

Klassen

In een van de vorige notebooks zagen we hoe we door types te combineren complexe datastructuren konden opbouwen. Dit heeft echter ook nadelen, zoals dat we overal in onze code moeten weten hoe deze datastructuur exact in elkaar zit.

Stel bijvoorbeeld dat we een programma maken dat informatie over studenten bijhoudt, en over hun vakken en hun punten voor die vakken. Dan zouden we volgende data-structuren kunnen gebruiken:

```
In [1]: student1=[20201234,"Joske","Vermeulen","Tramezantlei 122", "Schoten",
                  [ ("Inleiding programmeren",11),("Gegevensabstractie en -structuren",
                  13)]]
student2=[20201235,"Sander","Slisse","Autolei 14", "Mortsel",
          [ ("Inleiding programmeren",15),("Computer Systemen en Architectuur",
          13)]]
```

De puntenlijst van een student kunnen we dan afdrukken met volgende functie:

```
In [2]: def printPuntenLijst(s):
        print("Punten van "+s[1][0]+" ". +s[2]+" (" +str(s[0])+")")
        for (vak,punt) in s[-1]:
            print(vak,"\t",punt)

printPuntenLijst(student1)
```

```
Punten van J. Vermeulen (20201234)
Inleiding programmeren    11
Gegevensabstractie en -structuren    13
```

Zoals we zien is om deze code te schrijven een goede kennis van de datastructuur vereist. Bovendien, als we wijzigingen aanbrengen aan de datastructuur, dan heeft dit gevolgen voor alle plekken in het programma waar de datastructuur gebruikt wordt.

Bijvoorbeeld, stel dat we niet enkel het punt willen opslaan, maar ook de datum waarop dit punt behaald werd:

```
In [3]: student1=[20201234,"Joske","Vermeulen","Tramezantlei 122", "Schoten",
                  [ ("Inleiding programmeren",11,(18,1,2020)),("Gegevensabstractie en -
                  structuren",13,(10,1,2020))]]
student2=[20201235,"Sander","Slisse","Autolei 14", "Mortsel",
          [ ("Inleiding programmeren",15,(10,1,2020)),("Computer Systemen en Ar
          chitectuur",13,(25,1,2020))]]
```

Door deze aanpassing werkt onze functie `printPuntenLijst` echter niet meer:

```
In [4]: printPuntenLijst(student1)
```

Punten van J. Vermeulen (20201234)

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-4-eb3647ff1d18> in <module>  
----> 1 printPuntenLijst(student1)
```

```
<ipython-input-2-bce04c3b211a> in printPuntenLijst(s)  
    1 def printPuntenLijst(s):  
    2     print("Punten van "+s[1][0]+" . "+s[2]+" (" +str(s[0])+")")  
----> 3     for (vak,punt) in s[-1]:  
    4         print(vak,"\t",punt)  
    5
```

```
ValueError: too many values to unpack (expected 2)
```

Dit soort problemen kunnen we vermijden door op 1 plaats functies te voorzien die de verschillende componenten van student teruggeven en aanpassen:

```

In [5]: def nieuweStudent(ID,vnaam,anaam,straat,gemeente,vakken):
        return [ID,vnaam,anaam,straat,gemeente,vakken]

def getID(student):
    return student[0]

def getVNaam(student):
    return student[1]

def getANaam(student):
    return student[2]

def getVakken(student):
    return student[-1]

def getVakNaam(vak):
    return vak[0]

def getVakPunt(vak):
    return vak[1]

def getVakExamenDatum(vak):
    return vak[-1]

def printPuntenLijst(student):
    print("Punten van "+getVNaam(student)[0]+". "+getANaam(student)+" (" +str(g
etID(student))+")")
    for vak in getVakken(student):
        print(getVakNaam(vak),"\t",getVakPunt(vak))

