

Inleiding tot Programmeren

Toon Calders

toon.calders@uantwerpen.be

Les 6: Klassen

Wat zagen we vorige keer?

- Samengestelde datatypes
- Recursie in Python
 - Functies die zichzelf aanroepen
 - Iedere aanroep heeft zijn eigen stackframe
 - Geen toegang tot variabelen van andere instanties
- Recursie staat vaak toe om elegante code te schrijven

Vlugge quiz

- Wat is de uitvoer van volgend code fragment?

```
def f(n):  
    n=n-1  
    return "*" + f(n)  
  
print(f(5))
```

(A)*****

(B)*

(C)Andere, namelijk:

Vlugge quiz

- Wat is de uitvoer van volgend code fragment?

```
def f(n):  
    n=n-1  
    return "*" + f(n)  
  
print(f(5))
```

(A)*****

(B)*

(C)Andere

```
File "C:/Users/toon/Google Drive/courses/2016/Inle  
es/06/quiz.py", line 3, in f  
    return "*" + f(n)  
File "C:/Users/toon/Google Drive/courses/2016/Inle  
es/06/quiz.py", line 3, in f  
    return "*" + f(n)  
RecursionError: maximum recursion depth exceeded
```

Vlugge quiz

- Wat is de uitvoer van volgend code fragment?

```
def f(n):  
    if n==0:  
        return ""  
    n=n-1  
    return "*" + f(n)
```

```
n=5  
print(f(n), n)
```

- (A)***** 5
- (B)***** 0
- (C)Andere, namelijk: ...

Vlugge quiz

- Wat is de uitvoer van volgend code fragment?

```
def f(n):  
    if n==0:  
        return ""  
    n=n-1  
    return "*" + f(n)
```

```
n=5  
print(f(n), n)
```

(A)***** 5

(B)***** 0

(C)Andere, namelijk: ...

Vlugge quiz

- Wat is de uitvoer van
volgend code fragment?

```
def f(L):  
    if L==[]:  
        return ""  
    del(L[0])  
    return "*" + f(L)  
  
L=list("password")  
print(f(L),L)
```

- (A)***** []
- (B)***** ['p', 'a', 's', 's', 'w', 'o', 'r', 'd']
- (C)***** password

Vlugge quiz

- Wat is de uitvoer van
volgend code fragment?

```
def f(L):  
    if L==[]:  
        return ""  
    del(L[0])  
    return "*" + f(L)  
  
L=list("password")  
print(f(L),L)
```

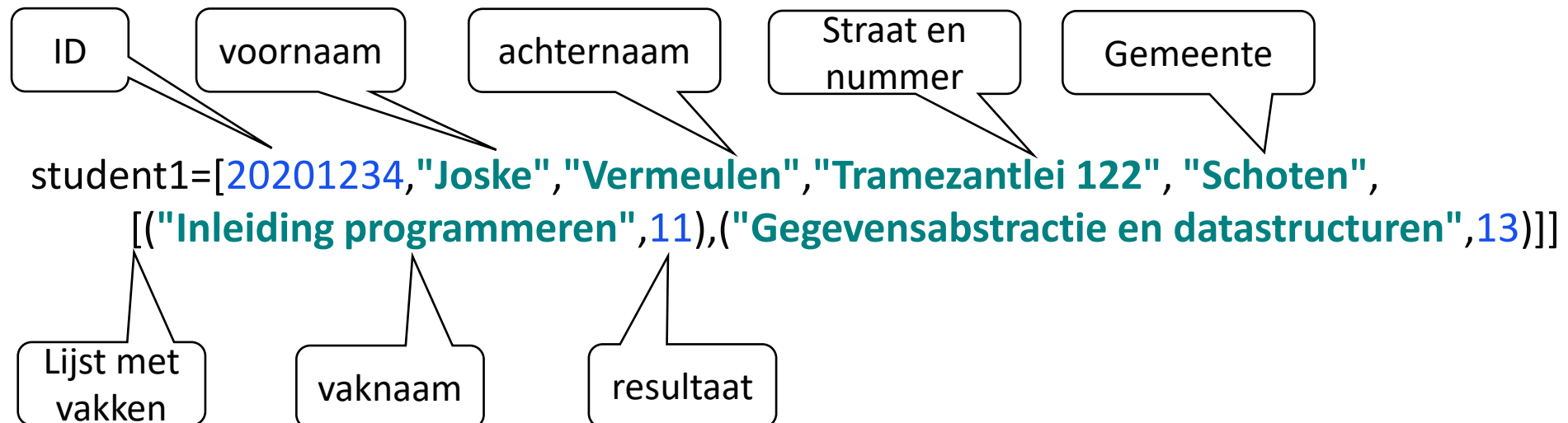
- (A)***** []
- (B)***** ['p', 'a', 's', 's', 'w', 'o', 'r', 'd']
- (C)***** password

