

Recursieve functies ¶

We zagen in de vorige notebook dat functies die andere functies aanroepen hun variabelen niet delen, maar elk een eigen frame met lokale variabelen hebben. Ook wanneer een functie zichzelf aanroept, worden er aparte frames aangemaakt, voor elke aanroep 1. En ook hier geldt dat een aangeroepen functie geen toegang heeft tot de variabelen van de aanroepende functie, zelfs niet als een functie door zichzelf wordt aangeroepen. Als f zichzelf aanroept, dan bevat de call stack twee local frames voor f , voor elke instantie een.

We illustreren dit best met een voorbeeld; we gaan een functie maken die volgend probleem oplost: *Bij het oplopen van een trap kan je telkens 1 of 2 treden tegelijk nemen; op hoeveel manieren kan je een trap met n treden oplopen?* Bijvoorbeeld, als we een trap met 5 treden hebben, kunnen we die oplopen als volgt: 1-1-1-1-1, 1-1-1-2, 1-1-2-1, 1-2-1-1, 2-1-1-1, 1-2-2, 2-1-2, 2-2-1; op 8 manieren dus. Deze 8 manieren kunnen we opsplitsen in twee groepen:

- Beginnend met een kleine stap: 1-1-1-1-1, 1-1-1-2, 1-1-2-1, 1-2-1-1, 1-2-2
- Beginnend met een grote stap: 2-1-1-1, 2-1-2, 2-2-1

Merk op dat de manieren waarbij we met een kleine stap begonnen, telkens beginnen met 1- en vervolgens met alle manieren om de overige 4 treden op te lopen. Het aantal manieren om een trap van hoogte 5 op te lopen en waarbij we met 1 trede beginnen is dus exact gelijk aan het aantal manieren om een trap van hoogte 4 op te lopen. Idem voor het beginnen met een grote stap: die beginnen allen met 2- gevolgd door een manier om een trap met 3 treden op te lopen. Het aantal manieren om een trap met 5 treden op te lopen is dus exact gelijk aan het aantal manieren om een trap van hoogte 4 op te lopen plus het aantal manieren om een trap van hoogte 3 op te lopen. Als we $M(n)$ gebruiken om het aantal manieren om een trap van hoogte n op te lopen, hebben we dus volgende wiskundige, recursieve formule:

$$M(0) = 1, M(1) = 1, \forall n > 1 : M(n) = M(n-1) + M(n-2)$$

Dit principe zullen we als volgt in code gieten:

- Een functie `manieren(n)` die berekent op hoeveel manieren we een trap met n treden kunnen oplopen
- Behandel de basisgevallen $n==0$ en $n==1$: geef 1 terug.
- Anders: geef `manieren(n-1) + manieren(n-2)` terug.

```
In [29]: def manieren(n):  
          if n<2:  
              return 1  
          else:  
              return manieren(n-1)+manieren(n-2)  
  
          print(manieren(5))
```

Er zijn dus tegelijk oproepen `manieren(5)`, `manieren(4)`, ..., `manieren(1)` actief, elk met een eigen frame, met een eigen versie van de parameter (input-variabele) `n`. Dit wordt ook mooi geïllustreerd in [Python Tutor](http://www.pythontutor.com/visualize.html#code=def%20manieren%28n%29%3A%0A%20%20%20if%20n%3E1%29%2Bmanieren%28n-2%29%0A%20%20%20print%28manieren%285%29%29&cumulative=false&curlInstr=14&heapPrimitives=frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false) (<http://www.pythontutor.com/visualize.html#code=def%20manieren%28n%29%3A%0A%20%20%20if%20n%3E1%29%2Bmanieren%28n-2%29%0A%20%20%20print%28manieren%285%29%29&cumulative=false&curlInstr=14&heapPrimitives=frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false>) :

Frames

Objects

Global frame

manieren

function
manieren(n)

