WT'S ADVANCED PYTHON LESBOEKJE

ALGORITMEN EN OBJECT GEORIËNTEERD PROGRAMMEREN

Jochum. van Weert
j.v.weert (at) sgdb.nl
Stedelijk Gymnasium 's-Hertogenbosch
v4.0 2025

INHOUD

1 INLEIDING

Je hebt inmiddels aardig wat ervaring met Python. Je hebt vorig jaar geleerd over loops en variabelen, functies en lijsten, floats en integers, etc. Kortom je hebt de belangrijkste python commando's en constructies geleerd en deze toegepast op verschillende opdrachten en problemen. We breiden nu je programmeer skills uit met 2 belangrijke principes: **Algoritmen** en **Object-Oriëntatie**.

Algoritmen:
Algoritmen zijn de "recepten" van de informatica. Voordat je een probleem met code te lijf kunt gaan, kun je van tevoren nadenken over je strategie. Hoe pak ik het probleem aan? Een algoritme is een uitgewerkte aanpak van een probleem. Een stappenplan dat daarna relatief makkelijk in code om is te zetten. Voor veel standaard problemen (sorteren, zoeken, routes bepalen, etc.) zijn in de loop der jaren veel slimme algoritmen bedacht. We maken kennis met een aantal van deze algoritmen.
Vaak zijn er meerdere algoritmen die bruikbaar zijn voor een probleem. We kijken ook hoe je de efficiëntie van een algoritme bepaalt, zodat we verschillende algoritmen kunnen vergelijken en de beste kunnen kiezen voor de toepassing die je wilt maken.

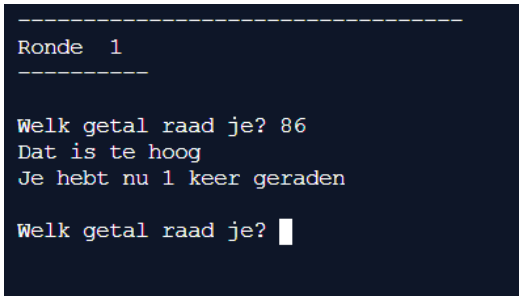
Object-Oriëntatie:
Objecten bieden je een heel nieuwe manier om je code op te bouwen. Zodra je grotere programma's gaat schrijven en helemaal als je samen met anderen aan dezelfde code gaat werken, bieden objecten een hele boel voordelen:

- Objecten bieden extra structuur en maken daarmee je code overzichtelijker
- Objecten helpen je voorkomen dat je code herhaalt. Dit voorkomt fouten en maakt je code beter te onderhouden
- Objecten maken code recyclebaar: objecten die je eerder geschreven hebt, kun je gemakkelijk in nieuwe programma's importeren en erop voortbouwen.
- Objecten maken het makkelijker je code in stukken te verdelen en daarmee is het makkelijker om samen te werken aan code (ieder zijn eigen afgebakende stuk). Je hoeft dan alleen nog maar afspraken te maken hoe de stukken code moeten samenwerken.

Je maakt je eerst de fijne kneepjes van bovenstaande onderwerpen eigen. Daarna gaan we uiteraard je nieuwe skills loslaten op het bouwen van een game. We gooien het wel over een andere boeg: we maken een tekst-adventure. Weer eens wat anders en omdat we ons dan niet druk hoeven te maken over graphics, kunnen we ons lekker focussen op het goed opbouwen van de code volgens Object-georiënteerde principes.

2 ALGORITMEN

2.1 INLEIDING: EEN RAADSPELLETJE



Om erachter te komen wat een algoritme eigenlijk is, gaan we een simpel raadspelletje spelen. De computer kiest een getal tussen de 1 en de 100 en jij moet raden. Als je een gok doet, vertelt de computer of het te hoog, te laag of precies goed was. Daarna mag je nog eens raden en je moet met zo min mogelijk stappen het juiste getal zien te vinden.

Je kunt het spelletje spelen via de link hieronder. Probeer het eens uit (klik rechtsbovenaan het scherm op "run in browser") en denk goed na wat je strategie precies is, terwijl je het spelletje speelt:

https://www.pythonmorsels.com/p/2gxtj/

Opdracht 2.1

Je hebt (misschien na even proberen) vast een goede strategie gevonden om het snel te raden. Omschrijf je strategie eens in woorden. Probeer zo precies mogelijk te zijn.

De strategie die je hebt gevolgd om het getal te raden is een algoritme. Je voert een aantal stappen steeds uit, net zo lang tot je het "antwoord" gevonden hebt.

Nu ben je zelf de uitvoerder geweest van het algoritme, maar meestal laten we dat de computer uitvoeren. Hiervoor moet je het algoritme precies beschrijven, zodat het daarna makkelijk naar (python)code kan worden omgezet. Bij het beschrijven van je strategie bij opdracht 2.1 zul je gemerkt hebben dat het erg lastig is om precies te zijn bij zo'n beschrijving. Daar moeten we dus even wat aan doen…

2.2 EEN ALGORITME BESCHRIJVEN

Een algoritme staat in principe los van een specifieke programmeertaal. Je kunt hetzelfde algoritme (bijvoorbeeld om snel het getal tussen 1 en 100 te raden), in veel verschillende talen opschrijven. De precieze details en schrijfwijzen verschillen tussen de talen, maar het algoritme blijft hetzelfde.

Hier zie je een voorbeeld van een stukje code met dezelfde functionaliteit, geschreven in 2 verschillende programmeertalen:

Python:	**Java:**

`snack = input("Favoriete snack? ")`
 	
`if (snack == "appeltaart"):`
`  print("Oh yeah!")`
`else:`
`  print("Gatverdamme")`
	`System.out.print("Favoriete snack? ");`
	`String snack = System.console().readLine();`
	
	`if (snack.equals("appeltaart"))`
	`  System.out.println("Oh yeah!");`
	`else`
	`  System.out.println("Gatverdamme");`

Zoals je ziet verschilt de code nogal van elkaar en dat heeft te maken met verschillen in opzet en schrijfwijzen bij verschillende programmeertalen. Toch zijn beide stukken code functioneel gelijk: ze hebben hetzelfde resultaat. Je zou dus kunnen zeggen dat ze beiden hetzelfde algoritme implementeren. *(Het vrij eenvoudige algoritme dat bepaalt of iemand van de juiste snacks houdt…)*

Je kunt je dus voorstellen dat het handig is om een algoritme te kunnen beschrijven op een manier die los staat van de specifieke implementatie (dus een beschrijving die geen Python of Java code gebruikt). Hiervoor zijn 2 gangbare opties: **pseudocode** en **flowcharts.**

2.2.1 PSEUDOCODE

De naam pseudocode zegt het eigenlijk al: het is een soort programmeercode, maar niet echt. De "programma's" die je opschrijft in pseudocode zijn niet direct uitvoerbaar door een computer. De schrijfwijze is net wat "losser" dan nodig is om het door een computer uit te laten voeren, maar gebruikt wel een "programmeer-achtige" notatie.

