekening betreft, en de lijst bestaat niet uit drie elementen maar uit 100… Dan ervaar je al snel de voordelen van een herbruikbare functie!

Functies gebruik je dus om herhaling in je code te voorkomen, maar ook om je code te structureren. Naast het gebruik van functies zijn er nog meer manieren om je code te structureren, zoals werken in verschillende bestanden en mappen. Ook dat leer je in dit hoofdstuk.

**Leerdoelen**

Aan het eind van dit hoofdstuk:

- Begrijp je wat een functie is
- Begrijp je hoe functies helpen om je code te structureren
- Begrijp je wat argumenten in een functie zijn
- Begrijp je het verschil tussen positionele en sleutelwoordargumenten
- Begrijp je hoe je argumenten optioneel maakt
- Begrijp je wat en wanneer een functie iets teruggeeft
- Begrijp je hoe je code in bestanden en mappen structureert
- Begrijp je wat een import is en hoe dit werkt

# Functies: een inleiding

Een functie is een stuk code dat je eerst definieert om later (meermaals) te gebruiken. Een functie kan optioneel **argumenten** hebben en kan optioneel een waarde teruggeven met **return**. In de basis ziet een functie er als volgt uit:

```
1 def kwadraat(getal):
2     kwadraat = getal ** 2
3     return kwadraat
```

Op regel één zie je dat een functie definitie begint met `def` (van *define*), gevolgd door een zelfgekozen naam. Na de naam volgen twee haakjes (`()`) met daarin *optioneel* één of meer **argumenten**. In dit geval is er één argument met de naam `getal`. De definitie sluit je af met een dubbele punt, waarna op regel twee het blok van de functie begint. Alle code in dit blok zal worden uitgevoerd als je de functie aanroept.

In de volgende paragrafen leer je meer over het werken met functies en hoe je ze gebruikt om je code te structureren.

# Werken met functies

Nu je een basisbegrip hebt van functies, leer je in deze paragraaf hoe je met functies werkt.

## Functies aanroepen

Bekijk nog eens de volgende eenvoudige functie:

```
1 def kwadraat(getal):
2     kwadraat = getal ** 2
3     return kwadraat
```

Als je het bestand met deze code uitvoert, zal er nog niets zichtbaars gebeuren. De functie is gedefinieerd, maar nog niet gebruikt. Daarvoor dien je de functie **aan te roepen**:

```
1 kwadraat(getal=10)  # Roep aan met naam van argument
2 kwadraat(10)  # Roep aan zonder naam van argument
```

Je ziet dat je de naam van het argument `getal` ook weg kunt laten. Verderop leer je hier meer over.

Voer je het bestand nu uit, dan zie je nog steeds niets zichtbaars gebeuren in Thonny (in de shell overigens wel). De functie geeft het resultaat terug met `return`, maar in dit geval wordt het resultaat nergens aan teruggegeven en 'verdwijnt' het.

Wijs je het resultaat toe aan een variabele, dan kun je er verder mee werken. Bijvoorbeeld het resultaat printen:

```
1 resultaat = kwadraat(getal=10)
2 print(resultaat)  # 100
```

Je kunt niet alleen het resultaat van een functie aan een variabele toewijzen, maar ook de functie zelf. Vervolgens kun je dan *die variabele* aanroepen:

```
mijn_functie = kwadraat
resultaat = mijn_functie(getal=10)
print(resultaat)  # 100
```

Omdat dit werkt, kun je een functie ook als argument meegeven aan een andere functie:

```
def kwadraat(getal):
    return getal ** 2

def bereken(berekening, getal):
    return berekening(getal=getal)

resultaat = bereken(berekening=kwadraat, getal=10)
print(resultaat)  # 100
```

De functie `bereken` ontvangt als eerste argument de functie `kwadraat`, die roep je vervolgens in het blok aan met het meegegeven getal. In je code moet je er wel voor zorgen dat de functie die als argument wordt meegegeven, de parameters heeft die je verwacht. Doe je dat niet, dan kunnen er fouten ontstaan, zoals hieronder:

```
def groet(naam):
    return f"Hallo, {naam}!"

def bereken(berekening, getal):
    return berekening(getal=getal)  # `groet` heeft geen argument `getal`

resultaat = bereken(berekening=groet, getal=10)  # Fout
```

Als je de functie `bereken` iets aanpast zodat hij geen sleutelwoordargument ontvangt (zie volgende paragraaf), dan gaat het wel goed, maar krijg je onverwachte resultaten:

```
def groet(naam):
    return f"Hallo, {naam}!"

def bereken(berekening, getal):
    return berekening(getal)  # Aangepast

resultaat = bereken(berekening=groet, getal=10)
print(resultaat)  # Hallo, 10!
```

Let er tot slot op dat je geen haakjes (`()`) gebruikt bij het meegeven als argument. Doe je dat wel, dan roep je de functie aan en geef je dus het resultaat mee.

### Opdracht 1: Kwadraat

Neem de hierboven gedefinieerde functie `kwadraat` en schrijf een stuk code dat als resultaat een lijst met alle kwadraten van 1 tot en met 100 oplevert.

Tip: met `getallen = range(1, 101)` maak je een lijst met de getallen 1 tot en met 100.

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is als volgt:

```
1 resultaat = []
2 for getal in range(1, 101):
3     resultaat.append(kwadraat(getal=getal))
```

## Argumenten in een functie

In deze paragraaf leer je meer over de argumenten in een functie.

#### Positionele argumenten en sleutelwoordargumenten

Eerder zag je al dat je één of meer argumenten kunt opgeven bij het definiëren van een functie. Deze argumenten dien je vervolgens ook op te geven als je de functie aanroept.

Het is aan de code die de functie aanroept of je de argumenten expliciet noemt of niet, onderstaande levert hetzelfde op:

```
1 def groet(naam):
2     print(f"Hallo, {naam}")
3
4 groet("John")  # Hallo, Johnn
5 groet(naam="John") # Hallo, Johnn
```

Stel dat je de functie aanpast naar het groeten met voor- en achternaam:

```
1 def groet(voornaam, achternaam):
2     print(f"Hallo, {voornaam} {achternaam}")
```

Wil je de functie nu aanroepen *zonder* de argumenten te benoemen, dan is het belangrijk de volgorde van de definitie aan te houden. Python zal namelijk de eerst opgegeven waarde koppelen aan het eerste argument, de tweede waarde aan het tweede argument, enzovoorts.

```
1 groet("John", "Doe")  # Hallo, John Doe
2 groet("Doe", "John")  # Hallo, Doe John
```

Roep je de functie op deze wijze aan, dan spreek je van **positionele argumenten** (omdat de positie uitmaakt).

Je kunt de functie ook aanroepen mét het benoemen van de argumenten. De volgorde maakt in dat geval niet meer uit:

```
groet(voornaam="John", achternaam="Doe")  # Hallo, John Doe
groet(achternaam="Doe", voornaam="John")  # Hallo, John Doe
```

Doordat je de argumenten expliciet koppelt aan de naam, maakt de volgorde niet uit, Python weet nu wat je bedoelt. Dit noem je **sleutelwoordargumenten** (*keyword argument*).

Je kunt positionele argumenten en sleutelwoordargumenten ook door elkaar gebruiken. Wel is het belangrijk dat sleutelwoordargumenten altijd *na* de positionele argumenten komen.

```
groet("John", achternaam="Doe")  # Hallo, John Doe
groet("Doe", voornaam="John")  # Werkt niet
```

In het laatste geval zul je een foutmelding krijgen dat `voornaam` tweemaal is opgegeven. De eerste waarde wordt automatisch aan het eerste argument (`voornaam`) gekoppeld. Nogmaals `voornaam` opgeven is dan niet mogelijk.

#### Optionele argumenten

In bovenstaande `groet`-functie zijn de twee argumenten `voornaam` en `achternaam` verplicht. Als je de functie aanroept met bijvoorbeeld alleen de `voornaam`, krijg je een foutmelding.

Er zijn situaties denkbaar waarbij je één of meer argumenten *optioneel* wil maken. Dit doe je door het een standaardwaarde toe te kennen bij het definiëren. Dit doe je door het argument te laten volgen door een is-teken (`=`), gevolgd door de standaard waarde.

```
def groet(naam, enthousiasme_niveau=1):
    uitroeptekens = "!" * enthousiasme_niveau
    print(f"Hallo, {naam}{uitroeptekens}")

groet("John")  # Hallo, John!
groet("John", enthousiasme_niveau=1)  # Hallo, John!
groet("John", enthousiasme_niveau=2)  # Hallo, John!!
groet("John", enthousiasme_niveau=0)  # Hallo, John
```

Zoals je ziet kun je de functie nu aanroepen met alleen een naam. In dat geval wordt er standaard één uitroepteken geplaatst. Je kunt dit nog steeds expliciet maken als je wil, door alsnog `enthousiasme_niveau=1` mee te geven. Je kunt ook andere waarden toekennen.

Een ander veelgebruikte mogelijkheid is `None` als standaardwaarde mee te geven, en hier in het functieblok op te controleren:

```
def groet(voornaam, achternaam=None):
    volledige_naam = voornaam
    if achternaam:
        volledige_naam += " " + achternaam

    print(f"Hallo, {volledige_naam}.")

groet("John")  # Hallo, John.
```

```
9 groet("John", "Doe")  # Hallo, John Doe.
```

### Opdracht 2: Begroeting

Maak een begroetingsfunctie die, naast de naam, een argument `dagdeel` accepteert. Begroet de persoon op de juiste wijze, bijvoorbeeld: `Goedemorgen, John.`. Als geen dagdeel wordt opgegeven, laat het dan weg uit de begroeting.

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is als volgt:

```
1 def begroeting(naam, dagdeel=None):
2     if dagdeel and dagdeel not in ["morgen", "middag", "avond"]:
3         print("Dagdeel moet 'morgen', 'middag' of 'avond' zijn")
4 
5     else:
6         if dagdeel:
7             goede = "Goede"
8             if dagdeel == "avond":
9                 goede += "n"
10            print(f"{goede}{dagdeel}, {naam}")
11        else:
12            print(f"Hallo, {naam}")
```

De term parameter verwijst naar de namen bij het definiëren van de functie. De term argument verwijst naar het object dat je meegeeft aan de functie. In deze en andere hoofdstukken lees je in beide gevallen de term argument.

```
1 def groet(naam):  # `naam` is een parameter
2     print(naam)
3 
4 groet(naam="John")  # `John` is het meegegeven argument
```

## Teruggeven van waarden uit een functie

In de meeste voorbeelden zag je dat de functie iets print. In de praktijk zul je een functie hiervoor niet vaak gebruiken, maar wil je dat de functie iets doet, en dan het resultaat *teruggeeft*. Dit teruggeven doe je met het sleutelwoord `return`.

```
1 def kwadraat(getal):
2     kwadraat = getal ** 2
3     return kwadraat
4 
5 resultaat = kwadraat(getal=10)
6 print(resultaat)  # 100
```

### Meerdere returns

Het teruggeven hoeft niet per se aan het einde van de functie, het kan bijvoorbeeld afhankelijk zijn van een `if-else` statement dat je terug geeft. Alle code na een uitgevoerde `return` wordt niet uitgevoerd.

```
def controleer_even_getal(getal):
    if getal % 2 != 0:
        return "Het opgegeven nummer is oneven."

    return "Het opgegeven nummer is even."


print(controleer_even_getal(7))  # Het opgegeven nummer is oneven.
print(controleer_even_getal(4))  # Het opgegeven nummer is even.
```

Als het getal oneven is, dan wordt de laatste `return` nooit bereikt.

In dit voorbeeld zie je de modulo-operator (`%`). Hiermee bereken je het restant van de deling van de twee getallen. Bijvoorbeeld: `10 % 2` levert `0` op, omdat twee vijf keer in 10 past. `9 % 2` levert `1` op, omdat 2 4 keer in 9 past, en je dan 1 overhoudt. Hiermee kun je dus eenvoudig berekenen of een getal even of oneven is.

### None

In de voorbeelden waar alleen een `print` in de functie voorkwam, lijkt het alsof de functie niets teruggeeft. Het is echter zo dat Python aan het einde van de functie impliciet `None` teruggeeft wanneer je geen expliciete `return` opgeeft. Je kunt dit eenvoudig controleren:

```
def groet(naam):
    print(f"Hallo, {naam}")
    # Geen return

resultaat = groet(naam="John")
print(resultaat)  # None
```

Je mag ook zelf `return None` toevoegen, of korter: alleen `return`. Dit kun je gebruiken als je met bijvoorbeeld een `if-else` statement de functie vroegtijdig wil verlaten zonder waarde:

```
def controleer_even_getal(getal):
    if getal % 2 != 0:
        return

    return "Het opgegeven nummer is even."
```

In dit geval geeft de functie `None` terug als het getal oneven is.

### Opdracht 3: Delen

Maak de volgende functie:

```
def deel_getallen(getal_1, getal_2):
    pass  # Vervang dit door je eigen code
```

Zorg ervoor dat het juiste resultaat wordt teruggegeven. Houd hierbij rekening met het feit dat je niet kunt delen door 0!

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is als volgt:

```
def deel_getallen(getal_1, getal_2):
    if getal_2 == 0:
        return "Delen door 0 is niet mogelijk"

    return getal_1 / getal_2
```

## Je code structureren met functies

Een functie kun je zien als een bouwsteen van je code. Met meerdere functies bouw je zo aan een uitgebreider programma. Het is raadzaam om je functies één verantwoordelijkheid te geven, dat houdt je code overzichtelijk en begrijpelijk. Geef je functienamen ook duidelijke, beschrijvende namen.

