

4.2 in-Functie

Als we willen kijken of een element in een lijst zit kunnen we alle elementen doorlopen met een index. Deze handeling is zoveel voorkomend dat hier een aparte methode voor gemaakt is. Door middel van {in} kan je kijken of iets “in” een lijst zit, het resultaat is een Boolean.

```
1 lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15]
2 print("13 in lijst : ", 13 in lst)
3 print(" 6 in lijst : ",  6 in lst)
```

Shell ×

```
>>> %Run Tuples.py
13 in lijst : False
 6 in lijst : True
```

4.3 for-loop

Wat ook vaak voorkomt is dat je alle elementen uit een lijst wil nalopen. Een voor een wil je de elementen gebruiken om bijvoorbeeld op het scherm te printen.

```
1 lst = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 15]
2 for x in lst:
3     print(x, end="~")
```

Shell ×

```
>>> %Run Tuples.py
1~2~3~4~5~6~7~8~9~10~11~12~15~
```

Dit hoeven we niet alleen met getallen te doen, we kunnen dit met alles doen wat zich in een lijst bevindt zoals een lijst van namen, of een lijst van lijsten.

```
1 for x in ["Djim", "Sem", "Roland", "Nigel", "Tobias", "Leon"]:
2     print(x, end="~")
```

Shell ×

```
>>> %Run test.py
Djim~Sem~Roland~Nigel~Tobias~Leon~
```

In het voorbeeld hierboven is de lijst een niet opeenvolgende lijst, we missen 13 en 14. Het komt vaak voor dat we een hele reeks getallen willen doorlopen die wel op volgende zijn. Als we bijvoorbeeld 1000 van die getallen eerst in een lijst moeten zetten is niet zo handig. Hiervoor is de functie “range” voor gemaakt.

```
range(start, stop, step)
```

Parameter Values

Parameter	Description
<i>start</i>	Optional. An integer number specifying at which position to start. Default is 0
<i>stop</i>	Required. An integer number specifying at which position to end.
<i>step</i>	Optional. An integer number specifying the incrementation. Default is 1

Als we dus een berekening willen maken met alle even getallen tussen 50 en 100 kunnen we dat op de volgende manier opschrijven:

```
1 for x in range(50, 100, 2):  
2     print(x, end="~")
```

Shell ×

```
>>> %Run Tuples.py
```

```
50~52~54~56~58~60~62~64~66~68~70~72~74~76~78~80~82~84~86~88~90~92~94~96~98~
```

Let op dat de {stop} waarde niet wordt meegenomen, dit is het moment dat de lijst stopt.

De “for x in range(10)” constructie wordt ook heel veel gebruikt als je iets 10 keer wil laten gebeuren. Dit is een kortere schrijfmethode dan met een while.

4.4 Random.choice

Met random kunnen we een keuze maken van een element uit een lijst. We maken een lijst van even getallen tussen [50, 100>, en daar halen we een willekeurig getal uit. Zo kunnen we ook van een lijst van namen een willekeurige naam selecteren.

```
1 import random  
2 x = random.choice( range(50, 100, 2) )  
3 print(x)
```

Shell ×

```
>>> %Run Tuples.py
```

```
84
```

```

1 import random
2 lijst = ["Arjan", "Martin", "Gerrit", "Marjon", "Marcel", "Hans"]
3
4 print(random.choice(lijst))

```

Shell ×

```
>>> %Run -c $EDITOR_CONTENT
```

Martin

4.5 Veel voorkomende fouten met list

Bij het gebruiken van lijsten en tuples zouden we het volgende kunnen doen. Maar waarom gaat dit verkeerd?

```

1 print( ["A", "B", "C"].append("D"))
2 print( ("A", "B", "C").append("D"))

```

Shell ×

```
>>> %Run test.py
```

```

None
Traceback (most recent call last):
  File "\\ssc.local\dvc_redir_mdw\kmb\Desktop\test.py", line 2, in <module>
    print( ("A", "B", "C").append("D"))
AttributeError: 'tuple' object has no attribute 'append'

```

Figuur 10: Fouten bij append list en tuples

Regel 2 gaat fout omdat dit een tuple is, en een tuple heeft geen append methode. Dit kan je ook in de foutmelding lezen.

