

Inleiding tot Programmeren

Toon Calders

toon.calders@uantwerpen.be

Les 7a: Vervolg klassen

Wat zagen we vorige keer?

- Klassen: zelf datatypes maken in Python
 - Self
 - Methods
 - `__init__`

Vervolg klassen

- Runtime testen van type, klasse
- Deepcopy bij klassen
- Inheritance

Runtime testen van type, klasse

- We kunnen at runtime types testen
 - `type(object)`: geeft het type van een object
 - Handig om diagnostische informatie te geven
 - `isinstance(object,(class,class,...))`: true als object een instantie is van een van de klassen
 - Kan gebruikt worden om een “guardian” te maken
 - Handig bij een “assert”

Isinstance voorbeeld

```
class button:
    def __init__(self, label):
        self.label=label

    def display(self):
        print(self.label)
```

```
b=button("quit")
print(type(b))
print(isinstance(b,(button)))

i=5
print(type(i))
print(isinstance(i,(button)))
```

```
<class '__main__.button'>
True
<class 'int'>
False
```

Guardian: voorbeeld

```
def fn(n):  
    if n <= 2:  
        return 1  
    return fn(n - 1) + fn(n - 2)  
  
def guardedfib(n):  
    if not isinstance(n,int):  
        print("only works for integers!")  
        return None  
    elif n<0:  
        print("only works for integers >= 0")  
        return None  
  
    return fn(n)  
  
print(guardedfib(b))  
print(guardedfib(3.5))  
print(guardedfib(-3))  
print(guardedfib(3))
```

only works for integers!
None
only works for integers!
None
only works for integers >= 0
None
2

Assertions

```
def printRij(rij):  
    assert(isinstance(rij,(list)))  
    for x in rij:  
        print(x,end='\t')  
  
matrix=[1,0,0]+[0,1,0]+[0,0,1]  
printRij(matrix[0])
```

```
File "C:/Users/Toon/PycharmProjects/les7/instance.py", line 41, in printRij  
    assert(isinstance(rij,(list)))  
AssertionError  
  
Process finished with exit code 1
```

is versus ==

- Bij vergelijken van twee variabelen:
 - “==” test op gelijke waarde
 - “is” test op referentie naar zelfde object

```
L=[1, 2, 3, 4]
```

```
K=L
```

```
U=[1, 2, 3, 4]
```

```
print("==", K==L, K==U)
```

```
print("is", K is L, K is U)
```

== True True

is True False

is versus ==

- Bij vergelijken van twee variabelen:
 - “==” test op gelijke waarde
 - “is” test op referentie naar zelfde object
- “==” doet een “diepe” test op gelijkheid

```
LL=[[0],[1],[2]]  
KK=LL  
UU=[[0],[1],[2]]
```

```
print("==", KK==LL, KK==UU)  
print("is", KK is LL, KK is UU)
```

== True True
is True False

“is” bij immutable types

- Twee strings, booleans, ints zijn gelijk volgens “is” als ze dezelfde waarde hebben

```
print(1 is 1, "abc" is "abc")  
s="def"  
t="def"  
print(s==t, s is t)
```

The diagram illustrates the execution of the code. Red arrows originate from the 'is' operators in the code and point to the 'True' outputs. Specifically, an arrow from the first 'is' in '1 is 1' points to the first 'True' in the first line's output. Another arrow from the second 'is' in '"abc" is "abc"' points to the second 'True' in the first line's output. A third arrow from the 'is' in 's is t' points to the second 'True' in the second line's output. A fourth arrow from the '==' operator in 's==t' points to the first 'True' in the second line's output.

True True
True True

Deepcopy werkt ook bij klassen

- De functies `copy` en `deepcopy` uit de module `copy` kunnen ook voor zelf-gedefiniëerde klassen gebruikt worden
 - `copy` maakt een ondiepe kopij (nieuw object, maar member variabelen refereren naar dezelfde objecten)
 - `deepcopy` maakt een diepe kopij
 - Deze functies maken gebruik van “meta-informatie”; dat is: informatie over het programma zelf

Voorbeeld: copy en deepcopy

```
class A:
    def __init__(self, L):
        self.lijst=L

    def geef_lijst(self):
        return self.lijst
```

Voorbeeld: copy en deepcopy

```
class A:  
    def __init__(self,L):  
        self.lijst=L  
  
    def geef_lijst(self):  
        return self.lijst
```

```
a=A([0,1,2,3])  
b=a  
b.geef_lijst()[3]=4  
print(a.geef_lijst()) → [0, 1, 2, 4]  
print(a is b) → True
```

Voorbeeld: copy en deepcopy

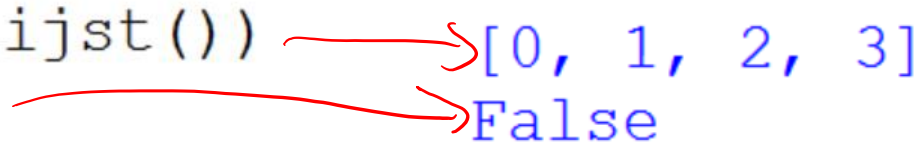
```
class A:  
    def __init__(self,L):  
        self.lijst=L  
  
    def geef_lijst(self):  
        return self.lijst
```

```
a=A([0,1,2,3])  
b=copy(a)  
b.geef_lijst()[3]=4  
print(a.geef_lijst()) → [0, 1, 2, 4]  
print(a is b) → False
```

