Python OO Programming

Les 1: OO basis

Kristof Michiels

Introductie tot object-georiënteerd programmeren

- Voorbeeld procedureel programmeren
- Introductie klassen als de basis van het schrijven van OO-code
- Relatie tussen klasse en object
- Elementen van een klasse en hoe functioneren ze samen?
- Herschrijven procedureel voorbeeld als klasse
- Andere voorbeelden
- Labo-oefening

Inleiding

Eerst iets over mezelf

- Voor jullie een nieuwe docent
- Niet nieuw binnen AP
- Lesgegeven binnen GDM
- Mijn profiel
- Mijn zonen: Eli 11j (Arduino-adept) en Kai (16j, Discord Bot programmeur)

Hoe wil ik het aanpakken? Mijn filosofie

- Jullie maakten een prachtige keuze qua opleiding
- Onze technologische wereld
- Hoe leert iemand programmeren?
- Het is geen race. Kan soms zo aanvoelen in een schoolcontext ;-)
- Zorg dat je er mee bezig bent
- Kijk rond je heen!
- Ruimte voor experiment
- De wapenwedloop docent-student

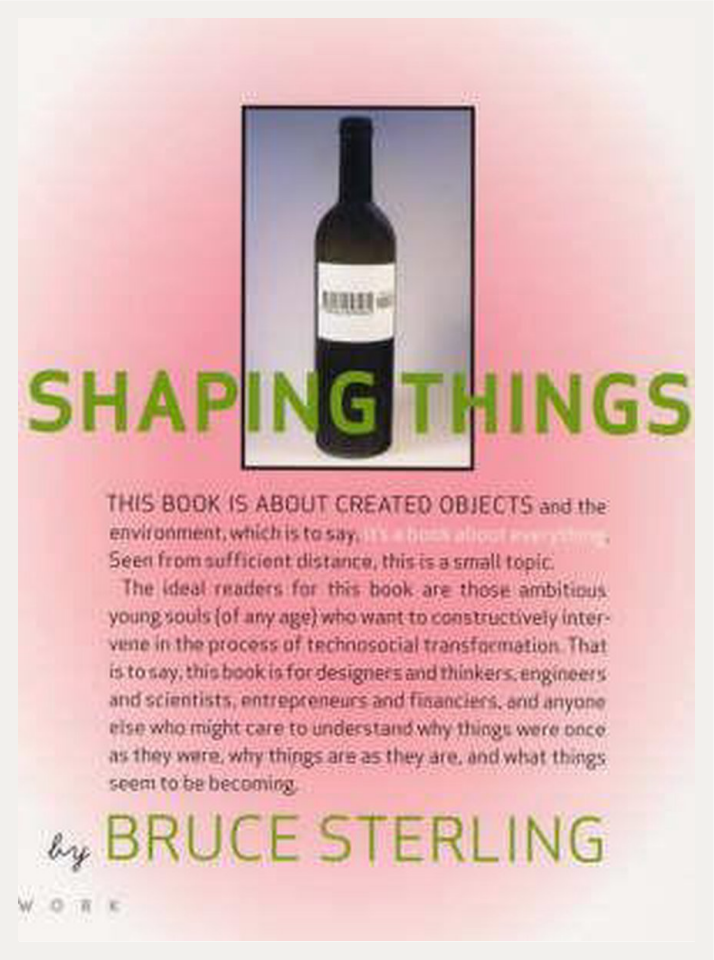
Python programming in eerste jaar

- Ik heb de lesmaterialen van vorig jaar geanalyseerd
- Ik zal dit vak ook geven aan de eerstejaars in het eerste semester
- Ik geef in semester 2 ook Python Web services
- Niet evident: nieuwe docent, buitengewone lessituatie voorbije jaar
- Daarom volgende afspraak:
  - laat gerust weten of er elementen zijn die jullie graag nog eens uitgelegd krijgen
  - ik ga ook luisteren welk soort invulling qua oefeningen jullie graag willen
  - Ik kan - indien gewenst - elke week een klein luik herhaling toevoegen

Wat gaan we dit jaar doen?

