

# Python: 7. Containers

Dr. Cornelis Stal

April 27, 2022

## 1 Containers

Python bevat een aantal mogelijkheden om data te verwerken in containers, namelijk `lists`, `dictionaries`, `sets` en `tuples`.

### 1.1 Lijsten (`lists`)

#### 1.1.1 Lijst aanmaken

Een list is een verzameling (of *collection*) elementen. Een lijst wordt aangemaakt door middel van vierkante haakjes en de individuele waarden worden gescheiden door komma's.

```
[ ]: # Maak een lege lijst aan
toestelLijst = []
```

#### 1.1.2 Indices

In een lijst kunnen we objecten met verschillende datatypes verwerken. De elementen van een list zijn geordend. Omdat ze geordend zijn, kunnen we ieder element van een lijst benaderen via een index, net zoals we de tekens van een string zouden benaderen. De indices beginnen bij 0 en niet bij 1, net als bij strings. De `index`-functie geeft de positie van een bepaald element in een lijst terug. Als het element niet in een lijst staat, geeft Python een `ValueError`-fout:

```
[ ]: # Maak een lijst aan met verschillende datatypes
toestelLijst = ['Wild T3', 'Topcon ES-105', 3.14]
# Geef een element van een lijst op een welbepaalde index
print(toestelLijst[1])
toestelLijst.index('Trimble S8')
```

#### 1.1.3 Lijsten in lijsten

Ook kunnen de elementen binnen een lijst bestaan uit andere lijsten. Dit zijn geneste lijsten:

```
[ ]: # Maak een lijst aan met verschillende datatypes
toestelLijst = [['Wild T3', 'Hoeken'], ['Topcon ES-105', 'Hoeken en
↳afstanden'], 3.14]
# Geef een element van een lijst op een welbepaalde index
print(toestelLijst[0])      # Geeft ['Wild T3', 'Hoeken']
print(toestelLijst[1][0])   # Geeft 'Topcon ES-105'
```

#### 1.1.4 Negatieve indices

Het is mogelijk om binnen een lijst van achteren naar voren te werken. Hiervoor wordt gebruik gemaakt van een negatieve index. Het laatste element van een lijst heeft een index -1, het voorlaatste element -2, ...:

```
[ ]: # Geef een element van een lijst op een welbepaalde index
print(toestelLijst[-1])    # Geeft de waarde 3.14
```

#### 1.1.5 Elementen gebruiken

Elementen kunnen gebruikt worden als onderdeel van een string:

```
[ ]: toestelLijst = [['Wild T3', 'Hoeken'], ['Topcon ES-105', 'Hoeken en
↳afstanden'], 3.14]
print('Een {} meet {}'.format(toestelLijst[0][0], toestelLijst[0][-1]))
```

#### 1.1.6 Waarden aanpassen

Elementen in een lijst kunnen eenvoudig gewijzigd worden door de oude waarde met corresponderende index te overschrijven.

```
[ ]: # Pas de waarde van een element aan
toestelLijst[2] = 'Trimble S8'
```

#### 1.1.7 Elementen verwijderen

Het verwijderen van een element op een bepaalde plaats in een lijst gebeurt met de **del**-functie en doet alle resterende waarden opschuiven in de lijst. Ook kunnen we elementen verwijderen door met de **remove**-functie te verwijzen naar de inhoud van het element:

```
[ ]: # Verwijder een element uit de lijst
del(toestelLijst[2])           # Op basis van een index
toestelLijst.remove('Trimble S8') # Op basis van inhoud
```

#### 1.1.8 Aantal elementen in een lijst

Een andere nuttige methode is **len**, waarmee het aantal elementen van een lijst verkregen kan worden:

```
[ ]: # Aantal elementen in een lijst
len(toestelLijst)
```

#### 1.1.9 Elementen toevoegen

Om elementen toe te voegen aan het einde van een lijst gebruiken we de **append**-functie. Elementen op een welbepaalde positie toevoegen kan met de **insert**-functie:

```
[ ]: # Elementen toevoegen
toestelLijst.append('Trimble S8')
toestelLijst.insert(1, 'Leica BLK360')
```