Het voorbeeld van hierboven zou in pseudocode zoiets kunnen zijn als:

```
Lees invoer
Als invoer gelijk aan "appeltaart"
  schrijf "Oh yeah!"
Als invoer ongelijk aan "appeltaart"
  schrijf "Gatverdamme"
```

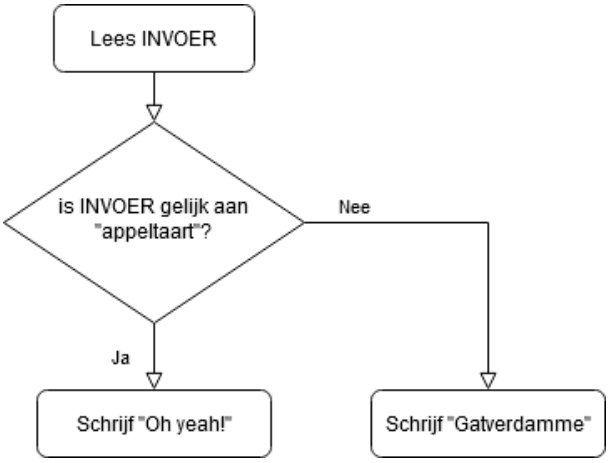
(Je ziet dat de pseudocode meer op de python versie dan op de Java versie lijkt. Dat is ook een van de redenen dat python een fijne taal is om in te leren programmeren: het heeft een wat minder "cryptische" schrijfwijze)

Pseudocode komt in vele vormen voor en is een nuttig instrument om algoritmen te beschrijven. In dit boekje kiezen we echter voor het alternatief: **flowcharts**.

Meer info en voorbeelden van pseudocode vind je op Wikipedia.

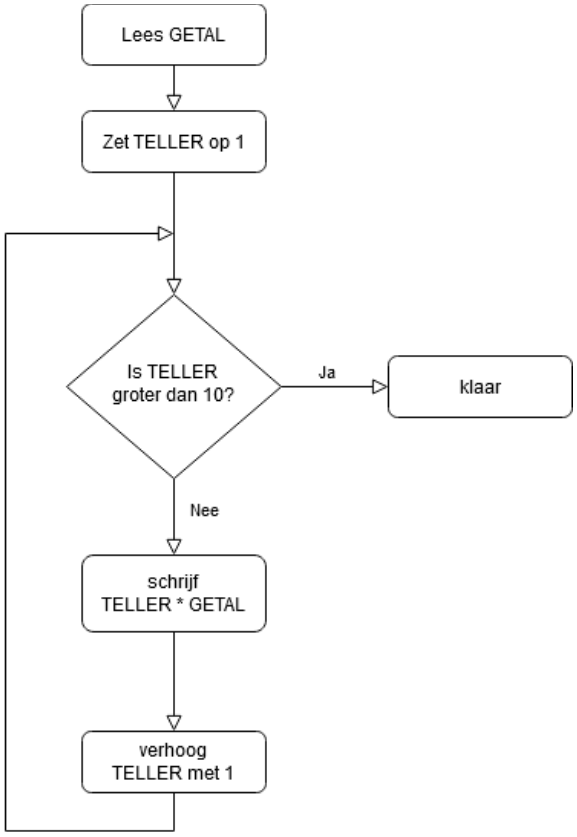
2.2.2 FLOWCHARTS

Flowcharts zijn een grafische manier om een algoritme weer te geven. Je bent ze vast wel eens tegengekomen. Het eenvoudige algoritme van de snacks zou er in een flowchart zo uit kunnen zien:



Je ziet dat de keuze (die je in een programmeertaal meestal als een if-else zou implementeren) wordt weergegeven als een ruit met daaruit 2 pijlen. In de ruit staat een vraag waarop het antwoord “ja” of “nee” is (een zogenaamde Boolean vraag). Afhankelijk van het antwoord wordt er een vervolgpad gekozen.

Een veel voorkomende constructie in algoritmes (en dus in computerprogramma’s) is de herhaling (loop). Die kunnen ook gemakkelijk in flowcharts worden opgenomen:



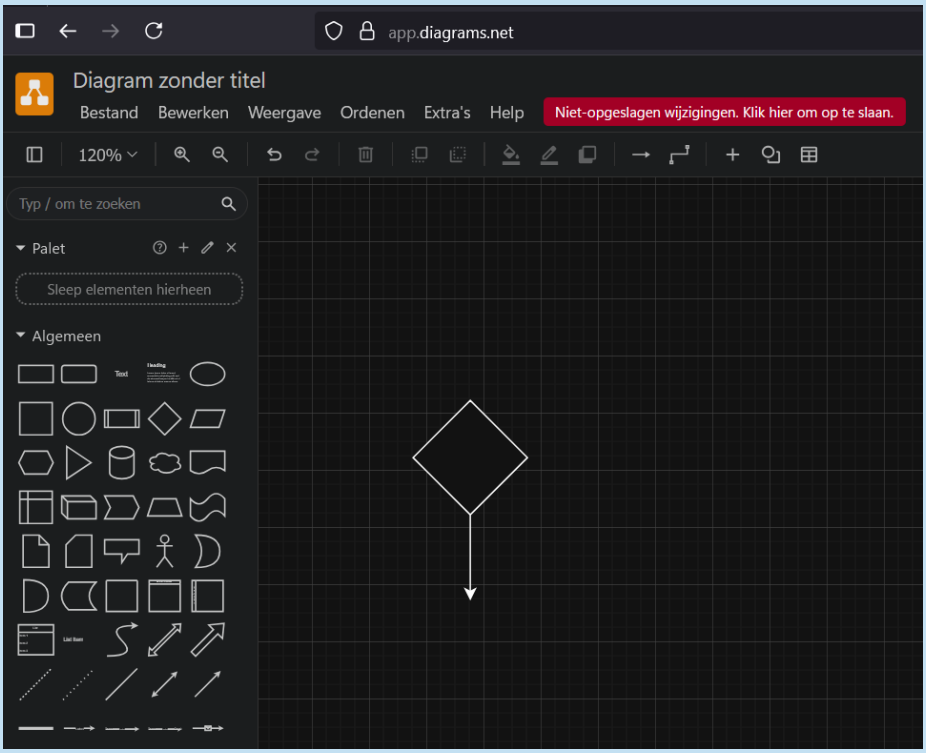
Opdracht 2.2
Leg in je eigen woorden uit wat voor algoritme de flowchart hierboven beschrijft.

Opdracht 2.3
Een algoritme slaat natuurlijk meestal op iets in een computerprogramma, maar elk "stappenplan" is feitelijk een algoritme.

Bekijk het volgende "algoritme":

- Als een eersteklasser zijn boeken is vergeten wordt er een kruisje in Magister genoteerd.
- Bij de eerste 3 kruisjes krijgt de leerling gewoonweg een mondelinge waarschuwing.
- Bij het 4e kruisje volgt een gesprek bij de Teamleider van klas 1
- Bij het 5e of opvolgende kruisjes wordt de leerling een dag geschorst.

Maak een flowchart van dit algoritme. Dat mag met pen en papier. Digitaal kan ook. De webtool https://app.diagrams.net/ is hier heel geschikt voor:



2.3 ALGORITMES VERGELIJKEN: ZOEKALGORITMES

Soms zijn er meerdere algoritmes te bedenken bij hetzelfde probleem. Een algoritme noemen we "**correct**" als het voor elke geldige invoer het juiste antwoord geeft. Correctheid is natuurlijk erg belangrijk, maar vaak zijn niet alle algoritmes even efficiënt. Laten we er eens twee bekijken.

Bij het spelen van het raadspelletje tussen 1 en 100 ben je zeer waarschijnlijk op een strategie uitgekomen die grofweg op het volgende neerkomt:

Raad eerst het getal 50.

- Als 50 te hoog is raad je 25
    - Als 25 te hoog is raad je 12
        - Als dit te hoog is…
            - …
            - …
        - Als dit te laag is…
            - …
            - …
    - Als 25 te laag is raad je 37
        - Als dit te hoog is…
            - …
            - …
        - Als dit te laag is…
            - …
            - …
- Als 50 te laag is raad je 75
    - Als 75 te hoog is raad je 62
        - Als dit te hoog is…
            - …
            - …
        - Als dit te laag is…
            - …
            - …
    - Als 75 te laag is raad je 87
        - Als dit te hoog is…
            - …
            - …
        - Als dit te laag is…
            - …
            - …

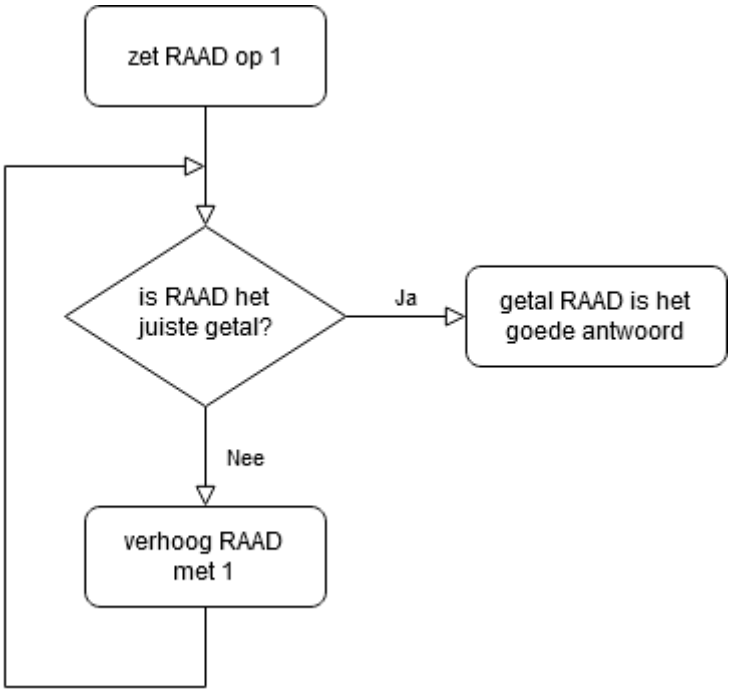
Op deze manier kom je vrij snel bij de oplossing. *(Als je het heel precies en systematisch volgt, kun je elk getal tussen de 1 en de 100 altijd raden in maximaal 7 stappen. Probeer het maar eens).*

Deze strategie is een erg bekend en efficiënt zoekalgoritme met de naam **Binary search**. We komen er zo nog verder op terug.

2.3.1 LINEAR SEARCH

Er is ook een simpeler algoritme dat de klus klaart, maar wel een stuk minder efficiënt is: **Linear search.**

Bekijk de flowchart van Linear search:



Opdracht 2.4

Leg in je eigen woorden uit hoe Linear Search werkt, gebaseerd op bovenstaande flowchart.

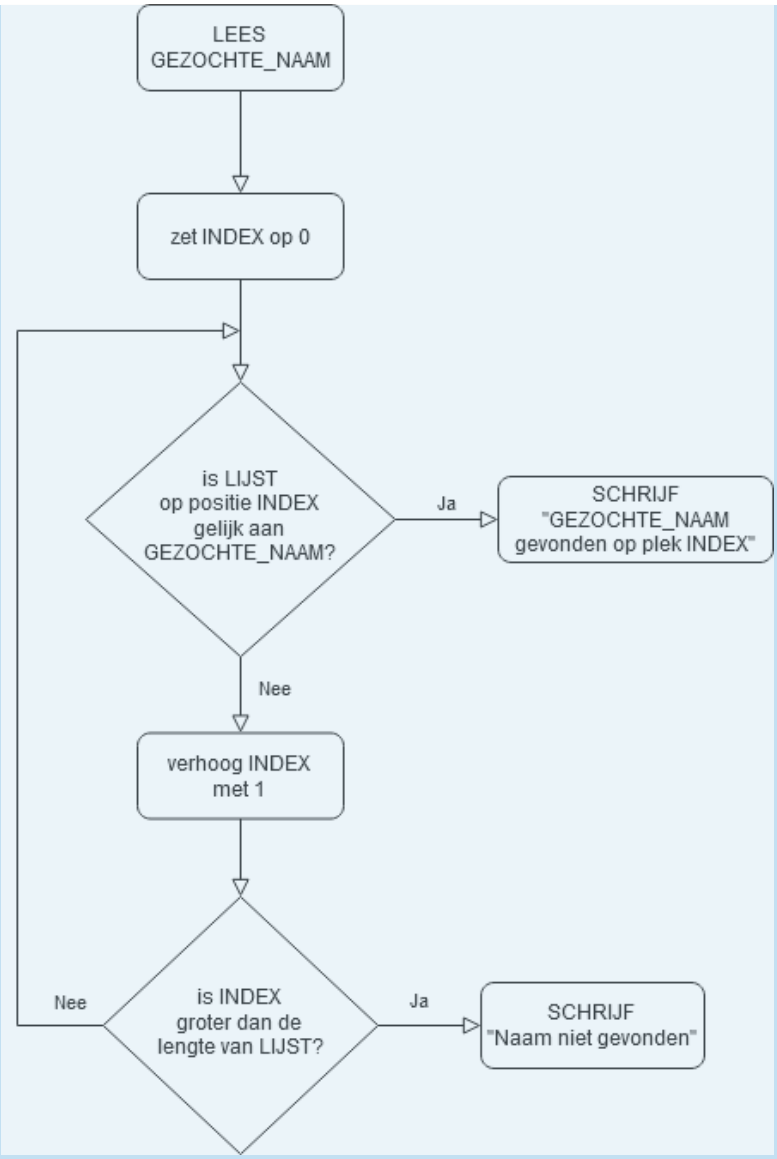
Het is een goede oefening om een algoritme, zoals het beschreven is in een flowchart (of pseudocode), zelf te implementeren in Python. Laten we een iets andere toepassing nemen: het zoeken van een naam in een namenlijst. Hiervoor kunnen we Linear Search ook gebruiken, met een kleine aanpassing:

Bij het cijfers raden gingen we ervan uit dat er altijd een oplossing gevonden zou worden. Immers (tenzij je vals speelt) zit het te raden getal altijd tussen de 1 en de 100, dus als je ze allemaal langsgaat, kom je vanzelf het goede antwoord tegen. In het geval van de namenlijst is er een vaste lijst met namen en kun je een zoekterm opgeven. Het kan natuurlijk zijn dat de naam niet voorkomt in de lijst. Hier moeten we bij het algoritme wel rekening mee houden.

Opdracht 2.5

Implementeer Linear Search om een namenlijst te doorzoeken.

Hier vind je alvast een opzet voor het python programma. Je moet dit zelf afmaken door linear search te implementeren die alle elementen uit de lijst naloopt om te kijken of de gezochte naam voorkomt in de lijst. Kijk goed naar de flowchart hieronder om te kijken hoe het algoritme er uitziet.



Hints:
- Je zult merken dat de implementatie in Python niet 1 op 1 de flowchart zal volgen. Dat is niet erg, als het idee achter het algoritme maar geïmplementeerd is.

- Je hebt een loop nodig die moet worden afgebroken zodra je de naam gevonden hebt. Hiervoor is het handig om het python commando `break` *te gebruiken.*

Je voelt al wel aan dat Linear Search niet zo'n heel efficiënt algoritme is, zeker als de lijst die doorzocht moet worden erg lang is. Over de efficiëntie van algoritmen kun je een aantal uitspraken doen:

- Best-case scenario
- Worst-case scenario
- Average case scenario
- Orde van complexiteit

2.3.2 BEST-CASE EN WORST-CASE SCENARIO

Het best-case scenario van een algoritme beschrijft simpelweg wat het minimale aantal stappen is waarin een algoritme zijn doel kan bereiken. Voor Linear Search is het best-case scenario 1. Als het te raden getal toevallig 1 is, hebben we het binnen 1 stap geraden, omdat dat de eerste "gok" is van het algoritme.