- Uiteraard Python OO programmeren
- Werken met tal van libraries, modules
- IoT / Hardware
- Links met AI aftasten

Bruce Sterling: Shaping Things



Sterling noemt de objecten van de toekomst "Spimes"

Onze tools

- Pycharm zou mijn voorkeur wegdragen
- Visual Studio Code?

Object-georiënteerd programmeren

Object-Georiënteerd programmeren

- Keuze binnen een programmeertaal die mogelijk maakt om variabelen en functies te groeperen
- Dat groeperen gebeurt binnen een data type dat we een klasse noemen
- Vanuit een klasse kan je objecten creëren
- Door je code te organiseren binnen klasses kan je je programma opdelen in kleinere onderdelen die eenvoudiger te begrijpen en te debuggen zijn

Object-Georiënteerd programmeren

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

Software modellen bouwen van fysieke objecten

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

Software modellen bouwen van fysieke objecten

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

State en behavior: een procedureel voorbeeld

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

def zetUit():
    global schakelaarIsAan
    schakelaarIsAan = False

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

State en *behavior*: een procedureel voorbeeld

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

Richting OO...

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

Maar eerst: een beetje theorie

De relatie tussen een klasse en een object

- Een klasse (class) is een template/blauwdruk/mal van een object
- Vanuit die klasse zullen we een object creëren
- We definiëren een klasse als: code die beschrijft wat een object zal herinneren (data of *state*) en de zaken die het object zal kunnen doen (de functies of *behavior*)

De relatie tussen een klasse en een object

Een bakvorm voor cake kan je vergelijken met een klasse...

Het procedureel voorbeeld herschreven

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

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

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

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

Het procedureel voorbeeld herschreven

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

Instantiëren / *instance*

- Een *instance* en object hebben feitelijk dezelfde betekenis
- Heel precies: het oLichtSchakelaar-object is een instance van de LichtSchakelaar-klasse
- Instantiëren: het proces van het creëren van een object uit een klasse
- We gingen in onze code door het instantiëringsproces om een LichtSchakelaar-object te creëren uit de LichtSchakelaar-klasse

Een klasse schrijven in Python

```
class <KlasseNaam>():
    def __init__(self, <optionele param1>, ..., <optionele paramN>):
        # extra initialiseringscode

    # Alle functies nodig om toegang te krijgen tot de data
    # Elke heeft de volgende gedaante:
    def <functieNaam1>(self, <optionele param1>, ..., <optionele paramN>):
        # functie-gerelateerde statements

    # ... meer functies
    def <functieNaamN>(self, <optionele param1>, ..., <optionele paramN>):
        # functie-gerelateerde statements
```

Onderdelen van een klasse

- Je begint met een class statement met de naam die je de klasse wil geven
  - Conventie: camel case, met de eerste letter in hoofdletters
  - Haakjes zijn optioneel (zet ze toch maar, worden gebruikt bij overerving)
  - Dubbelpunt op het einde is verplicht (geeft Python mee dat je de klasse zal beginnen te beschrijven)
- Binnen de klasse voegen we zoveel functies toe als we nodig hebben
  - Code binnen de functies: insprongen!
  - Elke functie vertegenwoordigt gedrag dat het object kan vertonen
  - Alle functies hebben minstens één parameter (self)
  - We noemen OOP functies ook *methods*

Onderdelen van een klasse

- Eerste *method* in elke klasse heeft de speciale naam __init__
- Elke keer je een object uit een klasse instantieert zal deze method automatisch uitgevoerd worden
- Daarom is deze method de logische plek om initialisatiecode te plaatsen
- Niet verplicht om te gebruiken, maar is een best practice om dit steeds te doen
- We noemen dit ook de initialisatie-method, waar je variabelen startwaarden geeft

Scope en instance variabelen

- In procedureel programmeren zijn er twee belangrijke scope levels: global en local
- Local scope gecreëerd binnen een functie: wanneer de functie eindigt, verdwijnen de lokale variabelen
- OOP creëert een derde niveau: object scope (ook wel class scope of instance scope genoemgd)
- Deze scope omvat alle code in de klasse-definitie
- Methods kunnen zowel local als instance variabelen hebben: elke variabele die niet met self. begint is een lokale variabele
- Deze lokale variabelen eindigen met het eindigen van het uitvoeren van de method: andere methods kunnen ze dus niet gebruiken
- Heel belangrijk om goed te begrijpen hoe objecten data bijhouden

