1 Gegevens

Doorheen een computerporgrammaatje gaan we bijna altijd met gegevens zitten goochelen. Maar wat zijn gegevens nu juist? En kunnen we zomaar van het ene type naar het andere gaan?

1.1 Verschillende soorten

Laten we beginnen met een eerste kennismaking met de meest voorkomend soorten gegevens.

1.1.1 Text: `str`

De eerste soort gegevens die we tegenkomen zijn teksten. Een tekst is een reeks van karakters. Een karakter is een letter, cijfer, leesteken, spatie, ... We beginnen en eindigen tekst altijd met aanhalingstekens (").

```
"Dit is een tekst"
"47"                          # ook dat is tekst
```

1.1.2 Getallen: `int` en `float`

Een volgende soort gegevens zijn getallen. We maken een onderscheid tussen getallen zonder en met komma:

- Getallen zonder komma noemen we integers (het Engelse woord voor gehele gatellen): `int`.
- Getallen met komma noemen we floats (het Engelse woord voor 'vlottende' komma): `float`.

```
47                            # integer
47.0                          # opgelet, een float!
3.14                          # float
```

In python kan je van iedere waarde opvragen tot welke soort gegevens het hoort. Hiervoor gebruik je de functie `type()`.
Weet jij het verschil tussen `"47"`, `47` *en* `47.0`*?*
Probeer het eens in de interactive shell!

```
>>> type("47")
>>> type(47)
>>> type(47.0)
```

1.1.3 Logische waarde: `bool`

Een andere veelvoorkomende soort gegevens zijn logische waarden. Er bestaan exact twee logische waarden:

- `True`: de uitkomst is juist.
- `False`: de uitkomst is fout.

```
True
False
# dit zijn de enige twee logische waardes die bestaan
```

1.2 Transformaties

Het is heel belangrijk dat je je realiseert dat python de verschillende soorten gegevens anders behandelt. Ook al lijken ze voor jou op elkaar, zo python ziet een groot verschil tussen `"47"` en `47`. Gelukkig kunnen we meestal heel eenvoudig een waarde van de ene soort omzetten naar een waarde van een andere soort. Dit doen we met functies die dezelfde naam hebben als de verschillende soorten gegevens. Namelijk:

- `str(_)`: een waarde omzetten naar tekst
- `int(_)`: een waarde omzetten naar getal (zonder komma)
- `float(_)`: een waarde omzetten naar kommagetal
- `bool(_)`: een waarde omzetten naar logische waarde

We hebben deze twee waardes verkregen van de gebruiker via de functie `input()`*:* `"12"` *en* `"36"`*.*
We willen de gebruiker de som (`+`*) van deze twee getallen tonen.*
We kunnen dat rechtsreeks met de gekregen tekst doen. Of we kunnen de waardes eerst omvormen naar getallen.
Is er een verschil, denk je?
Had je het juist? Kan je uitleggen waarom?

```
>>> "12" + "36"
>>> int("12") + int("36")
```

Laten we nog eens iets anders proberen. Het getal `3.14` *is een* `float`*, akkoord? Kunnen we dit omvormen naar een waarde van het type* `int` *denk je?*
Zullen we dat eens samen proberen?

```
>>> int(3.14)
```

1.3 Kort overzicht

- `str`
  Tekst, begint en eindigt met aanhalingstekens
  vb. `"Dit is een tekst"`
- `int`
  Gehele getallen (getallen zonder komma)
  vb. `47`

- `float`
  Getallen met komma
  vb. `3.14`

- `bool`
  Logische waarden
  `True` of `False`

- Gegevens omvormen van het de ene soort in de andere:
  `str(_)`, `int(_)`, `float(_)`, `bool(_)`

2 Variabelen

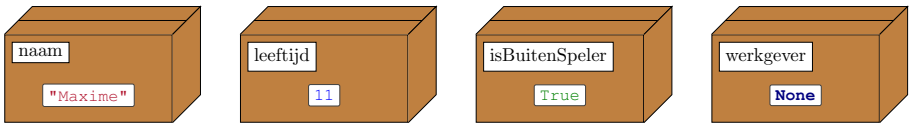
Wanneer we een programma maken, dan gaan we dit vaak in kleine stapjes opbouwen. Heel vaak hebben we het resultaat van een vorige stap nodig in de volgende. Dus moeten we dit resultaat (tijdelijk) kunnen onthouden tussen de verschillende stappen. Dit doen we door gebruik te maken van variabelen.

2.1 Wat is een variabele?

Een variabele is een soort doosje waarin we gegevens kunnen opslaan. We kunnen dit doosje een naam geven, zodat we het later nog weten wat er in zit. Zo een doosje kan ook leeg zijn, in python wordt dit aangegeven door het woordje `None`.

De volgende figuur toont een voorbeeld van vier variabelen die een paar eigenschappen van een jongen/meisje bijhouden.

- 1a: een variable `naam: str` voor je naam.
- 1b: een variable `leeftijd: int` voor je leeftijd.
- 1c: een variable `isBuitenSpeler: bool` om aan te geven of je graag buiten speelt.
- 1d: een variable `werkgever: str | None` om bij te houden waar je werkt, de waarde None betekent dat dit doosje leeg is (dus dat je nog niet werkt).



(a) (b) (c) (d)

Figure 1: Een voorbeeld van vier variabelen die een paar eigenschappen van een jongen/meisje bevatten.

2.2 Spelen met variabelen

2.2.1 Een nieuwe variabele maken

In python gebruiken we een isgelijkteken `=` om een variabele een nieuwe waarde te geven. De naam van de waarde komt voor het isgelijkteken en kan bestaan uit (hoofd)letters (A/a - Z/z), cijfers (0 - 9, maar niet het eerste karakter van de naam) en een liggend streepje (_). Als er geen variabele bestaat met de gegeven naam, dan wordt er een nieuwe gemaakt met deze naam, en krijgt deze de waarde die je rechts van het isgelijkteken staan hebt.

