Algemeen

Veranderlijke argumenten doorgeven aan functies

Als je een veranderlijk (*mutable*) object 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 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 onderstaand voorbeeld maken we eerst een kopie van de lijst die aan de functie werd doorgegeven. Daarna passen we de kopie, maar dus niet de originele lijst aan. Een kopie maken van een lijst 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 hierschrijven.

```
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

In Python zijn er 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 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]
```

Geneste lijsten initialiseren

Een rooster met m rijen en n kolommen kan voorgesteld worden door een lijst `l` van m lijsten, waarbij iedere lijst uit `l` n elementen bevat. Stel bijvoorbeeld dat je een rooster van 3 rijen en 2 kolommen wil maken, waarbij elk element uit het rooster een lege string is. Dit kan je doen door gebruik te maken van *list comprehensions*.

```
>>> m = 3
>>> n = 2
>>> rooster = [['' for _ in range(n)] for _ in range(m)]
>>> rooster
[['', ''], ['', ''], ['', '']]
>>> rooster[0][0] = 'A'
>>> rooster
[['A', ''], ['', ''], ['', '']]
```

In veel gevallen kan je ook de vermenigvuldigingsoperator `*` gebruiken om een lijst van een bepaalde grootte aan te maken, waarbij elk element van de lijst hetzelfde is. Maar als dit element mutable is, dan kan dit vreemde resultaten opleveren.

```
>>> m = 3
>>> n = 2
>>> [''] * m
['', '', '']
>>> rooster = [[''] * n] * m
>>> rooster
[['', ''], ['', ''], ['', '']]
>>> rooster[0][0] = 'A'
>>> rooster
[['A', ''], ['A', ''], ['A', '']]
```

Zoals je kan zien, wordt er bij het plaatsen van een `A` op positie `[0][0]` in het rooster ook een `A` geplaatst op posities `[1][0]` en `[2][0]`. De reden hiervoor is dat de vermenigvuldigingsoperator `*` aliassen maakt van de lijst, waardoor zowel `rooster[0]`, `rooster[1]` als `rooster[2]` verwijzen naar hetzelfde lijstobject. Als je dus één lijst aanpast, pas je ze allemaal aan. Dit kan je goed zien als je gebruik maakt van de Python Tutor.

Elementen van een lijst groeperen

In Python zijn er verschillende mogelijkheden om een lijst van elementen op te delen in groepen van elk n elementen. Je kunt bijvoorbeeld gebruik maken van een list comprehension.

```
>>> 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 gebruik maken van de ingebouwde functie `zip` 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']
>>> zip(lijst[::2], lijst[1::2])
[('a', 'b'), ('c', 'd'), ('e', 'f'), ('g', 'h')]
```

Stringvoorstelling van een rooster

De volgende *list comprehension* bouwt een de stringvoorstelling van een rooster op, waarbij de rijen van het rooster op afzonderlijke regels weergegeven worden. Met andere woorden, de regels worden telkens van elkaar gescheiden door een newline (de string waarop de buistenste `join` methode wordt aangeroepen). De elementen van de rijen worden telkens van elkaar gescheiden door één enkele spatie (de string waarop de binnenste `join` methode wordt aangeroepen).

```
>>> rooster = [['A', 'B', 'C'], ['D', 'E', 'F'], ['G', 'H', 'I']]
>>> print('\n'.join([' '.join(rij) for rij in rooster]))
A B C
D E F
G H I
```

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. Indien 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']
```

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]]
```