Python Development

Introductie tot programmeren in Python

Kristof Michiels

Inhoud

- We behandelen de basis van Python. Genoeg om van start te kunnen gaan met oefeningen die bestaan uit een reeks van relatief eenvoudige statements
- In latere hoofdstukken behandelen we meer aspecten van Python maar gaan daarnaast ook nog dieper in op een aantal zaken die hier korter worden behandeld
- Dit zal ons helpen om grotere en meer complexe problemen aan te pakken met onze code

Wat zien we in deze les?

- Code structureren in Python
- Output genereren met de print()-functie
- Commentaar in je code
- Coderegels laten doorlopen met \
- Variabelen en toewijzingen
- Expressies

Wat zien we in deze les?

- Expressies
- Strings
- Getallen / berekeningen maken
- None
- Input van de gebruiker

Code structureren in Python

Code structureren in Python

- Witruimte en indentatie leggen de structuur van blokken code vast ipv haakjes en puntkomma's
- Dit minimalisme is één van de meest karakteristieke aspecten van Python
  - Geen haakjes meer die ontbreken waarnaar je op zoek moet
  - Visuele structuur van de code komt overeen met de reële structuur. Dat maakt de code beter leesbaar
  - Python code ziet er vrij gelijk uit, welke programmeur ze ook geschreven heeft

```
vruchten = ["appel", "kiwi", "banaan", "mango"]
for vrucht in vruchten:
      print(vrucht)
          if vrucht == "banaan":
                break
```

Code structureren in Python

- Voor elk insprong-niveau gebruik ik 4 spaties
- Aanbevolen stijl - de zogenaamde PEP-8 die je vaak zult horen noemen: https://www.python.org/dev/peps/pep-0008/
- Gebruik geen tabs en meng geen tabs en spaties: indent-count wordt hierdoor verstoord
- :-) https://www.youtube.com/watch?v=SsoOG6ZeyUI (fragment Silicon Valley)

Output genereren met de print()-functie

Output genereren met de print()-functie

- Om output te genereren vanuit onze programma's gaan we in de cursus voornamelijk gebruik maken van de print()-functie
- Dit is de in Python gebruikelijke manier om zaken op je scherm te krijgen
- De wondere wereld van print(): https://realpython.com/python-print/

```
print("Welkom bij Python Development!")
```

Commentaar in je code

Commentaar in je code

- Commentaar laat toe om zelfgeschreven notities toe te voegen aan je programma's
- In Python gebruiken we een hashtag ("#") aan het begin van een lijn commentaar om aan te geven dat we hiermee te maken hebben
- We geven hiermee aan de interpreter mee dat wat volgt niet hoeft geïnterpreteerd te worden
- Python heeft geen notatie voor multi-line commentaar. Je gebruikt een hashtag aan het begin van elke regel commentaar

```
# Een typisch "Hallo wereld"-voorbeeld
boodschap = "Welkom bij Python Development!"
print(boodschap)
```

Coderegels laten doorlopen met \

Coderegels laten doorlopen met \

- Code is leesbaarder wanneer de regels kort wordt gehouden
- Aangeraden max aantal karakters door PEP-8 is 79: https://www.python.org/dev/peps/pep-0008/#maximum-line-length
- Als de regel toch langer wordt kan je gebruik maken van de backslash (= 'continuation character')
- Plaats je een \ op het einde van een lijn dan gedraagt Python zich alsof je je nog steeds op dezelfde regel bevindt

```
som = 1 + \
    2 + \
    3 + \
    4
print(som) # 10
```

Variabelen en toewijzingen

Variabelen en toewijzingen

- De meest gebruikte opdracht in Python is de toewijzing. Komt vrij goed overeen met gebruik in andere talen
- In tegenstelling tot veel andere computertalen is er geen declaratie van het variabele type of een scheidingsteken aan het einde van de regel nodig
- De lijn wordt beëindigd door het einde van de lijn
- Variabelen worden automatisch gemaakt wanneer ze voor het eerst worden toegewezen

```
getal = 12
```

Variabelen en toewijzingen

```
boodschap = "Welkom bij Python development!"
print(boodschap)

boodschap = "Python leren programmeren behoort tot het mooiste wat er bestaat"
print(boodschap)
```

- We maken een variabele aan en noemen ze "*boodschap*"
- Elke variabele is verbonden met een waarde. Dit is de informatie die met de variabele wordt geassocieerd
- Je kan de waarde van een variabele op elk moment wijzigen wijzigen in je programma. Python zal steeds de huidige waarde van de variabele bijhouden en kunnen weergeven

Variabelen en toewijzingen

- Een veel voorkomende, maar onnauwkeurige verklaring is dat een variabele een container is die een waarde opslaat, een beetje zoals een emmer. Dit zou redelijk zijn voor veel programmeertalen (bvb C)
- In Python zijn variabelen echter geen containers. In plaats daarvan zijn het labels of tags die verwijzen naar objecten in de namespace van de Python-interpreter
- Verschillende labels (of variabelen) kunnen naar hetzelfde object verwijzen, en wanneer dat object verandert, verandert ook de waarde waarnaar door al die variabelen wordt verwezen

```
a = [2, 4, 8]
b = a
c = b
b[1] = 6
print(a, b, c) # [2, 6, 8] [2, 6, 8] [2, 6, 8]
```

Variabelen en toewijzingen

- Met variabelen als containers houdt dit resultaat uit bovenstaand voorbeeld geen steek
- Hoe kan het veranderen van de inhoud van één container tegelijkertijd de andere twee veranderen?
- Als variabelen echter alleen maar labels zijn die naar objecten verwijzen, is het logisch dat het wijzigen van het object waarnaar alle drie de labels verwijzen, overal wordt weerspiegeld
- Als de variabelen verwijzen naar constanten of onveranderlijke waarden, is dit onderscheid niet zo duidelijk

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

Variabelen en toewijzingen

- Omdat de objecten waarnaar ze verwijzen niet kunnen veranderen, is het gedrag van de variabelen in dit geval consistent met beide verklaringen
- In dit geval verwijzen a, b en c in feite allemaal naar hetzelfde onveranderlijke integer-object met waarde 1
- De volgende regel, b = 2, zorgt ervoor dat b verwijst naar het integer-object 2, maar het verandert niet de referenties van a of c

Variabelen en toewijzingen

Python-variabelen kunnen op elk object worden ingesteld, terwijl variabelen in veel andere talen alleen het type waarde kunnen opslaan waarin ze zijn gedeclareerd. Het volgende is volkomen toegelaten:

```
a = "Hallo"
print(a)    # Hallo
a = 5
print(a)    # 5
```

Een nieuwe toekenning overschrijft alle eerdere toekenningen. De del-instructie verwijdert de variabele:

```
b = 5
del b
print(b) # Zal een NameError exception raisen
```

Naamgeving variabelen

- Gebruik korte, beschrijvende namen
- Vermijd het gebruik van hoofdletters
- Namen van variabelen kunnen enkel bestaan uit letters, getallen en *underscores* ("_")
- De naam mag starten met een *underscore*, maar niet met een getal
- Spaties zijn niet toegestaan. Maak eventueel gebruik van *underscores* om woorden van elkaar te scheiden
- Vermijd het gebruik van Python *keywords* als namen voor variabelen. Dit zijn gereserveerde woorden (zie volgende slide)

Gereserveerde woorden in Python

```
help("keywords")
```

```
Here is a list of the Python keywords.  Enter any keyword to get more help.

False               break               for                 not
None                class               from                or
True                continue            global              pass
__peg_parser__      def                 if                  raise
and                 del                 import              return
as                  elif                in                  try
assert              else                is                  while
async               except              lambda              with
await               finally             nonlocal            yield
```

Meerdere toewijzigen

```
voornaam, familienaam, beroep = "Kristof", "Michiels", "Docent"
```

- Je kan waarden aan meerdere variabelen toekennen in één enkele statement
- Dit maakt je programma's korter en beter leesbaar
- Je scheidt hiervoor de namen van de variabelen met komma's en doet hetzelfde met de waarden
- Python kent dan de respectievelijke waarde toe aan elke genoemde variabele

Constanten

```
STUDENTEN_PER_GROEP = 4
```

- Een constante is zoals een variabele, maar met een waarde die niet verandert tijdens de looptijd van je programma
- Python heeft geen ingebouwde ondersteuning voor constanten maar developers gebruiken een naam bestaande uit enkel hoofdletters om aan te geven dat het hier over een niet te veranderen constante gaat

Expressies

Expressies

- Python ondersteunt rekenkundige en soortgelijke expressies of uitdrukkingen op een manier zoals dat in de wiskunde en de meeste andere programmeertalen gebeurt
- Standaardregels van rekenkundige voorrang zijn van toepassing
- Volgende code berekent het gemiddelde van 12 en 4, waarbij het resultaat aan variabele c wordt toegekend:

```
a = 3
b = 5
c = (a + b) / 2
```

Expressies

- Expressies met enkel gehele getallen leveren niet altijd een geheel getal op
- Een deling van gehele getallen geeft een kommagetal terug
- Wil je enkel het gehele gedeelte terugkrijgen als geheel getal, gebruik dan de //-operator
- Expressies hoeven niet alleen numerieke waarden te bevatten: strings, Booleaanse waarden en vele andere soorten objecten kunnen op verschillende manieren in expressies worden gebruikt. We zullen dit in de cursus gaandeweg kunnen vaststellen

Strings

Werken met tekst: strings

- Een string bestaat uit een reeks tekst-karakters
- Alles binnen aanhalingstekens wordt beschouwd als een string
- Je mag gebruik maken van enkele of dubbele aanhalingstekens
- Deze flexibiliteit laat toe om aanhalings- en afkappingstekens te gebruiken in je strings
- Wees standvastig in je keuzes

```
boodschap = "Dit is een string."
boodschap = 'Dit is ook een string.'
boodschap = 'Mijn vriend vroeg: "Programmeer jij ook in Python?" '
boodschap = "Ik hou van het schrijven van Python programma's"
```

Hoofdlettergebruik wijzigen in een string

```
naam = "Guido van Rossum"
print(naam.title()) # Elk woord laten beginnen met een hoofdletter: Guido Van Rossum
print(naam.upper()) # Alles in hoofdletters: GUIDO VAN ROSSUM
print(naam.lower()) # Alles in kleine letters: guido van rossum
```

- We beschikken hiervoor over een aantal zgn. methods. Dit zijn acties die je op bepaalde data kan toepassen
- De dot-notatie (".") vertelt Python in het eerste vb. om de title-method uit te voeren op de variabele naam
- Elke method wordt gevolgd door haakjes, omdat methods vaak bijkomende informatie nodig hebben om hun werk te kunnen doen. Hier is dit evenwel niet het geval
- De lower()-method wordt vaak gebruikt om data op te slaan die door een gebruiker werd ingegeven

Variabelen gebruiken in een string: f-strings

```
voornaam = "Guido"
familienaam = "van Rossum"
volledige_naam = f"{voornaam} {familienaam}"
boodschap = f"Bedankt voor Python, {volledige_naam.title()}!"
print(boodschap)
```

- Om variabelen te gebruiken binnen strings plaats je de letter f onmiddellijk voor het eerste aanhalingsteken
- Je omringt de variabelen die je in de string gebruikt met accolades
- Python vervangt elke variabele door de waarde
- De f in f-strings staat voor "format"

Witruimte toevoegen aan strings

```
print("\tHelderheid")
print("Pythonisch programmeren:\nHelder\nGeloofwaardig\nEfficiënt")
print("Pythonisch programmeren:\n\tHelder\n\tGeloofwaardig\n\tEfficiënt")
```

- Met witruimte doelen we op niet-afdrukbare tekens als spaties, tabs en einde-lijnsymbolen
- Je gebruikt ze om je output op een beter leesbare manier te organiseren
- Een tab-insprong toevoegen aan je tekst doe je met "\t"
- Een nieuwe regel voeg je toe met "\n"
- Combinaties zijn mogelijk: "\n\t" zorgt ervoor dat Python op een nieuwe regel begint, met een tabinsprong

Witruimte elimineren binnen strings

```
boodschap = " Ik hou van Python "
print(boodschap.rstrip())
print(boodschap.lstrip())
print(boodschap.strip())
```

- Python maakt het eenvoudig om (eventueel) aanwezige witruimte te verwijderen
- Er kan een onderscheid worden gemaakt tussen witruimte links, rechts of aan beide zijden van een string
- Handig als je twee strings met elkaar wil vergelijken

Getallen / berekeningen maken

Werken met gehele getallen (integers)

```
getal1 = -2
getal2 = 3
print(getal1 + getal2) # 1
print(getal1 ** getal2) # -8
print((getal1 + getal2) * getal1) # -2
```

- Integers kunnen in Python opgeteld (+), afgetrokken (-), vermenigvuldigd (*) en gedeeld (/) worden
- Python gebruikt twee vermenigvuldigingssymbolen (**) om exponenten weer te geven
- Elke expressie kan meerdere bewerkingen bevatten. De volgorde van bewerkingen wordt gerespecteerd
- Je mag gebruik maken van haakjes om deze volgorde aan te passen

Werken met decimale getallen (floats)

```
getal1 = 0.4
getal2 = 0.2
print(getal1 + getal2) # 0.6000000000000001
print(getal1 * 2) # 0.8
```

- Python noemt elk kommagetal een float
- Deze term wordt in de meeste programmeertalen gebruikt en verwijst naar het feit dat een decimaalteken op elke positie in een getal kan voorkomen
- Houd er rekening mee dat je soms een willekeurig aantal decimalen in je antwoord kunt krijgen

Gehele en decimale getallen

```
print(9 / 3) # 3.0
print(13 + 2.0) # 15.0
print(4 * 5.0) # 20.0
print(2.0 ** 3) # 8.0
totaal_aantal_aardbewoners = 7_902_193_151
print(totaal_aantal_aardbewoners) # 7902193151
```

- Wanneer je twee getallen deelt is het resultaat altijd een float
- Python gebruikt standaard een float als resultaat voor elke bewerking die een float bevat, zelfs als de uitvoer een geheel getal is
- Bij grote getallen kun je cijfers groeperen met underscores om ze leesbaarder te maken

None

None

- None is een speciaal basisgegevenstype dat een enkel speciaal gegevensobject definieert met de naam None
- Wordt gebruikt om een lege waarde weer te geven. Vaak gebruikt als een tijdelijke aanduiding om een punt in een gegevensstructuur aan te geven waar uiteindelijk betekenisvolle gegevens zullen worden gevonden
- Je kan eenvoudig testen op de aanwezigheid van None, omdat er slechts één instantie van None is in het hele Python-systeem (alle verwijzingen naar None verwijzen naar hetzelfde object) en None is alleen gelijk aan zichzelf

```
var = None
if var is None:
    print("None")
else:
    print("Niet None")
```

Input van de gebruiker

Input van de gebruiker

- Je kan de functie input() gebruiken om invoer van de gebruiker te krijgen
- Gebruik de promptstring die u aan de gebruiker wilt tonen als invoerparameter:

```
naam = input("Naam? ")
print(naam)
leeftijd = int(input("Leeftijd? "))
print(leeftijd)
```

Input van de gebruiker

- Nadeel is dat de invoer binnenkomt als een string, dus als je het als een getal wilt gebruiken, moet je de functie int() of float() gebruiken om de string om te zetten naar een getal

```
basis = float(input("Basis in cm? "))
hoogte = float(input("Hoogte in cm? "))
oppervlakte = basis * hoogte
print(f"De oppervlakte van de rechthoek is {oppervlakte}cm2")
```

Python Development - les 1 - kristof.michiels01@ap.be

Python Development

If statements en loops

Kristof Michiels

Inhoud

- De code die we tot nog toe hebben gezien was volledig sequentiëel: de statements worden uitgevoerd van boven naar onder, één voor één
- In deze les zien we twee belangrijke technieken voor controle van de flow: if- statements en loops
- Deze bepalen of en onder welke voorwaarden (welke) code al dan niet mag worden uitgevoerd

If-statements

Basisvoorbeeld

```
weekdagen = ["maandag", "dinsdag", "woensdag", "donderdag", "vrijdag"]

for dag in weekdagen:
    if dag == "vrijdag":
        print(f"Thank god it's {dag}!")
    else:
        print(dag.title())
```

- We lopen hier een beetje vooruit: met een for-loop doorlopen we een lijst. Beiden komen heel snel aan bod
- Wat ons nu aanbelangt is het if-statement binnen de loop. We gaan ermee na of de huidige dag "vrijdag" is. Indien *True* drukken we een gepaste zin af. In elk ander geval drukken we gewoon de naam van de dag af
- Een beslissing dus! Let op het dubbelpunt na if en else en op de 4 spaties bij de statements die erop volgen

if-statements

- Een if-statement bevat een voorwaarde en een reeks statements die mogelijk kunnen worden uitgevoerd. Deze statements worden voorafgegaan door insprongen om aan te geven dat ze tot het if-statement horen
- De voorwaarde wordt geëvalueerd en indien *True* worden de statements effectief uitgevoerd; indien *False* worden deze statements overgeslagen
- Vervolgens gaat het programma verder met de instructies onder het if-statement

```
getal = 24
if getal % 2 == 0:
    print(f"Het getal {getal} is deelbaar door 2")
    # hier kunnen nog extra statements staan
print("En het programma loopt door...")
```

