

Local en global frame

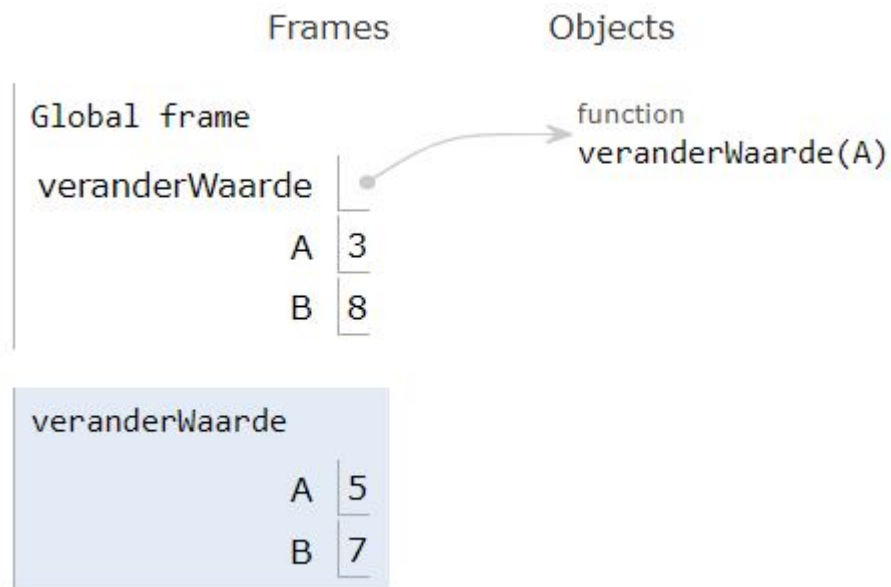
Telkens wanneer een functie wordt aangeroepen, wordt er een nieuwe lokale *frame* gemaakt waarin de variabelen van die functie zitten. De parameters van de functie, zeg maar de input-variabelen, horen daar ook bij. Nadat een functie beëindigd is, worden alle variabelen die tot die functie behoren, verwijderd. Een functie heeft enkel toegang tot haar eigen frame en het global frame, al raden we ten sterkste aan om enkel waarden naar een functie te communiceren via de parameters van die functie.

Volgend voorbeeld laat zien dat variabelen binnen een functie inderdaad lokaal zijn:

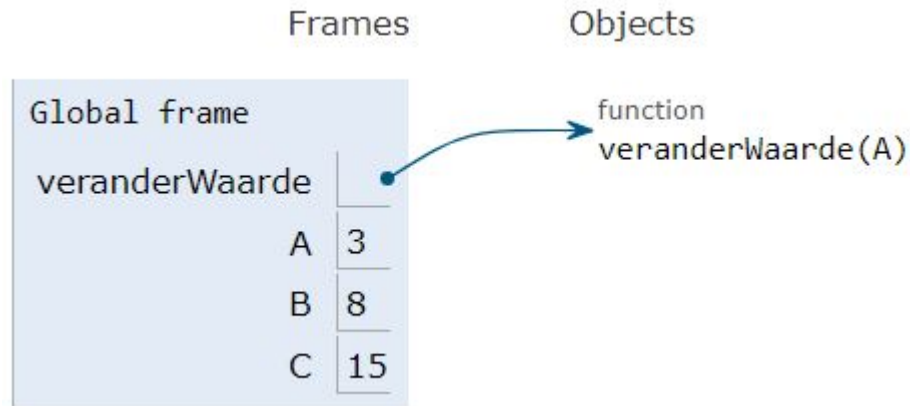
```
In [6]: def veranderWaarde(A):  
        A=5  
        B=7  
        print("(local) A =",A)  
        print("(local) B =",B)  
        return 3*A  
  
A=3  
B=8  
C=veranderWaarde(12+1)  
print("(global) A =",A)  
print("(global) B =",B)  
print("(global) C =",C)  
  
(local) A = 5  
(local) B = 7  
(global) A = 3  
(global) B = 8  
(global) C = 15
```

Bekijk de code hierboven zeker ook eens in [Python tutor](https://pythontutor.com/visualize.html#code=def%20veranderWaarde%28A%29%3A%0A%20%20%20%20A%3Dfrontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false)