Regel 1 is wat lastiger om te begrijpen. Append doe je op een lijst, maar het resultaat van de functie is None. Als je een lijst wilt uitbreiden met een element moet je eerst de lijst aan een variabele koppelen. Deze lijst kan je dan uitbreiden met append, en vervolgens kan je de lijst printen. Het verschil is dus wat is het resultaat van een functie.

```

1 lst = ["A", "B", "C"]
2 print( lst)
3 lst.append("D")
4 print( lst)

```

Shell ×

```
>>> %Run test.py
```

```

['A', 'B', 'C']
['A', 'B', 'C', 'D']

```

Figuur 11: Correct toevoegen van een element aan list

Het verwijderen van elementen uit een lijst als deze lijst in een for wordt gebruikt is ook een erg gevaarlijke actie. Dit geeft code die als je er een paar keer naar kijkt te begrijpen is, maar die erg verwarrend kan zijn.

```
1 lst = ["Djim", "Sem", "Roland", "Nigel", "Tobias", "Leon"]
2 for x in lst:
3     print(x, end="~")
4     lst.remove(x)
```

Shell ×

```
>>> %Run test.py
Djim~Roland~Tobias~
```

Het verstandigste als je in een lijst dezelfde lijst wil aanpassen, is om eerst een copy van de lijst te maken. Maar let op dit kan ook snel fout gaan.

```
1 lst = ["Djim", "Sem", "Roland", "Nigel", "Tobias", "Leon"]
2 lst2 = lst
3 for x in lst2:
4     print(x, end="~")
5     lst.remove(x)
```

Shell ×

```
>>> %Run test.py
Djim~Roland~Tobias~
```

Er wordt namelijk geen copy van de lijst gemaakt, maar een nieuwe variabele die naar dezelfde lijst staat te wijzen. Wat je dan in lst veranderd, veranderd ook in lst2, want het is hetzelfde in het geheugen. Je moet dus echt de copy() methode gebruiken om een copy te maken.

```
1 lst = ["Djim", "Sem", "Roland", "Nigel", "Tobias", "Leon"]
2 lst2 = lst.copy() # of lst[:]
3 for x in lst2:
4     print(x, end="~")
5     lst.remove(x)
```

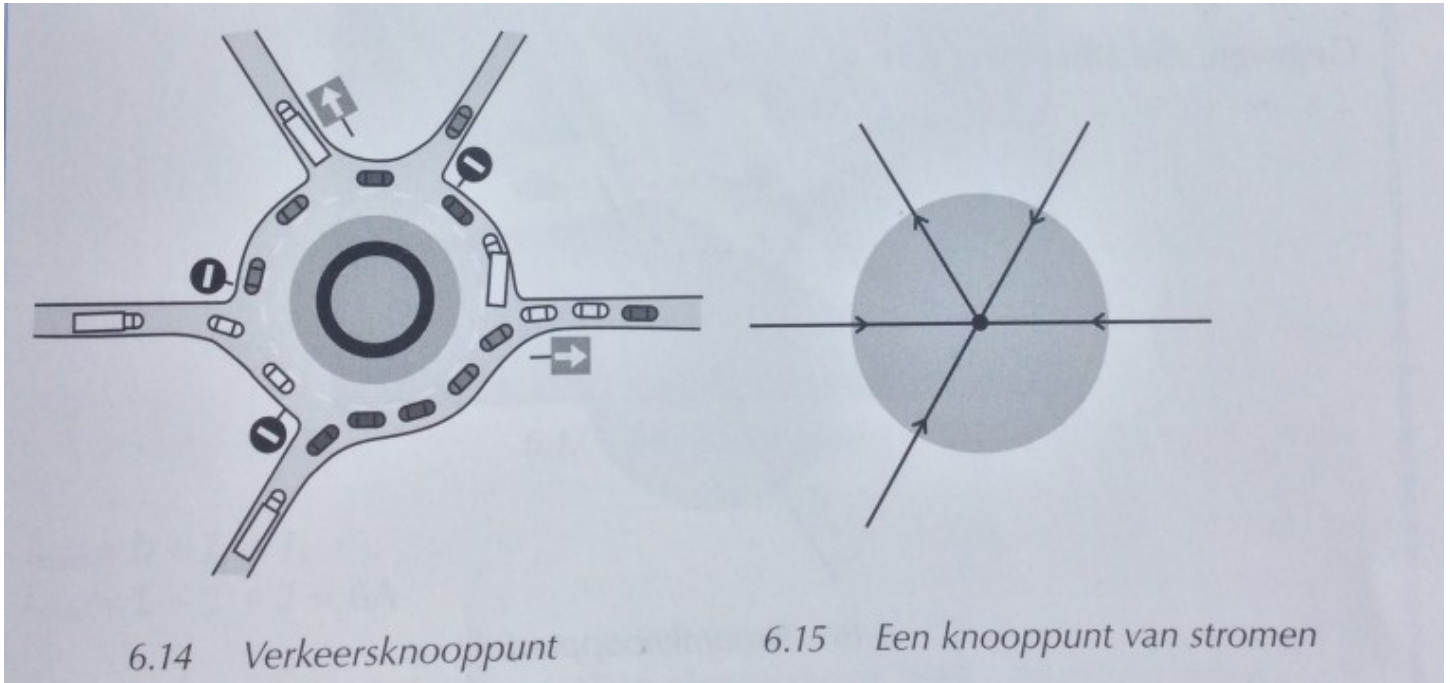
Shell ×

```
>>> %Run test.py
Djim~Sem~Roland~Nigel~Tobias~Leon~
```

