6IICT – PROG4

Leren programmeren met python

Inhoud

1 Object-georiënteerd programmeren ... 3
1.1 Introductie tot OOP ... 3
1.2 OOP – Een praktijkvoorbeeld ... 4
1.3 Basisprincipes van OOP (in Python) ... 5
1.3.1 Het begrip klasse en attribuut ... 5
1.3.2 Het begrip object ... 5
1.3.3 Het begrip methode ... 5
1.3.4 Het begrip eigenschappen en de __init__() methode ... 6
1.3.5 Nesten van objecten ... 8
1.3.6 Oefeningen ... 10
1.4 OOP – Inheritance ... 13
1.4.1 Het begrip gedeelde klasse (Inheritance of Erving) ... 13
1.4.2 De Child class ... 13
1.4.3 Meervoudige inheritance ... 15
1.4.4 Oefeningen ... 16
1.5 Klassen en data types in Python ... 18
1.6 OOP in Arduino ... 19

2 Recursie ... 20
2.1 Wat is recursie? ... 20
2.2 Recursieve definities ... 22
2.3 Recursie toepassen ... 23
2.4 Oefeningen ... 25

Dit document is niet finaal en zal door de loop van het jaar verder aangevuld/verbeterd worden. Schrijf daarom geen opmerkingen/notities in dit document!

1 Object-georiënteerd programmeren

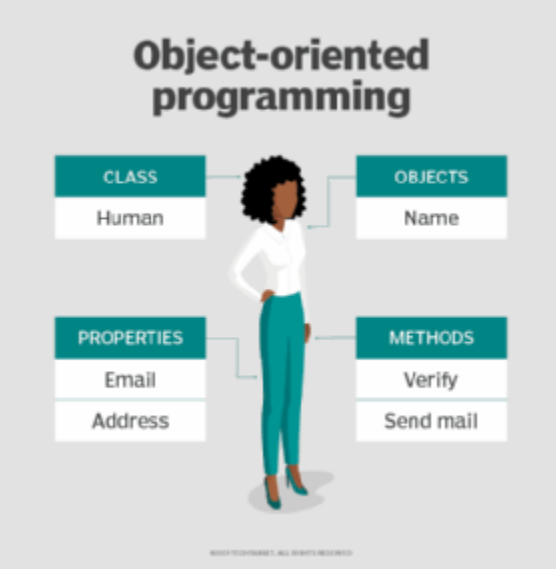
1.1 Introductie tot OOP

Zoals je uit de naam wel kunt afleiden, verwijst Object-georiënteerd programmeren of OOP naar talen die objecten gebruiken bij het programmeren. OOP werd voor het eerst gebruikt bij C++ met als hoofddoel om gegevens en de functies die erop werken samen te binden, zodat geen ander deel van de code toegang heeft tot deze gegevens dan die functie. OOP biedt een aantal grote voordelen:

- Opbouwen van programma's uit standaard modules die met elkaar communiceren.
- Laat ons toe het programma te splitsen in kleine, gemakkelijk oplosbare, delen.
- OOP systemen kunnen makkelijk geüpdatet worden.
- Meerdere instanties van objecten kunnen naast elkaar bestaan.
- Afschermen van gegevens zorgt voor de creatie van veilige programma's.
- Door erving kunnen we redundante code elimineren.
- Object-Oriented ontwerp staat ons toe om meer details van een model te implementeren.

Het is dan ook niet verwonderlijk dat OOP snel uitgroeide tot een groot succes. Voor een lange tijd was het zelfs de norm om zoveel mogelijk Object-Oriented te schrijven. OOP heeft echter ook belangrijke nadelen, waardoor een programmeur goed moet nadenken over wanneer deze wel/niet te gebruiken. Enkele nadelen zijn:

- OOP bestanden zijn groter dan procedurele benaderingen. Dit leidt tot tragere programma's.
- OOP is niet universeel. Het is dan ook niet toepasbaar voor ieder type probleem.
- OOP is, vooral voor beginners, moeilijk om correct toe te passen.



Figuur 1: Voorbeeld van OOP met als object "persoon" van de klasse "de mens"

1.2 OOP – Een praktijkvoorbeeld

Ik schrijf dit terwijl ik zit aan mijn keukentafel. Naast mij staat een fruitschaal. In de schaal liggen appels. Deze appels delen bepaalde eigenschappen, maar hebben ook verschillen. Ze delen hun naam, hun prijs, en hun leeftijd, maar ze hebben alle (iets) verschillende gewichten. Op de schaal liggen ook peren. Net als de appels zijn ze een soort fruit, maar ze hebben ook een hoop verschillen met appels: verschillende namen, verschillende kleuren, verschillende bomen waar ze aan groeien. Toch delen ze ook dingen met appels. Beide zijn stukken fruit en kan je opeten. Dit in tegenstelling tot de tafel waar ik aan zit, die zou ik natuurlijk niet proberen op te eten.

Je kan niet de hele wereld overnemen in een computerprogramma. In plaats hiervan maken we een benadering. Dit wordt ook wel een ***model*** genoemd. Een manier om de wereld te modelleren is met ***objecten*** zoals appels, peren en tafels. Sommige van die objecten hebben een hoop zaken gemeen, bijvoorbeeld, iedere appel deelt veel ***eigenschappen*** met iedere andere appel. Het klinkt daarom zinvol om een ***klasse*** "Appel" te definiëren, die de eigenschappen bevat die alle appels delen, en dan voor iedere individuele appel deze eigenschappen in te vullen. Sommige eigenschappen zijn voor iedere appel hetzelfde, zo heeft iedere appel een klokkenhuis. Dit soort eigenschappen worden ***attributen*** genoemd. Het is ook mogelijk om functionaliteiten of ***methodes*** aan de klasse toe te voegen. Dit zodat we met iedere appel bepaalde dingen kunnen doen. Zoals ze opeten. Hetzelfde kan ik doen voor peren, die hun eigen klasse "Peer" moeten krijgen. En hoewel appels en peren van elkaar verschillen, delen ze ook eigenschappen die mij het gevoel geven dat ik ze een ***gedeelde klasse*** moet geven: de klasse "Fruit." Ieder object dat thuishoort in de klasse Fruit heeft in ieder geval de functionaliteit dat ik het kan eten. Wat betekent dat iedere appel niet alleen behoort tot de klasse "Appel," maar ook tot de klasse "Fruit" – net als de peren.

Tabel 1: Overzicht kernwoorden

Kernwoord	Uitleg	Voorbeeld
Model	Vereenvoudigen van de wereld	We verwaarlozen luchtweerstand
Object	Onderdeel van de wereld	Een appel
Eigenschap	Kenmerk van een object	De kleur van een appel
Methodes	Functionaliteit van een object	Je kan appels eten
Klasse	Verzameling van gelijkaardige objecten	Iedere appel behoort tot Appel
Attribuut	Eigenschap die voor ieder object gelijk is	Iedere appel heeft een klokkenhuis
Gedeelde klasse	Klasse waarin andere klassen toebehoren	Appel en Peer behoort tot Fruit

Als ik er goed over nadenk: ik kan meer eten dan alleen appels en peren. Ik kan ook taart eten. En champignons. En brood. En drop. Dus misschien heb ik nog een andere klasse nodig, waartoe ook de klasse "Fruit" behoort. De klasse "Voedsel," misschien?

