PROGRAMMEREN 2: TOPIC 3

VARIABELEN, DATASTRUCTUREN EN OPSLAG VAN DATA

UCLL HOGESCHOOL

OVERZICHT

- Variabelen
- Types
- Datastructuren
- Garbage collection
- Call by semantics
- Vergelijking
- Opdrachten

VARIABELEN

VARIABELEN: HERHALING

Hoe zou je variabelen beschrijven in één zin?

VARIABELEN: HERHALING

Hoe zou je variabelen beschrijven in één zin?

Een variabele dient om informatie in op te slaan tijdens de uitvoering van een programma.

QUIZ

Wat is de waarde van `z` na uitvoering?

```python
x = 8
y = 4 * x
z = y / 2 + 6
```

4

8

10

22

QUIZ

Wat is de waarde van y na uitvoering?

```python
x = 0
x = 5
x = x - 2
y = 2 * x
```

-4

0

5

6

QUIZ

Wat is de waarde van a na uitvoering?

```python
a = 2
b = a + 4
c = b - 1
a = c + 2
a = a - b
```

-5

1

2

7

AANPASBAARHEID VAN VARIABELEN

Als je de waarde van een variabele kan veranderen na initialisatie, dan zegt men dat de variabele **aanpasbaar** (Eng. *mutable*) is.

Het tegengestelde is een **onaanpasbare** (Eng. *immutable*) variabele.

Regels rond aanpasbaarheid verschillen sterk tussen talen.

- In Python zijn alle variabelen steeds aanpasbaar (zie bv. vorige slides)
- In Java zijn variabelen aanpasbaar, tenzij je ze declareert met het speciale woord `final`
- In Rust zijn variabelen onaanpasbaar, tenzij je ze declareert met het speciale woord `mut`

AANPASBAARHEID IN PROGRAMMEERTALEN

Java

```java
final String naam = "Sofie";  // final --> onaanpasbaar
int leeftijd = 43;  // aanpasbaar
```

JavaScript

```javascript
const naam = "Sofie";  // const --> onaanpasbaar
let leeftijd = 43;  // let --> aanpasbaar
```

Rust

```rust
let naam = "Sofie";  // onaanpasbaar
naam = "Tristan";  // ERROR
let mut leeftijd = 43;  // mut --> aanpasbaar;
leeftijd = leeftijd + 1;  // OK
```

AANPASBAARHEID: WAAROM BELANGRIJK?

Variabelen aanpassen klinkt eenvoudig, maar heeft serieuze implicaties:

- Variabelen kunnen zomaar van type veranderen tijdens programma's
- Een programmeertaal moet extra moeite doen om aanpasbaarheid correct te ondersteunen in alle situaties
- Aanpasbaarheid is een veel voorkomende oorzaak van bugs. Het maakt redeneren over programma's moeilijker

Dit inzicht is gaandeweg doorheen de jaren gekomen en heeft impact het ontwerp van programmeertalen. Een ruwe vuistregel:

- Oudere talen beschouwen aanpasbaarheid als de norm
- Nieuwere talen beschouwen aanpasbaarheid als de uitzondering

AANPASBAARHEID: TIPS VOOR ONDERWIJS

Maak aan je leerlingen duidelijk dat aanpasbaarheid gevaren inhoudt.
Geef de voorkeur aan niet-aanpasbare variabelen indien mogelijk.

TYPES

TYPES

Elke variabele - *en ook andere datastructuren; zie verder* - heeft een bepaald type.
De meest voorkomende types zijn:

TYPES

Elke variabele - *en ook andere datastructuren; zie verder* - heeft een bepaald type.
De meest voorkomende types zijn:

- Tekst: `string`
- Gehele getallen: `int`
- Kommagetallen: `float`
- Booleanse waarden: `bool` of `boolean`

Sommige talen nemen `int` en `float` samen tot `number`.

Vaak noemt men deze de **primitieve** types.

TYPES: WAAROM?

Types worden gebruikt voor een aantal zaken:

- De computer kan afleiden hoeveel geheugen gereserveerd moet worden voor een variabele, op basis van het type
- Types geven aan welke 'acties' je met data kan doen. Bijvoorbeeld:
  - Wiskundige bewerkingen uitvoeren met een `int`
  - Een `string` omzetten naar hoofdletters
  - Een `bool` gebruiken in een `if`-statement
- Je kan bepaalde fouten in programma's opsporen zonder ze uit te voeren, door te controleren of de types overal kloppen

TYPES: DISCUSSIE

Sommige talen dwingen af dat in heel je programma alle types correct worden gebruikt. Indien er een fout is tijdens compilatie, kan het programma niet uitgevoerd worden tot de fout is verbeterd. Dit noemt men **statische type-checking** (Eng: *static type checking*)

Andere talen (waaronder Python) zijn 'losser' met types. Ze controleren types niet op voorhand. De programma's starten, maar zullen falen tijdens uitvoering wanneer code met een typefout wordt opgeroepen. Dit noemt men **dynamische type-checking** (Eng: *dynamic type checking*).