Les van vandaag

- Eigen datatypes maken: klassen
 - Definitie van een klasse
 - Methods, constructor
 - Inheritance
- Een **klasse definitie** is een verzameling van variabelen die gezamenlijk een object omschrijven samen met functies om die variabelen te manipuleren
= een user-defined user type
- Een **object** is een instantie van een klasse

Motiverend voorbeeld

- Stel we willen studenten opslaan in ons programma en gebruiken hiervoor een samengesteld datatype:



Motiverend voorbeeld

- We maken functies om b.v.b. een puntenlijst af te drukken:

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

Punten van J. Vermeulen (20201234)

Inleiding programmeren 11

Gegevensabstractie en datastructuren 13

Motiverend voorbeeld

- Vervolgens beslissen we: voeg de datum van het examen toe in de data structuur

```
student1=[20201234,"Joske","Vermeulen","Tramezantlei 122", "Schoten",  
          [ ("Inleiding programmeren",11,(18,1,2020)),  
            ("Gegevensabstractie en datastructuren",13,(10,1,2020))]  
          ]
```

- Dan moeten we alle stukken code die student gebruiken herschrijven!

Traceback (most recent call last):

File "C:/Users/Toon/PycharmProjects/pythonProject/les-12-10-2020/main.py", line 14, in <module>
printPuntenLijst(student1)

File "C:/Users/Toon/PycharmProjects/pythonProject/les-12-10-2020/main.py", line 11, in printPuntenLijst
for (vak,punt) in s[-1]:

ValueError: too many values to unpack (expected 2)

“Good Practice”

- Ga nooit rechtstreeks de data-structuur benaderen:

```
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]
```

Interface voor mijn
samengesteld data-type

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

“Good Practice”

- Verandert mijn datastructuur → enkel mijn interface wijzigen

```
def getVNaam(student):  
    return student[1]
```

```
def getMiddleInitials(student):  
    return student[2]
```

```
def getANaam(student):  
    return student[3]
```

- De rest van mijn code blijft ongewijzigd

“Good Practice”

- We willen dus de definitie van de datastructuur en functies om die data te manipuleren bij elkaar houden
- Dat is exact wat een klasse doet
- Een voorbeeld van een klasse die je reeds kent:
 - list: append, remove, discard, []

```
l=list()  
print(type(l))  
l.append(4)  
l.sort()  
print(l.pop())
```

```
# construeer nieuwe lijst  
# <class 'list'>  
# functie die iets toevoegt  
# functie die iets wijzigt in l  
# geeft iets terug
```

Klassen

- Dit is exact wat *class* doet in Python:

`class student:`

Naam van de nieuwe klasse, zeg maar type

Method

`def setValues(self, ID, naam, adres, vakken):`

`self.ID = ID`

`self.naam = naam`

`self.adres = adres`

`self.vakken = vakken`

Method

`def getID(self):`

`return self.ID`

Method

`def getNaam(self):`

`return self.naam`

Nieuw object aanmaken

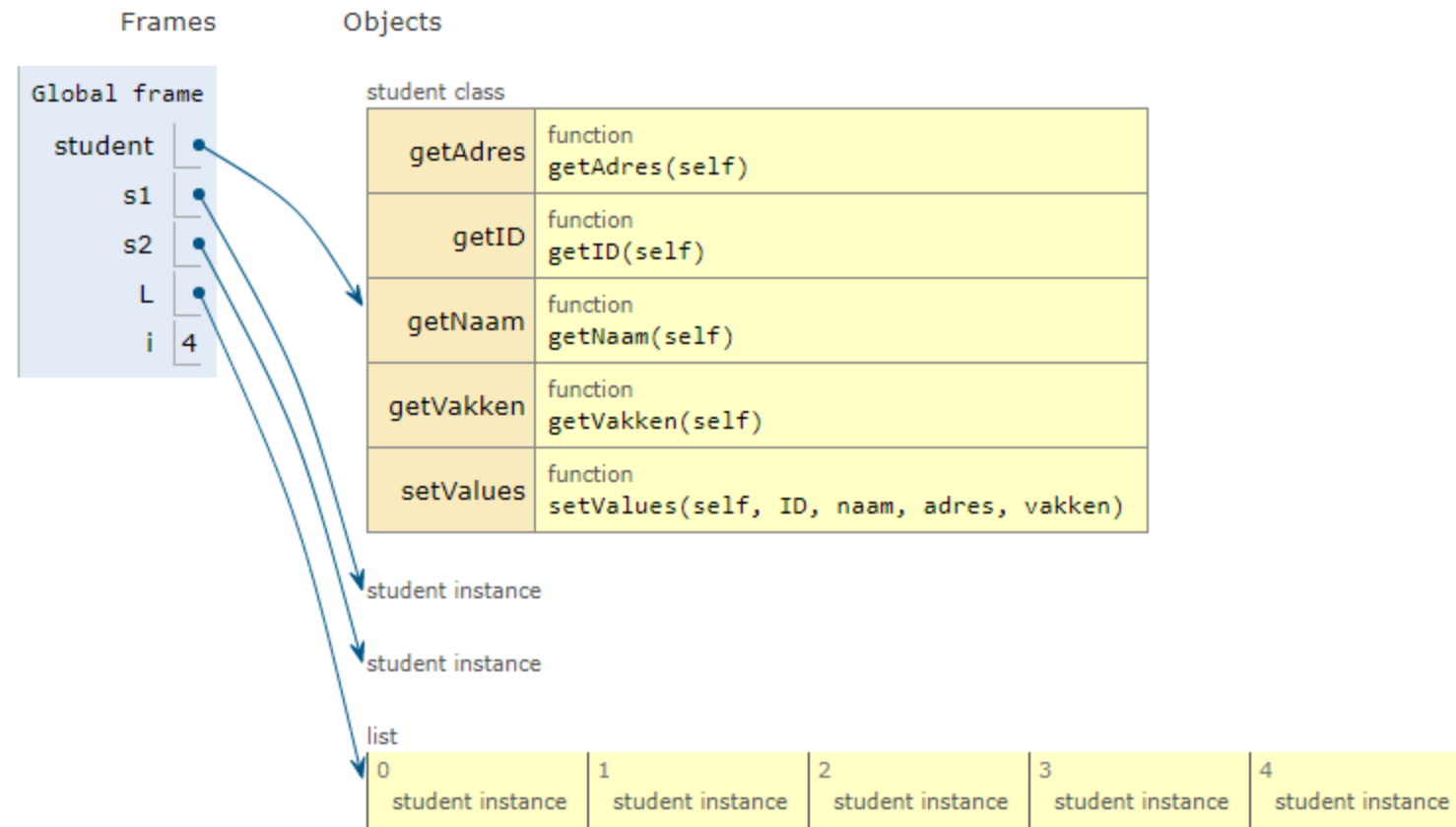
`s = student()`

...