Samengevat. Ik probeer de wereld (of een deel ervan) te modelleren, ik gebruik hiervoor objecten. In plaats van ieder object apart te modelleren, kan ik beter klassen van objecten definiëren, zodat ik algemene uitspraken kan doen over groepen objecten ("appels zijn rood of groen"). Ik kan spreken over relaties tussen klassen ("Appel en Peer zijn beide Fruit"), en ik methodes kan definiëren die op klassen werken ("Voedsel kan je eten. Dit zorgt ervoor dat het object verwijderd wordt uit de wereld"). Omdat ik objecten kan eten als ze horen tot de klasse "Voedsel," kan ik "Fruit" eten. En omdat ik "Fruit" kan eten, kan ik objecten eten die tot de klasse "Appel" behoren.

1.3 Basisprincipes van OOP (in Python)

Uit de tekst van **deel 1.2 OOP – Een praktijkvoorbeeld** zijn een aantal kernwoorden afgeleid. We zullen deze nu in meer technisch detail uitwerken.

1.3.1 Het begrip klasse en attribuut

Om objecten te maken gebruiken we klassen. Je kan deze beschouwen als een blauwdruk voor het creëren van een bepaald type object. Zoals je misschien al uit **deel 1.2** gemerkt heb, definiëren we klassen met hoofdletter. Je kan op basis van een klasse zoveel objecten maken als je nodig hebt.

Om een klasse te maken, gebruiken we het keyword “class”. Hieronder een voorbeeld.

```
class Cat:
   name = "Borysz"

print(Cat.name)
```

In dit voorbeeld is de variabele name een attribuut. Attributen zijn variabelen die voor alle objecten van een klasse hetzelfde zijn. Het is mogelijk om attributen van een klasse op te roepen.

<Klasse>.<attribuut>

Oefen mee

Maak een eigen klasse Dog. Geef deze Dog als attributen een naam (name) en massa (weight) naar keuze. Print vervolgens de naam en massa van Dog.

1.3.2 Het begrip object

Nu onze blauwdruk af is, kunnen we deze gebruiken om een object te genereren. Dit gebeurt als volgt.

```
class Cat:
   name = "Borysz"

my_cat = Cat()
print(my_cat.name)
```

Bij het aanmaken van my_cat heeft deze het attribuut name uit de klasse gekopieerd en in my_cat opgeslagen. Het is met andere woorden ook mogelijk een attribuut vanuit een object op te roepen:

<object>.<attribuut>

Oefen mee

Maak een object van de klasse Dog genaamd my_dog met attributen name en weight. Print deze.

1.3.3 Het begrip methode

We willen functionaliteit aan objecten toevoegen. Dit doen we door methodes aan de klasse toe te voegen. Bij het maken van een object van deze klasse, krijgt dit object ook toegang tot deze methode. Een methode van een object oproepen gebeurt als volgt:

<object>.<methode (<parameters>)>

Deze opbouw lijkt zeer sterk op die van een functie. Dit komt omdat methodes functies zijn. Maar wel functies die enkel uitvoerbaar zijn via een object. Ze delen dan ook alle eigenschappen van functies. Parameters zijn optioneel. Zo heeft een methode, net als een functie, soms extra informatie nodig om goed te werken. Zo heb je de prijs per kilo nodig om met de massa van een appel de prijs te bepalen.

In onderstaand voorbeeld is de methode miauw() toegevoegd aan de klasse Cat.

```
class Cat:
  name = "Borysz"

  def miauw(self):
    print(f"{self.name} says miauw.")

my_cat = Cat()
my_cat.miauw()
```

De parameter self

Net is gezegd dat parameters optioneel zijn. Dit klopt echter niet helemaal. Iedere methode verwacht minstens een parameter, “self”. We willen namelijk dat de methode toegang krijgt tot de informatie van het object (in dit geval name). “self” is met andere woorden een referentie aan het object (my_cat) waarvan je de methode (miauw()) oproept. De term “self” is niet verplicht. Je mag deze noemen wat je wilt. Wat wel belangrijk is, is dat dit de eerste parameter moet zijn. Desondanks is het een ongeschreven regel om “self” te gebruiken.

Let op: self staat niet in my_cat.miauw() . Dit is logisch omdat my_cat al refereert naar my_cat!

Oefen mee

Voeg aan de klasse Dog een methode bark() toe. Deze methode moet een tekst printen met erin de naam van de hond, gevolgd door “ says bark.”. Gebruik als eerste parameter een andere term dan self.

Oefen mee

Waarom geeft volgende code een foutmelding?

```
class Cat:
  name = "Borysz"

  def info(emotie, self):
      print(f"{self.name} is {emotie}")

my_cat = Cat()
my_cat.info("boos")
```

1.3.4 Het begrip eigenschappen en de __init__() methode

Een grote kracht van OOP is dat je met een klasse veel verschillende objecten kunt maken. Om bovenstaand voorbeeld verder te zetten zou ik ook een tweede kat kunnen aanmaken, een kat voor mijn buurman, … . Echter heeft deze kat ook de naam “Borysz”. Dit klopt natuurlijk niet. We hebben name behandeld als iets wat gelijk is voor ieder object (**attribuut**), terwijl het voor ieder object verschillend is (**eigenschap of property**)!

De makkelijkste manier om eigenschappen toe te voegen aan een object is met de __init__() methode. Deze wordt namelijk automatisch opgeroepen bij de creatie van het object. We passen het voorbeeld van de kat aan zodat iedere kat een eigen naam en leeftijd heeft.

```
class Cat:
  def __init__(self, name, years):
    self.name = name
    self.age = years

my_cat = Cat("Fons", 11)

print(my_cat.name)
print(my_cat.age)
```

Bij het aanmaken van het object my_cat moeten we nu een name (“Fons”) en years (11) opgeven. De **__init__()** functie zal deze variabelen namelijk gaan toewijzen als eigenschappen van my_cat. Met self wordt gerefereerd naar het object waarmee de methode uitgevoerd wordt. Eigenlijk staat er in __init__() dus:

```
# Example do not run!
my_cat.name = name
my_cat.age = years
```

“Fons” en 11 worden dus opgeslagen in twee locaties (eigenschappen) waar enkel my_cat bij kan. De eigenschappen moeten niet dezelfde naam hebben als de opgeslagen variabelen .Dit wordt echter meestal wel gedaan omdat dit logischer is (zie age en years). Een variabele aan een eigenschap van een object toewijzen gebeurt dus in het algemeen als volgt.

<object>.<eigenschap> = <variabele>

En om de eigenschap op te roepen.

<object>.<eigenschap>

Het valt hopelijk op dat dit analoog is aan het oproepen van een attribuut. Dit is logisch aangezien een attribuut eigenlijk een soort eigenschap is. Namelijk een eigenschap die voor ieder object exact hetzelfde is.

