FACULTEIT WETENSCHAPPEN

FACULTEIT INGENIEURSWETENSCHAPPEN EN ARCHITECTUUR

NASCHOLING ALGORITMEN EN PROGRAMMEREN

Module 1 Basiskennis programmeren in Python

Didactische wenken

Leen Brouns

UNIVERSITEIT GENT

IPVW ACADEMIE VOOR LEVENSLANG LEREN

Inhoudstafel

Didactische wenken

Disclaimer

Deze nota's bevatten didactische wenken voor de leerkracht, en zijn niet bedoeld als leermateriaal om in deze vorm voor te schotelen aan de leerlingen. Het is aan de leerkracht om er die zaken uit te halen die best aansluiten bij de les en de klasgroep van het moment.

1 Motivatie

1.1 Welke motivatie zit er achter de minimumdoelen?

De specifieke minimumdoelen voor Informaticawetenschappen zijn opgedeeld in verschillende onderwerpen (zie https://onderwijsdoelen.be/). De nascholingenreeks van UGent waartoe deze documentatie behoort, spitst zich toe op het onderwerp 'Algoritmen en programmeren', waarin gesteld wordt dat 'de leerlingen zelf ontworpen oplossingen voor concrete problemen programmeren'. De onderliggende (kennis)elementen zijn

- Algoritmen en datastructuren
- Algoritmische technieken
- Numerieke methodes
- Gebruik van softwarebibliotheken
- Gestructureerde programmeertaal
- Invoer van en uitvoer naar externe gegevensbronnen

Enkele begrippen die aanvullend opgesomd worden, ter concretisering:

- Algoritmische technieken zoals recursie, brute-force, gulzig (greedy), verdeel-en-heers en dynamisch programmeren.
- Datastructuren zoals lijst, stapel (stack), wachtrij (queue), boom, graaf, hashtabel.
- Externe gegevensbronnen zoals bestanden, databank, website, cloud.

Deze lijst geeft dan wel een idee van het te bewandelen pad, de vraag blijft: waar lopen we naartoe? En waarom willen we daar zijn?

Dit leidt ons naar een beschouwing over de maatschappelijke evoluties van de laatste jaren. Informatietechnologie kende een grote doorbraak, maar wordt momenteel vooral gebruikt om problemen op te lossen die gecreëerd werden door de aanbieders (Facebook en TikTok werden niet gemist door de voorgaande generaties; streamingsdiensten verdrongen de bestaande oplossingen; de gsm vervangt al lang het horloge,...). Bij deze toepassingen zit de gebruiker dus vooral aan de passieve kant van het creatieproces.

Van zodra de bestaande technologie (computers, apps op smartphones, Raspberry Pi / Arduino,...) echter creatief kan ingezet worden om eigen problemen op te lossen, worden er twee vliegen in één klap geslagen:

- Er kunnen veel grotere problemen opgelost worden, in minder tijd (de computer kan sneller veel meer input verwerken dan een mens).
- Het combineren van verschillende tools (rekenbladen, zelfgeschreven programma's, gedownloade bestanden,...) vergroot de creativiteit, het denkvermogen en de grip op de wereld die ons omringt.

We gaan dus de uitdaging aan om leerlingen tools aan te reiken om een actieve rol te spelen in het creatieproces.

Daarbij ambiëren we een manier van denken, waarbij problemen zo benaderd en herwerkt worden dat ze kunnen opgelost worden met de hulp van een computer. We zouden het ook kunnen formuleren als 'leer de computer jouw problemen op te lossen'. Het landschap dat passeert vóór dat doel bereikt wordt, is minstens even vormend. De volgende technieken worden toegepast bij het analyseren en oplossen van problemen (en allicht herken je technieken die niet enkel toepasbaar zijn in de informaticawetenschappen):

- Patroonherkenning en veralgemening (herkennen dat een probleem gelijkenissen vertoont met een eerder opgelost probleem)
- Decompositie (een probleem doordacht opsplitsen in deelproblemen)
- Abstractie (overbodige informatie negeren om het redeneren vlotter te laten verlopen zonder de ballast van irrelevante details)
- Modelleren (de juiste modellen gebruiken om op te redeneren, bijv. een boomstructuur bij een keuzeprobleem)
- Algoritmisch denken (een opeenvolging instructies uitschrijven die een probleem oplossen, en inzien dat de onderdelen en de volgorde essentieel zijn om het gewenste resultaat te leveren)

1.2 Welke motivatie geef ik de leerlingen mee?

De belangrijkste motivatie die aan leerlingen kan voorgelegd: vanaf nu kan ook de computer helpen om problemen op te lossen, en moet het niet meer allemaal 'te voet' of 'handmatig'.

Dit vraagt uiteraard een concretere invulling. Die concrete invulling kan samen met de leerlingen gezocht worden in de eerste les. Schets wat er mogelijk is op het einde van het jaar (bestanden inlezen, berekeningen, maxima zoeken, sorteren, teksten manipuleren,...) en vraag welk probleem ze hiermee zouden willen aanpakken.

Een paar voorbeelden: lijstjes van Spotify (of een andere streamingsdienst) van de hele klas bij elkaar leggen, bekijken welke lijstjes de grootste overlap hebben, welke artiest meest voorkomt, wat de gemiddelde duurtijd is van een track,... Maar ook: de computer gebruiken met extra hardware om metingen te doen in (de tuin van) de school, om dan concrete voorstellen te doen om de schoolomgeving aan te passen (veiliger / milieuvriendelijker / energievriendelijker te maken).

2 Wanneer welke leerstof?

Afhankelijk van de tijd die gespendeerd kan worden, de diepgang die beoogd wordt en het soort oefeningen dat aansluit bij de leefwereld van de klas, verandert de volgorde waarin leerstof aangeboden kan worden. Een mogelijkheid:

1. motivatie
2. variabelen en types (int, float, str, bool)
3. if
4. while
5. for (zowel met indices als foreach)
6. tekst (strings overlopen, indexeren, lengte, slicing, operatoren, constanten; nog geen methodes)
7. functies (gebruiken en schrijven)
8. methodes om tekst te bewerken
9. list

De volgorde die hier wordt voorgesteld heeft (zoals alles) voor- en nadelen. De list komt op het einde. Er werd voorrang gegeven aan het leren gebruiken en schrijven van functies, omdat dit toelaat grotere oefeningen te maken.

Zonder functies wordt de structuur van een programma al snel te zwaar (geneste lussen, duplicated code,...). Om ook oefeningen toe te laten die niet wiskundig geïnspireerd zijn, wordt het type voor tekst (string) al vrij snel gegeven. Dit is niet zo'n makkelijk onderwerp, maar er zijn twee voordelen:

- manipulatie van tekst spreekt iets meer tot de verbeelding dan manipulatie van getallen, en
- het verwerken van strings aan de hand van methodes laat nadien toe om de analoge bewerkingen op andere collecties (zoals de list) sneller te behandelen.

Andere volgordes zijn zeker verdedigbaar.

Naast de inhoudelijke oplijsting van constrolestructuren (if, for, while) en datastructuren (int, float, str, bool, list, eventueel set en/of dictionary) is het ook nuttig om een lijstje te maken van de vuistregels waaraan een efficiënte programmeur zich houdt. Dit kan in de les expliciet en stelselmatig aangevuld worden.

1. Eerst nadenken, dan programmeren. Nadenken gebeurt zonder computer.
2. Een programma is een leesbare tekst over de oplossing van een probleem (die bovendien kan uitgevoerd worden op een computer). Kies dus voor 'proper programmeren' (goed gekozen namen voor de variabelen, de juiste keuze- en herhalingsstructuren, spaties voor de leesbaarheid,...).
3. Oefening baart kunst: hoe meer je experimenteert, hoe beter je programmeert en problemen kan oplossen.
4. Test code vaak en grondig.
5. Voeg een volgend deel pas toe als het vorige grondig getest is en geen fouten meer bevat.
6. Voorzie commentaar, dat komt de leesbaarheid ten goede.
7. Een functie doet één ding.

3 Terminologie

3.1 Editeeromgeving, debuggen, implementeren, indenteren,...

Elk vakgebied gebruikt een eigen, specifieke woordenschat. Dat vakjargon zal ongetwijfeld ook in jullie les opduiken. Leerlingen pikken dit al doende op. Maar soms vragen begrippen wat extra verduidelijking – vooral als ze voorkomen in de vraagstelling van een toets. Hieronder worden een aantal begrippen opgesomd met hun verklaring. Gebruik dit *niet* als lijstje om uit het hoofd te (laten) leren, enkel als naslagwerk voor jezelf. Op het einde volgt nog een screenshot van de editeeromgeving Thonny, met visuele aanduiding van enkele begrippen.