Net als bij de namen van variabelen zijn er een aantal regels voor en afspraken over functienamen. De regels zijn:

- Gebruik alleen kleine- en hoofdletters (`a` tot en met `z`, `A` tot en met `Z`), *underscores* (`_`) en cijfers (`0` tot en met `9`)
- Gebruik geen cijfer als eerste teken

De stijl-afspraak is om enkel kleine letters en *underscores* te gebruiken voor functienamen (ook wel *snake_case* genoemd). Bijvoorbeeld: `mijn_functie`.

Als eenvoudig voorbeeld zie je hieronder een programma dat de BMI van de gebruiker uitrekent.

```
def verkrijg_gegevens():
    """
    Vraag lengte en gewicht om de BMI te kunnen berekenen.
    """
    lengte = input("Wat is je lengte (in cm) ")
    gewicht = input("Wat is je gewicht (in kg)? ")

    lengte = int(lengte)
    gewicht = int(gewicht)

    return lengte, gewicht
```

```
12
13
14 def bereken_bmi(lengte, gewicht):
15     """
16     BMI: Gewicht in kg, gedeeld door het kwadraat van lichaamslengte in meters.
17     """
18     return gewicht / (lengte/100)**2
19
20
21 def main(naam):
22     """
23     Hoofdfunctie, bereken het BMI van de gebruiker.
24     """
25     print(f"Welkom bij de BMI-calculator, {naam}.\n")
26
27     lengte, gewicht = verkrijg_gegevens()
28     bmi = bereken_bmi(gewicht=gewicht, lengte=lengte)
29     bmi = round(bmi, 2)
30
31     print(f"\n{naam}, je BMI is {bmi}.\n")
32
33 main("John")  # Roep main aan
```

Je ziet dat er drie functies zijn gedefinieerd. De functie `vraag_gegevens` handelt het uitvragen van de persoonsgegevens af. De functie `bereken_bmi` handelt het berekenen van de BMI af. De laatste functie (`main`) voegt tot slot alles samen.

Er zijn een aantal zaken die opvallen:

**Uitpakken** In de functie `verkrijg_gegevens` worden twee variabelen teruggegeven, gescheiden door een komma. In de functie `main` worden deze variabelen *uitgepakt*. In Uitpakken lees je nog eens over het uitpakken van `tuples`.

**Duidelijkheid** Elke functie heeft een duidelijke naam en een korte beschrijving van wat hij doet. Je hoeft enkel de functie `main` te lezen om te begrijpen wat er gaande is, zonder de inhoud van de overige functies te lezen.

**Volgorde** De volgorde van definiëren en aanroepen is relevant. In Python kun je pas iets aanroepen nadat het is gedefinieerd. Python leest elk bestand van boven naar beneden. Zou je `main()` naar boven verplaatsen, dan krijg je een foutmelding dat `verkrijg_gegevens` niet bestaat.

Je kunt de volgorde van de functies wel omdraaien. Pas op het moment dat je `main()` uitvoert, worden ook de andere functies aangeroepen, en op dat moment zijn ze al gedefinieerd. Het maakt dan niet uit dat bijvoorbeeld `bereken_bmi` eerder is gedefinieerd dan `verkrijg_gegevens`. Als beide maar beschikbaar zijn op het moment dat `main` wordt aangeroepen.

### Opdracht 4: Rekenen

Maak vier functies die elk twee getallen accepteren:

- delen
- vermenigvuldigen
- optellen

- aftrekken

Maak een vijfde functie die ook twee getallen accepteert en de naam van het gewenste type berekening. Print het resultaat.

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is als volgt:

```
# Defineer de berekeningsfuncties
def optellen(getal_1, getal_2):
    return getal_1 + getal_2

def aftrekken(getal_1, getal_2):
    return getal_1 - getal_2

def delen(getal_1, getal_2):
    if getal_2 == 0:
        return

    return getal_1 / getal_2

def vermenigvuldigen(getal_1, getal_2):
    return getal_1, getal_2


# De hoofdfunctie
def main(berekening, getal_1, getal_2):
    if berekening == "optellen":
        return optellen(getal_1, getal_2)
    elif berekening == "aftrekken":
        return aftrekken(getal_1, getal_2)
    elif berekening == "delen":
        return delen(getal_1, getal_2)
    elif berekening == "vermenigvuldigen":
        return vermenigvuldigen(getal_1, getal_2)
    else:
        return
```

# Je code verder structureren

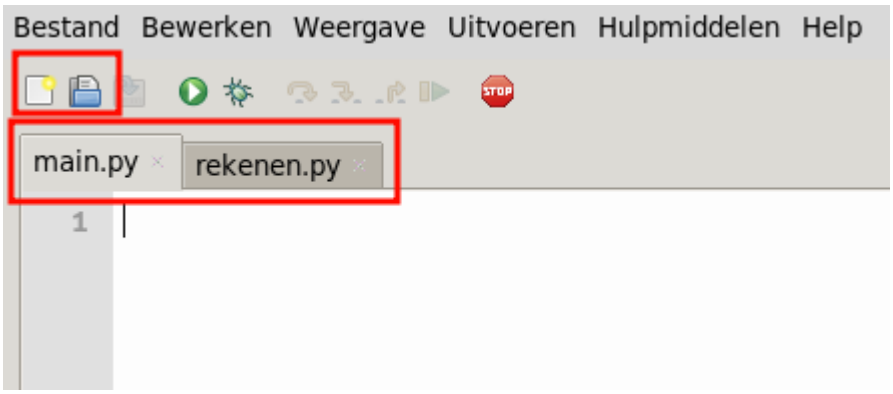
Je hebt geleerd dat je functies kunt gebruiken om je code op te delen in behapbare stukjes code, die je ook kunt hergebruiken. Als je programma groter wordt, zul je merken dat het handig is om je functies te verdelen over verschillende bestanden en mappen zodat je overzicht houdt. Hoe je dat doet leer je in deze paragraaf.

## Je code structureren met modules (bestanden)

De eerste stap in het structureren van je code is vaak het verdelen over meerdere bestanden. Dergelijke bestanden heten in Python **modules**. Hoe je de bestanden indeelt is aan jou en hangt af van de context van het programma. Dit kan bijvoorbeeld op onderwerp of, type functionaliteit zijn.

Tot dusver heb je steeds gewerkt in één `.py`-bestand. Om de werking van modules te demonstreren maak je nu twee bestanden aan in dezelfde map: `main.py` en `rekenen.py`.

In Thonny kun je een nieuw bestand maken en opslaan, naast het bestand waar je al in werkt. Het nieuwe bestand opent in een nieuw tabblad, zodat je eenvoudig tussen beide bestanden kunt wisselen.




*Meerdere bestanden in Thonny*

In `rekenen.py` plaats je twee functies: `kwadraat` en `getal_is_even`.

```
1 def kwadraat(getal):
2     kwadraat = getal ** 2
3     return kwadraat
4
5
6 def getal_is_even(getal):
7     if getal % 2 != 0:
8         return False
9
10    return True
```

Als je deze functies in `main.py` wil gebruiken, moet je ze **importeren**. Dit doe je met `import`. Hoewel het niet verplicht is, is het de gewoonte om dit bovenaan het bestand te doen. Er zijn twee manieren om de functies te importeren. Met de eerste manier importeer je de gehele module, en dus alle inhoud ervan, in één keer.

```
1 # main.py
2 import rekenen
```

Je gebruikt `import` met daarachter de naam van de module die je wil importeren. Je laat hierbij het achtervoegsel `.py` weg.

Vervolgens kun je de functies en andere gegevens uit `rekenen.py` gebruiken in `main.py`:

```
1 # main.py
2 import rekenen  # Importeer de gehele module `rekenen`
3
4 getal = 10
5 even_of_oneven = "oneven"
6
```

```
# Gebruik functie `kwadraat` uit module `rekenen`
kwadraat = rekenen.kwadraat(getal=getal)

# Gebruik functie `getal_is_even` uit module `rekenen`
is_even = rekenen.getal_is_even(getal=getal)

if is_even:
    even_of_oneven = "even"

print(f"Het kwadraat van {getal} is {kwadraat}. {getal} is een {even_of_oneven} getal.")
```

Je hebt de module `rekenen` als geheel geïmporteerd. Om hier nu iets uit te gebruiken is het nodig om de functienaam vooraf te laten gaan door `rekenen`, gescheiden door een punt. Doe je dit niet, dan zul je een fout krijgen omdat de functie niet in de huidige module is gedefinieerd.

Je kunt de functies die je nodig hebt ook direct importeren:

```
# main.py
from rekenen import kwadraat  # Importeer de functie `kwadraat` uit rekenen

# Gebruik functie `kwadraat` uit module `rekenen`.
# Deze is nu direct beschikbaar in `main`
kwadraat = kwadraat(getal=10)
print(kwadraat)  # 100
```

In dit geval heb je alleen de functie `kwadraat` nodig. Met `from ⋯ import ⋯` geef je aan wat je precies uit welke module wil importeren. In het huidige bestand kun je nu direct de naam van de functie gebruiken, zonder de module te noemen.

Wil je meerdere functies importeren, dan kan dat ook:

```
# main.py
from rekenen import kwadraat, getal_is_even
```

Je scheidt de te importeren items dan door een komma.

## Je code structuren met packages (mappen)

Net als met het werken met bijvoorbeeld foto's op je computer, kun je de modules ook nog verder organiseren door ze in mappen te plaatsen. Deze mappen noem je **packages**.

Niet elke map is een *package*, je dient er eerst een Python-bestand met de bestandsnaam `__init__.py` in te plaatsen (aan beide kanten van *init* staan twee *underscores*). Dit is het bestand dat zal worden uitgevoerd als je de *package* importeert. Over het algemeen kun je het leeg laten. Maak in de map waar je in werkt, een nieuwe map `functies` aan, met daarin drie Python-bestanden:

- `__init__.py`
- `begroetingen.py`

- `rekenen.py` (verplaats het voorgaande bestand naar deze map)

Plaats in `begroetingen.py` een eenvoudige `groet` functie.

```
def groet(naam):
    print(f"Hallo, {naam}.")
```

In dezelfde map als waar je de map `functies` hebt gemaakt, plaats je een bestand `main.py`.


*Indeling van modules en package*

Importeren werkt vrijwel hetzelfde als eerder, maar nu moet je ook aan geven uit welke *package* je wil importeren.

```
# main.py

# Importeer de gehele module `begroetingen` uit package `functies`
from functies import begroetingen

# Gebruik de functie `groet` uit de module `begroetingen`
begroetingen.groet(naam="John")
# Hallo, John.
```

Om alle functies van `begroetingen` te importeren (in dit geval is dat er maar één), gebruik je `from ⋯ import ⋯`. Je importeert nu `begroetingen` in zijn geheel, en alles is dus beschikbaar. Wel moet je je `groet` aan roepen met de module-naam `begroetingen` eraan voorafgaand.

Je kunt ook weer specifieke functies importeren:

```
# main.py

# Importeer de functie `kwadraat` uit de module `rekenen`, welke zich in
# de package `functies` bevindt.
from functies.rekenen import kwadraat

# Gebruik functie `kwadraat` uit module `rekenen`, nu direct beschikbaar in `main`
kwadraat = kwadraat(getal=10)
print(kwadraat)  # 100
```

Je ziet dat je weer de structuur `from ⋯ import ⋯` gebruikt. Maar nu geef je op uit welke module (rekenen) uit welke package (functies) je wil importeren door beide te benoemen, gescheiden door een punt (*dotted path*): `functies.rekenen`.

## Gebruik maken van de standaard bibliotheek

Naast dat je `import` kunt gebruiken om je eigen code te importeren, kun je het ook gebruiken om modules die standaard al aanwezig zijn in Python te importeren.

Python heeft een uitgebreide **standaard bibliotheek**. Hierin is allerlei functionaliteit beschikbaar, zodat je die niet zelf hoeft te programmeren. Te denken valt aan mathematische bewerkingen, werken met bestanden, werken met websites, en nog veel meer.

Bekijk de lijst met ingebouwde functionaliteit op docs.python.org/3/library/

Importeren werkt exact hetzelfde als met je eigen modules en packages:

```
1 from math import pi, floor
2
3 # Bereken oppervlakte cirkel (πr²)
4 straal_in_cm = 5
5 oppervlakte = pi * straal_in_cm**2
6
7 # Afkappen op geheel getal
8 oppervlakte = floor(oppervlakte)
9
10 print(f"De oppervlakte van een cirkel met een "
11       f"straal van {straal_in_cm}cm "
12       f"is ongeveer: {oppervlakte}cm")
```

## Importeren en uitvoeren

Als je een bestand uitvoert vanuit de shell of Thonny, dan zal Python alle code in het bestand uitvoeren. Zodra het een `import` tegenkomt, zal het de import uitvoeren en de code in het geïmporteerde bestand ook uitvoeren.

Dit kan voor problemen zorgen als je een Python-bestand hebt dat je soms direct wil aanroepen, en soms wil gebruiken in een import. Neem het volgende voorbeeld, met twee bestanden `bestand_1.py` en `bestand_2.py`.

```
1 # bestand_1.py
2 import bestand_2
3
4 print(bestand_2.getal_is_even(10))
5
6 # 100
7 # True
8
9 # bestand_2.py
10 def kwadraat(getal):
11     kwadraat = getal ** 2
12     return kwadraat
13
14
15 def getal_is_even(getal):
16     if getal % 2 != 0:
```

```
17         return False
18
19     return True
20
21 print(kwadraat(10))
```

In bestand_1 importeer je bestand_2 en print je het resultaat van de functie getal_is_even(10). In bestand_2 heb je echter nog een functie gedefinieerd (kwadraat) en voer je die uit op regel 21.

