

Algemeen

Itereren over de posities én de elementen van een collectie

Je kunt de ingebouwde functie `enumerate` gebruiken om een iterator op te vragen van collecties (*iterables*: objecten van samengestelde gegevenstypes zoals `str`, `list`, `tuple`, bestanden, ...), die telkens zowel de positie en het volgende element uit de collectie teruggeeft. Onderstaand voorbeeld geeft bijvoorbeeld aan hoe je voor een string (`str`) tegelijkertijd de posities en de karakters op die posities kunt overlopen.

```
>>> for index, karakter in enumerate('abc'):
...     print(f'index: {index}')
...     print(f'karakter: {karakter}')
...
index: 0
karakter: a
index: 1
karakter: b
index: 2
karakter: c
```

Je kunt dit bijvoorbeeld gebruiken als je synchroon de karakters van twee strings wil overlopen.

```
>>> eerste = 'abc'
>>> tweede = 'def'
>>> for index, karakter in enumerate(eerste):
...     print(f'{karakter}-{tweede[index]}')
...
a-d
b-e
c-f
```

In dat geval is het echter aangewezen om de ingebouwde functie `zip` te gebruiken, die net haar bestaansrecht haalt uit het feit dat ze een iterator teruggeeft voor twee of meer collecties.

```
>>> eerste = 'abc'
>>> tweede = 'def'
>>> for karakter1, karakter2 in zip(eerste, tweede):
...     print(f'{karakter1}-{karakter2}')
...
a-d
b-e
c-f
```

Werken met *floating point* getallen in doctests

Als een functie een *floating point* getal teruggeeft, dan kan dit problemen opleveren als je de correctheid van de functie wilt testen aan de hand van een doctest. Doctests voeren immers een exacte match uit tussen de string die het resultaat voorstelt in de doctest, en het resultaat dat door de functie wordt uitgeschreven of teruggegeven. Hiervoor wordt het resultaat van de functie eerst omgezet naar een string. Bij de vergelijking van de twee strings wordt dus geen rekening gehouden met afrondingsfouten die kunnen optreden bij het werken met *floating point* getallen. Deze afrondingsfouten zijn het gevolg van de beperkte precisie waarmee computers reële getallen kunnen voorstellen.

```
>>> 0.1
0.1
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
```

Als bij het uitvoeren van een doctest de verwachte uitvoer (een string) niet exact overeenkomt met de stringvoorstelling van het resultaat dat door de functie wordt teruggegeven, dan beschouwt de doctest het resultaat als verkeerd.

Dodona houdt hier wel rekening mee. Tenzij anders vermeld in de opgave, controleert Dodona of het resultaat van een functie die een *floating point* getal teruggeeft (hetzij direct, hetzij als element van een collectie) correct is tot op 6 cijfers na de komma. Dit komt ongeveer neer op het schrijven van een doctest volgens het volgende stramien.

```
def vermenigvuldig(x):  
  
    """  
    >>> abs(vermenigvuldig(0.1) - 0.3) < 1e-6  
    True  
    """  
  
    return 3 * x
```

Veranderlijke argumenten doorgeven aan functies

Als je een veranderlijk (*mutable*) object als argument doorgeeft aan een functie, dan kan die functie het object *in place* wijzigen. Dat kan expliciet de bedoeling zijn, maar soms is het niet wenselijk dat waarden die aan een functie doorgegeven worden, gewijzigd worden tijdens het uitvoeren van de functie.

Stel bijvoorbeeld dat we een lijst (*list*) doorgeven aan een functie. Wat we dan eigenlijk doorgeven aan de functie is een verwijzing naar die lijst en geen kopie van de lijst (*call by reference* in plaats van *call by value*). Hierdoor wordt de parameter waaraan de lijst wordt toegekend een alias voor de lijst, en kan de functie dus de lijst zelf wijzigen (lijsten zijn veranderlijke datastructuren).

```
>>> def aanpassen(lijst, element):  
...     lijst.append(element)  
...     return lijst  
...  
>>> lijst = ['a', 'b']  
>>> aangepast = aanpassen(lijst, 'c')  
>>> aangepast  
['a', 'b', 'c']  
>>> lijst  
['a', 'b', 'c']
```

In het voorbeeld hieronder maken we eerst een kopie van de lijst die aan de functie werd doorgegeven. Daarna passen we de kopie aan, maar dus niet de originele lijst. Een kopie van een lijst maken, kan je bijvoorbeeld met behulp van *slicing* (`lijst[:]`) of aan de hand van de ingebouwde functie `list` (`list(lijst)`).

```
>>> def aanpassen(lijst, element):  
...     kopie = lijst[:]  
...     kopie.append(element)  
...     return kopie  
...  
>>> lijst = ['a', 'b']  
>>> aangepast = aanpassen(lijst, 'c')  
>>> aangepast  
['a', 'b', 'c']  
>>> lijst  
['a', 'b']
```

De Python Tutor geeft je een grafische voorstelling van het verschil tussen bovenstaande voorbeelden:

- [voorbeeld zonder kopie](#)
- [voorbeeld met kopie](#)

Omdat we de verwijzing naar de originele lijst die aan de functie werd doorgegeven niet meer nodig hebben, kunnen we de functie `aanpassen` uit bovenstaand voorbeeld ook op de volgende manier herschrijven.

```
def aanpassen(lijst, element):
    lijst = lijst[:]
    lijst.append(element)
    return lijst
```

Hierbij wordt de verwijzing van de variabele `lijst` naar de originele lijst die aan de functie `aanpassen` wordt doorgegeven, vervangen door een verwijzing naar een kopie van de lijst. Deze vervanging is enkel zichtbaar binnen de functie, omdat de variabele `lijst` een lokale variabele is van de functie `aanpassen`. Ook dit [voorbeeld](#) kan je bekijken aan de hand van de Python Tutor.

Lijsten sorteren

Er zijn twee manieren om lijsten te sorteren. Ofwel roep je de lijstmethode `sort` aan op de lijst, ofwel geef je de lijst door aan de ingebouwde functie `sorted`. Er is echter wel een belangrijk verschil tussen deze twee opties. De lijstmethode `sort` wijzigt de lijst *in place* (en geeft geen nieuwe lijst maar de waarde `None` terug), terwijl de ingebouwde functie `sorted` een nieuwe lijst teruggeeft waarvan de elementen van klein naar groot gesorteerd zijn.

```
>>> lijst = [4, 2, 3, 1]
>>> lijst.sort()
>>> lijst
[1, 2, 3, 4]
>>>
>>> lijst = [4, 2, 3, 1]
>>> sorted(lijst)
[1, 2, 3, 4]
```

Elementen van een lijst groeperen

Er zijn verschillende mogelijkheden om een lijst van elementen op te delen in groepen van elk n elementen. Je kunt bijvoorbeeld een *list comprehension* gebruiken.

```
>>> lijst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> [(lijst[i], lijst[i + 1]) for i in range(0, len(lijst), 2)]
[('a', 'b'), ('c', 'd'), ('e', 'f'), ('g', 'h')]
```

Je kunt echter ook de ingebouwde functie `zip` gebruiken die twee of meer lijsten simultaan overloopt. De iterator geeft telkens een tuple terug met de elementen op de i -de positie in elk van de lijsten die aan de functie `zip` doorgegeven worden.

Als je dan de lijst met de elementen op de even posities (`lijst[::2]`) en de lijst van de elementen op de oneven posities (`lijst[1::2]`) simultaan overloopt met de ingebouwde functie `zip`, dan krijg je net hetzelfde resultaat.

```
>>> lijst = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
>>> list(zip(lijst[::2], lijst[1::2]))
[('a', 'b'), ('c', 'd'), ('e', 'f'), ('g', 'h')]
```

All en any

De ingebouwde functies `all` en `any` kunnen gebruikt worden om een lijst van Booleaanse waarden (`bool`) om te zetten naar één Booleaanse waarde. De functie `all` geeft `True` terug als en slechts als alle waarden uit de

lijst `True` zijn. De functie `any` geeft `True` terug als en slechts als de lijst minstens één keer `True` bevat.

```
>>> 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
```

Controle of bepaalde voorwaarden gelden

Soms moet je in een programma uitdrukkelijk 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 is door een `assert` statement te gebruiken.

```
>>> 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. Als dat niet het geval is, dan zal er een `AssertionError` opgeworpen worden met de boodschap (`str`) die wordt meegegeven op het einde van het `assert` statement. Als deze uitzondering (Engels: *exception*) niet wordt opgevangen (wat voor deze cursus altijd het geval zal zijn), dan wordt de uitvoer van de code ook beëindigd (*runtime error*) met een melding van de `AssertionError`.

De lijstmethode `insert`

De lijstmethode `append` kan gebruikt worden om een nieuw element toe te voegen aan het einde van een lijst. De lijst methode `insert` laat toe om een element toe te voegen op een aangegeven positie in de lijst. Als de positie die wordt doorgegeven aan de lijstmethode `insert` groter dan of gelijk is aan de lengte van de lijst, dan wordt het element achteraan de lijst toegevoegd.

```
>>> lijst = []
>>> lijst.insert(0, 'a')
>>> lijst
```

```

['a']
>>> lijst.insert(0, 'b')
>>> lijst
['b', 'a']
>>> lijst.insert(1, 'c')
>>> lijst
['b', 'c', 'a']
>>> lijst.insert(10, 'd')
>>> lijst
['b', 'c', 'a', 'd']

```

ISBN

De stringmethode join

De stringmethode `join` kan gebruikt worden om alle strings in een *iterable object* (bv. een lijst) samen te voegen tot één enkele string. Hierbij worden alle strings van het *iterable object* samengevoegd, van elkaar gescheiden door een scheidingsteken waarop de stringmethode `join` wordt aangeroepen.

```

>>> lijst = ['a', 'b', 'c']
>>> ' '.join(lijst)
'a b c'
>>> ''.join(lijst)
'abc'
>>> '---'.join(lijst)
'a---b---c'
>>> ' - '.join(lijst)
'a - b - c'

```

Samenraapsels

Simultaan twee of meer iterables overlopen

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

Stel bijvoorbeeld dat je de elementen van twee lijsten bij elkaar wilt 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 een *list comprehension* te gebruiken.

```
>>> 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 iterables die aan de functie `zip` werden doorgegeven uitgeput is (= een `StopIteration` heeft opgeworpen). Als je meerdere iterables simultaan wilt overlopen totdat alle collecties uitgeput zijn, dan kan je daarvoor de functie `zip_longest` uit de module `itertools` gebruiken.