De voorwaarde binnen een if-statement

De voorwaarde die kan gaan van eenvoudig tot complex evalueert in *True* of *False*. We noemen die voorwaarde een Booleaanse expressie, genoemd naar wiskundige George Boole. Ze bevat meestal een relationele operator die twee waarden, variabelen of complexe expressies vergelijkt. De operatoren kunnen de volgende zijn:

- < Kleiner dan
- <= Kleiner dan of gelijk aan
- > Groter dan
- >= Groter dan of gelijk aan
- == Gelijk aan
- != Niet gelijk aan

Meerdere if-statements

```
getal = int(input("Geef een geheel getal aub: "))
if getal > 0:
    boodschap = "groter dan 0"
if getal <= 0:
    boodschap = "kleiner dan of gelijk aan 0"
print(f"Het getal is {boodschap}")
```

- Bovenstaand voorbeeld vraagt een geheel getal aan de gebruiker en gebruikt twee if-statements om na te gaan of dat getal groter of kleiner/gelijk is aan 0
- We maken hier tweemaal gebruik van een if-statement met twee voorwaarden die elkaar uitsluiten. Dit is toegelaten, maar het kan efficiënter, namelijk met een if-else statement...
- Let op! Meerdere if-statements kunnen perfect ok zijn indien de voorwaarden elkaar niet uitsluiten

If-else-statements

- Een if-else-statement heeft een extra else-gedeelte (zonder voorwaarde!) met eigen blok instructies
- Wanneer de if-voorwaarde wordt geëvalueerd zal bij *True* de eerste blok code worden uitgevoerd, en bij *False* de tweede. Er zal steeds één en enkel één blok code worden uitgevoerd
- Je gebruikt een if-else-statement ipv 2 if-statements als de voorwaarden elkaar uitsluiten. Dit is efficiënter: slechts één voorwaarde moet worden gecontroleerd

```
getal = int(input("Geef een geheel getal aub: "))
if getal > 0:
    boodschap = "groter dan 0"
else:
    boodschap = "kleiner dan of gelijk aan 0"
print(f"Het getal is {boodschap}")
```

If-elif-else-statements

```
getal = int(input("Geef een geheel getal aub: "))
if getal > 0:
    boodschap = "groter dan 0"
elif getal < 0:
    boodschap = "kleiner dan 0"
else:
    boodschap = "gelijk aan 0"
print(f"Het getal is {boodschap}")
```

- Gebruik if-elif-else wanneer je meer dan twee mogelijkheden wil bieden
- Begint met een if-gedeelte gevolgd door één of meer elif-gedeeltes, gevolgd door een else (zonder voorwaarde). Elk gedeelte beschikt over een door 4 spaties voorafgegaan codeblok

If-elif-else-statements

```
leeftijd = 16
if leeftijd < 4:
    ticketprijs = 2
elif leeftijd < 12:
    ticketprijs = 10
elif leeftijd < 26:
    ticketprijs = 18
else:
    ticketprijs = 26
print(f"Jouw toegang tot de zoo kost ${ticketprijs}.")
```

- De voorwaarden worden van boven naar onder één voor één geëvalueerd: van zodra een *True* is gevonden wordt de betreffende codeblok uitgevoerd en wordt de rest van het statement overgeslagen
- Indien geen *True* wordt gevonden voert de interpreter de code van het else-blok uit

If-elif-statements

```
leeftijd = 16
if leeftijd < 4:
    ticketprijs = 2
elif leeftijd < 12:
    ticketprijs = 10
elif leeftijd < 26:
    ticketprijs = 18
print(f"Jouw toegang tot de zoo kost ${ticketprijs}.")
```

- Het else-gedeelte op het einde mag worden weggelaten
- Het is dan mogelijk dan van alle codeblokken geen enkele wordt uitgevoerd, omdat geen enkele voorwaarde in *True* resulteert
- Bovenstaand voorbeeld is ok, maar wat met een bezoeker van 26 of ouder?

Geneste if-instructies

```
leeftijd, sociaal_tarief  = 45, True
if leeftijd < 4:
    ticketprijs = 0
elif leeftijd < 12:
    ticketprijs = 10
elif leeftijd < 26:
    ticketprijs = 18
else:
    if sociaal_tarief:
        ticketprijs = 12
    else:
        ticketprijs = 24
```

De codeblokken binnen de if-*-statements kunnen (zo goed als) elke Python-code bevatten. Dat kunnen ook andere if, if-else, if-elif of if-elif-else statements zijn. We noemen dit geneste if-instructies.

Booleaanse operatoren

- Een Booleaanse expressie is een uitdrukking die in *True* of *False* resulteert. Deze kunnen bevatten:
  - True, False, relationele operatoren, functie-aanroepen die True of False teruggeven
  - Python kent ook 3 Booleaanse operatoren: not, and en or
    - not keert de waarde van een Booleaanse expressie om: True wordt False en False wordt True
    - and en or combineren Booleaanse waarden tot een Booleaans resultaat
      - Bij and moeten alle onderdelen True zijn om in True te resulteren: True and True => True; False and True => False; False and False and True => False
      - Bij or moet minstens één onderdeel True zijn om in True te resulteren: False or True => True; False or False or True => True; False or False => False

Booleanse logica: een voorbeeld

```
naam = input("Geef je naam in aub: ")
leeftijd = int(input("Geef je leeftijd in aub: "))
if naam == "Kristof" or naam == "Bart":
    print("Leuk, je bent een Kristof of een Bart!")
elif naam == "David" and leeftijd == 28:
    print("Wat een toeval: een 28-jarige David!")
else:
    print(f"Welkom {naam}")
```

Loops

Herhalen met een while- of for-loop

- Indien je een bepaalde taak meerdere keren wil uitvoeren dan kan je gebruik maken van één van Pythons beschikbare loop-constructies: een while-loop of een for-loop
- Beiden laten toe dezelfde groep statements meerdere keren uit te voeren
- Gebruik je ze efficiënt dan kan je complexe zaken uitvoeren (zoals berekeningen) in slechts een zeer beperkt aantal statements

De while-loop: basisvoorbeeld

```
getal = 1
while getal <= 5:
    print(getal)
    getal += 1
```

De while-loop: basisvoorbeeld

```
getal = int(input("Geef een geheel getal aub (0 om de app te verlaten): "))
while getal != 0:
    if getal > 0:
        print("Dat is een positief getal")
    else:
        print("Dat is een negatief getal")
    getal = int(input("Geef een geheel getal aub (0 om de app te verlaten): "))
```

De while-loop

- Zoals je hebt kunnen vaststellen laat een while-loop een codeblok (indentatie met 4 spaties) uitvoeren zo lang een voorwaarde resulteert in *True*
- Wanneer de laatste regel van de codeblok is bereikt wordt teruggekeerd naar het begin van de loop en wordt de voorwaarde opnieuw geëvalueerd, en indien *True*, de codeblok opnieuw uitgevoerd
- Dit duurt tot de loop-voorwaarde resulteert in *False*. Als dit gebeurt wordt de codeblok niet meer uitgevoerd en gaat het programma verder met het eerste statement na de loop
- De loop-voorwaarde kan wijzigen door gebruikersinput of door code binnen de loop-codeblok. Zorg er elk geval voor dat de voorwaarde ooit *False* teruggeeft, anders belandt je programma in een *infinite loop*

For-loops

- For-loops zorgen er net als while-loops voor dat een bepaalde codeblok verschillende keren na elkaar kan worden uitgevoerd
- Het mechanisme dat de loop bepaalt is evenwel verschillend: een for-loop in Python wordt uitgevoerd voor elk item in een collectie
- Deze collectie kan een range van gehele getallen zijn, de letters in een string, of zoals we later zullen zien: de waarden opgeslagen in een datastructuur zoals een list

De for-loop: basisvoorbeelden

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

```
lijst = ['aap', 'noot', 'peer']
for i in lijst:
    print(i)
```

```
getal = int(input("Geef een geheel getal: "))
print(f"De veelvouden van 3 tot en met {getal} zijn:")
for i in range(3, getal + 1, 3):
    print(i)
```

For-loops

- de codeblok (met indentatie van 4 spaties) binnen de for-loop bevat statements die meerdere keren uitgevoerd kunnen worden, meer bepaald voor elk item in de collectie
- Elk element in de collectie wordt gekopieerd naar een variabele vooraleer de loop de codeblok uitvoert voor dit element
- De variabele kan gebruikt worden in de codeblok zoals elke andere variabele

Range()

- Een collectie van integers kan geconstrueerd worden door aanroepen van de range()-functie
  - Roep je deze functie aan met 1 argument dan start de range met 0 en eindigt met het argument -1
  - Wanneer twee argumenten zijn meegegeven dan start de range met het eerste argument en eindigt met het tweede argument -1. range(2,5) bvb geeft een range terug van 2, 3 en 4
  - Een lege range wordt teruggegeven als het eerste argument groter is dan het tweede. De codeblok in de loop wordt in dit geval nooit uitgevoerd
  - Range() kent ook nog een derde argument, de step (een geheel getal). Is deze groter dan 0 dan begint de range bij het eerste argument en loopt tot argument2 - 1, telkens in sprongen gelijk aan de stepwaarde
  - Een negatieve stepwaarde zorgt ervoor dat een collectie van afnemende waarden ontstaat. range(0, -4, -1) geeft een range terug die bestaat uit 0, −1, −2 en −3

Geneste loops

- De statements binnen de codeblok van een loop kunnen op hun beurt een loop bevatten. We noemen dit geneste loops
- Elk type loop kan genest zitten binnen een ander type buitenloop

```
boodschap = input("Geef een boodschap (laat leeg om te stoppen): ")
while boodschap != "":
    aantal = int(input("Hoe vaak moet dit herhaald worden? "))
    for i in range(aantal):
        print(boodschap)
    boodschap = input("Geef een boodschap (laat leeg om te stoppen): ")
```

Python Development

Functies

Kristof Michiels

Wat zijn functies?

Wat zijn functies?

- Functies helpen om de complexiteit in een programma beheersbaar te houden
- Ze dienen meerdere doelen:
  - ze maken het mogelijk om code op te splitsen in kleinere units
  - ze laten ons toe code eenmaal te schrijven en vanop verschillende plaatsen aan te roepen wanneer nodig
  - ze laten toe om de onderdelen van onze toepassing individueel te testen

Wat zijn functies?

- Ze doen dat door een reeks statements af te zonderen voor later meermaals gebruik
- Een functie wordt uitgevoerd door ze aan te roepen vanuit je code
- Na het uitvoeren wordt teruggekeerd naar de locatie waar de functie werd aangeroepen en gaat de uitvoering van het programma verder

Wat zijn functies?

- We hebben in de eerste week reeds functies gebruikt: bvb input(), print(), int() en float()
- Het zijn functies die Python ons standaard biedt en kunnen vanuit elk Python programma worden aangeroepen
- Bovenstaande voorbeelden hebben één en slechts één helder doel: dat is ook hoe jouw functies horen te zijn!
- We leren in dit document onze eigen functies te schrijven en aan te roepen

De basis van functies

Basisvoorbeeld

```
def groet_gebruiker():
    """Toon een eenvoudige begroeting"""
    print("Hallo daar!")

groet_gebruiker()
```

- We definiëren functies met het trefwoord *def*, gevolgd door de naam van de functie en (voorlopig lege) haakjes en tot slot een dubbelpunt
- Op deze regel volgen (met telkens een insprong van 4 spaties) een reeks statements die zullen worden uitgevoerd wanneer de functie wordt aangeroepen
- De haakjes kunnen argumenten bevatten, dat is informatie die aan de functie van buitenaf wordt meegegeven

Basisvoorbeeld

```
def groet_gebruiker():
    """Toon een eenvoudige begroeting"""
    print("Hallo daar!")

groet_gebruiker()
```

- Merk in het vb op dat je in je code de functie moet aanroepen. Doe je dit niet, dan wordt ze nooit uitgevoerd
- Je kiest steeds voor een naam die helder beschrijft wat binnen de functie gebeurt
- Een goed gekozen functienaam bevat vaak een werkwoord (actie) en zelfstandig naamwoord
- Gebruik geen hoofdletters en scheid woorden door een underscore

Basisvoorbeeld

```
def groet_gebruiker():
    """Toon een eenvoudige begroeting"""
    print("Hallo daar!")

groet_gebruiker()
```

- Elke functie hoort een korte beschrijving te hebben van wat de functie doet
- Deze beschrijving komt onmiddellijk na de functie-definitie en gebruikt het docstring format
- Een goedgedocumenteerde functie zorgt ervoor dat andere programmeurs de functie kunnen gebruiken enkel en alleen door het lezen van de docstring

De basis van functies

```
def groet_gebruiker():
    """Toon een eenvoudige begroeting"""
    print("Hallo daar!")

for _ in range(5):
    groet_gebruiker()

print("De code (en het leven) gaan verder")
```

- Een functie kan zo vaak als nodig worden aangeroepen vanop verschillende plaatsen binnen de toepassing
- In bovenstaand voorbeeld een aantal keer binnen een loop. We gebruiken hier "_" als teller omdat we de for-lus louter als loop willen gebruiken en geen variabele teller nodig hebben binnen onze code

Informatie meegeven met parameters

Functies met parameters

- Onze voorbeeldfunctie miste flexibiliteit: stel dat je een andere boodschap wil meegeven?
- We kunnen de mogelijkheden en de flexibiliteit vergroten door het voorzien van één of meer argumenten
- De functie ontvangt die argumentwaarden in de vorm van parameter-variabelen die toegevoegd worden binnen de haakjes wanneer de functie wordt gedefinieerd
- Het aantal parameter-variabelen die in de functie staan beschreven geven aan hoeveel argumenten moeten worden meegegeven wanneer de functie wordt aangeroepen

Functies met parameters

```
def groet_gebruiker(naam_gebruiker):
    """Toon een gepersonaliseerde begroeting"""
    print(f"Hallo daar {naam_gebruiker.title()}!")

groet_gebruiker("Anneleen")
```

- In bovenstaand voorbeeld werd één parameter toegevoegd: *naam_gebruiker*
- Eventuele meerdere parameters worden gescheiden door een komma
- De functie wordt hier aangeroepen met het argument "Anneleen"
- Binnen de functie kunnen de waarden van de parameter variabelen worden gebruikt wanneer nodig

Functies met parameters

```
def teken_kader(breedte, hoogte):
    """Deze functie tekent een kader, vanaf een breedte x hoogte van 2x2 """
    if breedte < 2 or hoogte < 2:
        print("Ik kan de rechthoek niet tekenen: breedte of hoogte zijn te klein")
        quit()
    print("#" * breedte)
    for i in range(hoogte - 2):
        print("#" + " " * (breedte - 2) + "#")
    print("#" * breedte)

teken_kader(8, 6)
teken_kader(4, 4)
```

- In dit voorbeeld moeten 2 argumenten worden voorzien telkens de functie wordt aangeroepen omdat de functie-definitie dit vereist

Functies met parameters

```
def teken_kader(breedte, hoogte, teken_kader, teken_vulling):
    """Deze functie tekent een kader en gebruikt vier argumenten """
    print(teken_kader * breedte)
    for i in range(hoogte - 2):
        print(teken_kader + teken_vulling * (breedte - 2) + teken_kader)
    print(teken_kader * breedte)

teken_kader(8, 6, "*", "/")
```

- Wanneer de functie wordt uitgevoerd zal de waarde van het eerste argument worden gekoppeld aan de eerste parameter, en de waarde van het tweede argument aan de tweede parameter
- Wil je de functie nog meer flexibiliteit geven, dan kan je meer parameters gaan gebruiken (in dit vb: 4)

Default-waarden

- Wil je de functie aanroepen, dan zal je telkens 4 parameters moeten voorzien
- Wanneer bepaalde waarden frequent worden gebruikt dan kan je standaard (*default*) waarden voor parameters meegeven aan de functie-definitie. Parameters met default komen steeds achteraan
- De functie kan vanaf dan aangeroepen worden met 2, 3 of 4 argumenten. Bij 3 of 4 argumenten worden de default waarden overschreven.

```
def teken_kader(breedte, hoogte, teken_kader="#", teken_vulling=" "):
    ...

teken_kader(8, 6)
teken_kader(8, 6, "@")
teken_kader(8, 6, "#", ".")
```

Positionele vs trefwoord-argumenten

```
teken_kader(breedte=8, hoogte=6)
teken_kader(breedte=8, hoogte=6, teken_kader="@")
teken_kader(breedte=8, hoogte=6, teken_kader="#", teken_vulling=".")
```

- Een trefwoord-argument is een naam-waarde-paar dat je meegeeft aan een functie
- Je koppelt daarmee onmiddelijk een waarde aan een parameter
- Je geeft explicieter aan wat bij wat hoort en moet de volgorde van de parameters niet meer respecteren