Figuur 12: Verwijderen elementen uit list

Lijsten zijn dus heel krachtig om mee te werken, maar je moet goed weten waar je mee bezig bent. Het is dan ook altijd aan te raden om eerst even een kleine test te doen naar de werking voordat je een groot stuk code hebt gemaakt die opeens iets onverwachts doet.

4.6 Eerste wet van Kichhoff



1 wet van Kirchhoff:

de som van de stromen in een knooppunt is gelijk aan 0.

$$\sum I = 0$$

Wat betekent dit op een verkeersplein. Als je heel veel wegen op een knooppunt hebt, en van iedere weg komen er auto's naar het kruispunt moeten er andere wegen zijn waar de auto's weer wegrijden.

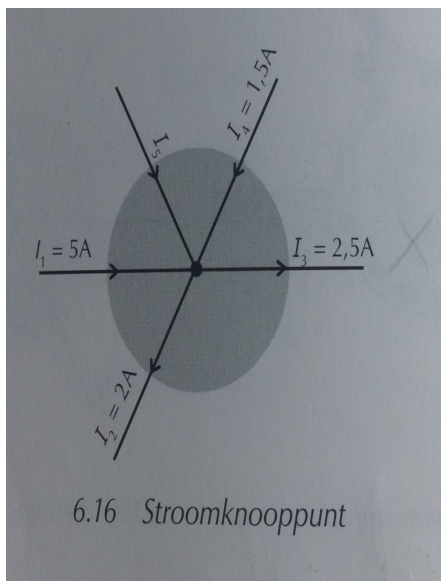
Als je van alle wegen op een na weet hoeveel auto's aan komen rijden, of wegrijden, dan kan je van de onbekende weg berekenen hoeveel auto's daar wegrijden of aankomen.

We gaan een functie maken die de stroom van auto's (of elektrische stroom, dat werkt hetzelfde) als input binnenkrijgt. Als we op enter drukken zijn we klaar met de invoer. Alle waarden zijn in een lijst gestopt.

Daarna gaan we die lijst aan een andere functie aanbieden die van de onbekende weg het aantal auto's (of stroom) berekend.

Om dit te doen spreken we het volgende af.

- We voeren alleen getallen in, of een enter om het af te sluiten.
- Auto's die naar het kruispunt rijden zijn positieve getallen
- Auto's die van het kruispunt afrijden zijn negatieve getallen.



$$I_1 = +5$$

$$I_2 = -2$$

$$I_3 = -2,5$$

$$I_4 = +1,5$$

I_5 kan dan berekend worden

De invoer ziet er als volgt uit.

```

1 def invoer():
2     lijst = []
3     while True:
4         i = input("Geef waarde +is naar knooppunt, -is vanaf knooppunt: ")
5         if i == "":
6             return lijst
7         lijst.append(int(i))
8
9 print(invoer())

```

Shell x

```
>>> %Run -c $EDITOR_CONTENT
```

```

Geef waarde +is naar knooppunt, -is vanaf knooppunt: 1
Geef waarde +is naar knooppunt, -is vanaf knooppunt: 2
Geef waarde +is naar knooppunt, -is vanaf knooppunt: -2
Geef waarde +is naar knooppunt, -is vanaf knooppunt:
[1, 2, -2]

```

Het resultaat van de invoer sturen we naar de berekenKirchhoff functie. Hieronder zie je de eerste aanzet voor het programma. We hebben alleen nog niet de onbrekende stroom berekend, we geven nu alleen nog maar 0 terug.

```

1 def invoer():
2     lijst = []
3     while True:
4         i = input("Geef waarde +is naar knooppunt, -is vanaf knooppunt: ")
5         if i == "":
6             return lijst
7         lijst.append(int(i))
8
9 def berekenKirchhoff(lijst):
10     ontbrekende = 0
11
12     return ontbrekende
13
14 lijst = invoer()
15 ontbrekende = berekenKirchhoff(lijst)
16 print(lijst, ontbrekende)
17

