

Funcities

Sommige stukken code willen we op meerdere plaatsen hergebruiken zonder de code telkens te moeten kopiëren. Net zoals we de ingebouwde functies "print" en "input" kunnen gebruiken.

In Python kunnen we functies *definiëren* met het keyword "def". Wanneer we een functie definiëren wordt die niet uitgevoerd. Dat gebeurt pas bij de *aanroep* (*call*) van de functie.

In volgend voorbeeld definiëren we een eenvoudige functie:

```
In [1]: def printToren():  
        print("  =")  
        print(" ===")  
        print("=====")
```

We overlopen even de verschillende elementen:

- *def* duidt aan dat we een functie gaan definiëren;
- *printToren* is de naam van de functie;
- de lege haakjes *()* duiden aan dat de functie geen parameters heeft; hier gaan we later dieper op in;
- vervolgens springen 3 regels in. Dit is de code die zal worden uitgevoerd wanneer de functie aangeroepen wordt.

Zoals we eerder al aangaven, doet de definitie van een functie zelf niets. Maar vanaf dat de definitie wordt uitgevoerd, zal Python deze functie kennen en kunnen gebruiken. Je kan dit vergelijken met een toekenning aan een variabele; op zich "doet" *x=5* niet veel, behalve dat vanaf dat statement variabele *x* gebruikt kan worden en waarde 5 heeft; voor *printToren* geldt hetzelfde; vanaf de definitie kent Python deze functie, en heeft die als "waarde" de lijnen code die uitgevoerd moeten worden.

Uitvoeren van onze functie doen we nu als volgt:

```
In [2]: printToren()  
  
        =  
        ===  
        =====
```

Willen we twee torens printen? Dan moeten we de functie uiteraard niet opnieuw definiëren, maar roepen we de functie gewoon twee maal aan:

```
In [3]: printToren()  
printToren()
```

```
=  
===  
=====  
=  
===  
=====
```

In een programmaatje staan alle regels in 1 bestand:

```
In [4]: def printToren():  
        print("  =")  
        print(" ===")  
        print("=====")  
  
        print("Eerste toren:")  
        printToren()  
        print("Tweede toren:")  
        printToren()
```

```
Eerste toren:  
=  
===  
=====  
Tweede toren:  
=  
===  
=====
```

Omdat Python jouw programma van boven naar onder leest en ondertussen de regels uitvoert moet de definitie van een functie voor de aanroep gebeuren. Is dat niet het geval, dan krijg je een foutmelding:

```
In [5]: A()  
  
def A():  
    print("Dit is A")
```

```
-----  
NameError                                Traceback (most recent call last)  
<ipython-input-5-002ba01ad178> in <module>  
----> 1 A()  
      2  
      3 def A():  
      4     print("Dit is A")  
  
NameError: name 'A' is not defined
```

Een functie is eigenlijk een klein programmaatje waarin we alle constructies die we reeds zagen, kunnen gebruiken. Volgend voorbeeld illustreert dit door verschillende functies te definiëren die samen een tekening maken.