Variabelen in functies en return-waarden

Variabelen in functies

- Variabelen kunnen ook gecreëerd worden binnen een functie
- Deze bestaan dan enkel binnen de functie, wanneer de functie wordt uitgevoerd
- De variabele houdt op te bestaan wanneer de functie eindigt en kan na afloop niet meer gebruikt worden
- We noemen dit lokale variabelen en spreken van een lokale *scope* (of reikwijdte)
- In onderstaand voorbeeld kan de variabele exponent niet opgevraagd worden buiten de functie

```
def bereken_macht(grondtal)
    exponent = 3
    print(f"De {exponent}-de macht van {grondtal} is {grondtal ** exponent}")
```

Return-waarden

- De statements in de functies die we hebben gezien zorgden met de print()-functie voor een resultaat
- We konden met argumenten het resultaat van de functie beïnvloeden, maar er was geen verdere communicatie met de code buiten de functie
- Soms zal je als resultaat van een functie een resultaat berekenen dat later in het programma en buiten de functie zal worden gebruikt

Return-waarden: een voorbeeld

```
def groet_gebruiker(naam_gebruiker):
    """Toon een gepersonaliseerde begroeting"""
    return f"Hallo daar {naam_gebruiker.title()}!"

print(groet_gebruiker("Anneleen"))
```

- We doen dit door de functie een return waarde te laten teruggeven met het return trefwoord gevolgd door de waarde die wordt teruggegeven
- Wanneer het return statement uitgevoerd is eindigt de functie meteen en wordt de controle teruggegeven aan de locatie waar de functie werd aangeroepen

Return-waarden

- Functies die waarden teruggeven met return staan vaak aan de rechterkant van een toekennings-statement (bvb getal = mijn_functie())
- Ze kunnen evenwel ook in andere contexten functioneren waarin een waarde nodig is: bvb bij een voorwaarde in een if-statement of een while-loop, of als argument voor een andere functie
- Een functie die geen waarde teruggeeft moet het return trefwoord niet gebruiken
- Je kan het return trefwoord evenwel gebruiken zonder waarde om ergens een einde van een functie te forceren
- Elke functie kan meerdere return statements bevatten (al dan niet met een waarde). Van zodra de interpreter een return tegenkomt bij het uitvoeren van een functie, stopt de functie

Functies gebruiken uit andere bestanden

Functies gebruiken uit andere bestanden

- Functies helpen je je code te structuren en overzichtelijk te houden
- Bij grotere programma's worden functies vaak in afzonderlijke bestanden (we noemen deze modules) bijgehouden
- Wil je ze gebruiken dan importeer je de module of de functie in je programma
- Je maakt een functie beschikbaar binnen je programma met een import-statement
- Je gaat dit binnenkort ook vaak doen voor functies uit libraries en modules die je niet zelf geschreven hebt

Een volledige module importeren

In onderstaand voorbeeld hebben we twee bestanden: kleuren.py en gebruik_kleuren.py

```
# kleuren.py
def geef_kleur()
    return ...

def geef_complementaire_kleur(primaire_kleur)
    return ...
```

```
# gebruik_kleuren.py
import kleuren
print(kleuren.geef_complementaire_kleur("geel"))
```

Een volledige module importeren

- Een module is een bestand dat eindigt op .py. Hier wordt *kleuren.py* gebruikt als module
- Om de functionaliteit van de module beschikbaar te maken in *gebruik_kleuren.py* voegen we een import statement toe bovenaan het bestand
- We roepen de externe functies aan door eerst te verwijzen naar de module, en dan met een dot-notatie de functie te vernoemen: bvb. kleuren.geef_complementaire_kleur()

Specifieke functies importeren

- Het is ook mogelijk slechts één of enkele functie(s) te importeren. Je scheidt ze door een komma
- Met deze syntax hoef je de dot-notatie niet te gebruiken wanneer je de functie aanroept
- Dit is de beste en meest efficiënte aanpak

```
from module_naam import functie_naam
```

```
from module_naam import functie_0, functie_1, functie_2
```

```
# gebruik_kleuren.py
from kleuren import geef_complementaire_kleur
print(geef_complementaire_kleur("geel"))
```

"as" gebruiken om de functie een alias te geven

- Als de functie-naam conflicteert met een andere functie die je gebruikt (of te lang is) dan kan je gebruik maken van een alias:

```
from module_naam import functie_naam as fie
```

```
# gebruik_kleuren.py
from kleuren import geef_complementaire_kleur as ck
print(ck("geel"))
```

"as" gebruiken om de module een alias te geven

- Je kan ook een alias geven aan een module en zo tot kortere notaties komen:

```
import module_naam as md
```

```
# gebruik_kleuren.py
import kleuren as kl
print(kl.geef_complementaire_kleur("geel"))
```

Alle functies in een module importeren

- Je kan aangeven aan Python om elke functie in de module te importeren door gebruik te maken van de asterisk operator (*)
- Voordeel hier is dat je de dot-notatie niet hoeft te gebruiken
- Te vermijden bij gebruik van grote modules

```
from module_naam import *
```

```
# gebruik_kleuren.py
from kleuren import *
print(geef_complementaire_kleur("geel"))
```

Verhinderen dat bij importeren code in een module wordt uitgevoerd

- Doe je door de code in de module bvb in een main()-functie te stoppen
- Met een if-statement kan je dan checken of het bestand daadwerkelijk autonoom werd uitgevoerd, of dat het werd geïmporteerd
- Deze structuur moet toegepast worden telkens je een programma maakt dat functies bevat die mogelijk door een ander bestand zullen gebruikt worden

```
if __name__ == "__main__":
      main()
```

Python Development

Lists

Kristof Michiels

Inhoud

Inhoud

- Wat zijn lists?
- Elementen wijzigen, toevoegen en verwijderen
- Een list ordenen
- Een list doorlopen
- Werken met delen van een list
- Tuples

Wat zijn lists?

Wat zijn lists?

```
python_data_types = ["str", "int", "float", "complex", "list", "tuple", "range",
    "dict", "set", "frozenset", "bool", "bytes", "bytearray", "memoryview"]
boodschap = f"Een bekend data-type is de {python_data_types[0]}."
print(boodschap)
```

- Een list is één van de ingebouwde data-types in Python
- Zoals de naam het zegt worden lists gebruikt om verzamelingen met data op te slaan
- Het voordeel van een list is dat data gestructureerd kan worden opgeslagen: in volgorde, gesorteerd, enz...
- Een list wordt aangemaakt door middel van deze [ ] vierkante haakjes en de individuele waarden zijn gescheiden door komma's
- Als naam voor een list wordt vaak een meervoud gekozen

Toegang tot elementen in een list

```
weekdagen = ["maandag", "dinsdag", "woensdag", "donderdag", "vrijdag"]
print(weekdagen) #['maandag', 'dinsdag', 'woensdag', 'donderdag', 'vrijdag']
print(weekdagen[0])
print(weekdagen[1])
boodschap = f"Bijna weekend, het is vandaag {weekdagen[4]}."
print(boodschap)
```

- Vraag je Python een list te printen dan krijg je de volledige list terug, inclusief de vierkante haakjes
- *Lists* zijn geordende collecties, dus kan je via de positie toegang krijgen tot de elementen
- Je gebruikt hiervoor de naam van de list, gevolgd door het index-getal van het gewenste element geplaatst binnen vierkante haakjes
- Het index-getal van het eerste element in de lijst is 0, tweede element is 1 enz...

Toegang tot elementen in een list

```
weekdagen = ["maandag", "dinsdag", "woensdag", "donderdag", "vrijdag"]
boodschap = f"Bijna weekend, het is vandaag {weekdagen[-1]}."
print(boodschap)
```

- Het laatste element krijg je door het gebruik van het index-getal -1, het voorlaatste element met -2 enz...

Elementen wijzigen, toevoegen en verwijderen

Elementen wijzigen in een list

```
lievelingskleuren = ["oranje", "groen","rood"]
lievelingskleuren[0] = "geel"
```

- We kunnen elementen wijzigen door de naam van de lijst op te roepen, gevolgd door de index van het te wijzigen element
- Als we er vervolgens een waarde aan toekennen dan wordt dit de nieuwe waarde van dit element

Elementen toevoegen met de append()-method

```
lievelingskleuren = []
lievelingskleuren.append("oranje")
lievelingskleuren.append("groen")
lievelingskleuren.append("rood")
print(lievelingskleuren) #['oranje', 'groen', 'rood']
```

- Er zijn verschillende manieren om nieuwe data toe te voegen aan een bestaande list
- Met de append()-method voeg je elementen toe aan het einde van een list
- Deze method maakt het eenvoudig om dynamisch te vertrekken vanaf een lege list. Met elke append()-call voeg je een nieuw element toe aan de lijst

Elementen toevoegen met de insert()-method

```
lievelingskleuren = ["oranje", "groen", "rood"]
lievelingskleuren.insert(0, "geel")
print(lievelingskleuren) #['geel', 'oranje', 'groen', 'rood']
```

- De insert()-method laat toe elementen toe te voegen op elke positie in een list
- Je doet dit door de index en de waarde van het nieuwe element mee te geven
- In het bovenstaande voorbeeld voegen we een element toe bij positie 0. Alle huidige elementen schuiven hierdoor een plaats door naar rechts

Elementen verwijderen met het del statement

```
lievelingskleuren = ["oranje", "groen", "rood"]
del lievelingskleuren[0] #['groen', 'rood']
print(lievelingskleuren)
del lievelingskleuren[1]
print(lievelingskleuren) #['groen']
```

Ken je de positie van een element dan kan je het verwijderen met het del statement.

Elementen verwijderen met de pop()-method

```
lievelingskleuren = ["oranje", "groen", "rood"]
verwijderde_kleur = lievelingskleuren.pop()
print(lievelingskleuren) #['oranje', 'groen']
print(verwijderde_kleur) #rood
andere_verwijderde_kleur = lievelingskleuren.pop(0)
print(andere_verwijderde_kleur) #oranje
```

- De pop()-method verwijdert standaard het laatste item in een list
- Geef je tussen de haakjes de index van een element mee dan verwijder je een specifiek element
- Deze method laat toe met dit element te werken na de verwijdering. Heb je geen plannen om dit te doen, verkies dan het del statement

Een item verwijderen op waarde met de remove()-method

```
lievelingskleuren = ["oranje", "groen", "rood", "paars"]
lievelingskleuren.remove("groen")
print(lievelingskleuren) #['oranje', 'rood', 'paars']
```

- Ken je de positie van een element niet, maar enkel de waarde? Dan kan je gebruik maken van de remove()-method
- De remove()-method verwijdert enkel het eerste element in de list met de betreffende waarde
- Als de waarde vaker kan voorkomen, dan is een loop noodzakelijk

Een list ordenen

Een list ordenen met de sort()-method

```
eighties_popmuziek = ["Prince", "Abba", "Sade", "Madonna"]
eighties_popmuziek.sort()
print(eighties_popmuziek) #['Abba', 'Madonna', 'Prince', 'Sade']
eighties_popmuziek.sort(reverse=True)
print(eighties_popmuziek) #['Sade', 'Prince', 'Madonna', 'Abba']
```

- De sort()-method maakt alfabetisch sorteren eenvoudig
- Let wel: de volgorde van de list wijzigt permanent, dus de oorspronkelijke volgorde gaat verloren
- We kunnen ook in omgekeerde alfabetische volgorde sorteren met het argument reverse=True

Een list ordenen met de sorted()-functie

```
eighties_popmuziek = ["Prince", "Abba", "Sade", "Madonna"]
print(sorted(eighties_popmuziek)) #['Abba', 'Madonna', 'Prince', 'Sade']
print(eighties_popmuziek) #['Prince', 'Abba', 'Sade', 'Madonna']
```

- Om een lijst te sorteren zonder de oorspronkelijke volgorde verloren te laten gaan kan je de sorted() functie gebruiken
- Ook deze functie accepteert een reverse=True argument om omgekeerd alfabetisch te ordenen

Een list "omdraaien" met de reverse()-method

```
eighties_popmuziek = ["Prince", "Abba", "Sade", "Madonna"]
eighties_popmuziek.reverse()
print(eighties_popmuziek)
```

- Het omdraaien van de oorspronkelijke volgorde van een list doe je door gebruik van de reverse()-method
- De oorspronkelijke volgorde gaat verloren, maar je kan deze ongedaan maken door de reverse()-method nogmaals toe te passen

De lengte van een list bepalen met de len()-functie

```
eighties_popmuziek = ["Prince", "Abba", "Sade", "Madonna"]
print(len(eighties_popmuziek)) #4
```

Gebruik de len()-functie om het aantal elementen in een list te bepalen

Een list doorlopen

Een list doorlopen

```
lievelingskleuren = ["oranje", "groen", "rood", "paars"]
for kleur in lievelingskleuren:
    print(f"Nog een mooie kleur: {kleur.title()}!")
```

- Vaak willen we een list doorlopen om op de elementen een bepaalde taak (of taken) te verrichten
- We kunnen hiervoor een for-loop gebruiken
- Gebruik van een enkelvoud en meervoud-naamgeving ("for item in lijst_van_items") maakt je code beter leesbaar

Numerieke lists maken met de range()-functie

```
for waarde in range(1, 10):
    print(waarde)
```

```
getallen = list(range(1, 10))
print(getallen) #[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Lists zijn bijzonder geschikt om reeksen getallen op te slaan
- De range()-functie maakt het gemakkelijk om een reeks getallen te genereren
- Het bovenste voorbeeld zal de getallen van 1 tot en met 9 afdrukken
- Je kan het resultaat van range() onmiddellijk capteren in een list door gebruik te maken van een list()-functie

Numerieke lists maken met range()

```
even_getallen = list(range(2, 15, 2))
print(even_getallen) #[2, 4, 6, 8, 10, 12, 14]
```

```
kwadraten = []
for waarde in range(1,10):
    kwadraten.append(waarde**2)
print(kwadraten) #[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

- We kunnen de range()-functie ook een derde argument meegeven
- Python gebruikt die waarde als een stapgrootte bij het genereren van getallen
- Met de range()-functie kan bijna elke denkbare set van getallen worden gecreëerd

Handige functies bij het werken met numerieke lists

```
getallenreeks = [0, 0, 1, 0, 2, 0, 2, 2, 1, 6, 0, 5]
print(min(getallenreeks)) #0
print(max(getallenreeks)) #9
print(sum(getallenreeks)) #19
```

Met behulp van deze functies kan je heel eenvoudig het minimum, maximum of de som van een numerieke list vinden

Verkorte notatie

```
kwadraten = [waarde**2 for waarde in range(1, 10)]
```

- In bovenstaande code combineer je de for loop en de creatie van nieuwe elementen in één en dezelfde regel code, en voeg je deze automatisch toe aan een lijst
- We noemen dit list comprehensions
  - nieuwe_lijst = [expressie for element in collectie]

Werken met delen van een list

Een list "slicen"

```
lievelingskleuren = ["oranje", "groen", "rood", "paars"]
print(lievelingskleuren[0:3]) #['oranje', 'groen', 'rood']
print(lievelingskleuren[1:4]) #['groen', 'rood', 'paars']
print(lievelingskleuren[:4]) #['oranje', 'groen', 'rood', 'paars']
print(lievelingskleuren[2:]) #['rood', 'paars']
print(lievelingskleuren[-3:]) #['groen', 'rood', 'paars']
```

- In Python verwijzen we naar een specifieke groep elementen in een list met de term "slice"
- Een slice maak je door de index van het eerste en het laatste element waarmee je wil werken mee te geven
- Net zoals bij range() stopt Python één element voor de tweede index die je hebt meegegeven
- Laat je de eerste index in een slice weg dan wordt automatisch gestart bij het eerste element van de list

Een list "slicen"

- Op gelijkaardige wijze wordt tot het einde van de list gegaan als je de tweede index weglaat
- Een negatieve index verwijst naar een element op zoveel posities van het einde van een list
- Een derde waarde geeft mee hoeveel elementen telkens mogen worden overgeslagen

Een slice van een list doorlopen

```
lievelingskleuren = ["oranje", "groen", "rood", "paars"]
for kleur in lievelingskleuren[:3]:
    print(kleur) # oranje groen rood
```

- Met een for-loop kan je een slice doorlopen

Een kopie maken van een list

```
lievelingskleuren = ["oranje", "groen", "rood", "paars"]
lievelingskleuren_kopie = lievelingskleuren[:]
lievelingskleuren.append("geel")
lievelingskleuren_kopie.append("magenta")
print(lievelingskleuren) #['oranje', 'groen', 'rood', 'paars', 'geel']
print(lievelingskleuren_kopie) #['oranje', 'groen', 'rood', 'paars', 'magenta']
```

Om een list te kopiëren maak je een slice die de oorspronkelijke list bevat door beide indexen weg te laten ([:])

Tuples

Tuples

```
mijn_tuple = (100, 30, 60)
print(mijn_tuple[0]) # 100
print(mijn_tuple[1]) # 30
```

- Een tuple is een list met het verschil dat een tuple onveranderlijk (of immutable) is
- Gebruik tuples wanneer je een reeks van waarden wil opslaan die niet veranderen tijdens de levenscyclus van het programma
- Ziet er exact uit als een list, maar wordt weergegeven met ronde haakjes
- Definieer je een tuple met één enkel element dan moet je een komma gebruiken na het eerste en enige element: mijn_tuple = (5,)

Een tuple overschrijven

```
mijn_tuple = (100, 30, 60)
mijn_tuple = (50, 30)
for getal in mijn_tuple:
    print(getal)
```

- Een tuple wijzigen gaat niet
- Je kan evenwel een nieuwe waarde toekennen aan de variabele die de tuple vertegenwoordigt
- Je kan een tuple doorlopen net zoals elke andere list

Python Development

Dictionaries

Kristof Michiels

Inhoud

- Wat zijn dictionaries?
- Werken met dictionaries
  - Toegang tot values
  - Een key-value-paar toevoegen
  - Een value wijzigen
  - Key-value-paren verwijderen

Inhoud

- Door een dictionary itereren met een loop
  - Door alle key-value-paren itereren
  - Itereren door alle keys (in een bepaalde volgorde)
  - Itereren door alle values
- Nesten
  - Een list van dictionaries
  - Een list in een dictionary
  - Een dictionary in een dictionary
- Dictionaries, lists en functies

Wat zijn dictionaries?