```

Shell

```
>>> %Run -c $EDITOR_CONTENT
```

```

Geef waarde +is naar knooppunt, -is vanaf knooppunt: 1
Geef waarde +is naar knooppunt, -is vanaf knooppunt: 2
Geef waarde +is naar knooppunt, -is vanaf knooppunt: 3
Geef waarde +is naar knooppunt, -is vanaf knooppunt:
[1, 2, 3] 0

```

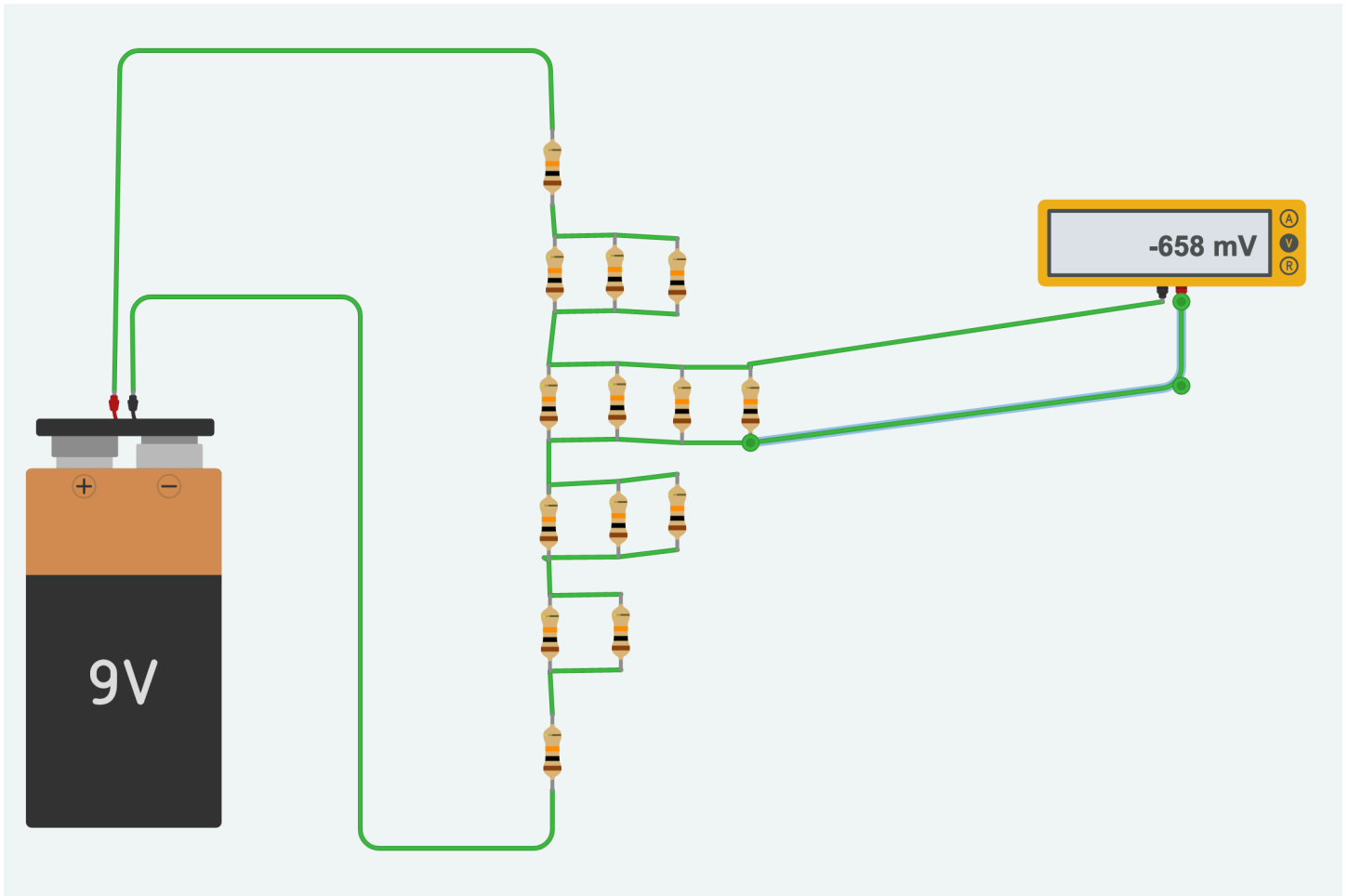
Om te testen gaan we niet iedere keer waarden invullen, we maken even een lijst aan die we aan de berekenKirchhoff functie geven.

