



NEW MEDIA &
COMMUNICATION
TECHNOLOGY

Basic Programming

Python 3.7

Inhoudstafel

Voorwoord	5
Hoofd-doelstellingen	5
Voorkennis	5
Oefening baart kunst.....	5
Lesmateriaal	5
Evaluatie.....	6
Software	6
Inleiding - Python.....	7
Syntax.....	7
Uitvoering.....	7
Python 3.6.....	8
Python features op een rijtje	8
Verkenning van Pycharm	8
Python Script	8
Python console	10
Eerste Python Programma	12
Python basis Sytax.....	13
Python Identifiers	13
Gereserveerde woorden	13
Alignering van codelijnen	13
Multi-Line Statements	14
Quotes.....	14
Commentaar	14
Meerdere statements op één lijn	14
Gebruik van dubbel punt	14
Variabelen en datatypes.....	15
Waarden toekennen aan variabelen	15
Naamgeving variabelen	15
Waar zijn de datatypes in Python gebleven?	15
Python number datatypes	16
Boolean datatype?	16
String	17
Datatype conversie	17
Output van variabelen.....	18
Syntax placeholder	19
Input van de gebruiker.....	19
Vaak voorkomende fouten	20
Code stijl	20
Basic operatoren.....	21
Rekenkundige operatoren	21

Vergelijkings operatoren.....	21
Toekenings operatoren.....	22
Bitoperatoren	22
Logische operatoren.....	22
Verzameling operatoren.....	23
Identity operatoren.....	23
Vorrangsregels	23
Selectiestructuren.....	25
Inleiding	25
Enkelvoudige selectie	25
Tweevoudige selectie.....	26
Meervoudige selectie	26
Switch in Python?	27
Deelproblemen.....	28
Inleiding.....	28
Functies.....	28
Functie zonder parameters	29
Functie met parameter(s)	29
Return statement	30
Bereik van variabelen	31
Deelproblemen?	32
String	33
Algemeen.....	33
String operatoren	33
Weergave van Strings	33
Stringfuncties.....	34
Herhalingen (lussen).....	36
Inleiding.....	36
For-lus & range.....	36
Voorwaardelijk herhaling: while	37
Oneindige lus	37
Break.....	37
Continue.....	38
Datastructuren.....	39
List	39
Basis operatoren.....	40
Ingebouwde functies & methodes.....	40
Tuple.....	41
Basis operatoren.....	41
Ingebouwde functies & methodes.....	41
Set	41
Basis operatoren.....	42

Dictionary	42
Input/output	44
Printen naar het scherm	44
Input-functie	44
Openen van een bestand	44
Status van een file-object.....	45
Close()-methode	46
Inlezen van en schrijven naar bestanden	46
Write()-methode	46
read()-methode	46
readline()-methode	47
readlines()-methode.....	47
Object georiënteerd programmeren	48
“Object”-voorbeeld uit de realiteit	48
Objecten uit de programmeerwereld	48
Klassen & objecten.....	49
Klassen & datatypes in Python.....	49
Eerste eigen klasse in Python	50
class	50
__init__()	50
__str__() en __repr__().....	51
Eigen methodes	52
Information hiding.....	54
Data hiding: public vs private attributen	54
Data encapsulation.....	55
Get- en set- methodes	55
Python’s versie van get- en set-methodes.....	56
Conclusie	58
Klasse attributen & methodes.....	59
Klasse versus instantie attributen	59
Voorbeeld klasse attribuut	60
Static methodes.....	60
Klasse methodes	61
Overerving.....	62
Basis syntax overerving.....	62
Subklasse uitbreiden.....	63
Meervoudige overerving	64
Operator overloading.....	65
Exception handling.....	67
17.1 Errors en exceptions	67
17.2 Afhandelen van exceptions.....	67
17.3 Afhandelen van specifieke exceptions.....	68

17.4 Toevoegen van een finally	69
17.5 Genereren van exceptions	70

Voorwoord

Welkom in de module 'Basic Programming'. Tijdens deze module maak je op een praktijkgerichte manier kennis met de programmeertaal Python. De focus binnen deze module ligt bij een doorgedreven kennis van de programmeertaal Python. Deze programmeertaal gebruiken we voor een kennismaking met OOP (Object Oriented Programming) en de programmatie van een Raspberry-Pi.

Hoofd-doelstellingen

De doelstellingen van deze module zijn:

- de basissyntax van de programmeertaal Python onder de knie krijgen;
- aanleren van top down analyse techniek waarbij een gegeven probleem verder in kleinere haalbare deelp Problemen opgesplitst wordt;
- 'zelfredzaam' worden: als toekomstige programmeur is het belangrijk om -op eigen
- initiatief- andere, extra bronnen te raadplegen om een probleem te kunnen verwerken;
- vlot kunnen werken met een versiebeheersysteem (Github)

Voorkennis

Deze module veronderstelt geen voorkennis van het programmeren. We starten vanaf nul. Deze cursus is ondersteunend aan de gegeven theorielessen. Dit impliceert dus niet dat alles wat in de theorieles verteld wordt, ook hier beschreven staat (integendeel zelfs). We adviseren tijdens de theorielessen voldoende notities te nemen en eventuele bordschema's over te nemen.

Oefening baart kunst

Een programmeertaal zoals Python onder de knie krijgen, heeft veel gelijkenissen met het zich eigen maken van een spreektaal: ook hier is veelvuldig oefenen essentieel en een noodzakelijke voorwaarde om Python zich eigen te maken. Aanvankelijk moet men niet enkel de verschillende programmeertermen uit elkaar houden; bovendien moet men daarnaast ook de logica van een programma leren ontdekken. Daarom adviseren we u alle oefeningen (meermaals) opnieuw te maken zonder daarbij oplossingen over te tikken of zomaar te kopiëren uit bestaande oefeningen. Integendeel, bestudeer eerst de talrijke voorbeelden uit de theorielessen. Doe aan zelf-exploratie: maak een kleine variant van de demo. Schakel dan pas over naar het labo.

Programmeer ook meerdere keren (al dan niet korte) momenten per week. Eénmaal per week een halve dag uittrekken is minder effectief dan meermaals per week een uur aan programmeren te spenderen. Programmeer met plezier. Programmeren is een creatieve en interessante bezigheid. Geniet ervan!