Wat zijn dictionaries?

- Net als lists laten dictionaries toe om verschillende (tot heel veel!) waarden toe te kennen aan één variabele
- In dit voor developers zeer belangrijk datatype sla je data op onder de vorm van key-value (sleutel-waarde) paren
- Waarden zijn voorzien van een unieke key, die een daarmee structuur en leesbaarheid aan je data geven
- Keys kunnen integers zijn, maar ook floats, strings, booleans, tot zelfs lists of dictionaries
- Bij getallen als key: de volgorde of startgetal zijn willekeurig
- De waarden hoeven niet uniek te zijn

Enkele voorbeelden van dictionaries

```
kunstenaars_geboortejaar = {
    "Keith Haring": 1958,
    "Jean-Michel Basquiat": 1960,
    "Andy Warhol": 1928,
}

andy_warhol = {
    "geboren": 1928,
    "gestorven": 1987,
    "stroming": "pop-art",
    "quote": "In the future everyone will be famous for fifteen minutes",
}

keith_haring = {} # een nieuwe, lege dictionary

print(type(kunstenaars_geboortejaar)) #<class 'dict'="">
```

Dictionaries

- Een dictionary-notatie bestaat uit accolades met één of meerdere key-value-paren erin
- Elk nieuw paar wordt toegevoegd aan het einde van de dictionary
- De key-value paren worden in de volgorde opgeslagen waarin ze werden toegevoegd (sinds Python 3.7)
- Het verwijderen van een key-value paar verandert niets aan de volgorde van de overblijvende paren
- Een nieuwe - lege - dictionary wordt aangemaakt en toegekend aan een variabele door gebruik te maken van accolades
- Een niet-lege dictionary kan worden gecreëerd door de key-value-waarden te scheiden door komma's
- Een dubbelpunt wordt als notatie gebruikt om elk key-value paar te scheiden

Werken met dictionaries

Toegang tot values: via de key

```
student_1 = {"favoriete_taal": "Python"}
print(student_1["favoriete_taal"])

student_2 = {"favoriete_taal": "JavaScript", "favoriete_serie": "Yellowjackets"}
print(student_2["favoriete_serie"])
```

Toegang tot de values kan ook met de get()-method. Voordeel is dat je een default waarde kan meegeven als de opgevraagde key niet zou bestaan. Indien je het tweede argument niet meegeeft, dan wordt indien de key niet bestaat de waarde None teruggegeven:

```
student_1 = {"favoriete_taal": "Python"}
print(student_1.get("favoriete_serie", "Geen favoriete serie meegegeven."))
```

Een key-value-paar toevoegen

```
student_1 = {"favoriete_taal": "Python"}
student_1["favoriete_serie"] = "Station Eleven"
student_1["favoriete_muziek"] = "Little Simz"

# of vertrekkend van een lege dictionary:

student_3 = {}
student_3["favoriete_taal"] = "Go"
student_3["favoriet_frontend_framework"] = "React"
```

Een value wijzigen

Je doet dit door te verwijzen naar de naam van de dictionary onmiddellijk gevolgd door de key tussen vierkante haakjes gevolgd door de nieuwe waarde die je eraan wenst te geven.

```
student_1 = {"favoriete_taal": "Python"}
student_1["favoriete_taal"] = "JavaScript"
```

Key-value-paren verwijderen

Met het del-trefwoord:

```
student_1 = {"favoriete_taal": "Python", "favoriete_kleur": "rood"}
del student_1["favoriete_kleur"]
```

Of met de pop()-method. De key moet steeds worden meegegeven. De value van deze key wordt teruggegeven en kan indien gewenst worden toegekend aan een value. Dit is hier te zien:

```
student_1 = {"favoriete_taal": "Python", "favoriete_kleur": "rood"}
kleur_van_student_1 = student_1.pop("favoriete_kleur")
```

De len()-functie

- Deze functie, die we al zagen bij lists, kan ook gebruikt worden om te weten te komen hoeveel key-value-paren een bepaalde dictionary bevat
- De dictionary wordt als enige argument meegegeven aan de functie
- Als resultaat komt het aantal key-value-paren terug. Bij een lege dictionary komt de waarde 0 terug

```
kunstenaars_geboortejaar = {
    "Keith Haring": 1958,
    "Jean-Michel Basquiat": 1960,
    "Andy Warhol": 1928,
}
print(len(kunstenaars_geboortejaar))
```

Door een dictionary itereren met een loop

Door een dictionary itereren met een loop

- Soms bevat een dictionary een groot aantal key-value-paren
- Bij groot mag je gerust aan 100000-en of zelfs nog veel meer denken
- Vaak ga je doorheen die collectie willen itereren en dat doe je met een loop

Door alle key-value-paren itereren

```
favoriete_talen = {
    "Kristof": "Python",
    "Tim": "C#",
    "Tom": "JavaScript",
}

for key, value in favoriete_talen.items():
    print(f"{key} heeft {value} als favoriete programmeertaal")
```

De in-operator

Wordt gebruikt om te bepalen of een bepaalde key (of value) aanwezig is. De operator evalueert in True als de key effectief aanwezig is, anders evalueert ze in False. Het resultaat kan overal gebruikt worden waar een Booleaanse waarde verwacht wordt (bvb in de voorwaarde van een if-statement of loop).

```
if "Keith Haring" in kunstenaars_geboortejaar.keys():
    print("Over Keith Haring hebben we al informatie toegevoegd.")

if 1967 not in kunstenaars_geboortejaar.values():
    print("We hebben nog geen kunstenaar geboren in 1967.")
```

Itereren door alle keys

```
favoriete_talen = {
    "Kristof": "Python",
    "Tim": "C#",
    "Tom": "JavaScript",
}

for key in favoriete_talen.keys():
    print(f"{key} heeft een favoriete programmeertaal")

#loopen door de keys is ook het default gedrag bij loopen door een dictionary
# dus hetzelfde als schrijven

for key in favoriete_talen:
    print(f"{key} heeft een favoriete programmeertaal")
```

Itereren door alle keys in een bepaalde volgorde

De natuurlijke volgorde van de elementen in een dictionary wordt bepaald door de volgorde waarin ze zijn toegevoegd. Wil je de keys evenwel sorteren dan kan je daar de sorted()-functie voor gebruiken.

```
favoriete_letters = {
    "o": "O is een otter, die zwemt in het meer",
    "k": "K is een koopman, die koffie verzond",
    "a": "A is een aapje, dat eet uit zijn poot",
}

for key, value in sorted(favoriete_letters.items()):
    print(value)
```

Itereren door alle values

```
favoriete_talen = {
    "Kristof": "Python",
    "Tim": "C#",
    "Tom": "JavaScript",
    "Hassan": "Python",
}

for favoriete_taal in favoriete_talen.values():
    print(favoriete_taal)
```

Herhalingen vermijden door gebruik van een set

```
favoriete_talen = {
    "Kristof": "Python",
    "Tim": "C#",
    "Tom": "JavaScript",
    "Hassan": "Python",
}

for favoriete_taal in set(favoriete_talen.values()):
    print(favoriete_taal)

# een set maken:

kleuren = {"blauw", "zwart", "geel", "oranje"}
```

Herhalingen vermijden door gebruik van een set

- Een set is een verzameling van elementen waarbij elk element uniek moet zijn. De set()-functie identificeert de unieke elementen in een lijst en bouwt een set uit die elementen
- Sets lijken wat op dictionaries omdat ze verpakt zitten in accolades. Ze bevatten evenwel geen key-value-paren maar enkelvoudige waarden

Nesten

Nesten

- Lists en dictionaries zijn zeer veelzijdige datatypes waarmee je veel richtingen uitkunt
- Zo kan je verschillende dictionaries onderbrengen in een list, of een list gebruiken als value gebruiken in een dictionary
- Je kan zelfs dictionaries gebruiken als values in een dictionary
- Dit noemen we nesten

Een list van dictionaries

```
kristof = {"favoriete_taal": "Python", "favoriete_muziek": "The War on Drugs"}
tim = {"favoriete_taal": "C#", "favoriete_muziek": "Lana Del Rey"}
tom = {"favoriete_taal": "JavaScript", "favoriete_muziek": "Admiral Freebee"}

lectoren = [kristof, tim, tom]
```

Uiteraard zou dit ook als één enkel statement kunnen geschreven worden.

Een list in een dictionary

```
kristof = {
    "favoriete_taal": "Python",
    "favoriete_muziek": "The War on Drugs",
    "favoriete_series": ["Yellowjackets", "Station Eleven", "Righteous Gemstones"],
}

for serie in kristof["favoriete_series"]:
    print(serie)
```

Een list in een dictionary

```
lotto_trekkingen = {
    "2022-02-19": [6,15,18,30,39,41,2],
    "2022-02-16": [2,3,14,18,31,39,45],
    "2022-02-12": [2,3,19,30,37,40,15],
}

for trekking, resultaten in lotto_trekkingen.items():
    print(f"{trekking}: ")
    for resultaat in resultaten:
        print(f"\t{resultaat}")
```

Een dictionary in een dictionary

```
lectoren = {
    "Kristof": {
        "favoriete_taal": "Python",
        "favoriete_muziek": "The War on Drugs",
        "favoriete_serie": "Better Call Saul",
    },
    "Tim": {
        "favoriete_taal": "C#",
        "favoriete_muziek": "Lana Del Rey",
        "favoriete_serie": "Station Eleven",
    },
    "Tom": {
        "favoriete_taal": "JavaScript",
        "favoriete_muziek": "Admiral Freebee",
        "favoriete_serie": "The Righteous Gemstones",
    },
}
```

Dictionaries, lists en functies

Dictionaries, lists en functies

- Dictionaries (en lists) kunnen net als andere datatypes als argumenten aan functies worden meegegeven, of als returnwaarde worden teruggegeven
- Dit laat ons toe om meerdere of een variërend aantal waarden mee te geven of te ontvangen

Een willekeurig aantal argumenten meegeven

- Python laat toe dat functies een willekeurig aantal argumenten kunnen ontvangen
- De asterisk in de parameternaam draagt Python op om een tuple aan te maken en alle waarden die meegegeven worden in deze tuple op te slaan

```
def geef_favoriete_series(*series):
    """Deze functie drukt een reeks favoriete tv-series af """
    for serie in series:
        print(serie)

geef_favoriete_series("Yellowjackets")
geef_favoriete_series("Yellowjackets", "Station Eleven", "The Righteous Gemstones")
```

Positionele en willekeurige argumenten combineren

- Positionele en willekeurige argumenten kunnen zonder problemen worden gecombineerd
- De parameter die een willekeurig aantal argumenten accepteert moet steeds als laatste in de functie-definitie worden geplaatst

```
def geef_favoriete_series(naam_kijker, *series):
    """Deze functie drukt een reeks favoriete tv-series af """
    print(naam_kijker)
    for serie in series:
        print(serie)

geef_favoriete_series("Kristof", "Yellowjackets")
geef_favoriete_series("Tim", "Yellowjackets", "Station Eleven", "Ted Lasso")
```

Willekeurige trefwoordargumenten gebruiken

- In het vorige voorbeeld accepteerde de functie een willekeurig aantal series als argument
- in bepaalde situaties wil je een functie evenwel in staat stellen een willekeurig aantal verschillende argumenten te accepteren
- In dat geval kan je je functie een ongelimiteerd aantal key-value-paren laten aannemen
- De dubbele asterisk voor het argument draagt Python op een lege dictionary aan te maken en er de eventuele binnenkomende key-value paren in onder te brengen

Willekeurige trefwoordargumenten gebruiken

```
def verzamel_gebruikersinformatie(voornaam, familienaam, **extra_informatie):
    """Een dictionary bouwen met alle informatie die we over de gebruiker kennen"""
    extra_informatie["voornaam"] = voornaam
    extra_informatie["naam"] = familienaam
    return extra_informatie

kristof_info = verzamel_gebruikersinformatie("Kristof", "Michiels",
    favoriete_taal="Python", favoriete_muziek="Little Simz",
    favoriete_serie="Station Eleven")

print(kristof_info)
```

Python Development

Werken met bestanden en foutafhandeling

Kristof Michiels

Inhoud

- Werken met bestanden
- Command-line argumenten
- Foutafhandeling in Python

Werken met bestanden

Werken met bestanden

- De input in onze programma's bleef tot nu toe beperkt tot gebruikersinput via het toetsenbord. Alle output werd dan weer getoond op het terminalscherm
- Bestanden kunnen eveneens gebruikt worden als input en output voor onze programma's. Deze kunnen dan snel grote hoeveelheden data analyseren. Bovendien kunnen ze data ook permanent vasthouden
- We zullen hier werken met tekstbestanden:
  - Ze bestaan uit bitreeksen die ASCII of UTF-8 karakters vertegenwoordigen
  - Tekstbestanden kunnen bewerkt worden in elke teksteditor
  - Dezelfde principes gelden grotendeels voor binaire bestanden

Een bestand openen

- Vooraleer we een bestand kunnen gebruiken moeten we het eerst openen met de open-functie
- De open-functie neemt twee string-argumenten: de naam van het bestand en de toegangsmodus ("r" voor read, "w" voor write en "a" voor append)
- De functie geeft een file-object terug waarop we verschillende methods kunnen toepassen om van het bestand te lezen of om er data naar weg te schrijven
- Nadat de lees- en/of schrijfbewerkingen afgerond zijn dient het bestand gesloten te worden. We doen dit expliciet met de close-method of kunnen dit ook overlaten aan Python

```
mijn_bestand = open("bestand.txt", "r")
#...
mijn_bestand.close()
```

Een bestand openen

- Gebruik je het with-blok en zorg je ervoor dat alle bestandsbewerkingen zich binnen dit blok bevinden, dan hoef je het bestand niet zelf af te sluiten. Python doet dit voor jou.

```
with open("bestand.txt", "r") as mijn_bestand:
    inhoud = mijn_bestand.read()
print(inhoud)
```

Regel per regel uit een bestand lezen

- Input lezen kan op verschillende manieren, maar enkel als het bestand geopend is in read-modus (met "r")
- De readline-method leest één regel en geeft het terug als string
- Elk volgend aanroepen van deze method geeft de volgende regel terug, van boven naar onder
- Wanneer het einde van het bestand is bereikt wordt een lege string teruggegeven

Regel per regel uit een bestand lezen

```
with open("bestand.txt","r") as mijn_bestand:
    for regel in mijn_bestand:
        print(regel)
```

```
mijn_bestand = open("bestand.txt", "r")
aantal_regels = 0
regel = mijn_bestand.readline()
while regel != "":
    aantal_regels += 1
    regel = mijn_bestand.readline()
mijn_bestand.close()
print(f"Het aantal regels in dit bestand is {aantal_regels}")
```

Volledig bestand in één keer lezen

- Soms kan het aangewezen zijn om alle data in een bestand in één keer uit te lezen
- Je kan dit doen door gebruik van de read-method of met de readlines-method
- De read-method geeft de volledige inhoud van het bestand terug als string. Via verdere verwerking kan de string dan in kleinere stukken worden opgebroken
- De readlines-method geeft een list terug waarbij elk element een regel is uit het bestand. Een loop kan gebruikt worden om door alle regels te itereren

Volledig bestand in één keer lezen

```
with open("bestand.txt","r") as mijn_bestand:
    regels = mijn_bestand.readlines()

for regel in regels:
    print(regel.rstrip())
```

```
mijn_bestand = open("bestand.txt", "r")
aantal_regels = 0
regels = mijn_bestand.readlines()
for regel in regels:
    aantal_regels += 1
mijn_bestand.close()
print(f"Het aantal regels in dit bestand is {aantal_regels}")
```

Einde-regel tekens

- Bij het uitlezen van een regel of een volledig bestand wordt telkens geëindigd met een lege string
- Je kan rstrip() gebruiken om dit weg te werken

```
with open("bestand.txt", "r") as mijn_bestand:
    inhoud = mijn_bestand.read()
print(inhoud.rstrip())
```

```
with open("bestand.txt", "r") as mijn_bestand:
    regels = mijn_bestand.readlines()
    for regel in regels:
        print(regel.rstrip())
```

