Programmeermethoden NA

Week 7: OOP & modules

Kristian Rietveld

http://liacs.leidenuniv.nl/~rietveldkfd/courses/prna2016/

Universiteit Leiden
The Netherlands

Tweede programmeeropdracht

```
import sys

def is_cijfer(kar):
    return kar >= "0" and kar <= "9"

def coderen(invoer, uitvoer):
    kar = invoer.read(1)
    while kar:
        if not is_cijfer(kar):
            uitvoer.write(kar)
        kar = invoer.read(1)

invoer = open("test.txt", "r")
coderen(invoer, sys.stdout)
```

Tweede programmeeropdracht

- Voor het maken van het histogram, gebruik bijv. een lijst en schrijf een nette functie om een histogram af te drukken. Denk aan `.format()` en `40 * "="`.
- Omdraaien getal: denk bijvoorbeeld aan voorbeeld van vorige week. Of loop met behulp van `reversed()`.
- Let op de opmerkingen in de opdracht, met name op de functies waarvan we verwachten dat jullie die zelf schrijven.
- Lees elk karakter uit het invoerbestand maar 1 keer, je mag de karakters niet terugzetten.
- Let op de normering, deze is verduidelijkt.

Tweede programmeeropdracht

Uiteraard verwachten we weer een LaTeX verslag.

- Zorg je dat kunt printen!
- Uitgebreider dan vorige keer!
  - Uitleg programma.
  - Defintie en uitleg Lychrel-getal.
  - Tijdsverantwoording met behulp van een tabel.
- Zie ook het zevende werkcollege.

OOP

```
invoer = open("bestand.txt", "r")
...
letter = invoer.read(1)
...
invoer.close()

lijst = list()
lijst.append(143)
lijst.append(542)
```

In dit programma maken we een object `invoer` van type `file`. We lezen uit het geopende bestand door de *methode* `read` te gebruiken. Tevens maken we een object `lijst` van type `list` waar we met behulp van de methode `append` elementen aan toevoegen.

OOP

In Python is het mogelijk om **object georiënteerd** (OO) te programmeren. OOP: Object Oriented Programming.

Binnen een OO-programma hebben we **objecten** (*trein*, *bus*, *donald*) van verschillende **klassen** (*Voertuig*, *Eend*). Klassen hebben hun eigen **methoden** (*deurenSluiten*, *stoppen*, *kwaken*). In plaats van **klasse** van een **object**, spreken we ook wel over het **type** van een **object**.

Een voorbeeld: de klasse `complex`. Met `z = complex(4, 3)` maken we een object `z` met type `complex`. `print z` vraagt aan `z` om zich af te laten drukken. En `z = z + z` vraagt `z` om zich met zichzelf op te tellen.

De file en list-objecten hebben ook methoden: `close`, `append`, enz.

Klassen schrijven

```
class Wagon(object):
    def __init__(self, hoogte, breedte, lengte):
        self.hoogte = hoogte
        self.breedte = breedte
        self.lengte = lengte

    def inhoud(self):
        return self.hoogte * self.breedte * self.lengte


geel = Wagon(3.5, 4.0, 20.5)
print "Inhoud", geel.inhoud()
```

Hier is `geel` een *object* van *type* (= *klasse*) Wagon.

Klassen schrijven (2)

In een klasse kunnen attributen worden opgeslagen. Hier zijn dat `hoogte`, `breedte`, `lengte`. Later mogen nog meer attributen worden toegevoegd.

`inhoud` is een member-functie. Binnen member-functies is het eerste argument altijd `self`, welke een instantie van de klasse bevat. Met behulp van `self` kunnen de attributen van deze instantie worden uitgelezen. Let op de punt-notatie!

De methode `__init__` is een methode die automatisch wordt aangeroepen bij het instantieren van de klasse.

Klassen schrijven (3)

We kunnen nog een klasse maken. We mogen hiervan de namen hergebruiken.

```
class Tanker(object):
    def __init__(self, straal, lengte):
        self.straal = straal
        self.lengte = lengte

    def inhoud(self):
        return math.pi * self.straal * self.straal \
                    * self.lengte

t = Tanker(10.0, 30.0)
# Roept aan: inhoud in Tanker, want t
# is een tanker.
print t.inhoud()
```

Readers en writers

Hoewel buitenstaanders de attributen van een klasse direct mogen uitlezen, gaat het vaak met speciaal geschreven methoden: *reader* (getter, accessor) en *writer* (setter, mutator).

```
class Tanker(object):
    # ... wat we al hadden ...

    def geefStraal(self):
        return self.straal

shell = Tanker(10.0, 30.0)
print shell.geefStraal()
```

En analoog voor writers. We noemen dit ook wel "Encapsulation".

Readers en writers (2)

Nog een uitbreiding voor `Tanker`:

```
def geefDiameter(self):
    return 2.0 * self.straal

def maaklang(self, t):
    self.lengte = t
```

Overerving (inheritance)

De klasse `PersonenWagon` wordt *afgeleid* (= *derived*) van de *ouder* (= *superklasse*) `Wagon`:

```
class PersonenWagon(Wagon):
    # We erven "alles" van Wagon
    def __init__(self, passagiers=0):
        # Initialiseer superklasse
        super(PersonenWagon, self).__init__(4.5, 4.0, 22.5)
        self.passagiers = passagiers

    def hoeveel(self):
        return self.passagiers

dubbelDek = PersonenWagon(450)
print dubbelDek.hoeveel()
print dubbelDek.inhoud()
```

Overerving (2)

Stel we hebben een klasse `Voertuig` met variabelen `gewicht` en `maxsnelheid` en een methode `belasting`. Er zijn afgeleide klassen `fiets` (met eigen methode `belasting`) en `auto` (met een extra variabele `soort`).