Voorbeeld: stad tekenen

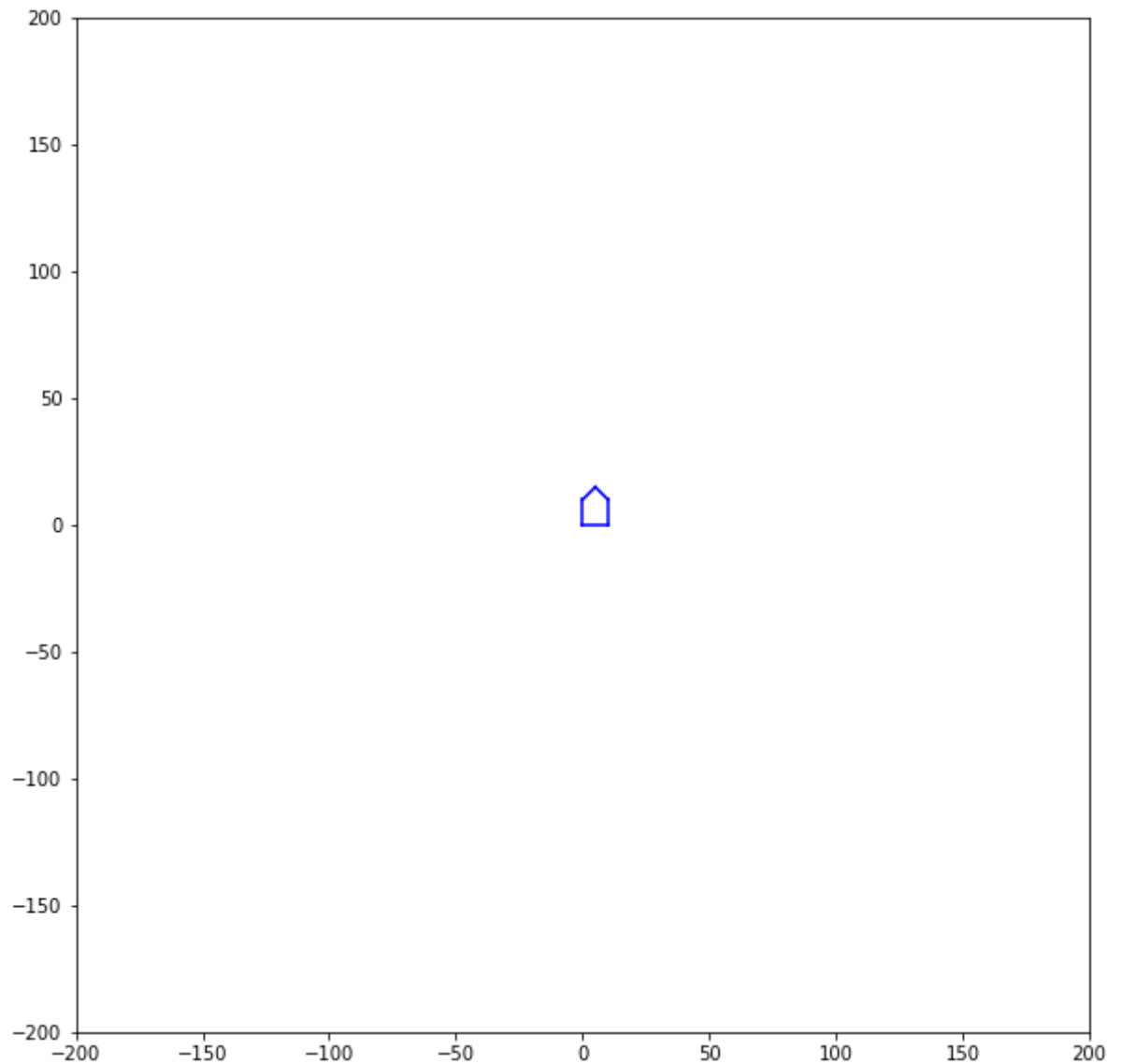
In dit voorbeeld tekenen we een stad bestaande uit verschillende rijen huizen.

```
In [6]: from turtle import *  
import math
```