2.2.2 Een nieuwe waarde voor een bestaande variabele

Eenmaal een doosje bestaat, dan ga zijn inhoud regelmatig veranderen door een nieuwe waarde. Gelukkig kunnen we heel gemakkelijk zo een nieuwe waarde toekennen aan een bestaande variabele! Eigenlijk is dat juist hetzelfde als wanneer we de variabele de eerste keer gemaakt hebben, ook gewoon met het `=` teken. Als de variabele deze keer al bestaat, dan wordt de waarde in dit doosje gewoon vervangen en wordt er geen nieuw doosje aangemaakt.

2.2.3 De waarde uit een variabele gebruiken

Wanneer we een doosje een inhoud geven, dan doen we dat natuurlijk omdat we daar later iets mee willen doen. Waardes kunnen we al gebruiken, en dat doen we gewoon door ze te typen waar we ze nodig hebben, toch? Wel, voor variabelen doen we gewoon hetzelfde: we typen gewoon de variabele waar we ze nodig hebben. Simpel he!

Hier vind je twee voorbeelden die beide 'hallo'— lijken te zullen tonen. Denk je dat beide voorbeelden hetzelfde zullen doen, waarom wel/niet?
Kan jij deze voorbeelden eens uitproberen?

```
>>> print("hallo")
```

```
>>> hallo = "bonjour"
>>> print(hallo)
```

Oh ja, `'bonjour'` *is het Franse woord voor* `'goeiedag'`*.*

2.2.4 Een groter voorbeeldje

Zullen we nu eens verschillende variabelen maken, aanpassen en gebruiken? En dat allemaal in één groot voorbeeld?! Neem zeker eens een kijkje naar het stukje code hieronder.

```
naam = "Maxime"              # een nieuwe variabele 'naam'
                             # met de waarde 'Maxime'

leeftijd = 11                # een tweede nieuwe variabele 'leeftijd'
                             # met de waarde '11'

activiteit = "kamp bouwen"   # een derde nieuwe variabele 'activiteit'
                             # voor de huidige bezigheid van Maxime

activiteit = None            # Maxime verveelt zich,
                             # en heeft dus niets te doen
                             # -> het doosje moet nu dus leeg zijn
                             #    dat is waarvoor we 'None'
                             #    gebruiken in python

activiteit = "avond eten"    # etenstijd

leeftijd = leeftijd + 1      # Maxime is jarig!
                             # de nieuwe waarde voor het doosje
                             # 'leeftijd' is de huidige waarde plus 1

print(f"{naam} is ondertussen al {leeftijd} jaar!")
```

Welke leeftijd zou er getoond worden door de `print()` *onderaan uit bovenstaand stukje code? Is dat 11 jaar, de waarde die we er in het begin in de variabele hebben ingestoken?*
Tip: je kan de code uit het voorbeeld hierboven kopiëren en plakken in de interactive shell.

EHBF - Eerste Hulp Bij Foutmeldingen

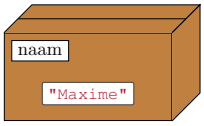
`NameError`

Je ziet deze error, wat nu?

Werkt je code niet en je krijgt in de plaats output in de vorm van:
`NameError: name 'abc' is not defined`
Dat de manier waarop python je vertelt dat hij de variabele `abc` *niet kent. Misschien heb je een typfoutje gemaakt in de naam van je variabele? Controleer of je de naam van je variabele overal hetzelfde hebt geschreven. En probeer daarna of het wel werkt.*

2.3 Kort samengevat

- Een variabele is een doosje waarin we gegevens kunnen opslaan.



- `mijnVariabele = <waarde>`
  Een nieuwe variabele maken we met het isgelijkteken `=`. De naam van de variabele komt voor het isgelijkteken en de waarde die we er in willen stoppen komt er achter.

- `mijnVariabele = <nieuweWaarde>`
  We kunnen een bestaande variabele ook een nieuwe waarde geven. Dit doen we net op dezelfde manier zoals we een nieuwe variabele maken. Maar nu bestaat de variabele al en wordt de waarde in het doosje gewoon vervangen.

- `print(mijnVariabele)`
  De waarde in een variabele kan je gebruiken door de naam van de variabele te typen op de plaats waar je deze waarde nodig hebt. Deze waarde kan je dan gebruiken zoals je dat met een gewone waarde zou doen.

- `mijnVariabele = None`
  Moet het doosje leeg zijn? Dan gebruik je de waarde `None`. In python geeft de speciale waarde `None` aan dat het doosje leeg is.

3 Voorwaardelijke uitvoering: `if ... elif ... else`

3.1 `if`-statement

In de uitvoering van een eenvoudig python programma, wordt iedere regel code gewoon uitgevoerd in de volgorde dat ze in het programma staan. Maar soms komt het voor dat je bepaalde code enkel wil uitvoeren als aan een bepaalde voorwaarde is voldaan. Bijvoorbeeld, je wil op het einde van een spel enkel de boodschap 'Je bent gewonnen!' tonen als de speler ook echt gewonnen heeft. In het andere geval wil je waarschijnlijk juist de boodschap 'Je bent verloren ...' tonen. Zo een voorwaarde koppelen aan de uitvoering van een stukje van de code kan je doen met een `if voorwaarde: ...`-statement. Het statement is opgebouwd als volgt:

```
# deze code wordt sowieso uitgevoerd
if voorwaarde:
    # code die enkel wordt uitgevoerd
    # als voorwaarde resulteert in True
    # anders wordt deze code overgeslagen
# deze code wordt ook terug altijd uitgevoerd
```

Hierin is `voorwaarde` een expressie tot een logische waarde (herinner je, dat is het type `bool`). Aleen en slechts alleen als de waarde van `voorwaarde` `True` is, dan wordt de code in het `if`-blok uitgevoerd. Deze voorwaarde kan van alles zijn:

- een vaste logische waarde:
  vb. `if True: ...` / `if False: ...` waarin het `if`-blok altijd / nooit wordt uitgevoerd.
- een variable zijn die een logische waarde bevat:
  vb. `if spelerIsGewonnen: print("Je bent gewonnen!")`.
- een expressie zijn die een logische waarde oplevert:
  vb. `if aantalPogingen < 3: print("Je bent een genie!")`.