Oefen mee

Pas de klasse Dog aan, zodanig dat deze twee eigenschappen name en weight heeft. Geef Dog ook een attribuut loyal mee. Deze heeft natuurlijk de waarde True. Print deze alle drie voor minstens twee verschillende objecten van de klasse Dog.

Oefen mee

Voeg aan de klasse Dog een methode is_age() toe. Deze methode moet een tekst printen met erin de naam en leeftijd van het object dat de methode oproept. Geef de leeftijd als parameter mee. Voer deze methode met twee objecten uit.

Tenslotte is het mogelijk om eigenschappen, of objecten, te verwijderen met het **del** keyword. Volgende blok code geeft een voorbeeld.

```
class Cat:
  def __init__(self, name, age):
    self.name = name
    self.age = age

  def miauw(self):
    print(f"{self.name}, age {self.age}, says miauw.")

my_cat = Cat("Fons", 11)

""" Method no longer works due to age being removed """
del my_cat.age
my_cat.miauw()

""" Method no longer works due to my_cat being removed """
del my_cat
my_cat.miauw()
```

Oefen mee

Voeg een methode has_weight() toe aan de klasse Dog. Deze moet de naam en massa van een object printen. Voer deze methode uit. Probeer vervolgens de property weight te wijzigen en de methode opnieuw uit te voeren. Verwijder tenslotte de property weight en voer de methode opnieuw uit.

1.3.5 Nesten van objecten

Net zoals het mogelijk is om lijsten in andere lijsten te nesten, is het mogelijk om objecten van de ene klasse, te nesten in objecten van een andere klasse. Stel bijvoorbeeld dat we de locatie van onze kat in het huis willen weten. Een locatie bestaat (vereenvoudigd) uit twee coördinaten. Momenteel is de locatie van de kat nog onbekend.

```
class Location:
    def __init__(self, x=-1, y=-1):
        # By default we do not know where our cat is
        self.x = x
        self.y = y

class Cat:
  def __init__(self, name, years, location):
    self.name = name
    self.age = years
    self.location = location

location = Location()
my_cat = Cat("Fons", 11, location)

print(my_cat.location)
print(my_cat.location.x)
```

Bij het printen van my_cat.location volgt een rare boodschap. In mijn geval.

<__main__.Locatie object at 0x0000021BD170BBE0>

Deze boodschap zegt eigenlijk het volgende. Het object locatie behoort tot de klasse Locatie en kan in het geheugen gevonden worden op **0x0000021BD170BBE0.** Deze boodschap verschijnt omdat we het hele object locatie printen. Indien we een eigenschap van locatie printen (via "." Kunnen we in de geneste objecten bewegen). my_cat.location.x geeft dus de waarde van het x-coördinaat.

Oefen mee

Voeg aan de klasse Dog een eigenschap parents toe. Deze eigenschap is een object van de klasse Parents. Dit object heeft op zijn beurt weer de eigenschappen father en mother, die de namen van de vader en moeder van de hond bevatten. Print waar in het geheugen het object parents opgeslagen wordt en print de naam van de moeder via een object van de klasse Dog.

Stel dat de locatie van de kat gevonden is en we dit willen updaten in my_cat. Dit kan als volgt.

```
class Location:
    def __init__(self, x=-1, y=-1):
        # Default we do not know where our cat is
        self.x = x
        self.y = y

class Cat:
  def __init__(self, name, years, location):
    self.name = name
    self.age = years
    self.location = location

location = Location()
my_cat = Cat("Fons", 11, location)

print(f"{my_cat.location.x}, {my_cat.location.y}")
location.x = 4
location.y = 9
print(f"{my_cat.location.x}, {my_cat.location.y}")
```

Het wijzigen van Location location, zal er ook voor zorgen dat Cat my_cat gewijzigd wordt. Herinner je dat my_cat.location simpelweg wees naar waar Location location zich in het geheugen bevindt. Het is met andere woorden een referentie ernaartoe. Wijzig de onderliggende waarde zoveel als je wilt, maar de plek waar de waarde gestockeerd is blijft hetzelfde. Deze zal de nieuwe waarde overnemen.

Oefen mee

Herneem voorgaande *oefen mee* wijzig de naam van de moeder via het object Parents parents. Print vervolgens de moeder opnieuw via my_dog. Wat gebeurt er als je de moeder wijzigt via het object Dog my_dog (my_dog.parents.mother = …)? Schrijf in commentaar waarom je denkt dat dit gebeurt.

Dit is niet altijd wat we willen. Stel dat je meerdere katten hebt. Wil je voor iedere kat een apart Locatie object maken? Het is mogelijk om dit op te lossen door een kopie van het Locatie object te maken. Dit kan met de **copy()** functie. Deze maakt een kopie van de meegegeven locatie en slaat ze op een andere plaats in het geheugen op.

```
from copy import copy # Get the copy() function

class Location:
```

```
    def __init__(self, x=-1, y=-1):
        self.x = x
        self.y = y

class Cat:
  def __init__(self, name, years, location):
    self.name = name
    self.age = years
    self.location = copy(location) # Create copy of location

  def has_location(self): # For ease of use
      return f"{self.name} is at {self.location.x}, {self.location.y}"

location = Location()
my_cat = Cat("Fons", 11, location)

print(my_cat.has_location())
location.x , location.y = 4 , 3
print(my_cat.has_location())
my_cat.location.x , my_cat.location.y = 0 , 2
print(my_cat.has_location())
```

Oefen mee

Herneem voorgaande *oefen mee* zorg dat een **kopie** van Parents parents in Dog my_dog bewaard wordt. Voeg een methode has_parents() toe aan Dog. Deze returnt de ouders van het object. Probeer vervolgens de ouders van de hond te wijzigen via Parents parents en Dog dog. Gebruik print om de resultaten te controleren.

1.3.6 Oefeningen

Je dient de opgegeven opdrachten te maken en in te leveren. Overige opdrachten zijn optioneel.

Opdracht 1.1 Family (1.3.4)

Maak een klasse Family. Je moet met behulp van deze klasse, objecten van je gezinsleden aanmaken. Ieder gezinslid moet volgende variabelen bezitten:

Variabel	Explanation
race	All people belong to the human race
sex	Sex of the family member (Man/Woman/X)
hair	Hair color of family member Brown/Blond/…
relation	Relation family member to you (father, sister, …)
age	Age of family member
name	Name of family member

Jij bepaalt zelf of het opslaan van deze data best als **attribuut** of **property** gebeurt.

Hiernaast moet het via **methodes** ook mogelijk zijn om:

Method	Explanation
Is_age()	Returns age of family member
is_sex()	Returns sex of family member
birthday()	Wish family member a happy birthday, it's age should increase.
related()	Explain relation between family member and yourself
paint_hair()	Change hair color of family member

Maak een aantal objecten van je Family (minstens 4). Probeer de methodes uit en kijk of ze werken.