### 1.1.10 Elementen sorteren

Om elementen in de juiste volgorde (alfabetisch, numeriek) wordt de `sort`-functie gebruikt. Een omgekeerde volgorde wordt verkregen met de `reverse`-functie. Deze functies werken overigens alleen als alle elementen hetzelfde datatype hebben:

```
[ ]: # Elementen sorteren
toestelLijst = ['Wild T3', 'Topcon ES-105', 'Trimble S8', 'Leica BLK360']
# Van klein naar groot of van A naar Z
toestelLijst.sort()
# Van groot naar klein of van Z naar A
toestelLijst.reverse()
```

### 1.1.11 Range

De `range`-functie is een ingebouwde functie om een lijst van integers te creëren. Deze functie is zojuist ook aan bod gekomen bij de bespreking van `for`-lussen:

```
[ ]: # De range functie
list(range(10))      # Een lijst met elementen van 0 tot en met 9
list(range(5,10))    # Een lijst met elementen van 5 tot en met 9
list(range(0,10,2))  # Een lijst met elementen van 0 tot en met 9 met een
                    ↪ increment van 2
```

### 1.1.12 Slicing

In de voorgaande voorbeelden werd getoond hoe we werken met één element in een lijst. Er bestaat echter ook een zeer compactere syntax om te werken met sublijsten, namelijk *slicing*. Enkele voorbeelden:

```
[ ]: nums = list(range(10))
# Geef het deel van de lijst van index 2 tot 4 (bovengrens exclusief)
print(nums[2:4])
# Geef het deel van de lijst van index 2 tot het einde
print(nums[2:])
# Geef het deel van de lijst van het begin tot index 2 (bovengrens exclusief)
print(nums[:2])
# Geef het deel van de lijst van het begin tot het einde
print(nums[:])
# Negatieve index
print(nums[:-1])
# Sublist toevoegen aan een slice van een lijst
nums[2:4] = [8, 9]
```

### 1.1.13 Lussen op lijsten

Lussen zijn nuttig om de elementen van een lijst te overlopen:

```
[ ]: toestelLijst = ['Wild T3', 'Topcon ES-105', 'Trimble S8', 'Leica BLK360']
      # Druk alle elementen van de lijst af
      for toestel in toestelLijst:
          print(toestel)
```

### 1.1.14 List comprehensions

Bij het programmeren worden data dikwijls getransformeerd het ene object naar het andere object, al dan niet voorafgegaan een bepaalde reeks handelingen. Bijvoorbeeld:

```
[ ]: # Handeling op een lijst zonder list comprehension
      nums = [0, 1, 2, 3, 4]
      squares = []
      for x in nums:
          squares.append(x ** 2)
      print(squares)
```

Deze code kan beduidend compacter uitgeschreven worden wanneer gebruik gemaakt wordt van *list comprehension*:

```
[ ]: # Handeling op een lijst met list comprehension
      nums = [0, 1, 2, 3, 4]
      squares = [x ** 2 for x in nums]
      print(squares)
```

“List comprehensions” kunnen ook voorwaarden bevatten:

```
[ ]: # List comprehension met voorwaarden
      nums = [0, 1, 2, 3, 4]
      even_squares = [x ** 2 for x in nums if x % 2 == 0]
      print(even_squares)
```

## 1.2 Dictionaries

Een *dictionary* is een object van het type `dict` en houdt koppels van sleutels en waarden, ofwel `{key, value}`-paren bij. Een *dictionaries* kan vergeleken worden een *hashmap* in Java of een *object literal* in JavaScript.

*Dictionaries* kunnen als volgt gebruikt worden: net zoals bij een lijst kan men elementen toevoegen, wijzigen en verwijderen uit een `dict`-object. Het grote verschil is echter dat elementen niet gekoppeld zijn aan een nummer of index zoals het geval is bij een lijst. Elk element in een *dictionary* heeft twee delen: een sleutel (`key`), en een waarde (`value`). Een bepaalde sleutel opzoeken in een `dict`-object geeft de waarde gekoppeld aan deze sleutel terug.