Over het algemeen geldt: types worden nuttiger naarmate:

- Het aantal lijnen code in een project toeneemt
- Meer programmeurs in aanraking komen met de code

TYPES: TIPS VOOR ONDERWIJS

Een leerling moet steeds kunnen afleiden wat het type van elke variabele in een programma is.

Laat leerlingen het type van elke variabele in hun programma's opschrijven:

- Als commentaar boven de declaratie
- Als deel van de naam van de variabele
- Als deel van de declaratie (kan enkel* in talen met statische type checking)

Vermijd dat variabelen van type veranderen binnen een programma. In talen zoals Python kan dit, maar wordt dit sterk ontmoedigt. In talen met statische type checking kan dit sowieso niet.

OMGAAN MET COMPLEXERE VORMEN VAN DATA

DATASTRUCTUREN

INLEIDING

Niet alle data kan voorgesteld worden met enkel de 4 primitieve types. Hoe stel je bijvoorbeeld het volgende voor?

- De examenresultaten van een student voor een bepaald semester
- Een personage in een videospel
- Een groep van personages in een videospel
- Een klaslokaal in het beheersysteem van de UCLL

We hebben nood aan meer complexe datastructuren.

DATASTRUCTUREN IN PYTHON (HERHALING VORIG JAAR)

Naam	Aanpasbaar?	Geordend?	Selecteer elementen via	Toegelaten types voor elementen
Tuple	❌	✅	index	Alles
Lijst	✅	✅	index	Alles
Dictionary	✅	❌ (*)	key	Key: enkel primitieve types Value: alles
String	❌	✅	index	Karakters

Doel van deze les: kennis verdiepen en uitbreiden.

SOORTEN OPSLAG

Onderscheid tussen:

- Tijdelijke opslag van data:
  - Deze data verdwijnt zodra een programma stopt
  - Primitieve types
  - Datastructuren
- Permanente opslag van data:
  - Deze data blijft bestaan, los van programma's
  - Bestanden
  - Gegevensbanken

STATISCHE EN DYNAMISCHE STRUCTUREN

Bij statische structuren wordt de (maximale) hoeveelheid geheugen voor de structuur op voorhand berekend tijdens compilatie.

Bij dynamische structuren is het geheugengebruik niet op voorhand geweten. Tijdens uitvoering vraagt een programma extra geheugen aan en geeft het nadien vrij indien mogelijk.

REEKSEN (ARRAYS) VERSUS LIJSTEN (LISTS)

Reeksen zijn statisch. Je moet aangeven hoe groot ze moeten zijn, of er wordt een standaardwaarde gekozen (bv. 10 elementen). Ze zijn ideaal voor tabellen en rijen.

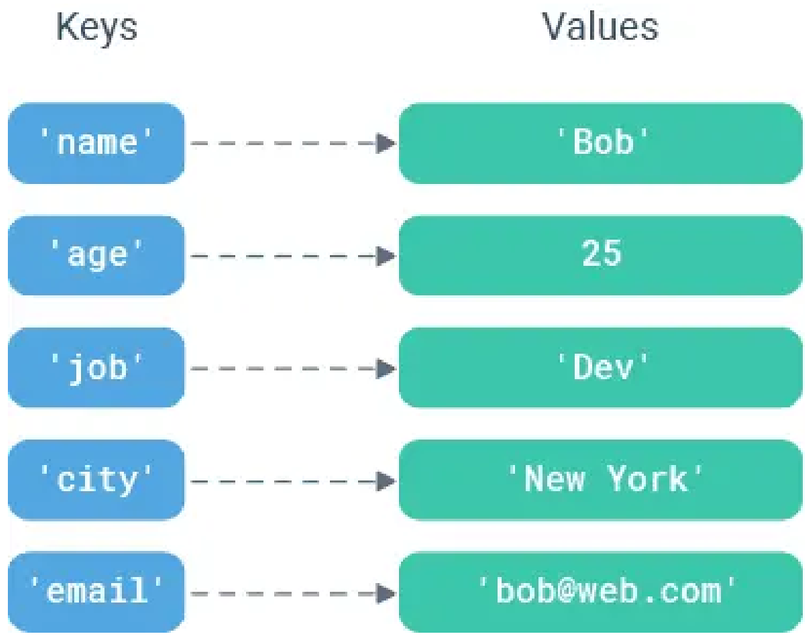
Lijsten zijn dynamisch: ze kunnen groeien en krimpen. Er zijn verschillende subtypes van lijsten (bv. linked lists). Ze zijn ideaal als de hoeveelheid data niet op voorhand geweten is tijdens compilatie. Ze zijn flexibiler dan arrays, maar bepaalde operaties duren langer.

Veel moderne talen hebben een hybride structuur: een dynamische array. Deze gedragen zich als arrays, maar zullen automisch herschalen als er te weinig plaats is. Een `list` in Python is een dynamische array.

MAPS

Maps lijken op lijsten, maar de volgorde van elementen ligt niet vast. Elementen zijn een combinatie van een sleutel (key) en bijhorende waarde (value).