Opdracht 1.2 Car (1.3.4)

Maak een klasse Car. Je moet met behulp van deze klasse, objecten van auto's aanmaken. Iedere auto met volgende variabelen bezitten:

Variabel	Explanation
brand	Brand of car
mileage	How many kilometers can you drive per liter of fuel
color	Color of car
max_tank	Maximum liter fuel the tank can hold
current_tank	Current tank of fuel, should start equal to max_tank

Jij bepaalt zelf of het opslaan van deze data best als **attribuut** of **eigenschap** gebeurt.

Hiernaast moet het via **methodes** ook mogelijk zijn om:

Method	Explanation
paint_job()	Change color of car
check_tank()	Return current_tank and how many kilometers you can still drive
praise()	Give a compliment about the brand of the car
fill()	Return current_tank to maximum value
drive()	Drive a variabele number of kilometers: • current_tank should decrease according to mileage • if current_tank is empty before arriving, return driven distance

Maak een aantal objecten van Car (minstens 4). Probeer de methodes uit en kijk of ze werken.

Opdracht 1.3 Raspberry Pi LED (1.3.4)

A. Ga naar de 6IICT-PROG4 Github Repository en haal het bestand "Opdracht 1.3 led_normal" af. Bouw aan de hand van de code een schakeling op waarbij afwisselend twee rode en twee groene LED's branden. Gebruik voor deze opdracht de Raspberry Pi

B. Maak een klasse LED. Deze bezit drie methodes __init__(), on() en off().__init__() heeft als argument een cijfer. Dit is de pin die in output-mode moet worden gezet. De on() methode moet dezelfde pin hoog maken. De off() methode moet dezelfde pin laag maken.

C. Maak vier objecten van LED. Twee rode en twee groene (naam vrij te kiezen). Gebruik deze om de schakeling te laten werken zoals in deel **A**.

Opdracht 1.4 Raspberry Pi Button (1.3.4)

A. Stel code op die voor twee knoppen een boodschap eenmalig print als de knop ingedrukt wordt. Deze boodschap is "Pin X geactiveerd". Hierbij is X de pin waarop de knop aangesloten is. Bij het loslaten van de knop moet eenmalig de boodschap "Pin X gedeactiveerd" verschijnen.

B. Maak een klasse Button. Deze heeft drie methodes __init__(), is_pressed() en is_released(). __init__() heeft als argument een cijfer. Dit is de pin die in input-mode moet worden gezet. De is_pressed() methode returnt True enkel bij het indrukken van de knop, False op ieder moment hiervoor en erna. De is_released() methode returnt True enkel bij het loslaten van de knop, False op ieder moment hiervoor en erna.

C. Maak twee objecten van Button (naam vrij te kiezen). Gebruik deze om de schakeling te laten werken zoals in deel **A**.

Opdracht 1.5 Finance Tracker (1.3.4)

Maak een Finance Tracker app. Ga naar de 6IICT-PROG4 Github Repository en haal de code in de map Finance Tracker af. De uitleg voor deze opdracht is terug te vinden in de README.md. Deze is het makkelijkst te lezen op Github.

Opdracht 1.6 Rectangle (1.3.5)

A. Creëer een klasse Rectangle. Deze heeft als eigenschappen width en height. Maak de eigenschappen "veilig" door te garanderen dat zowel breedte als hoogte positieve waardes zijn (hoe je dat doet laat ik aan jou over). Breid de klasse uit met methodes die de oppervlakte en omtrek berekenen.

B. Voeg vervolgens een eigenschap point toe. Deze eigenschap behoort op zijn beurt tot de klasse Point en bezit de eigenschappen x en y (coördinaten). Het object point verwijst naar de coördinaten van de linkerbovenhoek van een rechthoek. Schrijf ook een methode die de rechteronderhoek van een rechthoek returnt als een object van de klasse Point.

C. Creëer tenslotte een methode die een tweede rechthoek als parameter meekrijgt, en die het deel waar de twee rechthoeken overlappen retourneert als een rechthoek (deze laatste methode is veel moeilijker te schrijven dan de andere).

Opdracht 1.7 Student and course (1.3.5)

A. Een student heeft een voornaam, achternaam, geboortedatum (geformateerd als YYYY/MM/DD) en een administratienummer (s gevolgd door een willekeurig cijfer). Een cursus heeft een naam en een nummer (c gevolgd door een willekeurig cijfer). Creëer een klasse Student en klasse Course.

B. Creëer vervolgens minstens vijf studenten en drie cursussen (kan je hiervoor een for_lus gebruiken en op welke manier sla je al deze objecten dan op?).

C. Voeg een methode aan de klasse Student toe. Deze heeft als parameter een cursus. Schrijf de student voor deze cursus in (Moet je hiervoor een eigenschap aan Student of Course toevoegen?). Voeg tenslotte een methode aan de klasse Student toe die volgende gegevens van de student print:

- voornaam;
- achternaam;
- leeftijd (**NIET GEBOORTEDATUM)**
- naam van iedere ingeschreven cursus.

1.4 OOP – Inheritance

1.4.1 Het begrip gedeelde klasse (Inheritance of Erving)

Dog en Cat zijn zeer gelijkaardige dieren. Zo hebben ze beide een leeftijd, een naam, 4 poten, enz. Het lijkt dan ook gek dat we hiervoor twee verschillende klassen aanmaken. Het zou fijn zijn als deze klassen gebruik konden maken van een **gedeelde klasse**. Zo zouden ze alle Attributen, Eigenschappen en Methodes die ze gelijk hadden, kunnen overnemen van deze gedeelde klasse. Dit is mogelijk met behulp **Inheritance of Erving**. We definiëren hier twee type klassen.

- **Parent (super) class**: De klasse waarvan geërfd wordt.
- **Child (sub) class:** De klasse die erft van een andere klasse.

Honden en katten zijn beide zoogdieren. We kunnen dus een klasse Mammal maken en de klasse Cat hiervan laten erven. Dit gebeurt door in de ronde haakjes van Cat, Mammal mee te geven. In het algemeen gebeurt erving als volgt:

class <kind_klasse>(<ouder_klasse>)>

```
class Mammal():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def has_name(self):
        print(self.name)

x = Mammal("George", 10)
x.has_name()

""" Cat inherits everything from Mammal. Nothing new is added to Cat """
class Cat(Mammal):
    pass

y = Cat("Michael", 7)
y.has_name()
```

Oefen mee

Neem de klasse Mammal uit bovenstaand voorbeeld over. Voeg een klasse Dog toe en laat deze erven van Mammal. Maak vervolgens een object van Dog en print zijn naam.

1.4.2 De Child class

Momenteel is de klasse Cat identiek aan de klasse Mammal. Dit is natuurlijk niet nuttig. We kunnen **__init__()** toevoegen aan de klasse Cat om nieuwe properties toe te voegen. Dit overschrijft echter de **__init__()** van de klasse Mammal, waardoor deze nutteloos wordt. We lossen dit op door Mammal.__init__() op te roepen in de __init__() van Cat.

```
class Mammal():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def has_name(self):
        print(self.name)
```

```
x = Mammal("George", 10)
x.has_name()

class Cat(Mammal):
    def __init__(self, name, age):
        """ Mammal kan ook vervangen worden door super().
            self moet dan niet als parameter worden meegegeven. """
        Mammal.__init__(self, name, age)

y = Cat("Michael", 7)
y.has_name()
```

Het valt misschien op dat hier de methode __init__() rechtstreeks via de klasse is opgeroepen. Dit kan ook met de andere methodes van de klasse Mammal.