### 1.2.1 Dictionary aanmaken

*Dictionaries* worden gedeclareerd met accolades en elk element wordt eerst gedeclareerd door zijn sleutel, gevolgd door een dubbelpunt en tenslotte de waarde zelf. Een dict-object wordt dus als volgt aangemaakt:

```
[ ]: # Dictionary aanmaken
instr_dict = {'theodoliet': 'Wild T3', 'totaalstation': 'Topcon ES-105',
              'GNSS': 'Trimble S8', 'laserscanner': 'Leica BLK360'}
```

### 1.2.2 Dictionary bevragen

Vervolgens kan de waarde voor een bepaalde sleutel opgevraagd worden, en kunnen we testen of een bepaalde sleutel al dan niet bestaat.

```
[ ]: # Dictionary bevragen op basis van een sleutel
print(instr_dict['totaalstation'])      # Geeft 'Topcon ES-105'
# Controleren of een sleutel al dan niet bestaat
print('theodoliet' in instr_dict)       # Geeft 'True'
print('GNSS' not in instr_dict)         # Geeft 'False'
print('meetlint' in instr_dict)         # Geeft 'False'
```

Het rechtstreeks aanspreken van een niet-bestaande sleutel zal resulteren in een `KeyError`. In dat geval is het beter om de `get`-functie te gebruiken, voorzien van een te zoeken sleutel en een default-waarde:

```
[ ]: # Voorkom een KeyError door middel van de get-functie met default-waarde
print(instr_dict['meetlint'])
print(instr_dict.get('meetlint', 'N/A'))
print(instr_dict.get('GNSS', 'N/A'))
```

Een dict-object beschikt over een aantal nuttige methodes om alle sleutels (`keys()`), alle waarden (`values()`) en alle koppels (`items()`) te bevragen:

```
[ ]: # Print alle sleutels
print(instr_dict.keys())
# Print alle waarden
print(instr_dict.values())
# Print de gehele dictionary
print(instr_dict.items())
```

In combinatie met een lus kunnen zo alle elementen overlopen worden:

```
[ ]: # Haal alle instrumenten-keys op en druk deze af met de corresponderende waarde
for instrument in instr_dict:
    print('%s --> %s' % (instrument, instr_dict[instrument]))
```

Volledigheidshalve merken we ook op dat we opnieuw de `len`-functie kunnen gebruiken om het aantal elementen in een dict-object op te vragen:

```
[ ]: # Aantal elementen in een dictionary
print(len(instr_dict))
```

### 1.2.3 Elementen toevoegen of verwijderen

Nieuwe elementen kunnen toegevoegd worden aan een bestaand dict-object, en bestaande elementen kunnen worden verwijderd. Wanneer de waarde voor deze verwijderde entiteit gevraagd wordt, zal de default-waarde teruggegeven worden:

```
[ ]: # Entiteit toevoegen aan een dictionary
instr_list['meetlint'] = 'simpel lint'
print(instr_list['meetlint'])
# Entiteit uit dictionary verwijderen
del instr_list['meetlint']
print(d.get('meetlint', 'N/A'))
```

## 1.3 Sets

### 1.3.1 Algemene handelingen op een set

Een set is een ongeordende verzameling van verschillende elementen:

```
[ ]: # Set aanmaken
instr_set = {'theodoliet', 'totaalstation', 'GNSS', 'laserscanner'}
```

Zoals bij een list-object en een dict-object kunnen we bekijken of een element al dan niet aanwezig is in een set:

```
[ ]: # Set bevragen
print('GNSS' in instr_set)
print('meetlint' in instr_set)
```

Ook kunnen nieuwe elementen toegevoegd worden aan een set, en kunnen andere elementen verwijderd worden. Indien een nieuw element al behoort tot de set, zal er niets gebeuren. Unieke elementen kunnen namelijk slechts een keer voorkomen in een set:

```
[ ]: # Element toevoegen aan een set
instr_set.add('meetlint')
print('meetlint' in instr_set)
# Element verwijderen uit een set
instr_set.remove('meetlint')
print('meetlint' in instr_set)
# Bestaand element nog eens toevoegen
instr_set.add('laserscanner')
print('laserscanner' in instr_set)
```

Het aantal elementen van een set kan eveneens met de `len`-functie opgevraagd worden:

```
[ ]: # Geef het aantal elementen in een set
print(len(instr_set))
```

### 1.3.2 Lussen en sets

Men kan met behulp van een lus itereren over de elementen in een set. De syntax hiervoor is dezelfde als bij het itereren over een lijst. Aangezien de orde van de elementen echter niet vastligt, kan geen veronderstellingen gemaakt worden over de volgorde waarin de elementen in de lijst overlopen worden. We gebruiken daarom een speciale iterator, namelijk de `enumerate`-functie:

```
[ ]: # For-lus over een set
instr_set = {'theodoliet', 'totaalstation', 'GNSS', 'laserscanner'}
# De enumerator geeft een volgnummer en het object zelf terug
for idx, instr in enumerate(instr_set):
    print('#{}: {}'.format(idx, instr))
```