Een `if`-blok begint zodra je een `if`-statement gebruikt hebt. Iedere regel code waarvan je wil dat het tot dit blok behoort, moet je inspringen. Een regel code is ingesprongen als er minstens één spatie of tab voor staat. Het blok loopt door tot aan de eerstvolgende regel code die niet meer ingesprongen is. Let op, al de regels code die tot hetzelfde blok behoren, moeten allen gelijk ingesprongen zijn. Je mag hiervoor geen spaties en tabs door elkaar gebruiken!

Dat is een hele boterham, he? Kan jij dat allemaal nog volgen? Zullen we eens kijken naar een voorbeeldje? Probeer je eens om te voorspellen wat de output gaat zijn van het volgende programma? Nadien kan je het programma kopiëren en uitvoeren in de interactive shell om te zien of je voorspelling juist was.

```
spelerIsGewonnen = True
aantalPogingen = 2

if spelerIsGewonnen:
    print("Je ben gewonnen!")

    if aantalPogingen < 3:
        # je kan een if-statement ook in
        # een ander if-statement plaatsen
        print("Je bent een genie!")

if not spelerIsGewonnen:
    print("Je bent verloren ...")

print("Einde van het spel")
```

EHBF - Eerste Hulp Bij Foutmeldingen

`IndentationError`

Je ziet deze error, wat nu?

Als je een **`IndentationError`** *krijgt, dan is er iets mis met de inspringing van je code. Je kan het volgende controleren:*

- *Heb je een* **`if`***-statement gebruikt zonder bijhorend* **`if`***-blok? Een* **`if`***-statement moet minstens één regel code bevatten, anders krijg je deze error.*
- *Zijn alle regels code die tot hetzelfde blok behoren, allen gelijk ingesprongen? Je mag hiervoor geen spaties en tabs door elkaar gebruiken!*

3.2 `else`-statement

Net zoals dat je een blok code enkel kan laten uitvoeren als een bepaalde voorwaarde waar is, kan je een ander blok code enkel laten uitvoeren als diezelfde voorwaarde niet waar is. Dus je kan met eenzelfde voorwaarde er voor zorgen dat oftewel het ene blok code oftewel het andere blok code wordt uitgevoerd. Dit doen we met met het `else: ...`-statement. Het `else`-statement komt altijd na een `if`-statement en bevat een blok code dat enkel wordt uitgevoerd als de voorwaarde van het `if`-statement niet waar is. Bijvoorbeeld, nu kunnen we opnieuw op basis van de waarde van `spelerIsGewonnen` kiezen of we de boodschap 'Je bent gewonnen!' of 'Je bent verloren ...' tonen. Maar deze keer hebben we daar niet meer twee afzonderlijke `if`-statements voor nodig. We breiden het eerste `if`-statement uit met een `else`-statement op deze manier:

```
# deze code wordt sowieso uitgevoerd
if voorwaarde:
    # code die enkel wordt uitgevoerd
    # als voorwaarde resulteert in True
    # anders wordt deze code overgeslagen
else:
    # code die enkel wordt uitgevoerd
    # als voorwaarde resulteert in False
# deze code wordt ook terug altijd uitgevoerd
```

Laten we nog eens het voorbeeld van het spel bekijken. Maar deze keer gaan we maar één `if`*-statement gebruiken.*
Geeft de code de output die jij verwacht had?

```
spelerIsGewonnen = False
aantalPogingen = 7

if spelerIsGewonnen:
    print("Je ben gewonnen!")
else:
    print("Je bent verloren ...")
    if aantalPogingen < 10:
        print(
            "Maar je hebt het goed geprobeerd!"
        )

print("Einde van het spel")
```

3.3 `elif`-statement

Nu kunnen we al kiezen of we één blok code wel of niet willen uitvoeren door het `if`-statement te gebruiken. We kunnen ook al kiezen om eventueel een tweede blok code uit te voeren als het eerste blok code niet wordt uitgevoerd. Hiervoor hebben we het `else`-statement geïntroduceerd. Maar wat als we nu tussen meer dan twee blokken code moeten kunnen kiezen?

Bijvoorbeeld, we willen nog altijd de boodschap 'Je bent een genie!' tonen als de speler gewonnen heeft met minder dan 3 pogingen. En deze keer willen de boodschap 'Je hebt het goed gedaan!' tonen als de speler gewonnen heeft met minder dan 5 pogingen. En anders willen we de boodschap 'Je bent het goed geprobeerd!' tonen. Dit kunnen we doen met het `elif`-statement. Dit doen we als volgt:

```
# deze code wordt sowieso uitgevoerd
if voorwaarde:
    # code die enkel wordt uitgevoerd
    # als voorwaarde resulteert in True
    # anders wordt deze code overgeslagen
elif andereVoorwaarde:
    # code die enkel wordt uitgevoerd
    # als voorwaarde resulteert in False
    # en andereVoorwaarde resulteert in True
    # anders wordt deze code overgeslagen
else:
    # code die enkel wordt uitgevoerd
    # als beide, voorwaarde en andereVoorwaarde,
    # resulteren in False
# deze code wordt ook terug altijd uitgevoerd
```

Wist je dat je zoveel `elif`-statements kan toevoegen als je wil? Voor iedere bijkomend `elif`-statement geldt dat het bijhorend code blok enkel kan worden uitgevoerd als de voorwaarde van het `if`-statement en alle voorgaande `elif`-statements niet waar zijn.
Wist je ook dat je geen `else`-statement moet toevoegen als je dat niet wil?