Lesmateriaal

Naast een pdf-cursus, bestaat het officieel lesmateriaal ook uit de slides uit de theorieles, de verschillende labo-opgaves, over te nemen bordschema's, eigen notities,...

Ook online is er heel wat informatie te vinden. We verwijzen tijdens de cursus hierbij actief door naar:

<https://docs.python.org/3/index.html>

<https://www.pluralsight.com/> (enkel met licentie)

Evaluatie

Het eindcijfer van de module wordt als volgt samengesteld:

- Tussentijdse evaluatie (kwartaal 1): 10%
- Een schriftelijk examen over de theorie: 20%
- Een examen op je laptop: 70%

Software

Voor deze instapcursus en aansluitende module gebruiken we de ontwikkelomgeving van PyCharm 5 Professional Edition. Er zijn verschillende versies beschikbaar. Deze is downloadbaar via volgende url:

<https://www.jetbrains.com/pycharm/download/>

Een licentie is te verkrijgen via uw officieel e-mail adres:

<https://www.jetbrains.com/shop/eform/students>

Hou ermee rekening dat de installatie enige tijd en ruimte in beslag neemt.

Verdere Installatie-details worden in de les gegeven.

Alvast veel succes!

Stijn Walcarius

stijn.walcarius@howest.be

Inleiding - Python

Python is een programmeertaal die tussen 1985-1990 ontwikkeld werd door Guido van Rossum, destijds verbonden aan het Centrum voor Wiskunde en Informatica (daarvoor Mathematisch Centrum) in Amsterdam. Inmiddels wordt de taal doorontwikkeld door een enthousiaste groep, geleid door Van Rossum. Deze groep wordt ondersteund door vrijwilligers op het internet. De ontwikkeling van Python wordt geleid door de Python Software Foundation. Python is vrije software en valt onder de General Public License (GPL).

Python heeft zijn naam te danken aan het favoriete televisieprogramma van Guido van Rossum, Monty Python's Flying Circus.

Python wordt op veel plaatsen gebruikt in scripts ten behoeve van systeembeheer, als makkelijke taal voor het bouwen van portabele grafische user interfaces, bij web-applicaties en ga zo maar door. De installatieprogrammatuur van vele Linux distributies, waaronder RedHat, is bijvoorbeeld gebouwd met Python.

Python onderscheidt zich van andere (scripting) talen doordat het van de grond af aan al als object-georiënteerde taal is opgezet. Daarbij beschikt de taal over een grote standaardbibliotheek waarop u verder kunt bouwen. Mede hierdoor leent Python zich erg goed om snel applicaties te bouwen (Rapid Application Development).

Syntax

Python is ontwikkeld met het oog op leesbare code. Hieruit vloeit haar "zuivere" stijl voort. Met weinig woorden kan men veel zeggen. Dit uit zich op verschillende manieren.

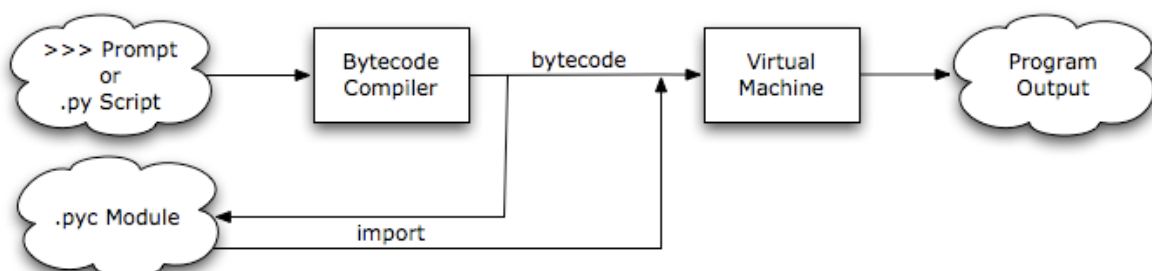
- Structuur binnen een python-file wordt aangebracht door indentatie, of regelinspringing in plaats van bijvoorbeeld accolades uit C/Java.
- Statements (vergelijkbaar met zinnen uit gewone taal) worden simpelweg beëindigd door het eind van de regel.
- Variabelen krijgen geen typedeclaratie. Python maakt gebruik van duck-typing. Een uitgebreide voorstelling van de syntax staat hieronder.

Elk van deze zaken worden verder uitvoerig behandeld.

Uitvoering

Een Python-programma wordt geschreven in een of meerdere tekstbestanden met de extensie .py. Om een Python-programma uit te voeren dient men de **Python-interpreter** aan te roepen, gevolgd door de naam van het bestand, waarin het begin van het programma wordt gecodeerd.

De interpreter zet de Python-programmacode niet meteen om in machine-instructies, maar compileert naar een tussenvorm, bytecode genoemd. Deze wordt opgeslagen in de vorm van een .pyc-bestand (met de c van compiled). De bytecode is onafhankelijk van het besturingssysteem - ze kan verhuisd worden naar een ander besturingssysteem. De volgende stap is het uitvoeren van de bytecode door de Python virtual machine.



Python wordt geleverd met een uitgebreide bibliotheek om van alles en nog wat standaard te kunnen bewerken. Het is erg eenvoudig om in Python herbruikbare code te schrijven. Doordat veel van de bibliotheken die mensen schrijven gratis aan anderen ter beschikking wordt gesteld, groeien de mogelijkheden van de bibliotheek

voortdurend. Python wordt zo tot een programmeertaal die voor razendsnel ontwikkelen van een nieuwe applicatie kan worden gebruikt, zonder dat de daarbij geproduceerde code onleesbaar wordt.

Python 3.7

In deze module gebruiken we Python versie 3. Hoewel onze lessen geheel op zichzelf staan, zul je, wanneer je online gaat zoeken naar informatie over Python, merken dat ook Python version 2 vaak wordt gebruikt. Maar let op: Python version 2 verschilt van Python version 3. Je kunt beide versies niet door elkaar gebruiken!