student1=nieuweStudent(20201234,"Joske","Vermeulen","Tramezantlei 122", "Schot
en",
        [("Inleiding programmeren",11,(18,1,2020)),("Gegevensabstractie en -
structuren",13,(10,1,2020))])
student2=nieuweStudent(20201235,"Sander","Slisse","Autolei 14", "Mortsel",
        [("Inleiding programmeren",15,(10,1,2020)),("Computer Systemen en Ar
chitectuur",13,(25,1,2020))])

printPuntenLijst(student1)

```

```

Punten van J. Vermeulen (20201234)
Inleiding programmeren    11
Gegevensabstractie en -structuren    13

```

We hebben dus langs de ene kant data en aan de andere kant functies die op die data werken en die bij elkaar horen. In zekere zin vormen de data en functies samen een nieuw *type* student. Dit kunnen we expliciet maken door die data en functies samen te voegen in een *klasse* (*class*) student. Alle studenten die we in ons programma gaan gebruiken worden dan *instanties* of *objecten* van die klasse. Net zoals "abcd" een instantie is van klasse string, en s.lower() een functie is die op een string s werkt, worden in het volgende programmaatje student1 en student2 instanties van klasse student. Bekijk deze code even om het idee te vatten; we bekijken de individuele componenten erna.

```

In [6]: def getVakNaam(vak):
        return vak[0]

def getVakPunt(vak):
    return vak[1]

def getVakExamenDatum(vak):
    return vak[-1]

class student:
    def nieuweStudent(self, ID, vnaam, anaam, straat, gemeente, vakken):
        self.ID=ID
        self.vnaam=vnaam
        self.anaam=anaam
        self.straat=straat
        self.gemeente=gemeente
        self.vakken=vakken

    def getID(self):
        return self.ID

    def getVNaam(self):
        return self.vnaam

    def getANaam(self):
        return self.anaam

    def getVakken(self):
        return self.vakken

    def printPuntenLijst(self):
        print("Punten van "+self.getVNaam()[0]+" ". +self.getANaam()+" (" +str(s
elf.getID())+"")
        for vak in self.getVakken():
            print(getVakNaam(vak), "\t", getVakPunt(vak))

student1=student()
student1.nieuweStudent(20201234, "Joske", "Vermeulen", "Tramezantlei 122", "Schot
en",
                        [("Inleiding programmeren", 11, (18, 1, 2020)), ("Gegevensabstractie en -
structuren", 13, (10, 1, 2020))])
student2=student()
student2.nieuweStudent(20201235, "Sander", "Slisse", "Autolei 14", "Mortsel",
                        [("Inleiding programmeren", 15, (10, 1, 2020)), ("Computer Systemen en Ar
chitectuur", 13, (25, 1, 2020))])

student1.printPuntenLijst()

```

Punten van J. Vermeulen (20201234)

Inleiding programmeren 11

Gegevensabstractie en -structuren

13

Definitie van een klasse

Met het keyword `class` duiden we aan dat wat volgt de *definitie* is van een klasse. Net zoals bij een functie doet een definitie zelf niets. `class` wordt gevolgd door de naam van de klasse, student in dit geval, en een dubbele punt. De definitie van de klasse is nu het ingesprongen gedeelte na de dubbele punt.

We bouwen ons voorbeeld op startende van een lege klasse. Omdat Python geen lege blokken toestaat, schrijven we na de definitie van onze klasse `pass` ; wat zoveel betekent als: niets te zien hier, loop maar door.

```
In [7]: class vb:
        pass
```

Instanties van een klasse maken

We kunnen nu *instanties* of *objecten* van deze klasse aanmaken door de naam van de klasse te nemen gevolgd door lege haken. Dus, in ons minimaal voorbeeld: `vb()` . Dit creëert een nieuw object van onze klasse. Dit object kunnen we toekennen aan een variabele, meegeven als parameter aan een functie, enzovoort. Het is een waarde net als `5` , `"abc"` , `[]` , of `{'a':1}` .