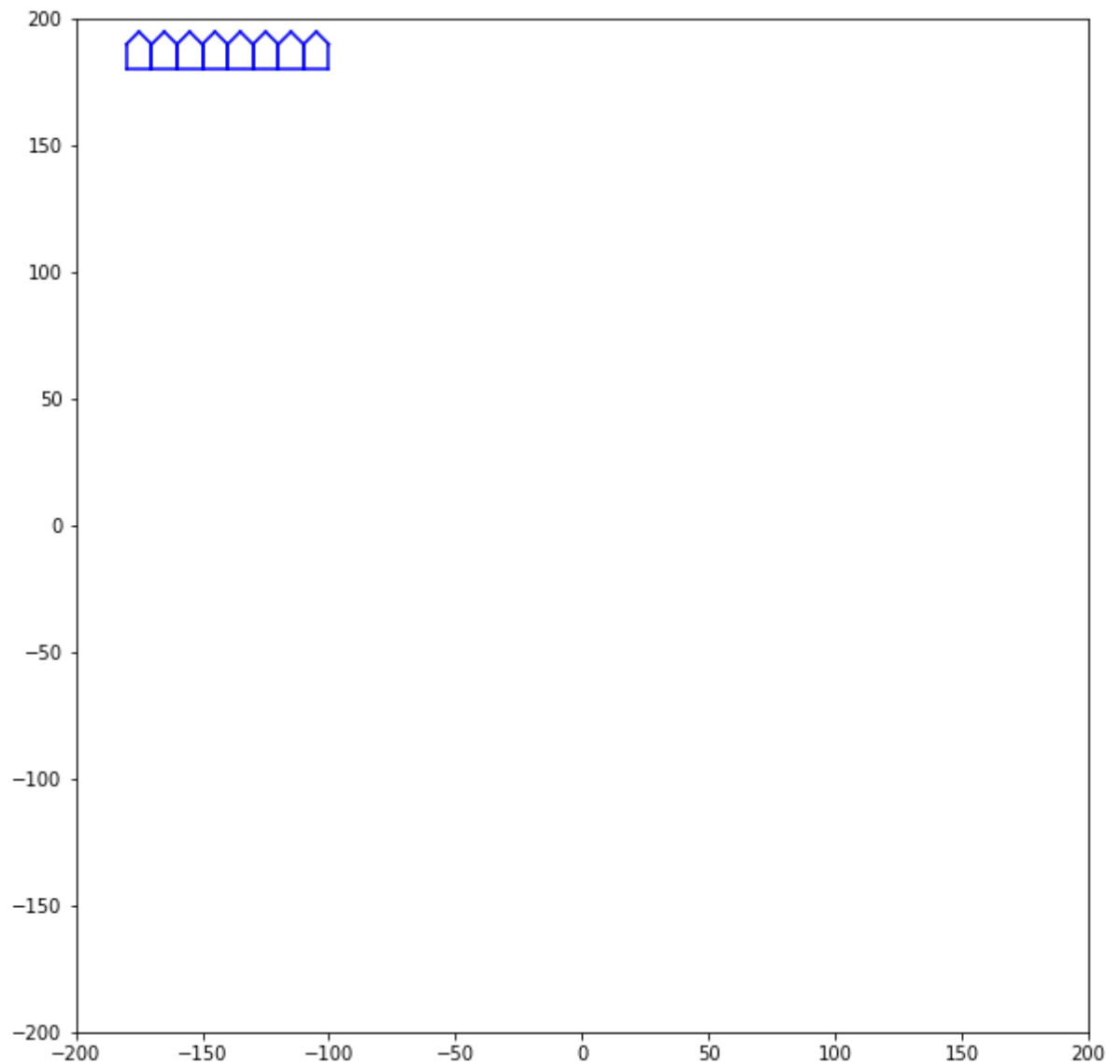
```
reset()
```

```
def huisje():  
    pendown()  
    forward(10)  
    left(90)  
    forward(10)  
    left(45)  
    forward(math.sqrt(50))  
    left(90)  
    forward(math.sqrt(50))  
    left(45)  
    forward(10)  
    penup()  
    left(90)
```