Programmeertaal (bijv. Python, Java, C++,...)
: taal die nog leesbaar is door mensen, maar toch ondubbelzinnig genoeg is om door de computer op één manier begrepen te worden; wordt gebruikt om programma's te schrijven die dan door de computer uitgevoerd kunnen worden

IDE (bijv. Visual Studio Code, Pycharm,...)
: Integrated Development Environment, editeeromgeving; een programma (= stuk software) dat toelaat om vlot code te schrijven, aan te passen, te verbeteren,...; bevat o.a. een editor, een debugger en mogelijks een command line interface en een tutor

Editor
: tekstverwerkingsprogramma; een programma dat helpt bij het schrijven van tekst; dat kan gaan om tekst in een natuurlijke taal of een programmeertaal; in dat laatste geval geeft de editor dikwijls specifieke kleuren aan specifieke onderdelen van de code; hiernaast de editor uit Thonny

oef_0.py *

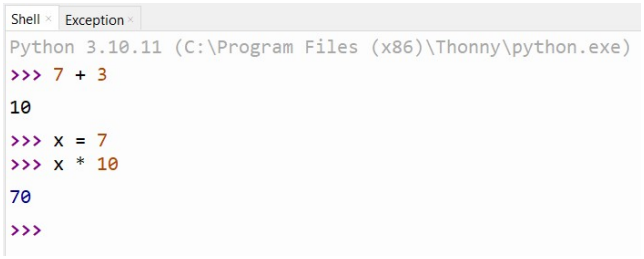
```
1 import math
2 print("Hello world!")
3 print( 10 * math.pi)
```

Debugger

programma dat helpt om fouten in code te vinden tijdens het uitvoeren van het programma; zo zijn de knoppen om te debuggen in Thonny bovenaan het menu te vinden:

Command line interface (CLI)

een stuk software dat toelaat om te interageren met de computer, waarbij de gebruiker (of user) enkel commando's (=code) intikt, en na elke regel reactie krijgt van de computer (de computer interpreteert de code onmiddellijk); hiernaast de CLI van Thonny

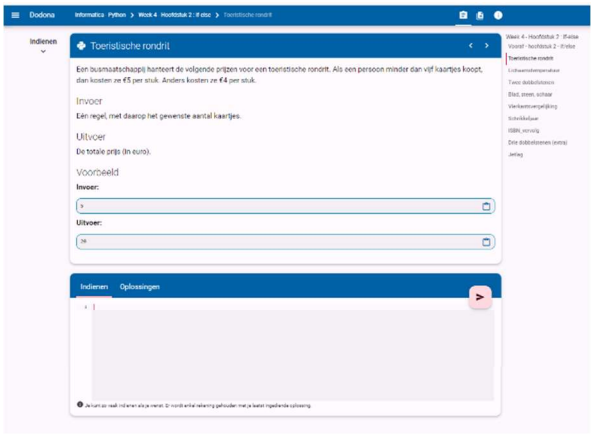


Tutor (bijv. pythontutor.com of onderdeel van Dodona-omgeving)

een stuk software dat visualiseert wat er tijdens de loop (het runnen) van een programma in het geheugen gebeurt; dit helpt om logische fouten op te sporen in een programma (net zoals een debugger, maar op een iets toegankelijker manier)

Online judge (bijv. Dodona)

online tool die zegt of een programma correct werkt; die tool kan dat uiteraard enkel doen als die zelf weet wat de bedoeling is van het programma, daarom zal de online tool zelf een lijst opdrachten aanbieden die dan opgelost en voorgelegd kunnen worden



Syntaxfout

een fout tegen de grammaticaregels van de programmeertaal (tikfouten, namen gebruikt die nog niet gekend zijn in het programma,...); deze fouten worden door de IDE (meer bepaald de editor) opgemerkt nog voor het programma uitgevoerd wordt

Logische fout

een fout tegen de logica (een verkeerde redenering die ervoor zorgt dat de berekende output niet dat is wat er eigenlijk gevraagd of beloofd werd); deze fouten worden tijdens het uitvoeren van het programma (*at runtime*) opgemerkt

Coderen

code schrijven (heel algemeen)

Implementeren

de code schrijven die ervoor zorgt dat een bepaalde functie (bijv. de functie `geef_vertaling(woord)`) doet wat ze belooft te doen (bijv. de vertaling van het gegeven woord teruggeven als resultaat van de functieoproep)

Compileren

omzetten van de volledige programmacode (geschreven in een programmeertaal, leesbaar door mensen) naar machinecode (geschreven in machinetaal, leesbaar door de computer), waarna het programma uitgevoerd kan worden; gebeurt o.a. bij de programmeertalen C, C++ en Rust

Interpreteren
: programmacode (geschreven in een programmeertaal, leesbaar door mensen) tijdens het uitvoeren regel per regel omzetten naar machinecode zodat die regel onmiddellijk kan uitgevoerd worden; gebeurt o.a. bij de programmeertalen Python en JavaScript
Zie ook https://www.freecodecamp.org/news/compiled-versus-interpreted-languages/ en https://www.baeldung.com/java-compiled-interpreted .

Indenteren
: code net iets meer naar rechts schrijven dan de code op de vorige regel (doorgaans vier spaties)

Hoofding van een functie
: de regel code die de functie definieert; dat houdt in dat de naam, de eventuele parameterlijst en het type van het resultaat vastgelegd worden

Implementatie van een functie
: de code die na de hoofding komt, en die wordt uitgevoerd als de functie opgeroepen wordt; die code zorgt er dus voor dat er gebeurt wat beloofd werd in de hoofding (daarbij is de naam van de functie normaliter een goede aanduiding – en anders hoort er documentatie voorzien te zijn voor wie de functie wil gebruiken)

3.2 Variabele, geheugenplaats en waarde

Een variabele is een verwijzing naar een geheugenplaats, waar een waarde (een getal, tekst, waar/vals-waarde of een combinatie van al deze) in bewaard wordt. Willen we dit hands-on voorstellen, dan is de variabele `leeftijd` het etiket dat op de geheugenplaats (het doosje) plakt waarin het getal 16 zit.
Als we dit voorstellen op papier, wordt de geheugenplaats voorgesteld door een kader; het etiket staat er boven (of ernaast, met een pijl naar de kader); de waarde wordt in de kader geschreven.

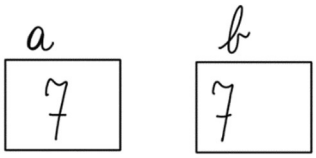
De code die hierbij hoort, initialiseert de variabele leeftijd met de waarde 16 aan de hand van het teken =, wat we de toekenningsoperator noemen.

```
leeftijd = 16
```

Deze code wordt gelezen als *'leeftijd krijgt de waarde 16'*.
Nu staat er rechts van de toekenningsoperator een waarde (in dit geval een getal). Het is ook mogelijk dat er de naam van een andere variabele staat. Dat is het geval in de tweede regel van onderstaande code.

```
a = 7
b = a
```

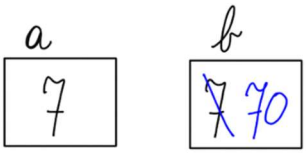
De eerste regel wordt opnieuw gelezen als *'a krijgt de waarde 7'*.
De tweede regel wordt gelezen als *'b krijgt de waarde **van** a'*. Omdat a de waarde 7 bevat, komt dit erop neer dat ook de variabele b de waarde 7 krijgt.



Willen we nu de variabele b toch een andere waarde geven, dan kan dit bijvoorbeeld met de code

```
b = 70
```

De oude waarde van b (7) wordt overschreven.



Laat ons terugkeren naar de variabele `leeftijd`, en stel dat de persoon met deze leeftijd vandaag jarig is. Dan zal de leeftijd een nieuwe waarde krijgen. Dat kan met de code

```
leeftijd = leeftijd + 1
```

wat kan gelezen worden als *'de leeftijd krijgt de waarde van zijn oorspronkelijke waarde, vermeerderd met 1*'.
Eerst wordt dus de rechteroperand (de uitdrukking rechts van de toekenningsoperator) geëvalueerd, en die wordt dan toegekend aan de variabele die links van de toekenningsoperator staat (de linkeroperand).

3.3 Statement versus expression

Zoek de definities op van een *statement* (opdracht) en een *expression* (uitdrukking).
Zijn onderstaande codefragmenten statements (opdrachten) of expressions (uitdrukkingen)? Code die nog niet gezien werd in de les, kan online opgezocht worden.

```
(a)     math.sin(30)
(b)     print("hallo")
(c)     discriminant
(d)     som = a + b
```

Antwoord:
Een expression (uitdrukking) is een stuk code dat een bepaalde waarde (getal, tekst, combinatie van beide) voorstelt (na eenvoudig ophalen uit het geheugen of een ingewikkelder berekening). Een expression kan gebruikt worden als rechteroperand voor de toekenningsoperator (=).
Een statement (opdracht) is een stuk code bestaande uit één of meer volledige regels die door de computer kunnen uitgevoerd worden.
Dus (a) en (c) zijn expressions, (b) en (d) zijn statements.

3.4 Les Nederlands versus les Informatica

Maken we een vergelijking tussen de les Nederlands en de les Informatica (partim programmeren), dan komen we op deze lijst. Merk op: het is in de informatica-les verleidelijk om enkel met de eerste twee situaties aan de slag te gaan...

Nederlands	Informatica – leren programmeren
grammatica / zinsbouw	syntax aanleren (theorieles)
eenvoudige teksten schrijven	kleinere programma's schrijven (geleide opgave)
teksten verbeteren	debuggen van eigen en andermans werk
begrijpend lezen	programma's lezen en interpreteren wat hun werking is
teksten structureren	programma's opsplitsen in logische eenheden (a.d.h.v. structuren en functies)
opstel schrijven	grotere programma's schrijven (open opgave)

In een theorieles komt er dikwijls aan bod *hoe* tekst geschreven moet worden (de grammatica of syntax van de taal), maar niet *wat* er moet geschreven worden. Vooraleer de leerkracht Nederlands de leerlingen kan vragen om een opstel te schrijven, zullen ze eerst heel wat voorbeelden van teksten gezien moeten hebben, om zich een beeld te kunnen vormen *wat* er verteld kan worden met de aangeboden syntax van het Nederlands.
Bij deze een pleidooi om alle mogelijkheden uit het lijstje aan bod te laten komen – en het lijstje misschien nog aan te vullen?