Het worst-case scenario van Linear Search is 100. Als het te raden getal toevallig 100 is, dan heeft Linear Search ook 100 stappen nodig om het te raden.

We hebben tot nu toe steeds getallen geraden tussen 1 en 100, maar het algoritme werkt natuurlijk ook als we een getal tussen bijvoorbeeld de 1 en de 1000 willen raden *(of zelfs tussen de 34 en de 1298 als je dat zou willen, dan moet het algoritme wel bij het laagste mogelijke getal beginnen en dus niet met 1).* In dat geval is het worst-case scenario van Linear Search gelijk aan 1000.

Omdat het worst-case scenario afhankelijk is van het aantal mogelijkheden, willen we het worst-case scenario het liefst in algemene termen beschrijven. We zeggen dus dat het worst-case scenario van Linear Search gelijk is aan N, waarbij N gelijk is aan het aantal mogelijkheden.

Opdracht 2.6

Wat is het best-case en worst-case scenario voor Linear Search als het raadspelletje gaat op getallen tussen de 34 en de 1298?

2.3.3 AVERAGE-CASE SCENARIO

Het best- en worst-case scenario zijn interessant, maar een betere indicatie van de kwaliteit van een algoritme is het average-case scenario. Dat beschrijft hoe het algoritme presteert als je het heel vaak zou uitvoeren op willekeurige invoer.

In het geval van Linear Search voor getallen tussen 1 en 100 is het average-case scenario 50. Gemiddeld genomen zul je 50 keer moeten "raden" met het linear-search algoritme om het juiste antwoord te vinden.

Als we dit weer wat algemener zouden beschrijven, zeggen we dat het average-case scenario van Linear Search gelijk is aan N/2. Dus gemiddeld genomen moeten we de helft van het aantal mogelijkheden proberen voordat we het antwoord gevonden hebben.

2.3.4 ORDE VAN COMPLEXITEIT

Hoewel worst- best- en met name average-case scenario's interessante eigenschappen van algoritmen zijn, wordt in de Informatica meestal een andere notatie gebruikt om de efficiëntie van een algoritme aan te duiden: de zogenaamde Orde van Complexiteit. Deze wordt weergeven in de zogenaamde "Grote O"-notatie. Meestal beschrijft deze het worst-case scenario, maar ook de andere scenario's kunnen met deze notatie beschreven worden.

De orde van complexiteit (worst-case) van Linear Search is: *O(N)*

Dit geeft aan dat het aantal stappen die Linear Search moet uitvoeren linear verband houdt met de grootte van N (het aantal mogelijkheden/elementen in de invoer). Dat wil simpelweg zeggen

dat als het aantal mogelijkheden (N) bijvoorbeeld 2x zo groot wordt, dat Linear Search er gemiddeld genomen ook 2x zo lang voor nodig heeft om het goede antwoord te vinden.

Dat klinkt als vanzelfsprekend, maar er zijn veel algoritmes die efficiënter zijn of juist minder efficiënt. Dat hangt helemaal van het algoritme en van het op te lossen probleem af.

Andere veel voorkomende Complexiteiten zijn bijvoorbeeld:

- $O(N^2)$ Kwadratisch: een stuk minder efficiënt dan lineair
- $O(\log N)$ Logaritmisch: een stuk efficiënter dan linear

We zullen van beiden nog voorbeelden zien verderop. We beginnen met Binary Search dat een complexiteit van $O(\log N)$ heeft.

2.3.5 BINARY SEARCH

We maakten al kennis met Binary Search als efficiënt zoekalgoritme. Je hebt het algoritme al handmatig uitgeprobeerd op het raadspelletje. Straks gaan we natuurlijk ook nog een implementatie van Binary Search maken in python.

De Orde van complexiteit van Binary Search is $O(\log N)$ en dat is een heel stuk efficiënter dan de $O(N)$ van Linear Search. Maar hoeveel efficiënter is dat eigenlijk? Even een stappenvoorbeeld om dat te illustreren. Voor het gemak spelen we even het raadspelletje voor cijfers tussen de 1 en 10.

Bij Linear Search neemt met elke stap van het algoritme het aantal mogelijkheden met 1 af. Stel het te raden getal is 8. Dan verloopt Linear Search als volgt:

Geraden getal	Overgebleven cijfers	Aantal mogelijkheden
(start)	1 2 3 4 5 6 7 8 9 10	10
1	~~1~~ 2 3 4 5 6 7 8 9 10	9
2	~~1 2~~ 3 4 5 6 7 8 9 10	8
3	~~1 2 3~~ 4 5 6 7 8 9 10	7
4	~~1 2 3 4~~ 5 6 7 8 9 10	6
5	~~1 2 3 4 5~~ 6 7 8 9 10	5
6	~~1 2 3 4 5 6~~ 7 8 9 10	4
7	~~1 2 3 4 5 6 7~~ 8 9 10	3
8	~~1 2 3 4 5 6 7~~ 8 ~~9 10~~	(klaar)

Elke keer valt er 1 cijfer af, totdat het goede cijfer gevonden is.

Bij Binary Search gaat het een stuk sneller. Elke keer valt de helft (!) van de overgebleven getallen af:

Geraden getal	Overgebleven cijfers	Aantal mogelijkheden
(start)	1 2 3 4 5 6 7 8 9 10	10
5	~~1 2 3 4 5~~ 6 7 8 9 10	5
7	~~1 2 3 4 5 6 7~~ 8 9 10	3
9	~~1 2 3 4 5 6 7~~ 8 ~~9 10~~	1
8	~~1 2 3 4 5 6 7~~ 8 ~~9 10~~	(klaar)

Dit gaat vooral veel verschil maken in grotere lijsten. Immers als het aantal mogelijkheden

100.000 is, is na 1 stap van Binary Search het aantal nog 50.000 en na 2 stappen nog 25.000. Na 3 stappen (12.500 mogelijkheden over) heb je dus al 87,5% van de opties uitgesloten!

In contrast: na 3 stappen Linear Search heb je 3 mogelijkheden uitgesloten, met nog 99.997 mogelijkheden te gaan (99,997%). Dat gaat nog wel even duren…

Een andere manier om de efficiëntie van Binary Search te zien is: "Als het aantal mogelijkheden verdubbelt, is er bij Binary Search 1 extra stap nodig om het antwoord te vinden". Dus het doorzoeken van een lijst met 100.000 elementen kost maar 1 stap meer dan het doorzoeken van een lijst met 50.000 elementen. Een lijst met 10.000.000 elementen kost maar 1 stap meer dan een lijst met 5.000.000 elementen, etc.

Hoe zit het dan met die *O(log N)*? Dit is eigenlijk een formele uitdrukking van wat hierboven staat. Het aantal stappen dat nodig is, is logaritmisch aan het aantal mogelijkheden (N). In de informatica gebruiken we standaard het grondtal 2 voor de logaritmen, dus log N is eigenlijk: $\log_2 N$

Voorbeeld: Als het aantal mogelijkheden 100 is, dan is het aantal stappen dat nodig is voor het vinden van het juiste element (in het worst-case scenario) gelijk aan

$\log_2 100 \approx 6.643$

Omdat er geen halve stappen mogelijk zijn, moeten we naar boven afronden en komt het aantal stappen op 7. *(Eerder noemden we al dat het raadspel in maximaal 7 stappen te "winnen" is.)*

Opdracht 2.7

Hierboven staat dat bij Binary Search het aantal te nemen stappen 1 hoger wordt als het aantal mogelijkheden verdubbelt.

a. Reken $\log_2 200$ uit om te checken of het inderdaad 8 stappen is voor 200 mogelijkheden
b. Hoeveel stappen zijn er nodig voor 10.000.000 elementen?
c. Hoeveel stappen zijn er nodig voor googol elementen (10^{100}, dus een 1 met 100 nullen)?
d. Wat is het maximale aantal mogelijkheden dat je kunt raden met 10 stappen?

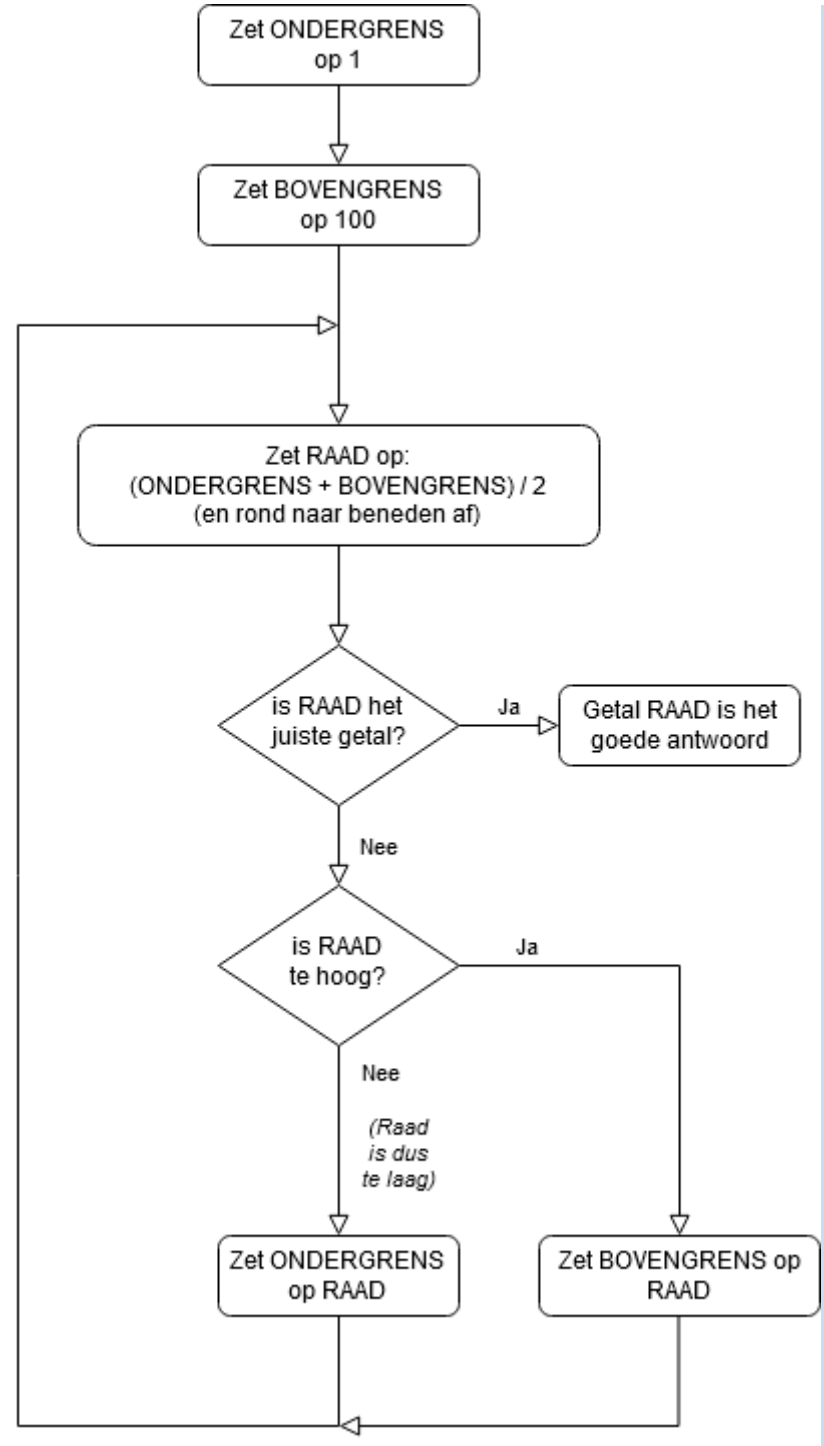
Ok, tijd om de rollen om te draaien en de computer te laten raden!

Opdracht 2.8

Maak het raadspelletje, maar dan andersom. Je neemt een getal in gedachten en laat de computer raden. Bij elk antwoord geef je aan of het te hoog of te laag is (of precies goed).

Zorg natuurlijk dat de computer het Binary Search algoritme gebruikt om zo efficiënt mogelijk te raden. Als je hebt getest of het goed werkt voor getallen tussen de 1 en de 100, probeer het dan ook eens voor grotere getallen (tussen 1 en 100.000 bijvoorbeeld), zodat je kunt zien hoe snel grote getallen geraden worden.

Het Binary Search algoritme is iets lastiger te implementeren dan Linear Search. Met name met het bijhouden van de bovengrens en ondergrens van de overgebleven getallen moet je even goed opletten. Kijk goed naar de flowchart die het algoritme schematisch weergeeft:



- In python kun je handig naar beneden afronden bij een deling door // te gebruiken i.p.v. / dus:
7 / 2 = 3.5
7 // 2 = 3

- Het is leuk om onderweg bij te houden hoe vaak er geraden is, zodat je dat na afloop van een ronde kunt afdrukken.

2.3.6 DE KEUZE VAN EEN ALGORITME

Nu we in de vorige paragrafen hebben gekeken naar Binary Search en Linear Search lijkt "Welk algoritme is het beste?" een hele domme vraag. Binary Search veegt immers de vloer aan met Linear Search qua efficiëntie, dus je kunt je afvragen of Linear Search überhaupt ooit gebruikt wordt. *(Behalve voor het pesten van informaticaleerlingen die per se dingen over algoritmen moeten leren natuurlijk…)*

We hebben voor het gemak tot nu toe 1 heel belangrijk ding niet expliciet vermeld (maar misschien was het je al opgevallen): Binary search werkt alleen voor gesorteerde lijsten!!

Het hele idee van het algoritme is juist dat je steeds slim het middelste element kiest en dan kijkt of je te hoog of te laag in de lijst zit en zo steeds de helft van de lijst kunt schrappen. Dat is leuk als je een getal tussen de 1 en de 100 wilt raden of zoekt in een telefoonboek, maar als je een lange lijst met namen hebt die niet gesorteerd is, is binary search onbruikbaar!

Oei! Dan is de keuze toch nog niet zo makkelijk…

Je hebt eigenlijk 2 opties bij een ongesorteerde lijst:

- Je gebruikt alsnog linear search
- Je sorteert de lijst eerst en dan doe je binary search

Maar dan komt het volgende probleem: Hoe sorteer je een lijst? En hoe efficiënt is dat? Weegt de tijd die het kost om de lijst te sorteren af tegen de winst die Binary Search oplevert?

Er zijn veel verschillende Sorteer-algoritmes (verderop in dit boekje bekijken we er een paar) met weer hun eigen eigenschappen en voor- en nadelen.

Voor het geval van een ongesorteerde lijst is de beste oplossing afhankelijk van je precieze toepassing:

- Als je de lijst maar 1x wilt doorzoeken, is linear search de beste optie
- Als je dezelfde lijst vaker wilt doorzoeken, dan is sorteren gevolgd door binary search de beste optie

Opdracht 2.9

Leg uit waarom dat zo is voor de 2 toepassingen

De afweging welk algoritme het beste is, is dus per geval verschillend en vereist een afweging van verschillende factoren. Daarom is het voor informatici belangrijk algoritmen te bestuderen en kennis te hebben van een "repertoire" aan standaard algoritmen.

2.4 SORTEERALGORITMES

Het sorteren van een lijst met waarden is een "probleem" dat in heel veel programma's en toepassingen een rol speelt. Denk bijvoorbeeld aan een kaartspelletje waarin kaarten gesorteerd op het scherm moeten worden getekend, of een database waar in de query een "ORDER BY" staat en er dus een gesorteerd antwoord moet komen.

Omdat het zo'n standaard probleem is, is het uitgebreid bestudeerd en zijn en een flink aantal algoritmen ontwikkeld. Sommige algoritmen zijn logisch en goed te begrijpen, maar wat minder efficiënt. Anderen zijn complexer, maar ook sneller.

Een hele mooie website om een aantal sorteeralgoritmen te bestuderen is deze:
https://www.toptal.com/developers/sorting-algorithms

Speel maar eens met de knoppen op de site. Er zijn 8 verschillende algoritmen die je kunt "loslaten" op 4 verschillende datasets. Met de knop "Play All" linksboven gaan alle algoritmes tegelijk aan de slag. Je ziet dan dat elk algoritme uiteindelijk een gesorteerde lijst oplevert, maar sommigen zijn een stuk sneller klaar dan anderen. Ook maakt de dataset erg veel uit bij sommige algoritmes.

Opdracht 2.10

Bestudeer op de website de "Selection sort". Probeer te doorgronden wat er gebeurt.

Op de volgende pagina staat een flowchart van Selection sort. Je ziet dat dit al best een complexe flowchart wordt.

Beschrijf in je eigen woorden hoe Selection Sort werkt.

Opdracht 2.11

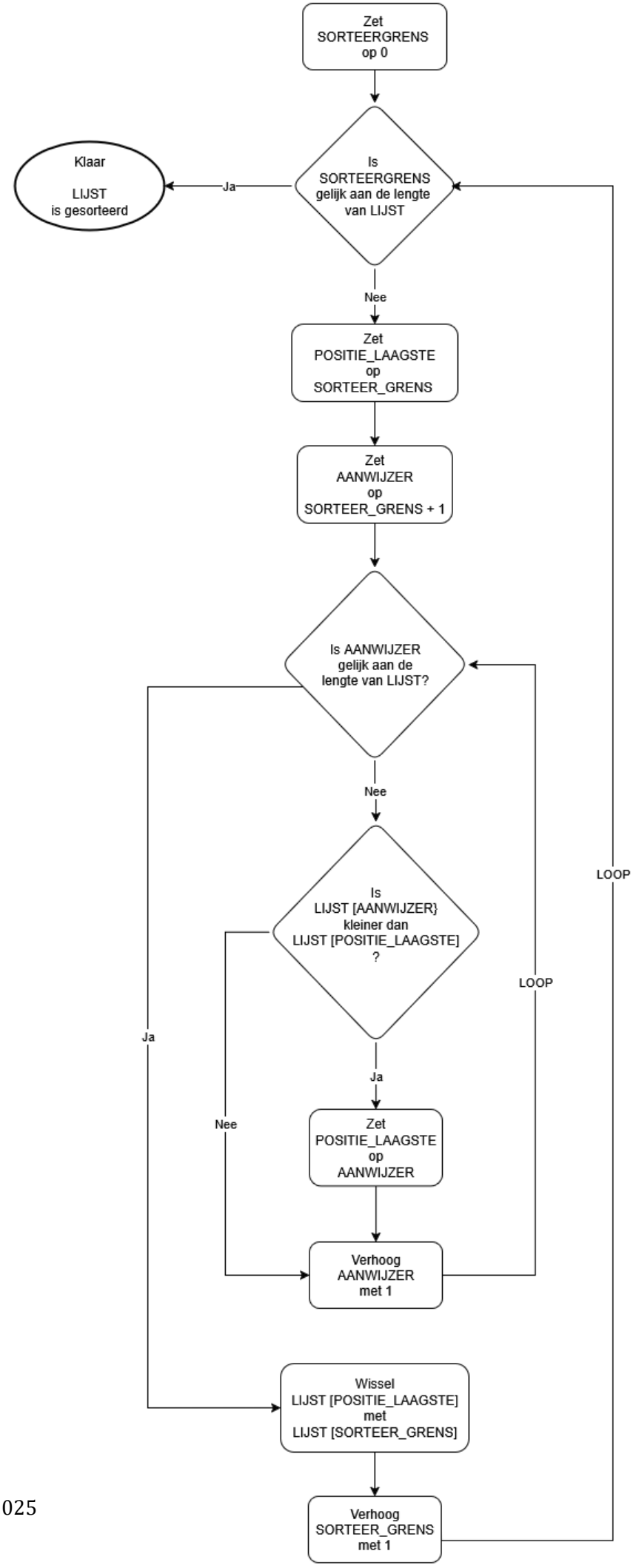
Selection sort is niet heel efficiënt (en wordt in de praktijk alleen als voorbeeld gebruikt voor leerlingen die algoritmes moeten bestuderen… 🤡)

Omdat het algoritme 2 loops bevat die "in" elkaar zitten (dat noem je een geneste loop), zijn er aardig wat "rondjes" nodig voordat alle items uit de lijst gesorteerd zijn. Selection sort is dan ook van complexiteit $O(N^2)$. Dat betekent zoveel als: "Als de te sorteren lijst 1 item groter wordt, doet het algoritme er twee keer zo lang over".

Kun je (in woorden) beredeneren waarom dit algoritme twee keer zo lang doet over het sorteren van (bijvoorbeeld) 8 items als over het sorteren van 7 items?

Opdracht 2.12 (bonusopdracht)

Bestudeer de werking van "Bubble Sort" op de website.
Maak nu zelf een flowchart van Bubble Sort.


Zet SORTEERGRENS op 0
Is SORTEERGRENS gelijk aan de lengte van LIJST
Ja
Klaar
LIJST is gesorteerd
Nee
Zet POSITIE_LAAGSTE op SORTEER_GRENS
Zet AANWIJZER op SORTEER_GRENS + 1
Is AANWIJZER gelijk aan de lengte van LIJST?
Nee
Ja
Is LIJST [AANWIJZER} kleiner dan LIJST [POSITIE_LAAGSTE] ?
Ja
Nee
Zet POSITIE_LAAGSTE op AANWIJZER
Verhoog AANWIJZER met 1
LOOP
LOOP
Wissel LIJST [POSITIE_LAAGSTE] met LIJST [SORTEER_GRENS]
Verhoog SORTEER_GRENS met 1

3 OBJECT ORIENTED PROGRAMMING (OOP)

3.1 CLASSES

In OOP draait alles om classes (klassen) en objects/instances (objecten of instanties). Bekijk het volgende filmpje met een voorbeeld en uitleg over de basics van Python classes:

Thenewboston Tutorial 29: Classes and Objects

Hierin leer je:

- Basis class syntax (schrijfwijze)
- Class variables
- Class functions (+self)
- Instanties (objects) maken
- Het maken van meerdere instanties (objects) van een klasse en het gebruik ervan

Opdracht 3.1:

Bouw de `Enemy()` class na uit het filmpje. Breid de `Enemy()` klasse uit met een nieuwe functie `heal()`. Deze functie verhoogt de life van de enemy met 2. Test je functie door een Enemy object te maken, een paar keer de `heal()` en de `attack()` functies ervan uit te voeren en vervolgens met de `checkLife()` functie de kijken of het goed heeft gewerkt.

Opdracht 3.2:

Wijzig de methode `attack()` zodanig dat elke attack een random getal tussen 0 en 3 kiest en dat van de `life` variabele afhaalt (schade is dus niet meer altijd 1, maar kan variëren). Schrijf vervolgens een programma dat twee objecten van het type `Enemy` maakt en ze om en om elkaar laat aanvallen (gebruik een loop). Als 1 van de 2 dood is, eindigt je programma met de melding wie er gewonnen heeft en hoeveel life de winnaar nog over heeft.

NB: Hier een refresher over de werking van random getallen in python

3.2 INIT EN METHODEN MET PARAMETERS

Bij het maken van een nieuw object is het vaak handig als meteen een stukje code wordt uitgevoerd dat de startwaarden van het object goed instelt. Hiertoe dient de speciale __init__() methode. Methodes kunnen natuurlijk (net als gewone functies) meerdere parameters hebben. Bekijk het volgende filmpje:

Thenewboston Tutorial 30: Init

Hierin leer je:

- Werking van de speciale __init__() functie bij classes
- (extra) parameters naast *self* voor functies

Opdracht 3.3:

Breid je `Enemy()` class van de vorige opdracht (opdracht 3.2) uit met een `__init__()` functie. Deze functie vraagt (naast de standaard benodigde `self`) om 3 parameters: een naam (als string), het aantal levenspunten (int) en de maximale schade die geïncasseerd kan worden per aanval (int). Sla deze waarden op in variabelen van de class.

Pas nu je `attack()` functie aan zodat de maximum schade die gedaan kan worden gelijk is aan het opgeslagen maximum uit de `init()` methode. De schade is dus een random getal tussen 0 en het maximum. Laat bij elke keer dat de `attack()` functie wordt aangeroepen ook op het scherm zien wat de schade was en hoeveel life de `Enemy` nog heeft. Druk daarbij ook steeds de opgeslagen naam van de Enemy af, dan kun je de 2 enemies makkelijker uit elkaar houden in de output.

Test je code door weer in een loop 2 instanties van `Enemy` met elkaar te laten vechten tot er een dood is. Experimenteer met verschillende waarden voor de init van de beide `Enemy` objecten om te zien hoe dit het gevecht beïnvloedt.

3.3 STANDAARD WAARDEN VOOR FUNCTIEPARAMETERS

(Geen filmpje bij dit stukje stof helaas ☹. Wt wil ze zelf gaan maken, maar vooralsnog geen tijd gehad…)

Zoals je weet heeft een functie parameters. Dat zijn de waarden die je de functie "meegeeft" als je hem gebruikt (aanroept). Dit geldt zowel voor functies die onderdeel zijn van een class als voor gewone functies. Voor de duidelijkheid een simpel voorbeeld:

```
def tel_op(a,b):
    return a+b
```

In het voorbeeld zijn a en b de parameters. Een aanroep van de functie zou nu kunnen zijn:

```
tel_op(10,86)
```

En als antwoord zal de waarde 96 worden teruggegeven. In de aanroep hierboven is dus voor a de waarde 10 en voor b de waarde 86 opgegeven. De functie gebruikt de opgegeven waarden in zijn code.

Soms kan het handig zijn om voor een functie een of meerdere standaardwaarden op te geven voor de parameters. Hiermee geef je de aanroeper van de functie de mogelijkheid om enkele parameters "leeg" te laten. Hiervoor wordt dan de standaardwaarde gebruikt. Een voorbeeld maakt dit duidelijk:

```
def tel_op(a, b=20):
    return a+b
```

In dit voorbeeld is in de functiedefinitie de waarde 20 als standaardwaarde opgegeven voor parameter b. Als we nu het volgende doen:

```
tel_op(10,86)
```

Krijgen we nog steeds hetzelfde antwoord: 96. Maar als we de waarde voor b weglaten als volgt:

```
tel_op(10)
```

Krijgen we als antwoord 30. Python verwacht 2 parameters, maar omdat er maar 1 gegeven is, gebruikt hij deze als waarde voor a en kiest hij de standaardwaarde (20) voor b. Simpel toch?

Let op:
Als een functie zowel parameters *met* standaardwaarde en parameters *zonder* standaardwaarde heeft, moeten die met standaardwaarde altijd achteraan. Dus:

(1) `def voorbeeld(a=10,b,c=20)` is fout.

(2) `def voorbeeld(b,a=10,c=20)` is goed.

Als je er even over nadenkt, dan is dat logisch. Stel je hebt de (foute) functiedefinitie van (1) en je doet de volgende aanroep:

```
voorbeeld(13,14)
```

Bedoel je nu dat `a` gelijk moet zijn aan 13, `b` aan 14 en `c` aan zijn standaardwaarde 20? Of bedoel je dat `a` gelijk is aan zijn standaardwaarde 10, `b` gelijk aan 13 en `c` gelijk aan 14? Dit is onduidelijk en dus niet toegestaan.

Als je de definitie van (2) hebt, dan is de aanroep wel duidelijk:

```
voorbeeld(13,14)
```

Dit betekent nu dat `b` gelijk is aan 13, `a` gelijk aan 14 en `c` gelijk aan zijn standaardwaarde 20

Opdracht 3.4:

In opdracht 3.3 heb je een `__init__()` functie toegevoegd aan je `Enemy()` class. Ook een `__init__()` functie kan standaardwaarden hebben.
Pas je `__init__()` functie uit opdracht 3 aan, zodat de parameters voor het aantal levenspunten om mee te beginnen en de maximale schade die geïncasseerd kan worden per aanval als standaardwaarde respectievelijk 20 en 3 krijgen. Het is nog wel altijd verplicht een naam op te geven.

Test je nieuwe `Enemy()` object door verschillende enemies te maken, waarbij je soms wel en soms geen waarden voor de `__init__()` parameters opgeeft.

3.4 CLASS VARIABLES EN INSTANCE VARIABLES

Er zijn verschillende termen voor variabelen die in een class gebruikt worden:

- Class variables: Variabelen die in de klasse definitie worden gezet. De waarde hiervan is dus hetzelfde voor alle objecten (instanties) van deze klasse.
- Instance variables: Variabelen die gemaakt worden in de __init__() methode van de klasse. Deze verschillen van waarde bij verschillende objecten (instanties) van deze klasse. (Vandaar ook de naam, instance variabelen hebben voor elke instance een unieke waarde)

Niet super lastig, maar belangrijk om deze termen te kennen en uit elkaar te kunnen houden. Bekijk het volgende filmpje voor een korte uitleg met een voorbeeld: Thenewboston tutorial 31

Opdracht 3.5:

Bekijk nog even je code van opdracht 3.4. Welke class variables en welke instance variables heb je gebruikt voor de Enemy class?

3.5 OVERERVING

Je hebt al gezien dat klassen een handige manier zijn om eenmalig een definitie van iets te schrijven (bijvoorbeeld de `Enemy` class uit de vorige opgaven) en meerdere objecten te maken die deze definitie gebruiken. Daar kunnen we nog een schepje bovenop doen met het principe van **overerving** (inheritance in het Engels). Bekijk het volgende filmpje:

Thenewboston tutorial 32: Inheritance

Hierin leer je:

- De basics van inheritance (een klasse alle eigenschappen van een andere klasse laten overnemen en hier dingen aan toevoegen)
- Overriding. Dit is het vervangen van een geërfde eigenschap (zoals een functie) door een andere invulling ervan.