### 1.3.3 Set comprehensions

Net zoals bij lijsten, kan een nieuwe set gecreëerd worden met behulp van *set comprehensions*:

```
[ ]: # Set comprehensions
from math import sqrt
nums = {int(sqrt(x)) for x in range(30)}
print(nums)
```

## 1.4 Tuples

Een `tuple`-object is een (onveranderlijke) geordende lijst van waarden. Een `tuple`-object lijkt op veel manieren op een lijst. Een van de belangrijkste verschillen met lijsten is dat `tuples` kunnen gebruikt worden als `keys` in een `dict`-object en als elementen in een set.

Het aanmaken van een enkele `tuple`-object en het creëren van een `dict`-object met `tuple` keys verloopt als volgt:

```
[ ]: # Enkele tuple aanmaken
t = (5, 6)
print(type(t))
# Tuple met keys aanmaken
d = {(x, x + 1): x for x in range(10)}
print(d)
```

Het is nu mogelijk na te gaan waar deze (of een andere) `tuple`-object zich in de `dict`-object bevindt:

```
[ ]: # Onderzoek of een tuple aanwezig is in een dictionary
print(d[t])
print(d[(1, 2)])
```

## 1.5 Oefeningen

### Opdracht: statistische berekeningen op scores

Schrijf een programma dat een aantal berekeningen uitvoert op basis van de punten (op 20) van 10 leerlingen:

```
resultaten = (13.0, 17.0, 5.0, 4.5, 9.0, 12.0, 8.0, 9.0, 14.0, 7.5)
```

De volgende zaken moeten worden beantwoord voor deze scores: - Wat is het gemiddelde? - Hoeveel leerlingen zijn er geslaagd? - Hoeveel leerlingen zijn er niet geslaagd? - Hoeveel leerlingen hebben een score in de volgende intervallen: - [0 - 6[ - [6 - 10[ - [10 - 13[ - [13 - 20]

Schrijf de resultaten uit.

```
[ ]: ## UW CODE HIER ##
```

### Opdracht: Opdracht: schoonspringen

In een wedstrijd schoonspringen wordt na elke sprong een score (tussen 0 en 100) toegekend door verschillende juryleden. De finale score van de deelnemer wordt dan bepaald door de hoogste en de laagste score te negeren en het gemiddelde te nemen van de andere scores. Dit gemiddelde wordt tenslotte afgerond naar het dichtstbijzijnde natuurlijke getal.

Bereken de eindscore voor een schoonspringer met de volgend deelscores:

```
scores = [59, 70, 75, 46, 89, 55, 91, 74, 56, 55, 62, 74, 83, 48]
```

```
[ ]: ## UW CODE HIER ##
```

### Opdracht: Opdracht: Manhattan-afstand

Voor twee vectoren  $x$  en  $y$  in  $N$  dimensies, met respectieve coördinaten  $(x_0, x_1, \dots, x_{N-1})$  en  $(y_0, y_1, \dots, y_{N-1})$  definieert men de Manhattan-afstand  $d_M(x, y)$  (geïnspireerd op het dambord-achtige stratenpatroon van Manhattan) als:

$$d_M(x, y) = \sum_{i=0}^{N-1} |x_i - y_i|$$

Schrijf een functie `manhattan(a, b)` die de hierboven gedefinieerde Manhattan-afstand als resultaat oplevert. De argumenten zijn lists die reële getallen bevatten. Indien de dimensies van de argumenten `a` en `b` niet gelijk zijn, wordt `-1.0` als resultaat gegeven. Bijvoorbeeld:

```
print(manhattan([1.0, 2.0], [-1.0, -2.0])) geeft een waarde 6.0
```

```
print(manhattan([1.0, 2.0, 3.0], [-1.0, -2.0])) geeft een waarde -1
```

```
[ ]: ## UW CODE HIER ##
```

### Opdracht: afwijkende lijsten

Gegeven zijn twee lijsten  $a$  en  $b$  die gehele getallen bevatten. Beide lijsten bevatten evenveel getallen. Het is de bedoeling om locaties  $i$  te vinden in deze lijsten, waarvoor geldt dat  $|a_i - b_i|$  strikt groter is dan  $d$ .