In Python noemt dit een **dictionary**. In PHP noemt dit een **associatieve array**. Beide namen zijn echter minder gangbaar dan 'map'.

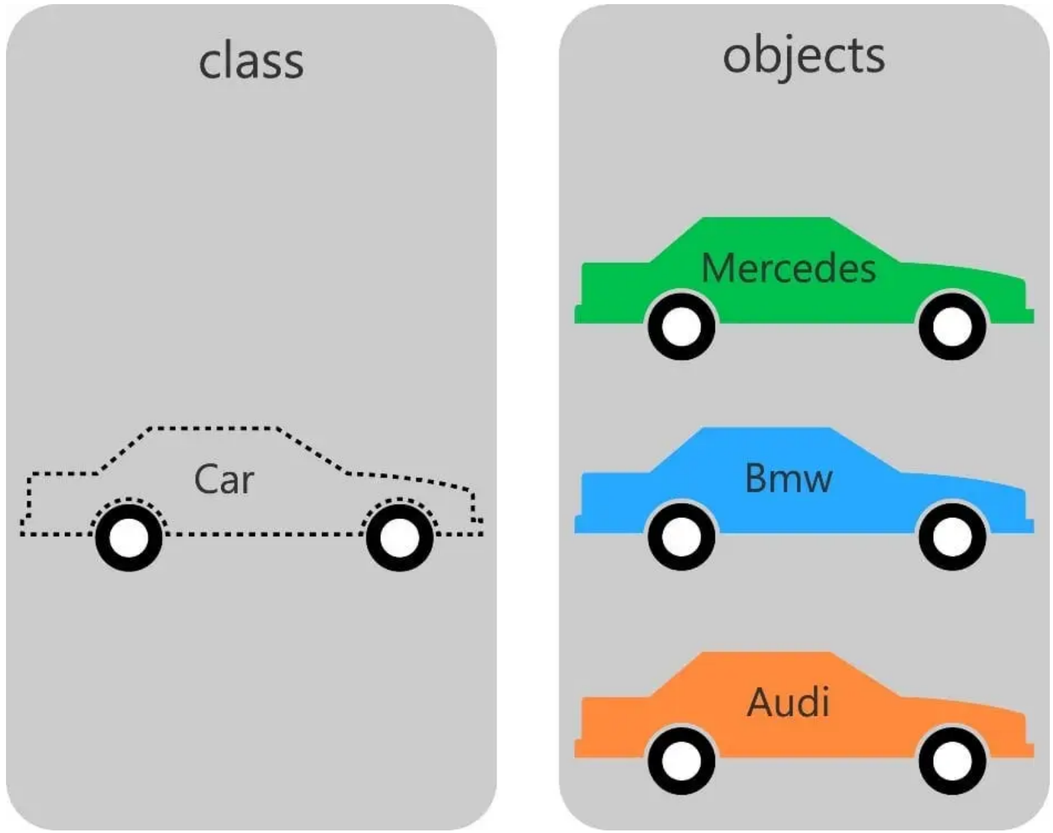


KLASSEN EN OBJECTEN

Klassen en objecten zijn een 'upgrade' van maps. Een object heeft **eigenschappen**en **methodes**. Andere benamingen voor eigenschappen (properties) zijn velden (fields).

Een klasse is een blauwafdruk. Ze beschrijft hoe bepaalde objecten gemaakt worden, welk soort kenmerken ze hebben, en welke methodes ervoor bestaan.

We gaan in latere lessen dieper in op gevorderde concepten van objectgericht programmeren.



DATASTRUCTUREN EN TYPES

Datastructuren bepalen hoe je op een bepaalde manier met data om moet gaan. Voor elke datastructuur bestaat er een type om dit te controleren. Logischerwijs kan je ook fouten tegen types maken bij datastructuren. Bijvoorbeeld:

- `append()` oproepen op een dictionary
- Een veld van een object proberen te gebruiken, terwijl dit niet bestaat

Elke klassen telt ook als een type. Door nieuwe klassen aan te maken, kan je dus nieuwe types creëren.

OPRUIMEN VAN DATA

OPRUIMEN VAN DATA

Een programma heeft geheugen nodig om uitgevoerd te kunnen worden. Geheugen vragen is makkelijk: maak een nieuwe variabele/datastructuur/object aan. Maar hoe geef je geheugen terug vrij?

In oudere programmeertalen zijn programmeurs zelf verantwoordelijk hiervoor. C++ heeft bijvoorbeeld het speciale woord `delete`.

Manueel beheer van geheugen is echter vatbaar voor fouten. Geheugenlekken (Memory leaks) komen veel voor en kunnen dramatische gevolgen hebben.

Moderne talen nemen het geheugenbeheer in eigen handen. Ongebruikte variabelen en datastructuren worden automatisch opgeruimd door een achtergrondproces genaamd de **Garbage collector**.

