De laatste knikker

De module `random`

De module random uit The Python Standard Library kan gebruikt worden om willekeur toe te voegen aan je Python code. Deze module bevat onder andere de volgende functies.

functie	korte omschrijving
`random()`	genereert een willekeurig reëel getal uit het interval $[0, 1[$
`randint(a, b)`	genereert een willekeurig geheel getal uit het interval $[a, b]$
`choice(s)`	kiest een willekeurig element uit de sequentie `s`
`sample(s, k)`	kiest willekeurig `k` verschillende elementen uit de sequentie `s`
`shuffle(l)`	schudt de elementen van de lijst l willekeurig door elkaar

Hieronder staan alvast enkele voorbeelden.

```
>>> import random

>>> random.random()
0.954131645221452
>>> random.random()
0.3548429482674793

>>> random.randint(3, 10)
5
>>> random.randint(3, 10)
8

>>> lijst = ['a', 'b', 'c']
>>> random.choice(lijst)
'b'
>>> random.choice(lijst)
'a'
>>> lijst
['a', 'b', 'c']

>>> random.sample(lijst, 2)
['a', 'c']
>>> random.sample(lijst, 2)
['b', 'a']
>>> lijst
['a', 'b', 'c']

>>> random.shuffle(lijst)
>>> lijst
['c', 'a', 'b']
```

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.

Elementen verwijderen uit een lijst

Als je een lijst hebt met n elementen, en je verwijdert het element op positie i. Dan moet je er op letten dat na het verwijderen van dat element, de elementen die op posities $i+1, i+2, \ldots, n$ stonden nu op posities $i, i+1, i+2, \ldots, n-1$ staan. De elementen op posities $0, \ldots, i-1$ blijven wel op dezelfde index staan.

Obscure feestdagen

De module `datetime`

De module datetime uit The Python Standard Library definieert een aantal nieuwe gegevenstypes die datums (`datetime.date` objecten) en tijdsintervallen (`datetime.timedelta` objecten) voorstellen. Hieronder alvast enkele voorbeelden.

```
>>> from datetime import date
>>> geboortedatum = date(1990, 10, 3)
>>> geboortedatum = date(day=3, month=10, year=1990)
>>> geboortedatum.day          # day is een eigenschap
3
>>> geboortedatum.month        # month is een eigenschap
10
>>> geboortedatum.year         # year is een eigenschap
1990
>>> geboortedatum.weekday()    # weekday is een methode !!
2
>>> vandaag = date.today()
>>> vandaag
datetime.date(2015, 11, 10)    # uitgevoerd op 11 oktober 2015
>>> from datetime import timedelta
>>> morgen = vandaag + timedelta(1)
>>> morgen
datetime.date(2015, 11, 11)
>>> verschil = morgen - vandaag
>>> type(verschil)
datetime.timedelta
>>> verschil.days
1
```

Curling

Sorteren volgens optionele parameter `key`

De lijstmethode `sort` en de ingebouwde functie `sorted` kunnen beide gebruikt worden om een gegeven lijst van elementen te sorteren. Ze verschillen echter in het feit dat de `sort` methode de lijst *in place* bijwerkt, terwijl de functie `sorted` een nieuwe gesorteerde lijst teruggeeft en de oude lijst intakt laat.

Daarnaast hebben beide heel wat gelijkenissen. Ze beschikken alletwee over een optionele parameter `reverse` waaraan een Booleaanse waarde kan doorgegeven worden die aangeeft of er van klein naar groot (waarde `False`, de standaardwaarde) of van groot naar klein (waarde `True`) moet gesorteerd worden. Beide beschikken ze ook over een tweede optionele parameter `key` die kan gebruikt worden om de volgorde van de elementen te bepalen. Deze volgorde zal immers gebruikt worden bij het sorteren.