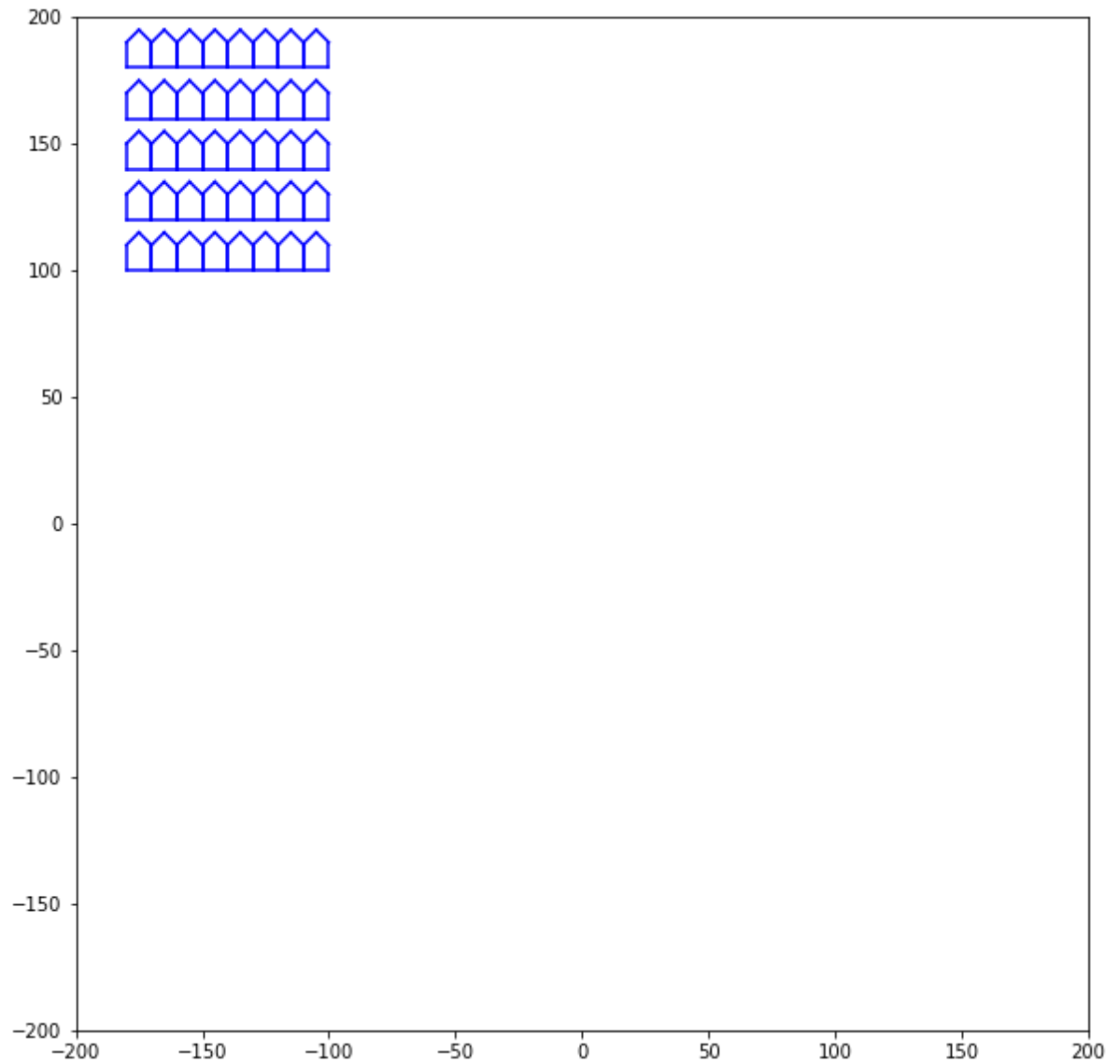
```
huisje()
```



```
In [7]: def street():  
        for nr in range(8):  
            huisje()  
            forward(10)  
        right(180)  
        forward(80)  
        right(180)  
  
reset()  
penup()  
goto(-180,180)  
street()
```



```
In [8]: reset()  
penup()  
goto(-180,180)  
for st in range(5):  
    street()  
    right(90)  
    forward(20)  
    left(90)
```



Ok, toegegeven; "stad" was misschien wat overdreven ...

Funcities met Input

Funcities zijn eigenlijk kleine programmaatjes en kunnen dus ook input krijgen. Deze input wordt gegeven door *parameters* toe te voegen aan de definitie van de functie, die vervolgens als variabelen kunnen gebruikt worden.

Volgende code print een toren af met hoogte h . Merk op dat we de operator `"*"` gebruiken om een string te herhalen; bijvoorbeeld: `"#"*5` is een string van 5 hekjes en `"bla"*3` is `"bla bla bla"`.

```
In [9]: h=5
        b=1 # Breedte van de huidige laag
        indent=h-1 # hoeveel we inspringen in deze laag
        for i in range(h):
            print(" " * indent + "=" * b)
            indent=indent-1 # de volgende laag printen we 1 verder naar links
            b=b+2 # en die wordt ook 2 "=" breder

        =
        ===
        =====
        =====
        =====
```

We kunnen dit nu opnieuw in een functie stoppen en van h een input parameter maken door h op te nemen binnen de haakjes bij de functiedefinitie:

```
In [10]: def printToren(h): # h is een input-parameter
        b=1 # Breedte van de huidige laag
        indent=h-1 # hoeveel we inspringen in deze laag
        for i in range(h):
            print(" " * indent + "=" * b)
            indent=indent-1 # de volgende laag printen we 1 verder naar links
            b=b+2 # en die wordt ook 2 "=" breder

        printToren(3) # printToren wordt uitgevoerd met h gelijk aan 3
        printToren(5) # en met h gelijk aan 5

        a=2
        printToren(2*a-1) # we kunnen hier eender welke expressie meegeven

        =
        ===
        =====
        =
        ===
        =====
        =====
        =====
        =
        ===
        =====
```