Dit is ook de reden dat we twee functie maken. We kunnen de twee functies dan ook apart van elkaar gebruiken.

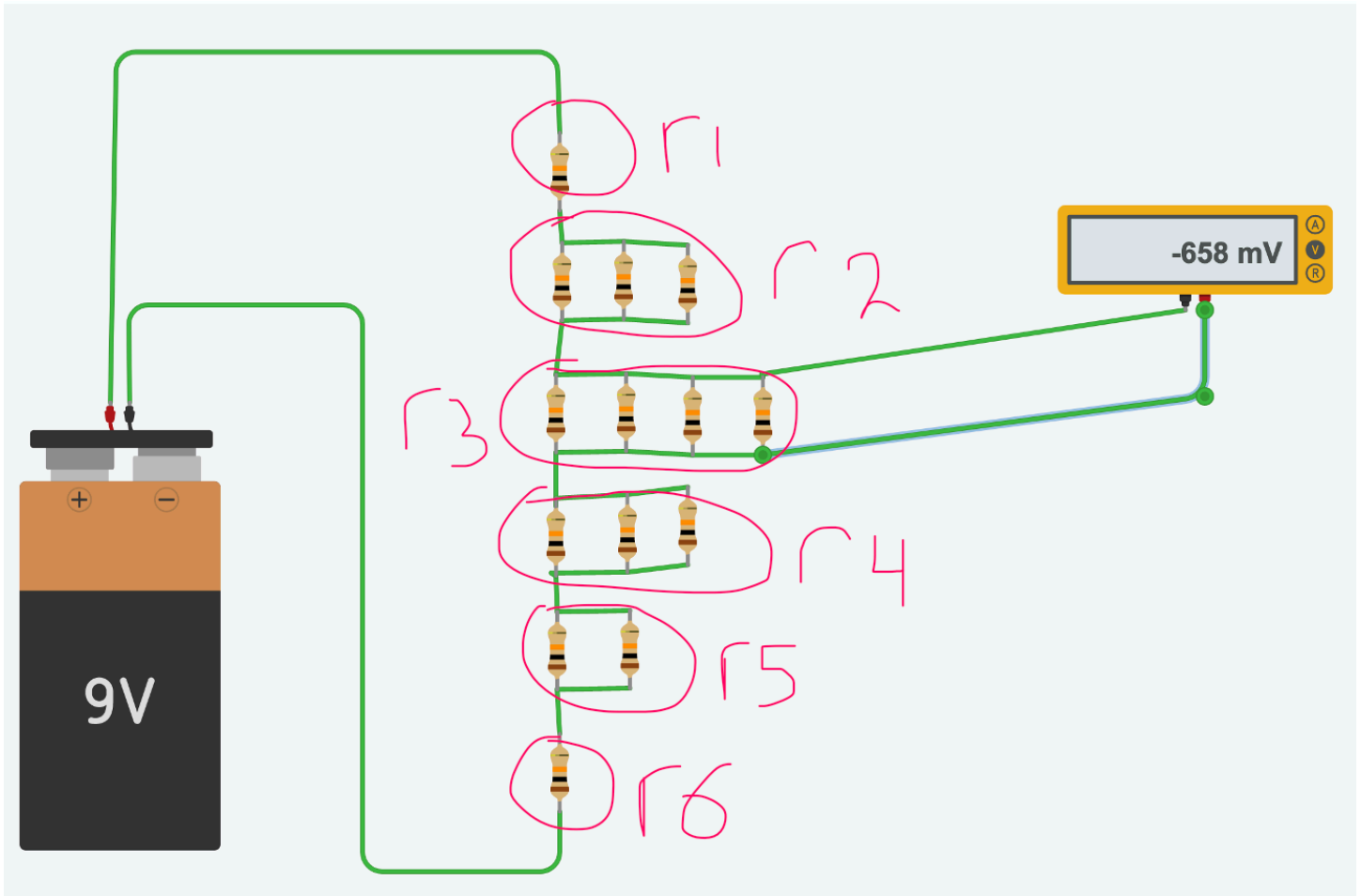
4.7 Slimme dingen doen met lijsten en tuples

We hebben bij het behandelen van functies het onderstaande probleem gezien.

Dit probleem kunnen we oplossen door steeds de bereken serie en bereken parallel aan te roepen. Dit kunnen we veel slimmer doen.