- Voordeel: geen geheugenlekken meer, veel minder kans op fouten
- Nadeel: de Garbage collector kan programma's vertragen en maakt de snelheid van programma's minder voorspelbaar

MEER OVER GEHEUGEN EN VARIABELEN

CALL BY VALUE EN CALL BY REFERENCE

QUIZ

Wat is de uitvoer van dit programma?

```python
a = 5
b = a
a = a + 8
print(b)
```

- 5
- 8
- 13
- iets anders

QUIZ

Wat is de uitvoer van dit programma?

```python
a = [5]
b = a
a.append(8)
print(b)
```

[5]

[5, 8]

[13]

iets anders

CALL BY SEMANTICS

Wat is hier aan de hand?

De waarden van variabelen worden opgeslagen in het geheugen van een computer. De locatie hiervan noemt men het **geheugenadres**. Een variabele verwijst naar een adres, die dan weer verwijst naar de effectieve waarde.

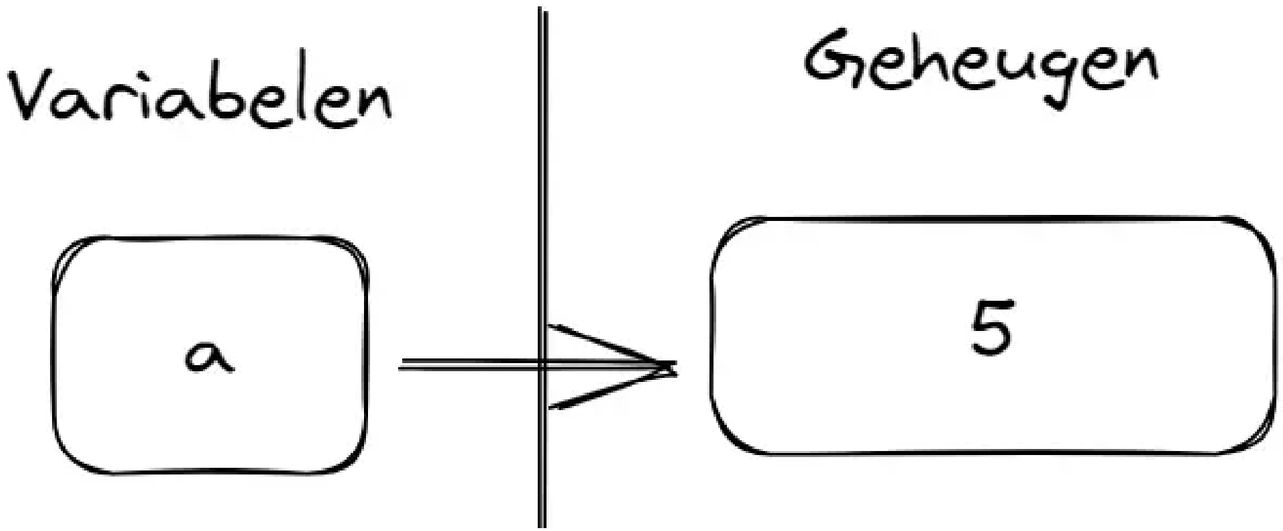
In Python kan je het adres waarnaar een variabele verwijst, opvragen via `id()`. Dit is gewoon een nummer, het heeft op zich weinig betekenis.

De twee voorbeelden illustreren twee technieken om met geheugen om te gaan. In het eerste voorbeeld wordt **call by value** gedemonstreerd. In het tweede voorbeeld wordt **call by reference** gedemonstreerd.