<klasse>.<methode (<parameters>)>

Dit is echter enkel mogelijk in de klasse Cat omdat deze erft van Mammal. Je kan dit zelf proberen door buiten de klasse Cat, de methode Mammal.has_name() uit te voeren.

Oefen mee

Pas de klasse Dog aan zodat deze zijn eigen __init__() heeft, maar nog altijd de eigenschappen toegewezen in Mammal behoudt. Voeg ook de property “race” toe. Maak vervolgens een object van Dog en print “name” en “race” van minsten twee objecten.

Het is ook mogelijk om methodes toe te voegen aan de Child class. Deze zullen enkel beschikbaar zijn voor objecten van de Child. Net zoals met __init__() is het ook mogelijk om andere methodes in de Child class toe te voegen met dezelfde naam als deze in de Parent class. De methode in de Parent class wordt dan wel overschreven.

```
class Mammal():
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def has_name(self):
        print(self.name)

class Cat(Mammal):
    def __init__(self, name, age, color):
        Mammal.__init__(self, name, age)
        self.color = color

    def has_color(self):
        print (f"{self.name} is {self.color}")

    def has_name(self):
        print (f"My cat is called {self.name}")


x = Mammal("George", 10)
x.has_name()
y = Cat("Michael", 7, "grey")
```

```
y.has_color()
y.has_name()
```

Oefen mee

Pas de klasse Dog aan zodat deze bij het oproepen van de methode has_name() eerst “My dog is called “ gevolgd door zijn naam returnt (Dus niet print!). Voeg vervolgens een nieuwe methode is_race() toe. Deze returnt een string met “My dog is a “ gevolg door zijn ras. Test deze methodes met minstens twee objecten.

1.4.3 Meervoudige inheritance

Het is mogelijk om een klasse te maken die van meerdere klassen erft. Dit gebeurt door meerdere “ouders tussen de ronde haakjes van de kind klasse te schrijven.

class <kind_klasse>(<ouder_klasse_1, ouder_klasse_2, …>)>

Als een object van deze kind klasse een methode oproept, zal Python als volgt werken. Het zoekt eerst de methode in de kind klasse. Bestaat deze hier niet, dan worden alle ouder klassen gecontroleerd, van links naar rechts (ouder_klasse_1 → ouder_klasse_n). Zodra de methode gevonden wordt, stopt de zoektocht en wordt deze methode uitgevoerd. Denk vooraleer onderstaande code uit te voeren eerst na over uit welke klasse de drie functies onderaan zullen komen.

```
class Mammal():
    def __init__(self, name, age):
        print(f"__init__() of Mammal used")
        self.name = name
        self.age = age

    def sound(self):
        print(f"{self.name} makes mammal sound")

class Baby():
    def __init__(self, name, sex, months_old):
        print(f"__init__() of Baby used")
        self.name = name
        self.sex = sex
        self.months_old = months_old

    def has_name(self):
        print(f"The baby has name {self.name}")

class Kitten(Mammal, Baby):
    def sound(self):
        print(f"{self.name} miauws")

kitten = Kitten("Borysz", 0.5) # Calls __init__()
kitten.has_name()
kitten.sound()
```

Oefen mee

Neem onderstaande code over. Voeg de methode has_name() toe aan de klasse Dog en Job. Voeg de methode is_age() toe aan Mammal en has_experience() aan Job. Voeg tenslotte . Je kiest zelf wat deze

methodes doen wanneer opgeroepen. Maak een object van de klasse Dog en roep een aantal methodes op. Denk eerst zelf na over uit welke klasse de opgeroepen methode komt.

Wissel de volgorde van inheritance **Dog(Job, Mammal)**. Vergeet niet om de parameters van het object van Dog aan te passen (Deze inherit de __init__() nu van Job). Wat verandert er nu aan de opgeroepen methodes? Er zal een probleem ontstaan met de methode has_name(). Los dit probleem op door aanpassingen in de klasse Dog. Er zijn hiervoor verschillende oplossingen.

Meervoudige inheritance wordt snel verwarrend en moeilijk. Ik raad daarom aan om het enkel te gebruiken indien het echt niet anders kan. Veel andere programmeertalen ondersteunen meervoudige inheritance niet, net omdat er veel fouten door ontstaan.

1.4.4 Oefeningen

Opdracht 1.8 Vehicle

A. Begin van onderstaande klasse Vehicle. Maak een klasse Bus die erft van Vehicle. Bus heeft dezelfde eigenschappen als Vehicle. Maar het aantal zitplaatsen (seats) heeft een default waarde van 50. Bus heeft ook een attribuut. Ieder object van Bus is namelijk van het bedrijf “De Lijn”.

```
class Vehicle:
    def __init__(self, brand, max_speed, mileage, seats):
        self.brand = brand
        self.max_speed = max_speed
        self.mileage = mileage
        self.seats = seats
```

B. Voeg de ticket() methode toe aan Vehicle. Deze methode returnt de kost om met dit voertuig te rijden. De kost is gelijk aan 5 gedeeld door het aantal zitplaatsen. Bij objecten van de klasse Bus moet een andere formule gebruikt worden. In bus moet aan de ticket() methode een parameter distance (km) worden meegegeven. Dit is voor welke afstand een persoon met de bus rijdt. De kost is nu gelijk aan 5 gedeeld door het aantal zitplaatsen maal de gereden afstand.

Opdracht 1.9 Rectangle and Square

A. Maak een klasse Rectangle. Deze heeft als eigenschappen: het x- en y-coördinaat van de linkerbovenhoek, en de width (w) en height (h) van de rechthoek. Rectangle heeft hiernaast ook nog enkele methodes. De methode area() returnt de oppervlakte, de methode perimeter() returnt de omtrek, tenslotte veranderen de methodes set_width() en set_height() respectievelijk de breedte en hoogte.

B. Creëer nu een klasse Square die erft van Rectangle. Bij objecten van Square moeten de methodes set_width() en set_height() zowel de breedte als de hoogte veranderen. Andere methodes blijven ongewijzigd.

Opdracht 1.10 Shape

Kopieer de code uit **Opdracht 1.9 Rectangle and Square** als startpunt van deze opdracht.

A. Rectangle en Square zijn vormen. Er zijn er echter nog een hoop meer. Maak een class Shape, waarvan Rectangle erft (en Square dus een kleinkind is). Shape heeft als eigenschappen: het x- en y-coördinaat van een punt van de vorm. Shape moet de methodes area() en perimeter() hebben. Deze methodes moeten niets doen (gebruik pass). Pas de klasse Rectangle aan zodat deze de __init__() van Shape gebruikt.

B. Maak een klasse Circle. Deze moet erven van Shape. Voeg in de klasse Circle de methodes area() en perimeter() toe. Voeg ook de methode __init__() toe, hierin moeten alle eigenschap(pen) die nodig zijn om de oppervlakte en omtrek komen, evenals de eigenschappen uit Shape.