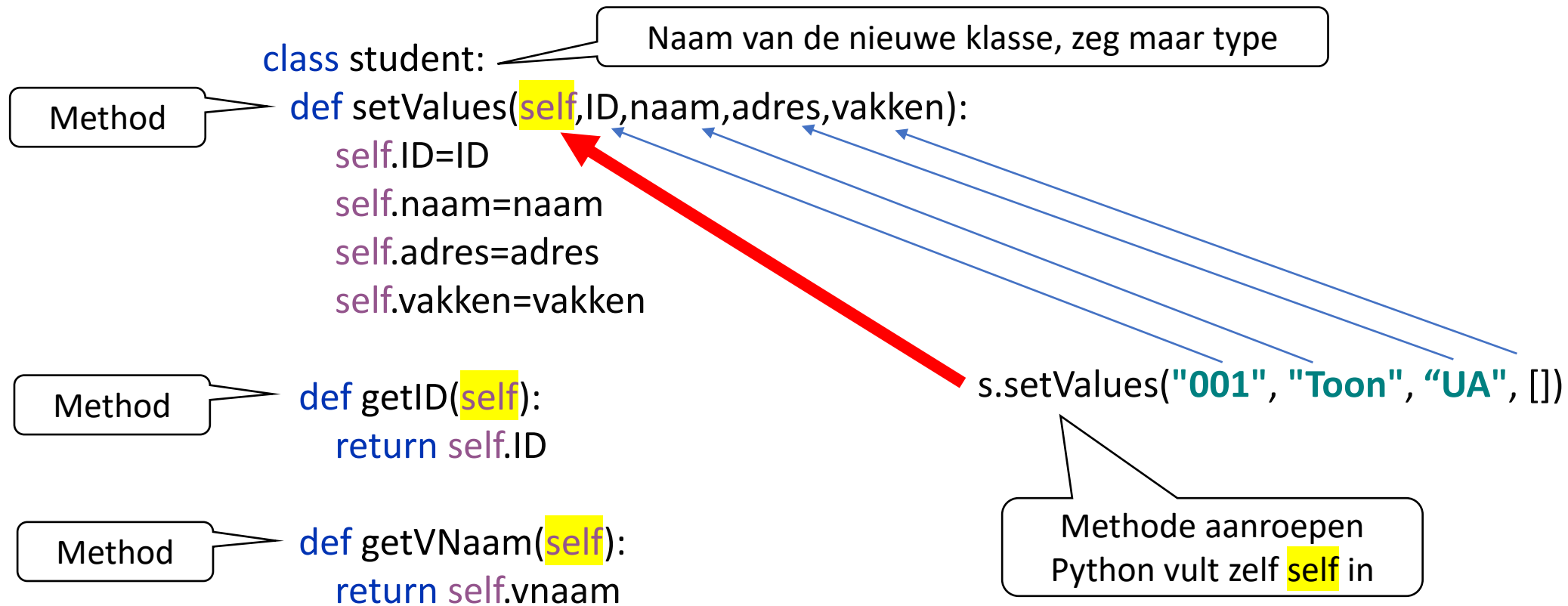
Klassen – aanmaken objecten

- Telkens we `student()` aanroepen, wordt er een nieuwe student aangemaakt

```
s1=student()  
s2=student()  
L=list()  
for i in range(5):  
    L.append(student())
```