(<https://pythontutor.com/visualize.html#code=def%20veranderWaarde%28A%29%3A%0A%20%20%20%20A%3Dfrontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false>). Volgende afbeelding illustreert de inhoud van het geheugen tijdens de uitvoer van `veranderwaarde(12+1)`, net na het uitvoeren van regel 3:



De waarden van `A` en `B` in het hoofdprogramma maken deel uit van het *global frame*, de waarden van `A` en `B` in `veranderWaarde` maken deel uit van het *local frame* van `veranderWaarde`. Eens de functie `veranderWaarde` is afgelopen, verdwijnt ook het frame van de functie. Alle variabelen daar zijn dus weg, enkel de returnwaarde blijft bewaard en werd toegekend aan `C`. Het frame net na de uitvoering van regel 10 (`C=veranderWaarde(12+1)`) ziet er dus uit als volgt:



Ook we nu opnieuw `veranderWaarde` zouden aanroepen, dan wordt opnieuw een local frame voor deze functie gemaakt en na afloop terug verwijderd. Het nieuwe local frame heeft niets met het vorige local frame te maken en neemt ook geen waarden over uit het vorige local frame. Bekijk bijvoorbeeld volgend programma:

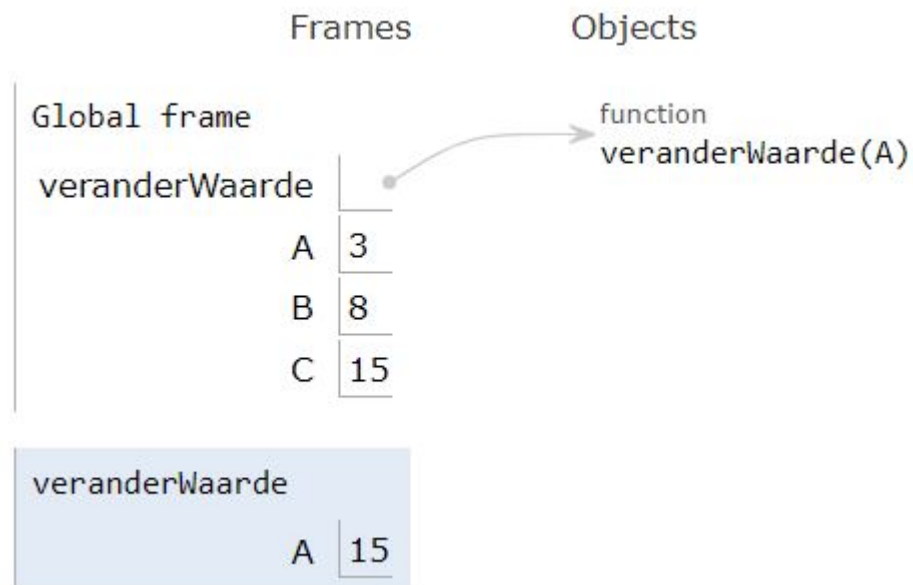
```
In [7]: def veranderWaarde(A):  
        A=5  
        B=7  
        print("(local) A =",A)  
        print("(local) B =",B)  
        return 3*A
```

```
A=3  
B=8  
C=veranderWaarde(12+1)  
print("(global) A =",A)  
print("(global) B =",B)  
print("(global) C =",C)  
veranderWaarde(C)
```

```
(local) A = 5  
(local) B = 7  
(global) A = 3  
(global) B = 8  
(global) C = 15  
(local) A = 5  
(local) B = 7
```

Out[7]: 15

De inhoud van het geheugen wanneer we op lijn 1 zitten van de aanroep op lijn 14 van `veranderWaarde` is als volgt:



Functies die andere functies aanroepen

Ook wanneer functies andere functies aanroepen, wordt er steeds een nieuwe frame aangemaakt, een per functie die actief is. Merk op dat een functie maar 1 andere functie tegelijk kan oproepen (we beschouwen *multi-threading* niet in deze cursus). Op elk moment hebben we dus een lijst van functies $f_1, f_2, \dots, f_k$ waarbij f_1 in het hoofdprogramma werd opgeroepen, f_2 in f_1 , ..., f_k in f_{k+1} en waarbij f_k de functie is die op dat moment actief is. Dit noemen we de *call stack*.