De klasse Shape voegt weinig toe aan het programma. Dit komt omdat het een zogenaamde interface klasse is. Dit soort klassen worden gebruikt om eigenschappen en methodes vast te leggen zonder hier effectief iets mee te doen. Door andere klassen hiervan te laten erven weten de ontwikkelaars wat er aan de kinderen toegevoegd moet worden. In het geval van Shape weten we dat alle kinderen een methode moeten hebben om de oppervlakte en omtrek te berekenen. Interfaces zijn vooral handig wanneer je met meerdere mensen aan een project samenwerkt.

Opdracht 1.11 Shape Fitter

Maak een Shape Fitter app. Ga naar de 6IICT-PROG4 Github Repository en haal de code in de map Area calculator af. De uitleg voor deze opdracht is terug te vinden in de README.md. Deze is het makkelijkst te lezen op Github. Je kan voor het eerste deel van de code teruggrijpen naar **Opdracht 1.9 Rectangle and Square**.

Opdracht 1.12 Coder Dojo

Maak een Coder Dojo. Ga naar de 6IICT-PROG4 Github Repository en haal de code in de map Coder Dojo af. De uitleg voor deze opdracht is terug te vinden in de README.md. Deze is het makkelijkst te lezen op Github.

1.5 Klassen en data types in Python

De meeste object-georiënteerde programmeertalen kennen een aantal basale data types (int, float, char, String, list, …). Hiernaast staan ze je toe om klassen te definiëren, wat neerkomt op het definiëren van nieuwe data types. Dit gold ook voor Python tot en met versie 2. Sinds Python 3 is echter ieder data type een klasse. Dit wilt zeggen dat de data types int, string, list, enzoverder allemaal klassen zijn. Variabelen van data types zijn dus objecten.

Dit klinkt misschien gek, maar in werkelijkheid ben je reeds geruime tijd bezig met OOP. Herinner je hoe de methode van een bepaald object op te roepen?

<object>.<methode (<parameters>)>

Stel nu dat we een variabele naam hebben. Dit is een string met erin “Borysz”. We kunnen deze variabele omvormen naar enkel kleine letters via .lower().

naam.lower()

lower() is dus een methode van de klasse String. Net zoals istitle(), strip() en nog vele andere.

Zelfs objecten van de klasse int en float hebben methodes. Deze worden echter zelden expliciet opgeroepen. Echter als je twee getallen optelt met + zal dit achter de schermen wel via een methode de twee getallen optellen!

Je kan dit ook zelf controleren. De functie type() zal de klasse teruggeven van de variabele die als parameter is meegegeven. Voer onderstaande code uit.

```
integer = 0

string = "Woord"

boolean = True

lijst = [integer, string, boolean]

class Klasse:
  def __init__(self, integer, string, boolean):
    self.integer = integer
    self.boolean = boolean
    self.string = string
klasse = Klasse(integer, string, boolean)

print(type(integer))
print(type(string))
print(type(boolean))
print(type(lijst))
print(type(klasse))
```

Hopelijk valt op dat lijst, ondanks dat deze bestaan uit verschillende variabelen, toch zijn eigen klasse is. Dit is ook nodig. Denk eens aan alle built-in methodes die python heeft voor lijsten. Indien lijsten niet hun eigen klasse hadden, zouden we al deze functionaliteiten niet kunnen gebruiken.

1.6 OOP in Arduino

Arduino gebruikt een aangepaste versie van C++. Met deze taal is OOP ook mogelijk. Een groot voordeel van OOP is dat het mogelijk is om code te hergebruiken. Zo kunnen we een klasse Led maken. Deze zorgt dat het makkelijk wordt om veel verschillende LED's aan te sturen zonder dat het aantal regels code sterk verhoogt.

```
class Led {
  private:
    byte pin;
  public:
    // equal to __init__()
    Led(byte pin) {
      // "this" is analogue of "self"
      this->pin = pin;
      pinMode(pin, OUTPUT);
      digitalWrite(pin, LOW);
    }
    // turn led on
    void on() {
      digitalWrite(pin, HIGH);
    }
    // turn led off
    void off() {
      digitalWrite(pin, LOW);
    }
};

// Create your objects in the global scope so you can
// get access to them in the setup() and loop() functions
Led led1(9);
Led led2(10);
Led led3(11);
Led led4(12);

// setup() is left empty since setup is done during creation of object.
void setup() { }

void loop() {
    // Code is abstracted. Making it easier to read.
    // The "difficult" parts are hidden in the class.
    led1.on();
    led2.off();
    led3.on();
    led4.off();
    delay(1000);

    led1.off();
    led2.on();
    led3.off();
    led4.on():
    delay(1000);
}
```

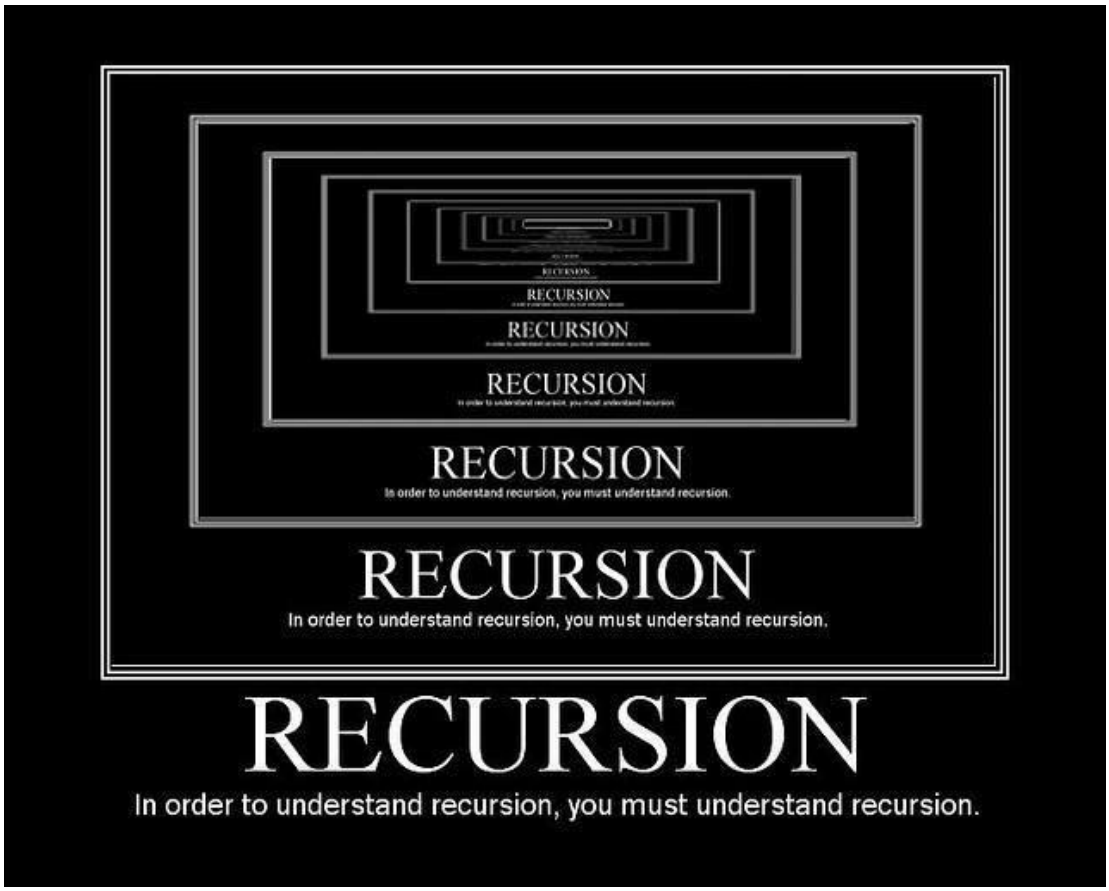
Hopelijk valt op dat deze code zeer analoog is aan degene van Opdracht 1.3 Raspberry Pi LED.

