Python Essentials

Inhoudsopgave

1. Inleiding: waarom Python?....4

2. Enkele demo-programma's....5

3. Variabelen....7

4. Datatypen....8

5. Getallen....8
Conversie tussen getallen....9
Rekenkundige expressies....10
Wiskundige functies....11

6. Sequences....12

7. Lijsten....12
Concatenatie....13
Een lijst herhalen....14
Slicing: een gedeelte van een lijst selecteren....15
Elementen uit de lijst verwijderen....16
Elementen uit de lijst wijzigen....17
Elementen aan een lijst toevoegen....18
Ingebouwde lijst-functies:....19
Ingebouwde lijst-methoden....20

8. Tupels....21

9. Range....22

10. Strings....24
Een deelstring zoeken....27
Een string opsplitsen....28
Characters....30

11. Bytes en byte-array....32

12. Eenvoudige I/O....33
De print-opdracht....33
De input-opdracht....33

13. Beweringen....34
Relationele operatoren....34
Logische operatoren....34
Lazy evaluation....35
Het testen van objecten....35

14. Opdrachten....36
De toekenningsopdracht....36

Verkorte schrijfwijzen....36
Samenvoegen en uitpakken....37
De toekenningsopdracht bij lijsten....38
Besturingsopdrachten....39
Het if-statement....39
Het if-else statement....40
Het geneste if-else statement....41
Het for-statement....42
Het while-statement....45

15. Command-line argumenten....46

16. Foutafhandeling....47

17. Functies....49
De definitie van een functie....50
De aanroep van een functie....50
Globale en lokale variabelen....50
De waarde die de functie teruggeeft....52
Parameters....53
Meerdere waarden teruggeven....55
Parameters met default-waarden....56
Functies met een onbepaald aantal argumenten....57
Argumenten-lijst uitpakken....57

18. Klassen in Python....58

19 Dictionary....61
Een dictionary met een for-loop doorlopen....62
Klasse versus dictionary....63

20. Modulen in Python....65
Een module importeren....66
Een module vinden....67
De inhoud van een module....67

21. Files in Python....68
Lezen van en schrijven naar files....68
Tekstfiles en binaire files....70
Files kopiëren....71
Het filesysteem....72

1. Inleiding: waarom Python?

De belangstelling voor Python is de laatste jaren enorm toegenomen.

Python heeft een aantal voordelen:

- compact
- eenvoudig
- sluit goed aan op pseudo-code
- geschikt om concepten uit te leggen
- sluit goed aan op de wiskundige notatie
- heeft een interactieve mode

Python heeft ook nadelen:

- het biedt weinig bescherming: de verantwoordelijkheid ligt bij de programmeur
- de documentatie is minder goed
- fouten zijn af en toe lastig te achterhalen
- de taal kan cryptisch zijn
- sommige taalconstructies zijn onhandig: 'True' en 'False' moeten bijvoorbeeld met een hoofdletter
- niet zo snel als C++ (een combinatie: cython. Zie http://docs.cython.org/index.html)

Kenmerken van Python:

- Python-code wordt omgezet naar byte-code. De byte-code wordt door de PVM (Python Virtual Machine) verwerkt. Daarom zijn Python-programma's trager dan C-programma's.
- Python is dynamisch:
  - variabelen in Python zijn niet statisch gekoppeld aan een type en hoeven niet vooraf gedeclareerd te worden.
  - aan objecten kunnen attributen toegevoegd en verwijderd worden
  - als een module geïmporteerd wordt, wordt de module ook meteen uitgevoerd
- Python is een OO-taal : in Python is alles een object: getallen, lijsten, dictionaries, functies en files.
- In Python spelen lijsten een centrale rol. Een lijst is een dynamische array.
  Een belangrijke concept is 'list comprehension'
- Gehele getallen kennen geen onder- en bovengrens.
- Strings zijn opgebouwd uit unicode-characters (Python 3)
- Python maakt onderscheid tussen mutable en immutable objects
- Python ondersteunt concepten van functionele talen: 'map', 'filter' en 'reduce'.
  Hiermee kunnen iteraties (zoals for- en while-opdrachten) worden vermeden.

2. Enkele demo-programma's.

Demo1.py:

```
x = 56
y = 78
print('x:', x)
print('y:', y)
print('x+y:', x+y)
```

Uitvoer:

```
x: 56
y: 78
x+y: 134
```

Toelichting:
In Python zijn variabelen niet gebonden aan een type.
De opdracht '**x = 56**' koppelt '**x**' aan de waarde '56' . Het type van '**x**' is nu '**int**'.
Het type van '**x**' kan later gewijzigd worden.

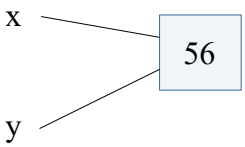
Een uitgebreider voorbeeld:

```
x = 56
print('x:', x)
print('type(x):', type(x))
print('id(x):', id(x))

y = 56
print('y:', y)
print('id(y):', id(y))

x += 1
print('x:', x)
print('id(x):', id(x))

x = 'string'
print('x:', x)
print('type(x):', type(x))
print('id(x):', id(x))
```

Uitvoer:

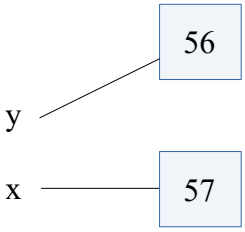
```
x: 56
type(x): <class 'int'>
id(x): 2003188352
y: 56
id(y): 2003188352
x: 57
id(x): 2003188368
x: string
type(x): <class 'str'>
id(x): 27083072
```

Toelichting:



'**x**' en '**y**' zijn eerst aan hetzelfde int-object gekoppeld. Ze hebben dezelfde 'identity' .

Na de opdracht '`x += 1`' is 'x' aan een ander int-object gekoppeld.



Bij de Cpython-implementatie is id(x) het intern geheugen-adres van 'x' .
(zie https://docs.python.org/3.5/library/functions.html#id)

Int-objecten zijn 'immutable'. Als '**x**' van waarde verandert, wordt '**x**' aan een ander object gekoppeld.

3. Variabelen.

In de voorgaande voorbeelden zijn variabelen gebruikt.
Een variabele is een naam (identifier) die verwijst naar een object.
Een identifier bestaat uit letters, cijfers en underscores (_) en mag niet beginnen met een cijfer.

De variabelen worden in een tabel (dictionary) bijgehouden:

De opdracht '`print(dir())`' geeft de lijst van variabelen.
Variabelen, die door Python zelf worden gebruikt, worden dan ook getoond. Deze variabelen beginnen en eindigen met een dubbele underscore: __name__ , __doc__ , etc.

De variabele '**x**' kan worden verwijderd met de opdracht: '`del x`'.

Voorbeeld:

```
x = y = 7
print([i for i in dir() if not (i.startswith('__') and i.endswith('__'))])
del x
print([i for i in dir() if not (i.startswith('__') and i.endswith('__'))])
```

Uitvoer:

```
['x', 'y']
['y']
```

4. Datatypen.

Belangrijke datatypen in Python zijn:

- numerical types: getallen: geheel , gebroken , complex
- sequence types: lijsten, tupels , ranges, strings, bytes
- sets (alle elementen zijn uniek)
- dictionaries (een verzameling (key, value)-paren)
- files

5. Getallen.

Python kent drie soorten getallen:

- gehele getallen (type int)
  - gehele getallen zijn 0, 1, 2, 3, ... en -1, -2, -3, -4, …

- gebroken getallen (type float) : bevatten een punt of het exponent-symbool 'e'
  - Voorbeelden
    - **17.0**
    - **0.333333333333**
    - **8e3 (= 8*10^3 = 8000.0)**
      43.5e-2 (= 43.5 * 10^{-2} = 0.435)

- complexe getallen (type complex)
  - Complexe getallen zijn van de vorm a+bj
  - Voorbeeld:

```
import cmath
z = cmath.sqrt(-1) # √(-1)
print(z)
print(z*z)

Uitvoer:

1j
(-1+0j)
```

Gehele getallen kunnen willekeurig groot zijn. Er is geen bovengrens of ondergrens.
Je hoeft geen rekening te houden met overflow en de berekeningen zijn altijd exact.

Gehele getallen kunnen ook binair, octaal en hexadecimaal weergegeven worden:

Binaire weergave:	`bin(1023):`	**0b1111111111**
Octale weergave:	`oct(8):`	**0o10**
Hexadecimale weergave:	`hex(16):`	**0x10**
	`hex(255):`	**0xff**

Gebroken getallen hebben een bovengrens en een ondergrens.
Deze is als volgt te achterhalen:

```
import sys
print(sys.float_info.max)
print(sys.float_info.min)
```

```
Uitvoer:

1.7976931348623157e+308
2.2250738585072014e-308
```

Verder zijn bij gebroken getallen de uitkomsten van berekeningen niet exact.

Voorbeeld:

```
print(2**100)     # geheel getal
print(2.0**100)   # gebroken getal
```

```
Uitvoer:

126765060022822940149670320537 6  # 31 cijfers, exact
1.2676506002282294e+30             # 16 cijfers achter punt
```

Opmerkelijke voorbeelden:

```
print((0.8-0.7) - 0.1)
print(25*0.28 -7)
print(1E100 + 24 - 1E100)
```

```
Uitvoer:

8.326672684688674e-17
8.881784197001252e-16
0.0
```

<u>Conversie tussen getallen.</u>

int → float: expliciet: `float(3) == 3.0`
automatische conversie: `3/1 == 3.0`

float → int:

- afbreken: `int(4.7) == 4`
- afronden: `round(4.7)== 5`

int,float → complex: expliciet: `complex(3) ==(3 + 0j)`
automatische conversie: `(-1)**0.5 == (6.123233995736766e-17+1j)`

Rekenkundige expressies.

Rekenkundige expressies zijn opgebouwd uit variabelen, numerieke waarden en rekenkundige operatoren.
Het evalueren van een expressie gebeurt op basis van de prioriteit van de operatoren.

Een voorbeeld:

```
3+4*2-5//2
```

De operatoren '*' en '//' hebben de hoogste prioriteit.