```
In [8]: v1=vb()
        v2=vb()
        print(vb(),v1,v2)

<__main__.vb object at 0x05760D18> <__main__.vb object at 0x05760CE8> <__main__.vb object at 0x05760CD0>
```

In de uitvoer van de `print` in vorig voorbeeldje zie je dat er drie verschillende objecten van klasse `vb` werden aangemaakt. Je kan zien dat het verschillende objecten zijn aan de hand van hun geheugenadres dat volgt na `at` . Een andere manier om de identiteit van objecten te testen is via de functie `id` die van elk object de unieke identifier geeft; zeg maar het paspoortnummer.

```
In [9]: print(id(v1),id(v2),id(vb()))

91622632 91622608 91621000
```

Data aan een object koppelen

We kunnen de inhoud van een object inspecteren via de dot (`.`) operatie. Onze objecten zijn momenteel nog leeg, dus kunnen we ook niets inspecteren.

In Python is het erg eenvoudig om aan een object data te koppelen, opnieuw door gebruik te maken van `.` . We kunnen bijvoorbeeld variabelen `x` en `y` toevoegen aan object `v1` :

```
In [10]: v1.x=3
         v1.y=7
         print(v1.x,v1.y)
```

3 7

Methods

Meestal gaan we echter niet rechtstreeks variabelen toevoegen aan objecten van een klasse, maar gaan we dat doen via functies die voor de hele klasse gedefinieerd zijn. Een definitie van een klasse bestaat dan typisch ook uit een verzameling functies, ook wel de *methods* van de klasse genoemd. Elke *method* heeft een verplichte eerste parameter die het object zelf weergeeft. De conventie is om die eerste parameter altijd `self` te noemen. Op die manier kunnen de methods makkelijk de data van het object benaderen.

Een method aanroepen op een object doen we opnieuw via de dot operator. Bij het aanroepen van een method van een klasse op een object van die klasse hoeven we niet zelf een argument te geven voor de parameter `self`. Python zal die zelf invullen. Anders gezegd: als `o` een object is van klasse `C` en klasse `C` heeft een method `m`, dan is de eerste parameter van de method `m` steeds `self`. Als dit de enige parameter is van deze method, dan roepen we `m` aan voor object `o` als volgt: `o.m()`, dus zonder parameter. Python vult dit zelf aan (en maakt er intern `C.m(o)` van).

Volgend voorbeeld illustreert het toevoegen en gebruiken van methods.

```
In [11]: class vb:
         def addData(self,x,y):
             self.x=x
             self.y=y

         def printData(self):
             print(self.x,self.y)

v1=vb()
v2=vb()
v1.addData(1,2)    # geen argument voor self nodig!
v2.addData(3,4)    # geen argument voor self nodig!

v1.printData()     # geen argument voor self nodig!
v2.printData()     # geen argument voor self nodig!
```

1 2
3 4

Zorg ervoor dat je nooit vergeet om in de defintie van een method `self` als eerste parameter op te nemen. Als je dat vergeet, zal namelijk nog steeds de eerste parameter hiervoor gebruikt worden met eigenaardige foutmeldingen tot gevolg:

```
In [12]: class vb:
    def addData(x,y): # self vergeten !!!
        self.x=x
        self.y=y

    def printData(): # self vergeten !!!
        print(self.x,self.y)

v1=vb()
v2=vb()
v1.addData(1,2)
v2.addData(3,4)

v1.printData()
v2.printData()
```

```
-----
TypeError                                Traceback (most recent call last)
<ipython-input-12-b36ae911cc67> in <module>
     10 v1=vb()
     11 v2=vb()
--> 12 v1.addData(1,2)
     13 v2.addData(3,4)
     14
```

TypeError: addData() takes 2 positional arguments but 3 were given

Je krijgt hier de fout dat `addData` maar 2 parameters heeft en jij 3 argumenten hebt gegeven. Erg verwarrend, maar dit komt omdat Python intern `v1.addData(1,2)` omzet naar `vb.addData(v1,1,2)` .

Vergeet ook niet om `self.` te zetten voor de variabelen die bij het object horen. Als je `self.` vergeet, dan wordt de waarde niet in het object opgeslagen, maar in een lokale variabele van jouw method. Deze lokale variabele verdwijnt dan vervolgens eens de functie is afgelopen. Dit gebeurt in volgend stukje code:

```
In [13]: class vb:
    def addData(self,x1,y1):
        self.x=x1
        y=y1    # We zijn de self. vergeten voor y; y is dus een lokale variab
    ele

    def printData(self):
        print(self.x,self.y)

v1=vb()
v1.addData(1,2)
v1.printData()
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-13-ad328834f170> in <module>
      10 v1=vb()
      11 v1.addData(1,2)
----> 12 v1.printData()

<ipython-input-13-ad328834f170> in printData(self)
      5
      6     def printData(self):
----> 7         print(self.x,self.y)
      8
      9

AttributeError: 'vb' object has no attribute 'y'
```

Initialiser

Merk op dat we de data van de objecten van klasse `vb` expliciet moeten toevoegen via een aanroep van de method `addData` . Als we dat vergeten dan kunnen we een andere method, `printData` , niet correct gebruiken omdat die veronderstelt dat data variabelen `x` en `y` aan ons object gekoppeld zijn. Eigenlijk willen we in dit geval dat de functie `addData` altijd een keer wordt uitgevoerd voordat we het object voor de eerste keer gebruiken. Dit is exact de functie van een *initialiser* in een klasse. Een initializer heeft steeds de naam `__init__` (twee *underscores* (`_`) gevolgd door *init* gevolgd door opnieuw twee *underscores*) en heeft als parameters steeds de obligate `self` en vervolgens alle bijkomende parameters die jij nodig acht.

Voor onze klasse `vb` zouden we bijvoorbeeld kunnen zeggen dat steeds wanneer een object van deze klasse wordt aangemaakt, er beginwaarden voor `x` en `y` gegeven moeten worden. Dat ziet er dan als volgt uit:


```
In [14]: class vb:
    def __init__(self,x,y):
        self.x=x
        self.y=y

    def addData(self,x1,y1):
        self.x=x1
        self.y=y1

    def printData(self):
        print(self.x,self.y)

v1=vb(1,2)    # omdat de initializer twee parameters heeft, zijn we nu
              # verplicht om 2 argumenten mee te geven bij creatie van een object
v1.printData()
```

1 2

Python zal standaard een lege initializer maken als wij er zelf geen voorzien. Dit verklaart waarom we lege haken hadden in onze eerste voorbeelden. Dit komt eigenlijk neer op volgende code:

```
In [15]: class vb:
    def __init__(self):
        pass

    def addData(self,x1,y1):
        self.x=x1
        self.y=y1

    def printData(self):
        print(self.x,self.y)

v1=vb()
#v1.printData() # deze laten we nu weg want die geeft een fout omdat v1.x en
                # v1.y niet bestaan
```

Een initialiser is enorm handig om te vermijden dat de inhoud van objecten geen correcte toestand weergeven. Een incorrecte toestand is bijvoorbeeld een ontbrekende variabele.

Methods die andere methods aanroepen

Een method kan ook een andere method aanroepen door gebruik te maken van `self.methodenaam(argumenten)`. Merk op dat we hier niet zelf een argument moeten geven voor de `self` parameter omdat we gebruik maken van de dot notatie bij de aanroep (`self dot methodenaam`). We kunnen bijvoorbeeld de initializer in het vorige voorbeeld `addData` laten aanroepen in plaats van zelf de variabelen `self.x` en `self.y` toe te wijzen.

```
In [16]: class vb:
          def __init__(self):
              self.addData(0,0)  # vanuit een method een andere method aanroepen

          def addData(self,x1,y1):
              self.x=x1
              self.y=y1

          def printData(self):
              print(self.x,self.y)

v1=vb()
v1.printData()
```

0 0

Voorbeeld: student

Met wat we net zagen kunnen we het voorbeeld van de klasse student volledig begrijpen. We herhalen het voorbeeld hier even ter referentie:

```

In [17]: def getVakNaam(vak):
          return vak[0]

def getVakPunt(vak):
    return vak[1]

def getVakExamenDatum(vak):
    return vak[-1]

class student:
    def nieuweStudent(self, ID, vnaam, anaam, straat, gemeente, vakken):
        self.ID=ID
        self.vnaam=vnaam
        self.anaam=anaam
        self.straat=straat
        self.gemeente=gemeente
        self.vakken=vakken

    def getID(self):
        return self.ID

    def getVNaam(self):
        return self.vnaam

    def getANaam(self):
        return self.anaam

    def getVakken(self):
        return self.vakken

    def printPuntenLijst(self):
        print("Punten van "+self.getVNaam()[0]+" ". +self.getANaam()+" (" +str(s
elf.getID())+"")
        for vak in self.getVakken():
            print(getVakNaam(vak), "\t", getVakPunt(vak))

student1=student()
student1.nieuweStudent(20201234, "Joske", "Vermeulen", "Tramezantlei 122", "Schot
en",
                        [("Inleiding programmeren", 11, (18, 1, 2020)), ("Gegevensabstractie en -
structuren", 13, (10, 1, 2020))])
student2=student()
student2.nieuweStudent(20201235, "Sander", "Slisse", "Autolei 14", "Mortsel",
                        [("Inleiding programmeren", 15, (10, 1, 2020)), ("Computer Systemen en Ar
chitectuur", 13, (25, 1, 2020))])

student1.printPuntenLijst()

```

Punten van J. Vermeulen (20201234)

Inleiding programmeren 11

Gegevensabstractie en -structuren

13

Merk op dat we de klasse student nu kunnen gebruiken als een type. Dit is niet enkel alsof; een klasse definieert een nieuw type dat we kunnen gebruiken net als elk ander type. Zo kunnen we bijvoorbeeld studentenlijsten maken:

```
In [18]: BA1=[student1,student2]  # een klein jaar blijktbaar ...
```

We zouden ook verder kunnen gaan en van vak ook een klasse maken, en in plaats van de naam van een vak op te slaan, het vak bij de student op te slaan. Verder hebben we de initialiser aangepast; om student correct te kunnen gebruiken hoeft er niet per se een vak te zijn dus laten we dit uit de initialiser weg en voegen we in de plaats daarvan methods toe om vakken en punten toe te voegen.

```

In [19]: class vak:
    def __init__(self, code, naam):
        self.code=code
        self.naam=naam

    def getCode(self):
        return self.code

    def getVakNaam(self):
        return self.naam

class student:
    def __init__(self, ID, vnaam, anaam, straat, gemeente):
        self.ID=ID
        self.vnaam=vnaam
        self.anaam=anaam
        self.straat=straat
        self.gemeente=gemeente
        self.vakken=[]
        self.punten={} # Lege dictionary die vakcodes koppelt aan punten

    def getID(self):
        return self.ID

    def getVNaam(self):
        return self.vnaam

    def getANaam(self):
        return self.anaam

    def getVakken(self):
        return self.vakken

    def addVak(self, vak):
        self.vakken.append(vak)

    def addGrade(self, v, g):
        self.punten[v.getCode()]=g

    def getGrade(self, v):
        return self.punten.get(v.getCode(), "-")

    def printPuntenLijst(self):
        print("Punten van "+self.getVNaam()[0]+" ". +self.getANaam()+" (" +str(s
elf.getID())+")")
        for v in self.getVakken():
            print(v.getVakNaam(), "\t", self.getGrade(v))

IP=vak("IP", "Inleiding Programmeren")
GAS=vak("GAS", "Gegevensabstractie en -structuren")
CSA=vak("CSA", "Computer Systemen en Architectuur")

student1=student(20201234, "Joske", "Vermeulen", "Tramezantlei 122", "Schoten")
student1.addVak(IP)
student1.addVak(GAS)

```