Laten we voor een laatste keer nog eens het voorbeeld van het spel bekijken. Maar deze keer mag jij de code schrijven. Lukt het jou om het `if ... elif ... else`-statement te gebuiken om de juiste boodschap te tonen?

- *Als de speler gewonnen heeft met minder dan 3 pogingen, toon je 'Je bent een genie!'.*
- *Als de speler gewonnen heeft met minder dan 5 pogingen, toon je 'Je hebt het goed gedaan!'.*
- *Anders toon je 'Je hebt het goed geprobeerd!'.*

3.4 `if ... elif ... else`: samengevat

- `if voorwaarde:`
  Een `if`-statement laat je toe om een blok code enkel uit te voeren als een bepaalde voorwaarde resulteert tot `True`. Het `if`-statement is altijd het begin van een `if ... elif ... else`-statement.
- `elif andereVoorwaarde:`
  Na een `if`-statement kan je meerdere `elif`-statements toevoegen als je wil. Een `elif`-statement laat je toe om een blok code enkel uit te voeren als de bijhorende voorwaarde `True`is en alle voorgaande voorwaarden `False`zijn. Een `if`-statement hoeft niet altijd gevolgd te worden door een `elif`-statement als je dat niet nodig hebt.
- `else:`
  Tot slot laat het `else`-statement je toe om een blok code enkel uit te voeren als al de voorgaande voorwaardes allemaal `False`zijn. Dit zijn de voorwaardes van het `if`-statement

en alle eventuele `elif`-statements. Net zoals een `elif`-statement, moet je het `else`-statement ook niet altijd gebruiken. Maar als je het `else`-statement gebruikt hebt, dan moet het wel altijd het laatste statement zijn in een `if ... elif ... else`-statement.

4 Voorwaardelijke lus: `while`

Het gebeurt wel eens dat je een stukje code meerdere keren wil uitvoeren. Dat is waarvoor een lus gebruikt kan worden. Volgend blokje code toont de opbouw van een lus in python:

```
while voorwaarde:
    # code die wordt uitgevoerd zolang
    # de voorwaarde resulteert in de logische waarde True
    #
    # net zoals bij een if-statement moeten al de regels code
    # die tot de lus behoren ingesprongen zijn
    # de lus loopt door tot de eerste regel code
    # die niet meer ingesprongen is
```

Het `while`-sleutelwoord geeft aan dat je een lus wil starten. Natuurlijk wil je ook niet dat de lus oneindig blijft doorgaan. Daarom geef je een voorwaarde mee aan de lus. Net zoals bij het `if`-statement, is de voorwaarde een expressie tot een logische waarde. Deze expressie wordt geëvalueerd telkens de lus opnieuw start. Dit gaat als volgt:

- Stap 1:
  De voorwaarde wordt geëvalueerd, en het resultaat is:
  - `True`: ga naar stap 2.
  - `True`: ga naar stap 3.
- Stap 2:
  De code in de lus uitgevoerd.
  Ga terug naar stap 1.
- Stap 3:
  De lus wordt niet meer opnieuw uitgevoerd. In de plaats gaat het programma meteen verder vanaf de eerste regel code na de lus.

Meestal gaan we een uitdrukking gebruiken als de voorwaarde van een lus. Maar we mogen ook gewoon een van de twee logische waardes rechtstreeks gebruiken.
Wat zouden de volgende twee lussen doen? Zullen we eens proberen in de interactive shell?

```
while: False
    print("Hallo")
```

```
while: True
    print("Hallo")
```

Tijd voor een echt voorbeeldje, ik denk dat je daar klaar voor bent! Ken je de reeks van Fibonacci? Dat is een getallenrij waarbij ieder getal de som is van de twee voorgaande getallen. De rij begint met tweemaal het getal 1. Wij hebben de eerste getallen al voor jou berekend: 1, 1, 2, 3, 5, 8, 13, 21, 34, ... Hebben we ze allemaal juist volgens jou?

Nu we weten wat de reeks van Fibonacci is, kunnen we deze reeks ook berekenen met een lus. We starten met de eerste twee getallen, en gaan proberen om de volgende tien getallen te berekenen.

```
# we starten met de eerste twee getallen van de reeks van Fibonacci
# en houden deze bij in de variabelen a en b
getalM2 = 1             # getal op twee plaatsen terug
getalM1 = 1             # vorige getal

# we houden bij hoeveel getallen we al berekend hebben
# we beginnen bij 2, want de eerste twee getallen kennen we al
getallenBerekend = 2

# we maken een lus die telkens een nieuw getal van de rij berekent
# dit blijven we doen tot we aan 10 getallen zitten
while getallenBerekend < 10:
    # we berekenen het volgende getal
    # door de som te nemen van de vorige twee
    # getallen
    volgendGetal getalM2 + getalM1

    # we tonen het getal
    print(f"Fibonacci getal {getallenBerekend}: {volgendGetal}")

    # in de volgende lus wordt:
    # 1. het vorige getal nu het getal op twee plaatsen terug
    getalM2 = getalM1
    # 2. het nieuwe getal nu het vorige getal
    getalM1 = volgendGetal

    # tot slot tellen we 1 op bij het aantal getallen dat we al
    # berekend hebben, anders riskeren we dat de lus nooit stopt!
    getallenBerekend += 1
print(
    f"We hebben {getallenBerekend} Fibonacci getallen berekend!"
)
```

In het voorbeeld schuiven we de getallen telkens op. Dit doen we hier:

```
getalM2 = getalM1
getalM1 = volgendGetal
```

We hadden ook dit kunnen doen:

```
getalM1 = volgendGetal
getalM2 = getalM1
```

Denk je dat het op deze manier ook gewerkt zou hebben? Waarom denk je van wel, of waarom denk je van niet? Probeer het zeker eens uit in de interactive shell!

Ken je de reeks van de even getallen?
Dit zijn de getallen 0, 2, 4, 6, 8, 10, 12, 14, 16, ...
Denk je dat jij deze reeks kan berekenen met een lus? Ik denk van wel! Voor deze reeks heb je maar één variabele nodig. Je kan starten met het getal 0. In iedere lus tel je er 2 bij op.
Is het je gelukt?

4.1 `break`

Is de voorwaarde van de `while`-lus waar, dan wordt de inhoud van de lus van de eerste tot de laatste regel uitgevoerd. Soms moeten we echter de uitvoering van de lus abrupt kunnen onderbreken. Het `break`-statement is een eerste statement die dit kan. Als de utvoering van het programma het `break`-statement tegenkomt, dan is de lus meteen gedaan. Ook het stukje code van de lus dat na dit `break`-statement komt, wordt zelfs niet meer uitgevoerd!

We zullen een voorbeelde gebruiken om dit duidelijker te maken. We willen het gemiddelde berekenen van 10 getallen die we aan de gebruiker vragen. Als de gebruiker geen 10 getallen heeft, dan zou de lus toch eerder moeten kunnen stoppen. En moet het programma het gemiddelde tonen van slechts deze paar getallen die de gebruiker gegeven heeft. Hieronder hebben we dit vertaald naar een stukje python code:
Zonder het groene gearceerde blok heb je een `while`-lus dat het gemiddelde van exact 10 waardes berekend. Met het groene gearceerde blok kan de gebruiker de lus eerder laten stoppen als hij geen 10 waardes heeft.

```
# we hebben twee variabelen nodig
som = 0                     # de tussentijdse som
aantal = 0                  # het huidige aantal gekregen getallen

while aantal < 10:          # de lus moet doorgaan tot 10 getallen

    # we gebruiken input() om een nieuw getal te vragen
    nieuweInput = input("Geef een getal: ")

    # als de gebruiker geen getallen meer heeft
    # dan vult hij niets meer in
    # in dat geval is nieuweInput een lege tekst
    # dit moeten we eerst controleren voor we verder kunnen
    if nieuweInput == "":
        # de gebruiker heeft geen getallen meer
        # we moeten de lus vroegtijdig stoppen!
        break

    som += int(nieuweInput) # we zetten de tekst om in een getallen
                            # die we dan bij som optellen
    aantal += 1

# tot slot bepalen we het gemiddelde en tonen we deze aan de gebruiker
gemiddelde = som/aantal
print(f"Het gemiddelde van de gegeven getallen: {gemiddelde}")
```

Heb je het `break`*-statement begrepen? Snap je waarom het voorbeeldje hierboven het gemiddelde juist berekend, ook in die gevallen waarbij de gebruiker geen 10 getallen heeft?*
We hebben hetzelfde voorbeeldje hieronder nog eens staan, maar dit keer wordt het aantal aan het begin van de lus verhoogd.
Denk jij dat er nu een andere waarde voor het gemiddelde berekend gaat worden? Of blijft het resultaat hetzelfde?
Probeer het eens uit, kan jij uitleggen wat er juist gebeurt?

```
som = 0
aantal = 0
while aantal < 10:
    aantal += 1

    nieuweInput = input(
        "Geef een getal: "
    )

    if nieuweInput == "":
        break

    som += int(nieuweInput)
gemiddelde = som/aantal
print("Het gemiddelde van de gegeven getallen:")
print(gemiddelde)
```

4.2 `continue`

Standaard wordt de blok code van een lus volledig uitgevoerd tijdens iedere herhaling. Maar soms willen we een herhaling (gedeeltelijk) kunnen overslaan. En dit kunnen we! Wanneer de uitvoering van het programma het `continue`-statement tegenkomt, dan stopt de huidige herhaling onmiddellijk en begint meteen de volgende herhaling. De inhoud van de lus dat na het `continue`-statement komt, wordt overgeslagen.

We willen een programma maken om het gemiddelde van 10 getallen willen berekenen. Deze getallen moeten aan de gebruiker gevraagd worden via de functie `input()`. Het is heel belangerijk dat er exact 10 getallen gegeven worden. Dus als de gebruiker per ongeluk op enter duwt zonder een getal te hebben gegeven, dan telt deze input niet. Laten we dit eens vertalen naar code:
Zonder het groene gearceerde blok heb je een `while`-lus dat het gemiddelde van exact 10 waardes opvraagd van de gebruiker. Met het groene gearceerde blok worden ongeldige waardes overgeslagen.

```python
# we hebben twee variabelen nodig
som = 0                     # de tussentijdse som
aantal = 0                  # het huidige aantal gekregen getallen

while aantal < 10:          # de lus moet doorgaan tot 10 getallen

    # we gebruiken input() om een nieuw getal te vragen
    nieuweInput = input("Geef een getal: ")

    # als de gebruiker ongeldige input geeft
    # dan mag deze stap niet tellen
    if nieuweInput == "":
        # de gebruiker heeft per ongeluk op enter geduwd?
        # we moeten deze stap overslaan!
        continue

    som += int(nieuweInput) # we zetten de tekst om in een getallen
                            # die we dan bij som optellen
    aantal += 1

# tot slot bepalen we het gemiddelde en tonen we deze aan de gebruiker
gemiddelde = som/aantal
print(f"Het gemiddelde van de gegeven getallen: {gemiddelde}")
```

4.2.1 In het kort

- `while voorwaarde:`
  Een `while`-lus gebruik je om een blok code te blijven herhalen. De uitvoering stopt pas tot de voorwaarde evalueert tot `False`.

- `break`
  Het `break`-statement stopt een lus meteen. En dat meteen op de plaats waar het `break`-statement zich bevindt. Een nieuwe herhaling wordt niet meer uitgevoerd.

- `continue`
  De resterende uitvoering van de blok code wordt overgeslagen vanaf waar het `continue`-statement wordt uitgevoerd. In de plaats wordt meteen een nieuwe herhaling opgestart, beginnend met een volgende evaluatie van de voorwaarde.

5 Herbruikbare blokken

```
# start van het spel
score = 0

# toon de score (als twee karakters lang)
print("Huidige score: ", end = "")
if score < 10:
    print("0", end = "")
print(str(score))

# je hebt geluk: je krijgt meteen wat punten cadeau!
score += 7

# toon de nieuwe score (opnieuw als twee karakters lang)
print("Huidige score: ", end = "")
if score < 0:
    print("0", end = "")
print(str(score))

# bonus verdient, goed bezig!
score += 5

# nieuwe score, dus weeral tonen
print("Huidige score: ", end = "")
if score < 0:
    print("0", end = "")
print(str(score))

# enz. ...
```

Interessant spel he? We zijn al meteen goed bezig! Maar er lijken wel veel regels code te zijn die telkens terugkomen. Dat kunnen we vast efficiënter doen. Jazeker, welkom in de wondere wereld van de functies!

Een functie is blokje code dat we schrijven, maar dat niet meteen uitgevoerd mag worden. Als we later in het programma naar deze functie verwijzen, dan wordt de code in de functie pas uitgevoerd. Nu hoor ik je al denken: "Maken we het zo niet gewoon ingewikkeld?". Nee, helemaal niet, het maakt de code juist veel leesbaarder! Functies zullen ons o.a. helpen als we:

- een stukje code niet éé keer, maar meermaals willen hergebruiken in het programma.
- de leesbaarheid van het programma te verhogen. Dat kunnen we omdat we functies een naam geven. Met deze naam, kan je samenvatten wat een hele blok code eigenlijk doet.
- Je gaat vast nog voorbeelden vinden waarbij je een functie zal willen gebruiken!

5.1 Eenvoudige functies

Een functie is een blok code dat we eerst definiëren en het daarna pas elders in het programma uit te voeren. Het `def`-statement geeft aan dat we een functie gaan definiëren. We definiëren, en gebruiken, een functie als volgt:

```
def duidelijkeFunctienaam():
    # code die pas uitgevoerd wordt
    # wanneer we later deze functie aanroepen

# hieronder wordt de functie opgeroepen
# en de code in de functie uitgevoerd
duidelijkeFunctienaam()
```

We hadden gezegd dat we het voorbeeldje hierboven zouden kunnen inkorten wanneer we functies kunnen toepassen, niet? Jawel, dan gaat we dat ook meteen eens proberen:

```
# we maken de functie om de score te tonen
# deze code in de functie wordt voorlopig overgeslagen
def toonScore():
    # toon de score (als twee karakters lang)
    print("Huidige score: ", end = "")
    if score < 10:
        print("0", end = "")
    print(str(score))

score = 0                   # start van het spel

toonScore()                 # toon de initiele score

score += 7                  # je hebt geluk: meteen wat punten cadeau!

toonScore()                 # toon de nieuwe score

score += 5                  # bonus verdient, goed bezig!

toonScore()                 # nieuwe score, dus weeral tonen

# enz. ...
```

Dat is nu toch veel leesbaarder, he.
Mission completed: we kunnen functies maken en gebruiken!

Oei, aan het tempo dat we bezig zijn, gaan rap scores boven de 100 hebben!
We zouden de score als 3 karakters ipv. slechts twee moeten tonen.
Ik heb iets gevonden dat lijkt te werken:

```
if score < 100:
    print("0", end = "")
toonScore()
```

Als ik dit `if`*-statement overal kopiëer voor iedere* `toonScore()`*, dan lijkt ons probleem opgelost, oef.*
Klopt dat denk je, zal de score nu correct als 3 karakters worden getoond?
Probeer het zeker eens uit!

Maar is dat nu wel de juiste oplossing? Ik blijf maar denken dat het beter kan, maar kan zelf niet vinden hoe.
Kan jij een betere oplossing voorstellen?
Laten we jouw oplossing ook eens uitproberen!

```
>>> ... # jouw oplossing, ik ken ze nog niet he ;)
```

`print(_, end = '...')`

Had je deze `end = '...'` *ook gezien in het voorbeeld?*
Hierdoor hebben we het normale gedrag van `print` *aangepast. Ipv. een nieuwe regel te starten na de* `print`*, wordt er nu niets gedaan na de* `print`*. De volgende* `print` *wordt dus meteen op dezelfde regel getoond.*

5.2 Functies met argumenten

In een functie definiëren we code dat we meermaals willen hergebruiken. Herinner je dat we soms variabelen maken om later in de code te kunnen gebruiken (hoofdstuk 2)? Zo gaan we ook code willen hergebruiken dat iets doet met een waarde. We gaan deze code willen kunnen oproepen, maar telkens met een andere waarde(s) als uitgangspunt. En ook dat kunnen we met functies. Al moeten we deze waarde(s) nu wel extra kunnen meegeven: dat gaat via functie-argumenten. We doen dit als volgt:

```
def functienaam(arg0: type, arg1: type, ...):
    # code van de functie
    # * binnen de functie kunnen we de argumenten
    #   gebruiken als gewone variabelen
    # * ': type' mag je gebruiken om aan te geven
    #   wat voor soort gegevens de argumenten zijn
    #   maar het is niet verplicht

# hieronder roepen we de functie aan
# en geven we waardes mee aan de argumenten
functienaam(waarde0, waarde1, ...)
```

Ben je al helemaal mee? Ik nog niet helemaal hoor. Maar da's niks, een voorbeeld doet wonderen. We gaan nog eens de score tonen. Deze keer gaan we het maximum er ook bij tonen. Het tonen van de score en het maximum gaan we telkens op dezelfde manier tonen. Dat zal dus een functie worden. Maar de score, alsook het maximum, zijn waardes die kunnen veranderen. Deze zullen we dus moeten meegeven als argumenten. We zouden bijvoorbeeld de functie zo kunnen opbouwen:

```
def toonScore(score: int, maximum: int):
    print(f"Huidige score: {score:3d}/{maximum:d}")

# voorbeeld van het aanroepen van de functie
toonScore(47, 50)  # de speler heeft bijna het maximum!
```

Hierin worden de score en het maximum als argumenten meegegeven. Daarna worden ze als zijnde variabelen gebruikt binnenin de functie. Als je deze zin leest, dan heb je alles gelezen van functies met argumenten. Je met glans en glorie geslaagd! Op naar de volgende ronde.

5.3 Het `return`-statement

6u58 . . . 6u59 . . . 7u00 BIEB BIEB BIEB!
Hey, het is wel zaterdag he!
Geen school vandaag, vlug snoozen en blijf ik nog vijf minuten liggen.

Herken je deze situatie? Natuurlijk niet, het is weekend, een ganse dag om te ravotten! Dan heb ik toch geen wekker nodig om op te staan, duh. Maar toch, kan je geloven dat sommigen toch eerst nog eens snoozen alvorens op te staan? Hoe zou de wekker dat afhandelen?

Snoozen wil zeggen: de wekker stopt en gaat vijf minuten later opnieuw af. De nieuwe tijd is dus vijf minuten na het moment dat onze schone slaper op snooze duwt. De nieuwe tijd berekenen is een taak dat een wekker wel vaker moet doen, een functie dus!
Alhoewel, het resultaat van de berekening hebben we wel nodig. Wanneer een functie eindigt, dan spring de uitvoering van het programma terug naar de plaats waar de functie opgeroepen was. Zijn we dan het resultaa gewoon kwijt? Helemaal niet, we kunnen het resultaat van een functie teruggegeven aan het programma. Dat is waarvoor we het `return`-statement gebruiken:

```python
def functieMetResultaat(arg0: type, arg1: type, ...) -> type:
    # de code van de functie
    # ditmaal met een waarde
    # die teruggegeven moet worden
    return waarde           # een constante waarde
                            # of een variabele
    # de functie wordt onheroepelijk gestopt
    # na het return-statement
    # code erna wordt NIET meer uitgevoerd

# nu kunnen we de functie oproepen
# en gebruiken op elke plaats
# waar we ook een waarde kunnen gebruiken
# vb: hier bewaren we het resultaat in een variabele
variabele = functieMetResultaat(waarde0, waarde1, ...)
```

En zo gaan we het `return`-statement meestal gebruiken. Maar daarnaast zullen deze weetjes je ook zeker helpen:

- De functie stopt meteen na het `return`-statement. Iedere regel code na een `return`-statement wordt niet (maar echt niet he) uitgevoerd. Het verloop van het programma springt meteen terug naar de plaats waar de functie werd opgeroepen.
- Het `return`-statement kan alleen maar gebruikt worden binnen een functie. Als je `return` gebruikt buiten een functie, dan zal het programma eindigen met een error daar waar het `return`-statement foutief gebruikt wordt.
- Ja hoeft niet altijd een waarde mee te geven met een `return`-statement. Je kan het `return`-statement ook gebruiken zonder een waarde mee te geven. Dan zorgt het `return`-statement er gewoon voor dat de functie stopt. Dit kan handig zijn in een `if`, `else`, `while` blok.

Allemaal goed en wel, maar hoe doen we dat nu? Wel, de functie die onze wekker gebruikt, zou er weleens zo uit kunnen zien:

```python
def tijdNaSnoozen(uren: int, minuten: int) -> Tuple[int, int]:
    minuten += 5            # we moeten er 5 minuten bijtellen
    if minuten >= 60:
        # we moeten de uren verhogen als er meer dan 60 minuten zijn
        minuten -= 60
        uren += 1
    return uren, minuten

# nu gaan we de functie gebruiken
# en kunnen we de nieuwe tijd voor de wekker
uren, minuten = tijdNaSnoozen(7, 1)
print(f"Wekker gaat nu af om {uren:02d}:{minuten:02d}")
```

```
>>> tijdNaSnoozen(7, 1)
>>> tijdNaSnoozen(7, 56)
```

Klopt het resultaat?
Is dat altijd zo denk je? Dan heb je dit nog niet geprobeerd:

```
>>> tijdNaSnoozen(23, 58)
```

Kan jij de functie aanpassen zodat dit ook werkt?

5.4 Recursieve functies

Zeer besten lezer ...
Sluit de ogen en ontspan.
(En doe ze ook maar weer open, anders wordt het moeielijk lezen ...)
Alleen zo zal de magie tot jou komen!

Welja, wat we nu gaan ontdekken is nu toch wel echt een klein beetje speciaal hoor. Dus ik vond zo een intro wel plezant! Misschien dacht je nu nog dat je een functie alleen maar kan gebruiken, pas nadat ze gedifiniëert is? Wel, dan gaat het volgende plezant worden!

5.4.1 n faculteit

Stel dat we nu eens het product willen weten van alle getallen van 1 tot en met een gegeven getal n. Bijvoorbeeld, we geven als input het getal 5, dan moet het resultaat 120 ($= 1*2*3*4*5$) zijn. Dit product wordt met een moeilijk woord 'faculteit' genoemd. Zo is '5 faculteit' gelijk aan 120, of '2 faculteit' gewoon 2 (want $1*2=2$). Een hele toffe eigenschap is dat als je '5 faculteit' kent, dan je dan heel eenvoudig '6 faculteit' kunt vinden. Ah ja, '6 faculteit' is gelijk aan 6*'5 faculteit' ($6*5*4*3*2*1 = 6*(5*4*3*2*1)$) oftewel 720. En dat is een algemene eigenschap, 'n faculteit' is gelijk aan n*'$n-1$ faculteit'

We kunnen nu de functie `def faculteit(n: int) -> int` maken. Maar we gaan niet het hele product uit rekenen in de functie, we gaan lui zijn! We gaan gewoon n vermenigvuldigen met het resultaat van `faculteit(n - 1)`. En we weten dat die functie `faculteit(_)` bestaat, want die zijn we nu aan het maken! Een beetje verward, niet? Geen zorgen, laten we het gewoon eens proberen:

```
def faculteit(n: int) -> int:
    if n == 1:                  # triviaal geval
                                # we moeten kunnen stoppen op het eind
        return 1                # '1 faculteit' is 1
    else:                       # recursief geval
        return n*faculteit(n - 1)   # n faculteit = n*'(n - 1) faculteit'
```

Een functie die zichzelf aanroept, dat noemen we nu een recursieve functie.