Daarom worden eerst '`4*2`' en '`5//2`' berekend:

```
4*2   = 8
5//2  = 2
```

Vervolgens '`3+8`' berekend: `3+8` = 11

Tenslotte wordt '`11-2`' berekend: `11-2` = 9

Een overzicht van rekenkundige bewerkingen:

bewerking	**operator**
optellen	+
aftrekken	-
vermenigvuldigen	*
delen	/
geheel delen (rest na deling ontbreekt)	//
de rest na gehele deling	%
tot de macht verheffen	**

In plaats van '`a**b`' kan ook '`pow(a,b)`' worden gebruikt.
De volgende eigenschap geldt: `a % b = a - b*(a//b)`

De prioriteitsvolgorde:

Prioriteit van hoog naar laag
**
* / // %
+ –

Een uitgebreid overzicht is te vinden op de site
https://docs.python.org/3/reference/expressions.html

Als je het niet zeker weet kun je haakjes gebruiken.
De uitdrukking `3+4*2-5//2` kun je vervangen door `3+(4*2)-(5//2)` .
De evaluatie van ' `3+8-2` ' gebeurt van links naar rechts, want de operatoren '+' en '-' hebben dezelfde prioriteit.

De expressie '`2**3**4`' wordt als volgt berekend:

- eerst wordt `3**4` berekend. Uitkomst: **81**
- vervolgens wordt `2**81` berekend. Uitkomst: **2417851639229258349412352**

Er geldt dus: `2**3**4` = `2**(3**4)`
Machtsverheffen gebeurt van rechts naar links.

De volgorde van berekening is van belang.
Er geldt: `(2**3)**4` ≠ `2**(3**4)`
want:

`(2**3)**4` = `8**4` = **4096**
`2**(3**4)` = `2**81` = **2417851639229258349412352**

Nogmaals: voorkom vergissingen door haakjes te gebruiken.

Wiskundige functies.

De site https://docs.python.org/3/library/math.html beschrijft een aantal wiskundige functies.

Ze staan in de module 'math' .

Belangrijke functies zijn:

- exp(x) (e**x)
- log(x), log2(x), log10(x)
- sqrt(x)
- sin(x), cos(x), tan(x)
- asin9x), acos(x), atan(x)

Belangrijke constanten zijn:

- e
- pi (**π**)
- inf (∞ , oneindig)

6. Sequences.

Een sequence is een rij elementen.

De basis sequence types zijn: lists, tupels en ranges.
De text sequence type is 'string'.
Binary sequence types zijn 'bytes' en 'bytearray'.

Bij een sequence zijn de elementen op twee manieren genummerd:

- van links naar rechts : de elementen worden oplopend genummerd, te beginnen met 0
- van rechts naar links : de elementen worden aflopend genummerd, te beginnen met -1

Het element met index (of rangnummer) i van sequence s wordt weergegeven met de subscript notatie: s[i] .

Met sequences is het volgende mogelijk:

- concatenatie: twee sequences kunnen samengevoegd worden
- herhaling: dezelfde sequence kan een aantal keer herhaald worden
- slicing: een gedeelte van de sequence selecteren

Concatinatie en herhaling zijn bij range-objecten niet mogelijk.

Het één en ander zal met 'lijsten' verduidelijkt worden.

7. Lijsten.

Een lijst voldoet aan de eigenschappen:

- Een lijst wordt aangegeven met blokhaken.
- De elementen zijn (willekeurige) objecten. De elementen hoeven niet van hetzelfde type te zijn.
- Een lijst heeft geen vaste lengte. Er kan steeds een element aan de lijst toegevoegd worden.
- Er kunnen elementen uit de lijst gewijzigd of verwijderd worden.

De laatste twee eigenscchappen geven aan dat een lijst 'mutable' is.

Een voorbeeld :

```
a = [4, 'fiets', [5.5,8.4]]
```

Dit voorbeeld laat zien dat de elementen niet van hetzelfde type hoeven te zijn.

Nog een voorbeeld:

```
a = [1,99,2,98,3,97]
```

'a' is als volgt genummerd:

a:

1	99	2	98	3	97
0	1	2	3	4	5
-6	-5	-4	-3	-2	-1

Een demo:

```
a = [1,99,2,98,3,97]
print(a[-1])
print(a[-3])
```

Uitvoer:

```
97
98
```

Concatenatie.

M.b.v. de '+'-operator kunne lijsten worden samengevoegd.

Voorbeeld:

```
a1 = [1,2,3]
a2 = [101,102,103]
print(a1+a2)
```

Uitvoer:

```
[1, 2, 3, 101, 102, 103]
```

Een lijst herhalen.

M.b.v. de '*'-operator kunnen lijsten herhaald worden.

Voorbeeld:

```
a1 = [1,2,3]
a2 = [101,102,103]
print(4*a1)
print(a2*3)
print()
```

Uitvoer:

```
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
[101, 102, 103, 101, 102, 103, 101, 102, 103]
```

Een lijst met 25 elementen kan als volgt worden verkregen:

```
n = 25
b = [0]*n
print(b)
```

Uitvoer:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Het is ook mogelijk met de opdracht:

```
b = [0 for _ in range(25)]
```

Slicing: een gedeelte van een lijst selecteren.

Uit een lijst kan een gedeelte worden geselecteerd.

Voorbeeld:

Bij de lijst 'a' met

```
a = [1,99,2,98,3,97]
```

geldt:

```
a[1:3] == [99, 2]           # vanaf 1 tot aan 3
a[1:]  == [99,2,98,3,97]    # vanaf 1
a[:4]  == [1,99,2,98]       # tot aan 4
a[:]   == [1,99,2,98,3,97] # helemaal: van begin tot eind
a[1:5] == [99, 2, 98, 3]
```

Je kunt bij slicing ook een stapgrootte meegeven.

Enkele voorbeelden met stapgrootte 2:

```
a[1:5:2] == [99, 98]
a[1::2]  == [99, 98, 97]
a[:4:2]  == [1, 2]
a[::2]   == [1, 2, 3]
```

Een lijst kan omgekeerd worden door slicing met negatieve stapgrootte toe te passen:

```
a[4:1:-1]) == [3, 98, 2]
a[4::-1])  == [3, 98, 2, 99, 1]
a[::-1])   == [97, 3, 98, 2, 99, 1]  # helemaal omgekeerd
a[::-2])   == [97, 98, 99]           # stapgrootte 2
```

Elementen uit de lijst verwijderen.

M.b.v. slicing kunnen elementen uit een lijst verwijderd worden/

Voorbeeld:

```
a = [1,2,3,4,5,6,7,8]

del a[6]   ; print(a)   # verwijder het zevende element
del a[1:3] ; print(a)   # verwijder het 2e en het 3e element
del a[:2]  ; print(a)   # verwijder de eerste twee elementen
del a[1:]  ; print(a)   # verwijder alle elementen behalve
                        #                het eerste element
del a[:]   ; print(a)   # verwijder alle elementen
```

Uitvoer:

```
1, 2, 3, 4, 5, 6, 8]
1, 4, 5, 6, 8]
[5, 6, 8]
[5]
[]
```

Nog een voorbeeld:

```
a = list(range(1,11))   ; print(a)
del a[1::2]             ; print(a)
print()
```

Uitvoer:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 3, 5, 7, 9]
```

Tot nu toe zijn elementen met een gegeven index (positie binnen de lijst) verwijderd.
Het is ook mogelijk elementen met een gegeven waarde te verwijderen.

Een voorbeeld:

```
a = [1,2,3,2,1]        ; print(a)
a.remove(2)            ; print(a)  # alternatief:
                                   #    del a[b.index(2)]
```

Uitvoer:

```
[1, 2, 3, 2, 1]
[1, 3, 2, 1]
```

Alleen element '2' met de kleinste index wordt verwijderd.

Het verwijderen van alle elementen met waarde '2' gaat als volgt:

```
a = [1,2,3,2,1,2,3,2,1] ; print(a)
while 2 in a:
      a.remove(2)       ; print(a)
```

Uitvoer:

```
[1, 2, 3, 2, 1, 2, 3, 2, 1]
[1, 3, 2, 1, 2, 3, 2, 1]
[1, 3, 1, 2, 3, 2, 1]
[1, 3, 1, 3, 2, 1]
[1, 3, 1, 3, 1]
```

Elementen uit de lijst wijzigen.

Met behulp van de toekenningsopdracht kan een element gewijzigd worden:

Een voorbeeld:

```
a = [1,2,3,4,5,6,7,8] ; print(a)
a[2] = 99             ; print(a)
```

Uitvoer:

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 99, 4, 5, 6, 7, 8]
```

Met behulp van slicing kan een groep elementen vervangen worden door een andere groep.

Een voorbeeld:

```
a = [1,2,3,4,5,6,7,8] ; print(a)
a[1:3] = [88]         ; print(a) # a[1:3] = 88 niet toegestaan
a[1:3] = [77,177,277] ; print(a)
a[2:5]= []            ; print(a)
```

Uitvoer:

```
[1, 2, 3, 4, 5, 6, 7, 8]
[1, 88, 4, 5, 6, 7, 8]
[1, 77, 177, 277, 5, 6, 7, 8]
[1, 77, 6, 7, 8]
```

Elementen aan een lijst toevoegen.

Het toevoegen van één element aan de lijst gaat als volgt:

Voorbeeld:

```
a  =  [1, 1, 1, 1]   ; print(a)
a.insert(0,6)        ; print(a)  # voor de lijst plaatsen
a.insert(3,7)        ; print(a)  # toevoegen voor het element
                                      # met positie 3
a.insert(len(a),8)   ; print(a)  # achter de lijst plaatsen
a.append(8)          ; print(a)  # methode 2
a = a + [8]          ; print(a)  # methode 3
```

Uitvoer:

```
[1, 1, 1, 1]
[6, 1, 1, 1, 1]
[6, 1, 1, 7, 1, 1]
[6, 1, 1, 7, 1, 1, 8]
[6, 1, 1, 7, 1, 1, 8, 8]
[6, 1, 1, 7, 1, 1, 8, 8, 8]
```

Met behulp van slicing kunnen meerdere elementen toegevoegd worden.

Voorbeeld:

```
a  =  [1, 1, 1, 1]   ; print(a)
a[:0] = [6,66]       ; print(a)  # voor de lijst plaatsen
a[4:4] = [7,77]      ; print(a)  # toevoegen voor het element
                                      # met positie 3
a[len(a):] = [8,88]  ; print(a)  # achter de lijst plaatsen
a.extend([8,88])     ; print(a)  # methode 2
a = a + [8,88]       ; print(a)  # methode 3
```

Uitvoer:

```
[1, 1, 1, 1]
[6, 66, 1, 1, 1, 1]
[6, 66, 1, 1, 7, 77, 1, 1]
[6, 66, 1, 1, 7, 77, 1, 1, 8, 88]
[6, 66, 1, 1, 7, 77, 1, 1, 8, 88, 8, 88]
[6, 66, 1, 1, 7, 77, 1, 1, 8, 88, 8, 88, 8, 88]
```

Als len(a) <= 100 dan is de opdracht `'a[100] = 999'` niet toegestaan.

Wel zijn de volgende opdrachten toegestaan:

```
a[100:] = [999]
a.insert(100,999)
```

In beide gevallen wordt '999' toegevoegd aan de lijst.