We kunnen ook meerdere parameters meegeven, zoals bijvoorbeeld met welk symbool we de toren maken:

```
In [11]: # Print een toren
# symb is het symbool waaruit de toren bestaat
# h is de hoogte van de toren
def printToren(symb,h):
    b=1
    indent=h-1
    for i in range(h):
        print(" " * indent + symb * b)
        indent=indent-1
        b=b+2

printToren("a",3)
printToren("-",6)

s="."
printToren(s,5)
```

```
a
aaa
aaaaa
-
---
-----
-----
-----
-----
-----
.
...
.....
.....
.....
```

Functies met uitvoer

We kunnen functies ook een uitvoer laten geven. In tegenstelling tot de invoer kunnen we bij de uitvoer slechts 1 resultaat terug geven met *return*. Volgende functie geeft een tekenreeks *s*, *n* maal herhaald terug; m.a.w., *s*n* :


```
In [12]: # Deze functie geeft s, n maal herhaald terug.
# Bijvoorbeeld: herhaal("ab",3) is "ababab"
def herhaal(s,n):
    resultaat=""      # in de variabele resultaat gaan we het resultaat opbou
    wen
    for i in range(n):
        resultaat=resultaat + s    # voeg n keer s toe aan resultaat
    return resultaat    # geef de inhoud van de variabele resultaat terug

print(herhaal("NA ",14))
print("Batman!")
```

```
NA NA NA NA NA NA NA NA NA NA NA NA NA NA
Batman!
```

Met het resultaat van een functie kunnen we "rekenen" zoals met een waarde zoals 5, 'abba', of True. We kunnen er echter geen waarde aan toekennen; het is geen variabele:

```
In [13]: s=herhaal("NA ",14)+"Batman!"
print(s)

herhaal("NA ",14) = "Batman"    # Dit geeft een fout

File "<ipython-input-13-04e61c86f878>", line 4
    herhaal("NA ",14) = "Batman"    # Dit geeft een fout
    ^
SyntaxError: cannot assign to function call
```

Let wel op: als een functie een waarde teruggeeft, moet die via alle paden die de uitvoering kan volgen een resultaat teruggeven. Neem bijvoorbeeld volgende functie:

```
In [14]: def grootste(a,b):
    if a>b:
        return a
    if b>a:
        return b

print(grootste(10,5))
print(grootste(7+3,8*1.35))
```

```
10
10.8
```

Voor de twee testgevallen loopt het prima. Maar wat als de twee getallen exact even groot zijn?

```
In [15]: print(grootste(7,3+4))
```

```
None
```

None?! Inderdaad; als de twee getallen exact even groot zijn, zullen zowel de eerste *if* als de tweede geen conditie hebben die *True* is, en dus komt Python bij de uitvoering helemaal geen *return* tegen; vandaar dus *None*; ofwel: Niets, Nada, Nothing, Ne rien, Nichts ...

Wat als we meerdere *return*'s tegenkomen? Dat kan helemaal niet! Vanaf het moment dat we een *return* tegenkomen stopt de functie onmiddellijk, ongeacht welke *for*, *while*, *if*, ... wordt uitgevoerd. Soms kunnen we dit in ons voordeel gebruiken, zoals in volgende functie die test of een gegeven getal priem is:

```
In [16]: def priem(x):
          if x<2:
              return False # 0 en 1 zijn geen priemgetallen
          for d in range(2,x): # Reminder: range(2,x) --> alle getallen van 2 tot en
met x-1
              if x%d==0: # als d een deler is van x; % geeft rest bij deling; rest
0 = d is een deler
                  return False # het is geen priemgetal want d is een deler; de fun
ctie stopt onmiddellijk
          return True # als we uit de for geraken zonder een return tegen te komen i
s het wel een priemgetal

          print(priem(4))
          print(priem(7))
          print(priem(9))
          print(priem(29))
```

```
False
True
False
True
```

Dit is een goed punt om even zelf aan de slag te gaan alvorens het volgende gedeelte te bekijken. Probeer volgende oefeningen te maken:

Oefeningen

1. Maak een functie *right_justify* die een string *s* als input neemt en deze string afdruckt met voldoende spaties ervoor zodat de laatste letter van *s* in kolom 70 terecht komt.