Python features op een rijtje

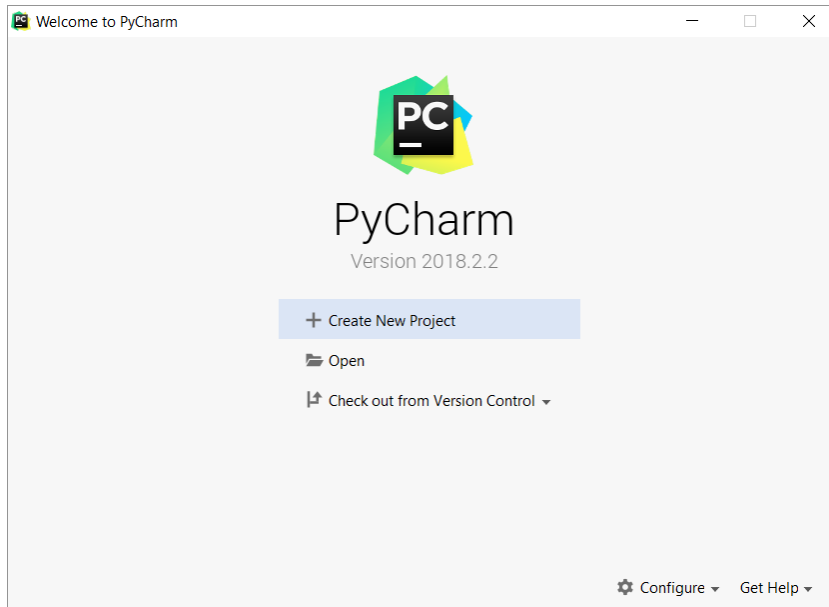
Python

- Is 'easy-to-learn': Python heeft weinig keywords, eenvoudige structuren en een heldere syntax. Hierdoor is Python relatief eenvoudig aan te leren.
- Is 'easy-to-read': Python code is gemakkelijk leesbaar.
- Is 'easy-to-maintain': Python's source code is eenvoudig onderhoudbaar.
- Heeft een uitgebreide standaard bibliotheek: Python beschikt over een uitgebreide bibliotheek die eenvoudig overdraagbaar is en voor verschillende OS-platformen (Unix, Windows, Macintosh) compatibel is.
- Is interactief: Python laat interactive testing en debugging van codesnippets toe.
- Is overdraagbaar: Python draait op verschillende hardware platforms met dezelfde interface op alle platformen.
- Is uitbreidbaar: De Python interpreter kan uitgebreid worden met extra modules. Deze modules laten programmeurs toe om nieuwe tools of bestaande uit te breiden om zo meer efficiënt te werken.
- Kan overweg met de meeste commerciële databases
- Ondersteunt GUI applicaties
- Scalable: Python levert een betere ondersteuning voor grotere programma's dan shell scripting.
- Is 'open-source', vrij populair en beschikt over een grote community
- Is toepasbaar op vele domeinen

Verkenning van Pycharm

Python Script

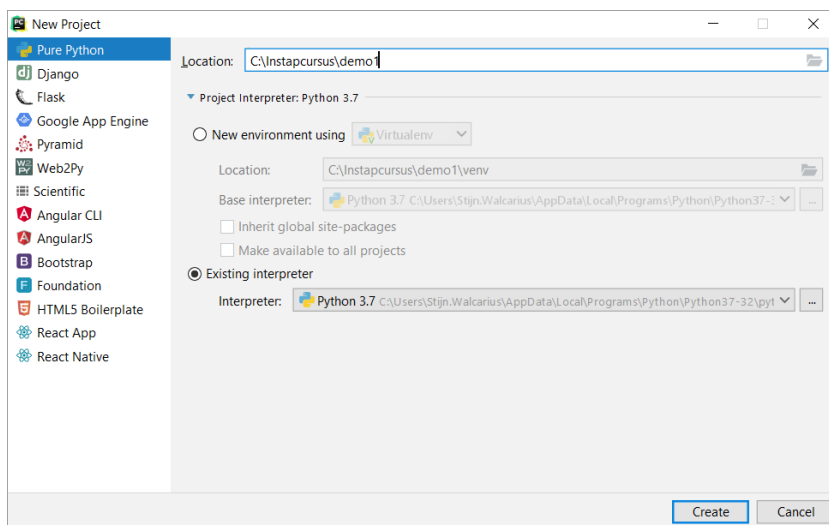
We starten steeds met een nieuw project:



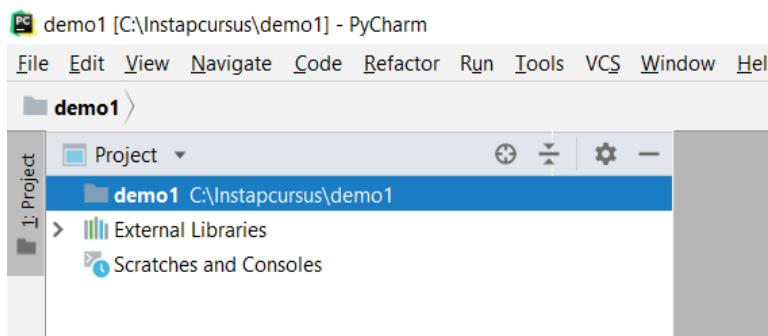
Vervolgens geeft u een locatie op waar de verschillende bestanden kunnen bewaard worden.

Bij de interpreter kan je:

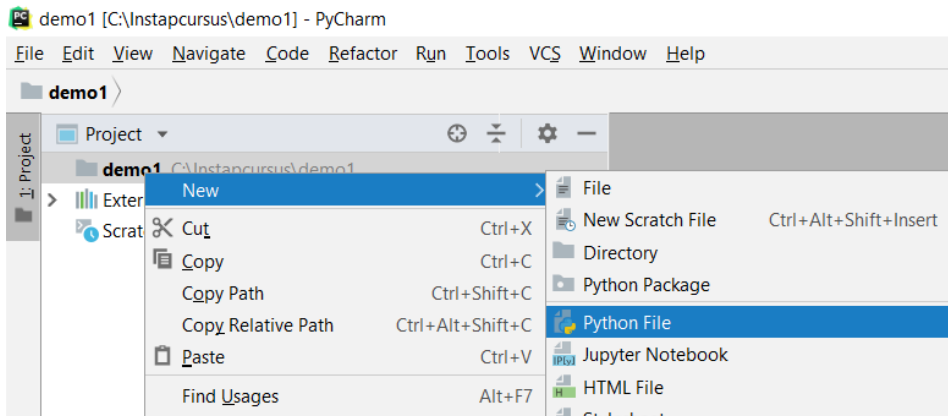
- ofwel kiezen voor een virtuele python interpreter (per project afzonderlijk)
- ofwel kiezen voor eenzelfde python interpreter voor alle projecten.



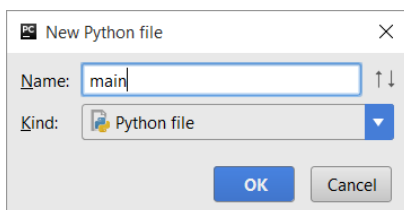
Na creatie opent zich het hoofdscherm.



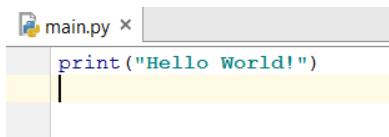
Een nieuwe python-file aanmaken, gebeurt heel eenvoudig:



Geef nu een willekeurige bestandsnaam op, bv 'main.py':

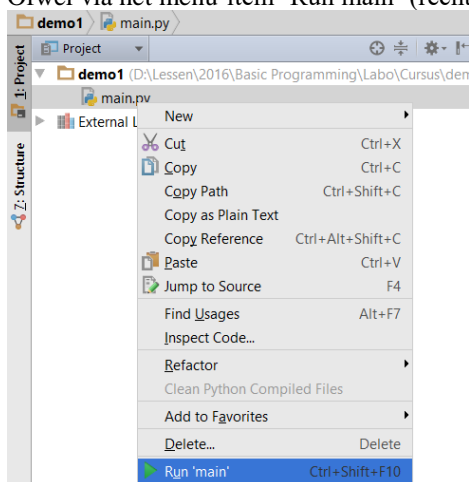


Je bent klaar om jouw eerste python-script te schrijven:



Uitvoeren kan door:

- Ofwel de sneltoets Ctrl +Shift +F10 (indien voor de eerste keer)
- Ofwel via het menu-item 'Run main' (rechter muisklik op python-bestand)



Python console

Binnen PyCharm kunnen we ook rechtstreeks via het console venster python code laten uitvoeren.

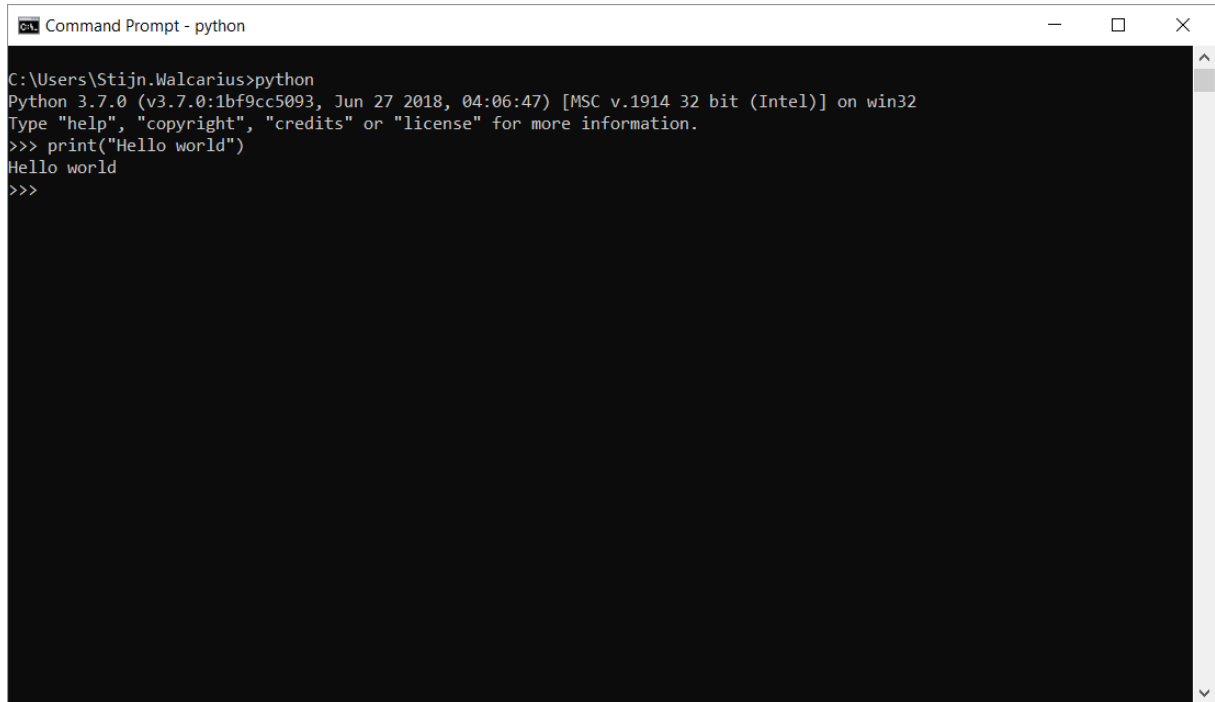
De console heeft beperkingen ten aanzien van de veiligheid en complexiteit (er is bijvoorbeeld een maximum run-tijd van 1 seconde, en er is geen mogelijkheid voor interactiviteit).

De python console kan via PyCharm opgestart worden:



```
Run: main ×  
C:\Users\Stijn.Walcarius\AppData\Local\Programs\Python\Python37-32\python.exe C:/Instapcursus/demo1/main.py  
Hello world!  
Process finished with exit code 0
```

Of via een prompt venster:




```
Command Prompt - python  
C:\Users\Stijn.Walcarius>python  
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32  
Type "help", "copyright", "credits" or "license" for more information.  
>>> print("Hello world")  
Hello world  
>>>
```

Via het python prompt kan je rechtstreeks communiceren met de interpreter om zo een programma te gaan ontwikkelen.

Eerste Python Programma

Laat ons een eerste python programma ontwikkelen via


- Interactive mode (python console):
Tik de volgende lijn code in & druk op enter:
`print("Welkom in de module Basic Programming")`

 Command Prompt - python

```
Microsoft Windows [Version 10.0.17134.228]
(c) 2018 Microsoft Corporation. All rights reserved.

C:\Users\Stijn.Walcarius>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:06:47) [MSC v.1914 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> print("Welkom in de module Basic Programming")
Welkom in de module Basic Programming
>>>
```

- Script mode (buiten PyCharm): maak een python-file aan (dit is een bestand met de extensie 'py').
Schrijf bovenstaande lijn in het bestand. Uitvoeren kan via het python-commando.

 Command Prompt

```
C:\Instapcursus\demo1>python main.py
Hello world!

C:\Instapcursus\demo1>_
```

- Script mode (via PyCharm): zie eerder in deze cursus.

Betekenis van bovenstaande regel code:

Wanneer je een programma uitvoert, krijg je ook de uitvoer (Eng.: output) van het programma te zien. Het voorbeeldprogramma hierboven bestaat maar uit één opdracht, `print("Hello, World!")` en het genereert één regel uitvoer:

Hello world!

Hier volgt een analyse van het programma:

`print` is de naam van een Python-opdracht, die boodschappen naar de uitvoer stuurt.

De haakjes `()` na de `print`-opdracht staan om de inhoud heen die je wilt afdrukken.

De aanhalingstekens `" "` worden gebruikt om het begin en einde van de tekst `Hello, World!` aan te geven. Zonder aanhalingstekens zou Python denken dat `Hello` als een opdracht bedoeld was. En dat zou als fout worden aangemerkt omdat zo'n opdracht niet bestaat in Python.

Python basis Syntax

Python Identifiers

Een Python identifier is een naam om een variabele, functie, klasse, module of project aan te spreken. Een identifier start steeds met een letter (klein/groot) of een underscore gevolgd door 0 of meerdere letters/cijfers/underscores.

Python laat geen speciale karakters binnen een identifier toe: karakters zoals @, \$, and % zijn dan ook niet toegestaan.

Let op, de Python-syntax is hoofdlettergevoelig!

Enkele afspraken rond het gebruik van conventies zijn:

- Klasse namen starten steeds met een hoofdletter. Al de rest start met een kleine letter.
- Een identifier dat start met één underscore wordt als private aanzien (verdere toelichting volgt later)
- Een identifier dat start met twee underscores wordt als 'strongly' private aanzien (verdere toelichting volgt later)
- Een identifier dat start en eindigt met twee underscores, is een specifieke Python methode dat je niet zelf mag oproepen (verdere toelichting volgt later)

Gereserveerde woorden

Onderstaande lijst somt de Python keywords op. Deze zijn gereserveerd waardoor je ze niet kan gebruiken voor eigen constanten/variabelen of andere identifiers. Alle keywords zijn in kleine letters. In deze cursus zullen elk van deze verder besproken worden.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	If	return
del	import	try
elif	In	while
else	Is	with
except	lambda	yield

Alignering van codelijnen

Soms wensen we een aantal coderegels (verder ook soms statements genoemd) steeds samen uit te voeren. Ze vormen samen een codeblok. Python ondersteunt geen accolades (zoals Java/C/...) om het einde van een codeblok aan te duiden. Binnen Python laten we deze coderegels een aantal spaties inspringen. Het aantal spaties is weliswaar willekeurig, maar alle coderegels moeten binnen een zelfde blok evenveel inspringen. De conventieregels adviseren om 4 spaties hiervoor te gebruiken.

Correcte syntax

```
if (4 == 4):
    print("Het antwoord is ")
    print("True")
else:
    print("Het antwoord is ")
    print("False")
```

Verkeerde syntax

```
if (4 == 4):
    print("Het antwoord is ")
    print("True")
else:
    print("Het antwoord is ")
    print("False")
```

Binnen PyCharm kan men via Ctrl+Alt+L de volledige code in één keer op een uniforme wijze laten aligneren. Dit verhoogt verder de leesbaarheid.

Multi-Line Statements

In python duidt elke nieuwe regel een nieuw statement aan. Python kan -indien gewenst- één codestatement over meerdere regels uitspreiden. Hiervoor gebruiken we het karakter '\'. Voorbeeld:

```
contactpersoon = "Walcarius " + \  
                 "Stijn " + \  
                 "stijn.walcarius@howest.be"
```

Statements met [], {}, or () hoeven dit karakter niet te gebruiken.

Quotes

Python accepteert het gebruik van zowel single ('), double (") als triple (""" or """) om Strings aan te duiden zolang zowel bij de start als het einde het zelfde karakter gebruikt wordt.

Commentaar

Een hash teken (#) dat niet binnen een string opgenomen is, duidt de start van commentaar aan. Commentaar wordt niet door de Python Interpreter uitgevoerd.

```
contactpersoon = "Walcarius Stijn" #e-mail: stijn.walcarius@howest.be
```

Ook lege lijnen worden door de interpreter overgeslagen.

Meerdere statements op één lijn

Binnen Python kunnen meerdere statements op één lijn, zolang dat deze gescheiden worden door een ';'.

```
contactpersoon = "Walcarius Stijn"; opleiding = "NMCT"; jaartal=2016
```

Gebruik van dubbel punt

Een groep van statements die samen een codeblok afbakenen, worden 'suites' in Python genoemd. Sommige complexere statements (zoals if, while, def, class) vereisen een specifieke header en een suite. De header wordt afgesloten met een ':'. In volgende hoofdstukken worden deze verder besproken.

Variabelen en datatypes

Van zodra we een programma van betekenis willen schrijven, moeten we gebruik maken van variabelen. We starten met het concept zelf te introduceren. Een variabele is een soort opbergdoos dat gebruikt wordt om waarden in op te slaan, zodat deze waarden in het programma kunnen gebruikt of veranderd worden.

Enkele typische situaties waar variabelen gebruikt worden:

- de berekening van een resultaat, bv. de remafstand van een voertuig;
- het bijhouden van een waarde, bv. de snelheid van een voertuig;
- de ingave van de gebruiker, bv. zijn/haar voornaam;
- bijhouden van een datum.

Al snel wordt duidelijk dat we variabelen kunnen onderscheiden op basis van de soort waarde die ze bijhouden. Sommige variabelen kunnen getallen bijhouden, andere richten zich op pure tekst, nog andere op datums, enz. We spreken over het datatype van een variabele. Het beschrijft wat de variabele kan bevatten.