2 Recursie

Recursie is een speciale techniek die ontstaat doordat het mogelijk is om in een functie een oproep te maken naar deze functie. Recursie kan bepaalde problemen op een elegante en efficiënte manier oplossen. Het is echter ook iets waar veel studenten in het begin moeite mee hebben. Daarom zal er een apart hoofdstuk aan gewijd worden.

2.1 Wat is recursie?

Een functie oproepen in de functie zelf. Dat is “alles” wat recursie is. Het klinkt misschien vreemd in het begin. Maar er is niks op tegen dat een functie andere functies oproept. Een functie mag iedere functie oproepen die reeds gedefinieerd is. Omdat een functie gedefinieerd moet zijn voordat jij hem van buitenaf aanroept, kan hij dus ook zichzelf aanroepen.



Oefen mee

Vooraleer onderstaande code uit te voeren. Kijk eerst de code na. Welke getallen print deze functie wanneer gestart (van buitenaf opgeroepen)? Voer vervolgens de functies uit.

Zonder recursie

```
""" Definiëren van functie """
def print_number_plus_two(x):
    x=x+2
    print(x)

""" Oproep van functie (buitenaf) """
print_number_plus_two(0)
```

Met recursie

```
""" Definiëren van functie """
def print_number_plus_two(x):
    x=x+2
    print(x)

    """ oproep van functie (binnenin) """
    print_number_plus_two(x)

""" Oproep van functie (buitenaf) """
print_number_plus_two(0)
```

Oefen mee

Zoals opgemerkt is er bij recursie momenteel een probleem. Denk eens even na wat dit probleem exact inhoudt. Wat is een mogelijke oplossing voor dit probleem?

Deze demonstratie toont een van de grootste gevaren van een recursieve functie. Slordig in elkaar gestoken, kan deze zichzelf in theorie oneindig keren oproepen. In praktijk zal Python nadat deze zichzelf 1000 heeft opgeroepen een "RecursionError" geven (voor meer info zie …) . Het mag dus duidelijk zijn dat het nodig is om recursieve functies zo te ontwerpen dat dit nooit gebeurt.

Het valt misschien op dat recursie veel weg heeft van een while-lus. Net als bij een while-lus, zal een recursieve functie het blok code erin blijven uitvoeren. Dit totdat er aan een bepaalde voorwaarde voldaan is. Voor bovenstaand voorbeeld is ook een implementatie met while-lus mogelijk.

```
""" Definiëren van functie """
def print_number_plus_two(x):
    while True:
        x=x+2
        print(x)

""" Oproep van functie (buitenaf) """
print_number_plus_two(0)
```

Er zijn veel problemen waarvoor recursie een elegante oplossing biedt. Daarom is het belangrijk dat je je bewust bent van de mogelijkheden van recursie, en dat je weet hoe je het kunt toepassen. Maar in een groot deel van de gevallen is een analoge oplossing op met while-lussen mogelijk. Hieronder alvast enkele voordelen van zowel recursie als while-lussen.

While-lus	Recursie
Kan sneller zijn (Bij slechte optimalisatie recursie)	Verhoogt duidelijkheid, minder code nodig
Gebruikt minder geheugen	Vaak lagere tijdscomplexiteit
Makkelijker te ontwerpen	Geeft error bij oneindige "lus"

Wanneer welke te gebruiken is niet altijd duidelijk. Meestal is de while-lus geprefereerd. Implementeer enkel een recursieve implementatie als:

- recursie de meest natuurlijke manier lijkt om het probleem op te lossen;
- het recursieve proces de functie niet te vaak in zichzelf oproept.

Deze "regels" zijn vaag omschreven. Het is dan ook de verantwoordelijk van de programmeur om, in functie van de oplossing, te bepalen of deze regels voldaan zijn.

2.2 Recursieve definities

Een wiskundige operatie waar recursie mogelijk is, is de faculteit. Wat volgt is de algemene definitie:

De faculteit van een natuurlijk getal n, genoteerd als n!, is het product van de getallen n tot en met 1.

$$\boldsymbol{n! = n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1}$$

Voor bijvoorbeeld n = 5 is de faculteit van n.

$$\boldsymbol{5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120}$$

Beschouw in dit geval de faculteit (!) als een functie die uitgevoerd wordt. Deze heeft in het voorbeeld een input 5 en als return-waarde het getal 120. Het is mogelijk om deze functie als volgt op te stellen.

```
def factorial_while(n):
    value = 1
    while n > 0:
        value = value*n
        n = n-1
    return value

print( factorial_while(5) )
```

Naast de algemene definitie is er echter ook een zogenaamde recursieve definitie.

$$\boldsymbol{1! = 1}\ \textit{en}\ \boldsymbol{n! = n \cdot (n-1)!}$$

Deze definitie is recursief omdat het refereert naar de faculteit van (n-1). Deze kan op zijn beurt weer refereren naar de faculteit van (n-2). Deze op zijn beurt naar … . Dit gaat zo door tot de faculteit van 1 bereikt is. Dit is volgens de definitie gewoon gelijk aan 1. Dit implementeren gebeurt als volgt.

```
def factorial_recursion(n):
    if n <= 1:
        return 1
    return n*factorial_recursion(n-1)

print ( factorial_recursion(5) )
```

Deze functie volgt exact de recursieve definitie van een faculteit. Als n gelijk is aan 1, returnt de functie 1. Anders returnt het n maal de faculteit van (n-1). Hierbij is <= gebruikt zodat ingeven van negatieve getallen geen RecursionError oplevert.

Het is in het begin moeilijk om te begrijpen hoe een recursieve functie werkt. Daarom zal ik het ook anders uitleggen. Het is namelijk mogelijk om dit proces voor te stellen als een aantal borden die je op elkaar stapelt in een kast. Het onderste bord (functies) wordt als eerste gestapeld (opgeroepen). Het bord erboven, kon pas gestapeld worden na het eerste bord. De borden blijven zich opstapelen, totdat de borden op zijn (er aan een voorwaarde voldaan is). Het doel van recursie is nu het eruit halen (returnen) van het onderste bord. Hiervoor moeten alle andere borden dus ook eruit. Eerst wordt het bovenste bord eruit gehaald, vervolgens het op een na bovenste, enzoverder. Een video van deze beschrijving is ook te vinden op smartschool onder de naam “recursion_visualisation”.