Ingebouwde lijst-functies:

Python bevat een aantal ingebouwde functies.
Zie https://docs.python.org/3/library/functions.html .

Veel gebruikte functies, die toegepast kunnen worden op lijsten, zijn:

- len() : de lengte van een 'sequence' of een 'collection'
- min() : het kleinste element van een 'iterable'
- max() : het grootste element van een 'iterable'
- sum() : de som van de elementen van een 'iterable' met number-items

Het gaat hier om generiek toepasbare functies.

De functies 'min' en 'max' kunnen alleen toegepast worden als de elementen onderling vergelijkbaar zijn.
De functie 'sum' kan alleen toegepast worden op getallen.

Voorbeeld.

```
print(min([6,3,5,9]))
print(min('noot','aap','mies'))
print(min('102','13'))
```

Uitvoer:

```
3
aap
102
```

Ingebouwde lijst-methoden.

Een lijst heeft de volgende methoden:
(zie https://docs.python.org/3/tutorial/datastructures.html
https://docs.python.org/3/library/stdtypes.html#list)

- append(x) : voeg 'x' aan de lijst toe
- extend(b) : breidt de lijst uit met 'b'
- insert(i,x) : voeg 'x' op posite 'i' toe
- remove(x) : verwijder uit de lijst het eerste element met waarde 'x'
- clear() : maak de lijst leeg
- copy() : maak een (shallow) copy van de lijst
- sort() : sorteer de lijst
  - sort(reverse=True) : sorteer van groot naar klein
- reverse() : keer de lijst om
- pop() : bereken het laatste element van a en verwijder het element
  - pop(i) : bereken a[i] en verwijder a[i]
- count(i) : bereken het aantal elementen van de lijst met waarde 'i'
- index(i) : bepaal de kleinste k met a[k] == i

Voorbeeld:

```
a = [1,-2,3,-4,1,-2,3,-4] ; print(a)
a.sort()                  ; print(a)
a.sort(reverse = True)    ; print(a)
a.reverse()               ; print(a)
i = a.pop()           ; print('na pop-aanroep:    i:',i,'a:',a)
i = a.pop(2)          ; print('na pop(2)-aanroep: i:',i,'a:',a)
print(a.count(-4))
print(a.index(1))
```

Uitvoer:

```
[1, -2, 3, -4, 1, -2, 3, -4]
[-4, -4, -2, -2, 1, 1, 3, 3]
[3, 3, 1, 1, -2, -2, -4, -4]
[-4, -4, -2, -2, 1, 1, 3, 3]
na pop-aanroep: i: 3 a: [-4, -4, -2, -2, 1, 1, 3]
na pop(2)-aanroep: i: -2 a: [-4, -4, -2, 1, 1, 3]
2
3
```

De meeste methoden wijzigen de lijst.

Een lijst is een mutable sequence.
Strings, tupels en ranges zijn immutable sequences.

8. Tupels.

Tupels zijn immutable lijsten: je niet ze kunt wijzigen.
Bij lijsten worden de elementen omsloten door de blokhaken '[' en ']' .
Bij tupels worden de elementen omsloten door de ronde haken '(' en ')' .

Voorbeelden van tupels zijn:

```
(1,3,5)
()      # lege tupel
(3,)    # tupel met één element
```

Bij een lijst met één element staat na het element een komma.

Ronde haken worden ook gebruikt bij expressies. Hiermee kan de evaluatievolgorde van een expressie-berekening worden vastgelegd.
Verder geldt voor een expressie 'e': e == (e) == ((e)) == (((e))) . M.a.w. om een expressie kunnen haakjes geplaatst, zonder dat de waarde verandert.

Er geldt daarom:

- (3) == 3 en type((3)) ==
- (3,) == "tupel met element 3" en type((3,)) ==

Je kunt van een tupel omzetten naar een lijst en andersom:

Demo:

```
t = (1,3,5) ; print(t)
a = list(t) ; print(a) # tuple → list
a.reverse() ; print(a) # t.reverse() is niet toegestaan
t = tuple(a); print(t) # list → tuple
t = (1,3,5) ; print(t)
t = t[::-1] ; print(t) # t wijst nu naar een ander tupel-object
```

Uitvoer:

```
(1, 3, 5)
[1, 3, 5]
[5, 3, 1]
(5, 3, 1)
(1, 3, 5)
(5, 3, 1)
```

Bij tupels kunnen de individuele elementen ook verkregen met de index-notatie.

Bij lijsten zijn een aantal functies en methoden behandeld.
De functies 'len', 'min', 'max' en 'sum' kunnen ook op tupels toegepast worden.
De methoden 'count' en 'index' kunnen ook op tupels toegepast worden.

9. Range.

Een range representeert een eindige rekenkundige rij. Bij een rekenkundige rij is het verschil tussen twee opeenvolgende termen constant.

Voorbeelden van eindige rekenkundige rijen zijn:

- 0 1 2 3 4 5 6 7 8 9
- 3 4 5 6 7 8 9
- -4 -3 -2 -1 0 1 2 3 4 5 6
- 0 2 4 6 8
- 10 9 8 7 6 5 4
- 7 4 1 -2 -5 -8

Een range-object is immutable en bestaat uit drie gegevens:

- begin-waarde (standaard 0)
- eind-waarde
- stapgrootte (standaard 1)

De elementen van de rij, die gerepresenteerd wordt, worden niet opgeslagen. Het geheugengebruik is dus beperkt.

Op basis van een range-object kan een lijst van de gepresenteerde elementen worden gemaakt.

Een demo:

```
r = range(10)        ; print(list(r))
r = range(3,10)      ; print(list(r))
r = range(-4,7)      ; print(list(r))

r = range(0,10,2)    ; print(list(r))
r = range(10,3,-1)   ; print(list(r))
r = range(7,-10,-3)  ; print(list(r))
```

Uitvoer:

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5, 6, 7, 8, 9]
[-4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6]
[0, 2, 4, 6, 8]
[10, 9, 8, 7, 6, 5, 4]
[7, 4, 1, -2, -5, -8]
```

Een range wordt vaak gebruikt in een for-loop.

Een demo:

```
for e in range(1,11):
   print(e*e, end = ' ')
print()
a = [e*e for e in range(1,11)] ; print(a)
```

Uitvoer:

```
1 4 9 16 25 36 49 64 81 100
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Ook bij ranges kunnen de individuele elementen ook verkregen met de index-notatie.

De functies 'len', 'min', 'max' en 'sum' kunnen ook op ranges toegepast worden.
De methoden 'count' en 'index' kunnen ook op ranges toegepast worden.

10. Strings.

Een string is een speciale lijst en voldoet aan de eigenschappen:

- De elementen van een string worden omsloten met '(enkele quote) of "(dubbele quote).
- De elementen zijn unicode characters.
- Een string is immutable
- Een string kan uit meerdere regels bestaan De string is dan omsloten door drie quotes aan het begin en aan het eind.
- In een string is '\' een escape-teken.
- Een 'raw string' wordt voorafgegaan door 'r' of 'R' en wordt letterlijk afgedrukt.
- Een character is een string met lengte 1

Een string is een stuk tekst.
Strings worden omsloten door '(enkele quote) of "(dubbele quote).

Een uitgebreid voorbeeld:

```
s = 'string'           ; print(s)  # omsloten door enkele quoten
s = "nog een string"   ; print(s)  # omsloten door dubbele quoten
ch = 'a'               ; print(ch) # een karakter
s = ''                 ; print('len(s):',len(s)) # de lege string
s = "een string"[4:]   ; print(s)  # slicing
ch = 'string'[-1]      ; print(ch) # laatste karakter van string
s = "aa'bb"            ; print(s)  # bevat enkele quote
s = 'aa"bb'            ; print(s)  # bevat dubbele quote
s = 'aa\'bb\"cc'       ; print(s)  # bevat enkele en dubbele quote
tekst = """regel 1
regel 2
regel 3"""             ; print(tekst) # meerdere regels
r = r'a\tb'            ; print(r)     # letterlijke tekst
s = 'a\tb'             ; print(s)     # '\t' wordt niet letterlijk
                                      # afgedrukt
print(r'a\tb' == 'a\\tb') # r'a\tb' is hetzelfde als 'a\\tb'
```

Uitvoer:

```
string
nog een string
a
len(s): 0
string
g
aa'bb
aa"bb
aa'bb"cc
regel 1
regel 2
regel 3
a\tb
a    b
True
```

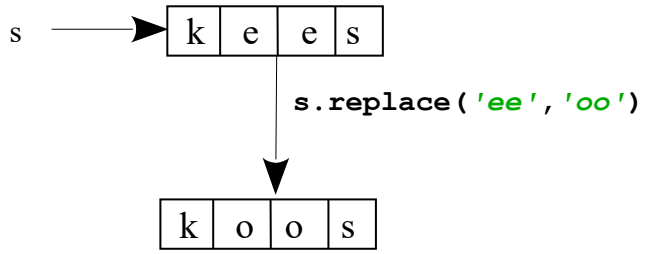
Het escape-teken '\' wordt ook gebruikt voor speciale tekens:

\n (newline)
\t (tab)
\r (return)
\\ (backslash)

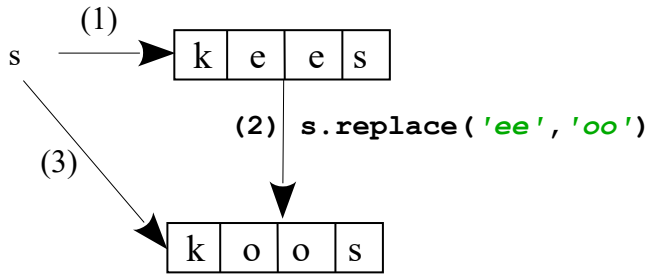
Een string is immutable: je kunt het niet wijzigen.
Een string-variabele kun je wel een andere string-waarde geven.