Klassen – aanroepen method

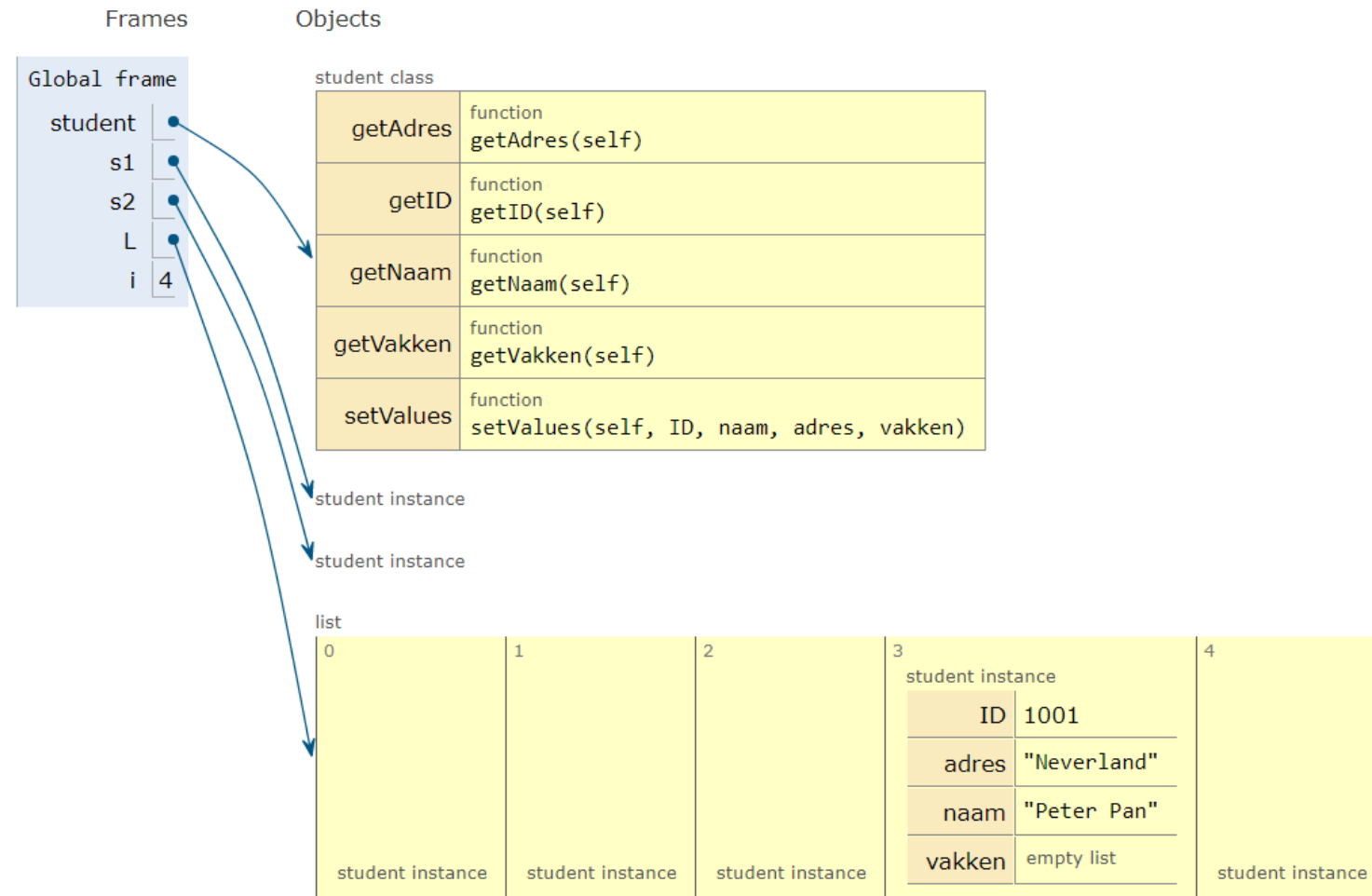


Klassen – aanroepen method

[Python tutor](#)

- Dankzij **self** weet de method op welk object het wordt aangeroepen

```
class student:  
    def setValues(self,ID,naam,adres,vakken):  
        ...  
...  
L[3].setValues(1001,"Peter Pan",  
               "Neverland",[])
```



Klassen - initialiser

- Voor de meeste klassen willen we dat de objecten een initiële waarde krijgen voordat we ze gebruiken.
- Te vermijden:

```
s=student()  
print(s.getNaam())  
s.setValues("001","Toon","UA",[])
```

Traceback (most recent call last):

...

AttributeError: 'student' object has no attribute 'naam'

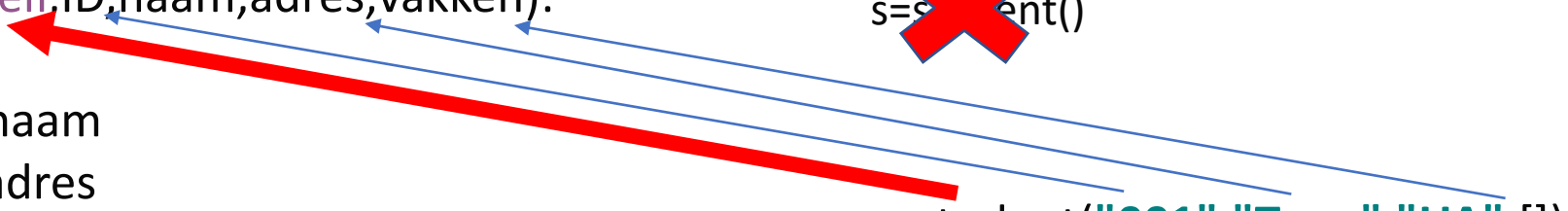
Klassen - initialiser

- Om dit te vermijden werd een speciale functie voorzien die wordt aangeroepen wanneer een nieuw object wordt aangemaakt:

```
class student:  
    def __init__(self, ID, naam, adres, vakken):  
        self.ID=ID  
        self.naam=naam  
        self.adres=adres  
        self.vakken=vakken  
    ...
```

~~s=student()~~

s=student("001", "Toon", "UA", [])
print(s.getNaam())



The diagram illustrates the execution of the `__init__` method. A red arrow points from the `__init__` method definition in the class to the `student()` call in the code snippet. Three blue arrows point from the arguments `"001"`, `"Toon"`, and `"UA"` to the `self` parameter in the `__init__` method signature. A large red 'X' is placed over the `s=student()` line, indicating it is incorrect or deprecated.