4 Zelfredzaamheid

Als er één vakgebied is waarvoor het internet veel informatie oplevert, dan is het wel informatica. De ontwikkelaars van het wereldwijde web bedienen er zichzelf gretig van. Online zoekwerk vult dan ook het gros van de werkdag van een informaticus: de technologie evolueert zó snel dat een oude cursus raadplegen geen optie is. Dan is de vraag: leren we beginnende programmeurs enkel online zoeken, of bieden we toch klassiek leermateriaal? Voor het middelbaar onderwijs is een stevige basis aan de hand van klassiek leermateriaal aangewezen. Te snel online zoeken houdt het risico in dat code van bedenkelijke kwaliteit of vergezochte oplossingen de goed gestructureerde opbouw van de cursus doorkruisen. Van zodra leerlingen zich iets comfortabeler voelen met de leerstof en zelf op onderzoek uit willen (bijv. om nieuwe stringfuncties te leren kennen), kan er gericht aan zoekskills gewerkt worden.
Een paar mogelijkheden.

(a) Bied de leerlingen een spiekbrief aan, het analogon van een formularium in de les fysica. Hier valt de syntax van de gekozen programmeertaal af te lezen uit eenvoudige voorbeeldprogramma's. Voorts staan er functies opgelijst die de leerlingen uit de kast kunnen trekken om hun programma daar te krijgen waar ze het willen hebben.
(b) Leer leerlingen foutmeldingen lezen. Laat hen de foutmeldingen eerst vertalen naar het Nederlands en met elkaar overleggen wat dit kan betekenen. Opzoeken op het internet kan in een later stadium ook – maar begeleid dit.
(c) Leer leerlingen testcases neerschrijven: niet teveel (geen tien gelijkaardige situaties) en niet te weinig (geen speciale gevallen vergeten).
(d) Wijs leerlingen er expliciet op dat programmeren voor 70 procent uit voorbereidend denkwerk bestaat dat bij voorkeur op papier gebeurt. Als dat goed ging, dan zal de tijd nodig voor coderen en debuggen beperkt zijn.

(e) Debuggen verloopt voor beginnende leerlingen best handmatig en heel bewust door het programmaverloop na te bootsen op papier. Concreet: teken geheugenplaatsen, zet er de juiste namen boven, vul hun waarden in, pas de waarden aan volgens wat de code regel per regel opdraagt. Als leerkracht kan je eerst zelf experimenteren met een tutor: een eenvoudige debugtool die visueel weergeeft wat er at runtime in het geheugen gebeurt. (Een losstaande tutor is te vinden op https://pythontutor.com/ ; o.a. Thonny https://thonny.org/ en Dodona https://dodona.be/ hebben hun eigen tutor.) Introduceer dat pas later in de lessenreeks; het risico bestaat anders dat leerlingen niet zelf diep genoeg in de interpretatie van de code duiken.

(f) Ga zelf tijdens lesvoorbereidingen op zoek naar achtergrondinformatie over bepaalde topics (strings, lijsten, bepaalde functies,...). Combineer hierbij de goede zoektermen: context (hier *Python*), onderwerp (bijv. *lengte van tekst*) en eventueel de soort vindplaats (bijv. *tutorial* om wat meer achtergrond te krijgen). Goed om weten: w3schools (https://www.w3schools.com/python/) biedt de mogelijkheid om voorbeeldcode te runnen, eventueel aan te passen, en opnieuw te runnen (gebruik de grote groene knoppen). Je vindt er allicht voorbeeldjes voor het theoriegedeelte van de les. Let wel, enkele topics die niet vereist zijn voor de minimumdoelen (en ook niet in het beschikbare tijdsbestek passen) staan vrij vooraan in deze tutorial (tuples, lambda's). De site stackoverflow is dan weer voor gevorderden: eerst staat de foute code van de vraagsteller; scrol verder naar beneden voor mogelijke antwoorden. Deze site is niet geschikt voor leerlingen, en allicht enkel van nut als je de oplossing zoekt op een concreet probleem dat zich stelt bij het uitproberen van gevorderde concepten.

5 Gestructureerd programmeren

Programmeren gebeurt 'gestructureerd' als een programma enkel gebruik maakt van drie welbepaalde structurele opdrachten om het verloop van het programma vorm te geven. Het typevoorbeeld van een niet-gestructureerde opdracht is het `goto`-statement dat toelaat om vanop een bepaalde plaats naar een willekeurige andere plaats in de code te springen. Dat statement is ondertussen uit de meer recente programmeertalen verdwenen (je vindt het dus niet meer in Python), maar ook `break` en `exit` zijn opdrachten die bij ondoordacht gebruik de structuur van een programma kunnen overhoop halen, en eigenlijk niet nodig zijn.
Het is immers bewezen dat, om eender welk programma (een omvormer van gegeven input naar gewenste output) te maken, volgende drie structuren voldoende zijn: de sequentiestructuur (gewone opvolging), de selectiestructuur (mogelijkheid om een keuze te maken), en de herhalingsstructuur (meerdere keren dezelfde stappen uitvoeren).

Didactische tip: laat de woorden `goto`, `break` en `exit` niet vallen in de les. Grijp wel in als je ze ziet passeren in een oplossing van leerlingen.

Om code overzichtelijk te houden, zal er ook veelvuldig gebruik gemaakt worden van functies of procedures: een geheel van opdrachten (enkele regels code) worden apart verpakt en kan uitgevoerd worden door elders in het programma met één opdracht het hele stuk code op te roepen.

Hieronder worden de drie structuren overlopen. De nadruk ligt hier op de valkuilen en de onnodig ingewikkelde constructies die beginnende programmeurs soms maken. Dat hoort bij het normale leerproces en het is nuttig dat leerlingen eerst hun eigen fouten maken. Maar de fouten mogen niet de tijd en de kans krijgen om ingesleten te raken. Hoe leesbaarder en korter de structuur van een programma, hoe makkelijker om door het bos de bomen te blijven zien. En om fut over te houden voor grotere projecten.

5.1 Sequentiestructuur

De sequentiestructuur is allicht de eenvoudigste structuur, maar voor beginners vaak de verrassendste. Code wordt uitgevoerd van boven naar beneden. Wat in de wiskundeles wel kan, kan in de informaticales niet.

Zo kan er in de wiskundeles geredeneerd worden 'Als x = 7 en y = x + 1, dan is y = 8', maar evengoed 'Als y = x + 1 en x = 7, dan is y = 8.' Wat bovenaan of onderaan het blad of het stelsel staat, doet er in principe niet toe.
In de informaticales doet de volgorde van de opdrachten er wél toe. Zo zullen deze regels code fouten opleveren:

```
y = x + 1  # NameError: name 'x' is not defined
x = 8
print(y)
```

terwijl deze code wel perfect kan:

```
x = 8
y = x + 1
print(y)
```

Merk op dat het al heel wat helpt als het =-teken in de informaticales wordt gelezen als *'krijgt de waarde van'*.

5.2 Selectiestructuur

De selectiestructuur laat toe om bepaalde opdrachten enkel uit te voeren als er aan een (of meerdere) voorwaarde(n) voldaan is. Code wordt dus *conditioneel* (afhankelijk van bepaalde condities) uitgevoerd.
Die condities of voorwaarden zijn vergelijkingen of uitspraken die tot een ja/nee-antwoord leiden. Er zal dus intensief gebruik gemaakt worden van booleaanse uitdrukkingen (logische uitdrukkingen, die de waarde `True` of `False` kunnen aannemen).

De syntax van de `if`-structuur kan teruggevonden worden op de spiekbrief.

Een eerste valkuil: indien er een vergelijking staat als voorwaarde, dan wordt er wel eens verkeerdelijk de toekenningsoperator (=) geschreven, in plaats van de vergelijkingsoperator (==). Gelukkig houdt de Python-interpreter deze fout tegen met volgende foutboodschap:

```
SyntaxError: invalid syntax. Maybe you meant '==' or ':=' instead of '='?
```

In veel andere programmeertalen passeert deze fout dikwijls onopgemerkt, en is het soms lang debuggen voor ze gevonden wordt.

Een tweede valkuil is code schrijven voor het geval dat *"er niets moet gebeuren"*. Dergelijke opdracht bestaat (`continue`) maar het vermelden ervan aan leerlingen is didactisch geen goed idee.
Hieronder een voorbeeld.

```
if het_regent == True:
    ?? niets doen ??
else:
    print("we gaan wandelen")
```

Hier is het `if`-gedeelte van de selectiestructuur eigenlijk leeg. Beter is te benadrukken dat als er *"niets"* moet gebeuren, er ook niets moet geprogrammeerd worden. Behandel dus alleen situaties waar er wel degelijk actie ondernomen moet worden. Dit kan in bovenstaand voorbeeld door de voorwaarde om te keren; dan is het `else`-gedeelte leeg en kan het weggelaten worden.

```
if het_regent == False:
    print("we gaan wandelen")
```

Houd de taal ook correct: laat de leerlingen nooit spreken van een '*if-lus*' want de selectiestructuur ís geen lus, ze wordt slechts één maal doorlopen, en keert nooit op haar stappen terug.

