Algemeen

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']
```

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
```

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.

None als standaardwaarde

Als je een functie wil definiëren met een optionele parameter, dan moet je aan die parameter altijd een standaarwaarde toekennen. Hierbij kan het gebeuren dat deze standaardwaarde niet gekend is op het ogenblik dat de functie gedefinieerd wordt, maar dat die pas kan bepaald worden op het ogenblik dat de functie wordt aangeroepen. Dat is bijvoorbeeld het geval als je een standaardwaarde wil berekenen op basis van argumenten die aan andere parameters doorgegeven worden. In dat geval is het een goed idee om als standaardwaarde de waarde `None` te gebruiken bij het definiëren van de functie, en in de body van de functie een waarde te berekenen indien aan de parameter de waarde `None` werd doorgegeven (betekenis: er werd niet expliciet een argument aan deze parameter doorgegeven).

```
def functie(eerste, tweede=None):
    if tweede is None:
        # ken dezelfde waarde toe aan de parameter tweede als het argument dat
        # werd doorgegeven aan de parameter eerste, in het geval dat er niet
        # expliciet een tweede argument werd doorgegeven aan de functie
        tweede = eerste

    ...
```

Dit kan je dus niet op de volgende manier oplossen

```
def functie(eerste, tweede=eerste):

    ...
```

omdat op het moment dat je de functie aan het definiëren bent, de parameter `eerste` nog niet naar een waarde verwijst. Dit is net alsof je een variabele gebruikt die nog niet gedefinieerd werd.

Je gebruikt best ook `None` als standaardwaarde, als de standaardwaarde die je eigenlijk wil toekennen een veranderlijk (*mutable*) gegevenstype heeft. Je schrijft dus beter niet

```
def functie(eerste, tweede=[]):

    # OPMERKING: in dit geval wordt er één enkele lege lijst aangemaakt op het
    #            ogenblik dat de functie gedefinieerd wordt; omdat lijsten
    #            mutable zijn, is het mogelijk dat deze lijst wordt aangepast
    #            telkens als de functie wordt aangeroepen; doorgaans is dit niet
    #            het gewenste gedrag

    ...
```

maar

```
def functie(eerste, tweede=None):

    # lege lijst instellen als standaardwaarde
    # OPMERKING: in dit geval wordt er bij elke aanroep van de functie een
    #            nieuwe lege lijst aangemaakt, specifiek voor die functie-aanroep
    if tweede is None:
        tweede = []

    ...
```

Als je aan een parameter een standaardwaarde wil toekennen die vastligt op het ogenblik dat je een functie definieert, en als die standaardwaarde een onveranderlijk (*immutable*) gegevenstype heeft (`int`, `bool`, `string`, `tuple`, ...) dan kan je die zonder problemen op de klassieke manier toekennen aan de parameter.

```
def functie(eerste, tweede=42):

    ...
```

Functies toekennen aan variabelen

In Python zijn functies zelf objecten van het gegevenstype `function`, waardoor je ze net als andere objecten kunt toekennen aan een variable. Dat kan handig zijn als je twee stukken programmacode hebt die zeer gelijkaardig zijn, en enkel verschillen door het feit dat er in het ene stuk code een andere functie moet aangeroepen worden dan in het andere stuk code.

Denk bijvoorbeeld aan een programma waarin voor een gegeven woordenlijst eerst alle woorden moeten afgedrukt worden die de letter `a` bevatten, en daarna ook alle woorden die de letter `b` bevatten. Dit zou je op de volgende manier kunnen oplossen

```
>>> woorden = ['appel', 'banaan', 'bes']
>>>
>>> def bevat_a(woord):
...     return 'a' in woord
...
>>> def bevat_b(woord):
...     return 'b' in woord
...
>>> for woord in woorden:
...     if bevat_a(woord):
...         print(woord)
...
appel
banaan
>>> for woord in woorden:
...     if bevat_b(woord):
...         print(woord)
banaan
bes
```

De twee `for`-lussen in bovenstaand codefragment kunnen nu als volgt herschreven worden

```
>>> functie = bevat_a
>>> for woord in woorden:
...     if functie(woord):
...         print(woord)
```

```
...
appel
banaan
>>> functie = bevat_b
>>> for woord in woorden:
...     if functie(woord):
...         print(woord)
banaan
bes
```

Waardoor we twee keer exact dezelfde lus krijgen. Om duplicatie van code te vermijden (we programmeren liever niet twee keer dezelfde code), kunnen we dit herschrijven door een nieuwe lus te introduceren die over de twee functie loopt

```
>>> functies = [bevat_a, bevat_b]
>>> for functie in functies:
...     for woord in woorden:
...         if functie(woord):
...             print(woord)
appel           # genereerd door eerste iteratie (functie == bevat_a)
banaan
banaan          # genereerd door tweede iteractie (functie == bevat_b)
bes
```