Complementaire reeksen

All en any

De Python functies `any` en `all` kunnen gebruikt worden om een lijst van booleaanse waarden om te zetten naar één booleaanse waarde. De functie `any` geeft `True` weer als en slechts als de lijst minstens één keer `True` bevat. De functie `all` geeft `True` weer als en slechts als alle waarden uit de lijst `True` zijn.

```
>>> a = ['True', 'True']
>>> b = ['True', 'False']
>>> c = ['False', 'False']
>>> d = ['True', 'True', 'True', 'False']
>>> e = ['False', 'True', 'False', 'False']

>>> all(a)
True
>>> all(b)
False
>>> all(d)
False

>>> any(b)
True
>>> any(c)
False
>>> any(e)
True
```

Ritsen

Simultaan twee of meer iterable objecten overlopen

Als je tegelijkertijd de elementen van twee of meer *iterable objecten* (objecten van samengestelde gegevens types die een geassocieerde iterator hebben) wil overlopen, dan kan je daarvoor gebruik maken van de ingebouwde functie `zip`. Deze functie geeft een iterator terug die eerst een tuple teruggeeft met het eerste element van alle iterable objecten die aan de functie `zip` werden doorgegeven, daarna een tuple met het tweede element van alle iterable objects, enzoverder.

Stel bijvoorbeeld dat je de elementen van twee lijsten bij elkaar wil optellen om daarmee een nieuwe lijst te maken waarvan het i-de element de som is van de i-de elementen van de twee originele lijsten. Dan kan dit op de volgende manier.

```
>>> eerste = [1, 2, 3]
>>> tweede = [4, 5, 6]
>>> opgeteld = []
>>> for term1, term2 in zip(eerste, tweede):
...     opgeteld.append(term1 + term2)
...
>>> opgeteld
[5, 7, 9]
```

Dit kan ook korter geschreven worden door gebruik te maken van een *list comprehension*.

```
>>> eerste = [1, 2, 3]
>>> tweede = [4, 5, 6]
>>> opgeteld = [term1 + term2 for term1, term2 in zip(eerste, tweede)]
```

```
>>> opgeteld
[5, 7, 9]
```

De iterator stopt van zodra één van de iterable objecten die aan de functie `zip` werden doorgegeven uitgeput is. Als je meerdere iterable objecten simultaan wil overlopen totdat alle objecten zijn uitgeput, dan kan je daarvoor gebruik maken van de functie `zip_longest` uit de module `itertools`.

Diffy

Tuples met één enkel element

Een tuple met één enkel element ziet er als volgt uit:

```
1,
```

Het essentiële onderdeel van deze notatie is de komma op het einde. Zoals bij elke expressie kan je optioneel ronde haakjes toevoegen, waardoor een tuple met één enkel element ook als volgt kan genoteerd worden:

```
(1, )
```

Maar het is de komma, niet de ronde haakjes, die zorgt voor de definitie van een tuple. Bekijk bijvoorbeeld het onderstaande voorbeeld:

```
>>> reeks = (3)             # een integer
>>> reeks
3
>>> isinstance(reeks, tuple)
False
>>> isinstance(reeks, int)
True
>>> reeks = (3, )           # een tuple
>>> isinstance(reeks, tuple)
True
```

De komma is noodzakelijk, omdat Python de notatie `(object)` anders interpreteert als het object zelf.

Elementen van een tuple herhalen

Net zoals strings kun je ook tuples vermenigvuldigen met een natuurlijk getal. Hierdoor wordt een nieuw tuple aangemaakt, dat een herhaling bevat van de elementen van het oorspronkelijke tuple.

```
>>> tuple = (2, 3)
>>> tuple * 4
(2, 3, 2, 3, 2, 3, 2, 3)
>>> element = (0, )
>>> 6 * element
(0, 0, 0, 0, 0, 0)
```

Energiecrisis Nieuw-Zeeland

Lijsten van vaste lengte initialiseren met een standaardwaarde

Als je een lijst met een vaste lengte `n` wil aanmaken, waarbij elk element dezelfde waarde `x` heeft, dan hoef je daarvoor geen `for`-lus of een *list comprehension* te gebruiken. Je kunt eenvoudigweg `[x] * n` schrijven.

```
>>> [' '] * 3
[' ', ' ', ' ']
```

```
>>> [1] * 5
[1, 1, 1, 1, 1]
```

Let echter op dat hierbij geen kopie van het object `x` wordt gemaakt, maar dat elk element van de lijst verwijst naar hetzelfde object `x`. Dit is vooral belangrijk als `x` een veranderlijk (*mutable*) object is.

```
>>> lijst = [[1, 2]] * 4
>>> lijst
[[1, 2], [1, 2], [1, 2], [1, 2]]
>>> lijst[0][1] = 666
>>> lijst
[[1, 666], [1, 666], [1, 666], [1, 666]]
>>> lijst[3].append(42)
>>> lijst
[[1, 666, 42], [1, 666, 42], [1, 666, 42], [1, 666, 42]]
```

Algemeen

Controle of bepaalde voorwaarden gelden

Soms moet je in je programmacode expliciet nagaan of er aan bepaalde voorwaarden voldaan is, en moet je programma reageren als één van de voorwaarden geschonden is. Eén van de manieren waarop je dit kunt doen in Python is door gebruik te maken van het `assert` statement.

```
>>> x = 2
>>> y = 2
>>> assert x == y, 'beide getallen zijn niet gelijk'
>>> x = 1
>>> assert x == y, 'beide getallen zijn niet gelijk'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: beide getallen zijn niet gelijk
```

De algemene syntaxis van een `assert` statement is

`assert <voorwaarde>, <boodschap>`

Het `assert` statement controleert of er aan de voorwaarde voldaan is. Indien dat niet het geval is, dan zal er een `AssertionError` opgeworpen worden met de boodschap die wordt meegegeven op het einde van het `assert` statement. Indien deze uitzondering niet wordt opgevangen (wat voor deze cursus altijd het geval zal zijn), dan wordt op het punt van het opwerpen van de `AssertionError` de uitvoer van de code ook beëindigd (*runtime error*).