Programmeermethoden NA

Week 5: Functies (vervolg)

Kristian Rietveld

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

Universiteit Leiden
The Netherlands

Functies

Vorige week bekeken we functies:

```
def bereken(a, x):
    return a * (x ** 2)
```

Voorbeeldaanroep:

```
x = 5
y = bereken(10, x)
```

Soorten parameters

Het is belangrijk om de verschillende soorten parameters uit elkaar te houden:

- globaal -- bestaan overal ("globale scope")
- lokaal -- bestaan alleen binnen een functie
- formeel -- staan in de functie-definitie
- actueel -- bij de aanroep van een functie

Variabelen

```
# bereken inhoud van blok l bij b bij h
def inhoud(l, b, h):
    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.

Globaal vs lokaal

Wat is de uitvoer van het volgende programma? Let op het verschil tussen globale en lokale variabelen.

```
x = 195

def test(q):
    # Let op: hier wordt een
    # *lokale* variabele x gemaakt.
    x = q
    print x

print x      # 195
test(2314)   # 2314
print x      # 195
```

Globaal vs lokaal (2)

Een kleine aanpassing: we maken geen lokale variabele x aan.

```
x = 195

def test(q):
    print x

print x      # 195
test(2314)   # 195
print x      # 195
```

Bij het zien van de variabele `x` in `test` gaat Python eerst op zoek in de lokale "scope". Hier wordt geen `x` gevonden en dan kijkt Python verder in de globale "scope" waar wel `x` wordt gevonden.

Globaal vs lokaal (2)

Maar wat nu als we binnen een functie aan een globale variabele willen toekennen en dus *geen* lokale variabele willen maken?

```
x = 195

def test(q):
    global x
    x = q
    print x

print x      # 195
test(2314)   # 2314
print x      # 2314
```

Belangrijk: doe dit in principe *niet*, alleen wanneer echt nodig. Het liefst gebruik je helemaal geen globale variabelen (alleen voor constanten, waaraan dus vanuit functies niet wordt toegekend).

Het gebruik van veel globale variabelen wordt als "bad practice" gezien.

Voorbeeld globaal vs. lokaal

```
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.

Opletten

De volgende code wordt geweigerd:

```
a = 5
b = 13

def kwadraat(a):
    a = a ** 2
    b = b + 1
    print "0: {} en {}".format(a, b)

kwadraat(a)
print "1: {} en {}".format(a, b)

a, b = 2, 7
kwadraat(b)
print "2: {} en {}".format(a, b)
```

In de functie `kwadraat` lezen en schrijven we naar `b` binnen hetzelfde statement. `b` wordt hier als lokale variabele gezien en heeft nog geen waarde.

Regel: als er ook maar ergens naar een variabele wordt geschreven, wordt er standaard vanuit gegaan dat dit een lokale variabele betreft.

Opletten (2)

Als we de globale b willen gebruiken, moeten we dat expliciet aangeven:

```
a = 5
b = 13

def kwadraat(a):
    global b
    a = a ** 2
    b = b + 1
    print "0: {} en {}".format(a, b)

kwadraat(a)
print "1: {} en {}".format(a, b)

a, b = 2, 7
kwadraat(b)
print "2: {} en {}".format(a, b)
```

Dit levert:

```
0: 25 en 14
1: 5 en 14
0: 49 en 8
2: 2 en 8
```

GGD bepalen

De grootste gemene deler (GGD) van twee positieve gehele getallen (>= 0, niet beide 0) wordt met het algoritme van Euclides als volgt berekend:

```
def ggd(x, y):
    while y != 0:
        rest = x % y
        x = y
        y = rest
    return x
```

Voorbeeldaanroepen:

```
print ggd(15, 21)
z = ggd(z, 7)  # type(z) == 'int'
```

Vereenvoudigen

```
# vereenvoudig breuk teller/noemer zoveel mogelijk
# aanname teller >= 0, noemer > 0
def vereenvoudig(teller, noemer):
    deler = ggd(teller, noemer)
    if deler > 1:
        teller = teller / deler
        noemer = noemer / deler

    return teller, noemer
```

Voorbeeldaanroep:

```
tel, noem = 15, 21
tel, noem = vereenvoudig(tel, noem)
print tel, noem
```

Top-down, bottom-up

- Bij **bottom-up** bedenk je eerst de functies voordat je deze nodig hebt.
  - Zoals hierboven. We schreven uit interesse eerst een functie GGD. Daarna gebruikten we deze in de functie vereenvoudig.
- Bij **top-down** programmeren maak je een functie zodra je deze nodig hebt.
- Voorbeeld: berekenen faculteit `y = x!`.
  - Bij bottom-up gebruik je `math.fac` uit `import math` (of je eigen module).
  - Bij top-down schrijf je de functie zelf wanneer nodig:

```
def faculteit(x):
    res = 1 # om resultaat in op te bouwen
    while x > 0:
        res *= x
        x -= 1

    return res
```

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

# Constanten als globale variabelen
peiljaar = 2016

# 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())
```

Files manipuleren

Een functie om niet-lege regels in een bestand `invoer` te tellen:

```
def tel_niet_lege_regels(invoer):
    tel = 0
    prevkar = '\n'
    kar = invoer.read(1)
    while kar:
        if prevkar != '\n' and kar == '\n':
            tel += 1
        prevkar = kar
        kar = invoer.read(1)

    return tel
```

NB het is meestal verstandig invoer, rekenwerk en uitvoer door verschillende functies te laten verrichten.

Files manipuleren (2)

Een functie die file `invoer` naar file `uitvoer` omzet, waarbij alleen cijfers aan het begin van een regel worden gekopieerd:

```
def omzetten(invoer, uitvoer):
    prevkar = '\n'
    kar = invoer.read(1)
    while kar:
        if prevkar == '\n' and \
           '0' <= kar and kar <= '9':
            uitvoer.write(kar)
        prevkar = kar
        kar = invoer.read(1)
```

Bepaling cijfer kan natuurlijk netter! In een aparte functie met Boolean returnwaarde.

Tweede programmeeropgave

- Begin met de simpele dingen: bestanden onveranderd kopieren, regelovergangen tellen, alle cijfers uit het bestand halen ...

- Werk vervolgens aan het decoderen. Begin eerst met "eenvoudige" files en maak het steeds een stapje moeilijker.

- Tenslotte, werk aan het histogram en het Lychrel-gebeuren.

Zie ook de werkcolleges!

Priemgetallen

Een **priemgetal** is een geheel getal > 1 dat alleen door 1 en zichzelf deelbaar is.

```
def priem(getal):
    deler = 2
    wortel = math.sqrt(getal) # uit import math
    geendelers = True
    while (deler <= wortel) and geendelers:
        if getal % deler == 0:  # (*)
            geendelers = False  # (*)
        deler += 1

    return geendelers
```

Bij `(*)` mag ook staan: `geendelers = (getal % deler != 0)`.

Priemgetallen (2)

We kunnen er ook voor kiezen direct te "returnen" zodra we zeker weten dat het niks meer wordt.

```
def priem2(getal):
    deler = 2
    wortel = math.sqrt(getal)

    while deler <= wortel:
        if getal % deler == 0:
            return False
        deler += 1

    return True
```

Er zijn overigens veel snellere programma's voor het bepalen van priemgetallen.

Default arguments

- Het is mogelijk om een "standaardwaarde" op te geven voor elk functieargument (dus voor elk formele argument).
- In dat geval is het niet meer verplicht om het actuele argument op te nemen.

```
def teken_cirkel(x, y, straal, kleur="blauw"):
    # Code om een cirkel te tekenen.
    ...

teken_cirkel(0, 0, 10)
teken_cirkel(0, 0, 10, "rood")
teken_cirkel(0, 0)  # Mag *NIET*!
```

Keyword arguments

- In combinatie met default arguments wordt er veel gebruik gemaakt van keyword arguments (ook in bv. matplotlib).
- Handig in het geval van functies met veel parameters en veel standaardwaarden (denk bv. aan plot functies).
- Niet alle parameters hoeven dan expliciet te worden opgegeven.
- Actuele parameters nemen de vorm `naam=waarde` aan.
- Keyword arguments **moeten** aan het einde van de reeks van actuele parameters worden geplaatst.

Keyword arguments (2)

```
def test(x, y):
    return x + y

test(3, 4)
test(y=4, x=3)
test(y=4)    # FOUT!
test(y=3, 4) # FOUT!
```

Keyword arguments (2)

```
def teken_lijn(p1, p2, kleur="zwart", dikte=1.0, pijl=None,
stippel=False):
    # hier wordt de lijn getekend
    pass

teken_lijn( (10, 10), (100, 10), stippel=True, dikte=2.0)
```

Lijsten

- We zagen al strings: rijtjes van karakters.
- We kunnen ook een "rijtje" van variabelen opslaan onder 1 variabelenaam.
- Dit kunnen we doen met lijsten, een lijst is een geordende lijst van variabelen.
  - De variabelen in een lijst hoeven niet van hetzelfde type te zijn.
  - Lijsten kunnen worden aangepast.
  - Er wordt vaak over lijsten gesproken als compound data type of sequence type.

Lijsten (2)

Lijsten kunnen worden aangemaakt met blokhaken:

```
a = [1, 2, 3, 4, 5]
b = [1.0, 2.5, 3.4]
c = [1, "test", 4.5, False]
```

Lijsten (3)

Indexing en slicing gaat precies zoals we hebben gezien bij strings:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7]
>>> len(a)
8
>>> a[6]
6
>>> a[2:5]
[2, 3, 4]
>>> a[3:]
[3, 4, 5, 6, 7]
>>> a[:6]
[0, 1, 2, 3, 4, 5]
>>> a[4] = “ha!”
>>> a
[0, 1, 2, 3, “ha!”, 5, 6, 7]
```

Lijsten (4)

Merk op dat het resultaat van `range( )` ook een lijst is (!):

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
>>> range(3,6)
[3, 4, 5]
```

We kunnen for-loops verder generaliseren, we bezoeken dan elk element van een lijst:

```
for ding in ['telefoon', 'pen', 'papier']:
    print ding
```

Tot slot

Werkcollege:

- Oefenen op papier.
- Werk aan de programmeeropgave. De deadline is vrijdag 21 oktober 2016.