Waarom noemen we dit een *stack*? Omdat het zich exact zoals een stack gedraagt. Als f_k een functie f_{k+1} aanroept, wordt die op het einde toegevoegd (*push*) en krijgen we $f_1, f_2, \dots, f_k, f_{k+1}$. Als f_{k+1} afgelopen is, verdwijnt het laatste element (*pop*) en krijgen we opnieuw $f_1, f_2, \dots, f_k$. Roept f_k nu een andere functie g aan (*push*), dan wordt de call stack $f_1, f_2, \dots, f_k, g$.

Op elk moment in het programma zal naast het global frame, ook voor elke functie in de call stack een local frame bestaan. *Het is essentieel om te begrijpen dat geen enkele functie toegang heeft tot het local frame van eender welke andere functie op de call stack.* Een functie heeft enkel toegang tot haar eigen local frame en in beperkte mate tot de global frame, al raden we ten zeerste af om in een functie gebruik te maken van de toegang tot de global frame. Communicatie tussen aanroepende en aangeroepen functie kan dus enkel en uitsluitend via de parameters (input van aanroepende naar aangeroepen functie) en return (output van aangeroepen naar de aanroepende functie).

We illustreren dit principe in het volgende stukje code dat gebruikt kan worden om het [vermoeden van Goldbach](https://nl.wikipedia.org/wiki/Vermoeden_van_Goldbach) (https://nl.wikipedia.org/wiki/Vermoeden_van_Goldbach) (elk geheel getal groter dan 2 kan geschreven worden als de som van 2 niet noodzakelijk verschillende priemgetallen) te testen:

```
In [3]: def deler(d,x):
        if d>x:
            return False
        while x>=d:
            x-=d
        return x==0

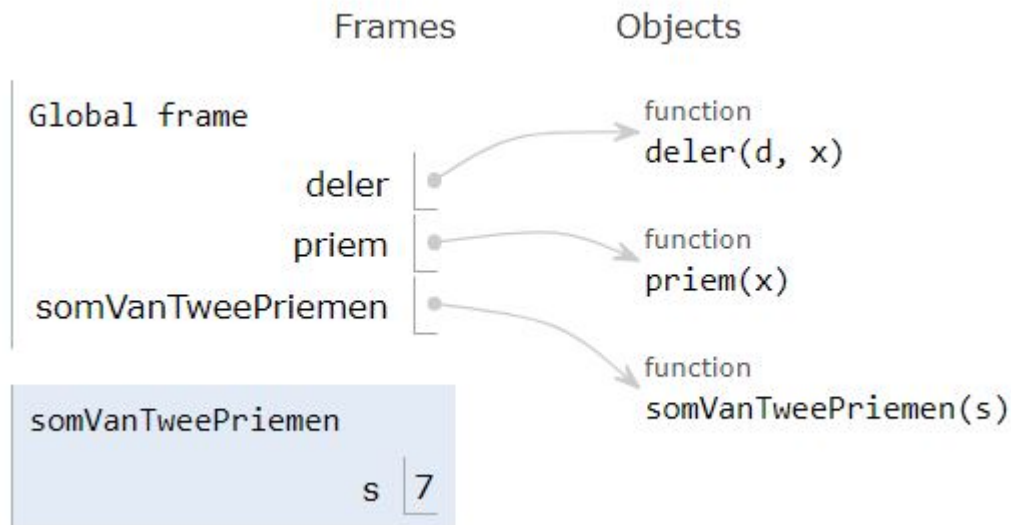
        def priem(x):
            for d in range(2,x):
                if deler(d,x):
                    return False
            return True

        def somVanTweePriemen(s):
            for x in range(2,s):
                if priem(x) and priem(s-x):
                    return (x,s-x)
            return None

        print(somVanTweePriemen(7))
        print(somVanTweePriemen(10))
```