Nog een tip die te maken heeft met taalaanvoelen: expliciet vergelijken met de waarde `True` is nooit nodig. De code

```
if het_regent == True:
    print("zet je kap op")
```

kan korter genoteerd worden als

```
if het_regent:
    print("zet je kap op")
```

Ga na: de if-structuur met de expliciete vergelijking wordt in het Nederlands de omstandige zin 'als het waar is dat het regent'; terwijl de if-structuur zonder expliciete vergelijking vlotter bekt als 'als het regent'.

Ook de vergelijking met False is overbodig. Vervang de code

```
if het_regent == False:
    print("we gaan wandelen")
```

door de kortere code

```
if not het_regent:
    print("we gaan wandelen")
```

Het moge duidelijk zijn dat dit finetuning van code is. Didactisch gezien kan het een goed idee zijn om de expliciete vergelijkingen met `True` en `False` eerst toe te laten, en pas na verloop van tijd te vragen om dit te vermijden.

Tot slot nog een pleidooi om de nodige zorg te besteden aan het kiezen van een juiste naam voor elke booleaanse variabele. Indien de naam van de booleaanse variabele met een vragende intonatie wordt uitgesproken, dan zou er automatisch *'ja'* of *'neen'* als antwoord moeten komen. Bij wijze van oefening kan er gevraagd worden om een aantal namen te scoren op hun geschiktheid om als naam van een booleaanse variabele te dienen. Enkele ongeschikte voorbeelden links, en enkele geschikte voorbeelden rechts:

```
kleur           is_rood
grootte         is_groot
leeftijd        senior
weerbericht     zonnig
geslacht        mannelijk
```

De twee namen voorgesteld op de laatste regel (`geslacht` en `mannelijk`) roepen om een kanttekening. Variabelen die slechts twee waarden kunnen aannemen, zijn bijna altijd aanleiding tot een booleaanse variabele – en die geef je dan best een naam die duidelijk aangeeft hoe de waarde `True` (dan wel `False`) geïnterpreteerd moet worden. Dus in die zin zou de voorkeur gaan naar de naam `mannelijk` (of `vrouwelijk`...). Wordt er echter afgestapt van de tweedeling, dan is een stringvariabele met naam `geslacht` veel beter.

5.3 Herhalingsstructuur

De herhalingsstructuur laat toe om bepaalde opdrachten meerdere keren uit te voeren.

Als bij de start van de herhaling gekend is hoe dikwijls de herhaling moet plaatsvinden, dan wordt de `for`-structuur aangeraden. Als maar tijdens het herhalen zelf zal blijken of de herhaling mag doorgaan of moet stoppen, dan wordt de `while`-structuur aangeraden. Het is een goed idee om hier strikt op toe te zien, en wel om twee redenen:

- Het laat beginnende programmeurs toe om op basis van duidelijke criteria een keuze te maken, dat spaart keuzestress.
- Het laat lezers van afgewerkte code toe om, op basis van de `for`- dan wel `while`-sleutelwoorden, besluiten te trekken over de aard van het geïmplementeerde algoritme.

5.4 Herhalingsstructuur: while

De `while`-structuur is het moeilijkst om onder de knie te krijgen, en daarom net is het een goed idee om deze als eerste aan te leren. Anders blijven leerlingen soms hangen in datgene wat ze al beheersen, en wordt er te dikwijls een `for`-lus gebruikt waar een `while`-lus beter past bij het algoritme.

De syntax van de `while`-structuur kan teruggevonden worden op de spiekbrief. Let op de tip onderaan. De structuur van de `while`-lus kan ook samen met de leerlingen ontdekt worden aan de hand van de volgende oefening.

Voorbereiding vóór de les:
Rangschik een boek kaarten zo dat bovenop een viertal zwarte kaarten zitten, en dan pas een rode kaart.

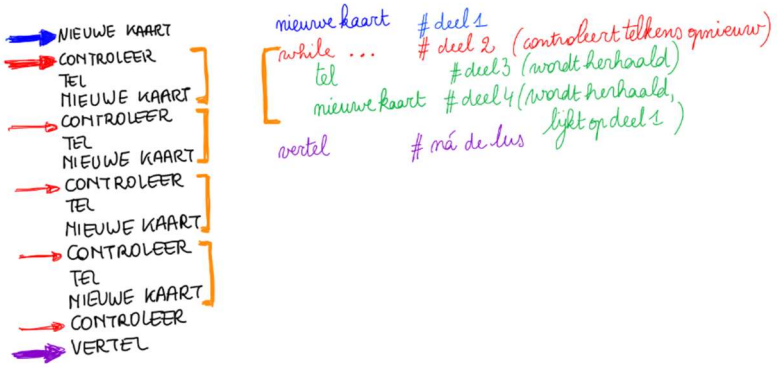
Voorbereiding voor het spel:
Vraag vier leerlingen vooraan in de klas, en hang hen elk een etiket op: NIEUWE KAART, CONTROLEER, TEL en VERTEL.
Geef de leerling met etiket GEEF het boek kaarten (gedekt).

Spelregels:
De leerling met etiket NIEUWE KAART mag enkel de bovenste kaart doorgeven aan CONTROLEER, zonder zelf te kijken.
De leerling met etiket CONTROLEER kijkt stiekem naar de kaart, en steek zijn duim omhoog als die rood is, of naar omlaag als die zwart is.
De leerling met etiket TEL zet bij elke duim omlaag (bij elke zwarte kaart) een streepje op het bord.
De leerling met etiket VERTEL mag alleen iets doen als de duim van CONTROLEER omhoog gaat: dan leest die het aantal streepjes op het bord luidop voor.

Spelverloop:
De leerling met etiket NIEUWE KAART mag starten; de andere volgen.
Ondertussen schrijft elke leerling van de klas neer wie er actie ondernomen heeft:

> NIEUWE KAART (= er wordt een kaart van de stapel doorgegeven aan CONTROLEER),
> CONTROLEER (= de kaart wordt gecontroleerd),
> TEL (= er wordt een streepje bijgezet op het bord) en
> VERTEL (= het totale aantal wordt luidop meegedeeld)

Bespreking:
Zet op bord welke woorden neergeschreven werden. Dat zouden de zwarte woorden uit onderstaand bordschema moeten zijn.



Bekijk samen de lijst woorden. Welke woorden komen meermaals voor, welke slechts een keer? Welke actie komt overeen met de pseudocode

    while kaart is zwart:

Duid elke actie aan die met deze pseudocode overeenkomt. Dat geeft telkens de start aan van eenzelfde pakketje code. Duid die pakketjes aan (zie oranje op het bordschema). Welk woord komt er vóór al deze pakketjes, welke woorden komen ná al deze pakketjes? Welk woord staat laatst in elk van de pakketjes, dus vlak voor er opnieuw gecontroleerd wordt? Hieruit kan afgeleid worden waarom een `while`-lus uit vier delen bestaat, en waarom deel 4 en deel 1 zo erg op elkaar gelijken (cfr. spiekbrief).

Van zodra de structuur van de `while`-lus ontdekt is, kan deze ingeoefend worden door strikte toepassing van volgend **stappenplan** bij het schrijven van code:

1. Zoek in de opgave hoe lang de herhaling moet duren: dat bepaalt wat in deel 2 komt. Schrijf dit alvast neer na het `while`-sleutelwoord.
2. Bepaal uit deel 2 het item waarop getest wordt.
3. Zet dat item klaar in deel 1 van de lus,
4. en zet het onmiddellijk opnieuw klaar in deel 4 van de lus (op het einde van het geïndenteerde stuk).
5. Bepaal daarna wat er ook nog herhaald moet worden, dat zal in deel 3 komen (tenzij deel 3 leeg blijft).
6. Controleer goed: opdrachten die slechts één keer moeten gebeuren, staan buiten de lus, dus na deel 4 en aan dezelfde kantlijn als het `while`-sleutelwoord.

Gaan leerlingen niet afknappen op dit 'handje vasthouden'? Dat is inderdaad goed mogelijk. Dikwijls is een antwoord (= dit stappenplan) maar nuttig als eerst de vraag gesteld werd (= 'hoe krijg ik die `while`-lus goed?'). Ga na wat voor uw leerlingen werkt: eerst zelf op onderzoek uit? Prima. Houd dan wel zelf deze **debugskills** achter de hand, toe te passen als leerlingen met vragen zitten bij de `while`-lus die ze net schreven:

1. Scan eerst de regel code die start met while: op welk item wordt getest?
2. Scan dan het einde van de lus: wordt het item daar opnieuw klaargezet (ingelezen, berekend, opgezocht)?
3. Zo nee, dan zijn er twee mogelijkheden:
   dit deel (deel 4) ontbreekt – dan zit de code zeker vast in een oneindige lus; of
   dit deel (deel 4) staat niet onderaan de lus – dan zijn er zeker en vast problemen met de start of het einde van de lus.
4. Als deel 4 wel juist staat, dan zit de fout niet in de structuur van de lus en moet ze ergens anders gezocht worden. (Misschien zit de fout in de voorwaarde zelf?)

Als de leerlingen het beu zijn (of de lestijd op is) om `while`-lussen te debuggen, dan is het tijd om bovenstaande debugskills of het stappenplan expliciet aan te leren.

5.5 Herhalingsstructuur: for

Hoewel de `while`-lus zeer algemeen inzetbaar is, zal deze vrij zwaar uitvallen als de elementen van een collectie overlopen moeten worden. Of als er op voorhand geweten is hoe dikwijls de lus doorlopen moet worden. In beide gevalen is een `for`-lus eenvoudiger van structuur.

Wie heeft leren programmeren in Java, C, C++, C#,... denkt bij de `for`-lus allicht onmiddellijk aan de structuur

```
for ( int i = 0; i < rij.size(); i ++ ){
      // doe iets met rij[i]
}
```

Hierbij is de 'motor' van de lus de teller, die een zelfgekozen start- en eindwaarde kan aannemen. In latere versies van de meeste programmeertalen kwam er dan de `foreach`-lus bij: een lus die alle elementen van een collectie overloopt, zonder beroep te doen op indexering. Hier wordt, afhankelijk van de taal, expliciet `foreach` (of `for each`) geschreven, dan wel het sleutelwoord `for` behouden.

Wie leert programmeren in Python, komt eerst de `foreach`-lus tegen (hoewel het sleutelwoord ook gewoon `for` is), en leert nadien dat deze constructie ook bruikbaar is om via indexering door een indexeerbare collectie (een `string` of een `list`) te lopen.

Concreet: een `for`-loop is een herhalingsstructuur die één voor één alle elementen van een collectie doorloopt. Een collectie is een variabele van een collectietype. En een collectietype is een gegevenstype met waarden die zijn samengesteld uit andere gegevenstypes. Zo is een string (type `str`) een collectie van karakters; een `list` is een collectie van elementen van willekeurig type; en een `range` is een opeenvolging van gehele getallen.

Voor elk van de opgesomde collectietypes kan diezelfde for-structuur geschreven worden. Het is echter aangeraden om zeer zorgvuldig om te gaan met de keuze van de naam van de lus-variabele. Hieronder enkele goede voorbeelden.

```
for letter in woord:    # woord is van type str en bestaat uit letters
    print(letter)

for getal in getallen: # getallen is van type list, elk element is een getal
    print(getal)

for naam in namen:      # namen is van type list, elk element is een naam
    print(naam)

for i in range(4):      # i overloopt de getallen 0 1 2 3
    print(i)
```

De laatste lus overloopt een opeenvolging gehele getallen, startend bij 0, stoppend net voor de bovengrens (= het meegegeven argument 4). Dit is met andere woorden de ideale manier om alle indices uit een indexeerbare collectie (een `string` of `list`) te overlopen. Let op! Een `for`-lus die alle elementen van een `range` afloopt, overloopt gehele getallen (die bovendien als teller of index fungeren). Daarom MOET de naam van de lus-variabele zo gekozen worden dat dit ook duidelijk is. Deze namen zijn dus wel toegelaten als lus-variabele:

```
i
k
tel
teller
index
```

Af te raden zijn:

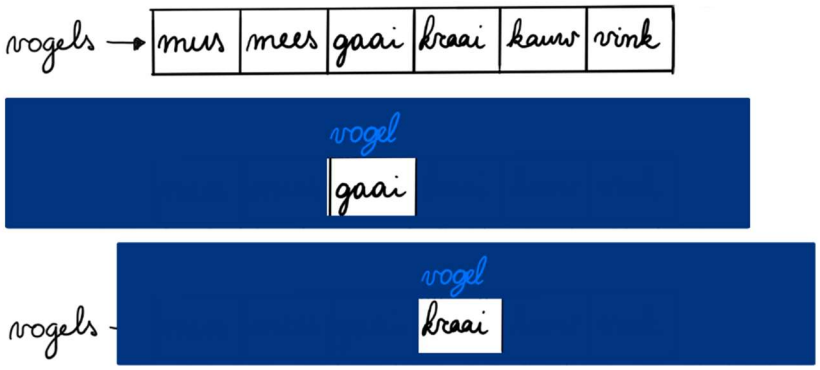
```
j           # trekt te hard op i, gevaarlijk in geneste lussen
l           # trekt te hard op het getal 1
getal       # wordt voorbehouden voor getallen die meer variëren
aantal      # een index heeft niets met een aantal te maken
wijsvinger  # is te lang (het 'aanwijzen' is er anders wel 😊)
schoenmaat  # een schoenmaat is geen teller
```

Het lijkt wat overtrokken om hier strikte regels voor op te leggen, maar het nut van deze regel komt pas naar boven als leerlingen beide manieren leren om een indexeerbare collectie (bijv. een `string` of een `list`) te overlopen.

De eerse manier werd zonet voorgesteld.

```
vogels = ["mus","mees","raaf","roek","duif"]
for vogel in vogels:
    print(vogel)
```

Deze manier om een lus te schrijven kan vergeleken worden met een 'kijkgat' in een 'schuivende afdekplaat'. De list `vogels` wordt aan het oog onttrokken; enkel dat ene element dat op dat moment in de lus aan de beurt is, is zichtbaar onder de naam `vogel`.



Hierbij moet ook opgemerkt worden dat elk element van de list wel geraadpleegd kan worden, maar niet vervangen[1] kan worden. Dus de code

```
vogels = ["mus","mees","raaf","roek","duif"]
for vogel in vogels:
    vogel = vogel + "je"    # ZINLOOS; de toekenningsoperator
                            # heeft geen effect op de list vogels
```

heeft geen effect op de list `vogels`.

[1] Merk op dat hier het woord *vervangen* staat, en niet *gewijzigd*. Het verschil is subtiel, en zou ons leiden naar de begrippen '*mutable*' en '*immutable*', maar dat is buiten de scope van dit document, en zeker buiten de scope van de lessen voor de leerlingen. Dat wordt pas interessant bij objectgericht programmeren. Hier te onthouden: als er in Python elementen in een list aangepast moeten worden, wordt de `for`-lus met indexering gebruikt in plaats van de `foreach`-lus.

Er zijn dus twee nadelen aan deze `for`-lus: tijdens het doorlopen van de lus kan er slechts één element tegelijk beschouwd worden (en diens positie is ook niet expliciet gekend), en getallen of woorden in een lijst kunnen niet veranderd worden. Voor veel toepassingen is dat echter geen bezwaar.

Dit wordt pas een bezwaar als er lijsten van getallen of woorden aangepast moeten worden; of als de positie van het element moet gekend zijn. In dit geval moet er teruggegrepen worden naar de 'oude' `for`-lus (hier bijv. in Java)

```
int[] rij = {11,55,77,99};
for ( int i = 0; i < rij.length; i ++ ){
      // doe iets met rij[i]
}
```

maar dan met de Python-syntax:

```
rij = [11,55,77,99]
for i in range( len(rij) ):  # i overloopt de getallen 0 1 .. len(rij)-1
    # doe iets met rij[i]
```

Toegepast op de lijst met vogelnamen:

```
vogels = ["mus","mees","raaf","roek","duif"]
for i in range( len(vogels) ):
    vogels[i] = vogels[i] + "je"    # ok, heeft effect op de list
```

Het is voor een beginnend programmeur geen evidentie om deze lus (met indexering) en de vorige lus (zonder indexering) uit elkaar te houden. Daarom is de juiste naamgeving essentieel. (Het wordt aan de lezer overgelaten om alle slechte combinaties van lussen uit te proberen.)

Tot slot twee voorbeelden waarbij de lus zonder indexering (de `foreach`-lus) niet gebruikt kan worden.

Voorbeeld met twee lijsten
Gegeven twee lijsten, een met namen van personen en een met hun respectievelijke lievelingskleuren.
Gevraagd: schrijf van elke persoon zijn lievelingskleur uit.

```
namen = ["Otis","Eric","Maeve","Adam"]
kleuren = ["rood","geel","zwart","blauw"]

for i in range( len(namen) ):
    print(f"{namen[i]} verkiest {kleuren[i]}")
```

De reden waarom de `foreach`-lus hier niet gebruikt kan worden is eenvoudig: een `foreach`-lus kan vergeleken worden met een 'kijkgat' in een 'schuivende afdekplaat' over één collectie, die het onmogelijk maakt om tegelijkertijd ook een kijkje te nemen in een andere collectie. Dus moet er expliciet via een plaatsaanduiding, of indexering, gewerkt worden.

Voorbeeld met twee elementen binnen dezelfde lijst
Gegeven een lijst gehele getallen. Tel hoeveel opeenvolgende koppels gelijke getallen er zijn.
Bijvoorbeeld: [10, 10, 10, 70, 70, 10, 10] bevat vier opeenvolgende koppels gelijke getallen.

```
getallen = [2,3,3,3,9,9,10,3,3]
aantal = 0
for i in range(1, len(getallen)): # range start bij 1, stopt bij len(...)-1
    if getallen[i-1] == getallen[i]:
        aantal += 1
print(f"er zijn {aantal} opeenvolgende koppels gelijke getallen")
```

Omdat in elke doorloop van de lus zicht moet zijn op *twee* elementen uit de lus, kan ook hier de `foreach`-lus niet gebruikt worden – die kan slechts één element per keer bekijken.

6 "Ik zie het niet"

Vroeg of laat komt de verzuchting *"Mevrouw, Mijnheer, ik zíe het niet".* Het helpt dan niet om te opperen *"Denk eens na",* want de impliciete vraag is *"Hóe moet ik denken?".*

We gaan na welke concrete technieken ingezet kunnen worden, maar duiden eerst een mogelijke oorzaak van de writer's block bij beginnende programmeurs.

De opgaven van een programmeeropdracht komt dikwijls in tekstvorm – in het Nederlands. De oplossing wordt verwacht in tekstvorm – bijvoorbeeld in Python. Voor kleine oefeningen kan het impliciete denkproces op taalniveau gebeuren: *"als"* wordt *"if"*, *"zolang"* wordt *"while"*,... Maar als de opdrachten groter worden, dan is dat niet meer mogelijk. Om de gedachten te vestigen bekijken we dit probleem.