```

student1.addGrade(IP,11)
student1.addGrade(GAS,13)

student2=student(20201235,"Sander","Slisse","Autolei 14", "Mortsel")
student2.addVak(IP)
student2.addVak(CSA)
student2.addGrade(IP,15)
student2.addGrade(CSA,13)

student1.printPuntenLijst()

```

```

Punten van J. Vermeulen (20201234)
Inleiding Programmeren    11
Gegevensabstractie en -structuren    13

```

Je vraagt je nu misschien af wat het grote nut was om in dit voorbeeld klassen te gebruiken. Het programmeren met klassen, ook wel *object-georiënteerd programmeren* genoemd, is essentieel om projecten met grotere complexiteit beheersbaar te houden. Zonder nu reeds al te veel in te gaan op de inhoud van de verschillende Software-engineering vakken die jullie verder in de opleiding nog gaan krijgen, sommen we hier enkele voordelen op:

- de opbouw van jouw programma wordt logischer, met functies die de data van een klasse beheren dicht bij de definitie van de klasse. Je maakt een eigen type aan, met methods om objecten van dit type te beheren.
- een andere programmeur, of jijzelf, kan de objecten van de klasse gebruiken zonder te hoeven weten hoe deze objecten er intern uit zien en aan welke voorwaarden ze moeten voldoen. Bijvoorbeeld: zonder te weten wat de method `append` van de klasse `list` intern exact doet, kan jij ze toch gebruiken. Achteraf kan je aan de elementen die je in de lijst zette ook aan, via de operator `[index]`, die eigenlijk niet meer is dan een method met een ietwat bijzondere naam en manier van argumenten doorgeven.
- zolang de *interface* (verzameling van methods om de inhoud van een object te beheren) niet van vorm wijzigt, kan je de hele interne structuur van een object wijzigen, zoals bijvoorbeeld functionaliteit toevoegen, zonder dat je ergens anders in jouw programma ook maar een letter code moet wijzigen.

Dit laatste punt illustreren we even. Stel dat we aan een vak ook een docent willen toevoegen, dan kunnen we de definitie van vak wijzigen. De rest van onze code hoeft echter niet te wijzigen omdat we de bestaande interface volledig behouden. We breiden hem enkel uit:

```

In [20]: class vak:
    def __init__(self, code, naam):
        self.code=code
        self.naam=naam
        self.docent=""

    def getCode(self):
        return self.code

    def getVakNaam(self):
        return self.naam

    def addDocent(self, d):
        self.docent=d

    def getDocent(self):
        return self.docent

##### Onder deze regel wijzigde niets #####

class student:
    def __init__(self, ID, vnaam, anaam, straat, gemeente):
        self.ID=ID
        self.vnaam=vnaam
        self.anaam=anaam
        self.straat=straat
        self.gemeente=gemeente
        self.vakken=[]
        self.punten={} # Lege dictionary die vakcodes koppelt aan punten

    def getID(self):
        return self.ID

    def getVNaam(self):
        return self.vnaam

    def getANaam(self):
        return self.anaam

    def getVakken(self):
        return self.vakken

    def addVak(self, vak):
        self.vakken.append(vak)

    def addGrade(self, v, g):
        self.punten[v.getCode()]=g

    def getGrade(self, v):
        return self.punten.get(v.getCode(), "-")

    def printPuntenLijst(self):
        print("Punten van "+self.getVNaam()[0]+" ". +self.getANaam()+" (" +str(s
elf.getID())+")")
        for v in self.getVakken():

```

```

        print(v.getVakNaam(), "\t", self.getGrade(v))

IP=vak("IP", "Inleiding Programmeren")
GAS=vak("GAS", "Gegevensabstractie en -structuren")
CSA=vak("CSA", "Computer Systemen en Architectuur")

student1=student(20201234, "Joske", "Vermeulen", "Tramezantlei 122", "Schoten")
student1.addVak(IP)
student1.addVak(GAS)
student1.addGrade(IP, 11)
student1.addGrade(GAS, 13)

student2=student(20201235, "Sander", "Slisse", "Autolei 14", "Mortsel")
student2.addVak(IP)
student2.addVak(CSA)
student2.addGrade(IP, 15)
student2.addGrade(CSA, 13)

student1.printPuntenLijst()

```

```

Punten van J. Vermeulen (20201234)
Inleiding Programmeren    11
Gegevensabstractie en -structuren    13

```