Voorbeeld.

```
s = 'kees'
s.replace('ee','oo')       # s blijft ongewijzigd
```




```
s = 'kees'
s = s.replace('ee','oo')       # s wijst nu naar 'koos'
```



De volledige code:

```
s = 'kees'                ; print(s) ; print('id:',id(s))
s.replace('ee','oo')      ; print(s) # s blijft ongewijzigd
s = s.replace('ee','oo') ; print(s) ; print('id:',id(s))
                            # s wijst nu naar een ander object
```

Uitvoer:

```
kees
id: 20486496
kees
koos
id: 19876704
```

Nog een voorbeeld.

```
s = "lower case"
print(s.upper()) ; print('s:',s)
s = s.upper()    ; print('s:',s)
print()
```

Uitvoer:

```
LOWER CASE
s: lower case
s: LOWER CASE
```

Een aantal mogelijkheden bij lijsten zijn ook toegestaan bij strings:

- strings combineren met de '+'-operator
- strings herhalen met '*'-operator
- slicing

Demo:

```
s = "string1" + "string2" ; print(s)
s = "herhaling"*3         ; print(s)
s = "deelstring"[:4]      ; print(s)
```

Uitvoer:

```
string1string2
herhalingherhalingherhaling
deel
```

Ook heeft de string-klasse de methoden:

- count(i) : bereken het aantal elementen van de lijst met waarde 'i'
- index(i) : bepaal de kleinste k met a[k] == i

Demo:

```
i = "hoeveel e's in deze niet veelzeggende lange zin".count('e')  ; print(i)
i = "hoeveel en's in deze niet veelzeggende lange zin".count('en'); print(i)
i = "de eerste 'e' in deze zin".index('e')                        ; print(i)
i = "de eerste 'en' in deze veelzeggende zin".index('en')         ; print(i)
i = "de tweede 'en' in deze veelzeggende zin"[12:].index('en')    ; print(i)
```

Uitvoer:

```
13
2
1
11
19
```

Verder is mogelijk:

- een deelstring zoeken
- een string opsplitsen in een (mutable) lijst karakters of een lijst met woorden

Een deelstring zoeken.

Een deelstring zoeken kan met de methoden 'index' en 'find'.
Als de deelstring niet wordt gevonden wordt bij 'index' een exception gegenereerd. Bij 'find' wordt de waarde '-1' terug gegeven.

Het nagaan, of een string 's' een gegeven deelstring 'ds' bevat kan op nog meer manieren:

- Test: **s.count(ds) > 0**
- Test: **ds in s**

Een demo:

```
s = 'zoek een woord in een regel'
try:
    i = s.index('woord') ; print(i)
except:
    print('not found')
i = s.find('woord')      ; print(i)
i = s.count('woord')     ; print(i)
b = 'woord' in s         ; print(b)
try:
    i = s.index('test')  ; print(i)
except:
    print('not found')
i = s.find('test')       ; print(i)
i = s.count('test')      ; print(i)
b = 'test' in s          ; print(b)
```

Uitvoer:

```
9
9
1
True
not found
-1
0
False
```

Een string opsplitsen.

Bij het opsplitsen van een string spelen whitespace-characters een rol.

Je kunt de whitespace-characters als volgt opvragen:

```
import string
print('whitespace-characters:', string.whitespace.encode())
```

Uitvoer:

```
whitespace-characters: b' \t\n\r\x0b\x0c'
```

Toelichting:
' ' : spatie
\t : tab
\n : newline linefeed
\r : (carriage) return
\x0b : \v : vertical tab (verouderd)
\xoc : \f : formfeed (verouderd)

Een string omzetten naar een lijst karakters gaat als volgt:

```
s = 'string'
a = list(s); print(a)
# alternatief: a = [e for e in s]
```

Uitvoer:

```
['s', 't', 'r', 'i', 'n', 'g']
```

Een zin opsplitsen in woorden gaat als volgt:

```
s = 'een string  met \t white \t\n characters'
a = s.split() ; print(a)
```

Uitvoer:

```
['een', 'string', 'met', 'white', 'characters']
```

Uit de zin worden white characters weggehaald. De overgebleven woorden worden in een lijst geplaatst.

Je kunt ook splitsen op basis van een separator.

Een voorbeeld:

```
s = 'eenvbstringvbmetvbeenvbseparator'
a = s.split('vb') ; print(a)
```

Uitvoer:

```
['een', 'string', 'met', 'een', 'separator']
```

Door een separator op te geven kun je ook lege strings in de lijst krijgen:

Een voorbeeld:

```
s = 'a b  c   d'
a = s.split(' ') ; print(a)
```

Uitvoer:

```
['a', 'b', '', 'c', '', '', 'd']
```

Met de seperator '\n' kan tekst opgesplitst worden in regels.

Een voorbeeld:

```
s = 'regel 1\n regel 2\n regel 3'
a = s.split('\n')  ; print(a)
# alternatief: a = s.splitlines() ; print(a)
```

Uitvoer:

```
['regel 1', ' regel 2', ' regel 3']
```

Een lijst samenvoegen tot een string.

Het samenvoegen van de lijst

```
['een', 'string', 'met', 'een', 'separator']
```

tot de string

```
'eenvbstringvbmetvbeenvbseparator'
```

kan met de opdracht

```
s = 'vb'.join(['een', 'string', 'met', 'een', 'separator'])
```

De lijst

```
['s', 't', 'r', 'i', 'n', 'g']
```

samenvoegen tot de string

```
'string'
```

gaat met de opdracht

```
s = ''.join(['s', 't', 'r', 'i', 'n', 'g'])
```

Bij deze opdracht is als separator de lege string '' gebruikt: er staat niet tussen de letters die samengevoegd worden.

Characters.

In Python 3 is een string opgebouwd uit unicode characters.

Unicode is een uitbreiding van ASCII.
Informatie over unicode is te vinden op de sites:

- https://en.wikipedia.org/wiki/List_of_Unicode_characters
- http://unicode.org/

ASCII koppelt karakters aan een 7-bits getal. Het beperkt zich hoofdzakelijk tot Latijnse letters, getallen en leestekens. Een ASCII-karakter kan gerepresenteerd worden door één byte.

Unicode koppelt meer dan 128.000 karakters aan een getal. Omdat het aantal byte-waarden 256 is, zijn meer bytes nodig voor de representatie.
Voor het omzetten van een unicode character naar bytes bestaan een aantal encoding-technieken. Een veelgebruikte encoding-techniek is 'utf-8'.
Bij Windows wordt vaak 'cp1252' gebruikt. Bij dosbox wordt vaak 'cp850' gebruikt.

Een voorbeeld van unicode characters zijn de griekse letters.
Ze zijn te vinden op de sites:

- https://en.wikipedia.org/wiki/List_of_Unicode_characters#Greek_and_Coptic
- http://symbolcodes.tlt.psu.edu/bylanguage/greekchart.html

Het afdrukken van 'π' gaat als volgt:

```
print('π')  # niet via het toetsenbord kun je π intikken
```

Op het toetsenbord ontbreekt echter 'π' . Je kunt deze code dus niet intikken.
Als de opdracht `'print('π')'` wordt opgeslagen in een source-file , wordt de file (in windows) opgeslagen in 'utf-8'-formaat. In 'cp1252' kan 'π' niet opgeslagen worden.

De unicode-waarde van 'π' is 0x03C0 .
Het afdrukken van 'π' gaat ook als volgt:

```
print('\u03c0')  # wel via het toetsenbord mogelijk
```

Het 'utf-8'-formaat van 'π' wordt kan als volgt bepaald worden:

```
print('π'.encode('UTF-8')) # alternatief: print('π'.encode())
                           # defaultencoding: utf-8
```

Uitvoer:

```
b'\xcf\x80'
```

'π' wordt dus gerepresenteerd door twee bytes met de hexadecimale waarden 0xcf en 0x80

De encoding-instellingen van Python kun je als volgt opvragen:

```
import locale
print(locale.getpreferredencoding())
import sys
print(sys.getdefaultencoding())
```

Uitvoer:

```
cp1252
utf-8
```

11. Bytes en byte-array

Strings zijn opgebouwd uit uncicode-characters.
Gegevens worden opgeslagen in een file als een rij bytes. Ook bij het versturen van gegevens worden deze omgezet naar bytes.

Python kent twee sequence-datastructuren voor het werken met bytes:

- bytes (immutable)
- bytearray (mutable)

Een bytes-object wordt als volgt gedeclareerd:

```
b = b'abc'
```

Dit object wordt ook verkregen door de opdracht:

```
b = 'abc'.encode() # default encoding: utf-8
```

Een uitgebreide demo:

```
b = b'abc'                                ; print(b)
b = 'abc'.encode()                        ; print(b)
b = 'abc'.encode('utf-16')                ; print(b)
b = bytes(10)                             ; print(b)
b = bytes([10])                           ; print(b)
b = bytes([97,98,99])                     ; print(b)
b = bytes(range(ord('a'),ord('a')+26))    ; print(b)
```

Uitvoer:

```
b'abc'
b'abc'
b'\xff\xfea\x00b\x00c\x00'
b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00'   # 10 0-bytes
b'\n'                                         # ord('\n') = 10
b'abc'
b'abcdefghijklmnopqrstuvwxyz'
```

Een bytearray-abject is de mutable variant van een byte-object.

Een demo:

```
b = b'abc'
ba = bytearray(b)           ; print(ba)
ba = bytearray(10)          ; print(ba)
ba = bytearray([97,98,99])  ; print(ba)
print(ba.decode())
```

Uitvoer:

```
bytearray(b'abc')
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
bytearray(b'abc')
abc
```

12. Eenvoudige I/O

Uitvoer naar het scherm gebeurt met de print-opdracht.
Invoer vanaf het toetsenbord gebeurt met de input-opdracht.

De print-opdracht.

Standaard geldt het volgende:
- bij een print-opdracht worden de gegevens gescheiden door een spatie.
- na een print-opdracht wordt begonnen op een nieuwe regel.

We laten in een tabel zien hoe we hiervan kunnen afwijken

Opdracht	Uitvoer	Toelichting
`print(27,39,68)`	**27 39 68**	
`print(27,39,68,sep='#')`	**27#39#68**	sep: separator
`print(27,39,68,sep = '%&$')`	**27%&$39%&$68**	
`print(27)` `print(39)`	**27** **39**	
`print(27,end = ' ')` `print(39)`	**27 39**	
`print(27,end = '')` `print(39)`	**2739**	
`print(27,39,sep='@',end='#')`	**27@39#99**	

De input-opdracht.

Gegevens van toetsenbord worden ingelezen met de functie 'input'.

Voorbeeld

```
naam = input("geef naam:")
print('naam:', naam)
```

Als een getal wordt ingelezen, moet een conversie van 'string' naar 'int' worden gemaakt.

Voorbeeld

```
s = input("geef getal:")
print('s:', s)
getal = int(s) # str -> int
print('getal:', getal)
```

Let op: voor getallen geldt: **21 < 101**,
voor strings geldt: **'101' < '21'**

13. Beweringen.

Als een keuze-opdracht of een herhalingsopdracht wordt uitgevoerd, wordt nagegaan of een bewering waar is.

Bij Python wordt 'waar' aangegeven met: **True**
en 'niet waar' aangegeven met **False**

Bij beweringen (ook wel logische expressies genoemd) worden relationele en logische operatoren gebruikt.

Relationele operatoren.