> *Schrijf een programma dat de plaats van de drie grootste getallen in een lijst uitschrijft.*

Nu is redeneren op taalniveau niet meer mogelijk, maar moet eerst het algoritme bepaald worden dat de input (een lijst getallen) omvormt tot de gewenste output (drie locaties). Als dat echter niet ingeoefend werd tijdens de makkelijkere oefeningen, kan de leerling niet terugvallen op gekende technieken.

6.1 Diagrammen

Een mogelijke tussenstap om van de Nederlandse opgave naar de oplossing in Python te komen, is het opstellen van een Nassi-Shneidermandiagram of pseudocode. Een Nassi-Shneidermandiagram visualiseert de flow van het programma, en laat toe om op de tekening een keuzestructuur of herhaling "met de vinger te volgen". Geneste structuren kunnen gradueel verfijnd worden. De omvorming tot code vraagt meestal niet meer veel (mentale) inspanning.

Nadeel is wel dat het gat tussen de Nederlandse opgave en het diagram nog vrij groot is. Want als je niet inziet wélke structuren (keuze, herhaling) er nodig zijn, kan je ze ook niet gebruiken als bouwsteen van het diagram. En het is ook niet altijd duidelijk welke informatie waar bewaard wordt: de datastructuren liggen niet expliciet vast.

Er zijn twee vragen die, vóór het opstellen van een diagram, explicieter beantwoord moeten worden:

(1) Welke geheugenplaatsen zijn er nodig – zowel voor input als output.
(2) Welke manipulaties zijn er nodig (bewerkingen, toekenningen) om het uiteindelijke resultaat op de juiste plaats in het geheugen te krijgen.

Enter pen, papier, puzzelstukken en reflectie over het eigen reflecteren.

6.2 Pen en papier

Van zodra beginnende programmeurs weten wat een geheugenplaats is en welke types er zijn, kunnen ze de gegevens uit de programmeeropdracht (de input) neerschrijven. Dit moet wel concreet gemaakt worden, zodat de voeten stevig op de grond blijven. Voor het gegeven probleem (de plaats van de drie grootste getallen) wordt dit bijvoorbeeld

30	600	80	100	40	70	-50	200

Merk op: de lijst is niet te lang, en niet te kort. De getallen zijn groot genoeg om niet met indices verward te worden, en niet te moeilijk (78 / 87) om snel genoeg te kunnen redeneren. Ze staan ook in vrij willekeurige volgorde.

Daarna wordt de output van het programma neergeschreven. Dat kan – door een mens – meestal 'op zicht' gebeuren. In dit geval zijn 1, 3 en 7 de gewenste posities. Let op: er wordt geteld vanaf nul. Het getal 30 staat hier op positie (of index) 0, het getal 600 staat op positie (of index) 1. Noteer de posities best kleiner en in een andere kleur onder de vakjes van de lijst.

Merk op: dikwijls is het aangeraden om het probleem niet in al zijn facetten ineens op te lossen. Een verdedigbare vereenvoudiging is hier: "Bepaal de positie van het grootste getal in de lijst." Of, afhankelijk van de voorkennis: "Bepaal de drie grootste getallen in de lijst." Dus laten we toewerken naar de output 600, 200, 100.

De volgende stap bestaat erin om onze denksnelheid te verlagen. De tussenstappen die we als mens impliciet gemaakt hebben, moeten expliciet gemaakt worden. Waarom schreven we 600, 200 en 100 neer? Omdat 600 het grootste is. En als we die 600 niet meer beschouwen, is 200 het grootst. Dat kunnen we expliciet maken, zodat we volgende stappen krijgen:

(a) Zoek het grootste in de lijst, schrijf dat uit.
(b) Verwijder dat uit de lijst.
(c) Zoek het grootste in de lijst, schrijf dat uit.
(d) Verwijder dat uit de lijst.
(e) Zoek het grootste in de lijst, schrijf dat uit.

Deze oplossing heeft alvast de verdienste dat ze zal werken, en dat ze leesbaar is. Er zit ook een duidelijke herhaling is, dus de stap naar een lus voelt natuurlijk aan.

Er zijn echter twee bezwaren. Het eerste kom je op het spoor als je het concrete, kleine voorbeeld even los laat en weer kijkt naar de oorspronkelijke opdracht – die we vereenvoudigd hadden. Indien we terug zouden keren naar de oorspronkelijke opgave, waarbij de locatie bewaard moet worden, hebben we een probleem: elementen uit een lijst verwijderen, verandert de locaties van alle elementen die erna komen. Ten tweede is deze oplossing niet zo efficiënt. Als we de input opblazen tot belachelijk grote proporties (er zijn bijvoorbeeld een miljoen getallen; die zouden dan in een bestand kunnen staan), dan zien we dat er drie keer door de hele lijst gelopen wordt – en twee keer ongeveer een half miljoen getallen van plaats verschoven worden.

Dus moet de denkoefening opnieuw uitgevoerd worden, om deze twee bezwaren weg te werken:

A. De oplossing is onvolledig (de indices gaan verloren).
B. De oplossing is inefficiënt.

Het is voor beginnende programmeurs niet zo eenvoudig om de efficiëntie van een algoritme juist in te schtaten. Het vergt ook wat ervaring om te weten welke efficiëntere alternatieven er zijn voor inefficiënte oplossingen. Daarom kan het geen kwaad om leerlingen met inefficiënte oplossingen te laten experimenteren – om hen daarna te laten reflecteren.

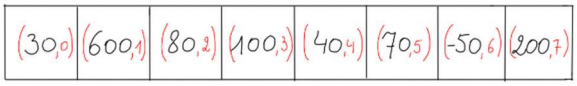
Stel dat efficiëntie nog even niet aan de orde is (bijvoorbeeld als er niet zo heel veel getallen in de lijst zitten). Dan kan het eerste bezwaar (de verloren indices) weggewerkt worden door het voorgestelde algoritme te behouden (nl. het grootste element herhaaldelijk uit de lijst verwijderen) en code toe te voegen.

Dat kan hier op twee manieren:

> ofwel wordt er extra informatie bewaard (dus meer geheugen gebruikt),
> ofwel wordt de nodige informatie berekend (dus meer rekentijd gebruikt).

Terzijde: dikwijls zal er een afweging gemaakt moeten worden tussen *'meer plaats'* of *'meer tijd'*. Maar als er zodanig hard bespaard wordt op (geheugen)plaats dat de leesbaarheid van de code eronder leidt, dan werd de verkeerde keuze gemaakt.

De eerste manier wordt gelaten als denkoefening, met bijgaande visuele tip:

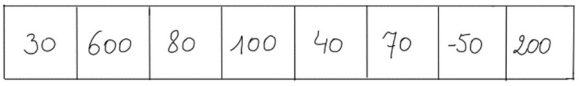


De tweede manier wordt enkel aangeraden aan mensen die écht tijd teveel hebben, want dit belooft aardig wat 'gepruts' en zal ook niet erg leesbaar zijn.

Laten we er nu van uitgaan dat we beide bezwaren willen wegwerken: de oplossing moet de juiste indices geven én efficiënt zijn.

Als er iets schort aan de efficiëntie van een algoritme,
is het dikwijls wijzer om het hele algoritme te her-denken
in plaats van code toe te voegen.

We kiezen er daarom voor om de elementen op hun plaats te laten, én de lijst niet te dikwijls te doorlopen. Het voorbeeld wordt er weer bijgenomen:

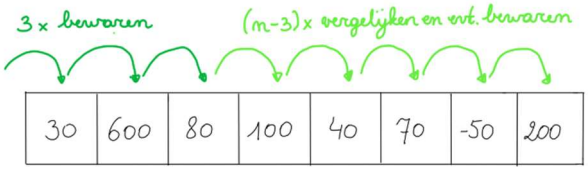


Als we voor elk element uit de lijst een beslissing kunnen nemen die herbezoek van dit element onnodig maakt, is efficiëntie meteen gegarandeerd (we lopen dan al zeker niet te dikwijls door de lijst; één keer door de lijst lopen is namelijk het minimum, het kan niet met minder). We werken in stapjes, en wanen ons computer.

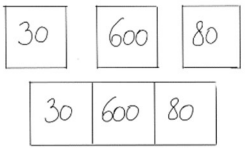
(a) De computer komt eerst het getal 30 tegen. Wat moet daarmee gebeuren? Er zijn slechts twee opties: vergeten of onthouden. We gaan voor onthouden, want misschien is dit wel het grootste uit de lijst, dus tekenen we een nieuwe geheugenplaats en kopiëren er 30 in.
(b) Zo wordt ook 600 bewaard, en 80. (Waarom? En waarom in twee nieuwe geheugenplaatsen?)

(c) Dan komt het getal 100. Moet dit bewaard worden? Zo ja, tekenen we nog een vierde geheugenplaats? Waar zal dat eindigen? Als we geen nieuwe geheugenplaats tekenen, hoe weten we dan in welk van de drie bestaande het getal 100 bewaard moet worden?
(d) Voor elk volgende getal wordt opnieuw nagegaan of en zo ja, waar het bewaard moet worden. Als er genoeg getallen de revue passeren, zullen er ook genoeg gevallen in de spotlights komen te staan.

Ga na dat de twee bezwaren van daarnet nu weggewerkt zijn.
In onderstaande afbeelding wordt de herhaling aangeduid met groene pijlen. De donkergroene pijlen duiden aan dat de eerste drie getallen zonder meer bewaard worden. De lichtgroene pijlen duiden aan dat de rest van de getallen ook een gelijkaardige behandeling krijgen, maar dan wel verschillend van de eerste drie. Indien de juiste datastructuren gekozen worden, dan kunnen de voorgestelde manipulaties dus met twee *opeenvolgende* lussen (geen *geneste* lussen) uitgevoerd worden.



Merk op dat er nog wel wat aandacht zal moeten gaan naar de datastructuur voor de hulpvariabelen (de informatie die tussentijds bewaard wordt); die zijn nog niet getekend. Twee mogelijkheden: ofwel drie losse variabelen, ofwel een lijst van lengte drie.



Beide keuzes kunnen onderzocht worden op hun gebruiksgemak.

Nu hebben we het antwoord op de twee vragen die beantwoord moesten worden:
(1) Welke geheugenplaatsen zijn er nodig – zowel voor input als output.
(2) Welke manipulaties zijn er nodig (bewerkingen, toekenningen) om het uiteindelijke resultaat op de juiste plaats in het geheugen te krijgen.

Pas als deze informatie verzameld is, kan er aan een diagram gedacht worden. Niet zeker of de vragen duidelijk genoeg beantwoord werden? Leg de werkwijze uit aan een gewillige toehoorder (mag ook een huisdier zijn), en luister zelf aandachtig naar de vlotheid waarmee dit gebeurt. Zit er systematiek in? Dan komt het goed. Teveel uitzonderingsregeltjes? Dan is schaafwerk nodig.

Merk op: bovenstaande oefening is een prima illustratie van het nut van een heap. Maar als deze oefening pas gegeven wordt als de heap-structuur aan bod komt, dan is alle fun voor de leerlingen er af. Dan wordt alle denkwerk onder de mat geveegd, en wordt 'programmeren' herleid tot slikken wat reeds gekauwd opgelepeld wordt. Daarvoor waarschuwt het citaat waar volgende paragraaf mee begint.

6.3 Stappenplan

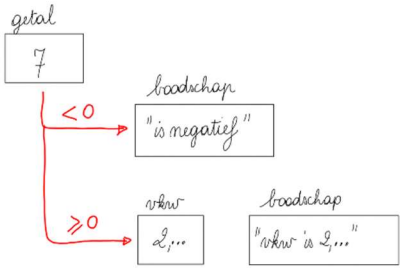
Een hoofdstuk in een programmeercursus dat een specifiek onderwerp behandelt (bijv. een while-lus) is typisch als volgt opgebouwd: eerst wordt een probleem geschetst, daarna volgt een programma dat dit probleem oplost, en tot slot worden alle onderdelen van het programma besproken. [...] Dit patroon van het introduceren van materiaal creëert - onbedoeld - de illusie dat programmeren triviaal en eenvoudig is. Het feit dat we allemaal, wanneer we een probleem aanpakken, beginnen met onvolledige en onjuiste programma's, die we dan geleidelijk aanpassen door onze implementatie uit te breiden, te verfijnen en te herstructureren totdat we bij een acceptabele oplossing uitkomen, lijkt onder de mat geveegd te worden alsof het een beschamend geheim is dat niet genoemd mag worden. Terwijl in tekstboeken en beginnerscursussen de uiteindelijke oplossing voor het probleem in detail wordt uitgelegd, wordt het proces - hoe we te werk gaan bij het ontwikkelen van de oplossing - bijna volledig verwaarloosd.

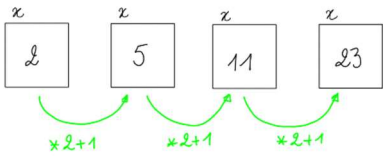
(vrij en met de hulp van DeepL vertaald uit
Caspersen and Kölling, STREAM: a first programming process, 2009)

Essentially, programming is one of the best-kept secrets of programming education!
(Sentance e.a., Computer Science Education, 2023, blz 221)

Of het nu om de wiskundeles, een fysicavraagstuk of de voorbereidingen van een feestje gaat: een stappenplan helpt dikwijls om het doel te bereiken zonder halverwege de draad kwijt te raken. Dat moet ook mogelijk zijn voor 'leren programmeren'. Hieronder een voorstel. Pas, samen met de leerlingen, aan aan de behoeften.

(1) **Lees de opgave** aandachtig. Bekijk het voorbeeld en reken het na – dat geeft soms extra informatie bij de opgave.
(2) Schrijf een **concreet voorbeeld** van input op, gebruik meteen 'geheugenplaatsen'.
(3) **Bereken** bij dit voorbeeld de output, zonder tussenstappen.
(4) Maak de **tussenstappen** concreet: hoe ben je tot de output gekomen? Doe dit meteen met de beperkingen die een computerprogramma ook kent (een 'totaaloverzicht' is niet mogelijk, variabelen worden één voor één bezocht).
    a. Gebruik **hulpvariabelen** indien nodig (dus teken nieuwe geheugenplaatsen en vul ze op).
    b. Duid met **pijlen** aan waar er een keuze gemaakt wordt (rood in de schets), of waar iets herhaaldelijk uitgevoerd wordt (groen in de schets).





(5) **Vervang** in gedachten het concrete **voorbeeld**: zijn er situaties waarin jouw methode niet werkt? Wat met negatieve getallen, wat als de input heel erg groot is,...? Speel hier gerust *advocaat van de duivel*. Als jouw methode niet werkt voor de *'specialere gevallen'* dan is ze niet algemeen genoeg. Pas de methode aan (en dat doe je in eerste instantie *niet* door met een if-structuur de speciale gevallen eruit te filteren!).
(6) Giet deze informatie in een Nassi-Shneiderman**diagram**.
(7) Zet het diagram over in **code** en test tussendoor kleinere onderdelen uit.
(8) **Test uit** met het voorbeeld uit de opgave.
(9) Test uit met **zelfgekozen voorbeelden (testcases)**; zorg ervoor dat je ook de 'speciale gevallen' uittest.

7 Werken met tekst

Bij het werken met tekst is het belangrijk dat de eerste kennismaking expliciet op papier of bord verloopt. Zet een stuk tekst op bord, toon dat deze tekst ergens in het geheugen staat (dus zet het in een kader, een geheugenplaats), en zet er dan de indices van de karaktertekens onder. Ook de alternatieve indexering, met negatieve indices, krijgt een plaats.

De code die hierbij hoort, is

```
tekst = "Tel je zegeningen."
```

Let op, het weglaten van de dubbele aanhalingstekens (dubbele quotes) zal tot foutmeldingen leiden.

Daarna kunnen alle operatoren en methodes uit het spiekbriefje uitgetest worden. Gebruik hiervoor niet per se de editor (waar volwaardige programma's geschreven worden), maar laat experimenteren in de CLI, de shell.

Een paar mogelijkheden:

```
>>> tekst = "Tel je zegeningen."
>>> tekst.replace(".","!")
'Tel je zegeningen!'
>>> tekst[-1]
'.'
>>> tekst = tekst.replace(".","?")
>>> tekst[-1]
'?'
>>> tekst[7:12]
'zegen'
>>> tekst[::-1]
'?negninegez ej leT'
>>> tekst.count("en")
2
>>> tekst.find(" ")
3
>>> tekst.isalnum()
False
```

Merk op: ga na of het gebruik van de string-methodes op dat moment al in het lesverloop past. Zolang er nog geen lijsten gezien zijn, is het gebruik van de methode `split()` bijvoorbeeld geen optie.

8 Functies

8.1 Gebruik van functies

Baseer de uitleg voor het gebruik van functies op enkele voorbeelden die al zijn voorgekomen in de lessen.

```
zin = "Tel je zegeningen."
lengte_zin = len(zin)
print("De lengte van de zin is")
print(lengte_zin)

print("De lengte van 'hallo' is")
print(len('hallo'))
```

Benoem hierbij deze begrippen:

- De functie draagt de **naam** `len`.
- De functie wordt twee keer **opgeroepen**.
- De functie krijgt één argument mee.
- De eerste keer is het **argument** de variabele `zin`.
- De tweede keer is het **argument** de string `"hallo"`.
- Het **resultaat** van de functie is blijkbaar een getal.
- Dat resultaat wordt bij de eerste functieoproep **bewaard** in de variabele `lengte_zin`.
- Het resultaat van de tweede functieoproep wordt **onmiddellijk gebruikt** in de uitschrijfopdracht.

8.2 Gebruik van een zelf verzonnen functie

Stel dat er een functie zou zijn die de plus-operator vervangt. Dat zou dan een functie zijn die van twee getallen de som berekent. Laat de leerlingen volgend programma herschrijven zonder plus-teken, maar dan wel met de functie met naam `som_van` die twee getallen als argument neemt. (Let op, we geven de functie bewust niet de naam `som`, want het is veel te verleidelijk om ook een variabele in het hoofdprogramma de naam `som` te geven. En dat geeft problemen.)

```
a = 15
b = 25
som = a + b
print(som)

print(17.5 + 22.2)
```

Dat zou volgend resultaat moeten opleveren:

```
a = 15
b = 25
som = som_van(a, b)
print(som)

print(som_van(17.5, 22.2))
```

Dan kunnen opnieuw de begrippen uit voorgaande paragraaf bespreken.