Als je naar de tekening hieronder kijkt kan je zien dat we eigenlijk een paar weerstanden in serie hebben. We zouden dus een lijst kunnen maken met weerstanden.



We gaan eerst met een makkelijker voorbeeld verder. Voorheen gebruikte we de onderstaande functie. Als we 3 weerstanden hadden, berekende we eerst de vervangingsweerstand van de eerste 2 weerstanden. Het resultaat daarvan gebruikte we samen met de andere weerstand om de nieuwe waarde te berekenen.

```
1 def berekenSerie(weerstand1, weerstand2):  
2     return weerstand1 + weerstand2  
3  
4  
5 totaal1 = berekenSerie(30, 20)  
6 totaal2 = berekenSerie(totaal1, 20)  
7
```

In de nieuwe opgave zien 6 vervangingsweerstand. Stel dat de weerstanden allemaal 300 Ohm zijn, dan kunnen we de volgende tuple maken (we gebruiken een tuple omdat we toch niets gaan toevoegen of verwijderen).

```
1 weerstanden = (300, 300, 300, 300, 300, 300)
```

We willen dus die weerstanden meegeven aan een functie die de vervangingsweerstand berekend. We gaan dus alle weerstanden in de lijst een voor een langs en die zetten we in serie.

```
1 WEERSTANDEN = (300, 300, 300, 300, 300, 300)
2
3 def berekenSerie(weerstandenLijst):
4     rVervanging = 0
5     for r in weerstandenLijst:
6         rVervanging += r
7     return rVervanging
8
9 print("Vervangingsweerstand = ", berekenSerie(WEERSTANDEN), "Ohm")
10
```

```
Shell x
>>> %Run -c $EDITOR_CONTENT
```

```
Vervangingsweerstand = 1800 Ohm
```

We kunnen in de lijst dus allemaal weerstanden zetten, van iedere waarde, en we krijgen de vervangingsweerstand terug.

Ditselfde kunnen we doen voor de parallel weerstand. Doordat de formule voor de parallel weerstand wat moeilijker is gebruiken we de oude methode met 2 weerstanden.

```
9 def berekenParallel(r1, r2):
10     if r1 == 0 and r2 == 0:
11         return 0
12     return (r1 * r2) / (r1 + r2)
13
14 print(berekenParallel(100, 100))
15
16
```

```
Shell x
>>> %Run -c $EDITOR_CONTENT
```

```
50.0
```

```

7 WEERSTANDEN_PARALLEL = (300,300,300)
8 def berekenParallel2(r1, r2):
9     if r1 == 0 and r2 == 0:
10         return 0
11     return (r1 * r2) / (r1 + r2)
12
13 def berekenParallel(weerstandenLijst):
14     if len(weerstandenLijst) == 0:
15         return 0
16     rVervanging = weerstandenLijst[0]
17     for r in weerstandenLijst[1:]: # [:1] betekend dat we bij element 1 beginnen ipv 0
18         rVervanging = berekenParallel2(rVervanging, r)
19     return rVervanging
20 print(berekenParallel(WEERSTANDEN_PARALLEL))
21
22

```

```
>>> %Run -c $EDITOR_CONTENT
```

```
100.0
```

We kunnen deze twee dingen combineren. Dit kunnen we doen op verschillende manieren, ik ga hieronder de simpelste manier gebruiken. We gaan een functie maken die een lijst opschoont.

We gaan een lijst maken van weerstanden, die staan allemaal in serie. Maar in die lijst zitten ook lijsten, dat zijn weerstanden die parallel aan elkaar staan.

```

20 WEERSTANDEN = ( 300
21                 ,(300, 300, 300)
22                 ,(300, 300, 300, 300)
23                 ,(300, 300, 300)|
24                 ,(300, 300)
25                 , 300
26                 )
27 def opschoonen(weerstanden):
28     newList = []
29     for r in weerstanden:
30         if type(r) is tuple:
31             r = berekenParallel(r)
32             newList.append(r)
33     return newList
34 print(opschoonen(WEERSTANDEN))
35
36