De relationele operatoren staan in de volgende tabel:

operator	**betekenis**
<	kleiner dan
<=	kleiner of gelijk aan
==	is gelijk aan
>=	groter of gelijk aan
>	groter dan
!=	is ongelijk aan

Logische operatoren.

De logische operatoren staan in de volgende tabel:

operator	**betekenis**
and	en
or	of
not	niet

In Python zijn ook beweringen van de volgende vorm toegestaan:

- `i < j < 10 # verkorte schrijfwijze van I < j and j < 10`
- `i < j < 10 < j*j <= 100 < j*j*j`

Lazy evaluation.

Het verwerken van logische expressies gebeurt op basis van "lazy evaluation".

Neem als voorbeeld de bewering:

```
(0 <= i < len(a)) and (a[i] > 100)
```

Eerst wordt nagegaan of "`0 <= i < len(a)`" waar is.

Als "`0 <= i < len(a)`" niet waar is dan weten we dat de hele bewering niet waar is.
Er wordt dan niet meer nagegaan of "`a[i] > 100`" waar is.

Bekijk de volgende bewering::

```
(i > 100) or (i < 0)
```

Eerst wordt nagegaan of "`i > 100`" waar is.

Als "`i > 100`" waar is, dan weten we dat de hele bewering waar is.
Er wordt dan niet meer nagegaan of "`i < 0`" waar is.

Kortom:

"**b1 and b2**" : eerst wordt b1geëvalueerd
als b1 niet waar is, wordt b2 **niet** geëvalueerd.

"**b1 or b2**" : eerst wordt b1geëvalueerd
als b1 waar is, wordt b2 **niet** geëvalueerd

Het testen van objecten.

Vaak wordt getest of een string niet leeg is, of een getal ongelijk aan nul is, of een object ongelijk aan 'None' is, enz.
Daarom worden bij het testen de waarden 0, 0.0 , '', [], () , 'None' gelijk gesteld aan 'False'.
De waarden ongelijk aan 0, 0.0 , '', [], () , 'None' hebben bij het testen de waarde 'True' .

'`if i != 0`' kan daarom vervangen worden door '`if i`'.

De tekst wordt hierdoor korter, maar helaas ook minder leesbaar.

14. Opdrachten.

De volgende opdrachten worden behandeld:

- de toekenningsopdracht
- de keuzeopdracht
- de herhalingsopdracht

De toekenningsopdracht.

De toekenningsopdracht kent aan een gegeven een waarde toe.

Voorbeelden van toekenningsopdrachten zijn:

```
x = 3
z = 8
y = (x + 1)*(z-4)    # x en z moeten bestaan
i = 11
i = i + 1             # verhoog i met 1 # alternatief: i += 1
                      # niet toegestaan: i++ of ++i
a = [i,2*i,3*i]
v = a[x-1]
```

Eerst wordt het gedeelte achter het '='-teken berekend. Vervolgens wordt het resultaat aan het object voor het '='-teken toegekend.

Er is een belangrijk verschil tussen "`x == 3`" en "`x = 3`".

"`x == 3`" betekent: "x is gelijk aan 3" . Dit is een bewering. (wel of niet waar)

"`x = 3`" betekent : "x wordt 3" of "geef x de waarde 3" . Dit is een opdracht.

Verkorte schrijfwijzen.

De volgende tabel laat zien hoe je opdrachten korter kunt schrijven.

Statement(s)	Verkorte schrijfwijze
`a = 3` `b = 3`	`a = b = 3`
`a = 3` `b = 4`	`a,b = 3,4`
`i = i+ 5`	`i += 5`
`j = j - 4`	`j -= 4`
`k = k*2`	`k *= 2`
`d = d/3`	`d /= 3`

Samenvoegen en uitpakken.

Meerdere expressies kunnen worden samengevoegd tot één tupel. (pack)

Voorbeeld:

```
t = (i+5,j-4,k*2,d//3
```

t is een tupel met vier elementen

Een tupel kan weer opgesplitst worden (unpack)

```
i,j,k,d = t
```

Een demo:

```
i,j,k,d = 10,20,30,40  ; print(i,j,k,d)
t = (i+5,j-4,k*2,d//3) ; print(t)         # pack
i,j,k,d = t            ; print(i,j,k,d)   # unpack

c1,c2,c3 = 'xyz'       ; print(c1,c2,c3) # unpack string
```

Uitvoer:

```
10 20 30 40
(15, 16, 60, 13)
15 16 60 13
x y z
```

De toekenningsopdracht bij lijsten.

Tupels, strings, ranges, bytes kunnen niet gewijzigd worden.

Lijsten kunnen wel gewijzigd worden.
Lijsten zijn verraderlijk: een lijst kan ook via een andere variabele gewijzigd worden.

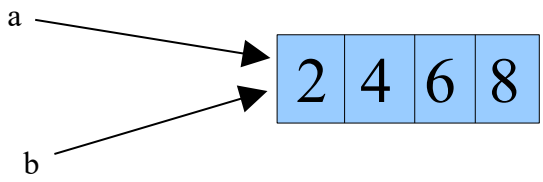
Bekijk het volgende voorbeeld:

```
a = [2,4,6,8] ; print(a)
b = a
b[2] = 22     ; print(a)
print(id(a),id(b))
```

Uitvoer:

```
[2, 4, 6, 8]
[2, 4, 22, 8]
35137336 35137336
```

De lijst 'a' is veranderd via 'b'. Dit komt omdat 'a' en 'b' naar hetzelfde object verwijzen.

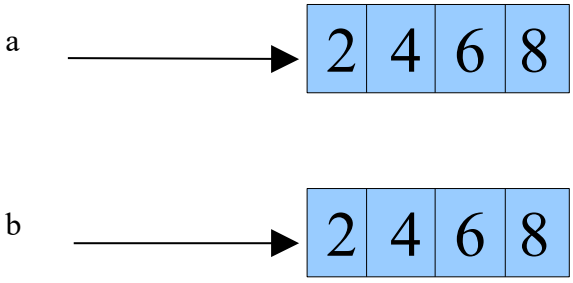


Dit kun je oplossen door "b = a" te vervangen door "b = list(a) "
Dan wordt een kopie van a gemaakt en aan b toegekend.

```
a = [2,4,6,8] ; print(a)
b = list(a)   # alternatief: b = a.copy()
b[2] = 22     ; print(a)
print(id(a),id(b))
```

Uitvoer:

```
[2, 4, 6, 8]
[2, 4, 6, 8]
35137456 35175920
```



Dit gaat goed zolang de elementen van de lijst immutable zijn.
Als een element mutable is, dan moet een "diepe kopie" gemaakt worden.

Besturingsopdrachten.

Besturingsopdrachten geven aan welke opdrachten uitgevoerd moeten. Ook wordt vastgelegd in welke volgorde dit moet gebeuren.

Python kent de volgende besturingsopdrachten:

- keuzeopdrachten
  - if statement
  - if else statement
- herhalingsopdrachten
  - for statement
  - while statement
- sprongopdrachten
  - break statement
  - continue statement

Het if-statement.

In Python heeft het if-statement de volgende opbouw:

```
if test:     # na test wordt een ':' geplaatst
   statement # eerst inspringen, dan de statement
```

Voorbeelden:

```
s = input('geef de waarde van i: ')
i = int(s)
if i%3 == 0:
   print(i, 'is deelbaar door 3')

s = input('geef saldo: ')
saldo = int(s)
if saldo < 0:
   print('waarschuwing!!!')
   print('het saldo is negatief')
```

Het if-else statement.

In Python heeft het if-else-statement de volgende opbouw:

```
if test:
   statement1
else:
   statement2
```

Voorbeelden:

```
s = input('geef de waarde van i: ')
i = int(s)
if i%3 == 0:
   print(i, 'is deelbaar door 3')
else:
   print(i, 'is niet deelbaar door 3')

s = input('geef saldo: ')
saldo = int(s)
if saldo < 0:
   print('waarschuwing!!!')
   print('het saldo is negatief')
else:
   print('gefeliciteerd!')
   print('het saldo is positief')
```

Het geneste if-else statement.

Bij een geneste if-else opdracht wordt een if-else opdracht binnen een andere if-else opdracht geplaatst.

Voorbeeld:

```
x = int(input('geef de waarde van x: '))
y = int(input('geef de waarde van y: '))

if x<y:
   print('x is kleiner dan y')
else:
   if x>y
      print('x is groter dan y')
   else:
      print('x is gelijk aan y')
```

In Python kan dit ook anders:

```
if x<y:
   print('x is kleiner dan y')
elif x>y:
   print('x is groter dan y')
else:
   print('x is gelijk aan y')
```

De laatste vorm heeft twee voordelen:

- het vraagt minder tikwerk
- er wordt minder diep ingesprongen.

Het for-statement.

Het for-statement wordt o.a. gebruikt voor het doorlopen van een range, een lijst, een tupel of een string.

We geven enkele voorbeelden:

Voorbeeld 1: druk 11^2, 12^2, …, 20^2 af

Dit gaat als volgt:

```
for i in range(11,21): # i heeft de waarden 11,12,...,20
   print(i*i)
```

Na iedere print-opdracht begint de uitvoer bij een nieuwe regel
Dit kan voorkomen worden door bij de print-opdracht een andere 'end'-waarde mee te geven.

```
for i in range(11,21):
   print(i*i,end=' '
```

De uitvoer is nu:

121 144 169 196 225 256 289 324 361 400

Voorbeeld 2: plaats 11^2, 12^2, …, 20^2 in een lijst.

Oplossing:

```
b = []              # b is een lege lijst
for i in range(11,21):
   b.append(i*i)
print('b =' ,b)
```

Uitvoer:

```
[121, 144, 169, 196, 225, 256, 289, 324, 361, 400]
```

Bij Python kan een lijst ook verkregen worden op list-comprehension:

```
[ <expressie> for i in range( .. )]
```

mogelijk.
Hierbij doorloopt i de range-waarden en wordt de bijbehorende expressie-waarde aan de lijst toegevoegd.

Het probleem kan hiermee als volgt worden opgelost:

```
b = [i*i for i in range(11,21)]
print('b =' ,b)
```

Voorbeeld 3: plaats van de kwadraten 11^2, 12^2, …, 20^2 , alleen de kwadraten, die niet deelbaar zijn door 3, in een lijst.

Oplossing:

```
a = []
for i in range(11,21):
   k = i*i
   if k%3 != 0:
      a.append(k)
print('a =' ,a)
```

compact:

```
a = [i*i for i in range(11,21) if i%3 != 0]
print('a =' ,a)
```

Het for-statement kan i.p.v. ranges ook toegepast worden op strings, lijsten en tupels.