- De functie draagt de **naam** `som_van`.
- De functie wordt twee keer **opgeroepen**.
- De functie krijgt twee argumenten mee.
- De eerste keer zijn de **argumenten** de variabelen `a` en `b`.
- De tweede keer zijn de **argumenten** de getallen `17.5` en `22.2`.
- Het **resultaat** van de functie is blijkbaar een getal.
- Dat resultaat wordt bij de eerste functieoproep **bewaard** in de variabele `som`.
- Het resultaat van de tweede functieoproep wordt **onmiddellijk gebruikt** in de uitschrijfopdracht.

8.3 Functie zelf schrijven

Nu rest ons nog het schrijven van de functie. Dit bestaat uit twee delen: de hoofding (waaruit de naam van de functie en de parameterlijst blijkt), en de implementatie (de code die ervoor zorgt dat de functie het resultaat juist berekent en ook effectief teruggeeft aan de oproepende code).

De implementatie kan hier op twee manieren: met tussenstappen, of zonder tussenstappen. Zorg wel dat de naam van de functie ook hier niet hergebruikt wordt als naam voor een (lokale hulp-)variabele.

```
def som_van(x, y):
    resultaat = x + y
    return resultaat

def som_van(x, y):
    return x + y
```

Vraag nu aan de leerlingen welke namen zij gekozen hebben voor de parameters. Veel kans dat a en b gekozen werden – maar dat is niet beter of slechter dan x en y.

Voeg tot slot nog commentaar toe vóór de functiehoofding, zodat het duidelijk is wat de functie doet.

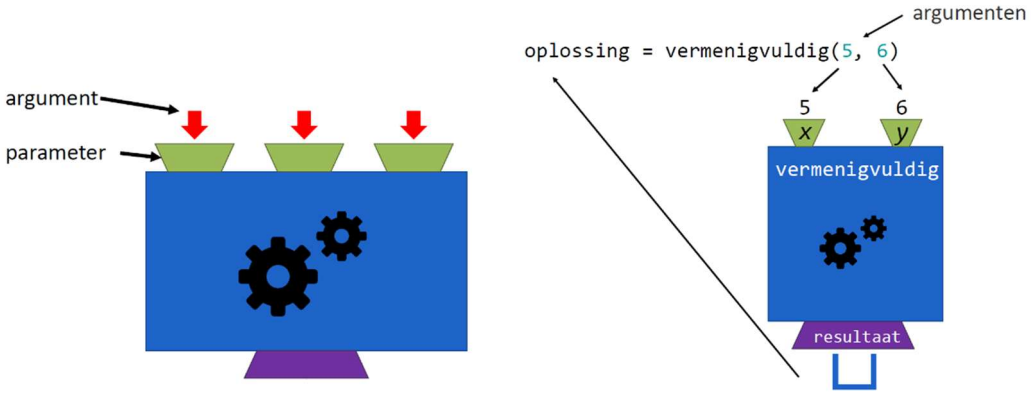
```
# bepaalt de som van de twee parameters en geeft het resultaat terug
def som_van(x, y):
    return x + y
```

8.4 Argument versus parameter

Een argument is een uitdrukking die wordt meegegeven wanneer een functie wordt opgeroepen. De waarde van dat argument zal toegekend worden aan de corresponderende parameter van de functie.

Een parameter is een variabele die binnen een functie gebruikt wordt, om te verwijzen naar de uitdrukking die als argument werd doorgegeven.

Een visuele voorstelling, uit de cursus Informatica van 1e bach Industrieel Ingenieur aan UGent:



Het belangrijkste is dat de leerlingen inzien dat de namen van de argumenten en parameters absoluut niet gelijk moeten zijn aan elkaar. Meer nog, de functie kent de namen van de argumenten uit het hoofdprogramma niet eens, en het hoofdprogramma kent de namen van de parameters uit de functie niet.

8.5 Functieoproep uitgebeeld

Benodigheden: vijf lokale of Chinese vrijwilligers om de rol op te nemen van de functie `som_van`, de kopieermachine, en drie vraagstellers die graag de som berekend zien van hun twee variabelen. Daarnaast ook twaalf geheugenplaatsen (glazen of potjes) van een verschillende kleuren (liefst twee, negen en één van dezelfde kleur). En tot slot etiketten voor de geheugenplaatsen, en inhoud voor de geheugenplaatsen. Meer uitleg en een demonstratie in de nascholing zelf.

9 Lijsten

9.1 Nut van lijsten

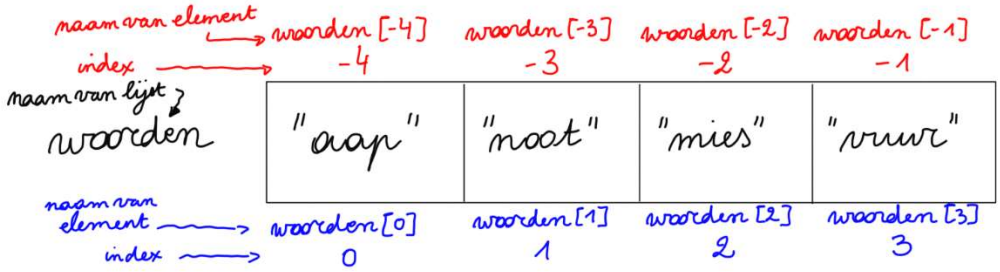
Illustreer het nut van lijsten aan de hand van volgende vraag, op te lossen zonder computer (pen en papier is een optie): de leerkracht geeft mondeling een aantal getallen, en de leerlingen wordt gevraagd om op het einde

- de som
- het gemiddelde
- de mediaan

te geven. Voor de eerste twee vragen zijn er twee, hooguit drie getallen te onthouden. Dat kan nog zonder pen en papier. Ook een computer heeft niet veel nodig: drie variabelen voor het nieuwe getal, de huidige som en het huidige aantal volstaan. Dus drie geheugenplaatsen: `getal`, `som` en `aantal`.
Voor de derde vraag moeten alle getallen onthouden worden. Waar zou een computer die getallen bewaren? In de variabelen a, b, c, d, e, f, g,... of `getal_1`, `getal_2`, `getal_3`, `getal_4`,... ? Dat is niet programmeerbaar, laat staan onderhoudbaar – want het is niet geweten hoeveel getallen er gegeven zullen worden. Daarom is het goed om een aantal getallen na elkaar te kunnen bewaren in één variabele. Dat is dan een variabele van type `list`.

9.2 Elementen in lijsten worden vanaf 0 geïndexeerd

Net zoals bij strings, heeft het eerste element in een lijst de index 0. Hieronder een lijst met vier elementen. De index van het laatste element is dus 3, één kleiner dan het aantal elementen in de lijst. (De alternatieve indexering met negatieve getallen kan ook, maar zal allicht minder gebruikt worden in oefeningen.)



9.3 Lijsten overlopen, doorzoeken en aanpassen

Lijsten kunnen doorlopen worden met een `foreach`-lus of een `for`-lus. Voor uitleg daarover, verwijzen we naar de sectie *Herhalingsstructuur: for*. En naar het spiekbriefje, onderdeel *herhaling (aantal vooraf gekend): for-lus.*

Net zoals bij strings, kan een deel van de lijst verkregen worden met *slicing*. Ook hier: zie spiekbriefje.

Voor alle functies die een lijst als argument vragen (`len`, `min`, `max`, `sum`) verwijzen we naar het spiekbriefje, onderdeel *functies die lijst-argument nemen.*

Ook alle methodes van de klasse `list` (dus methodes die op een gegeven lijst opgeroepen worden), zijn daar te vinden. Opgelet: er zijn twee soorten methodes.

- De methodes `index` en `count` **geven een resultaat** terug, net zoals de functies die we zelf leren schrijven.
- De methodes `append`, `insert`, `remove`, `sort` en `reverse` daarentegen geven geen resultaat terug. Ze **veranderen wel de lijst** waarop ze opgeroepen worden.

Belangrijk om in te zien, ook af te lezen uit het spiekbriefje: het resultaat van `index` en `count` moet bewaard of onmiddellijk gebruikt worden. De oproep van de methodes `append`, `insert`, `remove`, `sort` en `reverse` gebeurt alleen op de regel; er is geen resultaat om op te vangen of onmiddellijk te gebruiken in een test of print-opdracht.

Voor wie de vergelijking zou willen maken met string-methodes: die geven *altijd* een resultaat terug, die veranderen *nooit* iets aan de string waarop ze opgeroepen worden.

Verdere documentatie

Progra-meer.org, bundeling van workshopreeksen georganiseerd door UGent, UHasselt en KULeuven
https://progra-meer.org/

2LinK², vakvereniging voor leraren informatica en STEM
https://www.2link2.be/

Dwengo, vzw die kennis delen en samenwerking centraal stelt
https://dwengo.org/

Dodona, online judge voor programmeeroefeningen
https://dodona.be
https://dodona.be/nl/courses/?tab=featured

Youtube videoreeks UHasselt Tutorials (Dirk Peeters)
https://www.youtube.com/playlist?list=PL2iW_rkiCt7UqxL47lGkBaTgApj_QHJ0K

De programmeursleerling
https://www.spronck.net/pythonbook/dutchindex.xhtml

Wisselwerking

Graag ontvang ik feedback over vorm en inhoud van deze bundel, work in progress dd. 20 oktober 2023. Daarnaast wissel ik ook graag van gedachten met leerkrachten omtrent het vormgeven van nieuwe oefeningen, inzichten, ideeën,...

Contact opnemen kan via mail naar leen.brouns@ugent.be.