```

```
>>> %Run -c $EDITOR_CONTENT
```

```
[300, 150.0, 150.0, 150.0, 150.0, 300]
```

We hebben nu een lijst met weerstanden in serie staan. Dus het resultaat kunnen we aan de berekenSerie functie geven om het eindresultaat te berekenen.

```
19
20 WEERSTANDEN = ( 300
21                 ,(300, 300, 300)
22                 ,(300, 300, 300, 300)
23                 ,(300, 300, 300)
24                 ,(300, 300)
25                 , 300
26                 )
27 def opschonen(weerstanden):
28     newList = []
29     for r in weerstanden:
30         if type(r) is tuple:
31             r = berekenParallel(r)
32             newList.append(r)
33     return newList
34
35 serieLijst = opschonen(WEERSTANDEN)
36 print("Vervangingsweerstand = ", berekenSerie(serieLijst), "Ohm")
37
```

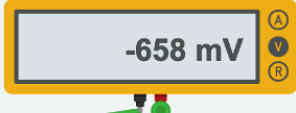
```
Shell
>>> %Run -c $EDITOR_CONTENT
1025.0
```

De opdracht is eigenlijk nog niet klaar.

We moeten met deze waarde de stroom door de schakeling berekenen.

```
35 serieLijst = opschonen(WEERSTANDEN)
36 vervangingsweerstand = berekenSerie(serieLijst)
37 uBatterij = 9
38 stroom = uBatterij / vervangingsweerstand
39 spanning_meter = stroom * berekenParallel((300, 300, 300, 300))
40 print(spanning_meter)
41
42
```

```
Shell
>>> %Run -c $EDITOR_CONTENT
0.6585365853658537
>>>
```



Dit is misschien de eerste keer heel veel werk om te maken, maar vanaf nu kan je iedere opstelling die er zo uitziet snel doorrekenen.

4.8 Opdrachten

4.8.1 Vectoren

Maak in Python een functie die twee paramaters binnenkrijgt. Beide parameters zijn vectoren. De twee vectoren kunnen we optellen tot een nieuwe vector. Iedere vector is een tuple, waarbij het eerste getal de x-waarde voorstelt, en het tweede getal de y-waarde. De vector (tuple) die we teruggeven kunnen we op de onderstaande methode maken. Maak natuurlijk ook testen voor deze functie.

```
1 x = 1
2 y = 2
3 result = tuple((x,y))
4 print(result)
```

4.8.2 Afstand coördinaten

Maak in Python een functie die twee paramaters binnenkrijgt. Beide parameters zijn coördinaten op een kaart. Iedere coördinaat is een tuple met twee getallen, het eerste getal is de x-positie en het tweede getal is de y-positie. Bereken met de Stelling van Pythagoras de afstand tussen de twee coördinaten. Deze berekende waarde wordt teruggegeven uit de functie.

Maak natuurlijk ook testen voor deze functie.

4.8.3 Speelkaarten

Maak een functie die een lijst van speelkaarten teruggeeft. We hebben twee sets waarmee we de speelkaarten maken.

- o Set 1: ("Harten", "Ruiten", "Klaveren", "")
- o Set 2: (2, 3, 4, 5, 6, 7, 8, 9, 10, "Boer", "Vrouw", "Heer", "Aas")

De de in-functie kan je alle verschillende combinaties maken. Iedere combinatie is een tuple. Alle combinaties worden in een lijst gestopt. Maak voor deze functie een test die controleert of het aantal speelkaarten klopt.

Deze lijst wordt aan een andere functie gegeven die de kaarten geschud weer teruggeeft. Let op dat de oorspronkelijke stapel kaarten ook behouden moet blijven. Hoe dit moet kan je op w3Schools vinden.

Zoekwoorden:

1. W3Schools Python random shuffle.
2. W3Schools Python copy list.

4.8.4 Nummerslot kraken

Kraak de code van het nummerslot.

Alle 4 de nummers zijn anders!



2	6	5	7	2 correcte cijfers, maar niet op de goede plaats
4	2	6	8	Heeft geen correcte cijfers.
0	4	1	5	1 correcte cijfer, op de verkeerde plaats
1	7	4	9	Heeft 2 correcte cijfers op de goede plaats.

Dit is genoeg informatie om achter de oplossing te komen. Maak de oplossing in een computerprogramma, je mag dus niets zelf beredeneren, je programma moet dan de redeneratie nadoen.

Wil je wat meer hulp dan kan je hieronder verder lezen:

Bovenstaande regels zijn hieronder weergegeven in Tuples.

```
43 combo = ( (2, 6, 5, 7), (0, 4, 1, 5), (4, 2, 6, 8), (1, 7, 4, 9) )
44 correctDigits = (2, 1, 0, 2)
45 correctPlaces = (0, 0, 0, 2)
46 count = 0
47 for tryNumber in range(0, 9999):
48     tupleDigits = getDigits(tryNumber)
49     if allDifferent(tupleDigits):
50         #count += 1
51         if checkPossibleDigits(correctDigits, combo, tupleDigits):
52             #count += 1
53             if checkPossiblePlaces(correctPlaces, combo, tupleDigits):
54                 print(t)
55                 count += 1
56 print(count)
57
```

Je loop alle nummers langs en zet ze om in een tuple van de nummers (regel 48).