manieren

n | 5

manieren

n | 4

manieren

n | 3

manieren

n | 2

manieren

n | 1

Wanneer we een recursieve functie schrijven is het erg belangrijk om steeds een basisgeval te voorzien en *eerst* te testen of we in een basisgeval zitten alvorens de recursieve aanroepen te doen. Mocht je dit vergeten zijn, of indien jouw recursieve aanroepen een fout maken bij de parameters, dan zal de functie oneindig recursieve aanroepen blijven doen, en telkens een frame aanmaken. Deze frames komen in het geheugen op een "stapel" (*stack*) terecht; als we het basisgeval vergeten, loopt het geheugen voorzien voor deze stapel vol en eindigt het programma. Python heeft standaard een maximale recursie-diepte van 1000, wat wil zeggen dat indien er meer dan 1000 frames op de *stack* staan, Python de uitvoering van het programma stillegt om te vermijden dat de stapel "over loopt". Je krijgt dan volgende foutmelding:

```
In [30]: def manieren(n):  
         return manieren(n-1)+manieren(n-2)  
  
print(manieren(5))
```

```
-----  
RecursionError                                Traceback (most recent call last)  
<ipython-input-30-df263bcf7abe> in <module>  
      2     return manieren(n-1)+manieren(n-2)  
      3  
----> 4 print(manieren(5))  
  
<ipython-input-30-df263bcf7abe> in manieren(n)  
      1 def manieren(n):  
----> 2     return manieren(n-1)+manieren(n-2)  
      3  
      4 print(manieren(5))  
  
... last 1 frames repeated, from the frame below ...  
  
<ipython-input-30-df263bcf7abe> in manieren(n)  
      1 def manieren(n):  
----> 2     return manieren(n-1)+manieren(n-2)  
      3  
      4 print(manieren(5))  
  
RecursionError: maximum recursion depth exceeded
```

Gebruik van recursie

Recursie wordt vaak gebruikt voor problemen die kunnen opgesplitst worden in kleinere deelproblemen van dezelfde soort; bijvoorbeeld, bij het traplopen konden we een moeilijk probleem ("trap met n treden oplopen") opsplitsen in twee net iets makkelijkere problemen ("trap met $n - 1$ treden oplopen" en "trap met $n - 2$ treden oplopen"). Deze strategie kunnen we voor veel problemen toepassen. Dit wordt uitgelegd in de notebook *Divide and Conquer strategieën* die tot uitbreidingsmateriaal behoort. Hier beperken we ons tot enkele klassieke voorbeelden.

Algoritme van Euclides voor grootste gemene deler

In de notebook *Algoritme van Euclides* zagen we een klassiek algoritme om de grootste gemene deler van twee getallen te berekenen:

Probleem: Bepaal GGD van 2 getallen Invoer: 2 getallen Uitvoer: de GGD van deze twee getallen 1. Noem het grootste van de beide getallen A, het andere B. 2. Trek B net zo vaak van A af totdat er 0 overblijft of een getal kleiner dan B ($A \bmod B$). 3. Wanneer er 0 overblijft, is B de ggd. 4. Zo niet, herhaal dan het algoritme met B en wat er van A over is.

Dit algoritme werd vervolgens geïmplementeerd met behulp van een dubbele while loop (we geven hier de implementatie in de vorm van een functie):

```
In [31]: def GGD(A,B):
          while B!=0:
              # stap 2
              while B<=A:
                  A = A - B

              # wissel A en B om
              T=A
              A=B
              B=T
          return A

          print(GGD(16,34))
          print(GGD(16,8))
```

2
8

We zouden er echter ook voor kunnen opteren om stap 4 (zo niet, herhaal het algoritme) te implementeren door *recursief de functie GGD opnieuw aan te roepen*; immers: de oplossing bestaat eruit om *hetzelfde* te doen maar met andere getallen. Dit geeft dan volgende implementatie:

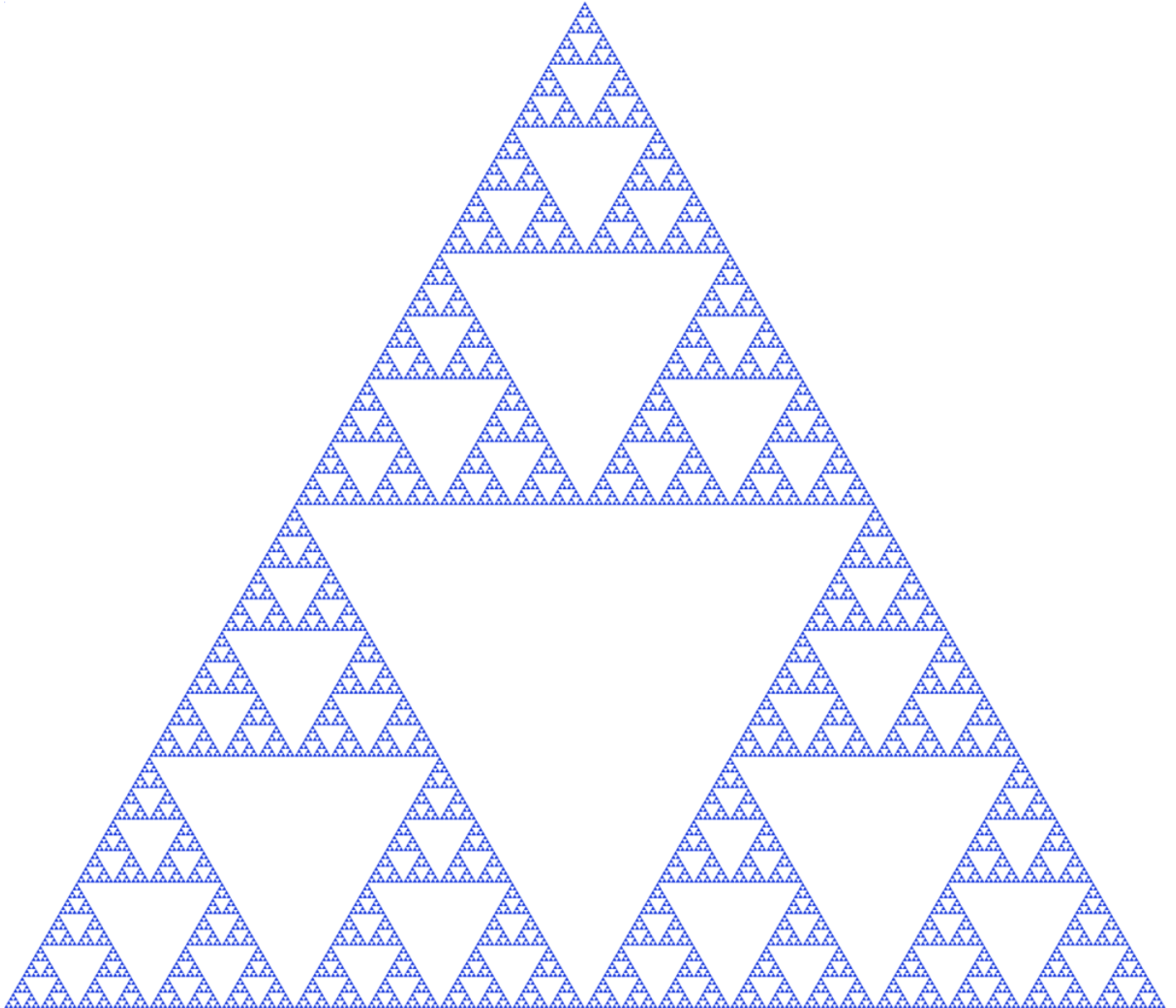
```
In [32]: def GGD(A,B):  
    if B==0:  
        return A  
  
    while B<=A:  
        A = A - B  
  
    return GGD(B,A) # recursieve aanroep van GGD!  
  
print(GGD(16,34))  
print(GGD(16,8))
```

```
2  
8
```

Merk op dat het uiteraard essentieel is om het basisgeval `B==0` in het begin van de functie te behandelen omdat de functie anders oneindig doorgaat (of toch tot Python er genoeg van heeft omdat de maximale recursiediepte bereikt werd).

Fractalen

Een andere klassieker is het maken van fractalen. Fractalen zijn figuren die zichzelf herhalen en worden gevormd via een recursieve definitie. Beschouw bijvoorbeeld volgende *Sierpinski fractaal* (https://nl.wikipedia.org/wiki/Driehoek_van_Sierpi%C5%84ski):



Als we deze figuur in detail bekijken, zien we dat de Sierpinski driehoek bestaat uit drie gestapelde Sierpinski driehoeken van halve grootte; twee aan de basis en één bovenaan. Dit geeft dus volgend algoritme:

Algoritme om Sierpinski driehoek te tekenen met breedte b en linker benedenhoek op (x,y) : Teken een Sierpinski driehoek met breedte $b/2$ en linker benedenhoek op (x,y) Teken een Sierpinski driehoek met breedte $b/2$ en linker benedenhoek op $(x+b/2,y)$ Teken een Sierpinski driehoek met breedte $b/2$ en linker benedenhoek op $(x+b/4,y+\sqrt{3}*b/4)$

(Met de stelling van Pythagoras kan je makkelijk nagaan dat de hoogte van een gelijkzijdige driehoek met zijde

$$\frac{b}{2}, \frac{\sqrt{3}}{4}b \text{ is.})$$

Wanneer we dit programmeren kunnen we niet oneindig doorgaan, dus stoppen we als de Sierpinski driehoek te klein wordt; dan tekenen we gewoon een driehoek. Als "te klein" nemen we bijvoorbeeld $b \leq 5$. Dit geeft de volgende code:


```
In [44]: from turtle import *
from math import sqrt

def sierpinski(b,x,y):
    if b<=5:
        penup()
        goto(x,y)
        pendown()
        for i in range(3):
            forward(b)
            left(120)
    else:
        h=sqrt(3)*b/4
        sierpinski(b/2,x,y)
        sierpinski(b/2,x+b/2,y)
        sierpinski(b/2,x+b/4,y+h)

reset()
sierpinski(300,-150,-150)
```

