

# **De programmeertaal Python**

## **Module 1**

- 01. Inleiding
- 02. Data-types basis
- 03. Basis statements
- 04. Data-types in detail: strings en lists
- 05. Files en encoding
- 06. Data-types in detail: dictionaries, tuples en sets
- 07. Functies
- 08. Modules

## **Module 2**

- 09. Inleiding classes
- 10. Python classes
- 11. Exceptions
- 12. Python Standard Library
- 13. Reguliere expressies (bonus)



# **Student-notities**

## **De programmeertaal Python**

01. Module 1 — Inleiding



Nijmegen



## Python

Scripting taal of programmeertaal?

- krachtiger dan TCL
- makkelijker, "cleaner" dan Perl
- makkelijker dan Ruby
- makkelijker dan Java
- makkelijker dan C++
- krachtiger dan Visual Basic

Eigenschappen



- object georiënteerd
- portabel
- krachtig
- makkelijk in gebruik
- makkelijk te leren
- vrij verkrijgbaar en verspreidbaar (open source)
- niet snel
- kan mixen met andere talen

Beschikbaar voor

- UNIX (inclusief Linux)
- MS Windows
- MacOS

Algemeen: [www.python.org](http://www.python.org)  
Documentatie: [docs.python.org](http://docs.python.org)

@ at computing

CC BY-NC-ND 4.0 | v12f – h01 – 1

## Aantekeningen

---

---

---

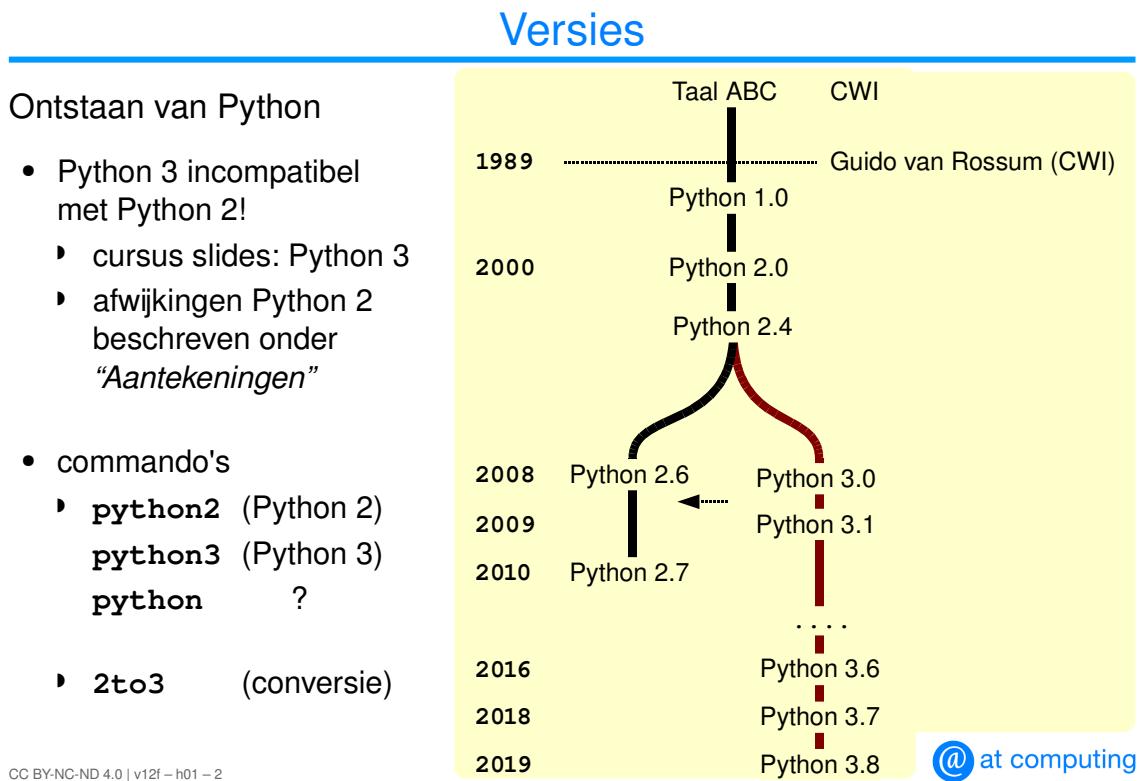
---

---

---

---

---



## Aantekeningen

---



---



---



---



---



---

Python 2.7 is in principe het laatste release voor Python versie 2. Tot het jaar 2020 worden er nog bugs opgelost.

---

Hier staan in deze cursus de verschillen beschreven tussen versie 2 en versie 3 (uitgangspunt voor de beschrijving op de slide).

---

# Toepassingen

## Toepassingsgebieden

- systeembeheer scripts
  - GUI
  - Internet scripting
  - database programmering
  - prototyping
  - ....
  - component integratie
    - Python Standard Library
    - Python Package Index
    - ....

## Bekende toepassingen

- YouTube
  - DropBox
  - Spotify
  - Instagram
  - Quora
  - Pinterest
  - onderdelen van Google
  - en vele andere!

## Aantekeningen

---

---

---

---

---

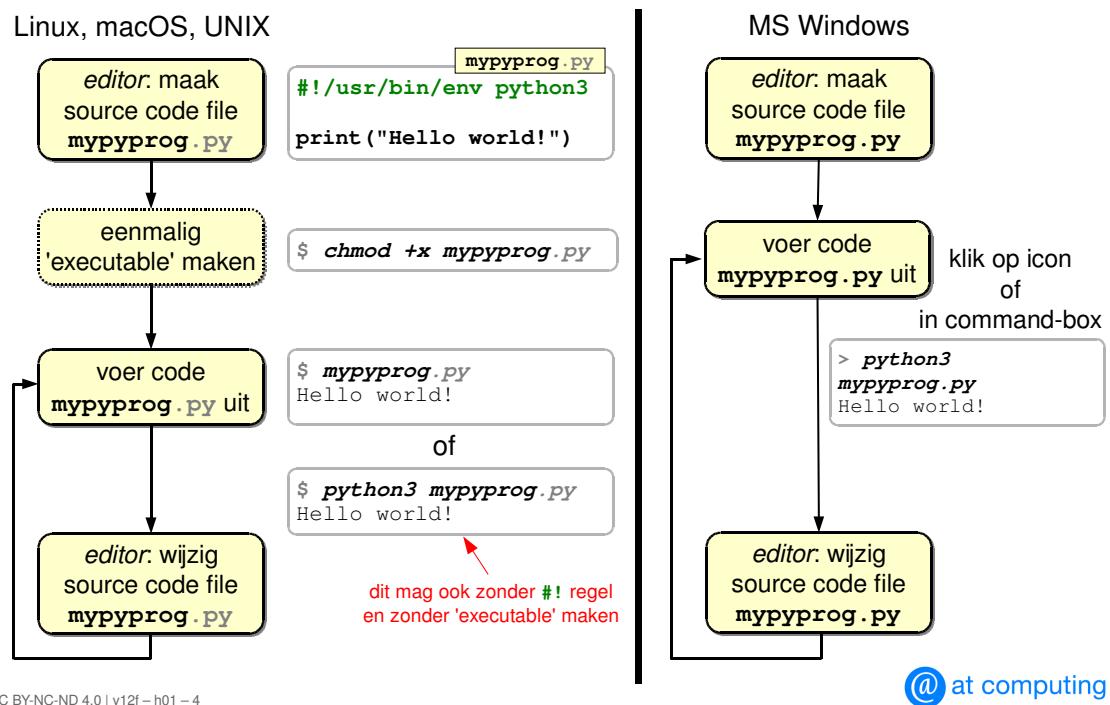
---

---

---

---

## Python programma maken en uitvoeren



CC BY-NC-ND 4.0 | v12f – h01 – 4

@ at computing

### Aantekeningen

Source code kan worden gecreëerd/onderhouden met een tekst-editor en daarna worden uitgevoerd vanaf de commando-regel of via een muisklik.

Echter, tijdens een ontwikkeltraject wordt vaak gebruik gemaakt van een *Integrated Development Environment* (IDE). Dit is een tool waarmee je een project met verschillende source files kunt manipuleren. Zo kun je source files bewerken én met een druk op de knop uitvoeren (Run) vanuit één omgeving.

Er zijn verschillende IDE-implementaties beschikbaar voor Python. De IDE IDLE wordt standaard meegeleverd bij Python.

In Python versie 2 is `print` een statement in plaats van een functie:

Versie 2	Versie 3
<code>print "Hello world!"</code>	<code>print("Hello world!")</code>

## Python shell

Python zelf interactief starten: *python shell*

```
$ python3
Python 3.3.0 (default, Feb 12 2013, 17:01:04)
[GCC 4.4.6 20120305 (Red Hat 4.4.6-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Bij prompt >>> Python commando's geven:

```
>>> a=2
>>> print(a)
2
>>> a
2
>>> print("Hello world!")
Hello world!
>>> quit()    of    ctrl-D    of    ctrl-Z (in command box)
$
```

CC BY-NC-ND 4.0 | v12f - h01 - 5



## Aantekeningen

---

---

---

---

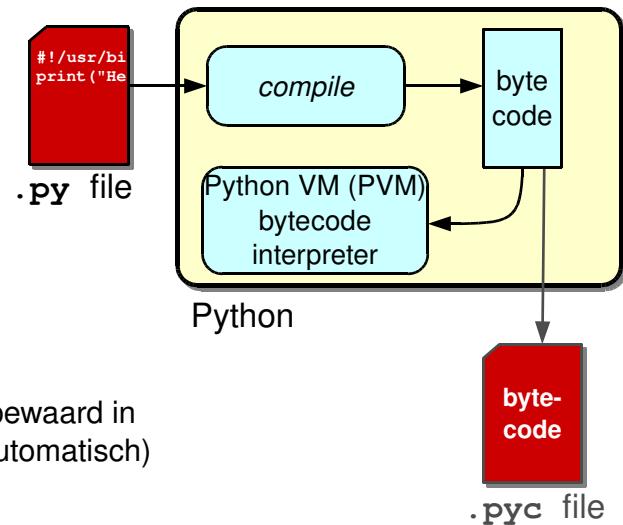
---

In Python versie 2 is `print` een statement in plaats van een functie:

Versie 2	Versie 3
<code>print a</code>	<code>print(a)</code>
<code>print "Hello world!"</code>	<code>print("Hello world!")</code>

## Python compilatie

Script vóór uitvoering gecompileerd



- bytecode wordt geïnterpreteerd
- bytecode van module eventueel bewaard in .**pyc** file (compiler probeert dit automatisch)
- efficiëntere interpreter
  - *pypy*

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Uitvoer genereren: `print()`

Uitvoer (default op stdout) met `print` functie

- uitvoer met regelovergang: `print("Hello world!")`
- uitvoer zonder regelovergang: `print("Hello world!", end="")`  
(default `end="\n"`)
- scheider `,` is spatie in uitvoer:  
geen scheider: `print("2 x 2 is", 2*2)`  
`print("2 x 2 is ", 2*2, sep="")`
- extra regelovergang: `print("regel 1\nregel 2")`
- lege regel: `print()`
- uitvoer naar standard error: `import sys`  
`print("Foutje!", file=sys.stderr)`

CC BY-NC-ND 4.0 | v12f – h01 – 7



### Aantekeningen

---



---



---

In Python versie 2 is `print` een statement in plaats van een functie:

Versie 2	Versie 3
<code>print "Hello world!"</code>	<code>print("Hello world!")</code>
<code>print "Hello world!",</code>	<code>print("Hello world!", end=" ")</code>
<code>print "2 x 2 is", 2*2</code>	<code>print("2 x 2 is", 2*2)</code>
<code>print "regel 1\nregel 2"</code>	<code>print("regel 1\nregel 2")</code>
<code>print</code>	<code>print()</code>
<code>print &gt;&gt;sys.stderr, "Foutje"</code>	<code>print("Foutje", file=sys.stderr)</code>

# **Student-notities**

## **De programmeertaal Python**

02. Module 1 — Data-types basis



Nijmegen



## Data types

Python is getypeerde taal: alle waarden hebben *type*

Soort	Voorbeeld	Eigenschap
Numbers	3.1415, 1234, 9999999999, 3+4j	immutable
Text (strings)	'Ni' 'Met "quote' "guido's"	immutable
Data (bytes)	b'Hello' b'\x00\xff'	immutable
Lists	[1, 2, 'Piet', 4]	mutable
Tuples	(1, 2, 'ABC', 4, 'U')	immutable
Dictionaries	{'Jan':42, 'Marie':38}	mutable
Sets	set([2, 3, 5, 7, 11])	mutable
Files	f = open('myfile', 'r') f.read()	
Booleans	False, True	immutable
NoneType	None ← geeft bewust aan: er is geen type/waarde	

Waarde bepaalt type:

```
a = 3      # a is nu getal
a = "Hello" # a is nu string
```

CC BY-NC-ND 4.0 | v12f – h02 – 1



## Aantekeningen

---



---



---

Versie 2	Versie 3
u'string in unicode'	'string in unicode'
'single byte-reeks (string)'	b'single byte-reeks'
b'single byte-reeks (string)'	b'single byte-reeks'

In Python 2 wordt een string opgeslagen als een (ASCII) byte-reeks. De b voor de open-quote wordt genegeerd.

## Numbers

---

Getallen in verschillende smaken

- integers

`-17, 5, 123456789012345678901234567890`

`0b10011010` (binair), `0o123` (octaal), `0x1234abcd` (hexadecimaal)

- floating point variabelen (C `double`)

`1.2345, 2.71828e-25, 8E12, 5.0`

- complexe getallen

`3+4j, 3.14-2.71j, 8j`

Bij gemixte expressies

- per deelexpressie integer “oprekken”  
naar floating point

`a = 1 + 11.3 * 3`

`34.9 = 1.0 + 33.9`

 at computing

CC BY-NC-ND 4.0 | v12f – h02 – 2

## Aantekeningen

---



---



---



---



---

Versie 2	Versie 3
<code>0o123</code>	
<code>0123</code>	

In versie 2 wordt onderscheid gemaakt tussen een integer en een long integer (“bignum”): een klein getal (dat in een `long` type uit de taal C past) wordt automatisch omgezet naar een “bignum” zodra het te groot wordt voor een `long` type. Je kunt in versie 2 afdwingen dat een klein getal gelijk als “bignum” wordt behandeld met een afsluitende `L` (bijvoorbeeld: `a = 25L`).

## Text

---

Text (strings) – opgeslagen in UNICODE

- genoteerd tussen quotes:                    `"hallo", 'hallo'`
- vrije keuze, dus kies handig:              `"kom 's morgens!"`  
`'met " dubbele quote'`
- met escape codes:                            `"string met \\, \t, \n en \"`  
`'regel 1\x0a regel 2' (0x0a = LF)`
- *raw string* maakt \ letterlijk:            `r'drie \\ backslashes'`  
`r"C:\temp\newfile.txt"`
- drievoudige quotes  
bij regelovergang:                            `"""een string over`  
`meerdere regels"""`  
`'''en nog eentje maar`  
`dan enkele quotes'''`

## Aantekeningen

Lijst van mogelijke escape sequences:

<i>Escape</i>	<i>Meaning</i>
\\	Backslash (\)
\'	Single quote (' )
\"	Double quote (" )
\a	ASCII Bell (BEL)
\b	ASCII Backspace (BS)
\f	ASCII Formfeed (FF)
\n	ASCII Linefeed (LF)
\r	ASCII Carriage Return (CR)
\t	ASCII Horizontal Tab (TAB)
\v	ASCII Vertical Tab (VT)
\ooo	Teken met octale waarde <i>ooo</i>
\xhh	Teken met hexadecimale waarde <i>hh</i>

Als het teken na de \ niet wordt herkend, dan wordt de \ en het daaropvolgende teken letterlijk genomen.

## Expressie (1)

---

Meeste operatoren werken naar verwachting, maar....

- deling      (met afronding):      **a / b**  
 $11.0 / 3.0 \rightarrow 3.66666\dots$   
 $11 / 3 \rightarrow 3.66666\dots$
- “heling”    (met afkapping):      **a // b**  
 $11.0 // 3.0 \rightarrow 3.0$   
 $11 // 3 \rightarrow 3$
- modulo:      **a % b**  
 $11.0 \% 3.0 \rightarrow 2.0$   
 $11 \% 3 \rightarrow 2$
- machtsverheffen:      **a \*\* b**
- bitsgewijze operatoren
  - bit-shift (links, rechts):      **a << b**      **a >> b**
  - AND, OR, XOR:      **a & b**      **a | b**      **a ^ b**

CC BY-NC-ND 4.0 | v12f – h02 – 4

 at computing

## Aantekeningen

Voorbeelden van bitsgewijze operatoren:

```

a = 0b00000111      # 0x07
b = 0b00001100      # 0x0c

c = a & b            # 0b00000100
c = a | b            # 0b00001111
c = a ^ b            # 0b00001011
  
```

---

Versie 2	Versie 3
$11 / 3 \rightarrow 3$	$11 / 3 \rightarrow 3.66666\dots$

In versie 2 levert een integer gedeeld door een integer een integer als resultaat (afgekapt).

## Expressie (2)

Meeste operatoren werken naar verwachting, maar.... (vervolg)

- logische **and** en **or**:  
`if expr1 and expr2: ....`  
`if expr1 or expr2: ....`
- **expr2** alleen berekend als nodig: `if j==0 or i/j < 5: ....`
- vergelijkingen
  - waarde gelijk aan, ongelijk aan: `if a == b:`      `if a != b:`
  - referentie naar zelfde object: `if a is b:`      `if a is not b:`  
(later meer)
  - combi vergelijking:  
is hetzelfde als:  
`if a < b <= c < d:`  
`if a < b and b <= c and c < d:`

Tabel met voorrangsregels: tab-6 in werkboek

[CC BY-NC-ND 4.0 | v12f - h02 - 5](#)



### Aantekeningen

---

---

---

---

---

---

---

---

---

---

---

## Toekenningen

### Toekenningsoperatoren

```
a = 5  
a += 4           # a is nu 9  
a *= 4           # a is nu 36
```

- varianten:       $+= \quad -= \quad *= \quad **= \quad //= \quad /= \quad \%=\newline &= \quad |= \quad ^= \quad <<= \quad >>=$

- algemeen

$a[i+3*j] += 7$     in plaats van     $a[i+3*j] = a[i+3*j] + 7$

- is sneller
- is duidelijker

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

---

## Strings allerlei (1)

Strings maken

```
a = 'parkeer'
b = 'bon'
c = """complexes<-
string"""
```

Strings en operatoren

```
boete = a + b          # boete wordt 'parkeerbon'
lekker = b * 2         # lekker wordt 'bonbon'
```

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

boete    **p a r k e e r b o n**

Indexeren en slice nemen

```
derde = boete[2]      # derde wordt 'r'
rechts = boete[-1]    # rechts wordt 'n'
maal   = boete[3:7]    # maal wordt 'keer' → slice
voor   = boete[0:5:2]  # van 0-5 elk 2e element: 'pre'
invers = boete[::-1]  # invers wordt 'nobreekrap'
```

String lengte

```
lengte = len(boete)  # lengte wordt 10
lengte = len(c)       # lengte wordt 15
```

CC BY-NC-ND 4.0 | v12f – h02 – 7

@ at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



---

## Strings allerlei (2)

Strings testen

```
if "on" in lekker:    # lekker bevat 'bonbon', dus True
    print('on')
if "en" in lekker:    # lekker bevat 'bonbon', dus False
    print('en')
if "mies" < "wim":   # True (lexicografisch, qua sortering)
    print('mies is kleiner')
```

String is  
geen getal

```
a = "123"
x = a * 3           # x wordt '123123123'
x = a + 3           # Exception! Geen + voor string en int

b = 123
x = b * 3           # x wordt 369
```

Conversies

```
a = "123"
b = int(a) + 3      # b wordt 126
c = str(b) * 3      # c wordt '126126126'
```

Variabele-type  
veranderen

```
a = "123"           # a is string
a = 312             # a is getal
```

CC BY-NC-ND 4.0 | v12f – h02 – 8

@ at computing

## Aantekeningen

Naast conversie-functies `int()` en `str()`, zijn ook andere conversies mogelijk:

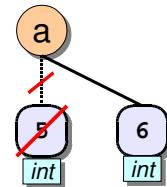
<code>str(42)</code>	→ "42"
<code>int("42")</code>	→ 42
<code>int("42", 8)</code>	→ 34
<code>int("42", 16)</code>	→ 66
<code>float("1.52")</code>	→ 1.52
<code>chr(65)</code>	→ 'A'
<code>ord('A')</code>	→ 65
<code>list('bon')</code>	→ ['b', 'o', 'n']
<code>tuple('bon')</code>	→ ('b', 'o', 'n')
<code>dict([('bon', 3), ('hoi', 7)])</code>	→ {'bon':3, 'hoi':7}

## Mutable vs. Immutable (1)

Python werkt met “references”

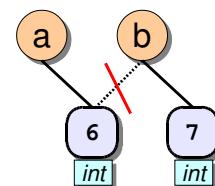
- naam wijst naar object van bepaald type

```
a = 5          # a is integer
a += 1         # integer is immutable
```



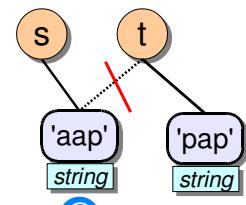
- na toekenning wijzen meerdere namen naar zelfde object

```
b = a          # a en b wijzen naarzelfde ding
if a == b:    # True
if a is b:    # True
b += 1         # integer is immutable
```



- analoog voor strings (ook immutable)

```
s = 'aap'
t = s          # s en t wijzen naarzelfde ding
t[0] = 'p'     # geeft exception!
t = 'p' + t[1:] # mag wel: nieuw object
```



CC BY-NC-ND 4.0 | v12f – h02 – 9

@ at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---

## Mutable vs. Immutable (2)

- lists, dictionaries en sets zijn *mutable*

```

l = [ 5, 4, 3 ]      # l is list
m = l                # l en m wijzen naar zelfde object
n = l[:]              # n wordt kopie van l (slice)

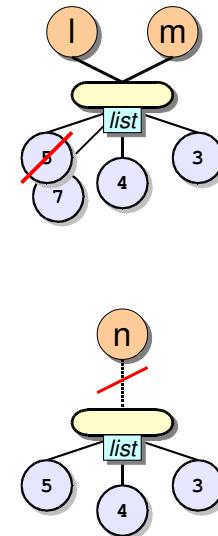
if l == m:            # True
if l == n:            # True
if l is m:            # True
if l is n:            # False

l[0] = 7
print(m[0])          # waarde 7
print(n[0])          # waarde 5
    
```

- object (waarde) verdwijnt als laatste referentie verdwijnt

```

n = 'aap'             # n wordt ander (type) object   of
n = None               # n refereert naar 'niets'     of
del n                 # n wordt verwijderd uit namespace
    
```



at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



# **Student-notities**

## **De programmeertaal Python**

03. Module 1 — Basis statements



Nijmegen



## Programmastructuren

Statements een-voor-een uitgevoerd

- einde van statement: einde van regel of ;
- constructie tussen haken () [] {} mag regeloverschrijdend zijn, of ↗

```
langevariabelenaam = (1 + 2 + 3 +
                      5 + 6 + 7 + 8)
langevariabelenaam = 1 + 2 + 3 + ↗
                      5 + 6 + 7 + 8
```

Loop van programma wijzigen

- **if-elif-elif-else**
- **for-else**
- **while-else**
- **break** of **continue**
- **pass**
- **try-except-else-finally**

## Aantekeningen

---

---

---

---

---

---

---

---

## Blokstructuren

### Blokstructuren in Python

- wat oog wil, is wet: indentatie is enige aanwijzing (ook voor compiler)
- geen verwarring mogelijk

```
if conditie1:  
    if conditie2:  
        statement1  
else:  
    statement2
```

```
if conditie1:  
    if conditie2:  
        statement1  
    else:  
        statement2
```

- gebruik voor indentatie geen TABS (voor Python altijd 8-voud) maar spaties!

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Coderingstijl

### Style Guide for Python Code (PEP8)

- commentaar: `# dit is commentaar`
- indentatie: 4 spaties
- regellengte: maximaal 79 tekens (afbreken met \↵)
- namen:
  - case-sensitive
  - naamconventies
    - functies: alleen kleine letters/cijfers
    - globale variabele: alleen kleine letters/cijfers, maar hoofdletters indien gebruikt als constante (`MAXBUF=256`)
    - package/module: alleen kleine letters/cijfers, kort (bestandsnaam)
    - class: **CapWords**

CC BY-NC-ND 4.0 | v12f – h03 – 3

 at computing

### Aantekeningen

---

---

---

---

---

---

---

Wijzigingen in Python worden doorgevoerd d.m.v. PEPs (Python Enhanced Proposals). PEP8 beschrijft de conventie voor de layout van je code, naamgeving van je variabelen, etc.

## Statement if

Algemene vorm

```
if conditie:  
    statement(s)  
elif conditie:  
    statement(s)  
else:  
    statement(s)
```

- **elif** en **else** optioneel
- slechts één statement bij **if/elif/else?** dan mag op zelfde regel
- voorbeelden

```
if x < 0:  
    print("x is negatief")  
elif x > 0:  
    print("x is positief")  
else:  
    print("x is nul")
```

```
if x < 0:    print("x is negatief")  
elif x > 0:  print("x is positief")  
else:        print("x is nul")
```

CC BY-NC-ND 4.0 | v12f – h03 – 4

@ at computing

### Aantekeningen

Python kent geen meerweg-keuze zoals het statement `switch` of `case` in andere talen. In Python kun je dit gemis opvangen met een `if-elif` constructie:

```
if    val == 1:  
    ...  
elif val == 2:  
    ...  
elif val == 3:  
    ...  
else:  
    ...
```

Python kent wel een ternary constructie:

```
smallest = x if x < y else y
```

## Statement while

Algemene vorm

```
while conditie:  
    statement(s)  
else:  
    statement(s)
```

- **else** optioneel (eenmalig na afloop van **while** lus, behalve bij **break**)
- voorbeelden

```
n = 42  
  
while n > 0:  
    print("Lang leve Python!")  
    n -= 1  
else:  
    print("Negatieve gevoelens?")
```

```
while func() > 0: print("func() is nog positief")
```

CC BY-NC-ND 4.0 | v12f – h03 – 5

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Statement **for** (1)

Alle elementen van reeks afhandelen

Algemene vorm

```
for eenellem in reeks:  
    statement(s)  
else:  
    statement(s)
```

- **else** optioneel (eenmalig na afloop van **for** lus, behalve bij **break**)
- voorbeelden

```
a = 'string'  
b = [1, 3, 5, 7]  
c = (2, 4, 6, 8)  
d = {'piet':1961, 'anja':1984}  
  
for e in a: print(e, end=' ')      # uitvoer: s t r i n g  
for e in b: print(e, end=' ')      # uitvoer: 1 3 5 7  
for e in c: print(e, end=' ')      # uitvoer: 2 4 6 8  
for e in d: print(e, end=' ')      # uitvoer: piet anja
```

CC BY-NC-ND 4.0 | v12f – h03 – 6

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Statement `for` (2)

### Gebruik van iterators

- stel: `for` iteratie op elementen van list in *gesorteerde* volgorde

```
lst = [13, 5, 77, 16, 9]

for val in sorted(lst):
    print(val, end=' ')      # uitvoer: 5 9 13 16 77

sortlst = sorted(lst)        # [5, 9, 13, 16, 77]
```

- functie `sorted` geeft nieuwe (tijdelijke) list terug met gesorteerde elementen
  - na `for` iteratie wordt deze list weer verwijderd
- nadeel: extra geheugenallocatie

- iterator

- geeft pas nieuwe waarde zodra daarom gevraagd wordt

```
for val in reversed(lst):
    print(val, end=' ')      # uitvoer: 9 16 77 5 13

rlist = list( reversed(lst) ) # [9, 16, 77, 5, 13]
```

CC BY-NC-ND 4.0 | v12f – h03 – 7



### Aantekeningen

De functie `sorted` is een standaard functie die een gesorteerde list teruggeeft. Daarentegen is `reversed` een standaard *iterator* die de waarden uit een reeks — in dit geval de list `lst` — een-voor-een aanreikt zodra daarom gevraagd wordt (in omgekeerde volgorde).

Veel functies die in Python 2 een list teruggeven, zijn in Python 3 omgebouwd tot iterators omdat ze minder geheugen nodig hebben voor opslag van de resultaat-list. Een voorbeeld van zo'n omgebouwde functie is `range` (zie volgende slide).

Je kunt de functie `list` gebruiken als je toch een echte list wilt bouwen met de waarden die een iterator teruggeeft, of de functie `tuple` als je van die waarden een tuple wilt bouwen.

Iterators worden geïmplementeerd als classes, die de methods `__iter__` en `__next__` (`next` in Python 2) bieden.

Een bijzondere variant van een iterator is een *generator-functie*, die zijn waarden teruggeeft met het `yield` statement. Hier komen we aan het einde van hoofdstuk 7 “Module 1 — Functies” op terug.

## Statement **for** (3)

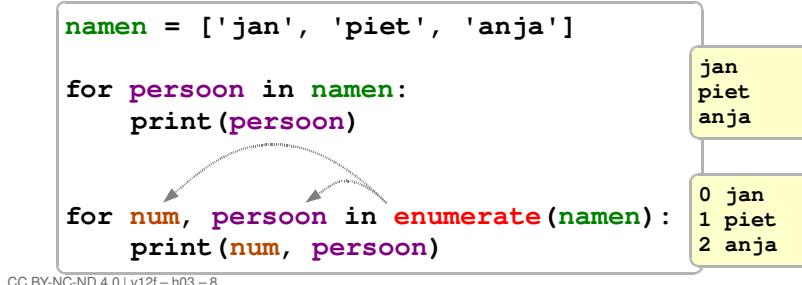
Iterator **range()**: genereer reeks getallen

```
for n in range(10):
    print(n, end=' - ')      # 0 - 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 -
```

- mogelijke aanroepen

<code>range(10)</code>	getallen van 0 t/m 9
<code>range(1, 11)</code>	getallen van 1 t/m 10
<code>range(3, 20, 5)</code>	getallen van 3 t/m 19 in stappen van 5 (dus 3, 8, 13, 18)
<code>range(5, 0, -1)</code>	getallen 5, 4, 3, 2, 1

Iterator **enumerate()**: genereer elementnummer bij iedere waarde



@ at computing

### Aantekeningen

---



---



---

De `enumerate` iterator geeft met iedere loopslag *twee* waarden af, resp. wat is het elementnummer en wat is de waarde die je vindt in dat element.

---

Versie 2	Versie 3
iterator: <code>xrange(500000000)</code>	iterator: <code>range(500000000)</code>
maak list: <code>range(500000000)</code>	maak list: <code>list(range(500000000))</code>

De iterator `xrange` in versie 2 is vergelijkbaar met de iterator `range` in versie 3.

## Statements **break** en **continue**

Breeklusconstructie af: **break**

```
namen      = ['erik', 'anja', 'john', 'petra', ....]
zoeknaam = input('Geef naam: ')

for persnum, persnaam in enumerate(namen):
    if persnaam == zoeknaam:
        print(zoeknaam, 'heeft nummer', persnum)
        break
    else:
        print(zoeknaam, 'niet gevonden!')

print('Tot je dienst')
```

Voorbeeld:  
Zoek ingegeven naam

Sla rest van huidige lusslag over: **continue**

```
som = 0

for getal in range(1, 101):
    if getal%13 == 0:
        continue
    som += getal

print(som)
```

Voorbeeld:  
Sommeer getallen van 1 t/m 100 uitgezonderd veelvouden van 13

@ at computing

### Aantekeningen

Stel dat de `for`-constructie geen `else` zou kennen, dan zou de melding dat de zoeknaam niet gevonden is via een extra indicator gegeven moeten worden:

```
.....
found = False
for persnum, persnaam in enumerate(namen):
    if persnaam == zoeknaam:
        print(zoeknaam, 'heeft nummer', persnum)
        found = True
        break

if not found:
    print(zoeknaam, 'niet gevonden')
```

## Exceptions (1)

Python is streng

```
>>> i=5
>>> j=0
>>> print(i/j)
Traceback (most recent call last):
ZeroDivisionError: division by zero

>>> a="hoi"
>>> print(a[17])
Traceback (most recent call last):
IndexError: string index out of range
```

Programma-executie stopt....

- vooraf testen? nee, dan wordt tweemaal getest!  
achteraf fout afvangen? ja!
- Python motto: “Easier to Ask for Forgiveness than ask for Permission (EAFP)”

CC BY-NC-ND 4.0 | v12f – h03 – 10

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Exceptions (2)

### Exceptions afvangen

```
i=5
j=0

try:
    print(i/j)
except Exception:
    print("Oei! dat mocht niet")
else:
    print("Het ging goed!") } optioneel

print("Hier komen we altijd")
```

### In eigen code exception forceren

```
if i < 0:
    raise Exception("i is negatief")      # eindigt met exception
else:
    print("met deze i kunnen we wat...")
```

CC BY-NC-ND 4.0 | v12f – h03 – 11

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Statement **pass**

### Statement **pass**

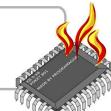
- op sommige plaatsen is statement verplicht, terwijl er niets te doen valt....  
☞ dummy statement **pass**

```
try:
    ....          # dit gaat mogelijk fout
except Exception:
    pass          # maar zorg dat programma verder loopt

# hier gaan we altijd verder
```

- ook mogelijk: je systeem als kachel...

```
while True:           # while 1: mag ook
    pass
```



 at computing

### Aantekeningen

---



---



---

In het voorbeeld op de slide zien we dat een integer waarde ongelijk aan 0 ook als True mag worden gebruikt.

In het algemeen gelden de volgende waarden als False:

False	
None	
0	(waarde 0)
''	(lege string)
[]	(lege list, tuple of dictionary)
()	
{}	



# **Student-notities**

## **De programmeertaal Python**

04. Module 1 — Data-types in detail: strings en lists



Nijmegen



## Variabelen dynamisch en sterk getypeerd

Variabelen nemen dynamisch type aan

```
a = "hallo daar"      # a is string
a = 4                  # a is integer
a = [1, 2, 3]          # a is list
a = {'aap': 4}         # a is dictionary
a = (4, 13, 7)         # a is tuple
```

Maar actie moet wel bij dat type passen

```
>>> a = 6              # a is integer
>>> b = a[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: 'int' object is not subscriptable
```

```
>>> a = "hallo daar"    # a is string
>>> a = a+7
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: must be str, not int
```

CC BY-NC-ND 4.0 | v12f – h04 – 1

@ at computing

### Aantekeningen

Als je wilt verifiëren of twee variabelen van hetzelfde type zijn óf je wilt verifiëren of een variabele van een bepaald type is, dan kun je de `type()` functie gebruiken. Een voorbeeld:

```
a = 'blurp'
b = 7
c = 352

if type(a) == type(b):           # levert False
    ...

if type(b) == type(c):           # levert True
    ...

if type(a) == str:               # levert True
    ...
```

## References

Toekenning bindt naam aan object

- oude binding ongedaan gemaakt

```
a = 7          # a gebonden aan int(7)
a = [1, 2, 3]  # 7 nu weg
a = 3          # lijst nu weg
```

- echter binding gaat ver

```
b = [1, 2, 3]      # b is lijst
c = b              # c is dezelfde lijst
c[1] = 7           # b[1] nu ook 7
```

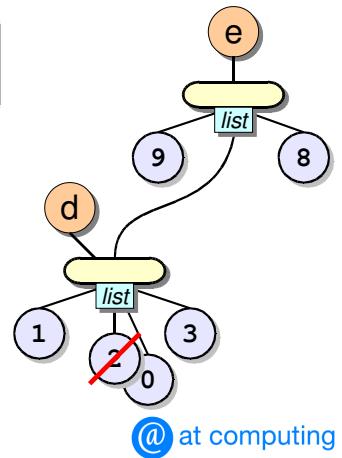
- nog verder zelfs

```
d = [1, 2, 3]      # d is lijst
e = [9, d, 8]       # e nu [9, [1, 2, 3], 8]
d[1] = 0            # e nu [9, [1, 0, 3], 8]
```

- echte kopie nodig?

```
f = [1, 2, 3]      # f is lijst
g = [9, f[:], 8]    # g nu [9, [1, 2, 3], 8]
f[1] = 0             slice # g nu [9, [1, 2, 3], 8]
```

CC BY-NC-ND 4.0 | v12f-h04 - 2



@ at computing

## Aantekeningen

---



---



---



---



---



---



---



---



---



---

## Standaard methods

Method: functie die bij datatype hoort

```
a = [7,1,3,2]      # a is list

a.sort()          # sort van list
#   a nu [1,2,3,7]

a.append(23)      # append van list (voegt 1 element toe)
#   a nu [1,2,3,7,23]

a.extend([6,5])  # extend van list (voegt hele lijst toe)
#   a nu [1,2,3,7,23,6,5]

n = a.pop()       # pop van list (verwijderd laatste element)
#   n wordt 5
#   a nu [1,2,3,7,23,6]

m = a.pop(3)     # pop van list (verwijderd element 3)
#   a nu [1,2,3,23,6]
```

en vele andere.....

CC BY-NC-ND 4.0 | v12f – h04 – 3

@ at computing

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Methods tonen met `dir`

Lijst van methods voor bepaald datatype

- opvragen met `dir(type)`

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__',
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse',
'sort']
```

- methods genaamd `__iets__` zijn *interne* methods (later meer)
- type kan zijn: `str, list, tuple, dict, int, float, ...`

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Method info tonen met `help`

Meer info over gebruik van methods

- opvragen met `help(type)` of `help(type.method)`

```
>>> help(list)
Help on class list in module __builtin__:

class list(object)
|   list() -> new empty list
|   list(iterable) -> new list initialized from iterable's items
|
|   Methods defined here:
|
|   __add__(...)
|       x.__add__(y) <==> x+y
<snip>
|   append(...)
|       L.append(object) -- append object to end
|
|   count(...)
|       L.count(value) -> integer -- return nr of occurrences of value
.....
>>> help(list.append)
....
```

CC BY-NC-ND 4.0 | v12f – h04 – 5

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Sequence (1)

Datatypes *string*, *list* en *tuple* veel gemeen: gezamenlijk “sequences”

```
s = "EYAWTKAPBWATA"
if "A" in s: ...          # 'A' in s ?
print(s[::-2])            # 'EATABAA'
c = len(s)                # 13
a = min(s); b = max(s)    # 'A' resp. 'Y'
```

```
l = [17, 51, 12, 3, 46, 25]
if 46 in l: ...          # 46 in l?
n = l[1:5:2]              # [51, 3] (1 tot 5!)
c = len(l)                # 6
a = min(l); b = max(l)    # 3 resp. 51
```

```
t = (23, "hallo", 77)
if 77 in t:               # True
u = t[:2]                  # (23, 'hallo')
c = len(t)                 # 3
a = min(t)                 # Exception: unorderable types!
b = max(t)                 # Exception: unorderable types!
```

CC BY-NC-ND 4.0 | v12f – h04 – 6

@ at computing

### Aantekeningen

---



---



---



---



---

Versie 2	Versie 3
2 < "aa" levert True op	2 < "aa" levert exception op

In versie 2 mogen de vergelijkingen `<`, `<=`, `>` en `>=` op gemixte typen (zoals integers en strings) gebruikt worden. In bovenstaand voorbeeld van de tuple is de integer 23 dan de kleinste waarde en de string 'hallo' de grootste.

## Sequence (2)

Veel-gebruikte sequence operaties:

<code>x in s</code>	test of element met waarde $x$ in $s$ zit
<code>x not in s</code>	test of element met waarde $x$ in $s$ niet zit
<code>s + t</code>	$s$ geconcateneerd met $t$
<code>s * n</code>	$n$ exemplaren van $s$
<code>n * s</code>	$n$ exemplaren van $s$
<code>s[i]</code>	element $i$ van $s$ , geteld vanaf 0
<code>s[i:j]</code>	slice van $s$ vanaf $i$ tot $j$
<code>s[i:]</code>	slice van $s$ vanaf $i$ tot einde
<code>s[i:j:k]</code>	idem, stapgrootte $k$
<code>len(s)</code>	lengte van $s$ (aantal objecten)
<code>min(s)</code>	kleinste element van $s$
<code>max(s)</code>	grootste element van $s$

Toekomstige types van “sequence-soort”: zelfde operaties

CC BY-NC-ND 4.0 | v12f – h04 – 7



### Aantekeningen

---

---

---

---

---

---

---

---

---

---

---

## String methods (1)

Strings hebben – naast sequence methods – nog extra methods

```
s = "!Jantje zag pruimen hangen!!"
c = s.count('an')    # 2 keer 'an'

i = s.find('ag')    # 9    s[9...] = 'ag...'
i = s.find('zzz')   # -1  niet gevonden

i = s.index('ag')   # 9
i = s.index('zzz')  # Exception!

u = s.upper()        # u bevat '!JANTJE ZAG ...'
z = s.strip('!')     # z bevat 'Jantje zag pruimen hangen'

p = s.replace('Jantje', 'Pietje')  # p bevat '!Pietje zag ...'

t = s.split()         # ['!Jantje', 'zag', 'pruimen', 'hangen!!']
for woord in t: print(woord)

v = "=".join(t)      # '!Jantje=zag=pruimen=hangen!!'
```

CC BY-NC-ND 4.0 | v12f - h04 - 8

string-method, niet list-method (string is scheider)

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## String methods (2)

Veelgebruikte string methods:

<code>s.count(s2[, b[, e]])</code>	tel aantal keer s2 in s of deel daarvan
<code>s.endswith(s2[, b[, e]])</code> <code>s.startswith(s2[, b[, e]])</code>	start of eindigt s (of substring) met s2?
<code>s.find(s2[, b[, e]])</code> <code>s.rfind(s2[, b[, e]])</code>	vind s2 in s vanaf links of vanaf rechts (-1 als niet gevonden)
<code>s.index(s2[, b[, e]])</code> <code>s.rindex(s2[, b[, e]])</code>	idem, met exception als niet gevonden
<code>s.isalnum()</code> , <code>s.isalpha()</code> , <code>s.isdigit()</code> , <code>s.islower()</code> , <code>s.isupper()</code> , <code>s.isspace()</code> , <code>s.istitle()</code> , <code>s.isprintable()</code>	test alle characters in s <i>true</i> als alle voldoen, <i>false</i> als niet of leeg
<code>s.lower()</code> , <code>s.upper()</code> , <code>s.capitalize()</code> , <code>s.swapcase()</code> , <code>s.title()</code>	maak hele string uppercase, lowercase, ...
<code>s.lstrip([s2])</code> , <code>s.rstrip([s2])</code> , <code>s.strip([s2])</code>	verwijder chars in s2 uit s (default: whitespace)

Merk op: string immutable, dus methods geven nieuwe string terug!

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---

## String methods (3)

Veelgebruikte string methods:

<code>s.replace(old, new[, cnt])</code>	vervang <i>old</i> door <i>new</i> in <i>s</i> (geeft nieuwe string)
<code>s.expandtabs([tabsz])</code>	vervang tabs door spaties in <i>s</i>
<code>s.join(seq)</code>	maak string van <i>seq</i> met scheider <i>s</i>
<code>s.split([sep[, max]])</code>	maak lijst van <i>s</i> , met scheider <i>sep</i>
<code>s.splitlines([keepends])</code>	maak lijst van regels (default: verwijder newlines)
<code>s.center(w), s.ljust(w), s.rjust(w)</code>	uitlijnen in breedte <i>w</i>

Merk op: string immutable, dus methods geven nieuwe string terug!

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## String formattering – klassiek (1)

Variabelen ingevuld voor %. in expressie

```
"v1 is %d\n" % v1
"de drie v's: %d %f %d\n" % (v1, v2, v3)
```

%d %i	geheel getal in decimale notatie
%o %x %X	geheel getal in octale of hexadecimale notatie
%e %E %f %F %g %G	floating point formaat met/zonder exponent, kies zelf
%c	character
%s %r	als string met <i>str()</i> , <i>repr()</i>
%%	letterlijke %

```
a=5.62
print(">%d<" % a)           # '>5<'
print(">%f<" % a)           # '>5.620000<'

a=1023
print(">%d<" % a)           # >1023<
print(">%o<" % a)           # >1777<          octaal
print(">%x< >%X<" % (a, a)) # >3ff< >3FF<    hex
```

CC BY-NC-ND 4.0 | v121 - 1104 - 11

@ at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



---



---

## String formattering – klassiek (2)

Formatting continues: addition between % and conversion code

<b>getal</b>	minimale veldbreedte, rechts uitlijnen
<b>-getal</b>	links i.p.v. rechts uitlijnen
<b>0getal</b>	voorloopnullen ipv spaties
<b>#</b>	0x ervoor bij %x en 0o bij %o
<b>.getal</b>	precisie-aanduiding

## Voorbeelden:

```
a=512
print(>%7d<" % a)      # >□□□512<           □ = spatie
print(>%-6d<" % a)      # >512□□□<
print(>%05d<" % a)      # >00512<
print(>%-05d<" % a)      # >512□□<           ☺
print(>%#x<" % a)      # >0x200<

b=5.115
print(>%7.2f<" % b)      # >□□□5.12<
print(>%1.f<" % b)        # >5.1<
```

CC BY-NC-ND 4.0 | v12f - h04 - 12



## Aantekeningen

---

---

---

---

---

---

---

---

## String formattering – `.format (1)`

String formattering method `str.format ()`

- variabelen insmelten voor {} in string – positioneel of benoemd

```
v = 5.3
w = 17

print( 'Values: {}, {}, {}'.format(v, "hey", w) )
# uitvoer: "Values: 5.3, hey, 17"

print( 'Values: {0}, {2}, {1}, {0}'.format(v, "hey", w) )
# uitvoer: "Values: 5.3, 17, hey, 5.3"

s = "{p1}, {p3}, {p2}, {{0}}".format(p1=v, p2="hey", p3=w)
# s bevat: "5.3, 17, hey, {0}"
```

CC BY-NC-ND 4.0 | v12f – h04 – 13

@ at computing

### Aantekeningen

---



---



---

Als je een dictionary gebruikt, mag je zelfs de keys meegeven om te refereren aan bepaalde elementen van de dictionary, zoals aangegeven in dit voorbeeld:

```
d = {'a':1, 'b':2, 'c':3}

print(d['a'])
print("{0[c]}, {0[b]}".format(d))
```

Merk op dat je de keys hier meegeeft *zonder* quotes!

## String formatting – `.format (2)`

Formatting gaat verder: {veld: specifier}

<u>specifier</u> :	[uitlijning]	[breedte]	[presentatie]
<	links (default string)	optioneel	c character
>	rechts (default int)	uitvullen en	d decimaal
^	centreren	precisie	o octaal
=	padding achter teken		x hexadecimaal
#	voorloop bij x, o en b		b binair
+	expliciet teken		e f g exponent, float of keuze
			s string
			% percentage (* 100)

### Aantekeningen

In Python 3.6 zijn *formatted string literals* (f-strings) geïntroduceerd, waarmee je — analoog aan de `format` method — accolade-paren in een speciaal soort string kunt opnemen om in de inhoud van variabelen in te smelten.

Een voorbeeld:

```
>>> naam = 'Piet Puk'
>>> leeftijd = 11
>>> aankondiging = f'En dit is {naam:^12} van {leeftijd:<4} jaar oud'
>>> print(aankondiging)
En dit is    Piet Puk    van 11    jaar oud
>>> print(f'En dit is {naam:^12} van {leeftijd:<4} jaar oud')
En dit is    Piet Puk    van 11    jaar oud
```

Als de waarde van de variabele `naam` later verandert, wijzigt de string in de variabele `aankondiging` *niet* automatisch mee!

## String formattering – .format (3)

Voorbeelden:

```
a=512
print(">{0:7}<".format(a))          # >□□□□512<      □ = spatie
print(">{0:^7}<".format(a))         # > 512  <
print(">{0:7}<".format(-a))        # > -512<
print(">{0:=-7}<".format(-a))      # >- 512<
print(">{0:<6}<".format(a))        # >512   <
print(">{0:05}<".format(a))         # >00512<
print(">{0:<05}<".format(a))       # >51200<      !!
print(">{0:#08x}<".format(a))       # >0x000200<

b=5.115
print(">{0}<".format(b))           # >5.115<
print(">{0:7.2f}<".format(b))      # >    5.12<
print(">{0:.1f}<".format(b))        # >5.1<
```

### Aantekeningen

---



---



---



---

Als je een enkele waarde wilt formatteren, kun je gebruik maken van de `format` functie:

```
getal = 7
bond  = format(getal, '>03d')
print(bond)                      → 007 (string variabele)
```

## Lists (1)

List: lijst met gegevens – types mogen verschillen

```

u = []          # lege list
v = [1, 2, 'ho']  # 3 elementen

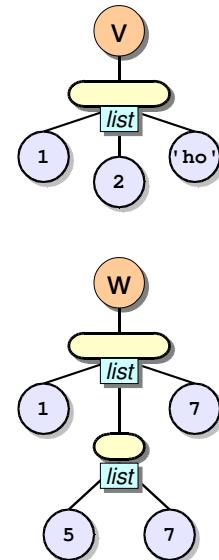
w = [1, [5, 7], 7] # 3 elementen: resp.
                    #      int, list, int

print(w[0])      # 1
print(w[1])      # [5, 7]
print(w[0:2])    # [1, [5, 7]]
print(w[1:])     # [[5, 7], 7]
print(w[1][0])   # 5

if 5 in w:       # False
    .....

if 5 in w[1]:   # True
    .....

```



CC BY-NC-ND 4.0 | v12f - h04 - 16

at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---

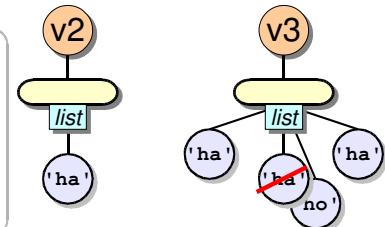
## Lists (2)

### Gebruik van operatoren

```
v2 = ["ha"]      # ['ha']
v3 = v2 * 3      # ['ha', 'ha', 'ha']

v3[1] = 'ho'     # ['ha', 'ho', 'ha']

v4 = v3 + v2      # ['ha', 'ho', 'ha', 'ha']
```



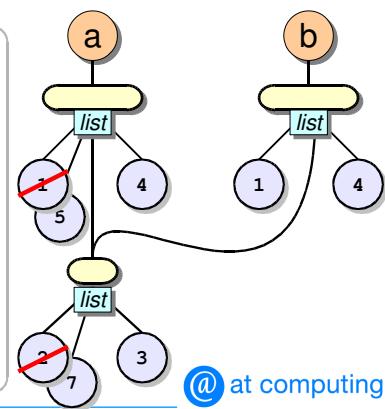
### Valkuil bij gebruik van slices

```
a = [1, [2, 3], 4]
b = a[:]          # b top-level kopie
                  # van a
a[0] = 5
print(b[0])       # waarde blijft 1

a[1][0] = 7
print(b[1][0])    # waarde 7 !!

import copy
c = copy.deepcopy(a) # c volledige kopie
```

CC BY-NC-ND 4.0 | v12f-h04 - 17



### Aantekeningen

---



---



---

In plaats van de notatie:

b = a[:]

mag ook:

b = a.copy()

Maar ook de .copy() method maakt slechts een top-level kopie!

## List methods en functies

```

v = [1,2,3]
v.reverse()          # [3,2,1]
v.append(4)          # [3,2,1,4]
v.insert(1, 6)        # [3,6,2,1,4]
v.sort()             # [1,2,3,4,6]

i = v.index(2)        # i=1  (v[1]==2)
v.extend([0,7])       # [1,2,3,4,6,0,7]

del v[4]              # [1,2,3,4,0,7]
a = v.pop()            # a=7, v=[1,2,3,4,0]
t = sum(v)             # t=10

s = "hoi!"             #
w = list(s)            # w=['h', 'o', 'i', '!']

z = sorted(v)           # z=[0,1,2,3,4]

```

- methods `sort()` en `reverse()` werken in-place (geven geen waarde terug)
- functie `sorted(list)` en iterator `reversed(list)` geven waarde terug

CC BY-NC-ND 4.0 | v12f – h04 – 18

@ at computing

## Aantekeningen

Het tussenvoegen of vervangen van één of meerdere elementen is (ook) mogelijk met de slice-notatie:

```

>>> v = [3, 2, 1, 4]

>>> v[1:1] = [6]
>>> v
[3, 6, 2, 1, 4]

>>> v[2:3] = [7, 8, 9]
>>> v
[3, 6, 7, 8, 9, 1, 4]

```

## List comprehension

List opbouwen:

```
l = []
for i in range(100):
    l.append(2**i)      # vul list l met tweemachten
```

Kan ook in één keer:

```
l = [ 2**i for i in range(100) ]
```

Dit kan ook:

```
l = [ 2**i for i in range(100) if i%13 == 0 ]
```

Zelfs dit kan:

```
a = [1,2,3]
b = [4,5]
l = [ (i,j) for i in a for j in b ]
print(l)          # [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

CC BY-NC-ND 4.0 | v12f – h04 – 19

@ at computing

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## List comprehension versus element vermenigvuldiging

Voorbeeld element vermenigvuldiging:

```
>>> a = [ [0]*2 ]*3
>>> a
[[0, 0], [0, 0], [0, 0]]
>>> a[0][1] = 7
>>> a
[[0, 7], [0, 7], [0, 7]]
```

三

```
>>> a = []
>>> a.append([0, 0])
>>> a.append( a[0] )
>>> a.append( a[0] )
>>> a[0][1] = 7
```

vergelijk:     $x = [1, 2, 3]$   
               $y = x$

## Voorbeeld list comprehension:

```
>>> a = [ [0]*2 for i in range(3) ]
>>> a
[[0, 0], [0, 0], [0, 0]]
>>> a[0][1] = 7
>>> a
[[0, 7], [0, 0], [0, 0]]
```

≡

```
>>> a = []
>>> a.append([0, 0])
>>> a.append([0, 0])
>>> a.append([0, 0])
>>> a[0][1] = 7
```

vergelijk: `x = [1, 2, 3]`  
`y = [1, 2, 3]`

CC BY-NC-ND 4.0 | v12f – h04 – 20





# **Student-notities**

## **De programmeertaal Python**

05. Module 1 — Files en encoding



Nijmegen



## Data in bestanden

### Twee soorten bestanden

- binaire bestanden
  - bytes in bepaald (gestandaardiseerd) formaat
  - geen verschil tussen diverse besturingssystemen
  - voorbeelden: .jpg .gif .png .pdf
- tekstuele bestanden
  - tekst voor weergave op beeldscherm/printer/...
  - encoding/decoding
  - regeleinde kan verschillen per besturingssysteem
    - UNIX-achtigen: in file: *linefeed (0x0a)*  
in Python-string: *linefeed ('\n')*
    - Windows: in file: *carriagereturn/linefeed (0x0d0a)*  
in Python-string: *linefeed ('\r\n')*  
dus conversie nodig bij lezen/schrijven

CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f – h05 –

 at computing

### Aantekeningen

---

---

---

In Python versie 3 worden strings in het geheugen opgeslagen in Unicode formaat. Dit impliceert dat de characters die gelezen worden uit een tekstbestand (dus als bij het openen van de file *geen b* wordt aangegeven) tijdens het lezen worden omgezet naar Unicode formaat in geheugen.

Als bytes worden gelezen uit een binaire file (*wel een b toegevoegd bij het openen*), worden deze bytes ongemodificeerd in geheugen geplaatst als bytes object.

## Binaire files – voorbeeld

Voorbeeld: lezen van .gif file

```

try:
    tux = open("tux.gif", "rb")
except Exception:
    ...
    'read'   'binary'

head  = tux.read(3)      # 3 bytes header: 'GIF'
vers  = tux.read(3)      # 3 bytes versie: '87a' of '89a'
wpx   = tux.read(2)      # 2 bytes breedte (little endian)
hpx   = tux.read(2)      # 2 bytes hoogte (little endian)

breed = wpx[1] * 256 + wpx[0]      # converteer naar integer
hoog  = hpx[1] * 256 + hpx[0]      # converteer naar integer

print("%s: versie %s, %dx%d" % (head, vers, breed, hoog))
    # uitvoer: b'GIF': versie b'89a', 595x842

tux.close()

```

In binary mode: **bytes** objects (`b"...."`)

CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f - h05 -



## Aantekeningen

Het eerste deel van een GIF file bevat de volgende informatie:

Offset	Length	Contents
0	3 bytes	"GIF"
3	3 bytes	"87a" or "89a"
6	2 bytes	<Logical Screen Width>
8	2 bytes	<Logical Screen Height>
10	1 byte	various markers
11	1 byte	<Background Color Index>
12	1 byte	<Pixel Aspect Ratio>
...		

In Python 2 geven de methods van de open file altijd een *string* (`str`) object terug, ook in geval van een binaire file.

## Binaire files – mode en methods

`f = open(filenaam [, mode])`

Wijze van openen (mode):

'rb'	voor lezen
'r+b'	voor lezen en schrijven
'wb'	voor schrijven
'w+b'	voor lezen en schrijven
'xb'	voor schrijven
'x+b'	voor lezen en schrijven
'ab'	voor append
'a+b'	voor append en lezen

	als niet-bestaat dan eerst creëren, anders leegmaken
	als niet-bestaat dan eerst creëren, anders leegmaken
	als niet-bestaat dan eerst creëren, anders exception
	als niet-bestaat dan eerst creëren, anders exception
	als niet-bestaat dan eerst creëren
	als niet-bestaat dan eerst creëren

Veelgebruikte methods:

<code>f.close()</code>	sluit file
<code>buf = f.read([size])</code>	lees en return alle bytes tot EOF (of size bytes)
<code>nrw = f.write(buf)</code>	schrijf bytes in buf naar f en return aantal bytes
<code>off = f.seek(offset [, hoe])</code>	positioneer op positie offset t.o.v. hoe: 0: BOF (default), 1: CUR, 2: EOF
<code>off = f.tell()</code>	geef flepositie t.o.v. BOF

CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f - h05 -

@ at computing

## Aantekeningen

---



---



---

Naast de genoemde methods, kun je ook nog attributen (variabelen) opvragen van het “file” object:

<code>f.name</code>	naam van de geopende file
<code>f.mode</code>	mode waarmee de file geopend is
<code>f.closed</code>	boolean die aangeeft of de file gesloten is
<code>f.encoding</code>	string die de encoding aangeeft in geval van een tekstuele file; we komen hier later in dit hoofdstuk uitgebreid op terug

## Tekstuele files – voorbeeld

Voorbeeld: toon alle regels en totaal aantal regels

```
fnaam = "invoerfile.txt"

try:
    f = open(fnaam, "rt")      # open voor lezen/text mode default
except Exception:             # (dus mag weggelaten worden)
    ...
    'read'   'text'

linecnt = 0
for linebuf in f:            # lees regel-voor-regel
    print(linebuf, end='')   # toon regel (zonder extra newline)
    linecnt += 1              # tel regel

f.close()                     # sluit file

print(fnaam, "bevat", linecnt, "regels")
```

In text mode: **str** objects (".....")

[CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f – h05 –](#)



### Aantekeningen

Alternatieven voor het voorbeeld op de slide, gebaseerd op `.readline()`:

<pre>f = open(filenaam, "rt") linecnt = 0 line    = f.readline()  while line:     print(line, end="")     linecnt += 1     line = f.readline()  f.close()</pre>	<pre>f = open(filenaam, "rt") linecnt = 0  while True:     line = f.readline()     if not line: break     print(line, end="")     linecnt += 1  f.close()</pre>
---	---

## Tekstuele files – mode en methods

`f = open(filenaam [, mode])`

Wijze van openen (mode):

' <b>r</b> '	voor lezen ('r' is default en 't' is default, dus mag worden weggelaten)
' <b>r+t</b> '	voor lezen en schrijven
' <b>w</b> '	voor schrijven
' <b>w+t</b> '	voor lezen en schrijven
' <b>x</b> '	voor schrijven
' <b>x+t</b> '	voor lezen en schrijven
' <b>a</b> '	voor append
' <b>a+t</b> '	voor append en lezen

' <b>r</b> '	voor lezen ('r' is default en 't' is default, dus mag worden weggelaten)
' <b>r+t</b> '	voor lezen en schrijven
' <b>w</b> '	voor schrijven als niet-bestaat dan eerst creëren, anders leegmaken
' <b>w+t</b> '	voor lezen en schrijven als niet-bestaat dan eerst creëren, anders leegmaken
' <b>x</b> '	voor schrijven als niet-bestaat dan eerst creëren, anders exception
' <b>x+t</b> '	voor lezen en schrijven als niet-bestaat dan eerst creëren, anders exception
' <b>a</b> '	voor append als niet-bestaat dan eerst creëren
' <b>a+t</b> '	voor append en lezen als niet-bestaat dan eerst creëren

Extra methods voor tekstuele files:

<code>line = f.readline([size])</code>	lees regel, maximaal <i>size</i> characters ( <i>size</i> optioneel); implicit bij <code>for</code> iteratie op <i>fle</i> object (vorige slide)
<code>lines = f.readlines([szhint])</code>	lees regels en return list, maximaal <i>szhint</i> characters
<code>f.writelines(lines)</code>	schrijf alle strings in list naar <i>f</i> (zelf '\n' toevoegen)
<code>newoffset = f.seek(offset)</code> <code>curoffset = f.tell()</code>	seek alleen naar absolute byte offset, bij voorkeur verkregen via <code>f.tell()</code>

[CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f – h05 –](#)

@ at computing

## Aantekeningen

### Buffering

De `open` functie kent een optionele parameter `buffering` waarmee je kunt regelen hoe geschreven data-bytes worden opgespaard door de interpreter voordat ze aan de lokale kernel worden doorgestuurd.

De default waarde voor deze parameter is `-1`: bytes die naar `files` worden geschreven worden opgespaard tot moten van typisch 4 KiB alvorens ze werkelijk worden geschreven, terwijl bytes die naar de terminal worden geschreven worden opgespaard per regel.

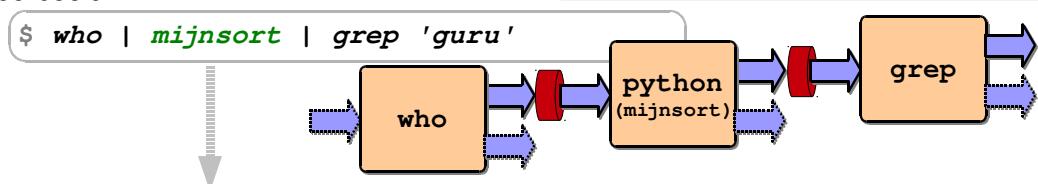
De parameter `buffering` kan expliciet op `0` worden gezet om de buffering geheel uit te schakelen en op `1` om altijd buffering op regel-basis te krijgen. Als voor buffering een waarde wordt meegegeven die groter is dan `1` geeft die waarde de zelfgekozen buffergrootte aan.

Met de `f.flush()` method kun je te allen tijde迫ceren dat een gedeeldelijk gevulde buffer naar de kernel wordt gestuurd.

## Gebruik van stdin, stdout en stderr

Op commandoregel manipuleren met stdin, stdout, stderr

- bestand aankoppelen met < **file** (stdin), > **file** (stdout) of 2> **file** (stderr)
- pipe aankoppelen met | symbol voorbeeld:



Python programma: sorteert regels van stdin en schrijft naar stdout

```
import sys                      # laad module 'sys'
                                 
alles = sys.stdin.readlines()    # lees regels van stdin in list
                                 
alles.sort()                     # sorteer list
                                 
sys.stdout.writelines(alles)     # schrijf regels naar stdout
```

CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f - h05 -

@ at computing

## Aantekeningen

---



---



---

In het tweede deel van dit hoofdstuk zullen we zien dat voor alle tekstuele files een encoding moet worden ingesteld. Deze is essentieel bij het omzetten van gelezen bytes naar een Unicode string en bij het omzetten van te schrijven bytes vanuit een Unicode string. Deze encoding wordt gespecificeerd bij het openen van een file. Voor de (reeds geopende) file objecten `sys.stdin`, `sys.stdout` en `sys.stderr` is deze encoding ingesteld op de default encoding.

Bijvoorbeeld:

```
import sys
print(sys.stdin.encoding)      → UTF-8
```

Met de environment variabele `PYTHONIOENCODING=...` kun je voor deze drie file objecten een andere encoding afdwingen.

## Encoding (1)

### Encoding

- welk bitpatroon (getal) moet je sturen om bepaald teken te genereren op printer/beeldscherm/...?
- verschillende standaarden
  - **ASCII** 7 bits (getalswaarden 0-127)  
cijfers 0-9, letters A-Z en a-z, leestekens, ....
  - **ISO-8859-..** 8 bits (ook getalswaarden 128-255)  
**ISO-8859-1** West-Europese extensie (aka **Latin1**)  
**ISO-8859-15** West-Europese extensie met € (aka **Latin9**)
  - **Windows-1252** 8 bits, Microsoft's afgeleide van **ISO-8859-1** (aka **cp-1252**)
  - en vele andere standaarden 😊

en welk bitpatroon heb je hier voor nodig?

teken	ASCII	Latin1	Latin9	cp-1252
A	0x41	0x41	0x41	0x41
¤	N/A	0xA4	N/A	0xA4
€	N/A	N/A	0xA4	0x80

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



---



---

## Encoding (2)

### Unicode

- abstractielaa met voor elk mogelijk teken een uniek 'codepoint' (getal)
  - aanvankelijk 16 bits (65.536 codepoints)  
later meer bits, genoeg voor 1.100.000 codepoints (momenteel 10% in gebruik)
  - codepoints      0-127      compatibel met **ASCII** reeks  
                      128-255     compatibel met **ISO-8859-1 (Latin1)**
  - notatie hexadecimaal (4 tot 6 posities) met formele beschrijving  
voorbeelden: **U+0041**    **LATIN CAPITAL LETTER A**  
**U+20AC**      **EURO SIGN**
- conversie van Unicode naar bytes
  - **UTF-8**      variabel aantal bytes per teken – defacto standaard  
**ASCII** tekens nog steeds in 1 byte (hardware-/software-compatibiliteit)
  - **UTF-16**      twee bytes per teken
  - .....

CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f – h05 –



### Aantekeningen

UTF-8 is een techniek om een Unicode-waarde om te rekenen naar een reeks van 2 tot 6 bytes, met uitzondering van de ASCII-waarden: die blijven 1 byte. Dit heeft als groot voordeel dat bestanden met ASCII-tekens leesbaar blijven en niet meer diskruimte nodig hebben. Verder is het prettig dat allerlei ASCII-georiënteerde hardware in gebruik kan blijven. Bij UTF-8 speelt endianess geen rol.

Bij UCS-2 wordt ieder teken opgeslagen in 2 bytes; daarvoor worden de eerste 65.536 Unicode codepoints 1-op-1 omgezet. Bij UCS-4 wordt ieder teken opgeslagen in 4 bytes en kunnen alle Unicode codepoints 1-op-1 worden omgezet.

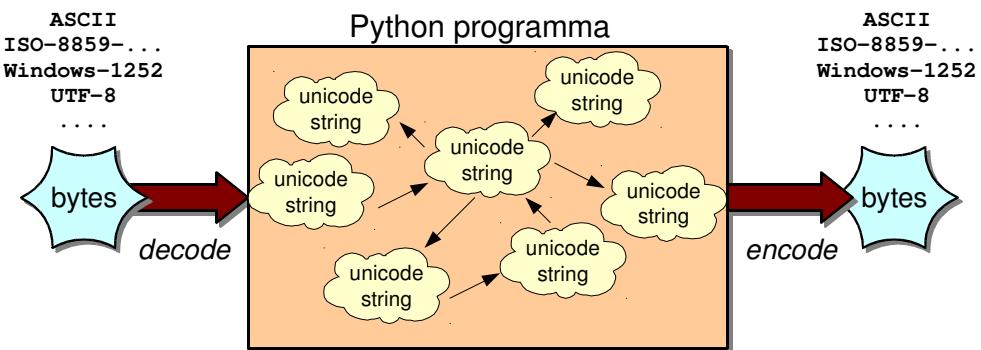
Bij UTF-16 worden de eerste 65.536 Unicode codepoints 1-op-1 omgezet naar 2-byte porties en een deel van de hogere codepoints worden nog afgebeeld in “gaten” uit die eerste reeks. Voor de andere hogere codepoints worden 4-byte porties gealloceerd. UTF-32 is gelijk aan UCS-4: een 1-op-1 afbeelding van de Unicode codepoints. Bij UTF-16 en UTF-32 encoding speelt endianess wel een rol.

Meer informatie over Unicode: [www.unicode.org](http://www.unicode.org)

## Encode/decode (1)

### Unicode “sandwich”

- alle string-operaties binnen Python-programma in Unicode
- data van/naar buiten: conversie naar/van Unicode



- lezen van data: bepaling van type encoding niet triviaal
  - informatie van leverancier van data, of
  - gokken....

CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f – h05 –

@ at computing

### Aantekeningen

Als je het type encoding niet (precies) weet van een file die je cadeau hebt gekregen, kun je de `detect` functie gebruiken uit de `chardet` module om een gooi te doen:

```
import chardet

f = open("promol.txt", "rb")          # binary open (encoding onbekend)

alldata = f.read()                   # lees gehele file

gokje = chardet.detect(alldata)    # laat een gokje doen

print(gokje)
```

De `detect` functie geeft een dictionary terug met de sleutels `encoding` en `confidence`:

```
{'confidence': 0.87625, 'encoding': 'utf-8'}
```

In Python 2 wordt met niet-Unicode strings gewerkt binnen het programma.

## Encode/decode (2)

Voorbeeld: schrijven van file in verschillende formaten

```

promo = "Sale: sm\u00f8rrebr\u00d8d voor 2\u20ac\n"
        # Sale: smørrebrød voor 2€
f = open("promo1.txt", "w", encoding='UTF-8')
f.write(promo)
f.close()

f = open("promo2.txt", "w", encoding='Latin9')
f.write(promo)
f.close()

f = open("promo3.txt", "w", encoding='Windows-1252')
f.write(promo)
f.close()

f = open("promo4.txt", "w")
f.write(promo)
f.close()

print(promo)  ← gebruiken 'preferred encoding'

```

CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f – h05 –

```

import locale
prefenc = locale.getpreferredencoding()
print("Preferred encoding:", prefenc)
# Preferred encoding: UTF-8

```

 at computing

### Aantekeningen

---



---

Een voorbeeld van een programma waarmee je een file in Windows-1252 encoding kunt omzetten naar een file in UTF-8 encoding:

```

fi = open("invoer.txt", "r", encoding='Windows-1252')
fo = open("uitvoer.txt", "w", encoding='UTF-8')

for line in fi:
    fo.write(line)

```

In Python 2 wordt de parameter `encoding` voor de `open` functie niet ondersteund (meer informatie aan het einde van dit hoofdstuk).

## Encode/decode (3)

Encodings kennen niet per definitie *alle* tekens

```
promo = "Sale: smørrebrød voor 2€\n"

f = open("promo5.txt", "w", encoding='ASCII')
f.write(promo)      UnicodeEncodeError: 'ascii' codec can't encode character '\xf8'

f = open("promo6.txt", "w", encoding='Latin1')
f.write(promo)      UnicodeEncodeError: 'latin-1' codec can't encode character '\u20ac'
```

Fouten bij encoding negeren of vervangen door '?':

```
f = open("promo5.txt", "w", encoding='ASCII', errors='ignore')
f.write(promo)          # Sale: smrrebrd voor 2

f = open("promo5.txt", "w", encoding='ASCII', errors='replace')
f.write(promo)          # Sale: sm?rrebr?d voor 2?
```

### Aantekeningen

In de string “Sale: smørrebrød voor 2€\n” hebben we nu de werkelijke tekens ø en € gebruikt in plaats van de \u escape sequences. Dit betekent dat deze tekens ook in de source code file terechtkomen; deze file moet dan door de Python interpreter (en je editor) ook volgens de juiste encoding worden gelezen.

In Python 2 is de default encoding voor je source code ASCII, waardoor bovenstaande string een foutmelding oplevert:

```
SyntaxError: Non-ASCII character '\xc3' in file ...
```

Je kunt dan een andere encoding definiëren door de volgende string als “magic comment” in de eerste of tweede regel van je source file op te nemen (dit voorbeeld geldt voor utf-8 encoding):

```
# -*- coding: utf-8 -*-
```

In Python 3 is de default encoding voor je source code UTF-8. Met de “magic comment” regel kun je daarvan afwijken.

## Strings en bytes (1)

Type **str** is geen type **bytes**!

- **str** is reeks codepoints
- **bytes** is reeks byte-waarden (0–255)

```
v1 = b'abc\x00\xff'          # bytes literal
v2 = bytes([97, 98, 99, 0, 255])    # iterable conversie
v3 = bytes(3)                  # reeks van 3 0x00 bytes

print(v1, v2, v3)            # b'abc\x00\xff' b'abc\x00\xff' b'\x00\x00\x00'

for i in v1:
    print(i, end=' ')        # uitvoer: 97 98 99 0 255
```

### Aantekeningen

---

---

---

---

---

---

---

Het data-type **bytes** is (net als een string) immutable. Daarnaast bestaat ook het data-type **bytearray** dat een mutable variant is van het **bytes** type. Beide types leveren bij een `for`-iteratie een reeks van integer waarden; elk byte heeft een waarde van 0-255 (inclusief).

In Python 2 is er geen onderscheid tussen type **str** en type **bytes**.

## Strings en bytes (2)

### Conversies

- van **bytes** naar **str**: `bytes.decode()`
- van **str** naar **bytes**: `str.encode()`



voorbeeld: € is U+20AC

```

prijsss = '10\u20ac'                                # string: 10€
prijsu8 = prijsss.encode('utf-8')

print('String: %s, bytes: %s\n' % (prijsss, prijsu8))
# String: 10€, bytes: b'10\xe2\x82\xac'

prijs2 = prijsu8.decode('utf-8')      # string: 10€
  
```

- van ene naar andere encoding: via Unicode (string) tussenstap

```

prijsl9 = prijsu8.decode('utf-8').encode('latin9')
print(prijsl9)
# b'10\x94'
  
```

CC BY-NC-ND 4.0 | CC BY-NC-ND 4.0 | v12f – h05 –

@ at computing

### Aantekeningen

In het voorbeeld op de slide wordt de `.encode` method gebruikt om de string `prijsss` om te zetten naar UTF-8 geëncodeerde bytes:

```
prijsu8 = prijsss.encode('utf-8')
```

Een conversie met de `bytes` functie was ook mogelijk geweest:

```
prijsu8 = bytes(prijsss, encoding='utf-8')
```

Voorbeeld van omzetting naar ASCII geëncodeerde bytes:

```

prijsasc = prijsss.encode('ascii', 'ignore')           → 10
prijsasc = bytes(prijsss, encoding='ascii', errors='ignore')

prijsasc = prijsss.encode('ascii', 'replace')          → 10?
prijsasc = bytes(prijsss, encoding='ascii', errors='replace')
  
```

In Python 2 is er geen onderscheid tussen type `str` en type `bytes`.

## Python 2 strings

In Python 2 is een string (type `str`) een reeks bytes, analoog aan het type `bytes` in Python 3. Door een `u` vóór de open-quote op te nemen kun je in Python 2 een Unicode string definiëren, analoog aan een “normale” string in Python 3.

Bij Python 2 mag je een `b` vóór de open-quote opnemen, maar die wordt genegeerd. Het kan zinvol zijn om deze toch te plaatsen als het expliciet om een byte-reeks gaat; het conversie-programma `2to3` zal deze `b` dan overnemen in de Python 3 source code.

Een voorbeeld van een Python 2 programma:

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-

a = 'ab€'
print a, len(a)

u = u'ab€'
print u, len(u)

print

for i in a:
    print hex(ord(i))
print

for i in u:
    print hex(ord(i))
```

De uitvoer van dit programma is<sup>1</sup>:

```
ab€ 5
ab€ 3

0x61
0x62
0xe2
0x82
0xac

0x61
0x62
0x20ac
```

De weergave van de variabele `a` en `u` als string is gelijk: op de derde positie staat het

---

1. De functie `ord` laat de getalswaarde van een teken zien; de functie `hex` vertaalt deze integer naar een hexadecimale string.

euro-teken.

In de variabele `a` is dit euro-teken uit de source file ingelezen als 3 losse bytes (UTF-8) en wordt als 3 losse bytes weer naar de terminal gestuurd; die terminal interpreteert deze bytes weer als een euro-teken (terminal is op UTF-8 ingesteld). De lengte 5 voor variabele `a` geeft aan dat het om 3 losse bytes gaat (afgezien van de tekens `\n` die ook meegeteld). De `for`-iteratie geeft ook 5 losse bytes.

In de Unicode variabele `u` is het euro-teken uit de source file ingelezen als 3 bytes UTF-8 en wordt opgeslagen als 1 Unicode teken. De lengte 3 geeft dit ook aan. De `for`-iteratie geeft 3 losse (multi-byte) waarden voor de 3 tekens.

Python 2 ondersteunt — in tegenstelling tot Python 3 — ook impliciete conversies:

```
$ python2
>>> a = u'hello ' + 'world'
>>> a
u'hello world'
```

De string `world` wordt nu opgewaardeerd naar een Unicode string, analoog aan een `int` die wordt opgewaardeerd naar een `float` ten tijde van een optelling bij een `float`. Echter, deze *impliciete* conversies verlopen altijd via een ASCII tussenstap:

```
>>> valuta = u'€'.encode('UTF-8')
>>> valuta
'\xe2\x82\xac'
>>> a = u'10' + valuta
UnicodeDecodeError: 'ascii' codec can't decode byte 0xe2 in position 0:
ordinal not in range(128)
```

Een *expliciete* conversie is hier wel mogelijk:

```
>>> a = u'10' + valuta.decode('UTF-8')
>>> a
u'10\u20ac'
```

Ergo:

In Python 3 is Unicode het uitgangspunt en zo hoort het ook.

In Python 2 is ASCII het uitgangspunt en dat levert de nodige complicaties.

## Python 2 files en encoding

Bij het openen van een file (functie `open`) kun je in de mode de `b` toevoegen voor “binary”. Hiermee forceer je dat er geen carriage-return bytes worden verwijderd bij lezen of toegevoegd bij schrijven; zonder de `b` wordt dit namelijk wél gedaan. Verder heeft de `b` geen consequenties: in alle gevallen levert de method `.read()` (en consorten) een *string* op (ook bij “binary”) en wordt er geen rekening gehouden met encoding.

Stel dat je in Python 2 een file leest met UTF-8 data:

```
$ python2
>>> f = open("promo1.txt")           open kent geen encoding parameter!
>>> b = f.readline()
>>> b
'Sale: sm\xc3\xb8rrebr\xc3\xb8d voor 2\xe2\x82\xac\n'
>>> type(b)
<type 'str'>

>>> u = b.decode('UTF-8')
>>> u
u'Sale: sm\xf8rrebr\xf8d voor 2\u20ac\n'
>>> type(u)
<type 'unicode'>
```

De variabele `b` is een *string* en dat zou ook zo geweest zijn als we de file “binary” hadden geopend en met de method `.read()` alle bytes hadden gelezen. Een aparte stap is hier nodig om deze string naar de Unicode variabele `u` om te zetten.



# **Student-notities**

## **De programmeertaal Python**

06. Module 1 — Data-types in detail: dictionaries, tuples en sets



Nijmegen



## Dictionaries (1)

Dictionary: verzameling key-value paren

```

d1 = {}                      # lege dictionary
d2 = {'Jan':42, 'Marie':38}   # gevulde dictionary

d1["hoi"] = "Bonjour"         # maak entry aan
s = d1["hoi"]                 # s='Bonjour'
s = d1["hallo"]               # Exception!

if "hoi" in d1: ....          # True
if "Bonjour" in d1: ....      # False

d1["ja"] = "Oui"
d1["nee"] = "Non"

for k in d1: print(k, d1[k])   # vraag alle keys op
d1[(1, [1,2],"aa")] = 5       # Exception!

```

Sleutels

- niet wijzigbaar
- toegestaan: getal, string, tuple, frozenset (dus list en set niet)

CC BY-NC-ND 4.0 | v12f-h06 - 1

@ at computing

## Aantekeningen

---

Vanaf versie 2.7/3.1 kunnen de key-value paren in een dictionary ook gecreëerd worden met *dictionary comprehension*. Enkele voorbeelden:

```

d5 = {i:i*i for i in range(2, 7)}
      → {2: 4, 3: 9, 4: 16, 5: 25, 6: 36}

d6 = {j:True for j in 'aeiou'}
      → {'o': True, 'e': True, 'u': True, 'i': True, 'a': True}

```

## Dictionaries (2)

Toepasbaar voor

- associatieve arrays (conversietabellen)

```
d = {}  
d['hoi'] = 'bonjour'
```

- flexibele lijsten en arrays (indexen niet per se aaneengesloten en zelfde type)

```
naam = {}  
naam[10**100] = 'googol'  
naam[3.141592653589] = 'pi'  
naam[(0, 0)] = 'oorsprong'
```

- gedeeltelijk gevulde matrices

```
m = {}  
m[(4, 1, 7)] = 42  
m[(9, 0, 5)] = 88
```

- records (classes wellicht beter – later meer)

```
piet = {}  
piet['naam'] = 'piet puk'  
piet['leeftijd'] = 11
```

CC BY-NC-ND 4.0 | v12f - h06 - 2

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Dictionaries (3)

Meer details over dictionaries

```
a = {'eerste':3, 'tweede':99}

# elementen opvragen:
b = a.get('hoi')           # b=None
b = a.get('hoi', 17)        # b=17
b = a.setdefault('hoi',17)   # b=17, a['hoi']=17 view: 'kijkgaatje' naar sleutels

ka = a.keys()              # dict_keys(['tweede', 'hoi', 'eerste'])
va = a.values()             # dict_values([99, 17, 3]) view: 'kijkgaatje' naar waarden

for k in a:
    print('bij', k, 'hoort', a[k])      # willekeurige volgorde
                                         # of volgorde van toevoegen

for k,v in a.items():
    print('bij', k, 'hoort', v)

# in gesorteerde volgorde
for k in sorted(a):          # returnt gesorteerde lijst
    print('bij', k, 'hoort', a[k])  # print in sleutelvolgorde

del a['hoi']                 # verwijder element met sleutel 'hoi'
a.clear()                    # maak hele dictionary leeg: a == {}

CC BY-NC-ND 4.0 | v12f-h06-3
```

### Aantekeningen

Vanaf versie 2.7/3.1 is een geordende dictionary beschikbaar in de collections module. Sleutels worden dan geleverd in volgorde van invoegen. Een voorbeeld:

```
import collections
od = collections.OrderedDict()
od['pear'] = 3
od['banana'] = 7
od['apple'] = 2
od['melon'] = 5
od['banana'] = 4      # wijziging verandert de volgorde niet

for k in od: print(k, end=' ')      → pear banana apple melon
```

## Dictionaries en EAFP

Verschillende manieren van indexeren in dictionary

```
if key in mijndict:  
    a = mijndict[key]  
else:  
    a = "gewenste default-waarde"
```

Versus EAFP

```
try:  
    a = mijndict[key]  
except Exception:  
    a = "gewenste default-waarde"
```

Versus `get` method

```
a = mijndict.get(key, "gewenste default-waarde")
```

Versus `setdefault` method (maakt entry eventueel aan)

```
a = mijndict.setdefault(key, "gewenste default-waarde")
```

CC BY-NC-ND 4.0 | v12f – h06 – 4

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

---

## Dictionary methods

Verkort overzicht:

<code>len(a)</code>	aantal elementen in a
<code>del a[k]</code>	gooi a[k] weg
<code>v = a.pop(k)</code>	gooi a[k] weg en return waarde
<code>a.clear()</code>	maak heel a leeg
<code>a.copy()</code>	maak kopie van a
<code>if k in a: if k not in a:</code>	test of k in a zit
<code>a.items()</code>	dict_items: view naar a's (key, value) paren
<code>a.keys()</code>	dict_keys: view naar a's keys
<code>a.values()</code>	dict_values: view naar a's values
<code>a.get(k[, x])</code>	a[k] als k in a, anders x
<code>a.setdefault(k[, x])</code>	a[k] als k in a, anders x (zet entry in dict)
<code>k, v = a.popitem()</code>	returnt en verwijdert willekeurig (key, value) paar

## Aantekeningen

---



---



---

Versie 2	Versie 3
<code>if a.has_key('hallo'):</code>	<code>if 'hallo' in a:</code>
<code>a.iteritems()</code>	<code>a.items()</code>
<code>a.iterkeys()</code>	<code>a.keys()</code>
<code>a.itervalues()</code>	<code>a.values()</code>

In versie 2 bestaat de method `has_key` die inmiddels “deprecated” is.  
 De `iter...` method-varianten worden niet meer ondersteund in versie 3. De niet-`iter...` method-varianten geven in versie 2 *lists* terug in plaats van *views*.

## Ritsen met `zip`, dictionary construeren met `dict`

Met `zip()` meerdere sequences samenritsen tot één iterator (serie tuples)

```
l = ['a', 'b', 'c']
u = ['A', 'B', 'C']

for paar in zip(l, u):
    print(paar[0], "heeft als hoofdletter", paar[1])

print(list( zip(l, u) )) # [('a', 'A'), ('b', 'B'), ('c', 'C')]
```

Met `dict()` dictionary maken van geritste lijst

```
low2up = dict( zip(l, u) )

print(low2up)          # {'a': 'A', 'c': 'C', 'b': 'B'}

for low in low2up:
    print(low, 'heeft als hoofdletter', low2up[low])
```

CC BY-NC-ND 4.0 | v12f – h06 – 6

 at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

---

---

## Tuples (1)

Tuples: lijken op lists, maar niet wijzigbaar (immutable)

```
s = (1, 2, 3, 'ho')      # 4 elementen
t = (1, (5, 7), 7)      # 3 elementen
u = t[0]                  # 1
v = t[1]                  # (5, 7)
w = t[0:2]                # (1, (5, 7))
x = t[1:]                 # ((5, 7), 7)

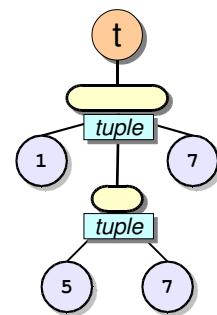
if 5 in t:                # False
...
if 5 in t[1]:              # True
...

for el in s:
    print(el)

t = 1, 5, 7
```

CC BY-NC-ND 4.0 | v12f - h06 - 7

haakjes mag je weglaten, mits syntactisch eenduidig



@ at computing

### Aantekeningen

---



---



---



---



---



---

Versie >= 2.6 en versie 3
t.index(7) # index van waarde 7 → 2
t.count(7) # aantal keren waarde 7 → 1

## Tuples (2)

```

i = ("ha")           # instinker: 'ha' !!
                     # (haken mogen altijd)
j = 3*i             # 'hahaha'

u = ("ha",)          # ('ha', )
v = 3*u              # ('ha', 'ha', 'ha')

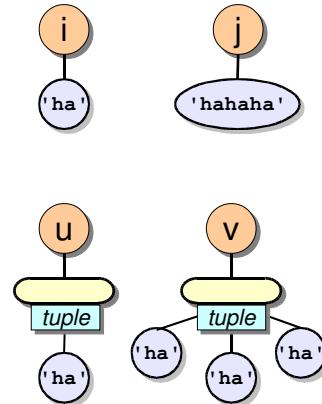
v[1] = 'ho'          # Exception!

w = tuple('hah')     # ('h', 'a', 'h')

l = [1, 2, 3, 5, 7]
t = tuple(l)          # (1, 2, 3, 5, 7)

a = 4
b = 3
(c, d) = (a,b)        # c=4,   d=3
c, d = a,b            # c=4,   d=3
b, a = a,b            # waarden omwisselen

```



Haken zijn optioneel bij tuples

at computing

### Aantekeningen

---



---

Python 3 biedt ook de mogelijkheid van *starred assignment*:

```

a, b = 3, 7           # a=3, b=7
a, b = 3, 7, 11       # Exception: aantal waarden links en rechts ongelijk
a, *b = 3, 7, 11       # a=3, b=[7, 11]

```

Door een \* voor de variabele-naam te plaatsen, wordt die variabele een *list* met daarin alle overblijvende waarden.

Nog een voorbeeld:

```
eerste, *tussen, laatste = range(0, 16, 3)
```

De variabele eerste bevat nu 0 (integer), de variabele laatste bevat 15 (integer) en de variabele tussen bevat [3, 6, 9, 12] (list).

## Set en frozenset (1)

Set: verzameling van *immutable* waarden

- bevat alleen unieke waarden
- geen indices (geen gedefinieerde volgorde)
- waarden eventueel uitlezen via `for` loop
- frozenset niet wijzigbaar

```
a = set(['a', 'b', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a'])    of
a =      {'a', 'b', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a'}
           # {'a', 'c', 'b', 'r', 'd'}
if 'z' in a: ...      # False

b = {'a', 'l', 'a', 'c', 'a', 'z', 'a', 'm'}
     # {'a', 'l', 'c', 'z', 'm'}

x = a - b            # letters in a die niet in b zitten
                      # set(['r', 'd', 'b'])

x = a | b            # letters in a of in b:
                      # set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
```

CC BY-NC-ND 4.0 | v12f – h06 – 9

### Aantekeningen

Ook andere *immutable* (hashable) data-types kun je gebruiken als elementen in een set, zoals:

```
set(['peter', 'john', 'anne'])
set([1, 3, 5, 7, 9])
```

Versie <= 2.6	Versie 2.7, versie >= 3.1
set([1,2,3,4])	set([1,2,3,4]) of {1,2,3,4}

In versie 2.7 en 3.1 (of hoger) geldt de notatie {} nog steeds voor een lege dictionary en set() voor een lege set!

## Set en frozenset (2)

Set: verzameling

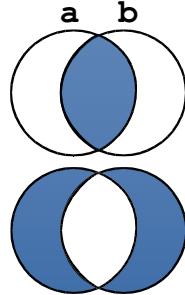
```
a= {'a', 'b', 'a', 'c', 'a', 'd', 'a', 'b', 'r', 'a'
b= {'a', 'l', 'a', 'c', 'a', 'z', 'a', 'm'}

x = a & b      # letters in a én in b:
                # set(['a', 'c'])
x = a ^ b      # letters in óf a óf b:
                # set(['r', 'd', 'b', 'm', 'z', 'l'])

a.add('z')      # element erbij
a.update('wxy') # meer erbij
                # set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'x', 'z'])

a.remove('x')   # 'x' eruit:
                # set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'z'])

f = frozenset(['a', 'l', 'a', 'c', 'a', 'z', 'a', 'm'])
f.remove('z')   # Exception!
```



CC BY-NC-ND 4.0 | v12f – h06 – 10

@ at computing

### Aantekeningen

Met de `{...}` notatie is het ook mogelijk om de elementen in een set te genereren met *set comprehension*.

*Voorbeeld:*

Stel dat je elementwaarden genereert door de `choice` functie uit de `random` module 10 maal aan te roepen (deze functie kiest een willekeurig element uit een sequence). Dan levert dat bij een list 10 elementen op die niet uniek hoeven te zijn; bij een set kun je minder dan 10 elementen overhouden die wel uniek zijn:

```
l = [random.choice('abcdefghijklmnopqrstuvwxyz') for i in range(10)]
print(l)      → ['h', 'n', 's', 'c', 'm', 'w', 'k', 'e', 'v', 'v']

s = {random.choice('abcdefghijklmnopqrstuvwxyz') for i in range(10)}
print(s)      → {'a', 'f', 'i', 'k', 'm', 'l', 'q', 's', 'v'}
```



# **Student-notities**

## **De programmeertaal Python**

07. Module 1 — Functies



Nijmegen



## Functies

Goede programmeur = luie programmeur

- schrijf nooit iets vaker op dan nodig...
- meermalen zelfde code nodig? dan naam geven!

```
def telop(a, b):  
    s = a + b  
    return s
```



- gebruik van functie `telop`

```
x = telop(4, 5)      # x is nu 9
```

- ander gebruik van functie `telop`

```
x = telop('co', 'la')  # x is 'cola'
```

### Aantekeningen

---

---

---

---

---

---

---

---

## Gebruik van functies

Uit functie `telop()` blijkt

- functies kunnen parameters mee krijgen
- parameters hebben geen type in functie-definitie
- functie niet geïnteresseerd in type, zolang operaties op type mogen

```
def telop(a, b):
    s = a + b
    return s
```

```
x = telop(1, 1)          # x is nu 2
x = telop('12', '34')    # x is "1234"      (concateneer strings)
x = telop([1,3], [5,7])  # x is [1,3,5,7]  (concateneer lists)
x = telop((1,3), (5,7)) # x is (1,3,5,7) (concateneer tuples)
x = telop('12', 34)      # Exception!
```

- *schrijf functie zo algemeen mogelijk!*  
dus niet:

```
def telop(a, b):
    s = a + b + 0
```

hierdoor werkt bijv. string-“optelling” niet meer...

- bepaald type argument verwacht? verifieer met functie `type()`

CC BY-NC-ND 4.0 | v12f – h07 – 2

@ at computing

### Aantekeningen

---



---



---



---



---



---

Als je wilt verifiëren of een argument het verwachte type heeft, kun je de functie `type()` gebruiken.

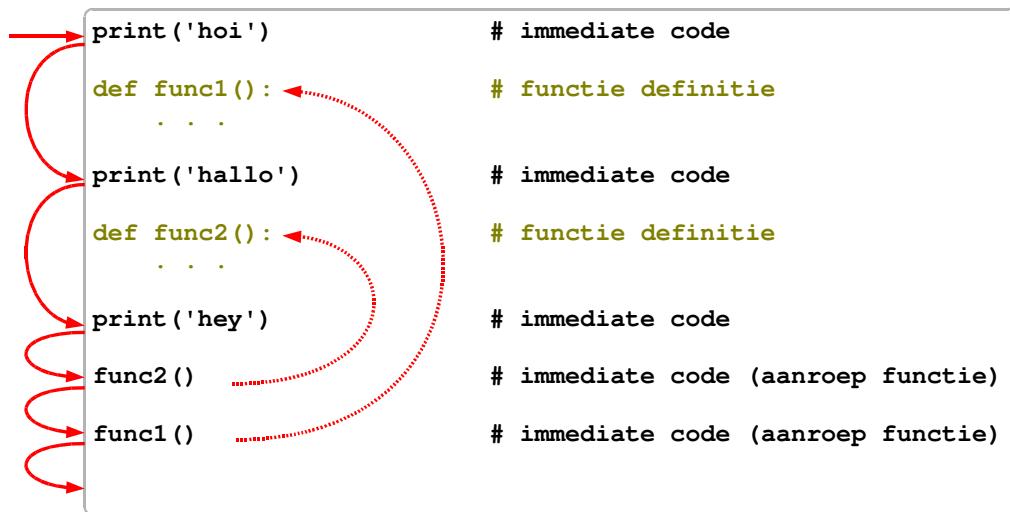
Stel dat je zeker wilt weten dat argument `a` een integer is:

```
if type(a) == int:
    ...
```

Uiteraard kun je ook andere data types checken (`str`, `list`, `dict`, `tuple`, ...).

## Programma flow (1)

Functie moet bekend zijn vóór aanroep



Beter:

- eerst *alle* functies, daarna *alle* immediate code
- dan geldt wel: programma “op z'n kop” ☺

CC BY-NC-ND 4.0 | v12f – h07 – 3

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

## Programma flow (2)

- Alternatief:
- plaats main code zelf ook in functie  
conventie: **main()**
  - roep **main()** aan op laatste regel van source code

```
def main():
    print('hoi')          # main code
    print('hallo')        # main code
    print('hey')          # main code

    func2()              # main code (aanroep functie)

    func1()              # main code (aanroep functie)

    def func1():          # functie definitie
        ...

    def func2():          # functie definitie
        ...

main()
```

A red circle highlights the `main()` function at the top of the code. A dotted arrow points from the bottom of the `main()` function back up to the start of the function definition.

CC BY-NC-ND 4.0 | v12f – h07 – 4

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

Voordeel van de werkwijze op de slide is dat de main code nu vooraan staat in de source code file, maar ook dat je globale variabelen vermeidt.

In hoofdstuk 8 “Module 1 — Modules” zullen we zien dat het gebruikelijk is om (herbruikbare) functies in een module-file te zetten en deze file met een import in te lezen.

## Gebruiksaanwijzing van functie (1)

Documentatie vaak pas achteraf gemaakt

- slechte aanwijzingen
- slecht commentaar
- Python conventie: gebruiksaanwijzing in code als *docstrings*

```
def telop(a, b):
    """
        tel twee dingen bij elkaar op
        a:      het eerste ding
        b:      het tweede ding
        return: het resultaat

        kan exception genereren als a en b
        niet opgeteld kunnen worden
    """

    s = a + b      # hiero de optelling
    return s
```

moet aan begin  
van functie staan

*Docstrings consequent gebruiken!*

CC BY-NC-ND 4.0 | v12f – h07 – 5

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Gebruiksaanwijzing van functie (2)

### Opvragen van docstring

```
>>> help(telop)
Help on function telop in module __main__:

telop(a, b)
    tel twee dingen bij elkaar op
        a:      het eerste ding
        b:      het tweede ding
    return: het resultaat

    kan exception genereren als a en b
    niet opgeteld kunnen worden
(END)                                     (melding van pager)
```

[CC BY-NC-ND 4.0 | v12f – h07 – 6](#)

@ at computing

### Aantekeningen

---

Docstrings worden opgeslagen onder de naam `__doc__` in de namespace van een functie. Ook onder die naam kan de docstring opgevraagd worden:

```
>>> print(telop.__doc__)

tel twee dingen bij elkaar op
    a:      het eerste ding
    b:      het tweede ding
return: het resultaat

kan exception genereren als a en b
niet opgeteld kunnen worden
```

## Parameters

Twee soorten parameters

- positie
- naam

```
def setcursor(regel, kolom):
    statements...
```

Mogelijke aanroepen

```
setcursor(13, 26)           # resp. regel en kolom

setcursor(regel=13, kolom=26)
setcursor(kolom=26, regel=13)
```

Kies dus duidelijke namen voor parameters!

Ook mogelijk: “starred argument” voor iterable

```
posi = [13, 26]
setcursor(*posi)      # effectief: setcursor(13, 26)
```

 at computing

### Aantekeningen

Bij een *starred argument* worden de elementen van de iterable als afzonderlijke positionele parameters overgedragen aan de aangeroepen functie.

Een voorbeeld waaruit de toevoeging van de \* duidelijk blijkt:

```
z = ["dit", "is", "een", "onzin"]
print("Let op:", z, sep='+')          # Let op:+['dit', 'is', 'een', 'onzin']
print("Let op:", *z, sep='+')         # Let op:+dit+is+een+onzin
```

Een starred argument is ook mogelijk voor named parameters in combinatie met een dictionary:

```
posi = {'kolom':26, 'regel':13}

setcursor(**posi)      # effectief: setcursor(kolom=26, regel=13)
```

## Default waarde voor parameters

Parameters kunnen default waarde krijgen

```
def setcursor(regel=0, kolom=0):
    """
    ...
    """
    statements
```

Mogelijke aanroepen

```
setcursor()          # regel=0, kolom=0
setcursor(2)         # regel=2, kolom=0
setcursor(2, 45)     # regel=2, kolom=45
setcursor(kolom=16)  # regel=0, kolom=16
setcursor(regel=9)   # regel=9, kolom=0
```

CC BY-NC-ND 4.0 | v12f – h07 – 8

@ at computing

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Variabel aantal parameters (1)

Functie-definitie kan aangeven: er kunnen nog meer parameters komen

```
def f(x, y, *restje):
    """
    .....
    """
    statements
```

Resterende parameters: in tuple **restje**

```
f(1, 2)           # x=1, y=2, restje=()
f(1, 2, 3)        # x=1, y=2, restje=(3, )
f(1, 2, 3, 4, "nogwat") # x=1, y=2, restje=(3, 4, 'nogwat')
```

CC BY-NC-ND 4.0 | v12f – h07 – 9

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Variabel aantal naam-parameters (2)

Ook variabel aantal benoemde parameters

```
def f(x, y, **restje):
    """
    deze functie krijgt drie parameters mee:
    x en y:    positionele parameters
    restje:   dictionary met naam/waarde paren
    """
```

Aanroepen:

```
f(4, 5)                      # x=4, y=5, restje={}

f(1,2,kleur='rood',dikte=7) # x=1, y=2,
                            # restje={'kleur':'rood', 'dikte':7}
```

Na benoemde parameter alleen nog andere benoemde parameters:

```
f(1, kleur='rood', dikte=7, 2)
```

SyntaxError!

CC BY-NC-ND 4.0 | v12f – h07 – 10

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Parameters: recapitulatie

Parameter-definitie:

```
def f(x, y, *positioneelrest, **naamrest):  
    """  
    ....  
    """  
  
    statements
```

Parameter-gebruik:

```
f(1,2)                  # x=1, y=2, positioneelrest=(), naamrest={}  
  
f(5, 10, 88, 16, kleur='rood', dikte=8)  
                      # x=5, y=10, positioneelrest=(88, 16)  
                      # naamrest={'kleur':'rood','dikte':8}  
  
f(32, kleur='paars', 10, 88)  
                      # SyntaxError: non-keyword arg  
                      # after keyword arg
```

CC BY-NC-ND 4.0 | v12f – h07 – 11

@ at computing

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

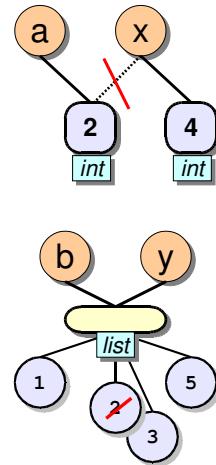
## Parameters: wijze van overdracht

Parameternaam *referentie* naar meegegeven waarde

- als meegegeven waarde
  - immutable: gedrag als “call by value”
  - mutable: gedrag als “call by reference”

```
def func(x, y):
    x = 4    # oorspr. waarde immutable, dus nieuwe x
    y[1] = 3  # oorspr. waarde mutable, dus in-place

a = 2
b = [1, 2, 5]
func(a, b)
print(a, b)  # 2  [1, 3, 5]
```



- wijziging voorkomen als
  - aanroeper: `func(a, b[:])`
  - aangeroepene: `def func(x, y):  
 y = y[:]`

CC BY-NC-ND 4.0 | v12f – h07 – 12

@ at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---

## Returnwaarde

Returnwaarde mag tuple zijn

```
import random

def geefstel():
    heren = [ 'jan', 'piet', 'klaas', 'erik', 'kees', 'wim' ]
    dames = [ 'els', 'anja', 'erika', 'anna', 'loes', 'kim' ]

    heer = random.choice(heren) # kies random element uit list
    dame = random.choice(dames) # kies random element uit list

    return (dame, heer)

paar      = geefstel()          # paar[0]: dame, paar[1]: heer
vrouw, man = geefstel()
```

Geen returnwaarde: impliciet **None** teruggeven

CC BY-NC-ND 4.0 | v12f – h07 – 13

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

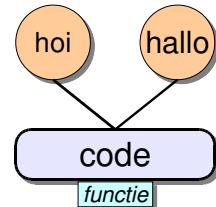
## Functie is een variabele!

Inhoud van functie: code

- functie kan toegekend worden

```
def hoi():
    print("bonjour")

hoi()          # toont 'bonjour'
hallo = hoi   # hallo nu ook functie
hallo()        # aanroep toont ook 'bonjour'
```

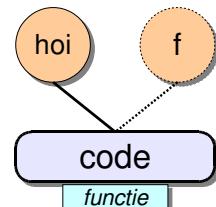


- functie kan als parameter gebruikt worden

```
def herhaal(f, n):
    for i in range(n):
        f()

def doeit():
    print('tot ziens')

herhaal(doeit, 3)    # 3 keer regel 'tot ziens'
herhaal(hoi, 5)      # 5 keer regel 'bonjour'
```



CC BY-NC-ND 4.0 | v12f – h07 – 14

at computing

### Aantekeningen

Op de slide wordt een voorbeeld gegeven van een functie die een andere functie als argument krijgt.

Het is ook mogelijk dat een functie een functie als returnwaarde teruggeeft:

```
def myfunc(x):
    def subfunc1(s, n):
        ...
        def subfunc2(s, n):
            ...
            if x < 5: return subfunc1
            else:       return subfunc2

    a = myfunc(7)
    a('hoi', 5)           of ineens:           myfunc(7)('hoi', 5)
```

## Functie als parameter: `map` (1)

Stel: functie-aanroep nodig voor elke waarde uit reeks

- functie om kwadraat uit te rekenen

```
def kwadraat(n):
    return n*n
```

- reeks waarden waarvoor kwadraat benodigd is

```
vals = [17, 21, 13, 42]
```

- gewenst: som van kwadraten

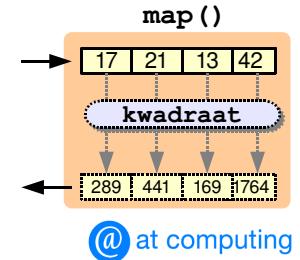
```
som = 0

for i in vals:          # eigen loop
    som += kwadraat(i)
```

of

```
som = sum( map(kwadraat, vals) )
```

CC BY-NC-ND 4.0 | v12f – h07 – 15



### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



---



---

In Python 2 geeft de `map` functie een *list* terug.

## Functie als parameter: `map` (2)

### Iterator `map`

- meegegeven functie aangeroepen voor ieder element van sequence
- genereert reeks waarden
- verwerking
  - ▷ zelf waarden langslopen in `for`-iteratie
  - ▷ conversie naar `list()` of `tuple()`, of sommeren met `sum()`
  - ▷ ....
- nog een voorbeeld

```
def klinkerweg(s):
    ret=""
    for c in s:
        if c not in "aeiouAEIOU":
            ret += c
    return ret

woorden = ['dit', 'is', 'fantastisch']

wrdn = tuple( map(klinkerweg, woorden) ) # ('dt', 's', 'fntstsch')
```

CC BY-NC-ND 4.0 | v12f – h07 – 16

### Aantekeningen

Stel dat je de som wilt uitrekenen van een lijst getallen in een comma-separated string. De `.split()` method geeft een list met strings (getallen), dus daarop kun je niet de `sum` functie loslaten. Je kunt dan via de `map` iterator alle getallen omzetten naar integers:

```
total = sum( map(int, "123,456,789".split(',')) )
```

Je mag ook meerdere sequences meegeven aan de `map` iterator maar dan moet de meegegeven functie ook meer argumenten accepteren:

```
def telop(a, b): return a+b

a = [17, 23, 54, 99]
b = [33, 53, 11, 0]
r = list( map(telop, a, b) ) # [50, 76, 65, 99]
```

## Functie als parameter: `filter`

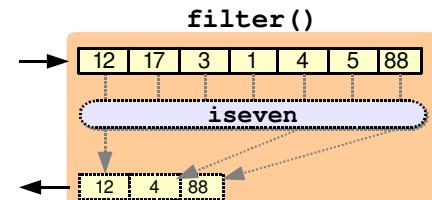
### Iterator `filter`

- meegegeven functie aangeroepen voor ieder element van sequence
- genereert reeks waarden
  - waarden waarvoor aangeroepen functie `True` returnt
  - verwerking in `for`-iteratie, door conversie of door sommeren
- voorbeeld

```
def iseven(v):
    return v%2 == 0 # True of False

a = [12, 17, 3, 1, 4, 5, 88]
e = list( filter(iseven, a) )

print("even:", e)      # [12, 4, 88]
```



CC BY-NC-ND 4.0 | v12f – h07 – 17

@ at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



---



---

In Python 2 geeft de `filter` functie een *list* terug.

## Anonieme functie: lambda

### Lambda expressie

- kan simpele functie vervangen (bestaande uit `return` statement)
- behoeft geen naam: *anonieme functie*
- gebruiken op plek waar referentie naar functie wordt verwacht
- voorbeelden

```
vals = [17, 21, 13, 42]
som = sum( map(lambda n: n*n, vals) )
```

aanroep-  
parameter      return-  
waarde

```
a = [12, 17, 3, 1, 4, 5, 88]
e = list( filter(lambda v: v%2==0, a) )
print("even:", e)      # [12, 4, 88]
```

CC BY-NC-ND 4.0 | v12f – h07 – 18

@ at computing

### Aantekeningen

---



---



---



---

Je kunt een lambda expressie ook weer een naam geven, zodat je zo'n expressie kunt uitvoeren alsof het een functie is:

```
power = lambda getal, macht=2: getal ** macht

a = power(7, 3)          # 343
a = power(7)             # 49
```

## Geneste functies

Functie is soort variabele, dus ook binnen functie te definiëren

```
def func():
    " demo functie "
    a=1
    b=2

    def hulfunc():
        c=3
        d=4
        return c + d

    a = 5
    b = hulfunc()

func()          # Wel OK
hulfunc()       # Niet OK!
```

- alleen `func()` kan `hulfunc()` zien en aanroepen
- wie heeft toegang tot variabelen als `a` en `c`?

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Scope regels

## LEGB: voorrangsregels

## Local (binnen functie zelf)

Enclosing function local (ook via `nonlocal`)

**Global** (immediate code of via `global`)

## Built-in

```
a=1
def f():
    print(a)      # 1

    def g():
        print(a) # 1
    g()
f()
```

```
a=1
def f():
    a=2          # local
    print(a)    # 2

    def g():
        a=5      # local
        print(a) # 5
    g()
    print(a)    # 2

f()
print(a)      # 1
```

```
def f():
    a=2
    print(a)      # 2
    def g():
        nonlocal a
        a=3
        print(a)      # a van f()
    g()
    print(a)      # 3
f()
```

```
a=1
def f():
    global a, b
    a=2          # globale a
    b=3          # globale b
f()
print(a, b) # 2 3
```

CC BY-NC-ND 4.0 | v12f – h07 – 20

## Aantekeningen

Het keyword `nonlocal` wordt niet ondersteund in Python versie 2.

# Generators

Functie die reeks waarden genereert: *generator* (soort iterator)

- binnen **for** lus te gebruiken
  - bij iedere aanroep volgende waarde teruggeven met **yield**
    - geef waarde terug naar aanroeper
    - pauzeer functie
    - hervat functie bij nieuwe aanroep

```
def afscheid():
    yield 'houdoe'
    yield 'ajuuus'
    yield 'doei'
    return

for kreet in afscheid():
    print(kreet)

# output: houdoe
#         ajuus
#         doe
```

```
def kwadraten(max):
    for i in range(1, max):
        yield i ** 2
    return

for k in kwadraten(5):
    print(k)

# output: 1
#          4
#          9
#          16
```

CC BY-NC-ND 4.0 | v12f - h07 - 21



## Aantekeningen

---

---

---

---

---

---

---

---

---

# **Student-notities**

## **De programmeertaal Python**

08. Module 1 — Modules



Nijmegen



## Modules

### Programma

- kan bestaan uit vele functies
- sommige functies ook bruikbaar binnen andere programma's
- gemeenschappelijke functies samenvoegen in nieuwe file: *module*
- voorbeeld: **stringhulpjes.py**

```
def klinkerweg(s): .....
def medeklinkerweg(s): .....
def klinker2upcase(s): .....
def medeklinker2upcase(s): .....
```

- voorbeeld: **rekenhulpjes.py**

```
priem = (2,3,5,7,11,13,17,19,23,29)

def telop(a, b): .....
def kwadraten(max): .....
```

CC BY-NC-ND 4.0 | v12f - h08 - 1

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

## Modules gebruiken

Module laden: `import modulenaam`

- leest file in en voert alles uit (functie-definitie is ook statement!)
  - `def`'s
  - globale variabelen: toekenningen/definities
  - classes (later meer)
  - immediate code (code buiten functies)
- maar eenmaal per module
- functies of globale variabelen uit module expliciet aanduiden

```
import rekenhulpjes
print(rekenhulpjes.priem)
for k in rekenhulpjes.kwadraten(10):
    print(k)
```

```
from rekenhulpjes import kwadraten, priem
print(priem)
for k in kwadraten(10):
    print(k)
```

CC BY-NC-ND 4.0 | v12f - h08 - 2

@ at computing

## Aantekeningen

Ook mogelijk:

```
from rekenhulpjes import *
```

In dit geval worden alle namen geïmporteerd uit de module `rekenhulpjes`, behalve de interne namen die beginnen met één underscore (“weak names”).

Ook mogelijk:

```
import rekenhulpjes as rh
print(rh.priem)
```

In dit geval wordt een alias `rh` gemaakt om het latere typewerk wat te verkorten.

## Herkomst van modules

### Locatie van module-file

- **xxx.py** gezocht via directory-lijst in **sys.path**
  1. directory van top-level programma
  2. directories in environment variabele **PYTHONPATH**
  3. standaard directories
- dus lijst uit te breiden via environment variable **PYTHONPATH**

```
$ export PYTHONPATH=/usr/local/mypylib:/pytlib # UNIX voorbeeld  
$ mijnprog.py
```

```
$ python3  
>>> import sys  
>>> print(sys.path)  
['...', '/usr/local/mypylib', '/pytlib', '/usr/lib64/python3.3', ...]
```

top-level directory

```
> set PYTHONPATH=%PYTHONPATH%;C:\pytlib # Windows voorbeeld
```

grafische omgeving: My Computer → Properties → Advanced → Environment Variables

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Immediate code in module

Ook immediate code (geen onderdeel van functie) mag in module

Module kan aan variabele `__name__` “zien” of

- gestart ten gevolge van `import`: `__name__ == <modulenaam>`
- gestart als los programma: `__name__ == '__main__'`
  - vaak gebruikt om testcode uit te voeren
  - voorbeeld:

```
def kwadraat(n):  
    return n*n  
  
if __name__ == '__main__':      # test functie  
    if kwadraat(3) == 9:  
        print('PASS')  
    else:  
        raise Exception("kwadraat is rot")
```

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Docstrings in modules

Docstrings voor module en voor functies

```
rekenhulpjes.py
"""
Deze module bevat diverse
rekenkundige functies.
"""

def square(n):
    """
    Deze functie geeft het kwadraat
    terug van de meegegeven waarde.
    """
    return n*n

def telop(v1, v2):
    """
    Deze functie telt twee meegegeven
    waarden op en geeft uitkomst terug.
    """
    return v1+v2
```

```
$ python3
>>> import rekenhulpjes
>>> help(rekenhulpjes)
Help on module rekenhulpjes:

NAME
    rekenhulpjes

DESCRIPTION
    Deze module bevat diverse
    rekenkundige functies.

FUNCTIONS
    square(n)
        Deze functie geeft het kwadraat
        terug van de meegegeven waarde.

    telop(v1, v2)
        Deze functie telt twee meegegeven
        waarden op en geeft uitkomst terug.

FILE
    /..../..../rekenhulpjes.py
```

CC BY-NC-ND 4.0 | v12f – h08 – 5

@ at computing

## Aantekeningen

---



---



---



---



---



---



---



---



---



---



---

# **Student-notities**

## **De programmeertaal Python**

09. Module 2 — Inleiding classes



Nijmegen



# Object Oriëntatie

Tot dusver voornamelijk simpele data-types (**str**, **int**, **float**, ....)

Stel je wilt complexere items beschrijven, zoals:

- boek (titel, auteur, prijs, ISBN, ....)
  - persoon (voornaam, achternaam, geboortedatum, geslacht, ....)
  - bank rekening (rekeningnummer, eigenaar, saldo, kredietlimiet, ....)
  - ....

## Werkwijze

- zelf abstract data-type definiëren: *class*
  - variabele(n) van dat data-type: *instance of object*

## Aantekeningen

---

---

---

---

---

---

## Data abstraction

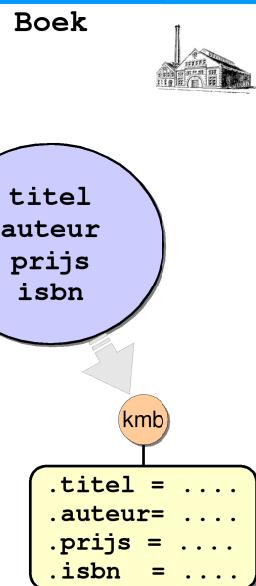
## Voorbeeld: data-type Boek

- definieer abstract data-type **Boek**  
dat losse *attributes* (variabelen) samenbundelt
    - ☞ **class**fabriek (mal) waarmee je instances 'produceert'
  - creëer variabele van abstract data-type **Boek**
    - ☞ **instance**

## bijvoorbeeld **kmb**

```
kmb.titel  = 'Koken met Berendien'  
kmb.auteur = 'Berendien'  
kmb.prijs   = 10.99      # correcte eenheid?  
kmb.isbn    = 123        # correct nummer?
```

- ▶ geen verificatie (bijv. ISBN moet bestaan uit 10 of 13 cijfers in 5 groepen)
  - ▶ wat als variabele-type in toekomst wijzigt (bijv. prijs als **int** i.p.v. **float**)



CC BY-NC-ND 4.0 | v12f – h09 – 2

## Aantekeningen

---

---

---

---

---

---

---

---

---

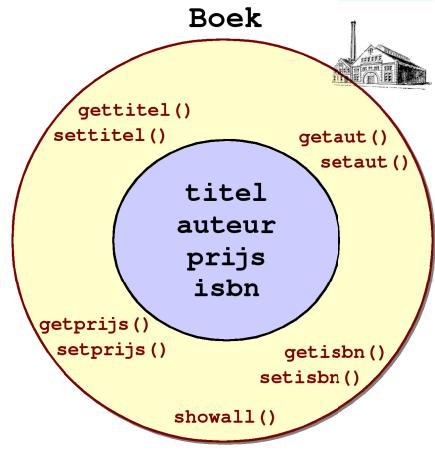
# Encapsulation

## Voorbeeld: data-type Boek

- voeg functies toe aan *class* om *attributes* te vullen/verkrijgen/bewerken  
 **methods**

mogelijkheid tot

- ▶ verificatie
  - ▶ type conversie
  - ▶ berekenen
  - ▶ andere bewerkingen...



bijvoorbeeld **kmb**

```
kmb.settitel('Koken met Berendien')
kmb.setaut('Berendien')
kmb.setprijs(10.99)      # conversie?
kmb.setisbn(123)         # afwijzen!
kmb.showall()            # toon attributen
```

CC BY-NC-ND 4.0 | v12f - h09 - 3

@ at computing

## Aantekeningen

---

---

---

---

---

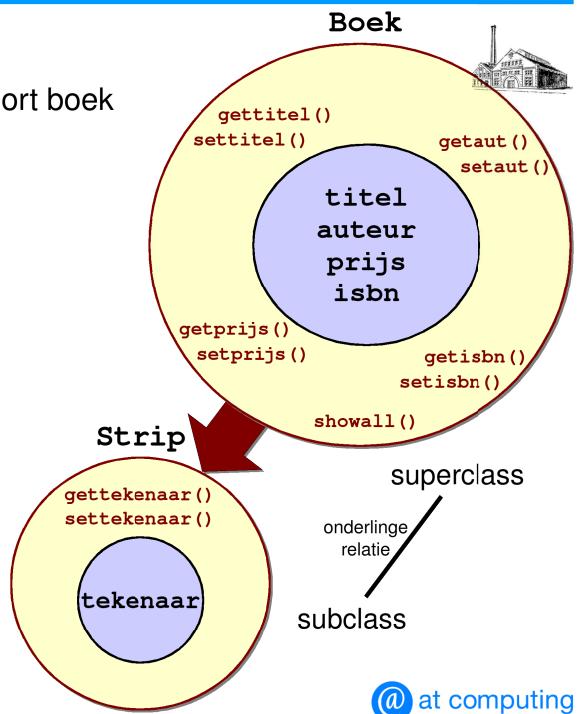
---

---

## Inheritance (1)

## Voorbeeld: data-type Boek

- stel extra attribute(s) voor bepaald soort boek voorbeelden:
    - stripboek (tekenaar)
    - reisgids (land/regio)
    - biografie (hoofdpersoon)
  - breid bestaande class niet uit, maar creëer afgeleide class
    -  **subclass**
    - erft attributes en methods van bestaande class (superclass)
    - voegt nieuwe attributes en methods toe



CC BY-NC-ND 4.0 | v12f - h09 - 4

## Aantekeningen

## Inheritance (2)

### Voorbeeld: data-type Boek

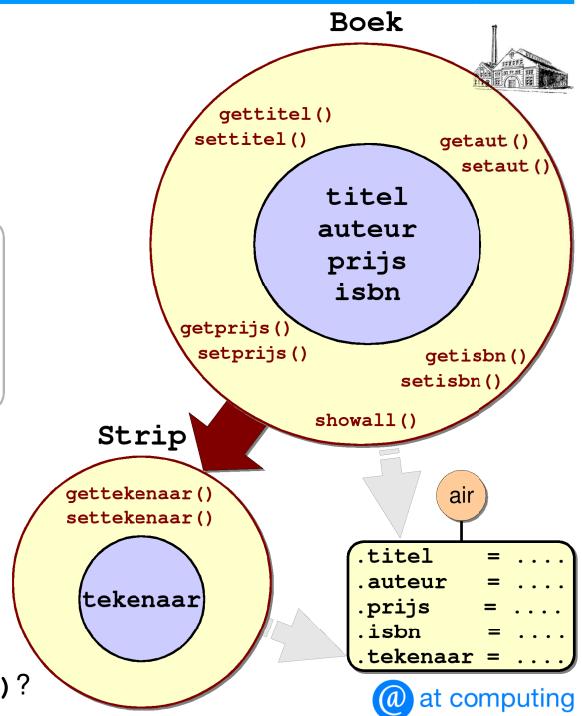
- instances van data-type **Boek** mogelijk, maar ook van data-type **Strip**

bijvoorbeeld **air** van data-type **Strip**

```
air.settitel('Asterix in Rome')
air.setaut('Goscinny')
air.setprijs(13.80)
air.setisbn(1234567890123)
air.settekenaar('Uderzo')
air.showall()
```

- bij aanroep van **settitel()**
  - ontbreekt in class **Strip**
  - dan gebruikt vanuit class **Boek**
- bij aanroep van **settekenaar()**
  - gebruikt vanuit class **Strip**
- maar wat toont aanroep **air.showall()**?

CC BY-NC-ND 4.0 | v12f-h09 – 5



### Aantekeningen

---



---



---



---



---



---



---



---

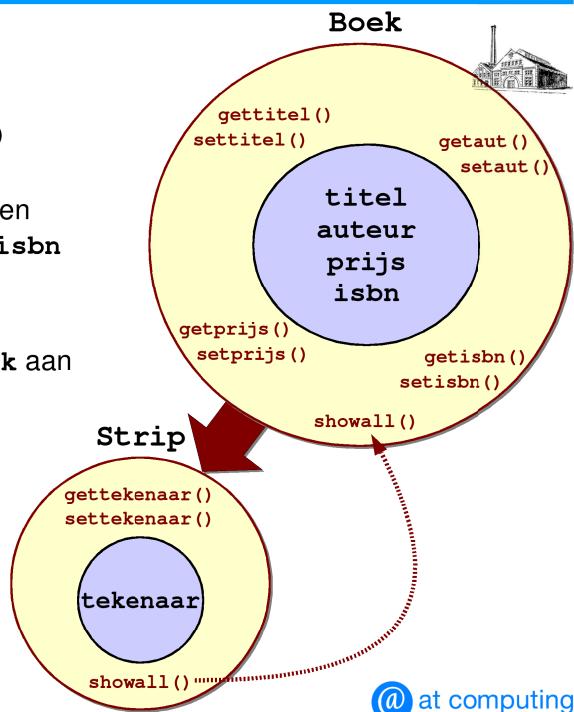


---

## Inheritance (3)

### Voorbeeld: data-type Boek

- mogelijkheid om method van superclass te overrulen in subclass, bijv. `showall()`
- method `showall()` in **Boek** kent alleen attributes **titel**, **auteur**, **prijs** en **isbn**
- method `showall()` in **Strip**
  - roept method `showall()` van **Boek** aan (hergebruik van code)
  - voegt na aanroep informatie over tekenaar toe



### Aantekeningen

---



---



---



---



---



---



---



---



---

## Polymorfisme en dynamic binding

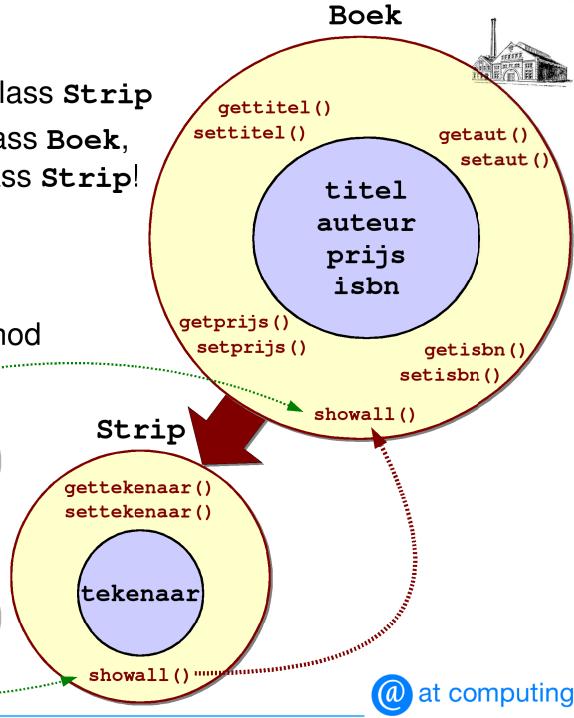
Voorbeeld: data-type **Boek**

- instance (object) van class **Boek** of class **Strip**
- wat je kunt doen met instance van class **Boek**, kun je ook doen met instance van class **Strip**!  
echter, niet andersom!
- type van instance bepaalt welke method wordt aangeroepen

`kmb.showall() # type Boek`

`air.showall() # type Strip`

CC BY-NC-ND 4.0 | v12f-h09 - 7



@ at computing

## Aantekeningen

---



---



---



---



---



---



---



---



---



---

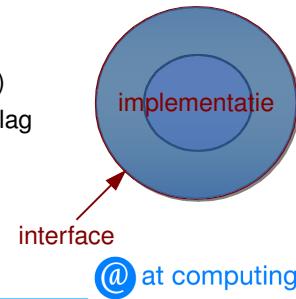
## Recapitulatie (1)

### Programmeertalen

- *procedureel*
  - instructies lineair uitvoeren
  - gebruik van *functies* staat centraal (hergebruik van code)
  - benadering vaak top-down
- *object-georiënteerd*
  - gebruik van *objecten* staat centraal
    - groeperen van meerdere data-elementen in abstract data-type
    - vastleggen van bewerkingen op data-elementen via methods
  - scheiding tussen *interface* en *implementatie*
    - interface: geboden functionaliteit (aanroep van methods)
    - implementatie: interne werking van methods en wijze van opslag implementatie moet kunnen wijzigen zonder wijziging van interface
  - benadering bottom-up

Python bruikbaar in beide hoedanigheden!

CC BY-NC-ND 4.0 | v12f – h09 – 8



@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Recapitulatie (2)

### Aspecten object-oriëntatie

- *data abstraction*
  - data-elementen groeperen in nieuw data-type: *abstract data-type*
- *encapsulation*
  - bewerkingen toevoegen aan abstract data-type: *methods*
- *inheritance*
  - soms behoeft om functionaliteit uit te breiden of te specialiseren, maar niet ten koste van juiste werking van oorspronkelijke data-type
  - nieuw abstract data-type dat functionaliteit van ander type overneemt en uitbreidt
- *polymorphism*
  - één interface voor objecten van verschillend data-type:  
method `gettitle()` bruikbaar voor object van type **Boek** én van type **Strip**
- *dynamic binding*
  - kiezen van passend interface voor specifiek data-type:  
andere method `showall()` voor object van type **Boek** dan van type **Strip**

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

# **Student-notities**

## **De programmeertaal Python**

10. Module 2 — Python classes



Nijmegen



## Data abstractie

## Definieer class

```
class Ballon(object):
    def setmax(self, maxv):
        self.maxvol = maxv
        self.curvol = 0
    def blaasop(self, vol):
        if self.curvol+vol > self.maxvol:
            raise Exception("Knal!")
        else:
            self.curvol += vol
    def loopeleeg(self):
        self.curvol = 0
    def hoevol(self):
        return self.curvol
    def toon(self):
        print("inhoud: %d/%d" %
              (self.curvol, self.maxvol))
```

## Maak instances van class

```
b = Ballon() # maak Ballon

c = Ballon() # feestje: nog een

b.setmax(20) # b's curvol/maxvol

c.setmax(10) # c's curvol/maxvol

b.blaasop(18) # blaas b op tot 18

b.toon()      # inhoud 18/20

v = b.hoevol()

b.blaasop(3) # Exception: Knal!
```

- eerste parameter **Ballon**-functie: “huidige instantie” (traditie: **self**)

CC BY-NC-ND 4.0 | v12f – h10 – 1



## Aantekeningen

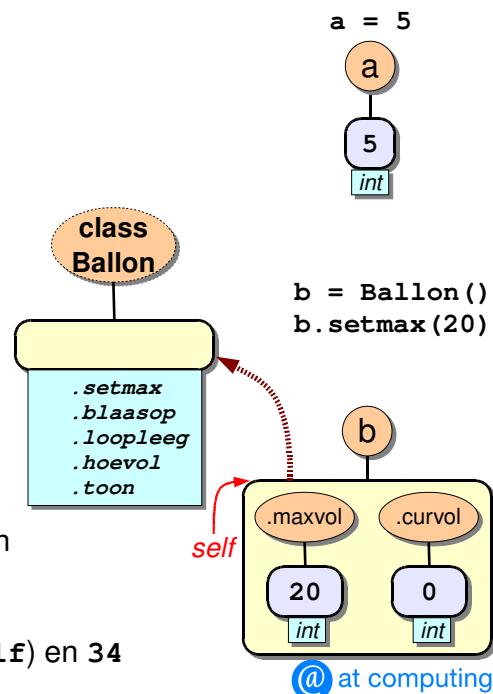
Ook voor een class kan de documentatie worden opgenomen in de source code zelf, analoog aan een functie: de string die volgt op de kopregel van de class geldt als docstring.

## Class inhoud

Class heeft

- attributen
  - variabelen zoals `maxvol` en `curvol`
  - ieder instantie krijgt set eigen attributen  
(iedere `Ballon` heeft eigen `curvol`)
- methods (functies)
  - aangeroepen voor specifieke instantie, zoals `b.blaasop(34)`
  - eerste parameter van method altijd referentie naar instantie
  - aanroep `b.blaasop(34)` is gelijk aan aanroep `Ballon.blaasop(b, 34)`

`blaasop()` krijgt parameters `b (self)` en `34`



CC BY-NC-ND 4.0 | v12f – h10 – 2

@ at computing

## Aantekeningen

---



---



---



---



---



---



---



---



---

## Constructie van object

Zonder aanroep van `setmax()`

- nieuwe ballon – na `b=Ballon()` – kent `.maxvol` en `.curvol` niet, dus
- iedere aanroep van andere method levert exception, behalve `loopleeg()`

Dus liever automatische initialisatie bij creatie: method `__init__()`

```
class Ballon(object):
    def __init__(self, maxv):
        self.maxvol = maxv
        self.curvol = 0
    def blaasop(self, vol):
        if self.curvol+vol > self.maxvol:
            raise Exception("Knal!")
        else:
            self.curvol += vol
    def loopeeg(self):
        self.curvol = 0
    def hoevol(self):
        return self.curvol
    def toon(self):
        print("inhoud: %d/%d" %
              (self.curvol, self.maxvol))
```

```
# maak Ballon aan
# aanroep __init__
b = Ballon(20)

# opblazen gaat goed
b.blaasop(17)

# Exception: geen param
c = Ballon()
```

CC BY-NC-ND 4.0 | v12f - h10 - 3

@ at computing

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Encapsulatie

Encapsulation in Python niet afgedwongen

```
b = Ballon(50)      # __init__ aanroep
b.toon()            # output: inhoud: 0/50

b.curvol = 100     # rechtstreeks variabele in instance aanpassen
b.toon()            # output: inhoud: 100/50
```

Bij voorkeur niet doen:

gebruik *interface blaasop()* in plaats van *implementatie curvol*

Drempel opwerpen: naam in class (“name mangling”)

```
class Ballon(object):
    def __init__(self, maxv):
        self.__maxvol = maxv
        self.__curvol = 0
    ....
```

```
b = Ballon(50)
b.__Ballon__curvol = 100
```

CC BY-NC-ND 4.0 | v12f – h10 – 4

@ at computing

## Aantekeningen

---



---



---



---



---



---

Andere programmeertalen bieden vaak de mogelijkheid om “private methods” te definiëren binnen een class, die alleen bedoeld zijn om vanuit een andere method in diezelfde class aangeroepen te worden.

Python biedt geen ondersteuning voor private methods. Echter, method-namen mogen ook beginnen met twee underscores, waardoor een aanroep van “buitenaf” wat moeilijker wordt gemaakt.

## Inheritance (1)

Inheritance: als class niet 100% bevult

- erf ervan
- pas aan naar eigen behoefte

```
class FigBallon(Ballon):
    """ Ballon met vorm """

    def __init__(self, maxv, vorm):
        """ initialiseer """
        super().__init__(maxv)
        self.vorm = vorm

    def toon(self):
        super().toon()
        print("vorm:", self.vorm)
```

```
# maak gewone ballon
gb = Ballon(30)

# maak figuurballon
fb = FigBallon(9, "mickey")

# method uit Ballon
fb.blaasop(5)

# method uit FigBallon
fb.toon()

# methods uit Ballon
gb.blaasop(11)
gb.toon()
```

- functie **super**: bepaalt superclass van huidige class en geeft eerste parameter door (**self**)
- alles wat met **Ballon** (superclass) kan, kan ook met **FigBallon** (subclass)

CC BY-NC-ND 4.0 | v12f – h10 – 5

@ at computing

### Aantekeningen

De `super` functie zoekt uit wat de superclass is van de huidige class (in dit geval is de superclass `Ballon`); vervolgens wordt van die superclass de betreffende method aangeroepen.

Als je de `super` functie niet gebruikt, ziet de aanroep van de `__init__` method van `Ballon` (vanuit de `__init__` method van `FigBallon`) er als volgt uit:

```
Ballon.__init__(self, maxv)
```

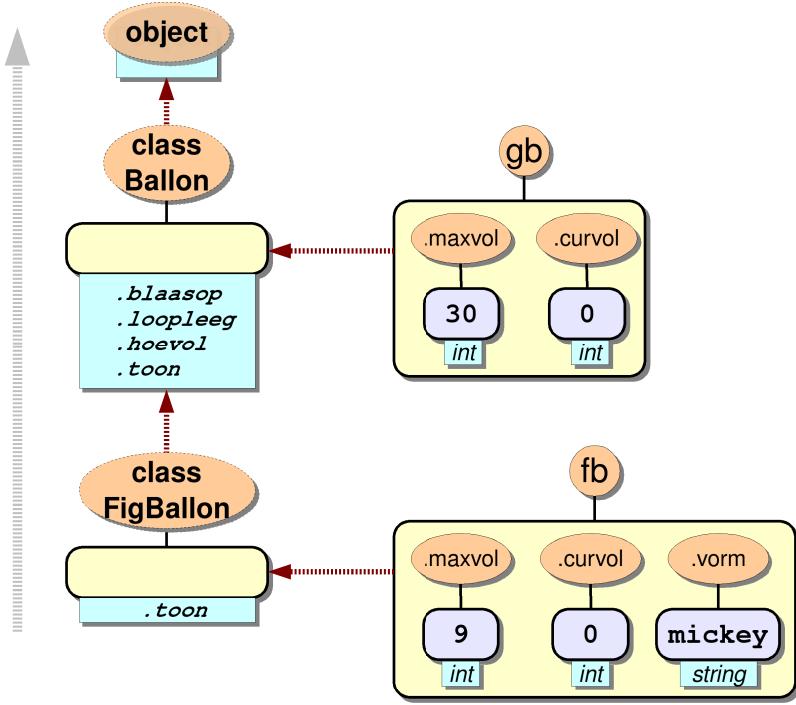
Versie 2	Versie 3
<code>super(FigBallon, self).__init__(maxv)</code> <code>super(FigBallon, self).toon()</code>	<code>super().__init__(maxv)</code> <code>super().toon()</code>
<code>old-style: class Ballon:</code> <code>new-style: class Ballon(object):</code>	<code>new-style: class Ballon:</code> <code>new-style: class Ballon(object):</code>

In Python 2 wordt onderscheid gemaakt tussen *old-style classes* (bijvoorbeeld “`class Ballon:`”) en *new-style classes* (bijvoorbeeld “`class Ballon(object):`”). In geval van een old-style class is de `super` functie niet bruikbaar.  
In Python 3 zijn beide vormen van class definities (impliciet) new-style.

## Inheritance (2)

zoekrichting voor methods  
(method in subclass overruelt method in superclass)

CC BY-NC-ND 4.0 | v12f – h10 – 6



## Aantekeningen

---

---

---

---

---

---

---

---

## Polymorfisme en dynamic binding

Python kiest zelf juiste functie

```
def doemetballon(b):
    """ krijgt Ballon mee """
    n = b.hoevol()           # uit Ballon
    b.toon()                 # uit Ballon of FigBallon ????
    
gb = Ballon(20)
fb = FigBallon(37, "hart")

doemetballon(gb)          # aanroep: Ballon.toon()

doemetballon(fb)          # aanroep: FigBallon.toon()
```

Eigenlijk al bekend

- functie `telop()` accepteert integers, strings, lists, ... dus alles wat '+' snapt
- flink gebruik van maken

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Aandachtspunten

### Opmerkingen

- naamgeving volgens PEP8
  - class in CapWords-notatie: **Ballon**, **FigBallon**
  - instance in kleine letters: **gb**, **fb**
- locatie
  - class-definities bij voorkeur in module(s)
  - instance-allocatie in programma-code zelf
- standaard data-types zijn ook classes (**str**, **int**, **list**, **dict**, ....)

```
$ python3
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__',
 '__iadd__', '__imul__', '__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
 '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear',
 'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

CC BY-NC-ND 4.0 | v12f – h10 – 8

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

---

---

## Speciale methods (1)

Naast `__init__` nog meer speciale methods....

### `__str__`

aangeroepen als string-variant van instantie nodig is  
bijvoorbeeld: `str(mijnballon)`

```
print(mijnballon)  
b = '%s' % mijnballon
```

```
class Ballon(object):  
    ...  
    def __str__(self):  
        return "inhoud: %d/%d" % \  
            (self.curvol, self.maxvol)
```

```
a = Ballon(20)  
a.blaasop(16)  
  
print(a) # inhoud: 16/20
```

### `__repr__`

aangeroepen als *formelere* string-variant van instantie  
nodig is, d.w.z. string van waaruit oorspronkelijke instantie  
geregenereerd kan worden

voor `repr(mijnballon)` of '`%r`' % `mijnballon`, maar  
ook voor `str(mijnballon)`, `print(mijnballon)`, '`%s`'  
als `__str__` ontbreekt

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Speciale methods (2)

### Implementatie van operatoren

\_\_add\_\_ aangeroepen voor optelling: type-specifiek rekenen

```
a = Ballon(40) # a is 0/40
a.blaasop(20) # a is 20/40
b = Ballon(50) # b is 0/50
b.blaasop(10) # b is 10/50
c = a + b      # c is 30/90
```

```
class Ballon(object):
    ...
    def __add__(self, ander):
        nb = Ballon(
            self.maxvol+ander.maxvol)
        nb.blaasop(
            self.curvol+ander.curvol)
        return nb
```

- kan voor alle operatoren

<u>__sub__</u>	voor -
<u>__mul__</u>	voor *
<u>__truediv__</u>	voor /
<u>__mod__</u>	voor %
... . . .	

### Aantekeningen

---



---



---



---

#### Python versie 2:

Voor de deling (operator `/`) kunnen twee speciale methods aangeboden worden: de `__div__()` voor de “classic” deling en de `__truediv__()` voor de “true” deling. Bij een integer deling zal de *classic* deling een integer waarde als resultaat geven ( $7/2=3$ ), terwijl de *true* deling een float waarde als resultaat geeft ( $7/2=3.5$ ). De *true* deling wordt geactiveerd als de `division` optie van de `__future__` module is aangeschakeld, en anders wordt de *classic* deling gebruikt.

#### Python versie 3:

Er wordt altijd gekozen voor de *true* deling (method `__truediv__()`).

## Speciale methods (3)

Stel: liever *integer* optellen bij ballon (verhogen .curvol)

<pre>a = Ballon(40) # a is 0/40 a.blaasop(20) # a is 20/40 b = a + 10    # b is 30/40</pre>	<pre>class Ballon(object):     ...     def __add__(self, ander):         nb = Ballon(self.maxvol)         nb.blaasop(self.curvol+ander)         return nb</pre>
---	---

roep `__add__()` aan

Echter: wat als operanden verwisseld worden?

<pre>a = Ballon(40) # a is 0/40 a.blaasop(20) # a is 20/40 b = 10 + a    # b is 30/40</pre>	<pre>class Ballon(object):     ...     def __radd__(self, ander):         return self.__add__(ander)     or:     return Ballon.__add__(self, ander)     or:     return self+ander</pre>
---	---

roep `__add__()` aan: "snapt" andere operand niet!

roep `__radd__()` aan

CC BY-NC-ND 4.0 | v12f-h10-11

@ at computing

### Aantekeningen

---



---



---



---

Uiteraard is het verstandig om in de `__add__` en `__radd__` methods te checken welk data type de andere operand heeft. Als je het data type niet "snapt" (zoals de `__add__` method van het data type `int` onze ballon niet begrijpt), geef je de waarde `NotImplemented` terug.

Voor elke operator kan een r-versie (`__rsub__()`, `__rmul__()`, `__rdiv__()`, `__rtruediv__()`, `__rmod__()`, etc) worden geïmplementeerd om de situatie af te handelen dat "ons" type operand aan de rechterkant van de operator staat.

## Speciale methods (4)

**\_\_getitem\_\_** aangeroepen voor indexeren met []  
werkt ook met **for** lus

```
class Achteruit(object):
    """ding dat achteruit telt"""

    def __init__(self, d):
        self.data = d

    def __getitem__(self, index):
        return self.data[-1-index]
```

```
x = Achteruit("string")
c = x[0]    # "g"
c = x[1]    # "n"

for c in x:
    print(c, end=' ')
print()      # uitvoer:
            # g n i r t s

a = Achteruit( range(6) )

for i in a:
    print(i, end=' ')
print()      # uitvoer:
            # 5 4 3 2 1 0
```

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Ingebouwde types en inheritance

Inheritance: als class niet 100% bevult of als uitbreiding gewenst

- voorbeeld: element-rotatie gewenst voor *ingebouwde* class `list`

```
r = RotList([7, 33, 9, 5])

print(r)      # [7, 33, 9, 5]

r.reverse()  # [5, 9, 33, 7]
r.append(2)  # [5, 9, 33, 7, 2]

r.rrotate()  # [2, 5, 9, 33, 7]

v = RotList([1, 6])

x = r + v
y = r * 2
z = 3 * r
```

```
class RotList(list):
    def rrotate(self):
        self.insert(0, self.pop())

    def lrotate(self):
        self.append(self.pop(0))

    def __add__(self, val):
        nl = super().__add__(val)
        return RotList(nl)

    def __mul__(self, val):
        nl = super().__mul__(val)
        return RotList(nl)

    def __rmul__(self, val):
        return self*val
```

- alle niet-gedefinieerde methods (zoals `append`) erf je van `list`

CC BY-NC-ND 4.0 | v12f – h10 – 13

@ at computing

### Aantekeningen

In het voorbeeld op de slide, zou je wellicht verwachten dat de optelling (operator `+`) en vermenigvuldiging (operator `*`) van een `RotList` instantie prima afgehandeld kan worden door de `__add__()` en `__mul__()` methods van de `list` class. Echter, dit zijn methods die zelf een nieuwe instantie creëren als “resultaat-instantie” en die instantie zou dan van het type `list` worden (dus *niet* van `RotList`). Daarom hebben we “wrapper” methods `__add__()` en `__mul__()` aan de `RotList` class toegevoegd; zij roepen hun evenknie in de `list` class aan en zetten de “resultaat-instantie” alsnog om naar een `RotList` type.

Als er ook nog slices genomen worden van de variabele `r` (bijvoorbeeld `p = r[:]`), dan zou er om dezelfde reden ook een “wrapper” method aan de `RotList` class toegevoegd moeten worden:

```
def __getslice__(self, first, last):
    newslice = list.__getslice__(self, first, last)
    return RotList(newslice)
```

## Data in class

Class kan ook variabelen bevatten

- uitgevoerd bij definitie van class
- voorbeeld: aangemaakte instanties tellen

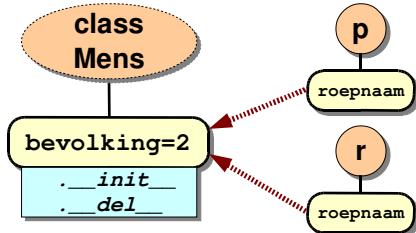
```
class Mens(object):
    bevolking = 0 # aantal

    def __init__(self, naam):
        Mens.bevolking += 1
        self.roepnaam = naam

    def __del__(self):
        Mens.bevolking -= 1
```

- moment van `__del__` aanroep niet gegarandeerd
- soorten variabelen

<code>bevolking</code>	class variabele
<code>roepnaam</code>	instance variabele



```
p = Mens('piet')
r = Mens('riet')

print(Mens.bevolking)      # 2

p = None # Mens-instantie nu weg

print(Mens.bevolking)      # 1

del r      # Mens-instantie nu weg

print(Mens.bevolking)      # 0
```

@ at computing

CC BY-NC-ND 4.0 | v12f - h10 - 14

## Aantekeningen

---



---



---



---



---



---



---



---



---



---

## Scope regels

In hoofdstuk 7 “Module 1 — Functies” hebben we de scope-regels belicht, ook bij referenties naar variabelen vanuit functies. Soortgelijke scope-regels zijn er ook voor instance en class variabelen.

Neem het volgende stukje code:

```
print(Mens.bevolking)    # 2
print(p.bevolking)        # 2
print(r.bevolking)        # 2

p.bevolking = 7
print(p.bevolking)        # 7
print(r.bevolking)        # 2
```

Bij de eerste referenties naar `p.bevolking` en `r.bevolking` blijkt dat `bevolking` geen instance variabele is, dus kijkt Python of het een class variabele is; in beide gevallen wordt de waarde in `Mens.bevolking` getoond.

Als de toekenning `p.bevolking = 7` wordt gedaan, wordt een *nieuwe instance variabele* in object `p` gemaakt met de waarde 7, analoog aan de toekenning `a=7` die een nieuwe lokale variabele `a` maakt in een functie. De instance variabele `bevolking` bestaat niet in object `r` dus de referentie `r.bevolking` laat nog steeds de waarde in `Mens.bevolking` zien!

# **Student-notities**

## **De programmeertaal Python**

11. Module 2 — Exceptions



Nijmegen



## Exception handling

### Motivatie exception handling

- met name foutafhandeling
  - motto: “Easier to Ask for Forgiveness than for Permission” (EAFP)
  - toch vooraf testen: dubbel werk (Python test zelf ook)

```
if i >= 0 and i < len(x):      # test 1
    b = x[i]                  # test 2 (impliciet)
else:
    b = 0
```

test vaker dan

```
try:
    b = x[i]                  # enige test
except Exception:
    b = 0
```

- levert betere plaats voor foutafhandeling
  - niet waar fout zelf optreedt, maar
  - daar waar fout goed af te handelen is!

CC BY-NC-ND 4.0 | v12f – h11 – 1

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

## Exception classes

### Soorten exceptions

- in smaken (als type), als classes geïmplementeerd
- standaard hiërarchie

```
BaseException
Exception
    ArithmeticError
        FloatingPointError
        OverflowError
        ZeroDivisionError
    AssertionError
    AttributeError
    BufferError
    EOFError
    ImportError
    LookupError
        IndexError
        KeyError
    MemoryError
    NameError
        UnboundLocalError
    ReferenceError
```

```
BaseException
Exception
    OSSError
        ....
        FileExistsError
        FileNotFoundError
        PermissionError
        ....
        RuntimeError
        NotImplemented
        SyntaxError
        IndentationError
        TabError
    SystemError
    TypeError
        ....
    KeyboardInterrupt
    SystemExit
```

[CC BY-NC-ND 4.0 | v12f – h11 – 2](https://creativecommons.org/licenses/by-nd/4.0/)

@ at computing

### Aantekeningen

De hiërarchie onder BaseException is bij Python 2 anders dan bij Python 3:

```
Exception
    StandardError
        ArithmeticError
            FloatingPointError
            OverflowError
            ZeroDivisionError
        AssertionError
        AttributeError
        BufferError
        EOFError
        EnvironmentError
            IOError
            OSSError
        ImportError
        LookupError
            IndexError
            KeyError
```

```
Exception
    StandardError
        MemoryError
        NameError
        UnboundLocalError
        ReferenceError
        RuntimeError
        NotImplemented
        SyntaxError
        IndentationError
        TabError
        SystemError
        TypeError
        ....
    KeyboardInterrupt
    SystemExit
```

## Exception classes afvangen

### Statement **try/except**

- selectie mogelijk: afhankelijk van exception andere actie

```
sals = {'piet':2000, ...}
mndu = {'piet': 170, ...}

try:
    uursal = salperuur(sals,mndu)

except KeyError:
    print("onbekende persoon!")

except ZeroDivisionError:
    print("iemand met nul uren!")

else:
    print("alles ging goed!")
```

```
def salperuur(sal, werktijd):
    """
    sal:      salaris per maand
    werktijd: uren per maand
    returnt:  uurloon
    """

    result = {}

    for p in sal:
        result[p] = sal[p]/werktijd[p]

    return result
```

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Exceptions en inheritance

Algemene exception afhandeling

```
try:  
    uursal = salperuur(sals,mndu)  
  
except Exception:  
    print("er ging iets fout!")
```

Specifieke exception afhandeling

```
try:  
    uursal = salperuur(sals,mndu)  
  
except ArithmeticError:  
    print("berekening ging fout!")  
except LookupError:  
    print("aanduiding ging fout!")  
except Exception:  
    print("kweenie wat fout is!")  
else:  
    print("alles ging goed!")
```

```
BaseException  
    Exception  
        ArithmeticError  
  
ZeroDivisionError  
....  
    LookupError  
    IndexError  
    KeyError  
....
```

@ at computing

Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Exception attributen

### Attributen van exception

- exception object “opvangen” met **as var**
  - attribuut **var.args**
  - tuple met meer info
  - inhoud varieert per type exception (vaak string)

```
try:
    uursal = salperuur(sals, mndu)

except ArithmeticError as e:
    print("berekening ging fout:", e.args)
except LookupError as e:
    print("aanduiding ging fout:", e.args)
```

Mogelijke uitvoer:

```
berekening ging fout: ('division by zero',)
of
aanduiding ging fout: ('eric',)
```

CC BY-NC-ND 4.0 | v12f – h11 – 5

@ at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



---

Versie < 2.6	Versie 3 (en >= 2.6)
except LookupError, a:	except LookupError as a:

## Exception forceren

Zelf exception forceren met **raise**

```
raise excnaam      raise dit type exception
raise excnaam(data) raise met extra data
raise             re-raise huidige exception
```

```
def salperuur(sal, tijd):
    if len(sal) != len(tijd):
        raise Exception('lengte ongelijk', len(sal), len(tijd))
    result = {}
    for pers in sal: result[pers] = sal[pers] / tijd[pers]
    return result
```

```
try:
    uursal = salperuur(sals, mndu)
except ....:
    ...
except Exception as ex:
    print(*ex.args)           # uitvoer: lengte ongelijk 37 38
```

CC BY-NC-ND 4.0 | v12f – h11 – 6

## Aantekeningen

---

---

---

---

---

---

---

---

---

## Eigen exception classes

## Eigen exception class met meer betekenis

- door onderscheid in type (naam)

```
class OnnozeleVout(Exception):
    pass

def myfunc():
    if ...:
        # er gaat iets fout...
        raise OnnozeleVout
```

```
try:  
    myfunc()  
except OnnozeleVout:  
    print("ging fout")  
else:  
    print("ging goed!")
```

- door onderscheid in opgeslagen waarden

```
class FoutInfo(Exception):
    def __init__(self, ernstcode):
        super().__init__()
        self.ernst = ernstcode

def myfunc():
    if ...:
        # er gaat iets fout...
        raise FoutInfo(5)
```

```
try:  
    myfunc()  
except FoutInfo as info:  
    if info.ernst > 7:  
        print("ging heel fout!")  
    else:  
        print("ging gewoon fout!")  
else:  
    print("ging goed!")
```

## Aantekeningen

---

---

---

---

---

---

---

---

---

## Niet-afgevangen exceptions

Als exception optreedt en

- niet binnen **try/except**
    - verlaat functie  
(lokale variabelen opruimen)
    - probeer passende **except** te vinden in aanroeper
  - geen passende **except**
    - verlaat functie  
(lokale variabelen opruimen)
    - probeer passende **except** te vinden in aanroeper
  - immediate code: geen passend
    - programma stopt met foutmelding

```
def f():
    raise FoutInfo(1)

def g():
    f()

def h():
    try:
        g()
    except FoutInfo as a:
        print("FoutInfo exception")
        if a.ernst > 7: # te heftig?
            raise # niet oplossen
```

geef exception alsnog door

## Aantekeningen

---

---

---

---

---

---

---

## Gegarandeerde afsluiting (1)

Soms moet codeblok altijd uitgevoerd worden

- of er nu exceptions optreden of niet (ook bij `re-raise`)
- afdwingen met `finally`
  - uitgevoerd vlak vóór verlaten `try` blok
  - ook bij `return` binnen `try` blok:

```
def a():
    try:
        print("we proberen...")
        f()
        print("het lukte!")
        return 5

    finally:
        print("dit doen we altijd...")
```

Uitvoer wordt:

we proberen...
het lukte!
dit doen we altijd...

of

we proberen...
dit doen we altijd...

### Aantekeningen

Een ander voorbeeld:

```
f = open("mijnfile", "r")

try:
    for regel in f:
        print(regel)
    ...
finally:
    f.close()
    del f
```

## Gegarandeerde afsluiting (2)

Alternatief voor **try/finally**: statement **with**

Algemene vorm

```
with expressie [as varx]:  
    statement(s)
```

- expressie levert object waarvan aan
  - begin `__enter__()` method wordt aangeroepen
  - einde `__exit__()` method wordt aangeroepen  
(ook als exception optreedt)
- voorbeeld

```
with open("mijnfile", "r") as f:  
    for regel in f:  
        print(regel, end="")
```

CC BY-NC-ND 4.0 | v12f – h11 – 10

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

Het `with` statement is beschikbaar vanaf Python versie 2.6 (in versie 2.5 te gebruiken met `from __future__ import with_statement`).

## Meerdere exceptions afvangen

Meerdere exceptions afvangen met één `except`

```
try:
    f()
except (LookupError, MemoryError, EOFError):
    print("Een LookupError, MemoryError of EOFError exception!")

    import sys
    if issubclass(sys.exc_info()[0], LookupError):
        ...
    ...
```

Voorkom eventueel ongecontroleerde crash bij exceptions

```
try:
    .... normale programma code ...
except Exception:
    import sys; import traceback
    print("Exception type:", sys.exc_info()[0])
    print("Exception value:", sys.exc_info()[1])
    print("Traceback:")
    traceback.print_tb( sys.exc_info()[2] )
```

 at computing

## Aantekeningen

In hoofdstuk 7 “Module 1 — Functies” opperden we de mogelijkheid om de main code te schrijven als de functie `main()` in plaats van als immediate code. In dat geval is het veel makkelijker om de gehele main code in te bedden in een `try... except ...` constructie:

```
try:
    main()
except Exception:
    ...
    ...
```

In plaats van `sys.exc_info()[0]` werd eerder de variabele `sys.exc_type` gebruikt en in plaats van `sys.exc_info()[1]` de variabele `sys.exc_value`. Het gebruik van deze globale variabelen geeft echter problemen bij multi-threaded applicaties. De *functie* `exc_info` returnt een tuple met drie waarden.

# **Student-notities**

## **De programmeertaal Python**

12. Module 2 — Python Standard Library



Nijmegen



## Python Standard Library

Goede programmeur = luie programmeur

- niet steeds opnieuw wiel uitvinden
- gebruik functies uit Python Standard Library
  - string hulpjes (met name reguliere expressies)
  - serialisatie (object persistentie)
  - OS gerelateerde faciliteiten
  - netwerk protocollen
  - HTML en XML parsing
  - multimedia ondersteuning
  - grafische interfaces (GUI)
  - multi-threading en multi-processing
  - debugging en profiling
- daarnaast vele andere bibliotheken uit *Python Package Index*



CC BY-NC-ND 4.0 | v12e – h12 – 1

@ at computing

### Aantekeningen

---

---

---

---

<https://docs.python.org>  
<https://pypi.python.org>

Python Standard Library (kies: Library Reference)  
Python Package Index

## Command line arguments and exit

Aanroep-parameters script opvragen: list `sys.argv`

```
import sys
print('er zijn', len(sys.argv)-1, 'parameters')
for arg in sys.argv[1:]:
    print(arg)
```

- commandonaam altijd `sys.argv[0]`

Eindig programma

```
import sys

sys.exit()          # eindig met exit code 0

if ....:
    sys.exit(1)      # eindig met exit code 1

if ....:
    sys.exit('Dit ging helemaal fout!')
        # print melding en exit code 1
```

CC BY-NC-ND 4.0 | v12e – h12 – 2

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Tijd

### Tijd-gerelateerde functies

```
import time

time.sleep(2.5)                      # 2.5 seconden wachttijd

cputime = time.process_time()         # CPU consumptie

epoch = time.time()                  # seconden sinds Epoch

lt = time.localtime( time.time() )    # epoch naar named tuple
print(lt)
time.struct_time(tm_year=2020, tm_mon=6, tm_mday=24, tm_hour=14,
tm_min=51, tm_sec=26, tm_wday=2, tm_yday=176, tm_isdst=1)

if lt.tm_mon > 6: ....

ft = time.strftime("%a, %d %b %Y %H:%M:%S", time.localtime())
print(ft)                            # uitvoer: 'Wed, 24 Jun 2020 14:51:44'
```

CC BY-NC-ND 4.0 | v12e – h12 – 3

@ at computing

### Aantekeningen

Met de functie `strftime` kun je de waarden die de functie `localtime` teruggeeft laten formatteren. Een kleine greep uit de formatteringscodes:

Code	Betekenis
%a	Locale's abbreviated weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%d	Day of the month as a decimal number [01,31].
%H	Hour (24-hour clock) as a decimal number [00,23].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%S	Second as a decimal number [00,61].
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.

## Filenamen opvragen

Wildcards afhandelen: `glob`

```
$ ls  
7.gif  aap.gif  aap.txt  card.gif
```

\* nul of meer willekeurige tekens  
? één willekeurig teken  
[...] één teken uit reeks

```
import glob  
  
names = glob.glob('[0-9].*')  # ['7.gif']  
  
names = glob.glob('aap.*')    # ['aap.gif', 'aap.txt']  
  
names = glob.glob('*gif')    # ['aap.gif', '7.gif', 'card.gif']  
  
names = glob.glob('?gif')    # ['7.gif']  
  
names = glob.glob('/etc/*')  # ['/etc/passwd', '/etc/group', ...]
```

CC BY-NC-ND 4.0 | v12e – h12 – 4

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

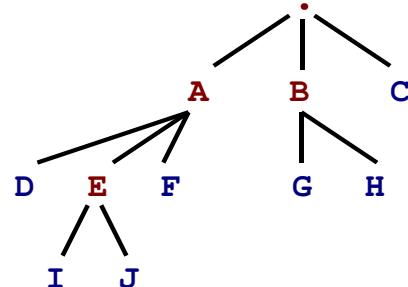
---

---

## Directory-boom: `walk` (1)

## Generator walk

- directory-boom doorlopen
  - werkwijze functie **walk**
    - aanroep: te doorlopen directory
    - return: tuple met in deze directory
      - huidige directory
      - list met subdirectories
      - list met non-directories



- voorbeeld:

<u>dize directory</u>	<u>subdirectories</u>	<u>non-directories</u>
.	['A', 'B']	['C']
./A	['E']	['D', 'F']
./A/E	[]	['I', 'J']
./B	[]	['G', 'H']

CC BY-NC-ND 4.0 | v12e - h12 - 5

@ at computing

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Directory-boom: **walk** (2)

Generator **walk** – voorbeeld: bepaal grootte van alle GIF files per directory

```
import os
import fnmatch
from os.path import join, getsize

for dezedir, subdirs, nondirs in os.walk('/usr'):
    som=0
    for naam in nondirs:
        if fnmatch.fnmatch(naam, '*.gif'):      # alleen GIF files
            padnaam = join(dezedir, naam)
            try:
                som += getsize(padnaam)
            except OSError:
                pass
    if som > 0:
        print('GIF files in', dezedir, 'gebruiken', som, 'bytes')

    if 'tmp' in subdirs:
        subdirs.remove('tmp')      # ga niet naar tmp subdir's
```

CC BY-NC-ND 4.0 | v12e – h12 – 6

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

---

## Activeer andere programma's

Gebruik van ander programma: module **subprocess**

- start ander programma en wacht op afloop

```
import subprocess

print('Inhoud van directory: ')
excode = subprocess.call('ls -l', shell=True)    óf
excode = subprocess.call(['ls', '-l'])
```

- lees data van ander programma via pipe

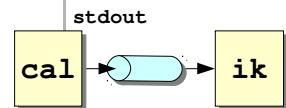
```
from subprocess import Popen, PIPE

cal = Popen("cal 1 2020", shell=True, stdout=PIPE,
            universal_newlines=True)
for line in cal.stdout:
    print( line.rstrip() )

cal.stdout.close()
excode = cal.wait()
```

- en vele andere mogelijkheden...

CC BY-NC-ND 4.0 | v12e – h12 – 7



@ at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---

De optie `universal_newlines=True` in het voorbeeld van `Popen` zorgt ervoor dat de gelezen regels vanuit de pipe van het type *string* zijn (zonder deze optie worden regels als *bytes* objecten teruggegeven).

## Netwerkdiensten

### Netwerk clients maken met `urllib.request`

```
import urllib.request

url  = "http://www.python.org/ftp/python/3.5.1/Python-3.5.1.tgz"
inet = urllib.request.urlopen(url)
ofil = open('pythonversie.tgz', 'wb')

while True:
    buf = inet.read(8192) # inet gedraagt zich als file
    if not buf: break
    ofil.write(buf)

web = urllib.request.urlopen("http://www.atcomputing.nl")

print( web.getcode() )      # 200 (HTTP status code)
print( web.geturl() )       # http://www.atcomputing.nl
print( web.info() )         # Date: Wed, 03 Feb 2016 10:13:15 GMT
                           # Server: Apache/2.2.3 (CentOS)
                           # Connection: close
                           # Content-Type: text/html
page = web.read()           # pagina als bytes object @ at computing
```

### Aantekeningen

In Python versie 2 is de functie `urlopen` onderdeel van de module `urllib` dus moet dan aangeroepen worden als:

```
inetw = urllib.urlopen("http://www.atcomputing.nl")
```

## Serialisatie: objecten omzetten naar byte-stream

Variabele omzetten en bewaren: **pickle**

```
import pickle

a = {}

# vul a
for i in range(100):
    a[i] = 2**i

tm = open("tweemachten", "wb")
pickle.dump(a, tm)
```

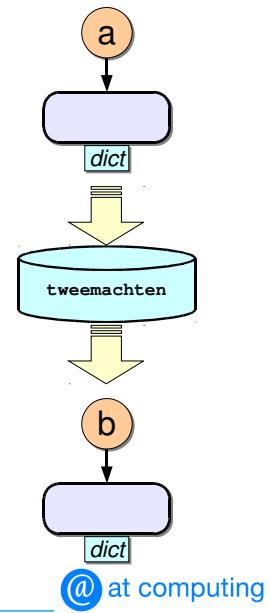
In ander Python programma:

```
import pickle

tm = open("tweemachten", "rb")
b = pickle.load(tm)

for i in b:
    print(i, 'heeft inhoud', b[i])
```

CC BY-NC-ND 4.0 | v12e - h12 - 9



### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



---



---

# **Student-notities**

## **De programmeertaal Python**

13. Module 2 — Reguliere expressies (bonus)



Nijmegen



---

Copyright © AT Computing 2020

Licensed under the Creative Commons BY-NC-ND License version 4.0 and up.

Versie: 12f

## Reguliere expressies – overzicht speciale tekens

Reguliere expressies: speciale tekens voor gebruik in zoekpatronen

.	één willekeurig teken
[abc]	één teken uit rij (hier: a, b of c)
[^abc]	één teken, juist niet uit rij (hier: niet a, b of c)
c*	nul of meer keren voorafgaand teken c
^ \$	begin resp. einde regel
\A \Z	begin resp. einde totale string (verschilt van ^ en \$ in multiline mode)
\b \B	woordgrens, niet-woordgrens
\d \D	cijfer [0-9], niet-cijfer [^0-9] postcode: \d\d\d\d * [A-Z] [A-Z]
\s \S	witruimte[ \r\n\t\f\v], niet-witruimte
\w \W	woordteken [a-zA-Z0-9_], niet-woordteken

### Aantekeningen

---



---



---



---



---



---



---



---



---



---

## Reguliere expressies – voorbeelden

### Voorbeelden

John Johnson likes to eat 14(!) sandwiches in each saloon

h.s	John <u>ohn</u> s <u>on</u> likes to eat 14(!) sandwich <u>hes</u> in each <u>saloon</u>
h[en]s	John <u>ohn</u> s <u>on</u> likes to eat 14(!) sandwich <u>hes</u> in each saloon
h[^e]s	John <u>ohn</u> s <u>on</u> likes to eat 14(!) sandwiches in each <u>saloon</u>
o*n	<u>ohn</u> <u>ohn</u> s <u>on</u> likes to eat 14(!) sandwich <u>ches</u> in <u>n</u> each <u>saloon</u>
h.*s	<u>ohn</u> <u>ohn</u> s <u>on</u> likes to eat 14(!) sandwich <u>ches</u> in each <u>saloon</u>
^Jo	<u>ohn</u> Johnson likes to eat 14(!) sandwiches in each saloon
on\$	John Johnson likes to eat 14(!) sandwiches in each saloon
John	<u>ohn</u> <u>ohn</u> s <u>on</u> likes to eat 14(!) sandwiches in each saloon
John\b	<u>ohn</u> Johnson likes to eat 14(!) sandwiches in each saloon
\d\d*	John Johnson likes to eat <u>14</u> (!) sandwiches in each saloon
\w	<u>ohn</u> <u>ohn</u> s <u>on</u> likes to eat <u>14</u> (!) sandwich <u>ches</u> in each <u>saloon</u>

### Aantekeningen

---

---

---

---

---

---

---

---

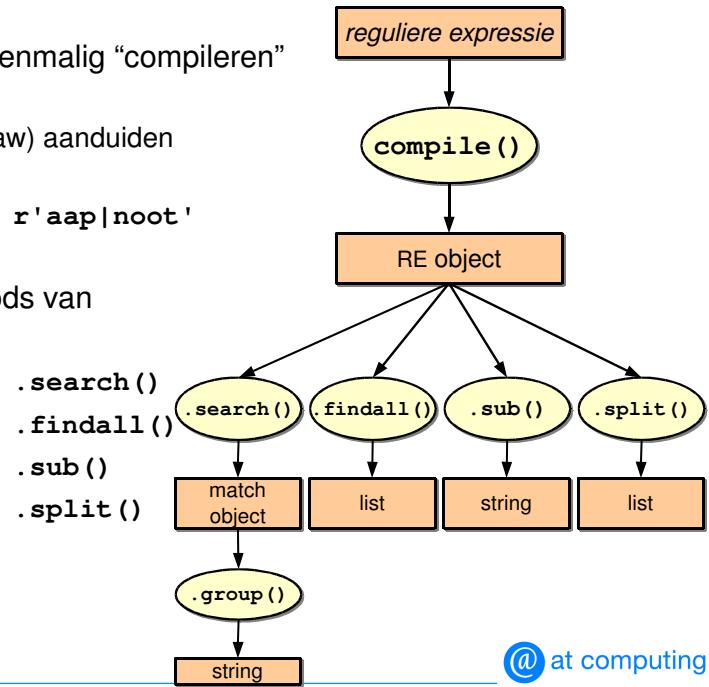
---

---

## Reguliere expressies – werkwijze

### Werkwijze in Python

- reguliere expressie string eenmalig “compileren” naar RE-object
  - expressie met `r'....'` (raw) aanduiden vanwege speciale tekens
  - vertical bar `|` is “of” : `r'aap|noot'`
- daarna (meermalen) methods van RE-object gebruiken
  - zoek eerste voorkomen:
  - zoek alle voorkomens:
  - zoek en vervang:
  - splits string:



CC BY-NC-ND 4.0 | v12f - h13 - 3

@ at computing

### Aantekeningen

---



---



---



---



---

De belangrijkste methods die we in dit hoofdstuk zullen zien:

- |                         |   |
|-------------------------|---|
| <code>.search()</code>  | zoek naar één match van RE binnen string    |
| <code>.findall()</code> | zoek naar alle matches van RE binnen string |
| <code>.split()</code>   | hak string in mootjes met RE als scheider   |
| <code>.sub()</code>     | zoek naar match van RE en vervang           |

## Reguliere expressies – werkwijze voorbeeld

Gebruik van `.search()` en `.findall()`

- voorbeeld: zoek postcode in string

```
import re

poco = r"\d\d\d\d *[A-Z][A-Z]"    # postcode reguliere expressie
line = "zijn adres is Scheerjeweg 14, 9999ZZ Louloene"
        RE-object
cre = re.compile(poco)           # compileer RE eenmalig

mat = cre.search(line)          # eventueel vaker bruikbaar
        match-object (None als niets gevonden) # op meerdere strings

if mat:                         # niet None: dan iets gevonden
    print( mat.group() )         # uitvoer: 9999ZZ

poli = cre.findall(line)        # eventueel meermalen
print(poli)                     # uitvoer: ['9999ZZ']
```

### Aantekeningen

---

---

---

---

---

---

---

---

---

## Reguliere expressies – herhalingen

Gezochte teken in string herhalen met { , }

$\{m, n\}$	minimaal $m$ keer, maximaal $n$ keer
$\{m\}$	precies $m$ keer
$\{m, \}$	$m$ keer of vaker
$\{ , n\}$	maximaal $n$ keer

## verkorte notaties:

$$\begin{array}{ll} \{0,1\} & \equiv ? \\ \{0,\} & \equiv * \\ \{1,\} & \equiv + \end{array}$$

voorbeeld: postcode

```
\d{4} ?[A-Z]{2}
```

- normaal: eerste match, vanaf daar langste  
met extra ?: eerste match, vanaf daar kortste

```
d = 'dit <b>moet</b> en <b>mag</b> je <b>nooit</b> vergeten!'
r = re.compile(r"<b>.*</b>")
l = r.findall(d) # returnt list met matches
print(l)          # [ '<b>moet</b> en <b>mag</b> je <b>nooit</b>' ]

r = re.compile(r"<b>.*?</b>")
l = r.findall(d)
print(l)          # [ '<b>moet</b>', '<b>mag</b>', '<b>nooit</b>' ]
```

CC BY-NC-ND 4.0 | v12f – h13 – 5

## Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Reguliere expressies – haakjes

### Ronde haken

- herhaling van expressie (groepering)

$r' ra(\text{ta})^+ '$



ra gevolgd door één of meer setjes ta  
(bijv. **ratatata**)

$r' [\text{A-Z}]^2 (-\text{d}^2)^2 '$  oud kenteken, bijv. **AZ-12-34**

- onthouden van matchdeel om
  - later opnieuw te matchen

$r' (.)^3 . \text{rotator} \backslash 1 '$



matcht op **rotator** en **leiwiel**  
(palindroom)

- later te gebruiken binnen vervanging of match-object

$r' ((\text{d}^4) ? (\text{A-Z})^2) ^*$

12      2      3      31

@ at computing

CC BY-NC-ND 4.0 | v12f - h13 - 6

### Aantekeningen

---



---



---



---



---



---



---



---



---



---



---



---



---

## Reguliere expressies – method `search`

Gebruik van dat wat tussen haakjes matcht

- voorbeeld: postcode in regel ontdekken

```
pcvind = re.compile( r'((\d{4}) ?([A-Z]{2}))' )
regel = 'AT Computing, Arnhemsestraatweg 33, 6881 ND te Velp'

pcmatch = pcvind.search(regel)      # zoek postcode

if pcmatch:
    pcgeheel = pcmatch.group(1)    # 6881 ND
    pccijfers = pcmatch.group(2)   # 6881
    pcletters = pcmatch.group(3)  # ND
```

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Reguliere expressies – method `findall`

In één keer alle voorkomens vinden

- method `findall` returnt list van
  - strings als *geen groepering* (haakjes) in RE
  - tuples als meer dan één groep in RE
- voorbeeld: alle postcodes in regel ontdekken

```
regel = '6881 ND te Velp and 3438 MR te Nieuwegein'

pcvind = re.compile( r'\d{4} ?[A-Z]{2}' )
pclist = pcvind.findall(regel)      # zoek postcodes
for pc in pclist:
    print(pc)          # '6881 ND'
                      # '3438 MR'

pcvind = re.compile( r'((\d{4}) ?([A-Z]{2}))' )
pclist = pcvind.findall(regel)      # zoek postcodes
for pc in pclist:
    print(pc)          # ('6881 ND', '6881', 'ND')
                      # ('3438 MR', '3438', 'MR')
```

CC BY-NC-ND 4.0 | v12f – h13 – 8

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Reguliere expressies – method `split`

Hak string in mootjes

- reeds gezien: `data.split(sep)`

```
data      = '3456,17,259.3,31'
waarden = data.split(',')
print(waarden)  # ['3456', '17', '259.3', '31']

data      = '3456 ,17, 259.3,,           31'
waarden = data.split(',')
print(waarden)  # ['3456 ', '17', ' 259.3', '', '           31']
```

- ook mogelijk: `sep=re.compile(data).split()`

voorbeeld: , en . als separator, witruimte en lege elementen negeren

```
data      = '3456 ,17, 259.3,,           31'
sep      = re.compile( r'\s*[.,]+\s*' )
waarden = sep.split(data)

print(waarden)  # ['3456', '17', '259', '3', '31']
```

CC BY-NC-ND 4.0 | v12f – h13 – 9

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

## Reguliere expressies – method `sub`

### Wijzigen met `sub`

```
vraag = 'is dit een zin of een onzin?'
zoek = re.compile(r'zin')
nieuw = zoek.sub(r'ding', vraag) # evt. 3e parameter 'count=1'
print(nieuw) # 'is dit een ding of een onding?'
```

### Uitgebreider voorbeeld met `sub`

```
kt = re.compile( r'[A-Z]{2}(-\d{2}){2}' ) # oud kenteken
tn = re.compile( r'\b0\d{9}\b' ) # telefoonnummers
pc = re.compile( r'\b\d{4} ?[A-Z]{2}\b' ) # postcodes

f = open("artikel")
hele = f.read() # lees file geheel
# anonimiseer alle:
# - kentekens
# - telefoonnummers
# - postcodes

hele = kt.sub("XX-XX-XX", hele)
hele = tn.sub("0123456789", hele)
hele = pc.sub("9999 ZZ", hele)

print(hele)
```

CC BY-NC-ND 4.0 | v12f - h13 - 10

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

---

## Reguliere expressies – opties

### Opties bij compileren van reguliere expressies

<b>I of IGNORECASE</b>	bij matchen uppercase en lowercase gelijk
<b>L of LOCALE</b>	gebruik voor \w en \W (woordtekens) of \b en \B (woordgrens) de huidige locale, bijv. voor Å of Æ
<b>M of MULTILINE</b>	^ matcht ook direct na \n in multiline string \$ matcht ook direct voor \n in multiline string
<b>S of DOTALL</b>	. (dot) matcht ook embedded \n
<b>U of UNICODE</b>	gebruik voor \w, \W, \b en \B UNICODE databases
<b>X of VERBOSE</b>	vriendelijker reguliere expressies toestaan: negeer witruimte (tenzij escaped) en tolereer commentaar vanaf #

```

regel = '6881 nd te Velp en 3438 mR te Nieuwegein'
poco = re.compile(r'\d{4} \ ? [A-Z]{2} # postcode' ,re.X|re.I)
allepc = poco.findall(regel)           # zoek alle postcodes
for p in allepc: print(p)            # '6881 nd' en '3438 mR'

```

CC BY-NC-ND 4.0 | v12f – h13 – 11

@ at computing

### Aantekeningen

---



---



---



---



---



---



---



---



---



---

## Reguliere expressies – functies

Functies beschikbaar die lijken op RE object *methods*

- eerste argument is reguliere expressie
  - andere argumenten zelfde als voor method
  - reguliere expressie gecompileerd per functie-aanroep (performance)
- voorbeeld: gebruik van *functie search*

```
m = re.search(r"\d{4} ?[A-Z]{2}",  
             "zijn adres is Scheerjeweg 4, 9999ZZ Louloene")  
if m:  
    print( m.group() )           # gevonden postcode: 9999ZZ
```

in plaats van method `.search`

```
r = re.compile(r"\d{4} ?[A-Z]{2}")  
m = r.search("zijn adres is Scheerjeweg 4, 9999ZZ Louloene")  
  
if m:  
    print( m.group() )           # gevonden postcode: 9999ZZ
```

CC BY-NC-ND 4.0 | v12f – h13 – 12

@ at computing

### Aantekeningen

---

---

---

---

---

---

---

---

---

---