Waarom Klassen?

- Encapsulatie: “verpakken” van een stukje functionaliteit
 - Enkele principes:
 - Initializer zorgt voor correcte beginwaarden
 - Data van een klasse wordt enkel via de members aangepast en opgevraagd
 - De gebruiker van de klasse kan de klasse gebruiken zonder te weten hoe die werkt
 - Weet jij hoe Turtle tekent?
 - Gevolg:
 - Aanpassen van datastructuur? Enkel de methods aanpassen!
 - Zie ook Abstract Data Types

Voorbeeld

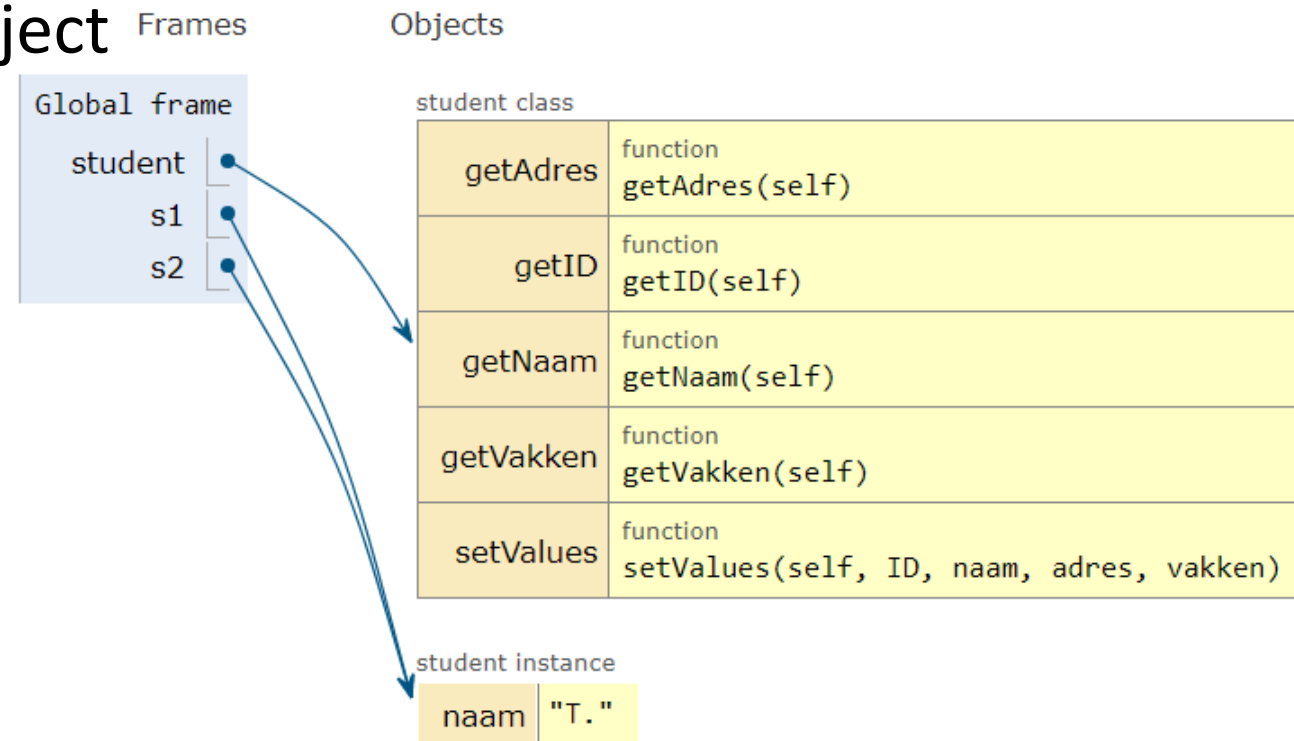
- “Matrix” klasse

Let op! Klassen zijn “mutable”

- Net zoals lijsten kunnen de methodes de inhoud van klassen aanpassen
- Ook bij objecten van een door de gebruiker gedefiniëerde klassen *refereren* variabelen naar het object

```
s1=student("001","Toon","UA",[])  
s2=s1  
s2.naam="T."  
print(s1.getNaam())
```

T.