Bestandspaden

Indien een bestand dat je wil openen zich in de programma-folder bevindt dan dien je enkel de bestandsnaam mee te geven. Bevindt het bestand zich op een andere plaats dan moet je het file path meegeven

- Dit kan een relatief pad zijn:

```
with open("mijn_bestanden/bestand.txt","r") as mijn_bestand
```

- Maar met een absoluut pad kan je verwijzen naar elke locatie op je computer:

```
absoluut_bestandspad = "/Users/kristofmichiels/bestand.txt"
with open(absoluut_bestandspad,"r") as mijn_bestand
```

Schrijven naar een bestand

- Schrijven kan naar een geopend bestand in ofwel write ("w") of append ("a")-mode
- Bestaat het geopende bestand nog niet dan wordt een nieuw, leeg bestand aangemaakt
- In write-modus wordt een geopend bestaand bestand volledig leeggemaakt
- In append-modus wordt een geopend bestaand bestand intact gelaten en wordt nieuwe data weggeschreven onderaan het bestand
- De write-method neemt één string-argument, namelijk de data die zal worden weggeschreven
- Andere datatypes kunnen eerst geconverteerd worden met de str-functie

Schrijven naar een bestand

- Je kan meerdere regels wegschrijven:
  - door ze ofwel bij elkaar te voegen tot één lange string
  - meerdere regels worden dan gescheiden door "\n"
  - je kan ook de write-method meerdere keren aanroepen

```
with open("bestand.txt", "w") as mijn_bestand:
    mijn_bestand.write("Ik hou van programmeren.")
    mijn_bestand.write("Python rocks!")
```

Toevoegen aan een bestand

```
with open("bestand.txt", "a") as mijn_bestand:
    mijn_bestand.write("Ik hou van programmeren.")
    mijn_bestand.write("Python rocks!")
```

Data opslaan

- De technieken die we daarnet hebben gezien kunnen gebruikt worden om programma-data vast te houden
- Een eenvoudige manier om dit te doen is door gebruik te maken van de json-module
- Deze module laat toe om eenvoudige Python data-structuren vast te houden in een bestand, en bij een volgend gebruik van je programma terug vast te nemen
- We maken hierbij gebruik van het JSON-dataformaat

json.dump() en json.load()

- We gebruiken json.dump() om een list als json op te slaan in een bestand
- De json.dump() functie neemt 2 argumenten: data om op te slaan en een bestands-object om de data aan toe te voegen

```
import json

getallen = [5, 2, 1, 7, 13, 23]

with open("bestand.json","w") as f:
    json.dump(getallen,f)
```

json.dump() en json.load()

- In onderstaand codevoorbeeld gebruiken we json.load() om de list vanuit het bestand terug in de toepassing te krijgen
- De json.load() functie neemt 1 argument: een bestands-object om de data uit in te lezen

```
import json

with open("bestand.json","r") as f:
    getallen = json.load(f)
```

Lezen en opslaan van applicatie-gegenereerde data

Dit is een oefening die jullie zelf maken!

Command-line argumenten

Command-line argumenten

- De Python-programma's die we schrijven worden uitgevoerd via de command-line (python mijn_bestand.py)
- Onze programma's kunnen input meekrijgen via command-line argumenten
- Deze argumenten worden bijgehouden in een variabele argv (argument vector) die deel uitmaakt van de sys (system) module
- argv is een list, waarvan de elementen met type-conversie functies kunnen worden omgezet naar het gewenste datatype

Command-line argumenten

```
import sys

print(f"Dit programma heeft {len(sys.argv)} command-line argumenten.")
print(f"De naam van het bestand is {sys.argv[0]}")

if len(sys.argv) > 1:
    print("De overige argumenten zijn:")
    for i in range(1, len(sys.argv)):
        print(sys.argv[i])
```

Command-line argumenten

- De sys-module is nodig en wordt geïmporteerd met het import statement
- Het eerste element van de sys.argv-list is de naam van het uitgevoerde bestand
- De volgende elementen in de list zijn de argumenten die na de bestandsnaam zijn meegegeven

Foutafhandeling in Python

Foutafhandeling in Python

- Tijdens het runnen van een programma kunnen zaken fout lopen: niet in het minst langs de input-kant
  - Bvb de gebruiker kan een niet-numerieke waarde ingeven waar er één wordt verwacht
  - Bvb de gebruiker kan verkeerdelijk een niet-bestaande bestandsnaam meegeven
  - Telkens een fout optreedt crasht ons programma
- Python gebruikt speciale objecten om fouten te beheren. Deze objecten heten exceptions
- Telkens een fout gebeurt en Python niet verder kan wordt een exception object aangemaakt

Try en except

- Als je code schrijft om de exception op te vangen, dan kan het programma verwijderen. Je doet dit met een try-except-blok
- Wanneer een exception zich voordoet binnen een try-blok stopt de verdere uitvoering binnen deze blok en springt de uitvoering onmiddellijk naar de aangewezen except-blok
- Elke except-blok kan aangeven welk soort exception het wil opvangen door na het except trefwoord onmiddellijk het type exception mee te geven
- Een except-blok zonder specifieke exception zal elk type exception opvangen (die nog niet door een specifieke exception werd behandeld)

Try en except

```
antwoord = input("Geef een bestandsnaam: ")
try:
    mijn_bestand = open(antwoord, "r")
except FileNotFoundError:
    print("Het bestand kon niet worden gevonden. Programma wordt afgesloten...")
    quit()
```

- De except-blok(ken) worden enkel in het geval van een fout uitgevoerd
- In het bovenstaande voorbeeld zorgt een niet-bestaand bestand voor een FileNotFoundError exception

Try en except

```
antwoord = input("Geef een bestandsnaam: ")
bestand_geopend = False
while bestand_geopend == False:
    try:
        mijn_bestand = open(antwoord, "r")
        bestand_geopend = True
    except FileNotFoundError:
        print("Het bestand kon niet worden gevonden")
        antwoord = input("Geef een bestandsnaam: ")
```

- Hier passen we bovenstaande code aan zodat het programma blijft proberen tot een bestaand bestand werd gevonden

Try en except: het else-blok

```
print("Geef 2 getallen. We bepalen de rest bij deling door. Druk 's' om te stoppen:")

while True:
    eerste_getal = input("\nEerste getal: ")
    if eerste_getal == "s":
        break
    tweede_getal = input("\nTweede getal: ")
    if tweede_getal == "s":
        break
    try:
        antwoord = int(eerste_getal) % int(tweede_getal)
    except ZeroDivisionError:
        print("Het is onmogelijk om te delen door nul.")
    else:
        print(antwoord)
```

Try en except: het else-blok

- Bovenstaand voorbeeld bevat ook een else-blok. Als het try-blok succesvol is verlopen zal vervolgens het else-blok worden uitgevoerd

Try en except: stilletjes falen

- Soms wil je bij een bepaalde exception geen crash, maar is er ook geen behoefte aan code die de exception opvangt
- Dan kan je door toevoegen van een pass statement verder gaan met het programma, alsof er niets gebeurd is

```
try:
    # code
except FileNotFoundError:
    pass
else:
    # code
```

Foutafhandeling in Python

- De concepten die in dit hoofdstuk werden geïntroduceerd kunnen gebruikt worden om een groot aantal verschillende fouten op te vangen in je code
- Pythons foutafhandeling geeft je veel controle over hoe met gebeurlijke fouten om te gaan
- Ze is vooral tot nut als je programma met iets externs moet omgaan (input, output)
- Door het gebruiken van try en except blokken kunnen je programma's alle fouten opvangen die anders tot een crash zouden leiden

Python Development

Werken met strings

Kristof Michiels

Werken met strings

- We hebben in de eerste les strings reeds kort behandeld en gaan ons hier concentreren op zaken die toen niet aan bod zijn gekomen
- Strings - letterlijk tekstreeksen - bestaan uit een opeenvolging van karakters
- We zagen eerder dat strings in Python onveranderlijk of immutable zijn. Dit betekent dat we ze niet kunnen wijzigen, maar wel (gedeeltes) kunnen kopiëren naar een andere string om hetzelfde effect te bekomen

Strings maken met aanhalingstekens

- We zagen reeds dat we strings wikkelen in enkele of dubbele aanhalingstekens
- We mogen ook gebruik maken van driedubbele aanhalingstekens
- Wordt vaak gebruikt voor strings die meerdere regels bestrijken
- Bij dergelijke strings worden zowel de einde-regelkarakters (\n) als andere spaties behouden en in acht genomen

```
lorem_ipsum = '''Lorem Ipsum is slechts een proeftekst uit het
    drukkerij- en zetterijwezen. Lorem Ipsum is de standaard proeftekst
    in deze bedrijfstak sinds de 16e eeuw, toen een onbekende drukker een
    zethaak met letters nam en ze door elkaar husselde om een
    font-catalogus te maken.'''
```

Strings maken met str()

- Je kan vanuit een ander datatype een string maken door gebruik te maken van de str()-functie
- Python gebruikt deze functie intern wanneer je de print()-functie uitvoert op niet-string objecten of wanneer je strings formatteert

```
mijn_list = [str(98.6), str(1.0e4), str(True)]
for element in mijn_list:
    print(type(element))
```

Escapen met \

- Plaats je een backslash (\) voor sommige karakters, dan krijgen deze een speciale betekenis
- De meest bekende (die we eerder tegenkwamen) is \n, die ervoor zorgt dat wat volgt op een nieuwe regel begint
- We zagen eerder ook al \t, wat staat voor een tab-insprong

```
palindroom_zin = "Baas, \nneem een racecar, \nneem een Saab."
```

Escapen met \

- Je kan ook \' or \" gebruiken om mee te geven dat je letterlijk een aanhalingsteken wil aangeven
- Of zelfs letterlijk een backslash: dan gebruik je er twee (\) waarbij de eerste de tweede escaped

```
krantenkop = "Koning sprak ons dit jaar vijf keer toe: \"Het geeft me houvast\""
```

r-strings of raw-strings

Een raw-string geeft de escapes letterlijk weer

```
krantenkop = r"Koning sprak ons dit jaar vijf keer toe: \"Het geeft me houvast\""
palindroom_zin = r"Baas, \nneem een racecar, \nneem een Saab."
```

Combineren met +

- Niet te verwarren met de f-strings die we eerder hebben gezien
- Je kan strings combineren met string variabelen door gebruik van de + operator
- Je kan letterlijke strings (geen string variabelen) ook combineren door ze gewoon achter elkaar te plaatsen
- Overspant de combinatie meerdere regels dan kan je ze wikkelen in haakjes

```
print("aap" + "noot" + "mies")
print("aap" "noot" "mies")
woordjes = ( "aap" "noot"
"mies")
```

Dupliceren met *

Je kan de *-operator gebruiken om een string te dupliceren

```
duplicaat = "noot" * 4 + '\n'
print(duplicaat)
```

Een karakter opvragen met [ ]

- Om een enkel karakter uit een string op te vragen kan je werken met het index-getal
- Een string is een tekstreeks en kan op dat vlak behandeld worden zoals een list: je kan erdoor loopen en je kan elementen opvragen (niet vergeten dat een string wel immutable is)
- Eerste karakter: indexgetal is 0
- Laatste karakter: als indexgetal kan je -1 gebruiken. Voor het voorlaatste -2 enz...

```
letters = 'abcdefghijklmnopqrstuvwxyz'
print(letters[0])
```

Een gedeelte van een string bekomen met slice

- [:] splitst de volledige string af van begin tot einde
- [start :] splitst af van karakter "start" tot het einde van de string
- [: einde] splitst af van begin van de string tot aan karakter "einde" - 1
- [start : einde] splitst af van karakter "start" tot aan karakter "einde" - 1
- [start : einde : step] splitst af van karakter "start" tot aan karakter "einde" - 1, met sprongetjes met als grootte "step"

Een gedeelte van een string bekomen met slice

Probeer zelf uit!

```
alfabet = "abcdefghijklmnopqrstuvwxyz"
print(alfabet[:])
print(alfabet[10:])
print(alfabet[10:15])
print(alfabet[-5:])
print(alfabet[12:-2])
print(alfabet[-10:-5])
print(alfabet[::5])
print(alfabet[2:10:3])
print(alfabet[10::4])
print(alfabet[:20:5])
```

Een gedeelte van een string bekomen met slice

Probeer zelf uit!

```
alfabet = "abcdefghijklmnopqrstuvwxyz"
print(alfabet[-1::-1])
print(alfabet[::-1])
print(alfabet[-40:])
print(alfabet[-60:-50])
print(alfabet[:100])
print(alfabet[80:85])
```

Lengte van een string verkrijgen met len()

De len() functie geeft je het aantal karakters in een string

```
alfabet = "abcdefghijklmnopqrstuvwxyz"
print(len(alfabet))
```

Splitsen met split()

- Je kan de split()-functie gebruiken om een string op te breken in een aantal kleinere onderdelen
- De splitsing gebeurt op basis van een scheidingsteken, ook "separator" genoemd
- Dat scheidingsteken geef je mee aan de functie. Bij afwezigheid gebruikt split() elke reeks spaties, tabs, einde-regel (\n) om te splitsen
- De gesplitste strings komen terug als elementen in een list

```
woorden = "aap, noot, mies, muis, poes"
print(woorden.split(","))
# ['aap', ' noot', ' mies', ' muis', ' poes']
print(woorden.split())
# ['aap,', 'noot,', 'mies,', 'muis,', 'poes']
```

Samenvoegen met join()

- De join()-functie biedt het tegenovergestelde van split(): het zet een list met strings om naar één enkele string
- Eerst geef je de string die alle elementen gaat verbinden
- Als argument van de join()-functie geef je de list mee die je wil omzetten

```
lijst_van_superhelden = ["Ant-Man", "Batwoman", "The Atom", "The Avengers"]
string_van_superhelden = ", ".join(lijst_van_superhelden)
print(string_van_superhelden)
nieuwe_regel_string_van_superhelden = "\n".join(lijst_van_superhelden)
print(nieuwe_regel_string_van_superhelden)
# zal elk element op nieuwe regel laten beginnen
```

Vervangen met replace()

- Je gebruikt de replace()-functie om elementen in je string te vervangen
- Eerste argument is de oude string die je wil veranderen
- Tweede argument is de string waarmee je de oude string wil veranderen
- Optioneel derde argument is het aantal keer dat je een string wil veranderen door de nieuwe. Laat je het weg dan wordt elke keer de oude string voorkomt deze vervangen door de nieuwe

```
mijn_string = "een zin als een ander"
mijn_string.replace("een", "geen", 1)
print(mijn_string) #Eerste "een" wordt vervangen
mijn_string.replace("een", "geen") # alle "een"-en worden vervangen door "geen"
```

Vervangen met replace()

- In sommige gevallen wil je complexere string-patronen herkennen en vervangen
- Bvb dat het patroon een vrijstaand woord is, of dat het zich bevindt aan het begin of einde van een string....
- We hebben hiervoor een uiterst krachtig instrument, namelijk regular expressions. We komen hierop later nog terug!