Scope en instance variabelen

- Instance variabelen worden aangemaakt wanneer ze eerst een waarde worden toegekend
- Ze hebben geen speciale declaratie nodig
- De __init__ method is de logische plaats om instance variabelen te initialiseren
- Elk object krijgt zijn eigen set van instance variabelen, onafhankelijk van andere objecten die uit dezelfde klasse worden gecreëerd

```
class MijnKlasse():
    def __init__(self):
        self.teller = 0
    def vermeerderen(self):
        self.teller = self.teller + 1
```

Een object creëren uit een klasse

- Om een klasse te gebruiken vraag je Python om een object te creëren uit die klasse
- Je doet dit, zoals we reeds zagen met volgend toekenningsstatement:

```
<object> = <klassenaam>(optionele argumenten)
```

- Deze statement invokeert een opeenvolging van stappen die eindigen met het toekennen van een nieuwe instance van die klasse aan de variabele die in het statement is meegegeven

Een object creëren uit een klasse

- Het proces bestaat uit 5 stappen:
  - Onze code vraagt Python om een object uit een klasse te creëren
  - Python alloceert geheugen voor een object en voert de __init__() method uit
  - De __init__() method loopt en het nieuwe object wordt toegekend aan de parameter self. Alle instance variabelen worden geïnitialiseerd in het object
  - Python kent het nieuwe object toe aan de oorspronkelijke aanroeper
  - Het resultaat wordt toegekend aan de variabele, die nu het object representeert
- Een klasse kan op twee manieren worden beschikbaar gemaakt: in hetzelfde bestand als de rest van de code, ofwel in een extern bestand (ingebracht met een import statement, we zien dit volgende week)

Methods van een object aanroepen

- Nadat een object is aangemaakt, kan je methods aanroepen met de volgende syntax:

```
    <object>.<methodnaam>(optionele argumenten)
```

```
oLichtSchakelaar = LichtSchakelaar()
oLichtSchakelaar.printStatus()
oLichtSchakelaar.zetAan()
oLichtSchakelaar.printStatus()
oLichtSchakelaar.zetUit()
oLichtSchakelaar.printStatus()
```

Verschillende instances creëren uit dezelfde klasse

- Eén van de belangrijkste kenmerken van OOP is dat je zoveel objecten uit een klasse kunt instantiëren als je nodig hebt
- Elk object dat je aanmaakt krijgt een eigen versie van de data
- Data veranderen bij één object zorgt er niet voor dat data bij een ander verandert

```
oLichtSchakelaar1 = LichtSchakelaar()
oLichtSchakelaar2 = LichtSchakelaar()
oLichtSchakelaar3 = LichtSchakelaar()
oLichtSchakelaar4 = LichtSchakelaar()
oLichtSchakelaar5 = LichtSchakelaar()
```

Python Data Types zijn klasses

- Alle ingebouwde Data Types zijn geïmplenteerd als klasses. We kunnen dit nagaan met de type() functie

```
mijnString = 'abcde'
print(type(mijnString))
# <class 'str'>
```

- De str klasse geeft ons een reeks methods die we kunnen aanroepen zoals .upper(), .lower(), .strip(), ...

```
mijnLijst = [10, 20, 30, 40]
print(type(mijnLijst))
# <class 'list'>
```

Python Data Types zijn klasses

- Alle lijsten zijn instances van de list klasse. Deze heeft verschillende methods zoals mijnLijst.append(), mijnLijst.count(), mijnLijst.index(), ...
- Wanneer je een klasse schrijft, dan definieer je een nieuwe data type
- Jouw code voorziet de details door te beschrijven welke data het onderhoudt en welke operaties het kan vervullen
- Je kan de ingebouwde type()-functie gebruiken

```
print(type(oLichtSchakelaar1))
# <class 'LichtSchakelaar'>
```