Voorbeeld 4: druk de karakters van een string afzonderlijk af.

Dit gaat als volgt:

```
for ch in 'string':
    print(ch, end = ' ')
```

Uitvoer:

```
s t r i n g
```

Voorbeeld 5: plaats 2^n met n in [41,73,52] in een lijst.

Oplossing:

```
a = [41,73,52]
b = [2**n for n in a]
print('b =' ,b)
```

Uitvoer:

```
b = [2199023255552, 9444732965739290427392, 4503599627370496]
```

Voorbeeld 6: Van een lijst van strings wordt een frequentie-lijst gemaakt. De frequentie-lijst geeft aan hoe vaak de letter 'e' voorkomt in de string.

Oplossing:

```
a = ['hoeveel','genezen','bereveel']
u = [list(i).count('e') for i in a]
print('u =' ,u)
```

Uitvoer:

```
u = [3, 3, 4]
```

Het while-statement.

For-statements kunnen alleen gebruikt worden als bekend is welke lijst van elementen doorlopen moet worden.

Het while-statement wordt gebruikt als een dergelijke lijst ontbreekt.

Het while-statement heeft de volgende structuur:

```
while test:
   statement
```

Voorbeeld:

Een rij getallen wordt ingelezen. De rij wordt afgesloten met een 0. Na afloop wordt de som van de ingelezen getallen getoond.

Oplossing:

```
som = 0
getal = int(input('voer een getal in: '))
while getal != 0:
   som += getal
   getal = int(input('voer een getal in: '))
print('de som van de ingelezen getallen is:', som)
```

15. Command-line argumenten.

De argumenten, die 'command-line' meegegeven worden, zijn opgeslagen in 'sys.argv'.

Het volgende voorbeeld telt de meegegeven argumenten bij elkaar op.

telop.py:

```
import sys

a = sys.argv
print(a)
a = [int(i) for i in a[1:]]
print(a)
print(sum(a))
```

Geef de volgende opdracht:

python telop.py 1 2 11 567

Uitvoer:

```
['telop.py', '1', '2', '11', '567']
[1, 2, 11, 567]
581
```

16. Foutafhandeling.

De volgende code leidt tot een foutmelding:

```
i = 0
x = 1/i
```

De foutmelding is: `ZeroDivisionError: division by zero`
Het programma wordt bovendien afgebroken.

We kunnen op verschillende manieren ons beschermen tegen fouten.

Methode 1: ga vooraf na of i != 0

```
i = 0
if i != 0:
   x = 1/i
```

De methode is gebaseerd op het LBYL-principe: Look Before You Leap
De methode heeft als nadeel, dat vooraf een test wordt gedaan. Dit is extra werk.

Methode 2 ; gebruik een "try... except ..."-constructie.

```
i = 0
try:
   x = 1/i
except:
   print('je kunt niet door nul delen')
```

Deze methode is gebaseerd op het EAFP-principe: Easier to Ask Forgivenes than Permission

Je kunt het type exception als volgt achterhalen:

```
import sys

i = 0
try:
   x = 1/i
except:
   print(sys.exc_info())
```

Uitvoer:

```
(<class 'ZeroDivisionError'>,
ZeroDivisionError('division by zero',),
<traceback object at 0x01C6DDF0>)
```

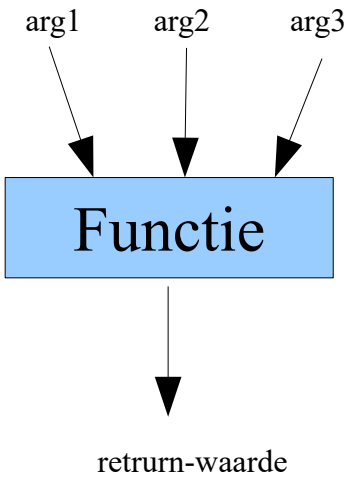
Op basis hiervan kun je het volgende programma schrijven:

```
i = 0
try:
   x = 1/i
except ZeroDivisionError as error_info:
   print(error_info)
```

Uitvoer:

```
division by zero
```

17. Functies.



Een functie heeft de volgende kenmerken:

- het bevat een aantal statements
- aan een functie kunnen parameters (argumenten) meegegeven worden.
- een functie geeft een waarde terug

Een functie heeft de volgende voordelen:

- door het programma op te bouwen uit functies, wordt het overzichtelijker
- een functie kun je opnieuw en met verschillende parameters gebruiken

We geven een voorbeeld van een functie:

```
def som(x,y):              # definitie van functie ,
                           # x en y zijn formele parameters
   z = x+y                 # z is een lokale variabele
   return z                # de waarde z wordt teruggegeven

print(som(2,3))            # de waarde z wordt teruggegeven

print(som('aap','je'))     # parameters zijn van type 'string'
a =3
b = 1e100                  # a en b zijn globale variabelen
print(som(a,b))            # Deze worden toegekend aan
                           #             de formele parameters.
```

De definitie van een functie.

Een voorbeeld van een functie-definitie is:

```
def som2(x,y):
    z = x+y
    return z
```

De parameters (argumenten) die met de definitie van een functie worden meegegeven heten: formele parameters.

De aanroep van een functie.

Een functie kan op verschillende manieren worden aangeroepen:

```
som(2,3)
som('aap','je')
som(a,b)
```

De parameters, die met een functie-aanroep worden meegegeven heten actuele parameters

Globale en lokale variabelen.

Variabelen, die binnen en functie gedefinieerd zijn, heten lokale variabelen. Ze zijn niet bekend buiten de functie. Variabelen, die voor de functie-definitie zijn gedefinieerd, heten globale variabelen. Zij zijn wel bekend binnen de functie, tenzij binnen de functie een lokale variabele dezelfde naam heeft.

Voorbeeld:

```
a = 3          # a is globaal
b = 7          # b is globaal

def g(x,y):
    z = x+y    # z is lokaal
    print('locals:', locals().keys())
    print('globals:', globals().keys())
    return z

print(g(a,b))
```

Uitvoer:

```
locals: dict_keys(['y', 'x', 'z'])
globals: dict_keys(['g', '__builtins__', '__package__','__file__',
                    '__doc__', '__spec__', '__cached__', 'b',
                    '__name__', '__loader__', 'a'])
10
```

Een lokale variabele en een globale variabele kunnen dezelfde naam hebben.

Voorbeeld:

```
a = 3          # a is globaal
b = 7          # b is globaal

def g2(x,y):
    a = 55
    print('a = ',a)
    z = x+y
    print('locals:', locals().keys())
    print('globals:', globals().keys())
return z

print(g2(a,b))
print()
print('a =',a)
```

Uitvoer:

```
a =  55
locals: dict_keys(['z', 'y', 'x', 'a'])
globals: dict_keys(['__name__', 'b', '__cached__', '__file__', 'g2',
                    '__builtins__', 'a', '__doc__', '__loader__',
                    '__package__', '__spec__'])
10

a = 3
```

Het is mogelijk een globale variabele binnen een functie te wijzigen. Dit moet dan expliciet aangegeven worden.

Voorbeeld:

```
a = 3          # a is globaal
b = 7          # b is globaal

def g3(x,y):
    global a
    a = 55
    print('a =',a)
    z = x+y
    print('locals:', locals().keys())
    print('globals:', globals().keys())
    return z

print(g3(a,b))
print()
print('a =',a)
print()
```

Uitvoer:

```
a = 55
locals: dict_keys(['z', 'x', 'y'])
globals: dict_keys(['__name__', '__doc__', '__loader__',
                    '__package__', 'b', '__builtins__', 'a',
                    '__cached__', '__spec__', 'g3', '__file__'])
10

a = 55
```

Advies: Wijzig een globale variabele niet.
De kans op fouten wordt groter..

De waarde die de functie teruggeeft.

Een functie geeft altijd een waarde terug.
Als geen return-opdracht wordt gegeven dan wordt de waarde 'None' teruggeven

Voorbeeld:

```
def doeNiets(i): pass # doe niets

x = doeNiets(0)
print(x)
```

Uitvoer:

None

Meestal wordt aan het einde van de functie-declaratie een return-opdracht gegeven. Het is ook mogelijk op andere plaatsen binnen de functie een return-opdracht te laten uitvoeren. Na het uitvoeren van de return-opdracht wordt de functie verlaten.

We geven een voorbeeld van een functie met meerdere return-opdrachten.

```
def myabs(i):
   if i < 0:
      return -i
return i

print(myabs(-3))
print(myabs(6))
```

Uitvoer:

3
6

Vraag: hoe kan de functie worden aangepast, zodat maar één return-opdracht nodig is?

Parameters.

Het is mogelijk een functie geen parameters mee te geven. Ook kan een functie geen return-statement bevatten.

Voorbeeld:

```
def printTafelVanVijf():
   for i in range(1,11):
      print(i,'keer 5 is',5*i)

printTafelVanVijf()
```

Uitvoer:

1 keer 5 is 5
2 keer 5 is 10
3 keer 5 is 15
4 keer 5 is 20
5 keer 5 is 25
6 keer 5 is 30
7 keer 5 is 35
8 keer 5 is 40
9 keer 5 is 45
10 keer 5 is 50

Bij de aanroep van een functie worden de actuele parameters in de lokale 'symbol table' geplaatst. De 'symbol table' bevat niet de objecten zelf maar de verwijzingen (references) naar de objecten.

Als een actuele parameter een getal, een string of een tupel is, dan kan deze binnen de functie niet gewijzigd worden.

Als de parameter een lijst of dictionary is, dan kan een element van de parameter gewijzigd worden. De parameter zelf kan niet gespijzigd worden.

Voorbeeld 1:

```
def f(i):
    i *=2;
    return i

j = 4;
print(f(j))
print(j)        # j blijft ongewijzigd
i = 5
print(f(i))
print(i)        # i blijft ongewijzigd
```

Uitvoer:

```
8
4
10
5
```

Voorbeeld 2:

```
def g(a):
    a = [4,5,6]
    print(a)

a = [4,2,6,1]
print('vooraf:', a)
g(a)
print('achteraf:', a)    # a blijft ongewijzigd.
```

Uitvoer:

```
vooraf: [4, 2, 6, 1]
[4, 5, 6]
achteraf: [4, 2, 6, 1]
```

Voorbeeld 3:

```
def h(a):
   a[1] = 25
   print(a)

a = [4,2,6,1]
print('vooraf:',a)
h(a)
print('achteraf:',a)      # een element van a is gewijzigd
print()
```

Uitvoer:

```
vooraf: [4, 2, 6, 1]
[4, 25, 6, 1]
achteraf: [4, 25, 6, 1]
```

Meerdere waarden teruggeven.