CALL BY VALUE 1

```python
a = 5
```

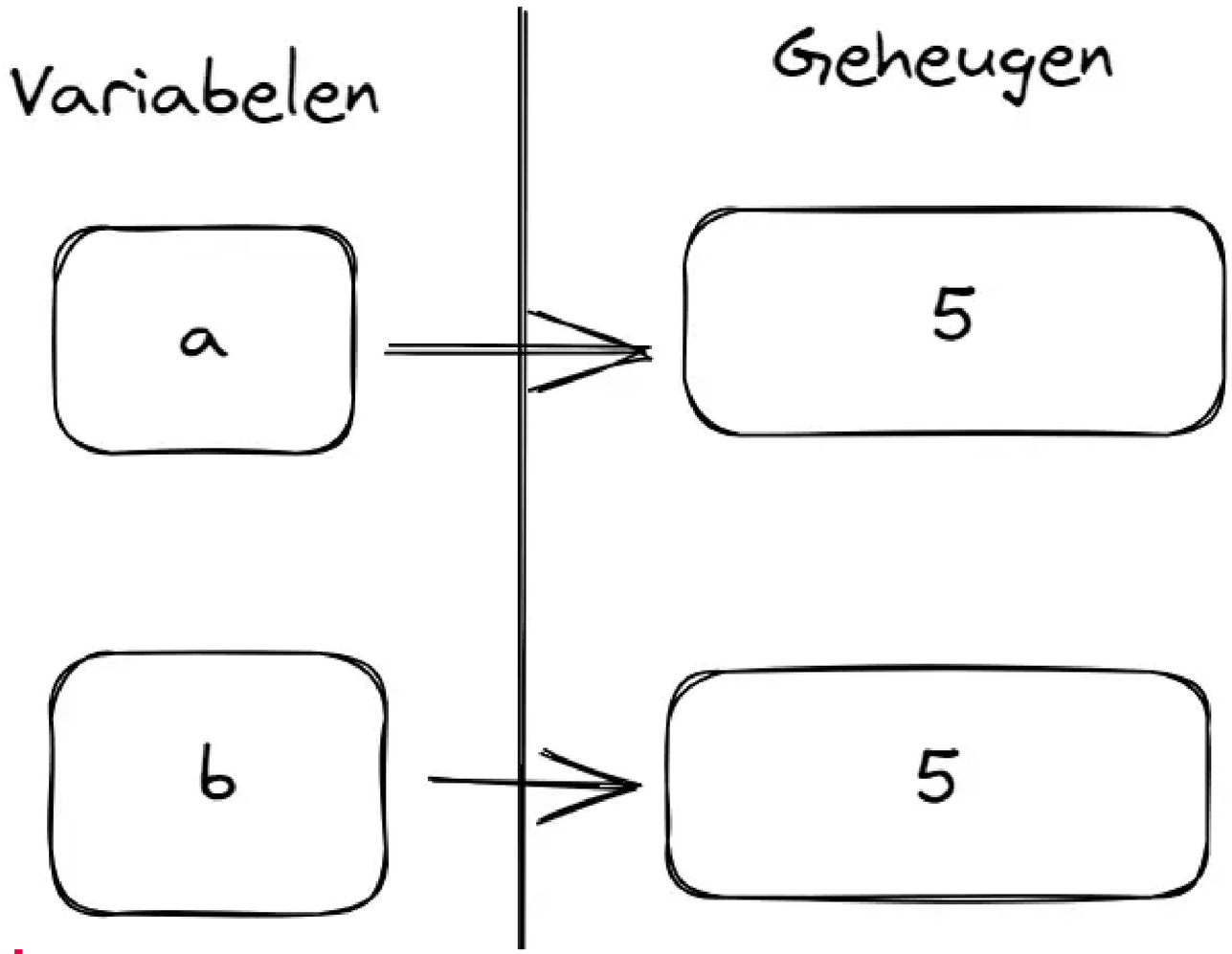
Initialiseer `a` en geef het de waarde `5`.



CALL BY VALUE 2

```python
b = a
```

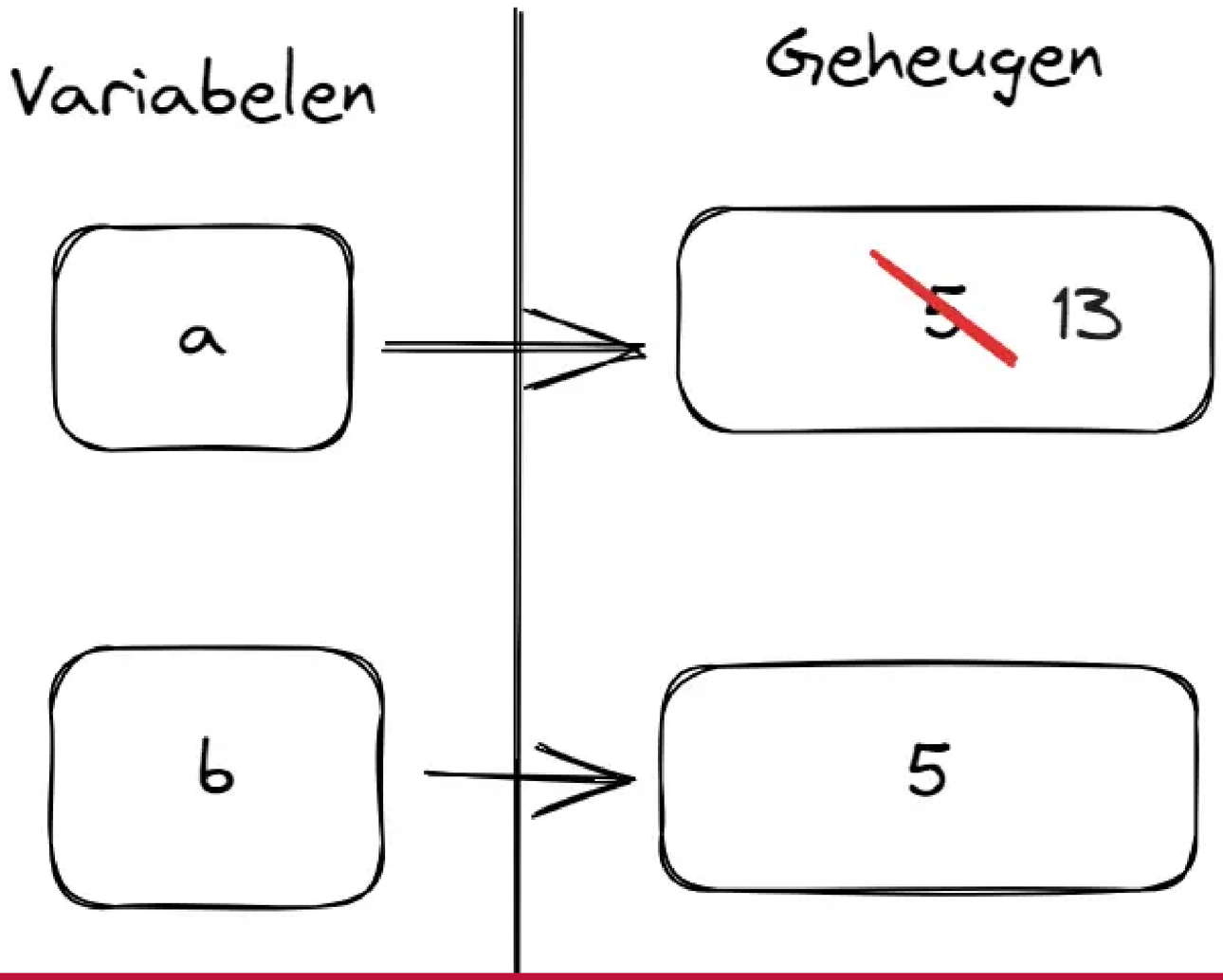
Initialiseer `b` en kopieer de huidige waarde van `a`. Deze nieuwe waarde heeft een aparte plek in het geheugen.