Een iets meer uitgebreid voorbeeld

Een iets meer uitgebreid voorbeeld

- We creëren een van iets meer toeters en bellen voorziene dimmer schakelaar
- Behalve een aan/uit switch heeft deze dimmer ook een slider die de helderheid van het licht bepaalt
- Met 11 posities, van 0 (volledig uit) tot 10 (volledige helderheid)
- Omdat we werken met een draaiknop zal de helderheid steeds met 1 toenemen of afnemen

Een iets meer uitgebreid voorbeeld

```
class DimmerSchakelaar():
    def __init__(self):
        self.schakelaarIsAan = False
        self.helderheid = 0

    def zetAan(self):
        self.schakelaarIsAan = True

    def zetUit(self):
        self.schakelaarIsAan = False

    def meerHelder(self):
        if self.helderheid \< 10:
            self.helderheid = self.helderheid + 1
```

Een iets meer uitgebreid voorbeeld

```
    def minderHelder(self):
        if self.helderheid > 0:
        self.helderheid = self.helderheid - 1

    def printStatus(self):
        print('Schakelaar is aan?', self.schakelaarIsAan)
        print('Helderheid is:', self.helderheid)

oDimmerSchakelaar = DimmerSchakelaar()
oDimmerSchakelaar.zetAan()
oDimmerSchakelaar.meerHelder()
oDimmerSchakelaar.meerHelder()
oDimmerSchakelaar.meerHelder()
oDimmerSchakelaar.meerHelder()
oDimmerSchakelaar.printStatus()
```

Een iets meer uitgebreid voorbeeld

- In de __init__() method gebruiken we 2 instance variabelen: self.schakelaarIsAan en self.helderheid
- We kennen startwaarden toe voor elk van beiden
- We hebben nu () methods: zetAan(), zetUit(), meerHelder(), minderHelder() en printStatus()

Een complexer fysiek object beschrijven

Complexer fysiek object beschrijven: een TV

- (spoiler: dit voorbeeld zal tonen hoe we kunnen werken met argumenten ;-))
- Laat ons eens nadenken over een televisie
- Meer data, dus meer state (aan/uit, stil, beschikbare kanalen, huidig kanaal, volume, volumerange)
- Meer functionaliteit, dus meer behavior (aan-/uitzetten, luider/stiller, kanaalhoppen, muten, informatie krijgen, naar specifiek kanaal gaan)

Complexer fysiek object beschrijven: een TV

```
class TV():
    def __init__(self):
        self.isAan = False
        self.isMuted = False
        self.kanaalLijst = [2, 4, 5, 7, 9, 11, 20, 36, 44, 54, 65]
        self.nKanalen = len(self.kanaalLijst)
        self.kanaalIndex = 0
        self.MINIMUM_VOLUME = 0 # constante
        self.MAXIMUM_VOLUME = 10 # constante
        self.volume = self.MAXIMUM_VOLUME

    def aanKnop(self):
        self.isAan = not self.isAan  # toggle tussen true en false
```

Complexer fysiek object beschrijven: een TV

```
    def volumeHoger(self):
        if not self.isAan:
            return
        if self.isMuted:
            self.isMuted = False  # volume wijzigen verandert muted in niet-muted
        if self.volume < self.MAXIMUM_VOLUME:
            self.volume = self.volume + 1

    def volumeLager(self):
        if not self.isAan:
            return
        if self.isMuted:
            self.isMuted = False  # volume wijzigen verandert muted in niet-muted
        if self.volume > self.MINIMUM_VOLUME:
            self.volume = self.volume - 1
```

Complexer fysiek object beschrijven: een TV

```
    def kanaalHoger(self):
        if not self.isAan:
            return
        self.kanaalIndex = self.kanaalIndex + 1
        if self.kanaalIndex > self.nKanalen:
            self.kanaalIndex = 0  # kom je aan het einde:terug naar het beginkanaal

    def kanaalLager(self):
        if not self.isAan:
            return
        self.kanaalIndex = self.kanaalIndex - 1
        if self.kanaalIndex < 0:
            self.kanaalIndex = self.nKanalen - 1  # ah begin: naar hoogste
```