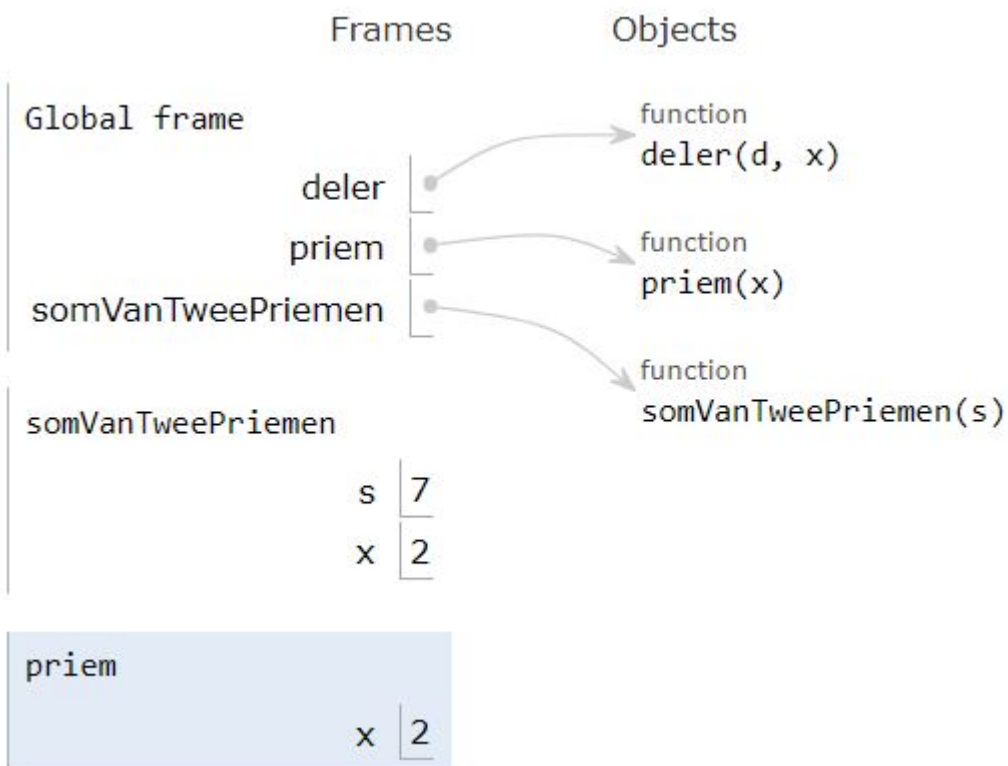
```
(2, 5)
(3, 7)
```

Bekijk het stukje code ook in [Python Tutor\(\)](#). Bij de start van de eerste aanroep van `somVanTweePriemen(7)` ziet het geheugen er als volgt uit:

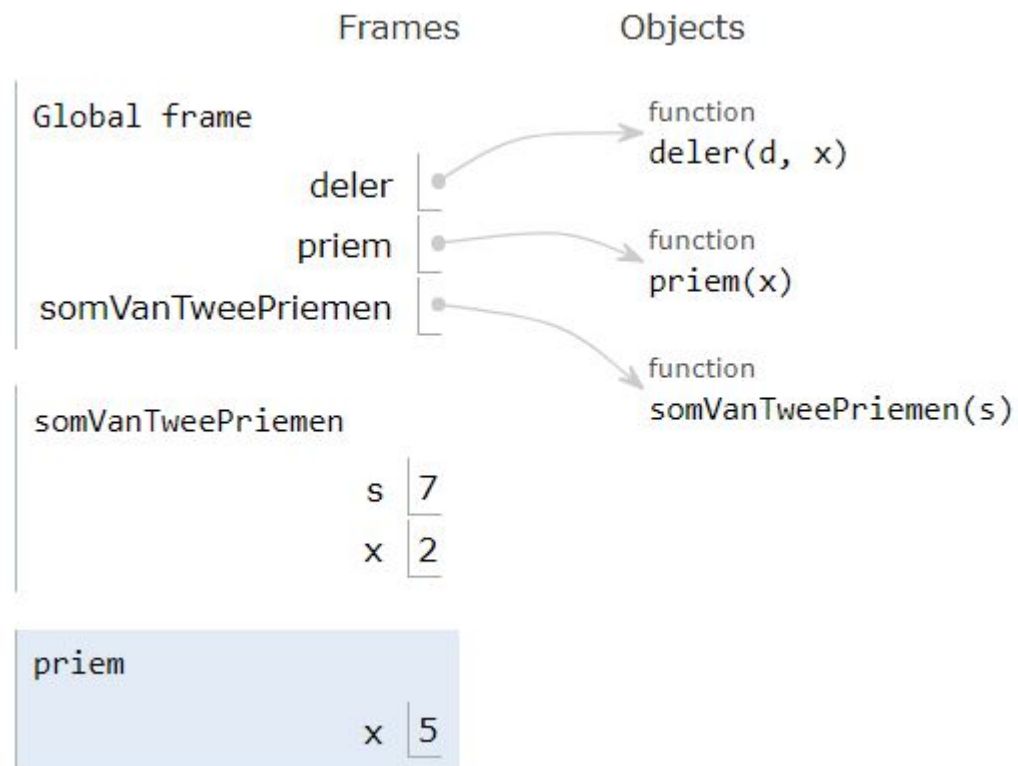


We hebben twee frames: de global frame en de frame van de aanroep `somVanTweePriemen(7)`. Binnen deze functie krijgt `x` waarde 2 en moet de uitdrukking `priem(2)` and `priem(5)` geëvalueerd worden. Hiervoor zal eerst `Priem(2)` uitgevoerd worden.

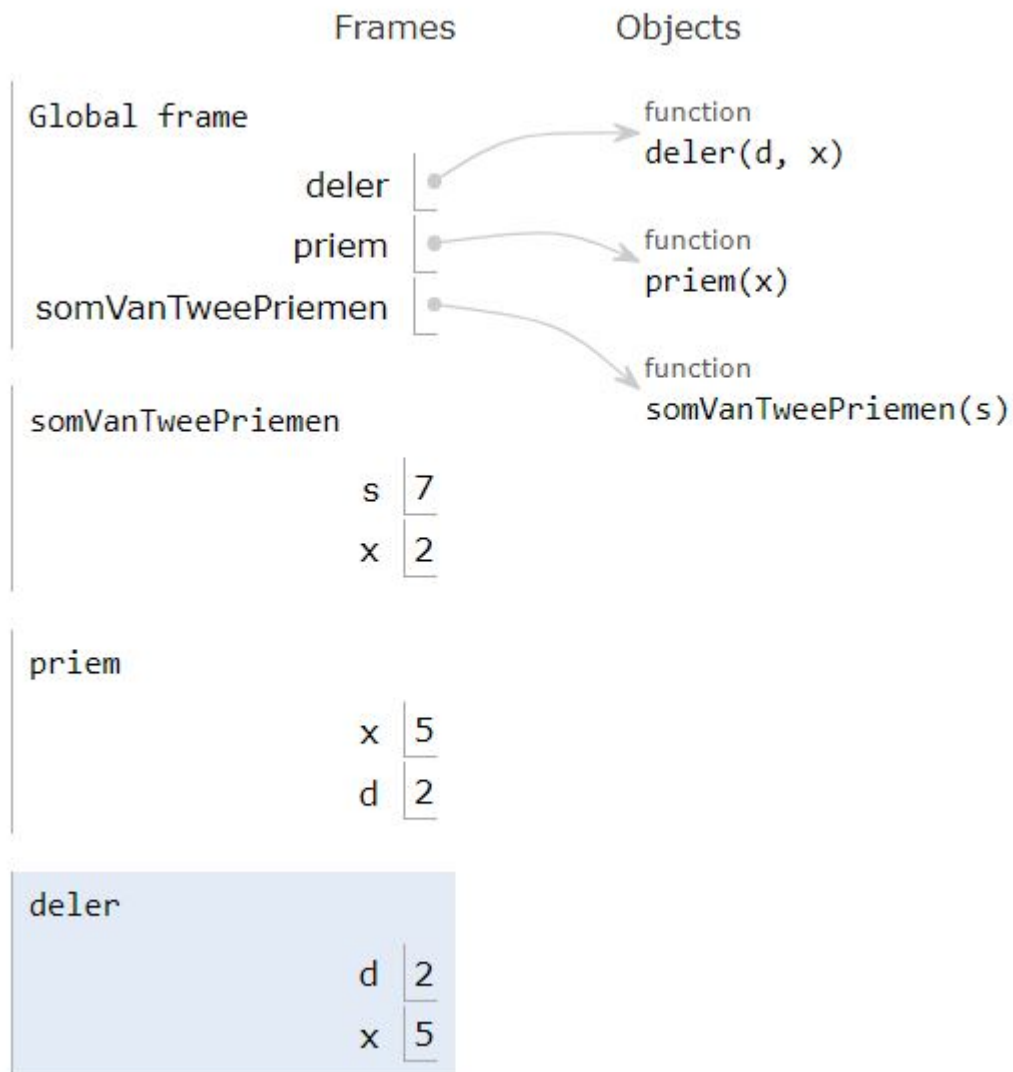
Bij het begin van de uitvoer van `priem(2)` zijn er dus 3 frames; het frame van `priem` is erbij gekomen. Merk op dat de loop-variabele `x` aan het frame van `somVanTweePriemen` werd toegevoegd.