CALL BY VALUE 3

```python
a = a + 8
```

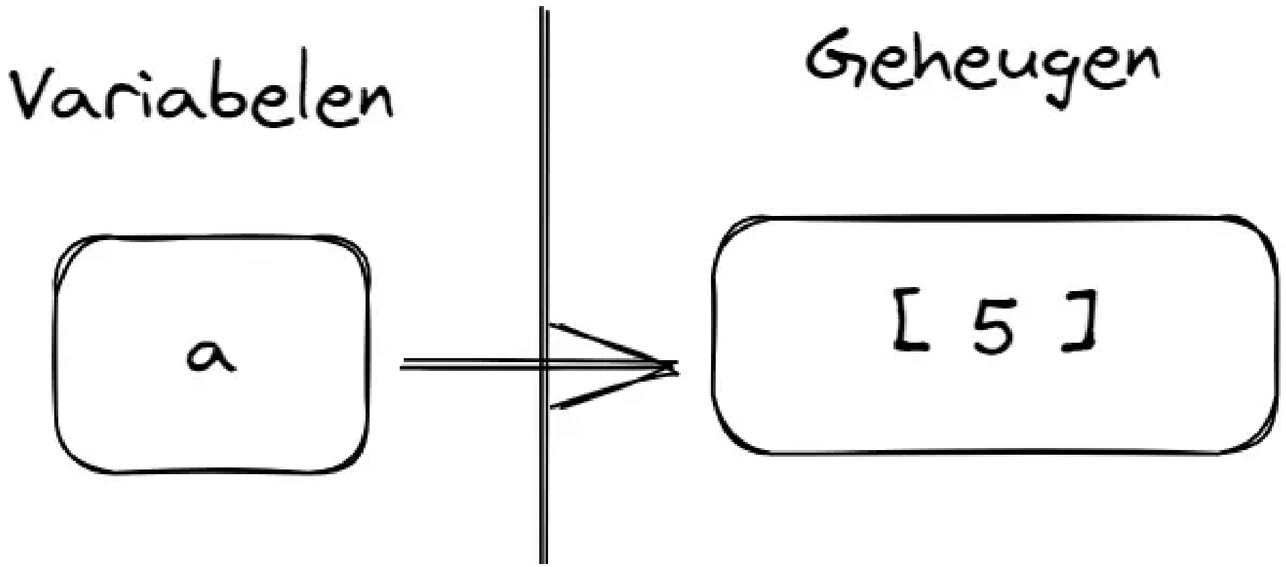
Verander de waarde van `a`. Dit heeft geen impact op `b`, omdat er geen verbinding is tussen beiden.



CALL BY REFERENCE 1

```python
a = [5]
```

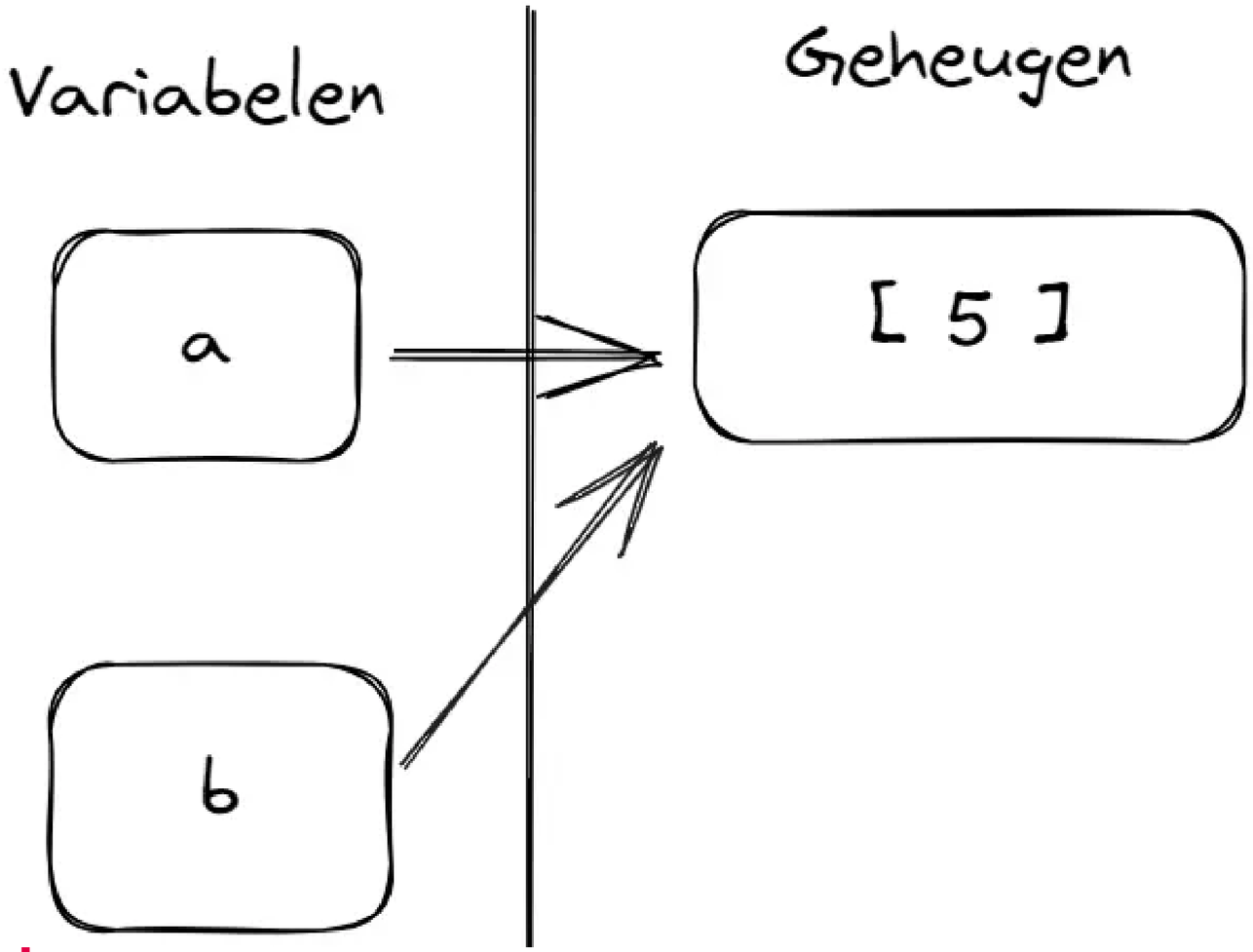
Initialiseer de lijst `a` en voeg de waarde `5` toe.