Strippen met strip()

- Werkt - indien aanwezig - witruimte-karakters (" ", "\t", "\n") weg links, rechts of langs beide kanten van een string
- strip() werkt links en rechts deze karakters weg
- lstrip() werkt links deze karakters weg
- rstrip() werkt rechts deze karakters weg
- Je kan aan strip ook andere karakters meegeven die mogen verwijderd worden

```
mijn_string = "    Teveel ruimte ? . !    "
print(mijn_string.strip())
print(mijn_string.strip("!?. "))
```

Zoeken en selecteren met find() en index()

- Python heeft twee methods, find() en index() om een string in een string te localiseren
- Van elks zijn er twee versies: starten van het begin, of starten van het einde van de string
- Beiden werken op dezelfde manier als de substring effectief gevonden wordt
- Als er geen gevonden wordt geeft find() -1 terug, en index() creëert een exception

```
woorden = "aap, noot, mies, muis, poes, noot"
woorden.find("noot") # 5
woorden.index("noot") # 5
woorden.rfind("noot") # 29
woorden.rindex("noot") # 29
```

count() en isalnum()

- Met count() tel je het aantal keer dat een substring voorkomt in een andere string
- Met isalnum() ga je na of de string bestaat uit enkel letters of cijfers

```
woorden = "aap, noot, mies, muis, poes, noot"
woorden.count("noot") # 2
woorden.count("oranje") # 0
woorden.count("mies") # 1
woorden.isalnum() # False, er zitten komma's in
```

startswith() en endswith()

Gaan na of een string met een bepaalde string starten of eindigen, en geven True of False terug

```
mijn_zin = '''Git comes with built-in GUI tools (git-gui, gitk),
but there are several third-party tools for users looking for a platform-specific
experience.'''
mijn_zin.startswith("Git") # True
mijn_zin.endswith("experience.") # True
mijn_zin.endswith("experience") # False
```

Hoofd- en kleine letters

- capitalize() zal van het eerste letter van de string een hoofdletter maken
- title() zal alle woorden een hoofletter geven
- upper() zal alle karakters in hoofdletters weergeven
- lower() zal alle karakters in kleine letters weergeven
- swapcase() wisselt kleine in hoofdletters en omgekeerd

```
tura_zin = "mooi het leven is mooi :-)"
print(tura_zin.capitalize()) # Mooi het leven is mooi :-)
print(tura_zin.title()) # Mooi Het Leven Is Mooi :-)
print(tura_zin.upper()) # MOOI HET LEVEN IS MOOI :-)
print(tura_zin.lower()) # mooi het leven is mooi :-)
print("Hallo".swapcase()) # 'hALLO'
```

Uitlijning

- center(): centreert de string in een aantal karakters
- ljust(): lijnt links uit in een aantal karakters
- rjust(): lijnt rechts uit in een aantal karakters

```
tura_zin = "mooi het leven is mooi :-)"
print(tura_zin.center(30)) # "  mooi het leven is mooi :-)  "
print(tura_zin.ljust(30)) # "mooi het leven is mooi :-)    "
print(tura_zin.rjust(30)) # "    mooi het leven is mooi :-)"
```

Python Development

Werken met booleans en getallen, Math en Random

Kristof Michiels

Inhoud

- Werken met booleans en getallen
- Wiskundige functies met de math library
- Gebruik van de random-module

Werken met booleans en getallen

Werken met booleans en getallen

We gooien een tweede blik op Pythons eenvoudigste ingebouwde datatypes:

- Booleans (met waarden True of False)
- Integers (gehele getallen zoals 12, -4 en 2000000)
- Floats (getallen met een zwevende komma zoals 3.14159, of ook exponenten als 1.0e7, dat staat voor 1 x 10 tot de zevende macht, of 10000000.0)

Werken met booleans

- We zagen eerder dat de enige waarden voor het booleaanse datatype True en False zijn. We hebben het belang gezien voor gebruik van vergelijkingsoperatoren
- In sommige gevallen zal je deze rechtstreeks gebruiken: zie eerder in de cursus
- In andere gevallen zal je de "truthiness" van de waarden van andere datatypes evalueren:
  - Truthy waarden zijn waarden die in een booleaanse context in True evalueren
  - Falsy waarden zijn waarden die in een booleaanse context in False evalueren
- De speciale Python-functie bool() kan elk ander data type converteren naar een boolean: bool() kan elke mogelijke waarde als argument nemen en geeft het booleaanse equivalent terug
- Daarbij geldt algemeen dat alle niet-nul-waarden als True worden beschouwd en nul-waarden als False

Werken met booleans

Voorbeelden van gebruik van de bool-functie:

```
print(bool(True)) # True ! Let op: omdat True niet nul of leeg is
print(bool(1)) # True
print(bool(0)) # False
print(bool(45)) # True
print(bool(-45)) # True
print(bool(False)) # False
print(bool(0)) # False
print(bool(0.0)) # False
print(bool(1.0)) # True
print(bool({})) # False
print(bool([])) # False
print(bool("")) # False
```

Werken met booleans

- False wordt behandeld als 0 of 0.0 wanneer je ze mixt met integers of floats
- True wordt in dat geval behandeld als 1 of 1.0

```
print(True + 2) # 3
print(False + 5.0) # 5.0
```

Werken met integers

- Integers of gehele getallen kunnen in Python opgeteld (+), afgetrokken (-), vermenigvuldigd (*) en gedeeld (/) worden. Dat zagen we eerder al.
- De iets minder gekende //-operator geeft het gehele gedeelte van het quotiënt terug
- Python gebruikt twee vermenigvuldigingssymbolen (**) om exponenten weer te geven
- Elke expressie kan meerdere bewerkingen bevatten. De wiskundige volgorde van bewerkingen wordt gerespecteerd
- Je mag gebruik maken van haakjes om deze volgorde aan te passen

Werken met integers

```
getal1 = -2
getal2 = 3
print(getal1 + getal2) # 1
print(getal1 ** getal2) # -8
print((getal1 + getal2) * getal1) # -2
print(14 // 3) # 4 is het gehele gedeelte van het quotiënt
```

Combinatie wiskundige- met toekenningsoperatoren

Je mag wiskundige operatoren combineren met toekenningsoperatoren:

```
a -= 3 # a = a -3
a += 8 # a = a + 8
a *= 2 # a = a * 2
a /= 3 # a = a / 3
```

De modulus operator

De modulus operator (%) produceert de rest bij deling

```
print(9 % 5) # 4
print(8 % 2) # 0
```

Met de divmod-functie krijg je zowel het (afgekapte) quotiënt als de rest

```
print(divmod(18,5)) (3, 3)
```

Werken met andere talstelsels

- Gehele getallen zijn in Python geacht decimaal te zijn (base 10), tenzij je een prefix gebruikt die verwijst naar een ander talstelsel of basis (base)
- Een base is hoeveel cijfers je kan gebruiken tot je "de één overdraagt". In het decimaal stelsel gebeurt dat bij 10, vandaar de term tiendelig talstelsel
- In Python kan je integers naast het decimale ook in drie andere talstelsels uitdrukken. Je doet dit door gebruik te maken van volgende prefixen:
  - 0b of 0B voor het binaire talstelsel (base 2)
  - 0o of 0O voor het octaal talstelsel (base 8)
  - 0x of 0X voor het hexadecimaal talstelsel (base 16)

Werken met andere talstelsels

- Enkele voorbeelden:
  - Een binaire (base 2) 0b10, wat staat voor 2 in het decimale talstelsel
  - Een octale (base 8) 0o10, wat staat voor 8 in het decimale talstelsel
  - Een hexadecimale (base 16) 0x10, wat staat voor 16 in het decimale talstelsel

```
print(0b10) # 2
print(0o10) # 8
print(0x10) # 16
```

Werken met andere talstelsels

Je kan in Python ook in de andere richting gaan en met de functies bin, oct en hex integers omzetten naar de respectievelijke andere talstelsels

```
print(bin(63)) # 0b111111
print(oct(32)) # 0o40
print(hex(342)) # 0x156
```

Datatypes omzetten naar integers met de int-functie

```
print(int(True)) # 1
print(int(False)) # 0
print(int(98.6)) # 98
print(int(1.0e4)) # 10000
print(int('99')) # 99
print(int('-23')) # -23
print(int('+12')) # 12
print(int('1_000_000')) # 1000000
print(int('10', 2)) # binaire 2
print(int('10', 8)) # octale 8
print(int('10', 16)) # hexadecimale 16
```

Floats

- Worden zoals we eerder zagen op dezelfde manier behandeld als integers
- Je kan gebruik maken van de operatoren +, –, *, /, //, **, en %) én van de divmod() functie
- Om om te zetten naar float gebruiken we de float-functie

```
print(float(True)) # 1.0
print(float(False)) # 0.0
print(float(76)) # 76.0
print(float("99")) # 99.0
print(43 + 2.) # 45.0
print(False + 0.) # 0.0
print(True + 0.) # 1.0
```

Nauwkeuriger na de komma rekenen met decimal

- Floats in programmeren gedragen zich niet helemaal zoals reële getallen uit de wiskunde.

```
print(10.0 / 3.0) # 3.3333333333333335
```

- We zien hier 5 op het 16de getal na de komma, terwijl het eigenlijk 3 zou moeten zijn. Het getal wordt dus niet 100% volledig accuraat gerepresenteerd.

Nauwkeuriger na de komma rekenen met decimal

- Met de decimal module laat Python ons toe om getallen exacter weer te geven. Dit is handig wanneer meer precisie nodig is, zoals in het bankwezen.

```
from decimal import Decimal
prijs = Decimal("19.99")
btw = Decimal("0.21")
totaal = prijs + (prijs * btw)
print(totaal) # 24.1879
```

Rationele breuken met de fractions module

- We kunnen breuken weergeven als fractions door gebruik te maken van de fractions module

```
from fractions import Fraction
print(Fraction(1, 3) * Fraction(2, 3)) # 2/9
print(Fraction(Decimal('1.0')/Decimal('3.0')))
#33333333333333333333333333/100000000000000000000000000000
```

- De fractions module biedt ook zaken als de grootst gemene deler:

```
import fractions
print(fractions.gcd(24, 16)) # 8
```

Complexe getallen

- Gebruik van complexe getallen wordt volledig ondersteund door Python, met hun notatie van een reëel en imaginair gedeelte
- We maken hiervoor over het algemeen gebruik van de cmath module

```
import cmath
getal = complex(5,3)
print(getal.real) # 5.0
print(getal.imag) # 3.0
print (cmath.phase(getal)) # de fase: 0.5404195002705842
print(cmath.polar(getal))
# polair / modulus en argument: (5.830951894845301, 0.5404195002705842)
print(cmath.rect(5.830951894845301, 0.5404195002705842))
# rechthoekig: (5+3.0000000000000004j)
```

Wiskundige functies met de math library

Wiskundige functies met de math library

- Python laat ons gebruik maken van een groot aantal wiskundige functies via de standard math library
- Je activeert de library via het import statement
- Ze bevat constantes zoals pi en e

```
import math
print(math.pi) # 3.141592653589793
print(math.e) # 2.718281828459045
```

fabs()

De fabs functie geeft de absolute waarde terug van zijn argument

```
import math
print(math.fabs(88.7)) # 88.7
print(math.fabs(-153.1)) # 153.1
```

floor() en ceil()

- Met de floor-functie krijg je het geheel getal onder het meegegeven getal terug
- Met de ceil-functie krijg je het geheel getal boven het meegegeven getal terug

```
import math
print(math.floor(78.7)) # 78
print(math.floor(-154.2)) # -155
print(math.ceil(68.4)) # 69
print(math.ceil(-154.7)) # -154
```

Faculteit van n (n!)

In de wiskunde is de uitdrukking 3! wordt gelezen als "drie faculteit" en is in feite een verkorte manier om de vermenigvuldiging van meerdere opeenvolgende gehele getallen aan te duiden.

```
import math
print(math.factorial(3)) # 6
print(math.factorial(10)) # 3628800
print(math.factorial(5)) # 120
```

pow()

- De pow-functie gaat een getal tot een bepaalde macht verheffen:

```
import math
print(math.pow(2, 3)) # 8.0
```

sqrt()

- Met de sqrt-functie bereken je de vierkantswortel uit een getal:

```
import math
print(math.sqrt(100.0)) # 10.0
```

Goniometrische functies

- Alle courante goniometrische functies zijn aanwezig: sin(), cos(), tan(), asin(), acos(), atan(), en atan2()- Er is ook een hypot()-functie om de hypotenusa of schuine zijde van een rechthoekige rechthoek te berekenen (stelling van Pythagoras). We hebben deze eerder berekend in een oefening

```
import Math
print(math.sin(90)) # 0.8939966636005579
print(math.hypot(3, 4)) # 5.0
```

Gebruik van de random-module

random.choice()

- De random-module biedt je aantal functies voor het genereren van pseudo-randomgetallen, zo ook choice()
- https://docs.python.org/3/library/random.html
- Bijna alle functies die binnen deze module voorzien zijn zijn afgeleid van de random-functie, die een willekeurige float genereert binnen de range [0.0, 1.0)
- De choice-functie geeft een willekeurige waarde terug uit een reeks (list, tuple, dictionary, string)

```
from random import choice
print(choice([4, 3, 6, 10])) #10
print(choice(("een", "twee", "drie", "vier"))) # "een"
print(choice(range(100))) # 83
print(choice("hemellichaam")) # t
```

random.sample()

Gebruik de sample-functie om meer dan één willekeurige waarde terug te krijgen

```
from random import sample
print(sample([1, 2, 3, 3, 5, 6, 7, 8], 3)) # [1, 8, 2]
print(sample(("een", "twee", "drie", "vier", "vijf", "zes"), 2)) # ['drie', 'twee']
print(sample(range(100), 5)) # [80, 36, 19, 88, 24]
print(sample("begunstigde", 5)) # ['d', 't', 'e', 'u', 'b']
```

random.randint() en random.randrange()

- Om een willekeurig integer-getal terug te krijgen uit een range kan je choice() of sample() gebruiken met range(), maar ook randint() of randrange()
- randrange() heeft zoals range() argumenten voor start (inclusief) en eind (exclusief) integers, maar ook een derde optioneel argument waarmee je sprongetjes doorheen de range maakt

```
from random import randint
print(randint(40, 88)) # 64
print(randint(1, 10)) # 6

from random import randrange
print(randrange(40, 88)) # 61
print(randrange(30, 70, 10)) # 50
print(randrange(28, 70, 10)) # 38
```

random()

De random-functie genereert zoals gezegd een willekeurige float binnen de range [0.0, 1.0)

```
from random import random
print(random()) # 0.9038328860897611
```

Python Development

Python toepassingen en virtuele Python-omgevingen

Kristof Michiels

Inhoud

- Python toepassingen
- Virtuele Python-omgevingen
- De Python interactieve shell

Python toepassingen

Een eenvoudige toepassing maken

- Een Python programma of script = elke hoeveelheid van opeenvolgende Python statements
- Als het aantal statements toeneemt is het zinvol om bijkomende structuur toe te voegen
- Een goeie en eenvoudige eerste stap: een controlerende functie (bvb main()) toevoegen in een bestand en deze functie aanroepen

```
def main():
    print("Kijk eens aan, ons eerste programma ;-)")

main()
```

Verhinderen dat bij importeren code wordt uitgevoerd

- Met een if-statement kan je checken of het bestand al dan niet geïmporteerd werd uitgevoerd
- Wanneer de interpreter een bronbestand inleest zet het een aantal speciale variabelen, zoals __name__
- Als je de module als programma runt (en dus niet importeert) dan kent de interpreter de string "main" toe aan deze variabele
- Goeie manier van werken: steeds onderstaande opzet gebruiken

```
def main():
    print("Kijk eens aan, ons eerste programma ;-)")

if __name__ == "__main__":
    main()
```

Een script starten van de command-line

- We hebben tot nog toe onze programma's uitgevoerd met de run-knop van VSCode, of via de terminal sessie die VSCode vervolgens voor ons aanmaakt
- We kunnen een Python ook starten via een externe terminal sessie: *python mijn_script.py*
- Bij Linux/Unix besturingssystemen spreken we een bash of zsh-shell
- Windows gebruikers kunnen ook een bash command-line verkrijgen via Git Bash. Installeer git via https://git-scm.com/ en Git Bash wordt automatisch mee-geïnstalleerd
- Je kan na installatie ook binnen VSCode kiezen voor een Bash terminal sessie

Command-line argumenten

- Onze programma's kunnen input meekrijgen via command-line argumenten (zie ook les 6): *python mijn_script.py arg1 arg2 argn*
- Deze argumenten worden bijgehouden in een variabele argv (argument vector) die deel uitmaakt van de sys (system) module
- argv is een list, waarvan de elementen met type-conversie functies kunnen worden omgezet naar het gewenste datatype
- De sys-module is nodig en wordt geïmporteerd met het import statement
- Het eerste element van de sys.argv-list is de naam van het uitgevoerde bestand
- De volgende elementen in de list zijn de argumenten die na de bestandsnaam zijn meegegeven

Command-line argumenten

```
import sys

print(f"Dit programma heeft {len(sys.argv)} command-line argumenten.")
print(f"De naam van het bestand is {sys.argv[0]}")

if len(sys.argv) > 1:
    print("De overige argumenten zijn:")
    for i in range(1, len(sys.argv)):
        print(sys.argv[i])
```

De in- en output van een programma kanaliseren

- We doen dit door gebruik te maken van command-line opties en van sys.stdin.read() en sys.stdout.write()
- Je krijgt er toegang mee naar en tot via:*python mijn_script.py zero 0 < input.txt > output.txt*

```
import sys
def main():
    output_string = "Hallo wereld"
    mijn_input = sys.stdin.read()
    sys.stdout.write(output_string)

main()
```

De in- en output van een programma kanaliseren

- Toevoegen aan output-bestand ipv overschrijven:*python mijn_script.py zero 0 >> output.txt*
- De output van een script doorgeven als input van een ander script (met het pipe-symbool): python script1.py | python script2.py

De argparse module: flags én argumenten

- Je kan een script zo configureren dat het zowel command-line opties als argumenten kan accepteren
- De argparse module geeft ondersteuning voor verschillende soorten van argumenten en kan ook hulpboodschappen meegeven aan gebruikers
- Om de argparse module te gebruiken kan je een instance van ArgumentParser creëren, het voorzien van argumenten, en vervolgens zowel de opionele als positionele argumenten uitlezen

De argparse module

- Een gedetailleerd overzicht van werken met de argparse valt buiten de scope van deze cursus
- Voor studenten die meer willen weten raad ik de volgende resource van harte aan: https://realpython.com/command-line-interfaces-python-argparse/

Een script onmiddellijk uitvoerbaar maken

```
#! /usr/bin/env python
```

- Op Unix/Linux/macOS kan je een script heel eenvoudig onmiddellijk uitvoerbaar maken
- Voeg de bovenstaande lijn code toe aan de toepassing...
- En verander de mode met *chmod +x mijn_toepassing.py*
- Wil je toegang tot het besturingssysteem of het administratieve gedeelte (bvb gebruikersbeheer) dan zijn er heel wat Python bibliotheken beschikbaar om deze taken te vervullen

Code organiseren in modules

- In de les over functies hebben we reeds gezien hoe we functies uit het ene bestand kunnen importeren om ze te gebruiken in een ander bestand
- We beschrijven een module als elk bestand dat Python objecten beschrijft, dus ook de bestanden die we importeerden zijn modules
- Als een bestand mijn_programma.py heeft als naam dan is mijn_programma de modulenaam
- Je gebruikt het import statement om een externe module te linken. Objecten uit externe module kunnen dan gebruikt worden als modulenaam.objectnaam
- Nogmaals: je kan bekijken hoe dit precies werkt in de les over functies

Modules als organisatieblokken voor grotere projecten

- Modules worden gebruikt om grotere projecten beheersbaar te houden
- Ook Python zelf werkt op modulaire manier. De meeste standaard Python functies zijn niet in de core van de taal ingebouwd maar worden aangeboden via specifieke modules, die je kan importeren wanneer nodig
- We noemen deze importeerbare reeks modules de Python standard library
- Het is aan te raden om, wanneer je programma's langer worden, je code gaat opsplitsen in modules. Ook naar mogelijk later hergebruik van functionaliteit is dit een goeie manier van werken

Modules

- Externe modules bevatten meestal Python broncode, maar ze kunnen ook gecompileerde C of C++ objectbestanden zijn
- C(++) of Python: beide types worden op dezelfde manier gebruikt
- Naast het groeperen van Python objecten helpen modules ook naamgevingsconflicten beheersen. Een objectnaam hoeft enkel uniek te zijn binnen de namespace van een module

Een voorbeeld van een module

```
# mijn_rekenmodule.py:
"""mijn_rekenmodule - Een voorbeeld van een rekenmodule - bevat een variabele en een
functie """
pi = 3.14159
def oppervlakte(straal):
    """oppervlakte(straal): geef de oppervlakte van een cirkel terug met straal
    'straal'."""
    global pi
    return(pi * straal * straal)
```

```
# ander_programma.py:
import mijn_rekenmodule
print(mijn_rekenmodule.pi)
print(mijn_rekenmodule.oppervlakte(3))
```

Klassen onderbrengen in modules...

```
#module.py
class Boek():
    def __init__(self, titel):
        self.titel = titel
```

```
import module

boek1 = module.Boek("Brave New World")
boek2 = module.Boek("World War Z")

print(boek1)
print(boek1.titel)
```

Doe je op dezelfde manier als met functies. Het houdt je code georganiseerd en gestructureerd.

Virtuele Python-omgevingen

Virtuele Python-omgevingen

- Handig Python feature en essentiële kennis voor Python developers
- Per projectfolder een privé-kopie maken van de interpreter en gebruikte dependencies
- Dependencies zijn alle software-componenten (zoals externe libraries) die nodig zijn om je project succesvol te kunnen runnen
- Op deze manier zijn meerdere projecten op je systeem mogelijk, elk met eigen Python-versie en componenten
- Wij gebruiken hier venv: https://docs.python.org/3/library/venv.html. Weet dat er verschillende alternatieven zijn.

Een virtuele omgeving opzetten

- Het venv-script wordt meegeleverd met de Python-interpreter
- De tweede venv op de lijn is de naam die we kiezen voor onze virtuele Python-omgeving. Je kan een andere kiezen, maar is zowat standaard om "venv" te kiezen
- Na uitvoeren van het script wordt een venv folder aangemaakt waarin de interpreter en eventueel ook later geïnstalleerde dependencies wordt opgeslagen

```
python -m venv venv
```

Een virtuele omgeving activeren

- We activeren de venv via een script in de virtuele omgeving
- Zichtbaar aan de prompt van je bash/zsh terminal dat venv actief is

```
# unix, mac:
source venv/bin/activate
# windows:
venv\Scripts\activate
```

Noot: bij een aantal Windows machines hebben we vastgesteld dat je best manueel naar de venv/Scripts directory navigeert en daar de scripts start als volgt: ". activate"

Windows + powershell: "Venv won't activate on this system" => https://dev.to/shriekdj/python-venv-or-virtualenv-wont-activate-on-windows-3e2

Een virtuele omgeving de-activeren

- Gebruik "deactivate" in de terminal om de virtuele omgeving te verlaten

```
deactivate
```

Python installer pip

- We gebruiken pip om externe dependencies te installeren. Meegeleverd met Python en dus ook aanwezig in de virtuele Python-omgeving
- Pip is een recursief acronym dat staat voor "Pip Installs Packages" of "preferred installer program" :-)
- Zorg er voor dat je virtuele omgeving geactiveerd is als je packages installeert
- Bekijk de verzameling van gepubliceerde Python packages op PyPI: https://pypi.org/
- Bijvoorbeeld om de requests-library te installeren:

```
pip install requests
```

Doorgeven of deployen van een virtual environment

- Dumpen naar een bestand:
  - pip freeze > requirements.txt
- Omgeving aanmaken op basis van een bestand:
  - pip install -r requirements.txt

De Python interactieve shell

De Python interactieve shell

- Als je de Python interpreter uitvoert zonder script in een terminal kom je terecht in de interactieve shell
- Veel Python handboeken beginnen met de interactieve interpreter. Ik vermeld het hier, net voor we programma's gaan beginnen schrijven :-)
- Waarom? Het is een handige tool om tijdens het programmeren snel iets uit te proberen
- Ik gebruik het heel vaak in die context
- De shell verlaat je met exit()

De Python interactieve shell

```
python
>>> tekst = "Hallo"
>>> for letter in tekst:
...     print(letter)
...
H
a
l
l
o
>>> exit()
```

Python Development

Objecten en klassen (1)

Kristof Michiels

Inhoud

- Wat zijn objecten en hoe maken we ze?
- Overerving
- Toegang tot attributen
- Soorten methods

Wat zijn objecten en hoe maken we ze?

Wat zijn objecten?

- In Python is alles een object. Dat wordt door Python achter de schermen geregeld
- Je krijgt met objecten zelf te maken wanneer je je eigen objecten wil maken of wanneer je het gedrag van bestaande wil wijzigen
- Onder object verstaan we een aangepaste datastructuur die zowel data (variabelen, noemen we attributen) en code (functies, noemen we methods) bevat
- Een object vertegenwoordigt een uniek exemplaar van een concreet ding. Zie objecten als zelfstandige naamwoorden en hun methoden als werkwoorden. Een object vertegenwoordigt een individueel ding, en zijn methoden bepalen hoe het interageert met andere dingen

Klassen maken met class

- Objecten worden geïnstantieerd - tot leven gebracht - uit een klasse-definitie
- Je kan ze beschouwen als blauwdrukken die beschrijven wat een object allemaal bevat
- We gebruiken class om een klasse te beschrijven
- Voor de klassenaam is het gebruikelijk met een hoofdletter te beginnen
- Je creëert een object door de klasse aan te roepen zoals je een functie aanroept

```
class Kunstwerk():
    pass

kunstwerk_a = Kunstwerk()
kunstwerk_b = Kunstwerk()
```

Object-attributen

- Een attribuut is een variabele binnen een klasse of object
- Je kan attributen toevoegen als een object is gecreëerd, maar ook nog daarna
- Een attribuut kan elk ander object zijn

```
kunstwerk_a.bewaarplaats = "Parijs, Louvre"
kunstwerk_a.gemaakt_in = 1503
kunstwerk_a.hangt_naast = kunstwerk_b
```

Klasse-attributen

- Met attributen worden meestal object-attributen bedoeld
- Er bestaan ook klasse-attributen: deze en hun waarden worden gedeeld door alle geïnstantieerde objecten
- Als je de waarde verandert in een object dan heeft het geen impact op het klasse-attribuut

```
class Kunstwerk:
    soort = "schilderij"

david_van_michelangelo = Kunstwerk()
print(Kunstwerk.soort) # schilderij
print(david_van_michelangelo.soort) # schilderij
david_van_michelangelo.soort = "beeld"
print(david_van_michelangelo.soort) # beeld
print(Kunstwerk.soort) # schilderij
```

Klasse-attributen

- Pas je de waarde van het klasse-attribuut aan dan heeft dit geen effect op de bestaande objecten
- Wel bij nieuwe objecten

```
Kunstwerk.soort = "gravure"
print(Kunstwerk.soort) # gravure
print(david_van_michelangelo.soort) # beeld
mona_lisa_van_leonardo = Kunstwerk()
print(mona_lisa_van_leonardo.soort) # gravure
```

Methods

- Een method is een functie binnen een klasse of object
- Een method ziet eruit als elke andere functie maar kan op meer specifieke manieren worden gebruikt (zien we verder in deze les)

```
class Kunstwerk:
    def vertel(self):
        print("Wat kan ik vertellen? Ik ben een kunstwerk, dat staat vast!")
```

Initialisatie

- Als je object-attributen wil toekennen wanneer een object wordt gecreëerd dan gebruik je de speciale Python object-initialisatie-method __init__()
- btw: de dubbele underscores noemen we dunder ;-) We spreken van dunder-methods
- __init__() initialiseert een individueel object uit de klasse-definitie: het bevat alles om het ene object te onderscheiden van het andere
- De eerste parameter is altijd self: het verwijst naar het individuele object

```
class Kunstwerk:
    def __init__(self):
        pass
```

Initialisatie

- Hier voegen we een parameter naam toe aan de __init__-method
- Wanneer we nu een object instantiëren moeten we een string meegeven voor de parameter "naam"

```
class Kunstwerk:
    def __init__(self, naam, kunstenaar):
        self.naam = naam
        self.kunstenaar = kunstenaar

mona_lisa_van_leonardo = Kunstwerk("Mona Lisa", "Leonardo da Vinci")
```

Wat gebeurt hier bij de instantiëring?

- De beschrijving van de Kunstwerk klasse wordt opgezocht
- Een nieuw object wordt in het geheugen geïnstantieerd/gecreëerd
- De __init__-method wordt aangeroepen: het nieuwe object wordt als self meegegeven, alsook de andere argumenten naam en kunstenaar
- De waarden van naam en kunstenaar worden opgeslagen in het object
- Het nieuwe object wordt teruggegeven en toegekend aan de variabele mona_lisa_van_leonardo
- Binnen de klasse-definitie verwijs je naar de attributen als self.attribuutnaam
- Het nieuwe object is zoals elk ander object in Python

Het self-argument

Python gebruikt self om de juiste attributen en methods te kunnen vinden

```
class Kunstwerk:
    def vertel(self):
        print("Wat kan ik vertellen? Ik ben een kunstwerk, dat staat vast!")

een_kunstwerk = Kunstwerk()
een_kunstwerk.vertel() # Wat kan ik vertellen? Ik ben een kunstwerk, dat staat vast!
```

Achter de schermen gebeurt het volgende:

```
Kunstwerk.vertel(een_kunstwerk)
```

Overerving

Overerving

- Bij het coden zal je vaak beroep kunnen doen op een bestaande klasse die objecten creëert op een manier die benadert wat je zelf nodig hebt
- Je kan de klasse aanpassen, of een nieuwe creëren met dezelfde code maar dat is niet duurzaam en efficiënt
- Een oplossing is overerving: een nieuwe klasse creëren afgeleid van een andere, maar met enkele toevoegingen of wijzigingen: een goede manier om code te herbruiken

```
class AangepasteFout(Exception):
    pass

raise AangepasteFout
```

Erven van een basis-klasse

- In je afgeleide klasse definieer je enkel wat je wenst toe te voegen of te wijzigen. Dit heft (in het Engels: overrides) het gedrag van de basis-klasse op
- Voor de originele klasse hebben we verschillende namen: parent, superclass of base klasse. Idem voor de afgeleide klasse: child, subclass of derived klasse

```
class Dier():
    pass

class Aap(Dier):
    pass

een_dier = Dier()
een_aapje = Aap()
```

Erven van een basis-klasse

- Een afgeleide klasse is een gespecialiseerde versie van een basis-klasse
- De relatie tussen beiden is er één van "IS EEN": een Aap is-een Dier
- Je kan checken of een klasse een afgeleide klasse is door gebruik van de issubclass()-functie

```
print(issubclass(Aap, Dier)) # True
```

Erven van een basis-klasse

Een afgeleide klasse erft alle functionaliteit van de basis-klasse

```
class Dier():
    def spreek(self):
        print("Ik ben een dier")

class Aap(Dier):
    pass

een_dier = Dier()
een_aapje = Aap()
een_dier.spreek() # Ik ben een dier
een_aapje.spreek() # Ik ben een dier
```

Gedrag opheffen of overriden

We kunnen elke method overriden, ook de __init__()-method

```
class Dier():
    def __init__(self, naam):
                self.naam = naam
    def spreek(self):
        print("Ik ben een dier")

class Aap(Dier):
    def __init__(self, naam):
        self.naam = "Meneer " + naam
    def spreek(self):
        print("Ik ben een aapje")
```

Methods toevoegen aan een afgeleide klasse

Een afgeleide klasse kan ook één of meerdere nieuwe methods toevoegen

```
class Dier():
    def __init__(self, naam):
                self.naam = naam
    def spreek(self):
        print("Ik ben een dier")

class Aap(Dier):
    def __init__(self, naam):
        self.naam = "Meneer " + naam
    def spreek(self):
        print("Ik ben een aapje")
    def vlooien(self):
        print("Vlooien, zo ontspannend, mmmm")
```

super()

Gebruik super() wanneer de afgeleide klasse iets nieuws doet (bvb een method overschrijft) maar toch iets nodig heeft van de base-klasse

```
class Dier():
    def __init__(self, naam):
        self.naam = naam

class Aap(Dier):
    def __init__(self, naam, verwantschap_mens):
        super().__init__(naam)
        self.verwantschap_mens = verwantschap_mens
```

Meervoudige overerving en de mro

- Objecten kunnen erven van meerdere base-klassen
- Wanneer je klasse verwijst naar een method of attribuut dat de afgeleide klasse niet heeft, dan zal Python zoeken in alle base-klassen
- Python volgt daarbij een "method resolution order"
- Elke klasse heeft een mro()-method die een list teruggeeft met daarin alle klassen - in volgorde - die zullen doorzocht worden naar een method of attribuut. De eerste die wordt gevonden wint
- Er is een gelijkaardig attribuut __mro__ die dezelfde list teruggeeft

Meervoudige overerving en de mro

```
class A():
        def toonA(self):
                print('Je bent in klasse A')

class B():
        def toonB(self):
                print('Je bent in klasse B')

class C(A, B):
        def toonC(self):
                print('Je bent in klasse C')

klasse_c = C()
klasse_c.mro()
```

Mixins

- Je kan zonder problemen een extra basis-klasse toevoegen aan je klasse-beschrijven, die enkel als helper-klasse zal functioneren: bvb logging is een goed voorbeeld
- Deze klasse deelt dan geen gemeenschappelijk methods met de eventuele andere basis-klassen
- Dergelijke basis-klassen worden vaak mixin-klassen genoemd

```
class MijnMixin():
    def loggen(self):
        import pprint
        pprint.pprint(vars(self))

class MijnKlasse(MijnMixin):
    pass
```

Toegang tot attributen

Directe toegang tot attributen

Object attributen en methods zijn in Python (normaal gesproken) publiek. Guido Van Rossum daarover: "we zijn allemaal volwassen"

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.naam = invoer_naam

mona_lisa_van_leonardo = Kunstwerk("Mona Lisa")
print(mona_lisa_van_leonardo.naam) # Mona Lisa
mona_lisa_van_leonardo.naam = "Mano Lasi"
print(mona_lisa_van_leonardo.naam) # Mano Lasi
```

Getter- en setter-methods

- Programmeertalen ondersteunen vaak private attributen die niet van buitenaf kunnen benaderd of gewijzigd worden. Developers moeten in dat geval getter- en setter-methods schrijven
- Python heeft geen private attributen, maar je kan wel getters en setters schrijven met aangepaste attributen-namen om wat veiligheid in te bouwen

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.verborgen_naam = invoer_naam
    def get_naam(self):
        return self.verborgen_naam
    def set_naam(self, invoer_naam):
        self.verborgen_naam = invoer_naam
```

Toegang tot attributen via properties

De beste manier om toegang te regelen tot attributen is via properties. Het kan op 2 manieren:

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.verborgen_naam = invoer_naam
    def get_naam(self):
        return self.verborgen_naam
    def set_naam(self, invoer_naam):
        self.verborgen_naam = invoer_naam
    naam = property(get_naam, set_naam)

mona_lisa_van_leonardo = Kunstwerk("Mona Lisa")
print(mona_lisa_van_leonardo.naam) # Mona Lisa
mona_lisa_van_leonardo.naam = "Mano Lasi"
print(mona_lisa_van_leonardo.naam) # Mano Lasi
```

Toegang tot attributen via properties

De tweede manier is door gebruik van decorators en door het vervangen van de method-namen get_naam en set_naam door naam:

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.verborgen_naam = invoer_naam
    @property
    def naam(self):
        return self.verborgen_naam
    @naam.setter
    def naam(self, invoer_naam):
        self.verborgen_naam = invoer_naam
```