Met object `rijwiel` van type `fiets` mag je gebruik maken van `rijwiel.belasting()`. Je krijgt dan de belasting speciaal voor een fiets.

Let erop dat je zelf de superklasse moet initialiseren.

Voorbeeld

Stel we willen een klasse schrijven om breuken op te slaan en mee te rekenen. We kunnen daartoe de volgende klasse schrijven:

```
class Breuk(object):
    def __init__(self, teller, noemer):
        self.teller = teller
        self.noemer = noemer

    def geefTeller(self):
        return self.teller

    def geefNoemer(self):
        return self.noemer

    def vereenvoudig(self):
        # TODO
        pass

    def telop(self, breuk):
        self.teller = self.teller * breuk.geefNoemer() + \
            breuk.geefTeller() * self.noemer
        self.noemer *= breuk.geefNoemer()

        self.vereenvoudig()

    def drukaf(self):
        print "{}/{}".format(self.teller, self.noemer)
```

Voorbeeld (2)

Vervolgens kunnen we een programma schrijven als:

```
b = Breuk(1, 4)
b.telop(Breuk(1, 2))
b.drukaf()
b.vereenvoudig()
```

Voorbeeld (3)

Met een kleine aanpassing kunnen we zelfs breuken optellen met de operator + en breuken direct netjes printen!

```
class Breuk(object):
    # ... wat we al hadden ...
    def telop(self, breuk):
        nieuwTeller = self.teller * breuk.geefNoemer() + \
            breuk.geefTeller() * self.noemer
        nieuwNoemer = self.noemer * breuk.geefNoemer()

        nieuw = Breuk(nieuwTeller, nieuwNoemer)
        nieuw.vereenvoudig()
        return nieuw

    def __add__(self, other):
        return self.telop(other)

    def __str__(self):
        return "{}/{}".format(self.teller, self.noemer)
```

Vervolgens is mogelijk:

```
b1 = Breuk(1, 4)
b2 = Breuk(1, 2)
b3 = b1 + b2
print b3
```

Geavanceerde onderwerpen

- Operator overloading
- Multiple inheritance
- Polymorfisme, late binding
- Klassen als iterators
- Class methods
- Properties

Modules en packages

- Tot nu toe alleen programma's geschreven die bestonden uit een enkel bestand.
- Code kun je verspreiden over meerdere `.py`-bestanden.
- Hoe roepen we een functie uit een ander `.py`-bestand aan?

`import` statement

- Een functie moet gedefinieerd zijn, voordat je deze in Python kunt aanroepen.
- Functies uit andere bestanden eerst *importeren*.
- We noemen dit soort andere bestanden "modules".
- Bundeling van modules: "package".

import statement (2)

```
# importeer de gehele module
import random

c = random.randint(1, 100)
```

Of:

```
# importeer een specfieke functie uit een module
from random import randint

# We hoeven nu niet de prefix "random." te gebruiken
c = randint(1, 100)
```

import statement (3)

```
# importeer de gehele module, maar onder een
# afgekorte naam
import random as R

c = R.randint(1, 100)
```

Zelf modules maken

- Hoe maken we nu zelf een module?
- Maak een aparte `.py`-bestand met daarin functies.
  - Let op: gebruik geen streepjes of spaties in de bestandsnaam!
- Importeer het bestand met `import`. Laat dan `.py` uit de naam weg.

Voorbeeld

Stel we maken het volgende bestand **`handig.py`**:

```
def hallo( ):
    print "hello world"

def telop(a, b):
    """Telt getallen a en b bijelkaar op"""
    return a + b

def vermenigvuldig(a, b):
    return a * b
```

Voorbeeld (2)

En in dezelfde directory een bestand met eens Python programma:

```
# importeer de gehele module,
# let op we laten ".py" weg!
import handig

handig.hallo()
c = handig.telop(a, b)
c = handig.vermenigvuldig(a, b)
```

Of

```
# importeer de gehele module,
# maar onder een afgekorte naam
import handig as h

h.hallo()
c = h.telop(a, b)
```

Voorbeeld (3)

Nog mooier, een bestand `breuk.py` met daarin:

```
class Breuk(object):
    def __init__(self, teller, noemer):
        self.teller = teller
        self.noemer = noemer

    def geefTeller(self):
        return self.teller

    def geefNoemer(self):
        return self.noemer

    def vereenvoudig(self):
        # TODO
        pass

    def telop(self, breuk):
        nieuwTeller = self.teller * breuk.geefNoemer() + \
            breuk.geefTeller() * self.noemer
        nieuwNoemer = self.noemer * breuk.geefNoemer()

        nieuw = Breuk(nieuwTeller, nieuwNoemer)
        nieuw.vereenvoudig()
        return nieuw

    def __add__(self, other):
        return self.telop(other)

    def __str__(self):
        return "{}/{}".format(self.teller, self.noemer)
```

Voorbeeld (4)

De code om met breuken te werken staat nu netjes in een apart bestand. We kunnen dit gebruiken in een programma dat we aan het schrijven zijn:

```
from breuk import Breuk

b1 = Breuk(1, 4)
b2 = Breuk(1, 2)
b3 = b1 + b2
print b3
```

Tot slot

- A.s. vrijdag: deadline opdracht 2!
- Werkcollege: werken aan de opdracht.
- Vragenuren: dinsdagmiddag, donderdagmiddag circa 15:30 - 17:00
- Op vrijdag zal er na 11:30 ook enige assistentie zijn.

- **Volgende week: geen Programmeermethoden!**