Voorbeeld: copy en deepcopy

```
class A:  
    def __init__(self,L):  
        self.lijst=L  
  
    def geef_lijst(self):  
        return self.lijst
```

```
a=A([0,1,2,3])  
b=deepcopy(a)  
b.geef_lijst()[3]=4  
print(a.geef_lijst())  
print(a is b)
```



[0, 1, 2, 3]
False

Deepcopy is subtiel

```
class Boek:
    def __init__(self, titel, d=None):
        self.doos=d
        if d!=None:
            d.voeg_boek_toe(self)
        self.titel=titel

    def set_doos(self, d):
        self.doos=d

    def get_doos(self):
        return self.doos

    def geef_weer(self):
        print(self, self.titel, " in doos ", self.doos)
```

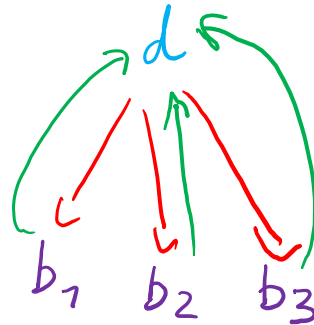
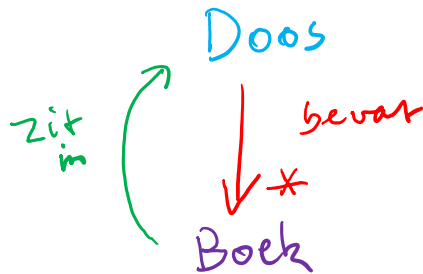
```
class Doos:
    def __init__(self):
        self.content=[]

    def voeg_boek_toe(self, b):
        self.content.append(b)
        b.set_doos(self)

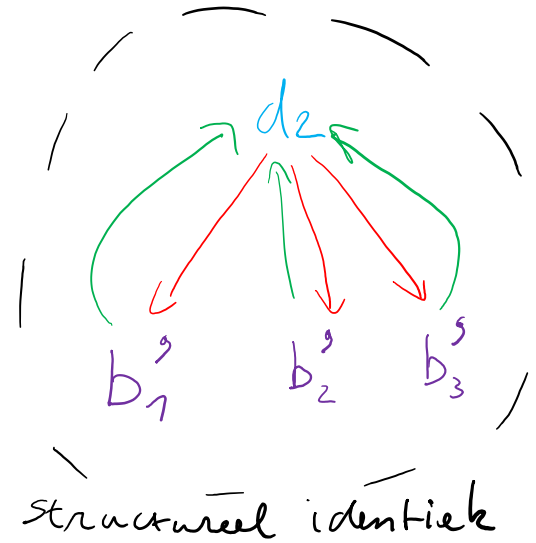
    def print_inhoud(self):
        print("Inhoud van ", self)
        for x in self.content:
            x.geef_weer()
```

Deepcopy is subtiel

```
d=Doos()  
b1=Boek("Python", d)  
b2=Boek("C++", d)  
b3=Boek("Java", d)
```



```
d2=deepcopy(d)
```



Overerving (*Inheritance*) *

- We kunnen klassen afleiden van een andere klasse
 - “kopiëren” van functionaliteit
 - We kunnen objecten van de nieuwe klasse gebruiken alsof ze een instantie zijn van de oude klasse
- B afgeleid van A: B is subklasse van superklasse A
- We kunnen ook methods overschrijven door ze te herdefiniëren
 - Indien het gedrag anders dient te zijn voor de subklasse
 - We kunnen nog steeds aan de oude functie via de aanroep:
naam van superklasse.naam van functie

Overerving

*

```
class dier:
    def __init__():
        self.geluid=""

    def maak_geluid(self):
        print(self.geluid)
```

```
class hond(dier):
    def __init__(self):
        self.geluid="woef"
```

Subklasse van dier

```
class kat(dier):
    def __init__(self):
        self.geluid="miauw"
```

```
k=kat()
h=hond()
```

```
k.maak_geluid()
h.maak_geluid()
```

Overerving

*

```
class dier:
    def __init__():
        self.geluid=""

    def maak_geluid(self):
        print(self.geluid)
```

```
class hond(dier):
    def __init__(self):
        self.geluid="woef"
```

Subklasse van dier

```
class kat(dier):
    def __init__(self):
        self.geluid="miauw"
```

```
k=kat()
h=hond()
```

```
k.maak_geluid() → miauw
h.maak_geluid() → woef
```

Polymorfisme

*

- Polymorfisme: gelijkvormigheid in interface tussen verschillende klassen, maar met een verschillende implementatie:

```
L=[kat(),kat(),hond(),hond(),hond(),dier()]\nfor x in L:\n    x.maak_geluid()
```

- We hoeven in dit geval zelfs niet te weten tot welke klasse het object x uit de lijst L behoort, zolang ze maar dezelfde methodes implementeren

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