# Algemeen

### Sorteren in dalende volgorde

Standaard sorteren de lijstmethode `sort` en de ingebouwde functie `sorted` de elementen van een lijst in stijgende volgorde. Beide hebben echter ook een optionele parameter `reverse` waaraan de waarde `True` kan doorgegeven worden om de elementen in dalende volgorde te laten sorteren.

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

### Tuples met één enkel element

Een tuple met één enkel element ziet er als volgt uit:

```
1,
```

Het essentiële onderdeel van deze notatie is de komma op het einde. Zoals bij elke expressie kan je optioneel ronde haakjes toevoegen, waardoor een tuple met één enkel element ook als volgt kan genoteerd worden:

```
(1, )
```

Maar het is de komma, niet de ronde haakjes, die zorgt voor de definitie van een tuple. Bekijk bijvoorbeeld het onderstaande voorbeeld:

```
>>> reeks = (3)             # een integer
>>> reeks
3
>>> isinstance(reeks, tuple)
False
>>> isinstance(reeks, int)
True
>>> reeks = (3, )           # een tuple
>>> isinstance(reeks, tuple)
True
```

De komma is noodzakelijk, omdat Python de notatie `(object)` anders interpreteert als het object zelf.

### Elementen van een tuple herhalen

Net zoals strings kun je ook tuples vermenigvuldigen met een natuurlijk getal. Hierdoor wordt een nieuw tuple aangemaakt, dat een herhaling bevat van de elementen van het oorspronkelijke tuple.

```
>>> tuple = (2, 3)
>>> tuple * 4
(2, 3, 2, 3, 2, 3, 2, 3)
>>> element = (0, )
>>> 6 * element
(0, 0, 0, 0, 0, 0)
```

### 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 gebruikmaken 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 gebruikmaken 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')]
```

## 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'
```

## Telefoonboekverdeling

### All en any

De Python 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 in Python kunt doen is door gebruik te maken van een `assert` statement.

```
>>> 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`.

## Samenraapsels

### Simultaan twee of meer iterable objecten overlopen

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

Stel bijvoorbeeld dat je de elementen van twee lijsten bij elkaar wil 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 gebruik te maken van een *list comprehension*.

```
>>> 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 iterable objecten die aan de functie `zip` werden doorgegeven uitgeput is. Als je meerdere iterable objecten simultaan wil overlopen totdat alle objecten zijn uitgeput, dan kan je daarvoor gebruikmaken van de functie `zip_longest` uit de module `itertools`.

# Gevederde vrienden

### 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 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]
```

# Koppelbureau

### Lijsten van vaste lengte initialiseren met een standaardwaarde

Als je een lijst (`list`) 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]]
```

## Specifieke info

In deze oefening wordt de pseudocode van het algoritme gegeven, maar moet je zelf wel nog bepalen welke datastructuren je zult gebruiken om dit algoritme zo eenvoudig mogelijk te implementeren. Met datastructuren bedoelen we de gegevenstypes van de objecten waarmee je de data zal voorstellen en bijhouden.

De gegevens die je voor deze opgave moet bijhouden zijn:

1. de vrouw waaraan elke man op dit moment gekoppeld is (met de mogelijkheid dat een man op dit moment niet aan een vrouw gekoppeld is)
2. de man waaraan elke vrouw op dit moment gekoppeld is (met de mogelijkheid dat een vrouw op dit moment niet aan een man gekoppeld is)
3. aan welke vrouwen uit zijn voorkeurslijst een man al gekoppeld geweest is tijdens het uitvoeren van het algoritme

Hier zijn een aantal mogelijkheden die je kunt overwegen:

1. je kan een lijst bijhouden waarin alle mannen worden bijgehouden die op dit moment niet aan een vrouw gekoppeld zijn; bij aanvang van het algoritme bevat deze lijst alle mannen; telkens een man aan een vrouw gekoppeld wordt, wordt die man uit de lijst verwijderd; als een man van een vrouw ontkoppeld wordt, wordt die man terug toegevoegd aan de lijst
2. je kan een lijst bijhouden waarin je op positie $i$ bijhoudt aan welke man vrouw $i$ op dit moment gekoppeld is; als op positie $i$ de waarde `None` staat, dan kunnen we dit meteen ook interpreteren als het feit dat vrouw $i$ op dit moment nog niet aan een man gekoppeld is
3. je kan een tweedimensionaal rooster van Booleaanse waarden (`True` of `False`) bijhouden, waarvan de waarde op positie $(i, j)$ aangeeft of man $i$ door het algoritme al gekoppeld werd aan vrouw $j$
4. als alternatief kan je ook een lijst bijhouden waarvan positie $i$ aangeeft met hoeveel vrouwen man $i$ reeds gekoppeld werd door het algoritme; bij aanvang van het algoritme zijn alle mannen nog aan geen enkele vrouw gekoppeld geweest