

Objecten zijn *mutable*

We werken in deze notebook nog even voort met het laatste voorbeeld uit de vorige notebook. Voer dus eerst volgend stukje code uit:

```

In [1]: 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

```

Objecten van klassen zijn mutable; toen we aan `student1` een vak en een punt toevoegden, bleef dit hetzelfde object, maar de inhoud was gewijzigd:

```

In [2]: student3=student(20201236, "Gaston", "Berghmans", "Autolei 34", "Mortsel")
        print(id(student3))
        student3.addVak(IP)
        print(id(student3))

93913512
93913512

```

Dit betekent ook dat indien meerdere variabelen naar hetzelfde object verwijzen en dit object verandert, dat deze gewijzigde waarde ook via de andere variabelen wordt gezien. Een minimaal voorbeeldje om dit te illustreren:

```

In [3]: student1.printPuntenLijst()
        student4=student1
        student4.addVak(CSA)
        student1.printPuntenLijst()

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

```

Student1 en student4 verwijzen naar hetzelfde object en daarom is elke wijziging aan student1 ook een wijziging aan student4 en omgekeerd:

```
In [4]: print(id(student1),id(student4))  
94103128 94103128
```

Dit is niet altijd slecht! In veel gevallen is dit zelfs uitermate wenselijk! Stel bijvoorbeeld dat we een foutje maakten in de naam van het vak IP . Dit vak zit in de vakkenlijst van alle drie onze studenten (we vergeten even student4 , want dat is eigenlijk student1). Moeten we nu de naam van dit vak bij alle drie de studenten gaan wijzigen? Het antwoord is: neen! Omdat we het vak niet als een string opsloegen in de studentobjecten, maar als object, verwijst in de vakkenlijsten van onze studenten de entry voor IP naar het object voor dit vak. Ook de variabele IP verwijst hiernaar, dus moeten we dit maar op 1 plaats wijzigen:

```
In [5]: IP.naam="Inleiding tot programmeren"  
student1.printPuntenLijst()  
print()  
student2.printPuntenLijst()  
print()  
student3.printPuntenLijst()  
print()  
  
Punten van J. Vermeulen (20201234)  
Inleiding tot programmeren      11  
Gegevensabstractie en -structuren      13  
Computer Systemen en Architectuur      -  
  
Punten van S. Slisse (20201235)  
Inleiding tot programmeren      15  
Computer Systemen en Architectuur      13  
  
Punten van G. Berghmans (20201236)  
Inleiding tot programmeren      -
```

Copy en deepcopy

Willen we nu een kopij maken van een object, dan kunnen we dat doen met de functies copy en deepcopy uit de module copy . Met copy maken we een ondiepe kopij, wat wil zeggen dat we wel een nieuw object maken, maar binnen dat object geen kopijen maken van de waarden. Met deepcopy doen we dat wel. Volgend voorbeeld illustreert dit:

```

In [6]: from copy import copy, deepcopy

class test:
    def __init__(self):
        self.lijst=[]

    def empty(self):
        self.lijst=[]

    def add(self,a):
        self.lijst.append(a)

    def slijst(self):
        s=""
        for i in self.lijst:
            s=s+str(i)+", "
        return "["+s[:-2]+"]"

t=test()
t2=t
tcopy=copy(t)
tdeepcopy=deepcopy(t)
print("id(t):\t\t",id(t),"\nid(t2):\t\t",id(t2),"\nid(tcopy):\t\t",id(tcopy),"\n\nid(tdeepcopy):\t\t",id(tdeepcopy))
print("id(t.lijst):\t\t",id(t.lijst),"\nid(t2.lijst):\t\t",id(t2.lijst),"\nid(tcopy.lijst):\t\t",id(tcopy.lijst),"\nid(tdeepcopy.lijst):\t\t",id(tdeepcopy.lijst))

t.add(1)
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())
t2.add(2)
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())
tcopy.add(3)
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())
tdeepcopy.add(4)
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())

t.empty()
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())
t.add(1)
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())
t2.add(2)
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())
tcopy.add(3)
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())
tdeepcopy.add(4)
print(t.slijst(),t2.slijst(),tcopy.slijst(),tdeepcopy.slijst())