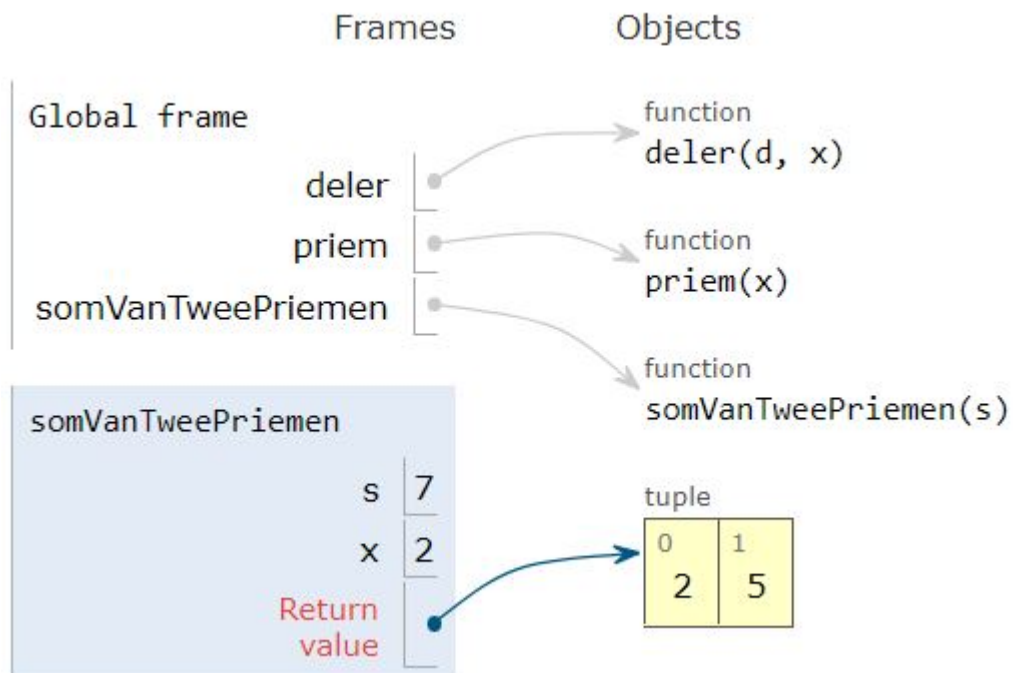
Priem(2) stopt bijna onmiddellijk en geeft True . De local frame van priem verdwijnt uit het geheugen. Daarna wordt de tweede term van de conjunctie priem(2) and priem(5) geëvalueerd en wordt er dus een nieuwe local frame, dit maal voor priem(5) aangemaakt:



Nu gaan we de for-loop wel in, en wordt `deler(2,5)` aangeroepen. Er wordt dus opnieuw een local frame toegevoegd, dit maal voor `deler` :



En zo gaat dit nog een tijdje door en groeit en krimpt de call stack tot we op het einde het paar (2,5) als output geven. De call stack net voordat de call `priem(5)` eindigt, ziet er bijvoorbeeld als volgt uit:



Daarna herhaalt de hele procedure zich voor `somVanTweePriemen(10)` , met bijvoorbeeld volgende call stack als tussentijds resultaat:

