1 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!

1.1 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.*

1.2 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 **??**)? 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.

1.3 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?

1.4 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!

1.4.1 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?

1.4.2 De reeks van Fibonacci

Herinner je je de reeks van Fibonacci nog uit hoofdstuk **??**? 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!

1.5 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.

1.6 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.