CALL BY REFERENCE 2

```python
b = a
```

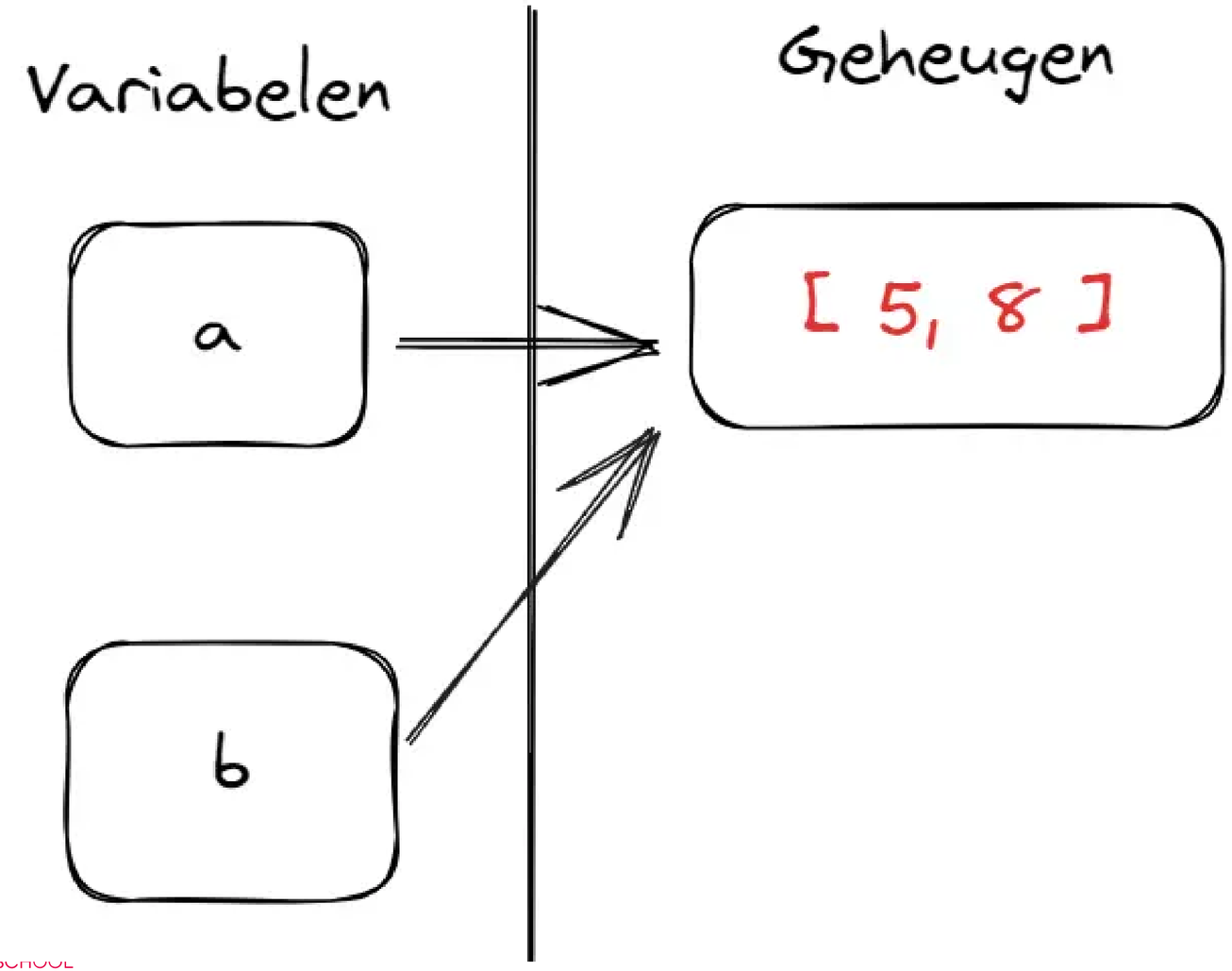
Initialiseer `b` door het adres van `a` te kopiëren. Beiden variabelen verwijzen dus naar dezelfde plek in het geheugen.



CALL BY REFERENCE 3

```python
a.append(8)
```

Voeg een extra waarde toe aan `a`. Dit heeft impact op `b`, omdat ze hetzelfde onderliggende geheugen delen.



CALL BY SEMANTICS: WAAROM?

Programmeertalen passen allerlei technieken toe om het geheugenverbruik te minimaliseren. Een van beide technieken helpt hierbij. Welke en waarom?

Call by value

Call by reference

CALL BY SEMANTICS: WAAROM?

Programmeertalen passen allerlei technieken toe om het geheugenverbruik te minimaliseren. Een van beide technieken helpt hierbij. Welke en waarom?

Call by value

Call by reference

Door call by reference toe te passen, kunnen we volledige lijsten, dictionaries, ... doorgeven in programma's zonder ze telkens te kopiëren.

Elke programmeertaal kiest zelf welke techniek wanneer wordt toegepast. Een ruwe vuistregel:

- Primitieve types maken gebruik van call by value
- Datastructuren maken gebruik van call by reference

Bij twijfel: zoek op!

(*) Python gebruikt technisch gezien geen van beiden, maar nog een andere vorm genaamd *call by assignment*.

TIPS VOOR ONDERWIJS

Vermijd om leerlingen direct te confronteren hiermee. Kies je voorbeelden en opgaven zorgvuldig zodat ze hier niet tegen opbotsen.

Visualiseer het geheugen met tekeningen, grafieken, ... om het minder abstract te maken.

VERGELIJKING TUSSEN DATASTRUCTUREN

WANNEER GEBRUIK JE WAT?

Gebruik **tuples** voor simpele, statische, geordende data.

Gebruik **lijsten** als:

- de volgorde van elementen belangrijk is
- volgorde en/of lengte mag variëren (elementen toevoegen/verwijderen of van plaats wisselen)

Gebruik **dictionaries** als:

- de volgorde van elementen niet vastligt of niet belangrijk is
- je de data kan beschrijven via keys

Gebruik **objecten** als:

- je de data kan beschrijven via kenmerken
- EN je wilt de data manipuleren via zelfgekozen methodes

OPDRACHTEN

OPDRACHTEN

Suggesties voor je portfolio:

- Maak oefeningen uit hoofdstuk 11, 12 en 13 van het handboek
- Onderzoek hoe call by semantics werken in andere programmeertalen
- Onderzoek **Sets**, een andere - minder gebruikte - datastructuur in Python. Vergelijk ze met de andere structuren. Zie ook hoofdstuk 14 in het handboek
- Python ondersteunt sinds kort *type hints*. Deze helpen met het controleren van types in programma's, maar zijn 'optioneel':
  - Zie bijvoorbeeld Python Type Checking (Guide)
  - Test het uit bij het schrijven van programma's. Helpt het met het opsporen van fouten? Zou je jouw eigen leerlingen Python leren met of zonder type hints?