Voer je nu bestand_1 uit, dan zie je dat niet alleen True wordt geprint, maar ook 100, het resultaat van de code op regel 21 in de module bestand_2. Dit is het gevolg van het feit dat alle code wordt uitgevoerd door Python, ook bij het importeren.

Als je wil dat regel 21 van module bestand_2 alleen wordt uitgevoerd als bestand_2 direct wordt uitgevoerd en niet als het wordt geïmporteerd, moet je te weten dat Python op de achtergrond aan elk bestand dat wordt uitgevoerd een speciale variabele __name__ koppelt. Als je bestand_2 importeert, zal __name__ de waarde bestand_2 krijgen. Maar als je bestand_2 direct uitvoert, zal het de naam __main__ krijgen, omdat het dan wordt beschouwd als het entreepunt van je programma.

De variabelen __name__ en __main__ hebben aan beide kanten van het woord twee *underscores*.

Dit gegeven kun je gebruiken om code alleen uit te voeren als een module direct wordt uitgevoerd. Bekijk de wijziging in bestand_2:

```
1 # bestand_2.py
2
3 ...
4
5 if __name__ == "__main__":
6     print(kwadraat(10))
```

Voer je nu bestand_1 uit, dan wordt bestand_2 geïmporteerd en krijgt __name__ de waarde bestand_2. Omdat __name__ dus geen __main__ is, zal de functie niet worden uitgevoerd. Voer je bestand_2 direct uit, dan krijgt __name__ de waarde __main__ en zal de functie wel worden uitgevoerd.

# Werken met klassen

## Inleiding

In Werken met functies heb je geleerd hoe je functies kunt gebruiken om herhaling van code voorkomen. Is een bepaalde taak vaak nodig, dan kun je er een functie van maken en deze elke keer aanroepen als het nodig is. In dit hoofdstuk leer je wat een **klasse** is, *class* in het Engels. Ook een klasse gebruik je - onder andere - om herhaling van code te voorkomen, maar is uitgebreider dan een functie.

In de volgende paragraaf leer je eerst wat een klasse is en hoe je er een maakt. Vervolgens leer je ermee te werken.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je wat klassen zijn
- Begrijp je wanneer je klassen gebruikt
- Begrijp je wat een methode is en welke type methodes er zijn
- Begrijp je wat overerving is en wat compositie is

## Klassen

Zonder dat je het weet ben je al verschillende klassen tegengekomen. Voer in Thonny of in een shell maar eens het volgende uit:

```
1 print(type("Python"))
2 # <class 'str'>
3
4 print(type(5))
5 # <class 'int'>
```

Vraag je het type op van een tekst, dan krijg je terug dat het type een *klasse* `str` is. Sterker, elke ingebouwd type is een bepaalde klasse. Met een klasse *kun je nu ook je eigen types definiëren*. Feitelijk zijn typen en klassen dus hetzelfde. Het heeft met historie te maken dat beiden begrippen bestaan. In (veel) oudere versies van Python was er verschil tussen ingebouwde typen en gebruiker-gedefinieerde klassen.

Een klasse is dus een type, maar wat kun je ermee? Een klasse bepaalt de structuur en het gedrag van één of meer objecten. De structuur verwijst naar welke gegevens de klasse kan bevatten, aangeduid met **attributen**. Hierover lees je later meer. Het gedrag verwijst naar wat een object *kan*, met **methodes**. Dit heb je ook al gezien bij de ingebouwde types:

```
1 mijn_tekst = "Lorem Ipsum"  # type <class 'str'>
2 mijn_getal = 42  # type <class 'int'>
3
4 # Roep de methode 'lower()' aan op `mijn_tekst`
5 mijn_tekst.lower()  # lorem ipsum
6
7 # Roep de methode 'lower()' aan op `mijn_getal`
8 mijn_getal.lower()  # Fout, int heeft geen methode `lower`
```

Hier zie je welk gedrag mogelijk is afhankelijk van de klasse (type). Een `str` heeft de methode `lower()` (dat alle tekst naar kleine letters omzet), maar een `int` heeft deze methode niet. Die heeft weer andere methodes (ander gedrag). Andere voorbeelden zijn de `list` (`.append()`, `.pop()`, etc.) of `tuple` (`.count()`, `.index()`).

Over methodes lees je in een volgende paragraaf meer.

### Je eigen type maken

Met een klasse maak je dus een eigen type. Hieronder zie je een eenvoudig voorbeeld:

```
1 class Gebruiker:
2     pass
```

Je maakt een klasse door het woord `class` te laten volgen door een zelfgekozen naam, gevolgd door een dubbele punt (`:`). Daaronder begint het blok. Omdat dit voorbeeld (nog) niets doet, maar je geen leeg blok mag hebben gebruik je `pass` om het blok te vullen. Hiermee geef je aan dat je niets wil doen.

De conventie voor klassennamen is om *CamelCase* te gebruiken. Dus Gebruiker, MijnKlasse, EenKlasseMetEenLangeNaam, etc. De uitzondering hierop zijn de ingebouwde types (`int`, etc.).

Net als alles bij variabelen en functies: gebruik duidelijke, omschrijvende namen voor je klassen. `X` is misschien een goede naam voor een social media platform, een klasse met de naam `X` vertelt weinig over wat het doet.

Een klasse is een *blauwdruk* voor objecten. Maak je een object aan op basis van een klasse, dan noem je dat object een **instantie** van die klasse.

```
1 # Maak twee instanties van `Gebruiker` aan
2 john = Gebruiker()
3 maria = Gebruiker()
4
5 print(type(john))  # <class '__main__.Gebruiker'>
6 print(type(maria))  # <class '__main__.Gebruiker'>
```

Hier maak je twee *instanties* van de klasse `Gebruiker` aan. Vraag je het type op, dan zie je inderdaad `Gebruiker` terug.

Een eigen type maken wordt met name handig als je ook je eigen *methodes* gaat toevoegen, ofwel gedrag gaat toevoegen. Een methode is feitelijk hetzelfde als een functie, maar dan als onderdeel van een klasse. Waar je een functie direct aanroept, roep je een methode altijd aan als onderdeel van de klasse waar het toe behoort.

Als voorbeeld kun je de methode `info` toevoegen aan de klasse `Gebruiker`:

```
1 class Gebruiker:
2
3     def info(self):
4         return "Gebruiker"
```

Je ziet dat je een methode hetzelfde aanmaakt als een functie, maar nu in het blok van de klasse. Verder valt op

dat het eerste (en in dit geval enige) argument `self` heet. De methode die je hier hebt gemaakt heet een **instantiemethode**, omdat het een methode is die werkt met de instanties van de klasse. Een instantiemethode *moet* als eerste argument een verwijzing naar de instantie bevatten. Het is niet verplicht om dit `self` te noemen, maar dit is wel zeer gebruikelijk.

```
1 john = Gebruiker()
2 john.info()  # "Gebruiker"
```

In bovenstaand voorbeeld heb je een instantie van de klasse `Gebruiker` gemaakt, genaamd `john`. Vervolgens roep je de methode `info` aan (die nu nog niet zoveel doet). Bij het aanroepen van de methode laat je `self` achterwege. In de methode verwijst `self` nu naar de instantie `john` (naar zichzelf dus, vandaar de conventie om het `self` te noemen).

In dit voorbeeld gebruik je `self` niet, dus waarom is het nuttig? Om dat te illustreren breidt je de klasse nog meer uit, met een speciale **initialisatiemethode** `__init__`. Zoals de naam doet vermoeden wordt deze methode automatisch aangeroepen als de klasse wordt geïnitialiseerd, ofwel als je `john = Gebruiker()` uitvoert. De `__init__` methode gebruik je om attributen te initialiseren.

Ook `__init__` heeft als eerste argument `self`, maar kan ook meer **argumenten** ontvangen (dit geldt overigens voor elke methode in een klasse):

```
1 class Gebruiker:
2
3     def __init__(self, naam, email):
4         self.rol = "Gebruiker"
5         self.naam = naam
6         self.email = email
7
8     def info(self):
9         return f"""
10        Naam: {self.naam}\n
11        E-mail: {self.email}\n
12        Rol: {self.rol}\n
13        """
```

In de `__init__` methode wijs je op regel vier de ontvangen *argumenten* `naam` en `email` toe aan de *attributen* `naam` en `email`. `self` verwijst naar de instantie, en `naam` en `email` worden nu attributen van de instantie. Verder voeg je een attribuut `rol` toe, die voor alle gebruikers (voor alle instanties) hetzelfde is.

In de `info` methode kun je nu verwijzen naar de attributen, zoals op regel 10 t/m 12. Maak je nu een instantie aan, dan moet je de argumenten `naam` en `email` opgeven. Dit werkt net als bij functies: je mag de naam weglaten, maar je mag het ook expliciet benoemen.

```
1 john = Gebruiker(naam="John", email="john@example.com")
2 john.info()
3 # Naam: John
4 # E-mail: john@example.com
5 # Rol: Gebruiker
6
```

```
print(john.email)  # john@example.com
```

Op de laatste regel zie je dat het attribuut `email` beschikbaar is voor deze instantie. Maak je een andere instantie aan, dan krijg je andere waarden:

```
maria = Gebruiker(naam="Maria", email="maria@example.com")
maria.info()
# Naam: Maria
# E-mail: maria@example.com
# Rol: Gebruiker

print(maria.email)  # maria@example.com
```

Attributen kun je ook per instantie wijzigen:

```
maria.email = "maria42@example.com"  # Wijzig e-mailadres van deze instantie
print(maria.email)  # maria42@example.com
maria.info()  # Teruggegeven informatie is ook veranderd
```

Het kan zinvol zijn om attributen niet direct te aan te passen, maar via een methode. Python is een flexible taal en het zal je niet *verbieden* attributen direct aan te passen. Maar als je beperkingen wil opleggen aan wat mogelijk is met een attribuut, kun je dat als volgt oplossen:

```
class Gebruiker:

    def __init__(self, naam, email):
        self._rol = "Gebruiker"
        self._naam = naam
        self._email = email

    def info(self):
        return f"""
        Naam: {self._naam}\n
        E-mail: {self._email}\n
        Rol: {self._rol}\n
        """

    def wijzig_email(self, nieuw_email):
        # Voer hier enkele checks uit
        # of het e-mailadres geldig is
        self._email = nieuw_email
```

Door een *underscore* voor de attributen te zetten, zeg je als het ware: "Gebruik deze attributen niet direct". Nogmaals: Python zal het niet voorkomen, het is eerder een conventie.

Met de nieuwe methode `wijzig_email` pas je nu het e-mailadres aan met het nieuw opgegeven e-mailadres. In deze methode kun je eerst allerlei checks uitvoeren (is het e-mailadres geldig, is het een e-mailadres dat tot het

bedrijf behoort, is het e-mailadres al niet aan een andere gebruiker gekoppeld, etc.).

Zou je de gebruiker van deze klasse het e-mailadres direct laten wijzigen (zoals in het eerdere voorbeeld), dan bestaat het gevaar dat al deze checks niet zijn uitgevoerd.

### Opdracht 1: Groet

Neem de klasse `Gebruiker` als uitgangspunt. Voeg nu een methode toe die de gebruiker groet. Maak het mogelijk de groet formeel en informeel uit te voeren.

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is als volgt:

```
class Gebruiker:
    _rol = "Gebruiker"

    def __init__(self, naam, email):
        self._naam = naam
        self._email = email

    def groet(self, formeel=False):
        if formeel:
            return f"Gegroet, {self._naam}."

        return f"Hé {self._naam}!"
```

## Overerving

Een belangrijk concept bij klassen is **overerving**. Dit houdt in dat je een klasse baseert op een andere klasse. Om dit te illustreren gebruik je nogmaals de klasse `Gebruiker`:

```
class Gebruiker:

    def __init__(self, naam, email):
        self._rol = "Gebruiker"
        self._naam = naam
        self._email = email

    def info(self):
        return f"""
        Naam: {self._naam}\n
        E-mail: {self._email}\n
        Rol: {self._rol}\n
        """
```

Stel dat je nu naast gebruikers, ook beheerders wil maken. Een beheerder is vergelijkbaar met een reguliere gebruiker, maar heeft - uiteraard - een andere rol.

Nu kun je een heel nieuwe klasse `Beheerder` maken, en de `__init__` en `info` methodes herschrijven met een enkele aanpassing. Met overerving kun je echter gebruikmaken van het feit dat een groot deel van de gegevens hetzelfde is.

```
1 class Beheerder(Gebruiker):
2     pass
3
4 ali = Beheerder(naam="Ali", email="ali@example.com")
5 print(ali.info())
6 # Naam: Ali
7 # E-mail: ali@example.com
8 # Rol: Gebruiker
```

Op regel één zie je dat `Beheerder` is gebaseerd op `Gebruiker`. Hierdoor heeft `Beheerder` direct dezelfde attributen (naam, e-mail) en methodes (`info`) als `Gebruiker`. De klasse `Gebruiker` noem je een *parent class* of een *super class*, `Beheerder` noem je een *child class* of een *sub class*.

Het probleem is dat de rol nog als "Gebruiker" staat genoteerd. Er zijn dus enkele aanpassingen nodig. De rol wordt gedefinieerd in de `__init__` methode en vervolgens gebruikt in de `info` methode.

