Algemeen

Itereren over posities en waarden van sequentietypes

De ingebouwde functie `enumerate` kan gebruikt worden om een een iterator op te vragen van een sequentietype (*sequence type*: strings, lijsten, tuples, bestanden, ...), die zowel de positie als de waarde van het volgende element van het type teruggeeft. Onderstaand voorbeeld geeft bijvoorbeeld aan hoe je tegelijkertijd de posities en de karakters op die posities kunt overlopen voor een string.

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

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

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

In dat geval is het echter aangewezen om gebruik te maken van de ingebouwde functie `zip`, die net haar bestaansrecht haal uit het feit dat ze een iterator teruggeeft voor twee of meer iterateerbare objecten.

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

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.

Lineup

Algemene info

Opmerkingen

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']
```

Heremietkreeften

Algemene info

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.

Koninginnen, paarden en pionnen

Specifieke info

Voor deze oefening bouw je best een rooster op dat overeenkomt met het schaakbord, en duid je voor iedere stuk de plaatsen aan die dat stuk bedreigd.

Om te bepalen welke plaatsen bedreigd worden door de koningin, moeten er 8 richtingen gecontroleerd worden. Hiervoor kun je eerst een lijst aanmaken met alle mogelijke richtingen.

```
richtingen = [(0, 1), (1, 0), (0, -1), (-1, 0), (1, 1), (1, -1), (-1, 1), (-1, -1)]
```

Waarbij voor `(r, k)` `r` overeenkomt met de verandering in rij (1 is omlaag, -1 is omhoog) en `k` overeenkomt met de verandering in kolom (1 is naar rechts en -1 is naar links). Zo kom `(0, 1)` overeen met de positie rechts en `(1, -1)` met de positie links onder.

Vervolgens kun je een lus schrijven die de richtingen overloopt en gegeven de verandering in rij en kolom, het stuk code uitvoert dat bepaalt welke posities door een koningin bedreigd worden in die richting.

Dronken mier

Algemene info

Strings wijzigen

Een belangrijk verschil tussen strings en lijsten is dat strings onveranderlijk (*immutable*) zijn en lijsten veranderlijk (*mutable*). Dit betekent dat je elementen van een lijst kunt vervangen door andere elementen zonder daarbij een nieuwe lijst te moeten aanmaken, of dat je nieuwe elementen aan de lijst kunt toevoegen of elementen uit een lijst kunt verwijderen. Met strings kan dat niet. Eenmaal een string is aangemaakt kan je die string zelf niet meer veranderen. Je kan enkel een nieuwe string aanmaken op basis van de bestaande string.

```
>>> s = 'darwin'
>>> l = ['d', 'a', 'r', 'w', 'i', 'n']
>>> s[0] = 'D'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> l[0] = 'D'
>>> l
['D', 'a', 'r', 'w', 'i', 'n']
```

Als je toch sommige karakters van een string wilt wijzigen, dan kan het dus handig zijn om eerst de string om te zetten naar een lijst van karakters (aan de hand van de ingebouwde functie `list`), de wijzigingen door te voeren op de lijst, en finaal alle karakters van de gewijzigde lijst terug aan elkaar te plakken (aan de hand van de stringmethode `join`).

```
>>> s = 'darwin'
>>> l = list(s)
>>> l
['d', 'a', 'r', 'w', 'i', 'n']
>>> l[0] = 'D'
>>> l
['D', 'a', 'r', 'w', 'i', 'n']
>>> s = ''.join(l)
>>> s
'Darwin'
```