Hint: gebruik string concatenatie (+) en herhaling (*). Merk op dat de functie *len* de lengte van een string geeft; bijvoorbeeld, *len("abba")* is gelijk aan 4.

```
In [17]: # Hier komt jouw oplossing
```

1. Schrijf een functie die volgend rooster afdruckt:

```
+ ---- + ---- + ||||| + ---- + ---- + ||||| + ---- + ---- +
```

Hint: om meer dan 1 string op 1 lijn af te drukken, kan je gebruik maken van *end*:

```
In [18]: print("abc",end=" ")
         print("d")
```

abc d

Met *end="..."* kan je aangeven hoe print wordt afgesloten; standaard is *end* gelijk aan `"\n"`; dat is: het *newline* karakter. Als je helemaal niets tussen twee prints wil afdrukken, gebruik dan *end=""* (lege string dus)

```
In [19]: # Hier komt jouw code
```

1. Voeg nu een parameter toe aan de functie die het aantal cellen in de breedte en de hoogte geeft.

```
In [20]: # Hier komt jouw code
```

Bereik (Scope) van een variabele

Met het bereik of de *scope* van een variabele bedoelen we de plaatsen in het programma waar deze variabele gekend is en gebruikt kan worden. Alle variabelen in een functie, inclusief de parameter variabelen, zijn *lokaal*, wat wil zeggen dat we van buiten de functie deze variabelen niet kunnen zien, aanpassen, of bewerken. Het geheel aan variabelen en functies in het hoofdprogramma noemen we de *global frame*. De variabelen binnen functies maken deel uit van een zogenaamde *local frame* dat daar boven op wordt geplaatst.

Het is belangrijk dat je wat volgt *begrijpt*; je hoeft de scoping regels van Python niet van buiten te leren; zelfs professionele Python programmeurs kennen niet alle details. In plaats daarvan leren we je verderop een aantal gouden stijlregels aan die problemen vermijden.

Volgend voorbeeld illustreert het principe van bereik van een variabele:


```
In [22]: def pasAAan(x):
          A=x
          print("A in pasAAan",A)
          printA()

          def printA():
              print("A in printA", A)

          A=7
          printA()
          pasAAan(3)
          printA()
          print(A)

A in printA 7
A in pasAAan 3
A in printA 7
A in printA 7
7
```

Dit kan vreemd overkomen, maar is eigenlijk erg nuttig omdat het ervoor zorgt dat we bij het schrijven van de code van een functie geen rekening moeten houden met de variabelen uit het hoofdprogramma. We kunnen dezelfde variabele namen gebruiken zonder schrik te moeten hebben dat we ongewenste neveneffecten veroorzaken omdat dezelfde naam ook in het hoofdprogramma gebruikt wordt.

Gouden stelregels

1. Definieer functies steeds bovenaan in jouw programma, of in een aparte *module* (je leert later wat een module is)
2. Communicatie tussen het hoofdprogramma en de functie gebeurt enkel via de parameters van de functie en *return*

Een voorbeeld van foutief gebruik:

```
In [23]: x=4

          def priem():    # Functie staat niet bovenaan
              for i in range(2,x):    # x wordt niet als parameter doorgegeven
                  if x%i==0:
                      return False
              return True

          print(priem())
          x=5
          print(priem())

False
True
```

Hoe het wel moet:

```
In [24]: def priem(x):    # Correct: Functie staat bovenaan
          for i in range(2,x):    # Correct: x werd als parameter doorgegeven
              if x%i==0:
                  return False
          return True    # Correct: de uitkomst wordt via return gegeven

x=5    # deze variabele heeft toevallig dezelfde naam als de parameter in de functie priem
        # dat mag. Dit is een andere variabele dan die in de functie.
print(priem(x))

y=7 # dit werkt bijvoorbeeld ook
print(priem(y))

print(priem(11)) # en dit ook
print(priem(4)) # en dit ook
```

```
True
True
True
False
```

In []: