

Programmeermethoden NA

Week 4: Files & Functies

Kristian Rietveld

<http://liacs.leidenuniv.nl/~rietveldkfd/courses/prna/>



Universiteit
Leiden

Vorige week: Controlestructuren

```
# gegeven
# 0 <= uur <= 24

if uur < 5:
    print "goedenacht"
elif uur < 12:
    print "goedemorgen"
elif uur < 18:
    print "goedemiddag"
else:
    print "goedenavond"
```

```
for i in range(10):
    print i, "--", i * i
```

```
x = 1
while x < 100:
    print x
    x += 2
```

Overzicht komende weken

- **Week 4:** 25 - 29 september: Files en functies.
 - (2 – 6 oktober: geen college PRna)
- **Week 5:** 9 - 13 oktober: Functies, vervolg & lijsten.
- **Week 6:** 16 - 20 oktober: Lijsten en algoritmes.
 - (23 – 27 oktober: geen college PRna)
- **Week 7:** 30 – 3 november: OOP, modules.
 - **Deadline opdracht 2:** vrijdag 3 november, 17:00 uur.

Files

- Invoer en uitvoer voor programma's staan vaak in bestanden, bijvoorbeeld `iets.cc`, `uitvoer.txt`, `mijnexperiment.csv`.
 - Invoer (`raw_input()`) en uitvoer (`print`) via het terminalvenster gaat eigenlijk ook via bestanden.
- Voor de tweede programmeeropgave gaan we een Python-programma schrijven dat een invoerbestand leest, de inhoud codeert (versleutelt) en schrijft naar een uitvoerbestand. En natuurlijk kan het programma ook ontcijferen.
- Later zal het lezen en schrijven van bestanden een belangrijk deel uitmaken van je "workflow": werken met data-bestanden die bijvoorbeeld resultaten van experimenten bevatten.

Files

```
invoer = open("jefile.txt", "r")  
uitvoer = open("onzeuitvoer.txt", "w")
```

...

```
letter = invoer.read(1)  
uitvoer.write(letter)  
print >>uitvoer, "hello world"
```

...

```
invoer.close()  
uitvoer.close()
```

File-objecten

- Met de functie `open` maken we een `file` object:

```
f = open(bestandsnaam, modus)
```

- Modus is `"r"` voor lezen, `"w"` voor schrijven.
- `f` is een object van type `file`. Met een object kun je bepaalde dingen doen door "memberfuncties" ("methoden") aan te roepen.
 - Bijvoorbeeld `".read(1)"` of `".readline()"`
 - Naam van object, gevolgd door een punt, gevolgd door de naam van de methode en argumenten.
- `sys.stdin` is het `file`-object voor terminalinvoer, `sys.stdout` voor uitvoer.

Tekstbestanden kopiëren

- Een *tekstbestand* (zoals elk Python-programma) bestaat uit regels gescheiden door regelovergangen (bij UNIX LF, Windows CR-LF). Meestal heeft de laatste regel ook een regelovergang. Daarna komen we aan bij het einde van het bestand: end-of-file, afgekort EOF.
- Met de volgende loop kopiëren we de invoer regel-voor-regel naar de uitvoer:

```
regel = invoer.readline()  
while regel != "":  
    uitvoer.write(regel)  
    regel = invoer.readline()
```

- Maar in Python kan het nog korter:

```
for regel in invoer:  
    uitvoer.write(regel)
```

Kopiëren aanpassen

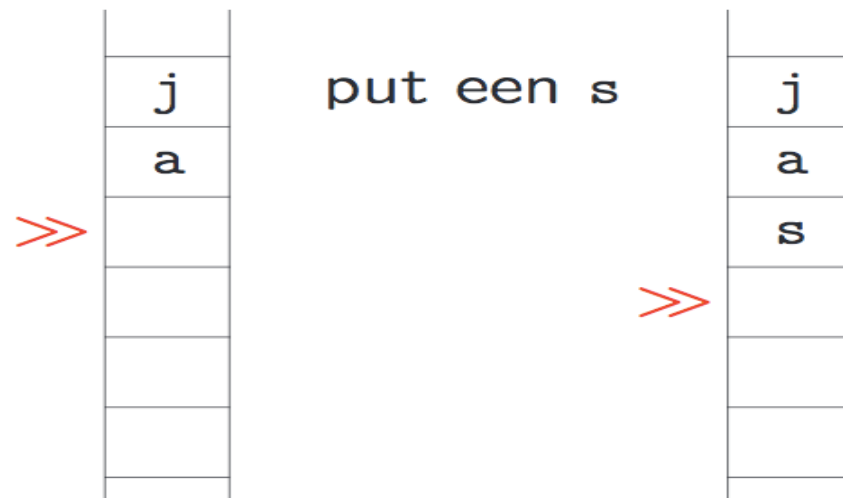
- We kunnen het basis kopieerprogramma stap voor stap wijzigen:

```
for regel in invoer:
    for kar in regel:
        # Wijzig dit voor de tweede opdracht
        if kar != "\n":
            uitvoer.write(kar)
```

- Bijvoorbeeld om alle regelovergangen weg te laten.

File pointers

- Files werken eigenlijk met "file pointers". De file pointer "wijst" naar de plek (een karakter) in het bestand waar we zijn met lezen en schrijven.
 - Denk aan een oude videoband en de leeskop.
 - `readline()` leest een regel vanaf de pointer en schuift de pointer daarna met de lengte van de gelezen regel op.
 - `write()` zet een n-aantal karakters neer en schuift ook n karakters op.



Schrijven naar bestanden

- Om te schrijven naar een `file` object kun je gebruik maken van de `write` methode of van `print`.
- Bij `write` **moet** de parameter een string zijn:

```
f.write("hello world")  
f.write(42)           # NEE!  
f.write(str(42))     # OK
```

Schrijven naar bestanden (2)

```
f = open("uitvoer.txt", "w")
print >>f, "Met print is het eenvoudiger"
print >>f, "Geheel getal: {0} Floating point: {1}." \
        .format(51, 3.1412345)
f.close()
```

Nog wat algemener

```
import sys
```

```
filename = raw_input()
```

```
try:
```

```
    invoer = open(filename, "r")
```

```
except IOError:
```

```
    print filename, "niet te openen!"
```

```
    sys.exit(1)
```

- We hebben try en except nodig om de error (exception) die wordt gegooid in geval van een fout op te vangen.

Tweede programmeeropgave: Geheim

- Voor de tweede programmeeropgave moet je een Python-programma schrijven dat de inhoud van een bestand kan ontcijferen:

xxaxxaDaaieetxexe xxxxaaiqqoosoo eeefaaefuonoo o%kktmmeqqsrrtomm!

moet worden (gegeven geheim getal $k=2$):

Dit is een test!

- En vice versa ...
- En is 196 een Lychrel-getal?

<http://liacs.leidenuniv.nl/~rietveldkfd/prna2017/opdr2.html>

Functionies

- Een **functie** is een zwarte doos waar iets in gaat en iets (anders) uitkomt.
 - "Machientjes", "Machientjesschema".
- Een goed gestructureerd Python-programma bestaat uit een verzameling van functies. Het uitvoeren van het programma begint bij de "globale" statements die niet in functies staan.
- Een functie moet worden gedefinieerd voordat deze kan worden aangeroepen.

Funcities (2)

- Sommige functies rekenen iets uit. Bijvoorbeeld: bereken het kwadraat van x en geef dit als antwoord.
- Andere functies verrichten een taak: druk een tabel af op het scherm, zet het infoblokje op het scherm. De functie geeft geen "antwoord" of waarde terug.
- Functies hebben nul of meer **parameters** of argumenten.
- De waarde die de functie oplevert noemen we de "returnwaarde" (return value).

Func tiedefinities

```
def functienaam(arg1, arg2, ..., argn):  
    blok van statements (met inspringen )  
    return iets
```

- Als we geen waarde willen teruggeven (retourneren), laten we het return-statement weg, of gebruiken we return zonder iets daarachter.

Voorbeeld

- Een functie om een taak te verrichten (geen returnwaarde):

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

- Een functie om iets uit te rekenen:

```
def inhoud(l, b, h):  
    return l * b * h
```

- Het aanroepen van deze functies gaat als volgt:

```
tekst_op_scherm()  
print inhoud(16, 37, 42)
```

Werking

- Hoe werken die functies nu precies?
 - Wanneer je een functie aanroept, dan onthoud je waar je bent gebleven.
 - Vervolgens "spring" je direct naar het begin van de desbetreffende functie. De parameters worden netjes doorgegeven.
 - De functie wordt regel voor regel uitgevoerd, totdat je bij het einde van de functie komt *of* tot het eerste return-statement dat je tegenkomt (welke het eerst komt wordt gekozen).
 - In dat geval "spring" je weer terug, naar waar je was gebleven (het onthouden "returnadres").
- Als je aan het programmeren bent, is het eenvoudiger om te denken in termen van "deze functie verricht deze taak".
- Alle functie-aanroepen komen op een **stapel** terecht, samen met de returnadressen.
 - Een aanroep voegt aan de stapel toe.
 - Een return haalt de bovenste van de stapel af.

Variabelen

```
# bereken inhoud van blok l bij b bij h
def inhoud(l, b, h):
    temp = 0
    temp = l * b * h
    return temp
```

- l , b , h , $temp$: lokale variabelen.
- Hun scope (waarin deze beschikbaar zijn) is de functie `inhoud`.
- l , b en h zijn de **formele** parameters van de functie en krijgen als startwaarde de waarde van de corresponderende **actuele** parameters.
- Bij de aanroep `t = inhoud(b, 5, x)` zijn b , 5 en x de actuele parameters.

Voorbeelden

- De volgende functie bepaalt of de parameter `jaar` een schrikkeljaar is:

```
def schrikkel(jaar):
```

```
    return jaar % 4 == 0 and (jaar % 400 == 0 or jaar % 100 != 0)
```

- 1963 niet, 2016 wel, 2017 niet, 2000 wel, 2100 niet.

Opletten!

- We moeten oppassen met het feit dat de types van de argumenten niet actief worden gecheckt.
- Verantwoordelijkheid ligt bij de aanroeper.

```
def telop(a, b):  
    c = a + b  
    return c
```

```
q = telop(12354, "hallo!!")
```

Meerdere returnwaarden

- We kunnen op eenvoudige wijze meerdere waarden retourneren vanuit een functie.
- We maken hierbij gebruik van een tuple, een eindige geordende rij van objecten.

```
def meerdere(a, b, c):  
    return (a, a + b, a + b + c)
```

Voorbeeldaanroepen:

```
x, y, z = meerdere(1, 2, 3)  
t = meerdere(1, 2, 3)  
print t[0], t[1], t[2]
```

Geen returnwaarde

- Als je een functie hebt **zonder** returnwaarde, dan hoef je geen return-statement op te geven.
- Het mag wel, gebruik dan return zonder iets erachter.

```
def print_even(x):  
    if x % 2 == 0:  
        print "EVEN!"  
        return  
  
    print "ONEVEN"
```

Commentaar

Boven iedere functie wordt duidelijk commentaar verwacht:

```
# vereenvoudig breuk teller/noemer zoveel mogelijk  
# aanname teller >= 0, noemer > 0
```

Wat schrijven we op?

- Maak een zin waarin de functienaam en de namen van de parameters voorkomen.
- Schrijf op wat vooraf geldt en wat na afloop (pre- en post-conditie).

Docstrings

Je kunt de documentatie van een functie als commentaar boven de functie zetten, maar je kunt ook gebruik maken van een speciale Python-feature, "docstrings":

```
def mijn_functie(x, y, z):  
    """Telt x en y bij elkaar op en vermenigvuldigt met z.  
    Returnwaarde: het resultaat van de berekening."""  
  
    result = (x + y) * z  
    return result
```

Als eerste statement van een functie mag een dergelijke docstring worden opgegeven.

Voorbeeld functieaanroepen

```
def hoogop(x):  
    x = x + 10  
    print x  
def maaknul(t):  
    t = 0  
    print t
```

```
x, m, q = 7, 3, 5  
hoogop(x)          # 17  
print x            # 7  
hoogop(m+8)        # 21  
print m            # 3  
maaknul(q)         # 0  
print q            # 5  
maaknul(42)        # 0
```

De waarde van de actuele parameter wordt doorgegeven aan de formele parameter. Er is een lokale "kopie" in de functie.

Globale structuur Python-programma

- Er zijn in Python weinig eisen voor de structuur van een programma.
 - Code hoeft niet verplicht in een functie te staan.
 - Er hoeft niet per se een "hoofd" of "main" functie te zijn.
- We mogen "globale" code en code in functies mengen, maar we lopen het risico dat de code onoverzichtelijk wordt.
- Laten we voorstellen om alle code sowieso in een functie te plaatsen.

Voorstel globale structuur

- `import` statements bovenaan het bestand.
- Daarna 'constante' variabelen.
- Zet alle code in functies.
- Maak ook een main (hoofd) functie, dat we zullen gebruiken als startpunt voor het programma.
- Een functie moet zijn gedefinieerd *voordat* deze kan worden aangeroepen.
- *(Later: functies verspreiden over meerdere bestanden).*

Voorstel globale structuur (2)

```
# Eerst alle import statements
```

```
import sys
```

```
# Dan alle hulpfuncties
```

```
def hulpfunctie(a):
```

```
    print "Hello world, a=", a
```

```
# De main-functie
```

```
def main():
```

```
    q = 10354
```

```
    hulpfunctie(q)
```

```
    return 0
```

```
# En tenslotte de "globale" code die main aanroept.
```

```
if __name__ == "__main__":
```

```
    sys.exit(main())
```

Tot slot

- Werkcollege:
 - Oefenen met functies.
 - Werk daarna aan de tweede programmeeropgave.
De deadline is op **vrijdag 3 november 2017**.
- Huiswerk: lees dictaat t/m hoofdstuk 9.

Programmeermethoden NA

Week 4



Universiteit
Leiden