Programmeer de functie `afwijkende_lijsten()` met als argumenten:

- de lijst `a`;
- de lijst `b`;
- de waarde `d > 0` en waarbij `d` geheel is.

Het resultaat van de functie is een lijst van locaties `i` waarvoor de bovenstaande voorwaarde geldt. De resultaatlijst is van klein naar groot geordend. Voorbeeld:

```
print(afwijkende_lijsten([1, 2, 4, 1, 4, 2],[1, 6, 4, 1, 5, -2], 3))  
resulteert in [1, 5]
```

```
[ ]: ## UW CODE HIER ##
```

### Opdracht: ozonconcentraties

Een weerkundig instituut krijgt dagelijks een bestand met de metingen van het ozongehalte in de lucht op een aantal vastgelegde meetplaatsen. Het ozongehalte wordt hierbij uitgedrukt als een reëel getal en de meetplaatsen als een String. In deze oefening krijg je van enkele dagen de metingen van al de vastgelegde meetplaatsen. Er wordt gevraagd om te bepalen in welke meetplaatsen het ozongehalte op meer dan `k` dagen een bepaalde gegeven kritische waarde overschreed. Ontwerp en implementeer een algoritme voor dit probleem en geef aan hoe daarbij de standaard datatypes kunnen worden gebruikt.

Schrijf een Python-functie `risicoGebieden(metingen: list, k: int, kritischeWaarde: float)`, die een lijst van metingen als argument heeft, alsook een natuurlijk getal `k` en een reëel getal dat de kritische waarde voorstelt. De elementen van de lijst `metingen` zijn van het type `tuple(str, float)` waarbij `destring` de meetlocatie voorstelt en de `float` de meting zelf. De uitvoer van de functie `risicoGebieden` is een set van meetplaatsen waarvan het ozongehalte de kritische waarde meer dan `k` maal overschreden heeft. Voorbeelden:

```
print(risicoGebieden([("Gent",90.1),("Antwerpen",120.9),("Brussel",181.1),("Brugge",70.7),("Gent",150.50),("Antwerpen",190.3),("Brussel",179.4),("Brugge",120.2),("Gent",190.2),("Brussel",200.1),("Brugge",110.1),("Gent",160.4),("Antwerpen",162.1),("Brussel",190.9),("Brugge",120.1),("Gent",180.7),("Antwerpen",125.3),("Brussel",190.1),("Brugge",177.5)], 2, 180)) geeft {'Brussel'})
```

```
print(risicoGebieden([("Gent",90.1),("Antwerpen",120.9),("Brussel",181.1),("Brugge",70.7),("Gent",150.50),("Antwerpen",190.3),("Brussel",179.4),("Brugge",120.2),("Gent",190.2),("Antwerpen",185.1),("Brussel",200.1),("Brugge",110.1),("Gent",160.4),("Antwerpen",182.1),("Brussel",190.9),("Brugge",120.1),("Gent",180.7),("Antwerpen",125.3),("Brussel",190.1),("Brugge",177.5)], 1, 180)) geeft {'Brussel', 'Antwerpen', 'Gent'})
```

```
print(risicoGebieden([("Gent",90.1),("Antwerpen",120.9),("Brussel",181.1),("Brugge",70.7),("Gent",150.50),("Antwerpen",190.3),("Brussel",179.4),("Brugge",120.2),("Gent",190.2),("Antwerpen",185.1),("Brussel",200.1),("Brugge",110.1),("Gent",160.4),("Antwerpen",182.1),("Brussel",190.9),("Brugge",120.1),("Gent",180.7),("Antwerpen",125.3),("Brussel",190.1),("Brugge",177.5)], 1, 180)) geeft {'Brussel', 'Antwerpen', 'Gent'})
```

```
ntwerpen",182.1),("Brussel",190.9),("Brugge",120.1),("Gent",180.7),("Antwerpen",125
.3),("Brussel",190.1),("Brugge",177.5),("Gent",155.2),("Antwerpen",199.2),("Brussel
",200.1),("Brugge",160.1),("Gent",145.2),("Antwerpen",179.2),("Brussel",170.1),("Br
ugge",150.1),("Gent",185.2),("Antwerpen",190.2),("Brussel",185.3),("Brugge",160.7)]
, 3, 180)) geeft ['Brussel', 'Antwerpen']
```

```
[ ]: ## UW CODE HIER ##
```