Complexer fysiek object beschrijven: een TV

```
    def muten(self):
        if not self.isAan:
            return
        self.isMuted = not self.isMuted

    def selecteerKanaal(self, nieuwKanaal):
        if nieuwKanaal in self.kanaalLijst:
            self.kanaalIndex = self.kanaalLijst.index(nieuwKanaal)
        #doe enkel iets indien nieuwKanaal in de lijst staat
```

Complexer fysiek object beschrijven: een TV

```
def showInfo(self):
    print()
    print('TV Status:')
    if self.isAan:
        print(' TV is: Aan')
        print(' Kanaal is:', self.kanaalLijst[self.kanaalIndex])
        if self.isMuted:
            print(' Volume is:', self.volume, '(geluid is muted)')
        else:
            print(' Volume is:', self.volume)
    else:
        print(' TV is: Uit')
```

```
oTV = TV()
oTV.aanKnop()
oTV.showInfo()
oTV.kanaalHoger()
oTV.kanaalHoger()
oTV.volumeHoger()
oTV.volumeHoger()
oTV.showInfo()
oTV.aanKnop()
oTV.showInfo()
oTV.aanKnop()
oTV.showInfo()
oTV.volumeLager()
oTV.muten()
oTV.showInfo()
oTV.selecteerKanaal(11)
oTV.muten()
oTV.showInfo()
```

Complexer fysiek object beschrijven: een TV

- De __init__ method creëert alle instance variabelen en zet hun startwaarden
- Bekijk rustig alle methods en hun functionaliteit

Argumenten meegeven aan een method

- Wanneer je een functie aanroept dient het aantal argumenten te matchen met het aantal parameters in het def statement van de functie

```
def mijnFunctie(param1, param2, param3):
    # functiestatements

mijnFunctie(argument1, argument2, argument3)
```

- Hetzelfde geldt voor methods en het aanroepen van methods
- Self hoef je evenwel niet mee te geven.
- Bekijk het voorbeeld van de selecteerKanaal()-method

Initializatie parameters

- De mogelijkheid om argumenten mee te geven bij het aanroepen van methods werkt ook bij het instantiëren van een object.
- Te gebruiken wanneer je verschillende instances verschillende startwaardes wil meegeven
- Bvb televisietoestellen van een verschillend merk
- We doen dit door parameters toe te voegen aan de __init__ method

```
class TV():
    def __init__(self, merk, locatie):
        self.merk = merk
        self.locatie = locatie
        ...
```

Initializatie parameters

- Parameters zijn lokale variabelen, maar we kennen ze toe aan instance variabelen: hierdoor krijgen we object scope
- Conventie in Python is om de namen gelijk te houden

```
class TV():
    def __init__(self, merk, locatie):
        self.merk = merk
        self.locatie = locatie
        ...
```

Terug naar het voorbeeld

- In de TV klasse kunnen we nu verschillende objecten creëren uit dezelfde klasse, maar met verschillende data

```
oTV1 = TV('Sony', 'Woonkamer')
oTV2 = TV('Samsung', 'Slaapkamer kinderen')
```

Samenvattend

- Een goed geschreven klasse kan op eenvoudige wijze worden herbruikt in verschillende programma's
- OOP kan het aantal globale variabelen drastisch reduceren
- Klasses moeten geen toegang hebben tot globale data. Ze voorzien zelf in hun code en data
- Verschillende instances van een klasse hebben geen toegang tot elkaars data
- Deze principes maken code eenvoudiger om te debuggen

Labo-oefening

Oefening

- Je kiest zelf een fysiek voorwerp uit en modelleert een klasse om dit voorwerp te beschrijven
- Zorg voor voldoende complexiteit (> de TV) op het vlak van *state* en *behavior*
- Je zorgt dat enkele van je methods argumenten aannemen
- Je voorziet enkele *instance parameters*
- In je code instantieer je enkele objecten en je speelt ermee door methods aan te roepen
- Je dient deze oefening in tegen uiterlijk volgende week maandag