M.b.v. tupels is het mogelijk meerdere waarden terug te geven.

Demo:

```
def machten(i):
   return i*i,i*i*i,i*i*i*i # of return (i*i,i*i*i,i*i*i*i)

u,v,w = machten(5)
print(u,v,w)
print(machten(6)[1])
t = machten(2)
print( t[-1])
u,v,w = t
print( u,v,w)
```

Uitvoer:

```
25 125 625
216
16
4 8 16
```

Parameters met default-waarden.

Parameters kunnen een default waarde krijgen.

Voorbeeld:

```
def f(i,t=0,h=0):
   return i + 10*t + 100*h

print(f(3))          # t == 0,  h == 0
print(f(3,4))        # h == 0
print(f(3,4,5))
print()
print(f(1,t=9))      # h == 0
print(f(1,h=2))      # t == 0
print(f(1,t=9,h=7))
print(f(1,h=8,t=5)) # t en h mogen in een andere volgorde
                     # staan.
```

Uitvoer:

```
3
43
543

91
201
791
851
```

Dit heeft de volgende voordelen:

- minder tikwerk
- de volgorde van de parameters hoeft niet onthouden te worden.

Een parameter, die bij de aanroep expliciet genoemd wordt, heet een keyword-parameter.

Functies met een onbepaald aantal argumenten.

In Python ligt het aantal argumenten bij de print-opdracht niet vast.

Dergelijke functies kun je ook zelf schrijven.
Voor het argument moet een '*' geplaatst worden. (dit heeft niets te maken met pointers uit C)

Een voorbeeld:

```
def product(*args):
   product = 1
   for i in args:
      product *= i
   return product

print(product())
print(product(1))
print(product(1,2))
print(product(1,2,3))
print(product(1,2,3,4))
```

Uitvoer:

```
1
1
2
6
24
```

Argumenten-lijst uitpakken.

Ga uit van de lijst `a = [34,35,36,37]`

Het product van de elementen van 'a' kan als volgt verkregen worden:

```
product(a[0],a[1],a[2],a[3])
```

De code is bewerkelijk en niet handig. Het werkt alleen goed als a vier elementen heeft.
Met de '*'-operator wordt de lijst uitgepakt in afzonderlijke argumenten.
Een verkorte notatie is hierdoor mogelijk:

```
product(*a)  # hetzelfde als product(a[0],a[1],a[2],a[3])
```

18. Klassen in Python.

We staan stil bij de volgende onderwerpen:

- attributen
- methoden
- constructor
- destructor
- klasse-attributen (onafhankelijk van een instantie)
- klasse-methoden (onafhankelijk van een instantie)
- de klasse weergeven via de print-opdracht
- overerving

Bij Python gelden de volgende regels:

- constructor: __init__(self, ...) (self verwijst naar het huidige object)
- destructor: __del__(self)
- voor print-opdracht: __repr__(self)
- attributen worden voorafgegaan door 'self.'
- attributen worden dynamisch gemaakt, meestal in de constructor
- er bestaan geen private attributen
- methoden hebben als eerste parameter 'self'
- klasse-attributen worden niet voorafgegaan door 'self'
- klasse-attributen worden voorafgegaan door '<klassenaam>'
- klasse-methoden hebben geen parameter 'self'
- klasse-methoden worden als volgt aangeroepen: <klassenaam>.<methode>

Het wordt geïllustreerd aan de hand van een voorbeeld:

```
class Clock():

   nclock = 0                    # klasse-attribuut

   @staticmethod
   def show_nclock():            # klasse-methode
      print(Clock.nclock)

   def __init__(self, hour=0, minute=0, second=0):  # constructor
      self.hour = hour           # attribuut
      self.minute = minute       # attribuut
      self.second = second       # attribuut
      Clock.nclock += 1

   def __del__(self):            # destructor
      Clock.nclock -= 1
      print("del aanroep")

   def __repr__(self):           # voor print-opdracht
      return "tijd: " + str(self.hour) + ":" + str(self.minute) + ":"
             + str(self.second)

c1 = Clock(10,25,30)
print(c1)
Clock.show_nclock()
c2 = Clock(10,25,31)
print(c2)
Clock.show_nclock()

del c1
Clock.show_nclock()

def f():
   Clock()
   Clock.show_nclock()
   # destructor wordt aangeroepen

print("f entrance")
f()
print("f exit")
Clock.show_nclock()
c2.show_nclock()      # werkt alleen als @staticmethod is toegevoegd
del c2
```

Uitvoer:

```
tijd: 10:25:30
1
tijd: 10:25:31
2
del aanroep
1
f entrance
del aanroep
1
f exit
1
1
del aanroep
```

De klasse 'ClockEx' is een uitbreiding van de klasse 'Clock'.

```
class ClockEx(Clock):    # ClockEx is een uitbreiding van Clock

   def tick(self):
      self.second = (self.second+1)%60
      if self.second == 0:
         self.minute = (self.minute+1)%6
      if self.minute == 0:
         self.hour= (self.hour+1)%24

c1 = ClockEx(10,25,30)
print(c1)
Clock.show_nclock()
c1.tick()
print(c1)
print()

c2 = ClockEx(23,59,59)
print(c2)
Clock.show_nclock()
c2.tick()
print(c2)
print()

del c1
c3 = c2
del c2    # del wordt niet aangeroepen,
          # want c3 verwijst ook naar het object
import time
time.sleep(5) # na 5 sec. verschijnt op het scherm: del aanroep
```

Uitvoer:

```
tijd: 10:25:30
1
tijd: 10:25:31

tijd: 23:59:59
2
tijd: 0:0:0

del aanroep
del aanroep # na 5 sec.
```

19 Dictionary

Een dictionary is een verzameling (key,value)-paren. De verzameling is niet gesorteerd.

De sleutels zijn uniek: ze mogen dus maar één keer voorkomen.

Voorbeeld:

```
cijferlijst = {'Piet':7.4, 'Jan':3.4, 'Kees':5.5}
print(cijferlijst)
```

Uitvoer:

```
{'Piet': 7.4, 'Kees': 5.5, 'Jan': 3.4}
```

Bij dit voorbeeld is de volgorde bij de uitvoer anders dan bij de invoer.
Bij een dictionary doet de volgorde er niet toe. Voor een efficiënte opslag wordt een hashing-techniek toegepast. Deze plaatst de elementen in een willekeurige volgorde.

Het volgende is mogelijk bij een dictionary:

- het toevoegen van een (key,value) paar
- het wijzigen van een (key,value) paar
- het verwijderen van een (key,value) paar
- dictionaries samenvoegen
- een dictionary met een for-loop doorlopen
- alle keys opvragen
- alle values opvragen
- een dictionary omzetten in een lijst met 2-tupels
- een lijst met 2-tupels omzetten in een dictionary

Voorbeeld:

```
cijferlijst = {'Piet':7.4, 'Jan':3.4, 'Kees':5.5}
print(cijferlijst)
print()

cijferlijst['Karel'] = 8.7     # toevoegen
print(cijferlijst)
print()

cijferlijst['Jan'] = 6.1       # wijzigen
print(cijferlijst)
print()

del cijferlijst['Karel']       # verwijderen
print(cijferlijst)
```

Uitvoer:

```
{'Jan': 3.4, 'Piet': 7.4, 'Kees': 5.5}

{'Jan': 3.4, 'Piet': 7.4, 'Karel': 8.7, 'Kees': 5.5}

{'Jan': 6.1, 'Piet': 7.4, 'Karel': 8.7, 'Kees': 5.5}

{'Jan': 6.1, 'Piet': 7.4, 'Kees': 5.5}
```

Dictionaries kunnen samengevoegd worden.

Voorbeeld:

```
cijferlijst2 = {'Peter':9.5, 'Henk': 2.3, 'Rob':7.9}
cijferlijst.update(cijferlijst2)
print(cijferlijst)
print()
```

Uitvoer:

```
{'Henk': 2.3, 'Rob': 7.9, 'Piet': 7.4, 'Jan': 6.1, 'Kees': 5.5,
 'Peter': 9.5}
```

Een dictionary met een for-loop doorlopen.

De volgende opdracht toont alleen de keys:

```
for i in cijferlijst:    # alleen de keys worden getoond
   print(i, end = ' ')
```

Uitvoer:

```
Peter Kees Jan Piet Henk Rob
```

Bovendien is de lijst niet gesorteerd.

Een gesorteerde lijst keys wordt met de volgende opdracht verkregen:

```
for i in sorted(cijferlijst): # alleen de keys worden getoond
   print(i, end = ' ')
```

De complete gesorteerde cijferlijst wordt als volgt verkregen:

```
for k,v in sorted(cijferlijst.items()):
   print(k,v)
```

Uitvoer:

```
Henk 2.3
Jan 6.1
Kees 5.5
Peter 9.5
Piet 7.4
Rob 7.9
```

Een gesorteerde lijst met keys wordt als volgt opgevraagd:

```
print(sorted(list(cijferlijst.values())))
```

Uitvoer:

```
[2.3, 5.5, 6.1, 7.4, 7.9, 9.5]
```

Klasse versus dictionary.

Gegevens,die bij elkaar horen kun je op meerdere manieren bij elkaar voegen.

We nemen als voorbeeld de gegevens van een artikel.
Een artikel bevat de volgende gegevens: nr, naam en prijs

Je kunt deze gegevens samenvoegen in een klasse.
Dit gaat als volgt:

```
class Artikel:
   def __init__(self, nr, naam, prijs):
      self.nr = nr
      self.naam = naam
      self.prijs = prijs

   def __repr__(self):
      return "<< nr: " + str(self.nr) + " naam: " + self.naam +
             " prijs: " + str(self.prijs) +  " >>"
```

Een lijst artikelen wordt als volgt gemaakt:

```
a1 = [None]*4

a1[0] = Artikel(35,'stoel',110)
a1[1] = Artikel(18,'bank',330)
a1[2] = Artikel(36,'kast',450)
a1[3] = Artikel(5,'tafel',180)

print(a1)
a1.sort(key = lambda art: art.nr)      #sorteer op nr
print(a1)
```

Uitvoer:

```
[<< nr: 35 naam: stoel prijs: 110 >>, << nr: 18 naam: bank prijs: 330 >>,
 << nr: 36 naam: kast prijs: 450 >>, << nr: 5 naam: tafel prijs: 180 >>]
[<< nr: 5 naam: tafel prijs: 180 >>, << nr: 18 naam: bank prijs: 330 >>,
 << nr: 35 naam: stoel prijs: 110 >>, << nr: 36 naam: kast prijs: 450 >>]
```

Je kunt de gegevens van een artikel ook samenvoegen in een dictonary.
Dit gaat als volgt:

```
d = {'nr':35,'naam':'stoel','prijs':110}
```

Een lijst artikelen wordt als volgt gemaakt:

```
a2 = [None]*4

a2[0] = {'nr':35,'naam':'stoel','prijs':110}
a2[1] = {'nr':18,'naam':'bank','prijs':330}
a2[2] = {'nr':36,'naam':'kast','prijs':450}
a2[3] = {'nr':5,'naam':'tafel','prijs':180}

print(a2)
a2.sort(key = lambda art: art['nr'])      # sorteer op nr
print(a2)
a2 = [sorted(art.items()) for art in a2]  # sorteer ieder element
print(a2)
```

Uitvoer:

```
[{'naam': 'stoel', 'prijs': 110, 'nr': 35},
 {'naam': 'bank', 'prijs': 330, 'nr': 18},
 {'naam': 'kast', 'prijs': 450, 'nr': 36},
 {'naam': 'tafel', 'prijs': 180, 'nr': 5}]
[{'naam': 'tafel', 'prijs': 180, 'nr': 5},
 {'naam': 'bank', 'prijs': 330, 'nr': 18},
 {'naam': 'stoel', 'prijs': 110, 'nr': 35},
 {'naam': 'kast', 'prijs': 450, 'nr': 36}]
[[('naam', 'tafel'), ('nr', 5), ('prijs', 180)],
 [('naam', 'bank'), ('nr', 18), ('prijs', 330)],
 [('naam', 'stoel'), ('nr', 35), ('prijs', 110)],
 [('naam', 'kast'), ('nr', 36), ('prijs', 450)]]
```

20. Modulen in Python.

De library van Python is een verzameling modulen.
Een module is een tekstfile met Python-code.
Door het gebruik van modulen wordt de Python-code verdeeld over meerdere tekstfiles .
Een overzicht van de beschikbare modules is te vinden in de Python Documentation, Library Reference

De volgende tabel beschrijft een een aantal modulen.

module	**functie**
sys	verkrijgen systeem-informatie, systeem-instellingen
os	werken met filesysteem
threading	werken met threads
socket	maken van internet-verbindingen
telentlib	toegang tot telnetservers
ftplib	toegang tot ftpservers
poplib	opvragen van email
BaseHtppServer	maken van webserver
anydbm	database voor strings
shelve	database voor objecten
sqlite3	sql-database

Een module importeren.

Als een module geïmporteerd wordt, gebeurt het volgende:

- de module wordt opgezocht
- de module wordt (zo nodig) gecompileerd (een '.pyc'-file wordt gemaakt)
- de module wordt gerund (dus: importeren houdt ook runnen in!)

Het programma, dat de module importeert, heeft vervolgens toegang tot de variabelen en functies binnen de module.

Voorbeeld 1.

De module 'sys' geeft informatie over het huidige systeem.

```
import sys
print(sys.platform)
```

Uitvoer:

```
win32
```

Voorbeeld 2.

Het kan ook als volgt:

```
from sys import platform # of  from sys import *
print(platform)
```

Voordeel: minder schrijfwerk
Nadeel: minder overzichtelijke code (als je meerdere modulen gebruikt, weet je niet bij welke module een variabele hoort)

Voorbeeld 3

Het is ook mogelijk een module-naam te vervangen.

```
import sys as s
print(s.platform)
```

Vooral bij lange module-namen biedt dat voordeel.

Een module vinden.

De zoek-volgorde is als volgt:

- de directory van de huidige (top-level) file
- de directories, die staan in de systeemvariabele PYTHONPATH
- de directories waar de standaard library modulen staan
- de directories, die worden genoemd in een '.pth'-file.
  (Deze file moet staan in de top level van de installatie directory)

Python kent ingebouwde modulen. Voorbeelden van ingebouwde modulen zin: 'sys' en 'time'
De niet-ingebouwd modulen zijn te vinden in: '<python-map>\Lib'

Modulen van derden worden geplaatst in '<python-map>\Lib\site-packages'

De inhoud van een module.

De inhoud van een module kan achterhaald worden door de opdrachten:

```
import sys
print(dir(sys))     # lijst met items
print(vars(sys))    # dictionary met items en beschrijvingen.
```

Verder kun je allerlei manieren aan de informatie komen. De meeste ontwikkelomgevingen geven informatie en de installatie van Python bevat documentatie. Verder is op internet informatie te vinden: https://docs.python.org/3/library/sys.html

21. Files in Python.

Lezen van en schrijven naar files.

We willen de volgende regels wegschrijven naar de tekstfile 'vb1.txt' :

1 een
2 twee
3 drie

Dit gaat als volgt:

```
f = open('vb1.txt','w')   # open de file 'vb1.txt'
                          # om in te schrijven.
f.write('1 een\n')
f.write('2 twee\n')
f.write('3 drie')
f.close()
```

'vb1.txt' staat in dezelfde directory als het python-programma.

Het lezen van een file gaat als volgt:

```
f = open('vb1.txt','r')   # open een file om uit te lezen
for line in f:
   if line[-1] == '\n':
       line = line[:-1]   # verwijder '\n'
                          # aan het einde van een regel
   print(line)
```

Uitvoer:

```
1 een
2 twee
3 drie
```

Het is mogelijk m.b.v. een lijst de regels in één keer weg te schijven:

```
f= open('vb2.txt','w')
f.writelines(['1 een\n','2 twee\n','3 drie'])
f.close()
```

De regels kunnen ook in één keer ingelezen worden.
De regels worden in een string-lijst geplaatst.

```
f = open('vb2.txt','r')
a = f.readlines()
f.close()
for s in a:
if s[-1] == '\n':
   s = s[:-1]
print(s)
```

De file kan in één keer ingelezen worden:

```
f = open('vb1.txt','r')
s = f.read()
f.close()
print('repr(s):',repr(s))
a = s.split('\n')
print(a)
for s in a:
   print(s)
print()
```

Uitvoer:

```
repr(s): '1 een\n2 twee\n3 drie'
['1 een', '2 twee', '3 drie']
1 een
2 twee
3 drie
```

Een file kan karakter voor karakter gelezen worden.

Voorbeeld:

```
f = open('vb1.txt','r')
s = f.read(1)
while s:
   print(s, end = '')
   s = f.read(1)
f.close()
print()
print()

f = open('vb1.txt','r')
s = f.read(1)
while s:
   print(repr(s), end = ',')
s = f.read(1)
f.close()
```

Uitvoer:

```
1 een
2 twee
3 drie

'1',' ','e','e','n','\n','2',' ','t','w','e','e','\n','3',
' ','d','r','i','e',
```

Tekstfiles en binaire files.

Voor tekstfiles in Windows geldt het volgende:

- bij het schrijven wordt '\n' omgezet in '\r\n'
- bij het lezen wordt '\r\n' omgezet in '\n'.

Als 'aaa\nbbb\nccc' wordt weggeschreven naar een tekstfile, dan worden 11+2 karakters in de file geplaatst.

Als we 'aaa\nbbb\nccc' letterlijk willen wegschrijven naar een file, dat kan dit met de opdracht:

```
f.write('aaa\nbbb\nccc','wb') #vb : write binary
```

Voorbeeld:

```
f = open('vb3.txt','w')
s1 = 'aaa\nbbb\nccc'
print('s1:',repr(s1))
print('len(s1):',len(s1))
f.write(s1)
f.close();

g1 = open('vb3.txt','r')
s2= g1.read()
print('s2:', repr(s2))
print('len(s2):',len(s2))
g1.close();

g2 = open('vb3.txt','rb')
s3= g2.read()
print('s3:', repr(s3))
print('len(s3):',len(s3))
g2.close()
```

Uitvoer:

```
s1: 'aaa\nbbb\nccc'
len(s1): 11
s2: 'aaa\nbbb\nccc'
len(s2): 11
s3: b'aaa\r\nbbb\r\nccc'
len(s3): 13
```

Files kopiëren

We laten drie methoden zien:

Methode 1: karakter na karakter kopiëren

```
def copyfile1(src,dest):
    f = open(src,'rb')
    g = open(dest,'wb')
    s = f.read(1)
    while s:
        g.write(s)
        s = f.read(1)
    f.close()
    g.close()
```

Nadeel: het kopiëren gaat erg traag

Methode 2: de hele file in één string inlezen en vervolgens de string wegschrijven

```
def copyfile2(src,dest):
    f = open(src,'rb')
    g = open(dest,'wb')
    s = f.read()
    if s:
        g.write(s)
    f.close()
    g.close()
```

Nadeel: deze methode werkt, als de file niet te groot is.

Methode 3: het inlezen en wegschrijven in datablokken van 2048 bytes.

```
def copyfile3(src,dest):
    f = open(src,'rb')
    g = open(dest,'wb')
    s = f.read(2048)
    while s:
        g.write(s)
        s = f.read(2048)
    f.close()
    g.close()
```

Voordeel: de methode is snel en het werkt ook voor grote files.

Het filesysteem

Bij de opdracht ' f = open('vb.txt','w') ' wordt een file geopend in de huidige werk-directory.

De huidige werk-directory kan als volgt worden opgevraagd:

```
import os
print(os.getcwd())
```

of

```
import os
print(os.path.abspath('.'))
```

We kunnen ook eerst een directory maken en daarin de file 'vb1.txt' plaatsen.

Dit gaat als volgt:

```
dirname = 'testdir'
if not os.path.exists(dirname):
   print( 'creating dir ' + dirname + ' ...')
   os.mkdir(dirname)
else:
   print( 'dir ' + dirname + ' exist')
f = open(dirname + '/vb1.txt','w')
f.write('aaa\nbbb\nccc')
f.close()
```

Uitvoer bij 1e keer:

```
creating dir testdir ...
```

Uitvoer bij 2e keer:

```
dir testdir exist
```

(Twee keer '`os.mkdir(dirname)`' aanroepen leidt tot een foutmelding.)

Als we de file in een subdirectory van een directory willen plaatsen dan moeten we dat als volgt doen:

```
dirname = 'testdir/subdir'
......
  os.makedirs(dirname)  # de directories 'testdir' en
                        #'subdir' worden gemaakt.
......
```

We geven een overzicht van opdrachten die betrekking hebben op files en directories.

	dos	**python**
listing directory	dir	os.listdir('.')
change directory	cd	os.chdir(…)
file copy	copy	shutil.copy(src, dest)
dir copy	xcopy	shutil.copytree(src, dest)
make dir	mkdir	os.mkdir(…)
delete file	del	os.remove(…)
delete dir	rmdir	os.rmdir(…) (werkt alleen bij een lege directory) shutil.rmtree(…) (werkt ook bij een niet-lege directory)
rename	rename	os.rename(src, dst)