Aan de parameter `key` kan een functie doorgegeven worden. Aan deze functie moet één enkel argument kunnen doorgegeven worden. Indien er een functie `f` wordt doorgegeven aan de parameter `key`, dan zal de volgorde bij het sorteren niet bepaald worden op basis van de elementen zelf, zoals standaard het geval is, maar op basis van de waarde die de functie `f` teruggeeft voor elk van de elementen (elk element worden dus afzonderlijk als een argument doorgegeven aan de functie `f`).

Stel bijvoorbeeld dat je een functie `f` definieert en deze functie doorgeeft aan de parameter `key`. Dan zal bij het sorteren eerst `f(element)` aangeroepen worden voor elk `element` uit de lijst die moet gesorteerd worden. Daarna zullen de elementen uit de lijst gesorteerd worden, op basis van de waarden die door de functie `f` teruggegeven worden voor elk van deze elementen. Op de eerste positie in de gesorteerde lijst vind je dus het `element` waarvoor `f(element)` de kleinste waarde heeft (of de grootste waarde als je `reverse=True` instelt), en op de laatste positie in de gesorteerde lijst vind je het `element` waarvoor `f(element)` de grootste waarde heeft (of de kleinste waarde als je `reverse=True` instelt).

De natuurlijke volgorde waarin tuples gesorteerd worden is door eerst te sorteren op het eerste element van het tuple, en indien dit gelijk is, verder te sorteren op het volgende element van het tuple. Stel bijvoorbeeld dat we een lijst van tuples hebben, waarbij elk tuple bestaat uit twee natuurlijke getallen. Deze tuples kunnen op de volgende manier gesorteerd worden.

```
>>> lijst = [(2, 7), (0, 10), (4, 0), (1, 6), (2, 5), (2, 6)]
>>> sorted(lijst)
[(0, 10), (1, 6), (2, 5), (2, 6), (2, 7), (4, 0)]
```

Als we echter de tuples eerst willen sorteren op basis van hun tweede element, en daarna pas op basis van hun eerste element, dan kunnen we dat op de volgende manier realiseren.

```
>>> def sorteersleutel(paar):
...     return paar[1], paar[0]
...
>>> lijst = [(2, 7), (0, 10), (4, 0), (1, 6), (2, 5), (2, 6)]
>>> sorted(lijst, key=sorteersleutel)
[(4, 0), (2, 5), (1, 6), (2, 6), (2, 7), (0, 10)]
```

Merk op dat deze volgorde niet gelijk is aan de omgekeerde volgorde van de natuurlijke sortering van de elementen.

Paarsgewijs vergelijken van elementen in een lijst

Als je een lijst hebt waarvan je de elementen één per één met elkaar moet vergelijken, kun je dit doen met een dubbele for lus die itereert over de indexen van de elementen in de lijst.

```
>>> lijst = ['a', 'b', 'c', 'd']
>>> for i in range(len(lijst)):
...     for j in range(i + 1, len(lijst)):
...         vergelijk(lijst[i], lijst[j])
```

Dit zal achtereenvolgens de volgende vergelijkingen doen:

- elementen op indexen 0 en 1 (‘a’ met ‘b’)
- elementen op indexen 0 en 2 (‘a’ met ‘b’)
- elementen op indexen 0 en 3 (‘a’ met ‘b’)
- elementen op indexen 1 en 2 (‘a’ met ‘b’)
- elementen op indexen 1 en 3 (‘a’ met ‘b’)
- elementen op indexen 2 en 3 (‘a’ met ‘b’)

Afstand tussen 2 punten

Om de afstand tussen twee punten met coordianten (x_1, y_1) en (x_2, y_2) te berekenen, kan gebruikt gemaakt worden van de formule

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Specifieke info

Voor de functie score kan het handig zijn om de lijst met stenen te sorteren op afstand tot de dolly. Zo staat de steen die het dichts bij de dolly ligt op de eerste positie, de tweede dichtste op de tweede positie, enz.

Five up

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
```

De stringmethode `join`

De stringmethode `join` kan gebruikt worden om alle string 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'
```