Een handigheid bij overerven is dat het alle kenmerken (attributen, methodes) van de *super class* overneemt in de *sub class*, maar dat deze wel aanpasbaar zijn. Neem onderstaande voorbeeld:

```
 1 class Beheerder(Gebruiker):
 2
 3     def __init__(self, naam, email):
 4         super().__init__(naam, email)
 5         self._rol = "Beheerder"
 6
 7 ali = Beheerder(naam="Ali", email="ali@example.com")
 8 print(ali.info())
 9 # Naam: Ali
10 # E-mail: ali@example.com
11 # Rol: Beheerder
```

In deze aangepaste versie van `Beheerder` definieer je de `__init__` methode opnieuw. Net als in de `Gebruiker` klasse ontvangt het de argumenten `naam` en `email`, welke aan de `__init__` methode van de *super class* worden doorgegeven. Dit doe je met het speciale woord `super()`.

Met `super()` roep je de *super class* aan. In dit geval roep je de `__init__` methode van de *super class*, dus van `Gebruiker` aan. Die `__init__` methode zorgt ervoor dat de attributen `naam` en `email` worden ingesteld, ook voor de *sub class*. Op regel vijf stel je tot slot `self._rol` in met de juiste rol "Beheerder". Hiermee overschrijf je de `self._rol` van `Gebruiker` als het ware.

Met een minimale aanpassing zorg je er zo voor dat een `Beheerder` de juiste rol krijgt en dat dit ook terugkomt in de `info` methode.

### Opdracht 2: Gebruiker toevoegen

Voeg aan `Beheerder` een methode toe die het mogelijk maakt een nieuwe `Gebruiker` toe te voegen. Laat

zien dat dit werkt, laat ook zien dat deze methode voor een `Gebruiker` niet beschikbaar is.

▼ *Klik om het antwoord te tonen*

Neem de klasse `Gebruiker` als basis:

```
class Gebruiker:
    _rol = "Gebruiker"

    def __init__(self, naam, email):
        self._naam = naam
        self._email = email

    def info(self):
        return f"""
        Naam: {self._naam}\n
        E-mail: {self._email}\n
        Rol: {self._rol}\n  # rol nog beschikbaar via `self`
        """
```

Dit is een mogelijke manier om `Beheerder` een nieuwe gebruiker te laten toevoegen:

```
class Beheerder(Gebruiker):
    _rol = "Beheerder"

    def voeg_gebruiker_toe(self, naam, email):
        return Gebruiker(naam=naam, email=email)


beheerder = Beheerder(naam="Ali", email="ali@example.com")
john = beheerder.voeg_gebruiker_toe(
    naam="John",
    email="john@example.com"
)

print(john.info())
# Naam: John
# E-mail: john@example.com
# Rol: Gebruiker


maria = john.voeg_gebruiker_toe(naam="Maria", email="maria@example.com")
# Fout
```

## Klassenattributen

Tot dusver heb je de rol gedefinieerd in de `__init__` methode. De rol is echter voor alle instanties hetzelfde. Elke instantie van `Gebruiker` heeft de rol "Gebruiker" en elke instantie van `Beheerder` heeft de rol "Beheerder".

Informatie dat instantie-overstijgend is, kun je ook in een klasse-variabele kwijt:

```
class Gebruiker:
    _rol = "Gebruiker"

    def __init__(self, naam, email):
        self._naam = naam
        self._email = email

    def info(self):
        return f"""
        Naam: {self._naam}\n
        E-mail: {self._email}\n
        Rol: {self._rol}\n  # rol nog beschikbaar via `self`
        """
```

Het attribuut `_rol` is nu buiten `__init__` gedefinieerd, direct als attribuut van de klasse `Gebruiker`. Je kunt het nu ook aanroepen met `Gebruiker._rol`. Maar ook elke instantie van `Gebruiker` heeft toegang tot `_rol`. Dit kan via `self`, zoals in de `info` methode, maar ook direct via het attribuut:

```
print(Gebruiker._rol)
john = Gebruiker("John", "john@example.com")
print(john._rol)  # Gebruiker
```

Dit maakt de definitie van de klasse `Beheerder` nog eenvoudiger, omdat je nu `__init__` niet meer hoeft te overschrijven:

```
class Beheerder(Gebruiker):
    _rol = "Beheerder"

ali = Beheerder(naam="Ali", email="ali@example.com")
print(ali.info())
# Naam: Ali
# E-mail: ali@example.com
# Rol: Beheerder
```

Let wel op: wijzig je de een klassenattribuut *via de klasse*, dan wijzigt het voor *alle* instanties!

```
class Beheerder(Gebruiker):
    _rol = "Beheerder"

beheerder1 = Beheerder("Ali", "ali@example.com")
beheerder2 = Beheerder("John", "john@example.com")

print(beheerder1._rol)  # Beheerder
print(beheerder2._rol)  # Beheerder
```

```
9
10 # Update de rol van de klasse
11 Beheerder._rol = "Administrator"
12
13 print(beheerder1._rol)  # Administrator
14 print(beheerder2._rol)  # Administrator
```

## Compositie

Naast overerving is **compositie** een veelgebruikte strategie bij het ontwerpen van klassen. Zoals het woord doet vermoeden gebruik je compositie om objecten met meerdere klassen op te bouwen.

Stel dat je allerlei gegevens over je gebruiker wil opslaan en beheren, zoals adresgegevens en profielgegevens (afbeelding, weergavenaam, website, etc.).

Je kunt de `Gebruiker` klasse uitbreiden met allerlei attributen. Maar dit maakt de klasse erg uitgebreid (en daarmee wellicht onoverzichtelijk). En wat als je in je systeem ook een `Factuur` klasse hebt, die precies dezelfde adresgegevens heeft? Soms is het dus beter om klassen op te splitsen. In dit voorbeeld maak je een (vereenvoudigde) klasse `Adres`.

```
1 class Adres:
2
3     def __init__(self, postcode, huisnummer):
4         self.postcode = postcode
5         self.huisnummer = huisnummer
6
7     def verkrijg_compleet_adres(self):
8         """
9         Haal ergens op basis van postcode en huisnummer
10        het volledige adres op
11        """
12        pass
```

Vervolgens breidt je de `Gebruiker` klasse uit met deze adresgegevens:

```
1 class Gebruiker:
2     _rol = "Gebruiker"
3
4     def __init__(self, naam, email, adres):
5         self._naam = naam
6         self._email = email
7         self._adres = adres
8
9     # ...
10
11    def verkrijg_adres(self):
12        return self._adres.verkrijg_compleet_adres()
```

De `__init__` methode is uitgebreid met `adres`, een instantie van `Adres`. Verder is er een methode `verkrijg_adres` toegevoegd, die niets anders doet dan een methode van `Adres` aanroepen. Dit is zeker niet verplicht, maar is wel gebruikelijk. Iemand die `Gebruiker` in de code gebruikt, hoeft zich zo niet druk te maken over waar de adresgegevens vandaan komen.

```
1 adres = Adres("1234AB", "1")
2 gebruiker = Gebruiker("Maria", "maria@example.com", adres)
3
4 gebruiker._adres.verkrijg_compleet_adres()  # Dit mag
5 gebruiker.verkrijg_adres()  # Maar dit heeft de voorkeur
```

Overerving en compositie zijn twee belangrijke concepten binnen het **objectgeoriënteerd programmeren**. Een goede manier om er over na te denken is:

**Overerving**
De relatie is van het type "is een". Een Beheerder *is een* Gebruiker.

**Compositie**
De relatie is van het type "heeft een". Een Gebruiker *heeft een* Adres.

### Opdracht 3: Profiel

Breidt de klasse `Gebruiker` uit met een profiel. Hanteer hierbij compositie, dus vergelijkbaar met het adres. Om het kort te houden bevat het profiel alleen een *bio* (een stukje over de gebruiker).

Zorg ervoor dat de *bio* via een methode opgehaald én aangepast kan worden *vanuit de gebruiker*.

▼ *Klik om het antwoord te tonen*

Een mogelijke uitwerking is (de klasse `Gebruiker` is kort gehouden):

```
1 class Profiel:
2
3     def __init__(self, bio):
4         self._bio = bio
5
6     def verkrijg_bio(self):
7         return self._bio
8
9     def update_bio(self, nieuwe_tekst):
10        self._bio = nieuwe_tekst
11
12
13 class Gebruiker:
14     _rol = "Gebruiker"
15
16     def __init__(self, profiel):
17         self._profiel = profiel
18
19     def bio(self):
```

```
        return self._profiel.verkrijg_bio()

    def update_bio(self, nieuwe_tekst):
        self._profiel.update_bio(nieuwe_tekst=nieuwe_tekst)


johns_profiel = Profiel(bio="Lorem Ipsum")
john = Gebruiker(profiel=johns_profiel)

print(john.bio())  # Lorem Ipsum

john.update_bio(nieuwe_tekst="Ipsum Lorem")
print(john.bio())  # Ipsum Lorem
```

## Verschillende typen methodes

Naast instantie methodes, welke je het meest zult gebruiken en die je tot dusver hebt gebruikt, zijn er nog andere soorten methodes beschikbaar in een klasse.

**Instantiemethode** Een instantiemethode gebruik je om met (data van) instanties te werken. Deze methodes zijn enkel beschikbaar via een instantie (`gebruiker.info()`) en hebben altijd als eerste argument een verwijzing naar de instantie (`self`).

**Klassemethode** Een klassemethode is een methode die je kunt aanroepen vanaf de klasse zelf. Je hebt er ook toegang mee tot attributen die tot de klasse behoren, zoals `_rol`. Je dient de methode te markeren als klassemethode door er `@classmethod` boven te plaatsen. In plaats van `self` als eerste argument, gebruik je `cls` als eerste argument. Bijvoorbeeld bij `Gebruiker`:

```
class Gebruiker:
    _rol = "Gebruiker"

    @classmethod
    def verkrijg_rol(cls):
        return cls._rol

john = Gebruiker()
john.verkrijg_rol()

Gebruiker.verkrijg_rol()
```

Naast dat je de methode vanaf de klasse kunt aanroepen, kan dit ook vanuit de instantie. Klassemethodes zijn nuttig als je iets wil doen dat vooraf gaat aan aanmaken van instanties. Ook worden ze vaak gebruikt bij *constructors*, bijvoorbeeld bij Gebruiker:

```
class Gebruiker:
    _rol = "Gebruiker"

    def __init__(self, naam, email):
        self._naam = naam
```

```
        self._email = email

    def info(self):
        return f"""
        Naam: {self._naam}\n
        E-mail: {self._email}\n
        Rol: {self._rol}\n
        """

    @classmethod
    def maak_anonieme_gebruiker(cls):
        return Gebruiker(naam="John Doe", email="johndoe@example.com")


john_doe = Gebruiker.maak_anonieme_gebruiker()
print(john_doe.info())
```

**Statische methode** Tot slot zijn er statische methodes, die geen toegang hebben tot attributen. Je markeert een methode als statisch door er `@staticmethod` boven te noteren. Verder heeft het geen `self` of `cls` als argumenten, omdat het geen (directe) toegang tot attributen heeft.

Een mogelijke situatie waarbij het logisch kan zijn om het wel te gebruiken is de methode `voeg_gebruiker_toe` die je in opdracht1 hebt toegevoegd. Deze methode heeft geen toegang nodig tot attributen van de beheerder, maar is wel een methode die duidelijk tot een beheerder behoort (reguliere gebruikers hebben de methode niet).

In de praktijk zul je vaker zien dat `voeg_gebruiker_toe` een reguliere functie is, die op een andere manier beperkt is zodat alleen beheerders er toegang toe hebben. Maar het is dus goed mogelijk om het als statische methode aan `Beheerder` toe te voegen.

```
class Beheerder:
    _rol = "Beheerder"

    @staticmethod
    def voeg_gebruiker_toe(naam, email):  # Geen self!
        return Gebruiker(naam=naam, email=email)
```

**Speciale methode** Een klasse heeft ook allerlei speciale methodes, vaak *dundermethodes* genoemd, omdat ze zijn omvat in *double underscores*. Een ben je er al tegengekomen: `__init__`. Andere voorbeelden zijn `__new__`, welke wordt aangeroepen bij het aanmaken van een nieuwe instantie van een klasse (nog voor `__init__`) en `__add__` wat bepaalt hoe optelling werkt voor de gegeven klasse.

Zie de python documentatie voor veelvoorkomende speciale methodes en hun werking.

## Wanneer gebruik je klassen?

Een klasse is een belangrijke pijler in het objectgeoriënteerd programmeren (zie kader). Werken met klassen heeft een aantal voordelen:

- Inkapseling (*encapsulation*): klassen houden data (via attributen) en de methodes om met die data te werken in een enkel object bij elkaar.

- Abstractie: klassen abstraheren complex gedrag weg. Een gebruiker van de klasse hoeft enkel te weten welke methode het moet aanroepen, niet hoe die methode werkt.
- Hergebruik: met behulp van overerving en compositie maak je het mogelijk delen van je code te hergebruiken, zodat herhaling voorkomen wordt.
- Consistentie: door je klassen goed in te richten met de juiste methodes om je data te bewerken, zorg je ervoor dat je gegevens altijd in een geldige staat zijn.
- Organisatie: Door inkapseling, abstractie en hergebruik zijn klassen een goede manier om je code te organiseren op een begrijpelijke wijze.

Tot slot zijn klassen voor veel mensen een 'natuurlijke' manier om over zaken na te denken. Vaak verwijst een klasse naar iets uit de echte wereld (een gebruiker, bijvoorbeeld). 'In het echt' heeft de gebruiker ook eigenschappen (naam, e-mail) en wil je iets doen met die gegevens.

Klassen kunnen dus zeer nuttig zijn om je code overzichtelijk en begrijpelijk te houden, maar dat wil niet zeggen dat het *altijd* de beste oplossing is. Werken met klassen kan ook nadelen hebben. Als er (te)veel abstractielagen in het systeem zijn, dan wordt het wellicht juist ingewikkelder om de code te begrijpen. Soms is een goede functie alles wat je nodig hebt!

Toch zul je klassen veel tegenkomen, ook als je besluit ze zelf niet toe te passen. Het is daarom goed om te begrijpen hoe ze werken.

Objectgeoriënteerd programmeren is een stijl van programmeren waarbij je gebruik maakt van *objecten* om je programma op te bouwen. Een object omvat de gegevens en de mogelijkheid om die gegevens te verwerken.

Een klasse vertegenwoordigd in dit systeem een object. De attributen bevatten de gegevens, met de methodes verwerk je de gegevens.

Lees op Wikipedia meer over object georiënteerd programmeren.

# Werken met bestanden

## Inleiding

In dit hoofdstuk leer je over het werken met bestanden. Werken met bestanden zul je vaak doen in je programma's. Een bestand kan van alles zijn, maar vaak zal het gaan om tekstbestanden. Daar ligt dan ook de focus op in dit hoofdstuk.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je hoe je een bestand opent en leest
- Begrijp je hoe je naar een bestand schrijft

## Werken met bestanden

Een bestand kan van alles zijn. Een eenvoudig tekstbestand, een Wordbestand, een afbeelding, een CSV-bestand, et cetera. Een bestand kan lokaal zijn opgeslagen, maar het kan net zo goed ergens op het internet staan.

Het is teveel om op alle mogelijkheden in te gaan. In dit hoofdstuk leer je de basis aan de hand van het werken met lokaal opgeslagen tekstbestanden.

### Bestanden openen

Er zijn verschillende acties die je kunt uitvoeren met een tekstbestand. Voordat je ermee kunt werken, dien je het eerst te openen. Vergelijk het met een Wordbestand: wil je dit lezen of er tekst aan toevoegen, dan open je het ook eerst. Waar je dit bij het Wordbestand doet door te dubbelklikken op de bestandsnaam, open je in Python een bestand met de ingebouwde functie `open()`.

Een moeilijkheid bij het openen van bestanden is dat je de juiste codering (*encoding*) moet kiezen wil je het bestand correct weergeven. Om moeilijkheden te voorkomen, maak je nu eerst zelf een bestand aan.

```
1 f = open("voorbeeld.txt", mode="wt", encoding="utf-8")
2 f.close()
```

Je leert hierna wat dit precies doet, voor nu is het alleen belangrijk te weten dat het bestand `voorbeeld.txt` wordt opgeslagen naast het bestand vanuit waar je deze code uitvoert. Heb je in Thonny deze code in `main.py` in een mapje 'Programmeren met Python` staan, dan zal `voorbeeld.txt` dus ook in het mapje 'Programmeren met Python' staan.

Werk vanaf nu verder in een Python-bestand dat in de map staat waar je `voorbeeld.txt` hebt aangemaakt.

### Naar tekstbestanden schrijven

Je hebt nu een leeg tekstbestand. Tijd om er tekst aan toe te voegen.

```
1 f = open("voorbeeld.txt", mode="wt", encoding="utf-8")
2 f.write("Dit is mijn eerste zin in een tekstbestand.\n")
3 f.write("Dit is mijn laatste zin.")
4 f.close()
```

Op regel 1 zie je weer dezelfde aanroep naar `open()`. Het eerste argument is de bestandsnaam. In dit geval is het alleen de naam, maar het kan ook een volledig pad zijn naar een bestand.

Het tweede argument, hier expliciet gemaakt, is de modus (*mode*). Open je een bestand, dan dien je op te geven wat je ermee wilt doen en wat voor bestand het is. De eerste letter geeft aan of je het wilt lezen (*read*), schrijven (*write*) of beide. In dit geval gebruiken we `w`, waarmee je een bestand zowel kunt lezen als schrijven. Bestaat het bestand nog niet, dan zal het worden aangemaakt. De tweede letter geeft aan of het een tekstbestand betreft of een binair bestand (waarover later meer).

| | |
|---|---|
| a | Voeg tekst toe aan het einde van het bestand. |
| w | Voeg tekst toe vanaf begin van bestand. Bestaande tekst wordt eerst verwijderd. |
| r | Lees de tekst, vanaf het begin. |

Met het laatste argument, `encoding`, geef je de codering van het bestand op. Je hebt het bestand aangemaakt in `utf-8`, dus open je het nu ook weer met die codering.

Op regels 2 en 3 van het bestand schrijf je nu twee regels naar het tekstbestand. Let op de `\n`, je dient zelf eventuele *linebreaks* toe te voegen.

Als je klaar bent met het werken met een bestand, moet je het altijd sluiten. Dit doe je met `.close()`.

Voer deze code uit en open `voorbeeld.txt`, bijvoorbeeld met Kladblok in Windows om te controleren of de tekst inderdaad is toegevoegd.

### Opdracht 1: Toevoegen

Met *mode* `wt` schrijf je naar een bestand. Als het nog niet bestaat, wordt het bestand aangemaakt. Bestaat het al wel, dan *overschrijf* je het, eventuele bestaande tekst wordt dus overschreven.

Open nu nogmaals het bestand `voorbeeld.txt` en voeg de tekst "Nieuwe laatste zin." toe zonder de al bestaande tekst te overschrijven.

▼ *Klik om het antwoord te tonen*

Een uitwerking is:

```
1 f = open("voorbeeld.txt", mode="at", encoding="utf-8")
2 f.write("\nNieuwe laatste zin.")
3 f.close()
```

Let op dat je `\n` toevoegt voor je tekst om de tekst op een nieuwe regel te laten beginnen.

## Tekstbestanden lezen

Nu je een bestand hebt met tekst, kun je het ook lezen. Je gebruikt weer `open()` om het bestand te openen, maar nu in *read-only* modus:

```
1 f = open("voorbeeld.txt", mode="rt", encoding="utf-8")
2 tekst = f.read()
3 print(tekst)
```

```
4 f.close()
```

Met `.read()` lees je het hele bestand in één keer in. In dit geval geen probleem, zo veel tekst is het niet.

Met `read()` kun je ook kiezen hoeveel tekens je wilt inlezen. Na het inlezen wordt de 'cursor' als het ware verplaatst na het laatst ingelezen teken. Roep je daarna nog een keer `read()` aan, dan lees je de rest van de tekst in.

```
1 f = open("voorbeeld.txt", mode="rt", encoding="utf-8")
2 tekst = f.read(43)  # De eerste zin bestaat uit 43 tekens
3 print(tekst)
4
5 tekst = f.read()  # Lees de rest van het bestand
6 print(tekst)
7
8 f.close()
```

### Opdracht 2: Per 50

Schrijf code dat `voorbeeld.txt` opent en 2 keer `.read(50)` aanroept. Wat is het resultaat?

▼ *Klik om het antwoord te tonen*

Een uitwerking is:

```
1 f = open("voorbeeld.txt", mode="rt", encoding="utf-8")
2 print(f.read(50))
3 print(f.read(50))
4 f.close()
```

```

```

De eerste aanroep naar `.read(50)` lees de eerste 50 tekens in:

```
1 "Dit is mijn eerste zin in een tekstbestand.
2 Dit is"
```

De tweede aanroep naar `.read(50)` leest de volgende 50 tekens in. Er zijn nog minder dan 50 tekens te lezen, maar dat maakt voor de aanroep niet uit:

```
1 " mijn laatste zin.
2 Nieuwe laatste zin."
```

Met `read()` kun je het bestand regel voor regel inlezen, maar daarvoor moet je wel weten hoeveel tekens elke regel bevat. Dat is niet heel handig. Gelukkig is er ook de methode `.readline()`:

```
f = open("voorbeeld.txt", mode="rt", encoding="utf-8")
regel_1 = f.readline()  # Lees de eerste regel
print(regel_1)

regel_2 = f.readline()  # Lees de tweede regel
print(regel_2)

regel_3 = f.readline()  # Lees de derde regel
print(regel_3)

f.close()
```

De `print()`-functie toont de `\n` niet, maar voer je `regel_1` in de shell in, dan zul je het wel zien.

Dit is al een stuk beter, maar het kan nog eenvoudiger. Een bestandobject is namelijk een *iterator*, en dus kun je de `for-loop` gebruiken.

```
f = open("voorbeeld.txt", mode="rt", encoding="utf-8")

for line in f:
    print(line)

f.close()
```

Op deze manier print je alle regels een voor een af.

### Opdracht 3: Kapitalen

Een `for-loop` gebruik je als je met alle regels iets wilt doen. Lees `voorbeeld.txt` nogmaals in en zorg ervoor dat alle regels in kapitalen (hoofdletters) worden geprint. Gebruik hiervoor de `upper()` methode van `str`.

▼ *Klik om het antwoord te tonen*

Een uitwerking is:

```
f = open("voorbeeld.txt", mode="rt", encoding="utf-8")
for line in f:
    print(line.upper())
f.close()
```

## Contextmanagers

Tot nu toe heb je het bestand steeds handmatig gesloten met `f.close()`. Maar omdat elk bestand dat geopend wordt, ook weer gesloten moet worden is er een handigere manier. Met een *contextmanager* zorg je ervoor dat elk bestand dat is geopend met `open()`, ook altijd weer wordt gesloten met `.close()`. Dat ziet er zo uit:

```
with open("voorbeeld.txt", mode="rt", encoding="utf-8") as f:
    for line in f:
        print(line)
```

In plaats van `f = open(⋯)` schrijf je nu `with open(⋯) as f:`, alles daarna is het blok van de contextmanager. In dit geval print je alle regels van het bestand. Omdat je met `with` werkt, hoef je het bestand niet meer expliciet af te sluiten, dit doet de contextmanager voor je. Zo kun je het sluiten niet per ongeluk vergeten.

## Werken met andere (binaire) bestanden

Met `open()` open je niet enkel tekstbestanden, maar ook binaire bestanden (zoals afbeeldingen). Bij het argument `mode` geef je dan `b` mee in plaats van `t`, een *encoding* is in dat geval niet nodig. In het geval van afbeeldingen zul je dit echter vaak met afzonderlijke *packages* doen. Zo is Pillow een veelgebruikte *package* om met afbeeldingen te werken.

Naast *packages* om met binaire bestanden te werken, zijn er ook *packages* om met andere tekstbestanden te werken. Een aantal hiervan zijn aanwezig in de standaard bibliotheek, bijvoorbeeld:

- `csv`, om met CSV-bestanden te werken
- `configparser`, om met `.ini` configuratiebestanden te werken
- `json`, om met JSON-bestanden te werken
- `html`, om met HTML-bestanden te werken

Dit zijn zeker niet de enige *packages* in de standaard bibliotheek, kijk op python.org voor meer!

Daarnaast zijn er ook nog *packages* van derde partijen, zoals de al eerder genoemde Pillow, om met (tekst)bestanden te werken. Een veelgebruikte *package* om met HTML te werken is bijvoorbeeld Beautiful Soup

# Werken met uitzonderingen

## Inleiding

In dit hoofdstuk leer je werken met uitzonderingen. Je zult in je programma's vaak uitzonderingen tegenkomen. Uitzonderingen zijn situaties die optreden als iets - al dan niet verwacht - fout gaat. Bijvoorbeeld: een functie waarmee je twee getallen deelt, zal een uitzondering opwerpen als je probeert te delen door 0. Als je in de vorige hoofdstukken fouten tegenkwam, dan stopte je programma. In dit hoofdstuk leer je hoe je fouten netjes afhandelt.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je wat een uitzondering is in Python
- Begrijp je hoe je uitzonderingen gebruikt om het programmaverloop te bepalen
- Begrijp je hoe je uitzonderingen afvangt en opwerpt

## Werken met uitzonderingen

Als je programmeert kunnen er fouten ontstaan. Veel van dit soort fouten zijn gelukkig te voorspellen zodat je ze van tevoren al kunt voorkomen of afhandelen. Zo'n fout onderbreekt als het ware je programmaverloop, en wordt daarom vaak een *uitzondering* genoemd. In deze paragraaf leer je werken met dergelijke uitzonderingen.

### Wat is een uitzondering?

Wat een uitzonderlijke situatie is, is contextafhankelijk en is ook een kwestie van perceptie. Een voorbeeld kan zijn dat je een bestand probeert te openen dat niet bestaat. In je programma ga je er vanuit dat het bestaat, misschien omdat je het zelf hebt aangemaakt. Als het dan toch niet bestaat, is dat met recht een uitzonderlijke situatie te noemen.

Bekijk maar eens wat er gebeurt als je een niet-bestaand bestand wilt open:

```
1 f = open("ik-besta-niet.txt", mode="rt", encoding="utf-8")
2
3
4 Traceback (most recent call last):
5   File "/home/erwin/programmeren-met-python/main.py", line 1, in <module>
6     f = open("ik-besta-niet.txt", mode="r", encoding="utf-8")
7 FileNotFoundError: [Errno 2] No such file or directory: 'ik-besta-niet.txt'
```

Voer je dit uit in Thonny, dan zie je in de shell een soortgelijke **traceback** als in dit voorbeeld. Het belangrijkste zie je op regel 7: `FileNotFoundError`. Dit maakt duidelijk welke fout er is ontstaan: het bestand is niet gevonden. Voor meer informatie lees je deze *traceback* helemaal. Zo lees je op regel 5 dat de fout zich voordeed op regel 1 in het bestand `main.py`. Op regel 6 lees je welke stuk code de fout heeft veroorzaakt.

Deze `FileNotFoundError` is één van de vele uitzonderingen - *Exceptions* in het Engels - die Python rijk is.

In de documentatie van Python lees je welke ingebouwde *exceptions* er allemaal zijn.

Een paar voorbeelden:

`IndexError`
: Als je een index probeert op te halen die niet bestaat.

`KeyError`
: Als de opgegeven sleutel niet bestaat in een *dictionary*.

`ValueError`
: Als het opgegeven argument een niet-geldige waarde heeft, bijvoorbeeld als je de wortel van een negatief getal wilt trekken: `math.sqrt(-4)`.

### Opdracht 1: IndexError

Schrijf code dat bewust een `IndexError` veroorzaakt.

▼ *Klik om het antwoord te tonen*

Een uitwerking is:

```
1 mijn_lijst = [1, 2, 3]
2 print(mijn_lijst[3])
```

De laatste index van deze specifieke lijst is 2 (de index begint bij 0, weet je nog?) `mijn_lijst[3]` zal dus niet werken.

## Uitzonderingen om het programmaverloop te bepalen

Tref je een uitzondering in je programma, dan is het niet erg fraai voor de gebruiker als heel je programma stopt en een - voor de gebruiker - cryptische *traceback* opwerpt.

Het is mooier om dergelijke *exceptions* af te handelen en als ze voorkomen een andere richting in je programma te kiezen. Het is hiermee vergelijkbaar met `if-else`: als dit, dan dat. Als het bestand bestaat, open het dan. Anders maken we het aan.

Wanneer gebruik je dan `if-else` en wanneer `try···except···`? Daar zijn geen harde regels voor, maar het wordt als 'Pythonisch' beschouwd om veel met `try···except···` te werken. Een bekende uitrdukking hierin is: "It's easier to ask for forgiveness than permission", afgekort tot EAFP. De tegenhanger is LBYL: "Look before you leap".

De voorkeursstijl in Python is om uit te gaan van de 'happy flow', en actie te ondernemen als je aannames niet kloppen (EAFP). Hier past `try···except···` dus goed bij. De andere stijl is om van tevoren alles proberen af te dekken, met `if-else` (LBYL).

Maar `if-else` is met name bedoeld om keuzes te maken op basis van condities: als de gebruiker A typt, doe dan x, anders y. `try···except···` is meer bedoeld voor die situaties waar de keuze al is gemaakt (open dit bestand), en waar je verwacht dat het meestal ook goed gaat. Gaat het toch fout, dan handel je het af.

### Uitzonderingen afvangen

Dit afhandelen van uitzonderingen doe je met `try.. except ..`, ofwel, je probeert iets en als dat niet lukt (er ontstaat een fout), dan doe je wat anders.

```
1 try:
```

```
    # Open het configuratiebestand
    with open("config.txt", mode="rt", encoding="utf-8") as f:
        print("Je configuratie is...")
except FileNotFoundError:
    print("Configuratie bestaat niet, maak een standaard configuratie aan.")
    with open("config.txt", mode="wt", encoding="utf-8") as f:
        f.write("Standaard configuratie")

        print("Je configuratie is...")

# De rest van je code
```

Op regel 1 start je met `try:`, in het blok eronder staat de code die je wilt uitvoeren. Bestaat het bestand niet, dan wordt dit afgevangen door de `except FileNotFoundError:` op regel 5. Je belandt dan in het tweede blok, waar je het bestand aanmaakt en er iets in zet.

Je programma gaat nu netjes door, ook als het bestand per ongeluk niet bestaat. Maar wat als er een ander type fout ontstaat? Bijvoorbeeld, het bestand bestaat wel maar staat op een plek waar je programma geen leesrechten voor heeft. Dan stop je programma alsnog.

Verwacht je dat deze uitzondering kan voorkomen, dan kun je die ook afvangen.

```
try:
    # Open het configuratiebestand
    with open("config.txt", mode="rt", encoding="utf-8") as f:
        print("Je configuratie is...")
except FileNotFoundError:
    print("Configuratie bestaat niet, maak een standaard configuratie aan.")
    with open("config.txt", mode="wt", encoding="utf-8") as f:
        f.write("Standaard configuratie")

        print("Je configuratie is...")
except PermissionError:
    print("Ik kon het configuratiebestand helaas niet lezen. "
          "Pas de rechten van het systeem aan.")

# De rest van je code
```

Op regel 11 zie je dat er nu ook wordt gecontroleerd op `PermissionError`. Je kunt net zoveel `except`-blokken toevoegen als nodig is.

#### Uitzonderingen opwerpen

In bovenstaande code handel je de fouten af en gaat je programma verder. Er zijn ook situaties waarin je niet weet hoe je de fout moet afhandelen. Stel dat je het in inlezen van een configuratiebestand, zoals hierboven, in een functie hebt geplaatst. Deze functie roep je op verschillende plaatsen in je code aan. Afhankelijk van waar het wordt aangeroepen wil je dat je programma doorgaat, stopt of een waarschuwing geeft.

De functie weet dus niet hoe het de fout moet afhandelen. In dat geval kun je de *exception* opnieuw *opwerpen*.

```
1 def read_config():
2     try:
3         # Open het configuratiebestand
4         with open("config.txt", mode="rt", encoding="utf-8") as f:
5             print("Je configuratie is...")
6     except (FileNotFoundError, PermissionError):
7             raise
```

De aanroeper van `read_config` kan nu zelf bepalen hoe het de fout afhandelt. Overigens zie je dat je op regel 6 op meerdere *exceptions* tegelijk kunt controleren. De `raise` zal de uitzondering die is opgetreden opnieuw opwerpen.

In dit geval is de `try⋯except⋯` niet heel nuttig, omdat de aanroeper ook een `try⋯except⋯` zal gebruiken. Het wordt nuttiger als je voordat je de uitzondering opnieuw opwerpt eerst nog een actie uitvoert. Ook kan het nuttig zijn in de functie op verschillende uitzonderingen te controleren en vervolgens een enkele uitzondering op te werpen.

```
 1 # Definieer een eigen uitzondering
 2 class MijnUitzondering(Exception):
 3     pass
 4
 5
 6 # Definieer de functie om het configuratiebestand te openen
 7 def read_config():
 8     try:
 9         # Open het configuratiebestand
10         with open("config.txt", mode="rt", encoding="utf-8") as f:
11             print("Je configuratie is...")
12     except (FileNotFoundError, PermissionError) as e:
13         # Log de fout
14         print("Fout gelogd")
15
16         # Werp een eigen uitzondering op
17         raise MijnUitzondering(f"Er ging iets niet goed: {e}")
18
19
20 # Roep de functie aan
21 try:
22     read_config()
23 except MijnUitzondering as e:
24     # Handel de fout op de juiste manier af
25     print(e)
```

Op regel 1 maak je een eigen uitzondering aan door een nieuwe `class` te maken die overerft van `Exception`. De uitzondering hoeft verder niets te doen.

In de functie controleer je op regel 12 op verschillende uitzonderingen. Nieuw is het stukje `as e`. Dit maakt het mogelijk om de uitzondering die optreedt opnieuw te gebruiken.

Op regel 14 log je de fout en op regel 17 werp je niet de originele uitzondering op, maar je eigen uitzondering. Het foutbericht bevat het originele foutbericht, die je krijgt door `e` mee te geven.

Roep je nu de functie aan, dan kun je controleren op `MijnUitzondering` en het op de juiste manier afhandelen.

### Opdracht 2: Uitzondering

Wat verwacht je dat `print(e)` op regel 25 toont?

▼ *Klik om het antwoord te tonen*

Omdat je eerder `raise MijnUitzondering(f"Er ging iets niet goed: {e}")` hebt gebruikt, zal de `e` op regel 25 die uitzondering met dat bericht bevatten. In dit geval bestaat `config.txt` niet en wordt de complete foutmelding dus:

```
Er ging iets niet goed: [Errno 2] No such file or directory: 'config.txt'
```

#### Opruimen

Het kan nodig zijn een bepaalde actie uit te voeren, ongeacht of de code in het `try`-blok wel of niet is gelukt. Een voorbeeld kan zijn om een geopend bestand altijd te weer te sluiten (ter illustratie, in de praktijk zul je hiervoor `with` gebruiken).

```
1 try:
2     f = open("ik-besta-niet.txt", mode="r", encoding="utf-8")
3     # Doe iets met de waarden in het bestand
4 except ValueError as e:
5     print(e)
6     raise
7 finally:
8     f.close()
```

Ook al bevat het bestand nu een waarde waar je niets mee kunt en wordt er een `ValueError` opgeworpen, het bestand zal eerst nog gesloten worden in het `finally`-blok.

# Je code testen

## Inleiding

Programmeren is niet alleen het schrijven van je programma, maar er ook voor zorgen dat je dit met vertrouwen doet. Het testen van je code helpt hierbij. Elke keer dat je je code uitvoert om te kijken of het werkt, test je handmatig je code. Alhoewel nuttig tijdens het ontwikkelen, is het niet erg compleet en efficiënt.

Je kunt je code ook *geautomatiseerd* testen. Voordelen daarvan zijn dat je snel allerlei scenario's kunt testen. Ook kun je de testen gebruiken om ervoor te zorgen dat wanneer je je code aanpast, het nog steeds het gewenste resultaat oplevert.

In dit hoofdstuk leer je werken met de module *unittest* uit de standaard bibliotheek om je code geautomatiseerd te testen.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Begrijp je wat een unittest is en waar het toe dient
- Begrijp je hoe je een unittest schrijft
- Begrijp je hoe je met testdata werkt

## Geautomatiseerd testen

In deze paragraaf leer je enkele concepten behorende bij het geautomatiseerd testen: de verschillende niveaus van testen, wat ***fixtures*** zijn en wat ***assertions*** zijn.

### Niveaus van testen

In het programmeren worden over het algemeen vier niveaus van testen onderscheiden.

**Unittest**

Een *unittest* richt zich op het testen van individuele componenten van je code, zoals functies en methoden. Heb je bijvoorbeeld een functie `kwadraat()`, dan is een mogelijke *unittest* de test of het resultaat 4 is als je als argument 2 opgeeft. Je weet zo dat je functie het juist resultaat levert.

*Unittests* zul je veel gebruiken tijdens het schrijven van je code. Elke keer als je een functie of methode maakt, test je dit. In de praktijk zul je zelfs *unittests* gebruiken om je functie vorm te geven. Er bestaan zelfs aanhangers van *Test Driven Development (TDD)*: je gebruikt testen om de gewenste implementatie te bereiken.

**Integratietest**

Integratietesten testen hoe verschillende onderdelen van je software met elkaar samenwerken. Dit niveau test het resultaat wanneer je verschillende onderdelen in je code combineert.

Je testen focussen zich dus niet op één functie of methode, maar op het samenspel ertussen. Om een bepaald resultaat te bereiken heb je vaak meerdere functies nodig. Met een integratietest test je de hele *flow*.

**Systeemtest**

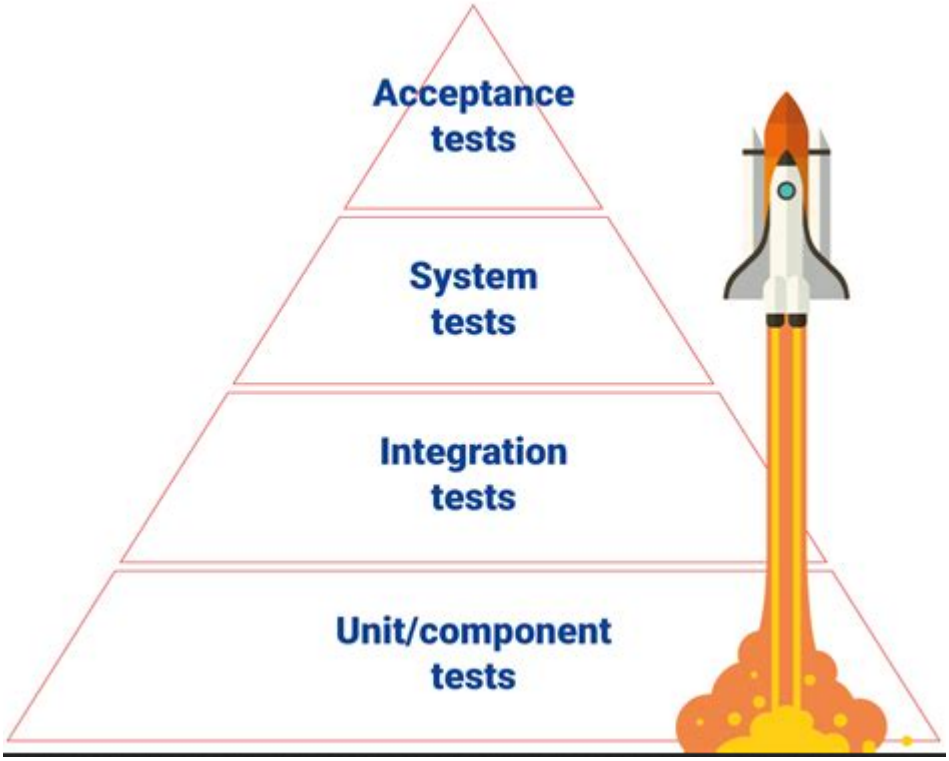
Deze testen richten zich op het valideren van het volledige systeem of de applicatie als geheel. Ze testen of het systeem aan de gestelde eisen voldoet en correct werkt in de beoogde omgeving.

**Acceptatietest**

Acceptatietesten worden uitgevoerd om te bepalen of het systeem voldoet aan de vereisten van de eindgebruiker. Dit kan worden gedaan door zowel de opdrachtgever als de gebruikers om ervoor te zorgen dat de software aan hun verwachtingen voldoet voordat deze in productie wordt genomen.

In dit hoofdstuk leer je werken met het kleinste blokje: de *unittest*. Dit omdat je als programmeur het meest zult gebruiken en vrij eenvoudig is uit te breiden naar integratietesten.

De systeemtesten en acceptatietesten zijn vaak de verantwoordelijkheid van andere collega's.




## Fixtures

Wanneer je gaat testen wil je zeker weten dat elke keer dat je de test herhaalt, je hetzelfde test. Dit kan betekenen dat bepaalde instellingen gedaan moeten zijn, dat een bepaald bestand aanwezig is of dat een bepaalde waarde in je database bestaat.

*Fixtures* zijn vooraf zijn vooraf ingestelde situaties die voor of na elke test worden uitgevoerd om de gewenste situatie te bereiken. In het werken met de module *unittest* in de volgende paragraaf zul je leren hoe je dit doet met de methodes `setUp()` en `tearDown()`.

## Assertions

Het laatste belangrijke concept bij testen is de *assertion*. In Python zijn dit statements die je gebruikt om beweringen te controleren. Je kunt hiervoor het ingebouwde `assert` gebruiken. Voer je onderstaande code uit, dan gebeurt er niets:

```
1 x = 5
2 assert x == 5
```

`x` is inderdaad 5, dus het statement `x == 5` is waar en `assert` geeft geen melding. Verander je de code zodat het statement onwaar wordt, dan krijg je een `AssertionError`.

```
1 x = 6
2 assert x == 5
```

```
Traceback (most recent call last):
  File "/home/erwin/programmeren-met-python/main.py", line 2, in <module>
    assert x == 5
AssertionError
```

In de module *unittest* zul je meestal niet direct `assert` gebruiken, maar ingebouwde methodes als `assertEqual` (test of twee waarden gelijk zijn) of `assertTrue` (test of iets waar is). Dit zijn schillen om `assert` heen die je testen kunnen verduidelijken.

## Werken met de module unittest

In deze paragraaf leer je werken met de ingebouwde module `unittest`. Maak eerst twee bestanden aan: `main.py`, die de code die je wilt testen zal bevatten en `test.py`, die de testen zal bevatten. Plaats de bestanden naast elkaar in dezelfde map.

### test.py

Zorg ervoor dat `test.py` er als volgt uitziet:

```
1 import unittest
2
3
4 if __name__ == "__main__":
5     unittest.main()
```

Voer je dit bestand nu uit, dan zul je de volgende output krijgen:

```
----------------------------------------------------------------------
Ran 0 tests in 0.000s

OK
```

En dit klopt, want je hebt nog geen testen geschreven. Een test toevoegen vraagt twee stappen. Ten eerste maak je een klasse aan die overerft van `unittest.TestCase`. Voeg onderstaande toe aan `test.py`:

```
1 class MijnTest(unittest.TestCase):
2     pass
```

Door een klasse te maken die overerft van `TestCase`, krijg je een aantal handige methodes tot je beschikking, zoals het werken met fixtures en de *assertions*. Over het algemeen zul je voor elke functie één **testcase** aanmaken en daar verschillende subtesten mee uitvoeren.

De daadwerkelijke testen voeg je toe door methodes aan de testcase toe te voegen die beginnen met `test_`:

```python
class MijnTest(unittest.TestCase):

    def test_mijn_eerste_test(self):
        pass

    def test_mijn_tweede_test(self):
        pass
```

Voer je `test.py` nu uit, dan zie je dat er twee testen zijn uitgevoerd. Aangezien ze nog niets doen, slagen ze ook allemaal.

```
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

### main.py

Voordat je verdergaat met testen, heb je code nodig om te testen. Voeg de volgende code toe aan `main.py`:

```python
def get_formatted_name(first, last):
    """ Maak een netjes opgemaakte naam. """

    full_name = f"{first} {last}"
    return full_name.title()
```

Zorg dat je begrijpt wat het doet en voer de functie een paar keer uit met verschillende variaties op namen (met en zonder hoofdletters, bijvoorbeeld).

### Je eerste echte test

Pas `test.py` aan zodat je nu je code kunt testen. Ten eerste importeer je `get_formatted_name`. Pas ook de naam van de klasse aan naar iets duidelijkers, net als de naam van de eerste test.

```python
import unittest

from main import get_formatted_name


class NamenTest(unittest.TestCase):
    """ Testen voor `get_formatted_name`. """

    def test_first_last_name(self):
        """ Werken namen zoals 'John Doe'? """
        verwacht = "John Doe"
        resultaat = get_formatted_name("john", "doe")
        self.assertEqual(resultaat, verwacht)
```

```
14
15
16      def test_mijn_tweede_test(self):
17          pass
18
19  if __name__ == "__main__":
20      unittest.main()
```

Vanaf regel 9 zie je de eerste test. De opbouw is hoe je een test vaak aanpakt:

- Je noteert de verwachte uitkomst, in dit geval "John Doe"
- Je voert de functie uit en haalt het resultaat op
- Je controleert of het resultaat overeenkomt met de verwachte uitkomst

Voer je `test.py` uit, dan zie je dat er weer twee test zijn uitgevoerd en dat het resultaat 'OK' is. De tweede test doet nog niets, maar de eerste test wel. Je test is dus geslaagd!

```
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

### Opdracht 1: Kapitale test

Pas de tweede test aan zodat het test of de functie ook goed werkt als je "JOHN" en "DOE" als argumenten opgeeft.

Welke test kun je nog meer verzinnen?

▼ *Klik om het antwoord te tonen*

Een uitwerking is:

```
1  import unittest
2
3  from main import get_formatted_name
4
5
6  class NamenTest(unittest.TestCase):
7      """ Testen voor `get_formatted_name`. """
8
9      def test_first_last_name(self):
10         """ Werken namen zoals 'John Doe'? """
11         verwacht = "John Doe"
12         resultaat = get_formatted_name("john", "doe")
13         self.assertEqual(resultaat, verwacht)
14
15     def test_first_last_name_capital(self):
16         """ Werken namen in kapitalen? """
```

```
        verwacht = "John Doe"
        resultaat = get_formatted_name("JOHN", "DOE")
        self.assertEqual(resultaat, verwacht)


if __name__ == "__main__":
    unittest.main()
```

Een mogelijke andere test die je kunt toevoegen is of "John", "Doe" werkt, dus waar de namen al netjes beginnen met een hoofdletter.

## Een falende test

Je hebt nu een werkende functie en een aantal testen om te testen of de code inderdaad werkt zoals verwacht. Maar ineens bedenk je: wat als deze functie wordt gebruikt in een ander stuk code, waar de gebruiker wordt gevraagd om zijn naam op te geven. En de gebruiker geeft niet twee namen, maar drie namen op. Werkt de code dan ook? Dit klinkt als een goede test om te schrijven.

Voeg de volgende test toe aan de testcase:

```
class NamenTest(unittest.TestCase):

    # ...

    def test_three_names(self):
        """ Werken namen zoals 'John Tweedenaam Doe'? """
        verwacht = "John Tweedenaam Doe"
        resultaat = get_formatted_name("john", "tweedenaam", "doe")
        self.assertEqual(resultaat, verwacht)
```

Voer het bestand `test.py` uit en zie het resultaat, wat ongeveer als volgt zal zijn:

```
..E
======================================================================
ERROR: test_three_names (__main__.NamenTest)
Werken namen zoals 'John Tweedenaam Doe'?
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/erwin/programmeren-met-python/test.py", line 24, in test_three_names
    resultaat = get_formatted_name("john", "tweedenaam", "doe")
TypeError: get_formatted_name() takes 2 positional arguments but 3 were given

----------------------------------------------------------------------
Ran 3 tests in 0.000s

FAILED (errors=1)
```

Helemaal bovenaan zie je `..E`, zodat je direct ziet dat er in dit geval drie testen zijn uitgevoerd waarvan er 1

faalde. Helemaal onderaan zie je ongeveer hetzelfde: er zijn drie testen uitgevoerd, 1 is gefaald met een fout. Dit laatste vertelt je dat er een fout is ontstaan in de code. Had er gestaan `FAILURES (failures=1)`, dan werkte de code wel maar was het resultaat anders dan verwacht.

In dit geval ontstaat er dus een fout in je code. In de *traceback* lees je wat er fout gaat, en waar: er ontstaat een `TypeError` omdat er drie argumenten aan de functie zijn meegegeven, terwijl er maar twee geaccepteerd worden.

### Opdracht 2: Verbeter de code

Pas de functie `get_formatted_name` aan zodat het ook werkt als je drie namen - voornaam, tweede naam, achternaam - wilt opmaken. Gebruik de testcase om te controleren of alle situaties (blijven) werken, dus ook als iemand twee namen opgeeft.

▼ *Klik om het antwoord te tonen*

Een uitwerking is:

```
def get_formatted_name(first, last, middle=""):
    """ Maak een netjes opgemaakte naam. """

    if middle:
        full_name = f"{first} {middle} {last}"
    else:
        full_name = f"{first} {last}"

    return full_name.title()
```

Zorg dat de tweede naam, `middle` in dit geval, optioneel is door het een standaard waarde te geven. Omdat optionele argumenten altijd achter verplichte argumenten moeten komen, plaats je `middle` aan het eind. In de functie controleer je vervolgens op of `middle` is ingevuld.

De testcase is ook iets aangepast. Voor de duidelijkheid kun je de functie nu overal aanroepen met sleutelwoord argumenten. De laatste test moet je ook nog aanpassen. In de eerste versie riep je `get_formatted_name("john", "middle", "doe")` aan. Dit moet worden: `get_formatted_name(first="john", last="doe", middle="middle")`.

De volledige testcase ziet er nu als volgt uit:

```
import unittest

from main import get_formatted_name


class NamenTest(unittest.TestCase):
    """ Testen voor `get_formatted_name`. """

    def test_first_last_name(self):
        """ Werken namen zoals 'John Doe'? """
        verwacht = "John Doe"
        resultaat = get_formatted_name(first="john", last="doe")
        self.assertEqual(resultaat, verwacht)
```

```
14
15      def test_first_last_name_capital(self):
16          """ Werken namen in kapitalen? """
17          verwacht = "John Doe"
18          resultaat = get_formatted_name(first="JOHN", last="DOE")
19          self.assertEqual(resultaat, verwacht)
20
21      def test_three_names(self):
22          """ Werken namen zoals 'John Middle Doe'? """
23          verwacht = "John Middle Doe"
24          resultaat = get_formatted_name(first="john", last="doe", middle=
    "middle")
25          self.assertEqual(resultaat, verwacht)
26
27
28  if __name__ == "__main__":
29      unittest.main()
```

## Uitzonderingen testen

Stel dat iemand de functie `get_formatted_name` aanroept met iets anders dan een tekst voor één van de namen, bijvoorbeeld een getal. Je kunt er dan vanuit gaan dat dat een fout is, niemand zal "1 Jansen" heten.

Je kunt de functie aanpassen door hier op te controleren en een `ValueError` op te werpen als één of meer van de namen geen tekst is. Dat ziet er als volgt uit:

```
1  def get_formatted_name(first, last, middle=""):
2      """ Maak een netjes opgemaakte naam. """
3
4      all_strings = all(
5          [isinstance(first, str),
6           isinstance(last, str),
7           isinstance(middle, str)]
8      )
9
10     if not all_strings:
11         raise ValueError("Alle namen dienen tekst te zijn")
12
13     if middle:
14         full_name = f"{first} {middle} {last}"
15     else:
16         full_name = f"{first} {last}"
17
18     return full_name.title()
```

Op regel 4 gebruik je `all()` om te kijken of alle argumenten van het type `str` zijn. Als dit zo is, dan zal `all_strings` waar zijn (`True`). Is één of meer van de argumenten geen `str`, dan evalueert `all` tot `False`.

Daarna controleer je of `all_strings` waar is en zo niet, werp je een `ValueError` met een bericht op. Dit wil je ook testen. Dat doe je als volgt:

```
1 class NamenTest(unittest.TestCase):
2
3     # ...
4
5     def test_not_str(self):
6         with self.assertRaises(ValueError):
7             get_formatted_name(first=1, last="doe", middle="middle")
```

In de test zorg je er bewust voor dat de `ValueError` wordt opgeworpen door het eerste argument een `int` mee te geven. Vervolgens controleer je met `with self.assertRaises(ValueError):` of de `ValueError` inderdaad wordt opgeworpen.

### Opdracht 3: Meer testgevallen

Je hebt nu een werkende functie met een aantal testen. Maar werken met namen is *tricky*! Het is altijd goed om na te denken over alle denkbare situaties, inclusief *edge cases*: situaties die zeer uitzonderlijk zijn maar wel kunnen voorkomen.

Bedenk nog meer zaken die fout zouden kunnen gaan en waar je op kunt testen. Je mag de testen uitschrijven, maar dit hoeft niet.

▼ *Klik om het antwoord te tonen*

Werken met namen is *tricky*. Hier zou je allemaal rekening mee kunnen houden:

- Iemand heeft een dubbele voornaam (geen tweede naam)
- Iemand heeft een dubbele achternaam
- Iemand heeft meerdere voornamen en/of tweede namen
- In veel landen zijn voor- en achternaam omgedraaid, houd je daar rekening mee?
- Iemand heeft een naam met bijzondere karakters
- Iemand heeft (nog) geen naam
- Iemands naam is in kapitalen
- Iemand naam is in allemaal kleine letters
- …

Deze lijst is zeker niet compleet. Namen zijn ingewikkeld, zeker als je namen gaat ontvangen die een herkomst hebben uit andere landen of culturen. De beste oplossing is daarom misschien nog wel: vraag in één veld om een naam en hoe de gebruiker het ook opgeeft, zo geef je het weer.

## Werken met fixtures

Elke keer dat je een test uitvoert, wil je wel dat je exact hetzelfde test. Stel dat je een functie hebt dat het aantal gebruikers uit een database ophaalt. Om te testen of dit correct werkt, is het van belang dat je in je test weet hoeveel gebruikers er in je (test)database aanwezig zijn, zodat je `self.assertEqual(verwacht, resultaat)` kunt uitvoeren. Het is dus zaak dat in dit geval je testdatabase voorafgaande aan elke test precies in dezelfde staat verkeert. Deze gegevens of instellingen die je als uitgangspunt wilt nemen noem je *fixtures*.

### setUp

Met de methode `setUp()` in je testcase zet je je *fixtures* klaar. De methode wordt voor elke afzonderlijke test aangeroepen en zorgt zo dus dat je gegevens voor elke test weer opnieuw klaar staan.

```
import unittest

class ListTestCase(unittest.TestCase):
    def setUp(self):
        self.mijn_list = [1, 2, 3]

    def test_sum(self):
        result = sum(self.mijn_list)
        self.assertEqual(result, 6)

    def test_append(self):
        result = self.mijn_list.append(4)
        self.assertEqual(result, [1, 2, 3, 4])

    def test_length(self):
        result = len(self.mijn_list)
        self.assertEqual(result, 3)

if __name__ == '__main__':
    unittest.main()
```

In dit voorbeeld begin je elke test met dezelfde lijst: `self.mijn_list = [1, 2, 3]`. Dit is niet alleen handig omdat je niet steeds een lijst hoeft uit te typen. Het is vooral belangrijk omdat je in `test_append` je data wijzigt (je voegt iets toe). Test je daarna de lengte van je lijst, dan weet je zeker dat dit op 3 uitkomt, omdat je `mijn_list` opnieuw hebt ingesteld op `[1, 2, 3]`.

### tearDown

Vergelijkbaar met `setUp` is `tearDown`. Deze methode wordt echter na elke test uitgevoerd, om eventuele data 'op te ruimen'. Denk aan het verwijderen van bestanden, leegmaken van een testdatabase of het resetten van instellingen.

```
import unittest
import sqlite3  # Een _package_ om met Sqlite databases te werken

class DatabaseManager:
    """ Een eenvoudige databasemanager. """

    def __init__(self, db_name):
        self.db_name = db_name
        self.conn = None

    def connect(self):
        self.conn = sqlite3.connect(self.db_name)

```

```
    def close(self):
        if self.conn:
            self.conn.close()

class MijnTestCase(unittest.TestCase):
    def setUp(self):
        """ Maak een database en connectie voor elke test. """
        self.db_manager = DatabaseManager('test.db')
        self.db_manager.connect()

    def tearDown(self):
        """ Sluit de database connectie na elke test. """
        self.db_manager.close()

    def test_table_creation(self):
        pass


if __name__ == '__main__':
    unittest.main()
```

In dit voorbeeld begin je elke test met dezelfde database. Na elke test sluit je de verbinding met de database netjes af.

# Project: KNMI

## Inleiding

Je bent aangekomen bij het laatste hoofdstuk. In dit hoofdstuk ga je aan de slag met een project, waarin veel van wat je hebt geleerd in voorgaande hoofdstukken nodig is om het project tot een goed einde te brengen. In het project ga je aan de slag met gegevens van het KNMI (Koninklijk Nederlands Meteorologisch Instituut).

De data bevat maand- en jaarwaarden van het meetstation in De Bilt. Het doel is een aantal vragen te beantwoorden over deze gegevens. In verschillende deelopdrachten werk je naar een eindresultaat toe.

**Leerdoelen**

Aan het einde van dit hoofdstuk:

- Kun je een project plannen
- Kun je een project structureren
- Kun je de juiste oplossing voor de problemen vinden

## Het project: KNMI

**Doel**

Het doel van het project is een aantal vragen over de KNMI-gegevens te beantwoorden. Om die vragen te beantwoorden ga je een programma schrijven, zodat je niet alle antwoorden handmatig hoeft op te sporen in de data.

### Plan

Download eerst de gegevens als `txt` bestand. Inspecteer de data zodat je een gevoel bij de gegevens krijgt.

De data is te verkrijgen via de website van het KNMI.

Deze gegevens ga je gebruiken om de volgende vragen te beantwoorden:

**Opdracht 1**

Wat was voor de jaren 1901, 2000 en 2023 gemiddeld de warmste maand? Wat was de gemiddelde temperatuur van de betreffende maand? Geef voor elk jaar het resultaat als volgt op: "Maand, temperatuur". Bijvoorbeeld: "Mei, 13.3".

**Opdracht 2**

Maak twee groepen aan: de eerste groep bevat alle maanden die door de jaren heen ooit een hoogste gemiddelde temperatuur hadden. De tweede groep bevat alle maanden die door de jaren heen ooit een laagste temperatuur hadden.

Maak twee groepen aan. De eerste groep bevat alle maanden die ooit in een jaar de hoogste gemiddelde temperatuur hadden. Als juni in 1910 bijvoorbeeld de warmste maand was, dan komt juni in deze groep.

De tweede groep bevat alle maanden die ooit in een jaar de laagste gemiddelde temperatuur hadden. Als maart in 1960 de koudste maand was, dan komt maart in deze groep.

Zijn er maanden die in beide groepen voorkomen?

**Opdracht 3**

Verkrijg het jaar met de warmste maand en het jaar met de koudste maand.

**Opdracht 4**

Maak een lijst met de vijf hoogste jaargemiddelden en een lijst met de vijf laagste jaargemiddelden. Maak vervolgens een lijst met alle jaren die één van die 10 temperaturen heeft.

Uit hoeveel jaren bestaat de lijst?

## Opdracht 1: Plan

Maak een plan voor jezelf. Neem hierin op wat je denkt nodig te hebben, hoe je projectindeling er ongeveer uit gaat zien, welke stappen je als eerst gaat zetten, etc.

Het hoeft niet tot in detail te kloppen en het is niet erg als er gaandeweg zaken wijzigen. Het gaat erom dat je actief nadenkt over het eindresultaat en hoe je denkt daar te komen.

▼ *Klik om het antwoord te tonen*

Een mogelijk plan is als volgt.

**Stap 1: Data opschonen**

```
De data ziet er vrij gestructureerd uit, maar moet nog wel worden
opgeschoond. De help-tekst die erboven staat is niet nodig. Verder is het nu
een `txt`-bestand, maar om de gegevens te analyseren is het handig om het om
te zetten naar een `CSV`-bestand.
```

**Stap 2: Structuur bepalen**

Kijkend naar de gegevens en naar de vragen, kun je al een beetje voorspellen hoe de structuur er van je project uit gaat zien. Zo wil je de data opschonen. Ook heb je verschillende berekeningen nodig, die je in functies kunt zetten. Misschien zijn er voor elke berekening ook wel een paar vaste handelingen nodig, die in een eigen functie kunnen. De belangrijkste functies wil je misschien wel testen. Tot slot wil je misschien een `main.py` gebruiken om de rest aan te roepen. Een mogelijke indeling kan dus iets zijn als:

```
knmi/
 ├── calculations.py  # De berekeningen om de vragen te beantwoorden
 ├── contstants.py    # Eventuele constanten
 ├── data.csv         # Opgeschoonde data
 ├── data.txt         # Ruwe data
 ├── main.py          # Hoofdbestand
 ├── services.py      # Functies die in meerdere berekeningen nodig zijn
 ├── tests.py         # Unit testen
```

**Stap 3: Data opschonen**

Schrijf een functie om de data op te schonen.

**Stap 4: Berekeningen schrijven**

Aan de slag met het schrijven van de code om de berekeningen te schrijven. Als je merkt dat je in herhaling valt, kun je de herhalende code in een functie in `services.py` plaatsen. De belangrijkste

functies kun je testen of ze tot het juiste resultaat leiden in `tests.py`.

**Stap 5: Samenvoegen**

Voeg alles samen in `main.py`.

## Data opschonen

Om te werken met de data moet het nog worden opgeschoond. Open de url, kopieer alle tekst en plak dit in een bestandje `data.txt`. Plaats dit bestandje in je projectmap, bijvoorbeeld `knmi`.

Het doel is de data op te schonen en om te zetten naar een CSV-bestand. CSV staat voor *Comma-separated values* en is een tekstbestand dat komma's (of andere tekens) gebruikt om waarden te scheiden. Met een CSV-bestand kun je tabelgegevens opslaan en lezen. Neem bijvoorbeeld onderstaande tabel:

| Kolom A | Kolom B | Kolom C |
|---|---|---|
| 1 | A | 3 |
| 10 | X | 5 |

In CSV-formaat zou dit er zo uit zien:

```
Kolom A,Kolom B,KolomC
1,A,3
10,X,5
```

De eerste regel bestaat uit de (optionele) koppen van de kolommen.Daarna bevat elke regel de rij met waarden, gescheiden door een komma.Na elke komma begint dus een nieuwe kolom.Een CSV-bestand sla je op als `voorbeeld.csv`.

### Opdracht 2: Data opschonen

Schrijf nu code dat `data.txt` neemt, het opschoont en omzet naar `data.csv`. Tips:

- Verwijder alle spaties tussen de kolommen.
- Verwijder de help-tekst en de lege regel tussen de koppen en de rest van de gegevens.

▼ *Klik om het antwoord te tonen*

Een uitwerking is als volgt. Sla deze code bijvoorbeeld op in `services.py` en roep het aan in `main.py`:

```python
# services.py
def clean_data(name):
    """
    Read in the raw data (txt) and clean it up. Remove help text,
    spaces and empty lines. Store as CSV.
    """

    text_file = f"{name}.txt"
    csv_file = f"{name}.csv"

    try:
```

```
        # Open the raw data
        with open(text_file, "r", encoding="utf-8") as file:

            # Loop over every line. If it is the header (starts with
"STN,YYYY")
            # or starts with 260, we want to keep it.
            # Remove spaces
            lines = []
            for line in file:
                if line.startswith("STN,YYYY") or line.startswith("260"):
                    lines.append(line.replace(" ", ""))

        # Write the cleaned lines to a new CSV-file.
        with open(csv_file, "w", encoding="utf-8") as file:
            file.writelines(lines)

    except OSError as e:
        print("Could not open file.", e)

# main.py
from services import clean_data


if __name__ == "__main__":
    clean_data("data")
```

## De opdrachten

Met de opgeschoonde data kun je nu de code gaan schrijven om de opdrachten op te lossen.Werk de opdrachten één voor één af.Merk je dat je code herhaalt: plaats het dan in een aparte functie, zodat je het eenvoudig kunt hergebruiken.Schrijf tijdens het schrijven van de code ook tests, om na te gaan of je de gewenste resultaten verkrijgt.

Verder enkele tips:

- Gebruik de module `csv` uit de standaard bibliotheek om het CSV-bestand in te lezen.
- Schrijf als eerst de functie om de data in te lezen
- Schrijf veelvuldig commentaren bij je code.Laat je het even liggen om later verder te werken, dan kun je teruglezen wat je aan het doen was.
- Maak voor de testen nep-gegevens aan die lijken op de echte data, maar dan veel beperkter.Zo kun je eenvoudig nagaan of je functies het gewenste resultaat opleveren.
- Probeer te itereren.Schrijf een eerste versie die je testen doen slagen.Probeer daarna je code eenvoudiger of duidelijker te maken.
- Kom je er niet uit, gebruik dan `print`-statements om te kijken wat er gebeurt in je code.
- Alles is op te lossen met alles wat je geleerd hebt in deze opleiding. Er zijn geen externe *tools* of andere *packages* uit de standaard bibliotheek nodig, behalve `csv`.

**Opdracht 1**

Wat was voor de jaren 1901, 2000 en 2023 gemiddeld de warmste maand? Wat was de gemiddelde temperatuur van de betreffende maand? Geef voor elk jaar het resultaat als volgt op: "Maand, temperatuur". Bijvoorbeeld: "Mei, 13.3".

**Opdracht 2**

Maak twee groepen aan: de eerste groep bevat alle maanden die door de jaren heen ooit een hoogste gemiddelde temperatuur hadden. De tweede groep bevat alle maanden die door de jaren heen ooit een laagste temperatuur hadden.

Maak twee groepen aan. De eerste groep bevat alle maanden die ooit in een jaar de hoogste gemiddelde temperatuur hadden. Als juni in 1910 bijvoorbeeld de warmste maand was, dan komt juni in deze groep.

De tweede groep bevat alle maanden die ooit in een jaar de laagste gemiddelde temperatuur hadden. Als maart in 1960 de koudste maand was, dan komt maart in deze groep.

Zijn er maanden die in beide groepen voorkomen?

**Opdracht 3**

Verkrijg het jaar met de warmste maand en het jaar met de koudste maand.

**Opdracht 4**

Maak een lijst met de vijf hoogste jaargemiddelden en een lijst met de vijf laagste jaargemiddelden. Maak vervolgens een lijst met alle jaren die één van die 10 temperaturen heeft.

Uit hoeveel jaren bestaat de lijst?