Vlugge quiz: output van ...

```
class A:
    def __init__(self,s):
        self.set_content(s)

    def get_content(self):
        return self.content

    def set_content(self,s):
        self.content=s

    def double(self):
        self.content+=self.content
```

```
def double(a):
    b=a
    b.double()
    return b
```

```
a=A("abc")
b=double(a)
print(b)
print(a)
```

(A)

Error

(B)

abcabc
abcabc

(C)

abcabc
abc

(D)

lets anders

Vlugge quiz: output van ...

```
class A:
    def __init__(self,s):
        self.set_content(s)

    def get_content(self):
        return self.content

    def set_content(self,s):
        self.content=s

    def double(self):
        self.content+=self.content
```

```
def double(a):
    b=a
    b.double()
    return b
```

```
a=A("abc")
b=double(a)
print(b)
print(a)
```

(A)

Error

(B)

abcabc
abcabc

(C)

abcabc
abc

(D)

lets anders

Vlugge quiz: output van ...

```
class A:
    def __init__(self,s):
        self.set_content(s)

    def get_content(self):
        return self.content

    def set_content(self,s):
        self.content=s

    def double(self):
        self.content+=self.content
```

(D1)

```
<__main__.A object at 0x00000229CB6C5D30>
<__main__.A object at 0x00000229CB6C5D30>
```

```
def double(a):
    b=a
    b.double()
    return b
```

```
a=A("abc")
b=double(a)
print(b)
print(a)
```

(D2)

```
<__main__.A object at 0x0000029846A35D30>
<__main__.A object at 0x0000029846A35518>
```

Vlugge quiz: output van ...

```
class A:
    def __init__(self,s):
        self.set_content(s)

    def get_content(self):
        return self.content

    def set_content(self,s):
        self.content=s

    def double(self):
        self.content+=self.content
```

(D1)

```
<__main__.A object at 0x00000229CB6C5D30>
<__main__.A object at 0x00000229CB6C5D30>
```

```
def double(a):
    b=a
    b.double()
    return b
```

```
a=A("abc")
b=double(a)
print(b)
print(a)
```

(D2)

```
<__main__.A object at 0x0000029846A35D30>
<__main__.A object at 0x0000029846A35518>
```

Vlugge quiz: output van ...

```
class A:
    def __init__(self,s):
        self.set_content(s)

    def get_content(self):
        return self.content

    def set_content(self,s):
        self.content=s

    def double(self):
        self.content+=self.content
```

```
def double(a):
    b=a
    b.double()
    return b

a=A("abc")
b=double(a)
print(b.get_content())
print(a.get_content())
```

(A)

Error

(B)

abcabc

abcabc

(C)

abcabc

abc

(D)

lets anders

Vlugge quiz: output van ...

```
class A:
    def __init__(self,s):
        self.set_content(s)

    def get_content(self):
        return self.content

    def set_content(self,s):
        self.content=s

    def double(self):
        self.content+=self.content
```

```
def double(a):
    b=a
    b.double()
    return b

a=A("abc")
b=double(a)
print(b.get_content())
print(a.get_content())
```

(A)

Error

(B)

abcabc

abcabc

(C)

abcabc

abc

(D)

lets anders

Voorbeeld: botsende bal

- Canvas waarop getekend wordt
- Functie move om bal te bewegen

Samenvatting

- *Klassen* groeperen data en code
- Een *instantie* van een klasse is een *object*
- Variabelen *refereren* naar objecten (net zoals bij lijsten, dicts, ...)
- Dit staat ons toe meer gestructureerd, modulair te programmeren
 - Leesbaardere code
 - Makkelijker debuggen
 - Gebruiker van een klasse moet de interne werking van de klasse niet kennen