Het voorbeeldje uit het filmpje is simpel, maar inheritance is een erg krachtig concept. Je kunt zo een hele stamboom van classes maken die steeds specifieker worden. Hierdoor heb je nergens dubbele code, maar wel veel verschillende bruikbare classes met (deels) andere eigenschappen.

Opdracht 3.6:

Ga weer verder met je code uit opdracht 3.4.

Maak een nieuwe class `Boss` en zorg dat dit een kindklasse is van de bestaande `Enemy` class. De `Boss` class erft nu dus alle variabelen en methoden van de `Enemy` class. Voeg de volgende dingen toe aan je `Boss` class:

a. Een beetje eindbaas is natuurlijk ontzettend *evil*. Voeg een nieuwe methode `evil_laugh()` toe die "Muhahahahah!!" op het scherm afdrukt.

b. De Boss class heeft een speciale eigenschap (het is tenslotte een eindbaas!) die hem moeilijker te verslaan maakt. Voeg een class variable met de naam `shield` toe aan de `Boss`. De waarde van `shield` is 3.

c. Override de `attack()` methode die geërfd is van `Enemy`.
Deze moet in de basis hetzelfde doen als de originele `attack()` methode van `Enemy`, maar met de volgende toevoeging:
Als de `Boss` geraakt wordt en zijn `shield` variabele is groter dan 0, dan heeft hij 50% kans om de aanval af te weren en dus 0 schade te krijgen. Als dat gebeurt wordt zijn `shield` variabele met 1 verlaagd. (Zo kan hij dit maar 3x in totaal doen). Er wordt ook een melding afgedrukt dat de aanval geblokt is.

Test je Boss door weer een gevecht te organiseren tussen een Boss en een Enemy., zoals je bij opdracht 3 gedaan hebt. Probeer ook eens een Boss tegen een Boss.

3.6 METHODEN VAN JE PARENT AANPASSEN

(Geen filmpje bij dit stukje stof helaas ☹. Wt wil ze zelf gaan maken, maar vooralsnog geen tijd gehad…)

Hoewel het fijn is dat je methoden die je geërfd hebt kunt vervangen, gooi je daarmee soms ook veel nuttige code weg. Je schrijft namelijk de hele methode opnieuw, terwijl je soms gewoon een kleine toevoeging wilt doen aan de geërfde methode.

Om dit te bereiken kun je de code van een methode van je parent weer oproepen als je als kindklasse de methode overschrijft.

Bekijk het volgende voorbeeld:

```
class Persoon:

    def __init__(self, voor, achter):
        self.voornaam = voor
        self.achternaam = achter

    def wie_ben_ik(self):
        print("Mijn naam is", self.voornaam, self.achternaam)

class Medewerker(Persoon):

    def __init__(self, voor, achter, stafnr):
        Persoon.__init__(self, voor, achter)
        self.stafnummer = stafnr

    def wie_ben_ik(self):
        Persoon.wie_ben_ik(self)
        Print("en mijn stafnummer is:", self.stafnummer)

x = Persoon("Frits", "Worschtenbroodt")
y = Medewerker("Hendrik", "Appelentaerte", 104)

x.wie_ben_ik()
y.wie_ben_ik()
```

Uitvoer:

```
Mijn naam is Frits Worschtenbroodt
Mijn naam is Hendrik Appelentaerte
en mijn stafnummer is: 104
```

In de code zie je 2 klassen: `Persoon` en `Medewerker`. `Medewerker` is een kindklasse van `Persoon` en erft zowel de `__init__()` methode als de `wie_ben_ik()` methode van `Persoon`.

Beide methoden worden in `Medewerker` overschreven door een nieuwe versie, maar op een andere manier dan tot nu: De code uit de originele methode van de parent wordt hierbij steeds eerst aangeroepen (`Persoon.__init__(self, voor, achter)` en

`Persoon.wie_ben_ik(self)`) en vervolgens worden er nog dingen aan toegevoegd. De schrijfwijze hiervan lijkt op het gewoon aanroepen van een methode met 2 verschillen:

- Je noemt de naam van de parent-klasse (in dit geval `Persoon`) in plaats van de naam van de instantie
- De `self` parameter moet ook worden opgegeven

Bekijk de code goed en zorg dat je goed begrijpt hoe het aanroepen van een methode van de parent werkt.

Opdracht 3.7:

Ga verder met je code uit opgave 3.6.

a:
In opgave 6 heb je de `Boss` class toegevoegd als subclass van `Enemy`. Hierbij is de `attack()` methode van de `Boss` een override van die van de `Enemy`. Hierbij heb je veel code herhaald (de `Boss` doet hetzelfde als de `Enemy`, maar heeft een kans op het gebruik van zijn shield) Verbeter je code door bij deze `attack()` methode alle dubbele code bij de `Boss` versie weg te halen en deze te vervangen door een aanroep van de attack van zijn parent (de `Enemy`). Zorg wel dat de `Boss` zijn shield kan blijven gebruiken natuurlijk.

b:
De Boss class erft de `__init__()` method van `Enemy()`. We willen nu ook de waarde van de shield variabele van de `Boss` kunnen instellen bij het maken van een `Boss` object. Override de `__init__()` methode die de `Boss` van `Enemy` heeft gekregen. Roep in de nieuwe `__init__()` van `Boss` de originele `__init__()` van de `Enemy` aan en voeg het instellen van de `shield` variabele toe.

3.7 DE KEUZE VAN EEN PROGRAMMEERPARADIGMA

Als je een programma gaat schrijven heb je de keuze om je code op een bepaalde manier op te bouwen. Je kunt ervoor kiezen om Object Georiënteerd te werken of juist Imperatief (zoals we vorig jaar bij het leren van Python gedaan hebben). Zo'n aanpak heet ook wel een Paradigma. Naast Object Georiënteerd en Imperatief, zijn er nog meer paradigma's. (Functioneel Programmeren is daarvan de bekendste. We gaan er hier niet verder op in, maar Wikipedia kan je er meer over vertellen.)

Het kiezen van een geschikt paradigma kan van een heleboel factoren afhankelijk zijn, zoals:

- Bekendheid/ervaring met het paradigma
- De noodzaak om samen te werken
- De uitgebreidheid/complexiteit van het programma
    - OO programma's laten zich makkelijker in delen testen (je kunt elke class afzonderlijk testen)
    - OO vergt wat meer planning vooraf, maar maakt de structuur wel een stuk overzichtelijk
- …

Kort door de bocht zou je kunnen zeggen:

- Wil je snel iets maken of een idee testen dat daarna niet of nauwelijks meer aangepast hoeft te worden en ben je de enige programmeur, dan ligt imperatief voor de hand.
- Is het een complexer programma, moet er worden samengewerkt, moet de code goed onderhoudbaar en aanpasbaar zijn in de toekomst, dan loont het om je code volgens Object Georiënteerde principes op te bouwen.

4 PO: TEXT-BASED RPG

Komt eraan…

5 LINKS EN VERVOLGSUGGESTIES

Nog geen genoeg van Python? Er valt nog oneindig veel te leren. Zowel over mogelijkheden van de taal Python zelf als over programmeren en algoritmiek in het algemeen. Een aantal suggesties voor verdere studie (geen toets stof uiteraard)

- The Hitchhikers guide to Python – Onofficiele handleiding over alles wat met Python te maken heeft. Van installatie tot goede programmeerstijlen
- Thenewboston YouTube tutorials – Een aantal filmpjes stonden gelinkt in dit boekje. De hele playlist is nuttig
- Learn Python the hard way – Goed praktisch leerboek over Python