Properties voor berekende waarden

- Een property kan ook een berekende waarde (een computed value) teruggeven
- Geef je geen setter-property mee, dan kan je de property niet veranderen (handig voor read-only attributen)

```
class Rechthoek():
    def __init__(self, basis, hoogte):
        self.basis = basis
        self_hoogte = hoogte
    @property
    def oppervlakte(self):
        return self.basis * self.hoogte

mijn_rechthoek = Rechthoek(4,6)
print(mijn_rechthoek.oppervlakte) # 24
```

Naamaanpassing voor meer privacy

- Python heeft een conventie voor attributen die niet zichtbaar horen te zijn buiten hun klasse definitie
- Je laat ze beginnen met 2 underscores: __
- Biedt geen perfecte bescherming, maar overtredingen zullen altijd zo bedoeld zijn :-)

```
class Kunstwerk():
    def __init__(self, invoer_naam):
        self.__naam = invoer_naam
    @property
    def naam(self):
        return self.__naam
    @naam.setter
    def naam(self, invoer_naam):
        self.__naam = invoer_naam
```

Soorten methods

Soorten methods

- Methods kunnen onderdeel van de klasse zelf zijn, terwijl andere onderdeel zijn van de objecten die uit die klasse worden geïnstantieerd (instance methods). Nog andere zijn geen van bovenstaande
- Indien de method niet wordt voorafgegaan door een decorator (@...) is het een instance method met als eerste argument self (deze hebben we tot nu toe gezien)
- Indien de method wordt voorafgegaan door de @classmethod decorator is het een class method met als eerste argument cls
- Indien de method wordt voorafgegan door de @staticmethod decorator is het een static method: het eerste argument is dan geen object of klasse

Class methods

- Een class method heeft betrekking op de klasse in zijn geheel
- Elke wijziging die je maakt aan de klasse beïnvloedt alle geïnstantieerde objecten
- Een voorafgaande @classmethod decorator geeft aan dat de functie die volgt een class method is
- De eerste parameter is de klasse zelf. Conventie is om deze parameter cls te noemen

Class methods

```
class Kunstwerk():
    teller = 0
    def __init__(self, invoer_naam):
        self.__naam = invoer_naam
        Kunstwerk.teller += 1
    @classmethod
    def aantal(cls):
        print("Kunstwerk heeft", cls.teller, "prachtige kunstwerken."

kunstwerk_a = Kunstwerk("Mona Lisa")
kunstwerk_b = Kunstwerk("David")
print(Kunstwerk.aantal) # Kunstwerk heeft 2 prachtige kunstwerken
```

Static methods

- Static methods hebben geen invloed op de klasse of de geïnstantieerde objecten
- We gebruiken ze voor ons gemak en bundelen ze in de klasse voor organisatorische redenen
- We geven het aan met de @staticmethod decorator en gebruiken geen self of cls parameter
- We hoeven ook geen object te instantiëren om de method te gebruiken

```
class Museum():
    @staticmethod
    def open_op_zondag():
        print('Het museum is open op zondag')

print(Museum.open_op_zondag()
```

Python Development - les 10 -

kristof.michiels01@ap.be

Python Development

Objecten en klassen (2)

Kristof Michiels

Inhoud

- Objectgeoriënteerd programmeren
- Compositie en aggregatie
- Magische methods
- Abstracte basisklassen

Objectgeoriënteerd programmeren

Objectgeoriënteerd programmeren

- Techniek die mogelijk maakt om variabelen en functies te groeperen
- Dat groeperen gebeurt binnen een data type dat we een klasse noemen
- Je bouwt een systeem dat bestaat uit objecten
- Door je code te organiseren binnen klasses kan je je programma opdelen in kleinere onderdelen die eenvoudiger te begrijpen en te debuggen zijn

Objectgeoriënteerd programmeren

- Voor kleinere programma's: weinig organisatiewinst, meer werk en complexiteit
- OOP (bij Python) is optioneel
- Python core developer Jack Diederich's PyCon 2012: “Stop Writing Classes” (https://youtu.be/o9pEzgHorH0/)
- In sommige gevallen werkt een eenvoudige functie of module beter
- Desalniettemin: terecht populaire keuze en vaak de efficientste manier

Software modellen bouwen van fysieke objecten

- Als we fysieke objecten beschrijven verwijzen we vaak naar hun eigenschappen
  - Een auto: kleur, afmetingen, merk, aantal deuren ...
- Sommige objecten hebben eigenschappen die slaan op hen en niet op andere objecten
  - Een doos: afgesloten, open, leeg, vol. Niet van toepassing op bvb een fiets
- Sommige objecten zijn in staat om acties te vervullen.
  - Een auto kan vooruit gaan, achteruit, links en rechts

Software modellen bouwen van fysieke objecten

- Als we objecten uit ons leven willen modelleren in code: beslissen welke eigenschappen het object zullen representeren en welke operaties het object kan vervullen
- Dit noemt men de *state* (data) en *behavior* (gedrag of acties) van een object

State en behavior: een procedureel voorbeeld

```
def zetAan():
    global schakelaarIsAan
    schakelaarIsAan = True

def zetUit():
    global schakelaarIsAan
    schakelaarIsAan = False

schakelaarIsAan = False
print(schakelaarIsAan)
zetAan()
print(schakelaarIsAan)
zetUit()
print(schakelaarIsAan)
zetUit()
print(schakelaarIsAan)
```

State en *behavior*: een procedureel voorbeeld

- De schakelaar kan in 2 posities zijn: aan of uit
- Om de *state* te modelleren hebben we hier een Booleaanse variabele nodig
  - We noemen hem schakelaarIsAan: True betekent aan, False betekent uit
  - Initieel staat de schakelaar op uit
- *Behavior*: de schakelaar kan twee acties uitvoeren, "zet aan" en "zet uit"
  - Hiervoor bouwen we 2 functies, zetAan() en zetUit()
  - Deze zetten de waarde van de variabele op respectievelijk aan en uit
- Op het einde is er wat testcode toegevoegd die wat met de schakelaar "speelt"

Richting OO...

- Voorgaande is een zeer eenvoudig voorbeeld
- Je botst snel op de limieten van deze aanpak wanneer je een tweede schakelaar wil aanmaken
- De code is niet erg herbruikbaar, terwijl dit net het doel is van functies
- Zondigt tegen het DRY principe
- We gaan deze code opnieuw schrijven volgens de OO principes

Het procedureel voorbeeld herschreven

```
class LichtSchakelaar():
    def __init__(self):
        self.schakelaarIsAan = False

    def zetAan(self):
        # zet de schakelaar aan
        self.schakelaarIsAan = True

    def zetUit(self):
        # zet de schakelaar uit
        self.schakelaarIsAan = False

schakelaar1 = LichtSchakelaar()
print(schakelaar1.schakelaarIsAan)
schakelaar1.zetAan()
print(schakelaar1.schakelaarIsAan)
```

Het procedureel voorbeeld herschreven

- De klasse definieert een enkele variabele, schakelaarIsAan. Deze wordt geïnitialiseerd in een functie.
- Ze bevat nog twee andere functies voor het gedrag: zetAan() en zetUit()
- Net zoals functies moeten aangeroepen worden dien je Python expliciet te vertellen om een object van de klasse te maken
  - In onze code dragen we op om de klasse te vinden
  - Er een object uit te creëren (of instantiëren)
  - En het resulterende object toe te kennen aan de variabele schakelaar1

Klassen onderbrengen in modules...

```
#module.py
class Boek():
    def __init__(self, titel):
        self.titel = titel
```

```
import module

boek1 = module.Boek("Brave New World")
boek2 = module.Boek("World War Z")

print(boek1)
print(boek1.titel)
```

Doe je op dezelfde manier als met functies. Het houdt je code georganiseerd en gestructureerd.

Compositie en aggregatie

Compositie en aggregatie

- Overerving is een zeer bruikbare techniek om een systeem van klassen uit te bouwen
  - Het is handig wanneer klassenfunctionaleit kunnen delen vanuit een base klasse (een "is een"-relatie)
  - Maar het kan uitnodigen tot de vorming van zeer complexe hiërarchieën

Compositie en aggregatie

- Vaak zijn compositie en aggregatie nuttiger om een systeem van klassen te vormen
  - Dit is het bouwen van complexe objecten uit andere meer eenvoudige objecten
  - Bij compositie is één object een deel van een ander object (een "heeft-een"-relatie)
  - Aggregatie drukt ook een "heeft-een" relatie uit, maar deze is iets losser: een object gebruikt een ander object, maar ze kunnen los van elkaar bestaan
- Verschillende ideeën kunnen geïsoleerd worden en in hun eigen klassen geplaatst
- En uiteraard: overerving en compositie kunnen gecombineerd worden

Compositie en aggregatie: basisvoorbeeld

```
class Soort():
    def __init__(self, naam):
        self.naam = naam

class Kunstenaar():
    def __init__(self, naam):
        self.naam = naam

class Kunstwerk():
    def __init__(self, soort, kunstenaar):
        self.soort = soort
        self.kunstenaar = kunstenaar

    def __str__(self):
        return f"Dit kunstwerk is een {self.soort.naam} en werd gemaakt door
        {self.kunstenaar.naam}"
```

Compositie en aggregatie: een voorbeeld

```
een_schilderij = Soort("schilderij")
een_kunstenaar = Kunstenaar("Francis Picabia")
een_kunstwerk = Kunstwerk(een_schilderij, een_kunstenaar)
print(een_kunstwerk)
# Dit kunstwerk is een schilderij en werd gemaakt door Francis Picabia
```

Compositie en aggregatie: een voorbeeld (2)

```
class Boek():
    def __init__(self, titel, prijs, auteur=None):
        self.titel = titel
        self.prijs = prijs
        self.auteur = auteur
        self.hoofdstukken = []

    def voeghoofdstuktoe(self, hoofdstuk):
        self.hoofdstukken.append(hoofdstuk)

    def getboekpaginatelling(self):
        resultaat = 0
        for h in self.hoofdstukken:
            resultaat += h.paginatelling
        return resultaat
```

Compositie en aggregatie: een voorbeeld (2)

```
class Auteur():
    def __init__(self, voornaam, familienaam):
        self.voornaam = voornaam
        self.familienaam = familienaam

    def __str__(self):
        return f"{self.voornaam} {self.familienaam}"


class Hoofdstuk():
    def __init__(self, naam, paginatelling):
        self.naam = naam
        self.paginatelling = paginatelling
```

Compositie en aggregatie: een voorbeeld (2)

```
auteur1 = Auteur("Leo", "Tolstoy")
boek1 = Boek("War and Peace", 39.95, auteur1)

boek1.voeghoofdstuktoe(Hoofdstuk("Hoofdstuk 1", 102))
boek1.voeghoofdstuktoe(Hoofdstuk("Hoofdstuk 2", 91))
boek1.voeghoofdstuktoe(Hoofdstuk("Hoofdstuk 3", 124))

print(boek1.titel)
print(boek1.auteur)
print(boek1.getboekpaginatelling())
```

Magische Methods

Magische Methods

- Magische methods (*magic methods* in het Engels) zijn speciale methods in Python
- Ze beginnen en eindigen telkens met dubbele underscores en worden hierdoor ook dunder-methods genoemd
- Ze zijn niet bedoeld om rechtstreeks door jou te worden aangeroepen. Het aanroepen gebeurt intern vanuit de klasse bij een bepaalde actie
- Enkele voorbeelden: __init__ (zagen we reeds in de vorige les), __add__, __len__, __repr__...

Magische methods

- Wanneer je in je code twee objecten bij elkaar optelt, zoals getal = 3 + 8, hoe weten deze integer objecten hoe ze de "+" moeten implementeren?
- Of in het geval van strings: kunstenaar = "Damien" + " " + "Hirst", hoe weet Python hoe deze strings aan elkaar te plakken?
- En in beide bovenstaande voorbeelden: hoe weten de objecten getal en kunstenaar hoe ze de "=" moeten implementeren?
- Het antwoord is telkens: magische methods :-)

Magische Methods: voorbeeld

- We vertrekken in dit voorbeeld van een eenvoudige klasse Kunstenaar. We maken een gelijk_aan method die de naam case-neutraal vergelijkt
- Het zou interessant zijn om 2 objecten te kunnen vergelijken en magische methods maken dit mogelijk

```
class Kunstenaar():
    def __init__(self, naam):
        self.naam = naam
    def gelijk_aan(self, andere_kunstenaar):
        return self.naam.lower() == andere_kunstenaar.naam.lower()

kunstenaar1 = Kunstenaar("Andy Warhol")
kunstenaar2 = Kunstenaar("ANDY WARHOL")
print(kunstenaar1.gelijk_aan(kunstenaar2)) # True
```

Magische Methods: voorbeeld

- Het enige dat we moeten doen is de method gelijk_aan te vervangen door de magische method __eq__

```
class Kunstenaar():
    def __init__(self, naam):
        self.naam = naam
    def __eq__(self, andere_kunstenaar):
        return self.naam.lower() == andere_kunstenaar.naam.lower()

kunstenaar1 = Kunstenaar("Andy Warhol")
kunstenaar2 = Kunstenaar("ANDY WARHOL")
print(kunstenaar1 == kunstenaar2) # True
```

Magische methods voor vergelijkingen

- __eq__(self, ander) self == ander
- __ne__(self, ander) self != ander
- __lt__(self, ander) self < ander
- __gt__(self, ander) self > ander
- __le__(self, ander) self <= ander
- __ge__(self, ander) self >= ander

Overzicht van magische methods: https://docs.python.org/3/reference/datamodel.html#special-method-names

Magische methods voor wiskundige operaties

- __add__(self, ander) self + ander
- __sub__(self, ander) self – ander
- __mul__(self, ander) self * other
- __floordiv__(self, ander) self // ander
- __truediv__(self, ander) self / ander
- __mod__(self, ander) self % ander
- __pow__(self, ander) self ** ander

Overzicht van magische methods: https://docs.python.org/3/reference/datamodel.html#special-method-names

Andere vaak gebruikte magische methods

- Naast __init__ ga je wellicht het meest gebruik maken van __str__ in je eigen methods
- Met deze method geef je mee hoe het object moet worden geprint. Het wordt gebruikt door print() en str()
- De interactieve interpreter gebruikt de __repr__ method om variabelen te outputten

```
class Kunstenaar():
    def __init__(self, naam):
        self.naam = naam
    def __str__(self):
        return f"Hallo, ik ben {self.naam}!"

kunstenaar1 = Kunstenaar("Andy Warhol")
print(kunstenaar1) # Hallo, ik ben Andy Warhol
```

Abstracte basisklassen

Abstracte basisklassen

- Je wil een basisklasse voorzien die een template biedt voor overerving naar andere klassen
- Enkel een blauwdruk, kan niet geïnstantieerd worden
- Subklassen worden dan concrete implementaties van dat idee
- Je wil afdwingen dat bepaalde methods moeten geïmplementeerd worden in de subklassen

Abstracte basisklassen

```
from abc import ABC, abstractmethod

class GrafischeVorm(ABC):
    def __init__(self):
        super().__init__()

    @abstractmethod
    def berekenOppervlakte(self):
        pass

class Cirkel(GrafischeVorm):
    def __init__(self, straal):
        self.straal = straal

    def berekenOppervlakte(self):
        return 3.14 * (self.straal ** 2)
```

Abstracte basisklassen

```
class Vierkant(GrafischeVorm):
    def __init__(self, zijde):
        self.zijde = zijde

    def berekenOppervlakte(self):
        return self.zijde * self.zijde

cirkel1 = Cirkel(10)
print(cirkel1.berekenOppervlakte())
vierkant1 = Vierkant(12)
print(vierkant1.berekenOppervlakte())
```

Abstracte basisklassen

- We maken gebruik van de ABC-module van de standard library
- We gebruiken de @abstractmethod decorator om aan te geven dat een method abstract is en elke subklasse deze method moet overschrijven

Interfaces

- Python geen expliciete taalondersteuning voor interfaces
- Een soort contract of belofte om een bepaald gedrag of mogelijkheden te hebben
- Hier: combinatie van meervoudige overerving en ABC
- We zouden de JSONify klasse kunnen gebruiken als interface voor een andere klasse

Interfaces

```
from abc import ABC, abstractmethod

class GrafischeVorm(ABC):
    def __init__(self):
        super().__init__()

    @abstractmethod
    def berekenOppervlakte(self):
        pass

class JSONify(ABC):
    @abstractmethod
    def toJSON(self):
        pass
```

Interfaces

```
class Cirkel(GrafischeVorm, JSONify):
    def __init__(self, straal):
        self.straal = straal

    def berekenOppervlakte(self):
        return 3.14 * (self.straal ** 2)

    def toJSON(self):
        return f"{{ \"cirkel\": {str(self.berekenOppervlakte())} }}"


cirkel1 = Cirkel(10)
print(cirkel1.berekenOppervlakte())
print(cirkel1.toJSON())
```