```

```

id(t):          96533208
id(t2):         96533208
id(tcopy):      96533376
id(tdeepcopy):  96533448
id(t.lijst):    94018024
id(t2.lijst):   94018024
id(tcopy.lijst): 94018024
id(tdeepcopy.lijst): 91363144
[1] [1] [1] []
[1, 2] [1, 2] [1, 2] []
[1, 2, 3] [1, 2, 3] [1, 2, 3] []
[1, 2, 3] [1, 2, 3] [1, 2, 3] [4]
[] [] [1, 2, 3] [4]
[1] [1] [1, 2, 3] [4]
[1, 2] [1, 2] [1, 2, 3] [4]
[1, 2] [1, 2] [1, 2, 3, 3] [4]
[1, 2] [1, 2] [1, 2, 3, 3] [4, 4]

```

Bekijk deze code goed. Inspecteer zeker ook [het Python tutor fragment van deze code](http://www.pythontutor.com/visualize.html#code=from%20copy%20import%20copy,deepcopy%0A%0Aclass%20t%25D%2B%22%5D%22%0A%20%20%20%20%0At%3Dtest%28%29%0At%3Dt%0Atcopy%3Dcopy%28t%29%0A%0Afrontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false) (<http://www.pythontutor.com/visualize.html#code=from%20copy%20import%20copy,deepcopy%0A%0Aclass%20t%25D%2B%22%5D%22%0A%20%20%20%20%0At%3Dtest%28%29%0At%3Dt%0Atcopy%3Dcopy%28t%29%0A%0Afrontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false>). Als je dit voorbeeld goed begrijpt, ben je helemaal mee met copy, deepcopy en mutable. We lichten enkele essentiële lijnen code toe:

- met `t2=t` laten we variabele `t2` naar hetzelfde `test` object verwijzen als `t`. Alles wat met `t` gebeurt, gebeurt dus ook met `t2` en vice versa.
- `tcopy` is een ondiepe kopij van `t` (en dus ook van `t2`); het is een nieuw object, maar `tcopy.lijst` verwijst naar hetzelfde lijstobject als `t` en `t2`. Alles wat er dus met `tcopy.lijst` gebeurt, gebeurt ook met `t.lijst` en `t2.lijst`.
- Voor `tdeepcopy` geldt dit niet; `tdeepcopy` staat volledig los van de andere drie `test` objecten en deelt zelfs de lijst niet.
- Wanneer we `t.empty()` uitvoeren, wordt `t.lijst=[]` uitgevoerd. Dat wil zeggen: vanaf dat punt verwijst `t.lijst` niet langer naar de lijst waarnaar die verwees, maar naar een nieuwe, lege lijst. Omdat `t2` gewoon een andere naam is voor `t`, is dit ook het geval voor `t2`. `t.lijst` en `t2.lijst` wijzen vanaf nu naar dezelfde, nieuwe lege lijst.
- Voor `tcopy` is dit niet het geval. `tcopy` is een ander object dan `t` en heeft haar eigen variabele `lijst`, die nog steeds naar de originele lijst met elementen `[1, 2, 3, 4]` wijst.

In de meeste gevallen, echter, kunnen we niet zomaar `copy` of `deepcopy` gebruiken maar zullen we een eigen functie moeten schrijven die een object op een correcte manier kopieert. Denken we bijvoorbeeld aan onze klasse `student`; als we een kopij maken van `student`, dan willen we dat elke `student` zijn of haar eigen lijst met vakken heeft, maar de vakken in die lijst moeten verwijzingen zijn naar het vakobject en geen kopij. In dit geval werkt dus helaas noch het gebruik van `copy`, noch het gebruik van `deepcopy`.