Nooit zomaar geloven wat je gezegt wordt, he! Niets zegt dat dit echt werkt. Er is maar één manier om daar achter te komen!
We proberen het gewoon uit! Wat denk je dat het volgende gaat geven?

```
>>> faculteit(1)
>>> faculteit(5)
>>> faculteit(6)
>>> faculteit(20)
```

Zaten we juist?

Waarom hebben we eigenlijk `if n == 1: return 1` *gebruikt? Zo het niet werken zonder dat stukje code? Denk je dat deze functie ook gaat werken?*

```
def faculteit(n: int) -> int:
    return n*faculteit(n - 1)
```

Probeer het eens uit!
Wat zie je gebeuren? En waarom denk je dat dit gebeurt?

5.4.2 De reeks van Fibonacci

Herinner je je de reeks van Fibonacci nog uit hoofdstuk 4? Of heb je dat hoofdstuk stiekem overgeslagen? In dat geval zullen hem eventjes opfrissen.

De reeks van Fibonacci is een getallenrij waarbij ieder getal de som is van de twee voorgaande getallen. De rij begint met tweemaal het getal 1. Als we dat toepassen, dan krijgen we voor de eerste getallen: 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
Oftewel, het n^{de} Fibonacci getal is de som van het $(n-1)^{ste}$ Fibonacci getal en het $(n-2)^{de}$ Fibonacci getal. En met de bijkomende criteria dat zowel het eerste als het tweede Fibonacci getallen beide 1 zijn.

Wel, voor de reeks van Fibonacci kunnen we ook handig gebruik maken van recursie. Maar nu we gaan we het voor de verandering eens omgekeerd doen: kan jij me het voorbeeld maken en er zo voor zorgen dat ik het ook snap?

We gaan proberen (en slagen!) de volgende functie maken:

```
def fibonacci(n: int) -> int:
    ...
```

Wat hebben we nodig?

- *Het eerste element is* 1.
  ⇒ `fibonacci(1) == 1`
- *Het tweede element is ook* 1.
  ⇒ `fibonacci(2) == 1`
- *Elk volgend element is de som van de twee voorgaande elementen.*
  ⇒
  `fibonacci(n) == fibonacci(n - 2) + fibonacci(n - 1)`

Kan jij de functie afmaken?
Is het je gelukt denk je?

```
>>> fibonacci(1)
>>> fibonacci(5)
>>> faculteit(10)
```

Krijg je het verwachte resultaat?

Je mag zeker je oplossing tonen! Ook als je het niet helemaal kan vinden, het lukt ons vast en zeker samen wel!

5.5 Functies die je al gebruikt hebt

Heb je dit hoofdstukje goed *gesnopen*? Dan is je misschien al opgevallen dat je (bestaande) functies eigenlijk al gebruikt hebt.

- Zo is `print(...)` een functie die we al heel vaak gebruikt hebben. Het is een functie die de argumenten die we meegeven toont op het scherm.
- Ook `waarde = input(...)` is een functie die we al gebruikt hebben. Deze functie toont de tekst die we als argument meegeven op het scherm, en wacht daarna tot de gebruiker iets intypt en op enter drukt. De functie geeft dan de tekst die de gebruiker heeft ingetikt terug.
- Nog een ander voorbeeld zijn de functies `int(...)`, `float(...)`, `str(...)`, ... Elk van deze functies probeert om de waarde, die we als argument meegeven, om te zetten naar het respectievelijke gegevens type.
- Tot slot zijn er nog zo vele andere functies die anderen voor ons al hebben gemaakt. Bijvoorbeeld, in het voorbeeld van de klok hebben we de functie `time()` gebruikt. Deze functie komt uit de `time`-module waar we eerst naar moesten verwijzen, anders had python niet geweten waar deze functie te vinden was.

5.6 Een korte samenvatting

- `def functienaam(): ...`
  Met het `def`-statement definiër je een functie. De code in de functie wordt niet meteen uitgevoerd, maar pas wanneer je deze functie aanroept.

- `functienaam()`
  Door de functienaam te combineren met haakjes, roep je de functie aan. Nu wordt de code in de functie uitgevoerd.

- `def functienaam(arg0: type, arg1: type, ...): ...`
  Functies kunnen één of meerdere argumenten hebben. Argumenten zijn waardes die je aan de functie doorgeeft. Binnen de functie kan je deze argumenten gebruiken als gewone variabelen.

- `def functienaam() -> returnType: return ...`
  Het `return`-statement
  - beëindigt de uitvoering van een functie meteen, zelfs als er nog code had achter gestaan.
  - kan gevolgd worden door een waarde. Dan wordt deze waarde door de functie teruggegeven, en kan je verder gebruiken in de code.
  - mag je alleen gebruiken binnen in een functie.

- `def functienaam(arg00: type, arg1: type, ...) -> returnType: return ...`
  Natuurlijk kan een functie zowel argumenten als een `return`-waarde hebben.

- Enkele voorbeelden van functies die je waarschijnlijk al gebruikt hebt:
  - `print(...)`
    toont die de argumenten die we meegeven op het scherm.
  - `waarde = input(...)`
    toont de tekst van het argument op het scherm, en geeft de input van de gebruiker daarna terug als `str`.
  - `int(...)`, `float(...)`, `str(...)`, ...
    probeert om de waarde die we als argument meegeven om te zetten naar het respectievelijke gegevens type.