Op regel 49 maak je een functie die kijkt of alle digits anders zijn in de tuple.

Verder kijk je of alle digits voldoen aan correctDigits. Dit doe je door een functie te maken die kijkt of het aantal digits overeenkomt met het getal wat je meegeeft.

Als laatste kijk je op regel 53 naar het getal wat je meegeeft en het getal wat waarvan je weet hoeveel er op de correcte plek staan of die overeenkomen.

Dit is best een moeilijke opdracht. Ga functie voor functie aan het werk. Maak eerst getDigits en dan allDifference. Bekijk dan hoeveel mogelijke getallen nog overblijven (5040).

Na CheckPossibleDigits blijven er nog 24 mogelijkheden over, en na CheckPossiblePlaces blijft er nog 1 over.

4.8.5 Rekenpuzzel

Hieronder nog een rekenpuzzel om op te lossen. Welke getallen horen in de hokjes. Voor ieder hokje maak je een variabele. Het eerste hokje noemen we A, tweede B enzovoort.

Er zijn 3 horizontale vergelijkingen, daar kan je 3 functies van maken met 3 parameters. Er zijn ook 3 verticale vergelijkingen. Daarvoor kan je ook weer 3 functies maken.

Als eerste ga je alle oplossingen zoeken voor de horizontale vergelijkingen. Deze oplossingen zet je in drie lijsten. Dan ga je kijken van de verschillende antwoorden welke combinatie van antwoorden ook op de verticale vergelijkingen werken.

Voor iedere horizontale functie laat je de drie variabelen van 0 tot 9 lopen.

Stappenplan:

1. Maak een functie die als resultaat een lijst geeft van alle combinaties van (0,0,0) tot (9,9,9). Het is handig om ook mee te geven dat een variabele een vaste waarde heeft (voor vergelijking 2 is dat de 8).
2. Maak functie die een cijfercombinatie binnenkrijgt en of die vergelijking True is bij invullen $a + b \times c == 20$.
3. Geef van stap 1 alle getallen aan de functie, en maak een lijst van alle combinaties.
4. Herhaal stap 2 en 3 voor regel 2 en 3.
5. Op dit moment heb je drie lijsten met alle oplossingen per regel. We gaan nu kijken welke combinatie van horizontale uitkomsten ook zorgt dat de verticale vergelijkingen uitkomen.

	+		×		= 20
+		×		-	
	×	8	×		= 48
-		×		+	
	×		-		= 38

=	=	=
0	80	10

4.8.6 Mijnenveger

Op <http://minesweeperonline.com/#> kan je het spelletje mijnenveger spelen. Dit was een erg populair spel toen Windows 3.2 voor uitkwam in de jaren 1992. We gaan in deze opdracht een matrix maken (list in list) van het spel. We doen de bommen in het veld, en tellen bij ieder vakje hoeveel bommen er aangrenzend zijn. Bovenaan de code zetten we de constanten neer die we in het spel gebruiken. Het moet zo werken dat als de constanten gewijzigd worden, het hele veld anders opgebouwd wordt.

```
Minesweeper.py x
1  import random
2
3  NUMBERBOMBS = 99
4  WIDTH = 30
5  HEIGHT = 16
6  EMPTY = " "
7  BOMB = "*"
8
```

Het eindresultaat moet eruitzien zoals hieronder weergegeven. Het eerste scherm is optioneel, deze is afgedrukt nadat het veld en de bommen neergezet zijn, maar voordat het aantal aangrenzende bommen is geteld.

Het spel maken we nog niet speelbaar, we berekenen alleen het veld.

Dit zou de aanroep van het hoofdprogramma kunnen zijn.

```
66
67  def game():
68      matrix = createMatrix(WIDTH,HEIGHT)
69      fillBombs(matrix, WIDTH, HEIGHT, NUMBERBOMBS)
70      printMatrix(matrix, WIDTH, HEIGHT)
71      print()
72      countBomb(matrix, WIDTH, HEIGHT)
73      printMatrix(matrix, WIDTH, HEIGHT)
74  |
75
76  if __name__ == "__main__":
77      game()
```



```
>>> %Run Minesweeper.py
```

[illegible]

>>>