2.3 Recursie toepassen

In het begin is het niet evident om recursieve oplossingen te bedenken. Er is een strategie ontwikkeld om beginnelingen hierbij te helpen. Deze strategie omvat vier stappen. Deze zullen overlopen worden aan de hand van sommatie. Ga ervanuit dat het getal i niet kleiner dan nul kan beginnen.

$$\sum_{i=0}^{n} i = n + (n+1) + \cdots + 1 + 0$$

Hiervoor is ook een recursieve definitie.

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$$

Stap 0: Nodige input

Deze stap is geen onderdeel van de strategie. Ik vermeld het voor de volledigheid. Begin met het maken van het geraamte van de functie. Denk hierbij na over welke input deze functie nodig heeft. De functie moet de sommatie van 0 tot en met n returnen. De input n lijkt daarom ook logisch.

```
def sum(n):
    pass

""" 5 chosen randomly as input """
print( sum(5) )
```

Stap 1: Wat is de simpelste input

De simpelste input is degene waarbij de functie zichzelf zo weinig mogelijk moet oproepen. Stel dat n gelijk is aan 5.

$$\sum_{i=0}^{5} i = 5 + 4 + 3 + 2 + 1 + 0$$

Het cijfer vijf komt van buitenaf. De cijfers 4, 3, 2, 1 en 0 van bunnen. De recursie gaat hier dus vijf niveaus diep. Stel dat n gelijk is aan 3.

$$\sum_{i=0}^{3} i = 3 + 2 + 1 + 0$$

De recursie gaat nu drie niveaus diep. Het lijkt dus wel duidelijk dat de simpelste input 0 is. Hierbij zal de functie zichzelf niet eens moeten oproepen.

$$\sum_{i=0}^{0} i = 0$$

Dit wordt de “basissituatie”, oftewel het diepste punt van de recursie. Met andere woorden moet bij dit getal de functie zichzelf niet meer oproepen. Bepaal de output (of return-waarde) van deze base case. In dit geval is de sommatie van 0 gelijk aan 0. Voeg dit specifiek geval van sommatie toe aan de functie.

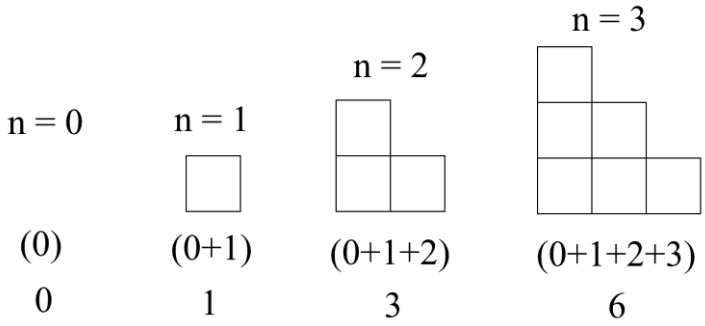
```
def sum(n):
    if n == 0:
        return 0

""" 5 chosen randomly as input """
print( sum(5) )
```

Stap 2: Werk een aantal voorbeelden

Het is belangrijk om een verband te vinden tussen de verschillende situaties. Hiervoor is het nodig om een aantal van deze situaties uit te werken. Doe dit eventueel visueel. Hierdoor is het eenvoudiger om verbanden te zien. Krabbelen op een kladblok is aangeraden!

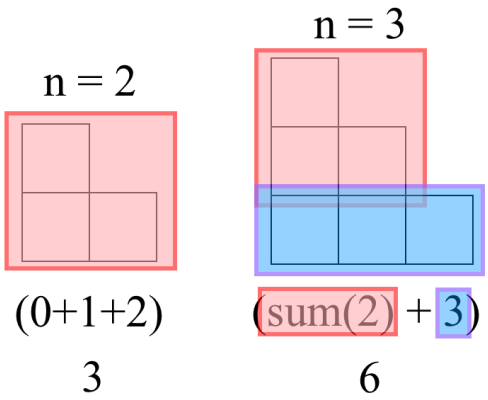
Sommatie kan gemakkelijk voorgesteld worden. Hier een voorbeeld voor n gelijk aan 0 tot en met 3.



Stap 3: Vind verband tussen voorbeelden

Probeer op basis van deze voorbeelden een verband op te stellen. In dit voorbeeld stel de vraag: “Als ik sum(0) gegeven heb, kan ik dan sum(1) zelf bepalen?”. Herhaal dit voor sum(1) en sum(2), sum(2) en sum(3), enzoverder.

Bijvoorbeeld voor het bepalen van sum(3) indien sum(2) gegeven is. Het is mogelijk om de kubussen van sum(2) op deze van sum(3) te plaatsen (rood). Het aantal niet ingevulde kubussen (blauw) is gelijk aan 3. Herhaal dit een aantal keer. Het patroon zal zich herhalen.



Stap 4: Generaliseer het patroon

Nu het verband tussen de voorbeelden duidelijk is, is het tijd om dit als een formule te noteren. Deze formule is hierboven al eens wiskundig geschreven (recursieve definitie). Ik doe het hier nogmaals, maar dan met de functienaam.

sum(n) = n + sum(n-1)

Bovenstaande regel betekent dus dat sum(n) als return-waarde n+sum(n-1) heeft. Vul de code aan.

```
def sum(n):
    if n == 0:
        return 0
    return n + sum(n-1)

""" 5 chosen randomly as input """
print( sum(5) )
```

Hopelijk valt wel op dat 0 **Stap 3: Vind verband tussen voorbeelden** en 0 **Stap 4: Generaliseer het patroon** in dit geval zeer eenvoudig zijn. Dit omdat de oefening zelf eenvoudig is. De strategie is zeker bruikbaar voor grotere/complexere toepassingen, maar vraagt dan wel meer tijd.

2.4 Oefeningen

Opdracht 2.1 Fibonacci

A. Het volgende getal in de reeks van Fibonacci is gelijk aan de som van de twee vorige. Schrijf een recursieve functie met als input een getal n. De functie returnt het n-de Fibonacci getal. Een recursieve definitie van het n-de Fibonacci getal fib(n) is als volgt.

fib(1) = 1, fib(2) = 1 en fib(n) = fib(n-1)+fib(n-2)

B. Om meer inzicht te krijgen in recursie. Pas de Fibonacci functie aan zodanig dat deze de “diepte” van recursie bijhoudt. Maak een diepte-parameter, startend bij 0. Verhoog deze met 1 bij het oproepen van de Fibonnaci-functie. Print deze diepte en het huidige getal n. Print ook de gereturnde waarde. Debug nu de code. Gebruik als breakpoint de diepte parameter. Bestudeer de output.

C. Maak de Fibonnaci-functie nu met behulp van een while-lus. Welke implementatie vind je het makkelijkst of moeilijkst? Waarom?

Opdracht 2.2 Common denominator

Opdracht 2.3 Towers of Hanoi

Opdracht 2.4 Game of Life