Waarden toekennen aan variabelen

De reservatie van geheugen bij de Python variabele gebeurt automatisch wanneer deze een waarde krijgt. Via het gelijkheidsteken wordt een waarde aan de variabele gegeven. Steeds staat hierbij

- links de variabelenaam
- rechts de waarde van de variabele

In de literatuur spreekt men ook over de toekenning van een waarde aan een variabele.

Voorbeelden:

```
woord = "Opleiding NMCT"  
testwaarde1 = 85.256  
waardeboolean = True
```

Het is ook mogelijk om op een zelfde lijn in één keer meerdere variabelen dezelfde waarde te geven. Hierdoor hebben deze variabelen onmiddellijk ook hetzelfde datatype. Voorbeeld:

```
getal1 = getal2 = getal3 = 2016  
a,b,x,y = 1,15,3,4
```

Naamgeving variabelen

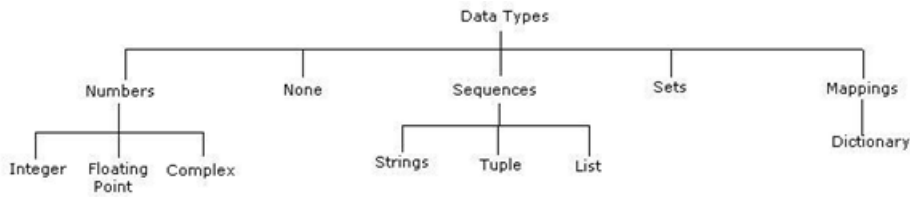
Programmeurs kiezen meestal zinvolle namen voor hun variabelen. Ze documenteren waarvoor de variabelen instaan. Variabelenamen kunnen een willekeurige lengte hebben. Ze kunnen zowel letters als cijfers bevatten. Ze kunnen niet starten met een cijfer. Het is toegestaan om hoofdletters te gebruiken, maar eerder vermelde conventies bevelen kleine letters aan.

De underscore (_) mag voorkomen in een variabelenaam en wordt veel gebruikt in namen met veel woorden, zoals mijn_naam of luchtsnelheid_van_een_ongeladen_vliegtuig.

Waar zijn de datatypes in Python gebleven?

In Python heeft elke variabele zijn datatype, maar je hoeft ze niet zoals in andere programmeertalen expliciet te vermelden. Hoe werkt dit? Python bepaalt het datatype van een aangemaakte variabele op basis van zijn eerste toekenning. Krijgt een variabele bijvoorbeeld een niet-decimale waarde toegewezen, dan wordt het datatype 'int'.

Python heeft volgende soorten datatypes:



In eerste instantie concentreren we ons op de numbers-datatypes en String-datatype.

Python number datatypes

Variabelen met een Number datatype kunnen een getal bijhouden. Python ondersteunt verschillende number datatypes:

- integer (niet decimale getallen)
- float (decimale getallen)
- complex (complexe getallen)

Enkele voorbeelden van waarden die onder elk number-datatype thuishoren:

integer	float	complex
0b10	0.0	3.14j
0b100	15.20	45.j
-786	-21.9	9.322e-36j
0o467	32.3+e18	.876j
-0x490A	-90.	-.6545+0J
9854	-32.54e100	3e+26J
0o62	70.2-E12	4.53e-7j

Opmerkingen:

- In Python 3 bestaat het datatype long niet meer
- Een int waarde kan naast een decimaal ook in een binair, octaal of hexadecimaal formaat opgeven worden. Hierbij worden bij de laatste drie formaten de waarde resp. vooraf gegaan door 0b, 0o, 0x.

Boolean datatype?

Python levert onder het boolean-datatype twee mogelijke waarden af: True en False. Elk object heeft een 'Truth value testing' wat inhoudt dat elk object een true of false kan afgeven. Via de ingebouwde bool-functie kan je dit vervolgens opvragen.

Volgende waarden leveren False op:

- None
- False
- 0 bij elk numeriek datatype
- Elke lege sequentie (zie verder)
- Elke lege mapping (zie verder)
- Objecten van eigen klassen als bepaalde voorwaarden voldaan zijn (zie verder)

Al de rest levert dus een True terug.

Enkele voorbeelden:

```
woord = "Opleiding NMCT"
print(bool(woord))
testwaarde1 = 85.256
print(bool(testwaarde1))
```

Doordat elk object een true of false kan opleveren, kan elk object in de conditie van een if of while gebruikt worden. Elk object kan ook opgenomen worden in een booleaanse operatie. Zie verder voor meer info.

String

Variabelen van het datatype String kunnen tekst (=opeenvolging van karakters) opslaan. De te bewaren waarde dient tussen enkelvoudige of dubbele aanhalingstekens neer geschreven worden.

Het datatype String levert heel wat handige methodes af, waarvan hieronder er reeds een aantal opgesomd zijn:

- Subset van een string: hierbij wordt een specifiek deel van de tekst opgehaald. Het begin en einde van een subsets wordt via [] en [:] aangeduid. Hou er mee rekening dat het eerste karakter op positie 0 staat.
- Het plus-teken laat toe 2 strings aan elkaar te plakken ('concatenatie van strings')
- Het *-teken herhaalt een aantal keren een string

Voorbeelden:

```
woord = "Opleiding NMCT"
print(woord)           # print de volledige string
print(woord[0])        # print het eerste karakter van de string
print(woord[2:5])      # print de karakters van positie 2 tem positie 5
print(woord[2:])       # print de string vanaf positie 2
print(woord * 2)       # print de string twee maal
print(woord + "TEST")  # print de samenstelling van 2 string
```

Resultaat:

```
Opleiding NMCT
O
lei
leiding NMCT
Opleiding NMCTOpleiding NMCT
Opleiding NMCTTEST
```

Datatype conversie

Soms wens je een conversie van het ene naar het andere datatype uitvoeren. Hiervoor gebruik je eenvoudig de corresponderende methode.

Op analoge wijze kan ook binnen een zelfde datatype het formaat gewijzigd worden. Bijvoorbeeld van decimaal naar binair.

De (voorlopige) lijst is:

int(x)	Zet om naar datatype integer
float(x)	Zet om naar een floating-point (decimaal getal)
complex(x)	Zet om naar datatype complex
str(x)	Zet om naar datatype string
hex(x)	Zet om naar een hexadecimaal formaat
oct(x)	Zet om naar een octaal formaat

Voorbeeld:

```

waardel = "123"           #datatype string
#waardel = waardel + 10    #uitvoeringsfout: waarom?
waardel = int(waardel) + 10 #datatype int
print(hex(waardel))        #datatype int, formaat hexadecimaal

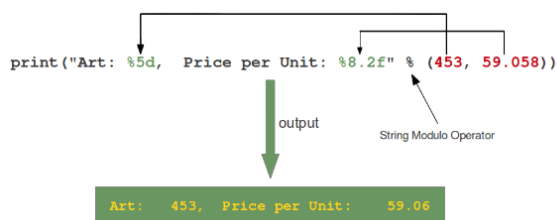
```

Output van variabelen

Er zijn verschillende werkwijzen om het print-commando in python te gebruiken. Elke methode heeft zijn voor- en nadelen.

- We sommen de verschillende variabelen op, telkens gescheiden door een komma. Het resultaat is de verschillende waarden met telkens een spatie tussenin.
`print(waardel, woord)`
- We maken een nieuwe string aan door gebruik te maken van het concatenatieteken (+) en de str-methode (in het geval de variabele geen string is)
`print("Het getal is " + str(waardel) + ", het woord is " + woord)`
- Een iets verouderde, maar nog steeds populaire versie is het gebruik van de string modulo operator %. In de string gebruiken we stakeholders: dit zijn plaatsen aangeduid met een % waar later een bepaalde waarde dient geplaatst te worden.

Bijvoorbeeld:



```

print("Art: %5d, Price per Unit: %8.2f" % (453, 59.058))

```

output

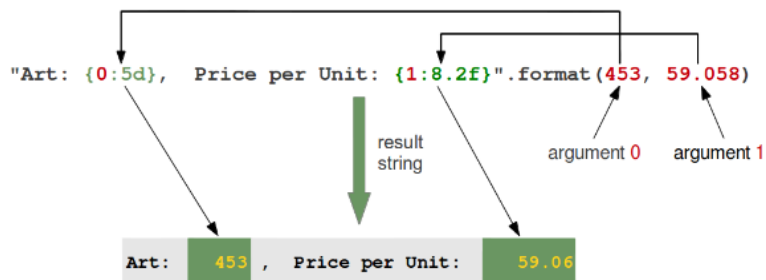
String Modulo Operator

Art: 453, Price per Unit: 59.06

In dit voorbeeld worden twee stakeholders gebruikt: “%5d” en “%8.2f”. De syntax voor een stakeholder wordt verder toegelicht.

- De meer recentere methode is via de format-methode. Ook hier werkt men met stakeholders, maar deze zijn nu aangeduid met accolades. Er zijn nu twee manieren beschikbaar:

In de eerste methode wordt de vermelde positie in de stakeholder gebruikt om de juiste parameter achteraan uit de format-methode op te halen. De gebruikte term hierbij is ‘position parameter’.



```

"Art: {0:5d}, Price per Unit: {1:8.2f}".format(453, 59.058)

```

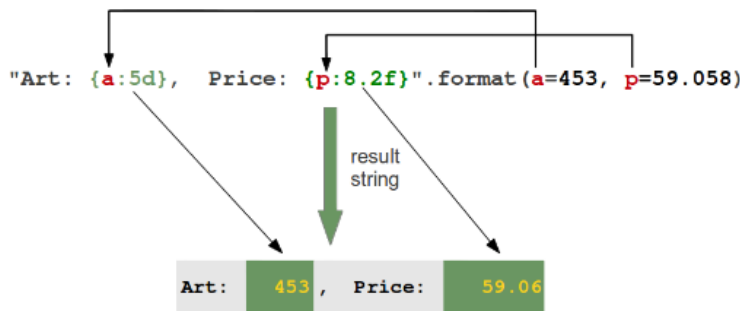
result string

argument 0

argument 1

Art: 453, Price per Unit: 59.06

In de tweede variant werkt men met keywords vooraan de stakeholder om de juiste parameter achteraan op te vragen.



In beide versies kan men via extra informatie in de stakeholder het gewenste formaat preciezer definiëren.

Syntax placeholder

Binnen een placeholder kan men het gewenste formaat via een specifieke syntax verder verfijnen.

Algemeen wordt steeds volgende structuur gerespecteerd:

[flags][width][.precision]type

Laat ons ter illustratie een concreet voorbeeld nemen: "%8.2f"

Het laatste karakter 'f' wijst op een decimaal getal (float). Het eerste cijfer (hier 8) is het minimum aantal af te printen karakters binnen de string (met inbegrip van het decimale teken). Het tweede cijfer (hier 2) beperkt het decimale gedeelte tot 2 cijfers na de komma. Indien nodig wordt het getal afgerond. De twee cijfers worden steeds getoond. Het procent-teken is niet nodig wanneer men gebruik maakt van de format-methode.

Volgende getallen worden dan ook als volgt weergegeven:

```
23.789 → " 23.79"
0.039 → " 0.04"
199.8 → " 199.80"
23 → " 23.00"
782324.127 → "782324.13"
```

Flags laten toe verdere verfijningen in de output toe. "%-8.2f" zorgt er extra voor dat de tekst links gealigneerd wordt.

Voor veel meer mogelijkheden verwijzen we door naar <https://docs.python.org/> of naar de talrijke online voorbeelden.

Input van de gebruiker

Er bestaan bijna geen programma's die geen input van data gebruiken. Deze data kan komen vanuit een database, een andere computer, muiskliks/bewegingen of het internet...

Vaak wenst men ook data rechtstreeks van het keyboard verwerken. In Python gebruiken we daarom het input()-commando. Als optionele parameter kunnen we een string meegeven.

De uitvoering verloopt als volgt: De optionele string zal eerst op het scherm afgeprint worden. Daarna stopt de verdere uitvoering van het programma totdat de gebruiker input ingegeven heeft en met een enter afgesloten heeft.

De input zal nu verwerkt worden. De ingetikte waarde is een string! Met de juiste conversiemethodes kan je deze gaan omzetten naar het gewenste datatype.

Voorbeeld:

```
opleiding = "NMCT"  
naam = input("Geef uw naam op: ")  
voornaam = input("Geef uw voornaam op: ")  
leeftijd = int(input("Wat is uw mentale leeftijd? "))  
leeftijd *= 2  
print("Welkom {1} {0} in {2}".format(voornaam, naam, opleiding))  
print("Uw werkelijke leeftijd is {0}".format(leeftijd))
```

Vaak voorkomende fouten

Deze lijst is onvolledig en zal tijdens het semester verder aangevuld worden.

Wanneer in Python een nog niet gedefinieerde variabele aangesproken wordt, krijg je een foutmelding.

Een andere vaak voorkomende fout heeft te maken met het per ongeluk verwisselen van de twee zijden van een -=opdracht (een *toekenningsoopdracht*).

Code stijl

Python heeft tot doel een leesbare programmeertaal te zijn. Daarom is er ook een ‘style guide’ voorhanden die programmeurs aanmoedigt om een consistente programmeerstijl te gebruiken.

Op onderstaande link vindt u een meer informatie hierover. We moedigen dan ook sterk aan om deze te volgen.

<https://www.python.org/dev/peps/pep-0008/>

Basic operators

Operators zijn speciale symbolen die bewerkingen vertegenwoordigen (optellen, vermenigvuldigen, ...). De waarden waarop de operator slaat, heten **operanden**.

De operators +, -, *, / en ** voeren een optelling, aftrekking, vermenigvuldiging, deling en een kwadratering uit, zoals in de volgende voorbeelden:

20+32 hour-1 hour*60+minute minute/60 5**2 (5+9)*(15-7)

We onderscheiden in Python volgende operatoren:

- Rekenkundige operatoren
- Vergelijkings (Relationale) Operatoren
- Toekenning Operatoren
- Logische Operators
- Bitwise Operators
- Membership Operators (deze komen later aan bod)
- Identity Operators (deze komen later aan bod)

Rekenkundige operatoren

Volgende tabel geeft een overzicht van de rekenkundige operatoren in Python.

Operator	Beschrijving	Voorbeeld
+ Optelling	Telt beide waarden op	
- Aftrekking	Linkerwaarde wordt van rechterwaarde afgetrokken	
* Vermenigvuldiging	Waarden worden vermenigvuldigd	
/ Deling	Linkeroperand wordt gedeeld door rechteroperand	
% Modulo	Bepaal rest door deling	10 % 3 = 1
** exponent	Macht	9**1 = 81
//	'Floor division': deling waarbij gedeelte na komma wordt verwijderd. Geen afronding tenzij indien één operand negatief is: afronding weg van 0	9//2 = 4 9.0//2.0 = 4.0 -11//3 = -4 -11.0//3 = -4.0

Vergelijkings operatoren

Volgende tabel geeft een overzicht van de vergelijkings operatoren in Python. Deze operatoren controleren de waarden van linker- en rechteroperand, en nemen dan een beslissing.

Operator	Beschrijving
==	Als beide operanden gelijk zijn, dan is de conditie waar.
!=	Als beide operanden niet gelijk zijn, dan is de conditie waar.
<>	Als beide operanden verschillend zijn, dan wordt de conditie waar.
>	Als de linkeroperand groter is dan de rechteroperand, dan wordt de conditie waar.

<	Als de rechteroperand groter is dan de linkeroperand, dan wordt de conditie waar.
>=	Als de linkeroperand groter of gelijk is dan de rechteroperand, dan wordt de conditie waar.
<=	Als de rechteroperand groter is dan de linkeroperand, dan wordt de conditie waar.

Toekennings operatoren

Volgende tabel geeft een overzicht van de vergelijkings operatoren in Python. Deze operatoren controleren de waarden van linker- en rechteroperand, en neemt dan

Operator	Beschrijving
=	Kent rechterwaarde toe aan linkeroperand.
+=	Voegt rechteroperand toe aan linkeroperand en kent resultaat toe aan linkeroperand
-=	Trekt rechteroperand van linkeroperand af en kent resultaat toe aan linkeroperand
*=	Vermenigvuldigt linkeroperand met rechteroperand en kent resultaat toe aan linkeroperand
/=	Deelt linkeroperand door rechteroperand en kent resultaat toe aan linkeroperand
%=	Bepaalt de rest bij deling van linkeroperand door rechteroperand en kent resultaat toe aan linkeroperand
**=	Voert de machtsberekening uit tussen linkeroperand als grondtal en rechteroperand als exponent, en kent resultaat toe aan linkeroperand
//=	Voert 'Floor division' uit tussen linker en rechteroperand, en kent resultaat toe aan linkeroperand

Bitoperatoren

Het onderzoek van de verschillende operatoren maakt deel uit van het labo.

Logische operatoren

Operator	Beschrijving
and	Als beide operanden waar zijn, dan is het geheel ook waar.
or	Vanaf zodra één van de operanden waar is, dan is het geheel ook waar.
not	Neem het omgekeerde van de operand

Voorbeelden:

```
if ((getall >= 100) and (getall <= 200)):
    print("Getall ligt tussen 100 en 200")
```

```
if (not getall):
```



```
print("Getall is verschillend van 0")
```

Verzameling operatoren

Deze komen later aan bod.

Identity operatoren

Deze komen later aan bod.

Voorrangsregels

Wanneer meerdere operatoren in een expressie voorkomen, hangt de volgorde van de evaluatie af van de voorrangsregels.

Voor de wiskundige operatoren volgt Python de conventies uit de wiskunde:

- Haakjes hebben de hoogste voorrang en worden gebruikt om een bepaalde volgorde af te dwingen. Bijvoorbeeld: $2 * (3-1)$ is 4 en $(1+1) ** (5-2)$ is 8.

Haakjes zijn ook nuttig om een expressie gemakkelijker leesbaar te maken, zoals in $(minute * 100) / 60$, ook al verandert het eindresultaat hier niet mee.
- Machtsverheffen heeft de daarop volgende voorrang, dus $2 ** 1 + 1$ is 3 en niet 4, $3 * 1 ** 3$ is 3 en niet 27.
- Vermenigvuldigen en delen hebben dezelfde rang; deze is hoger dan optellen en aftrekken. Deze laatste hebben eveneens dezelfde rang. Dus $2 * 3 - 1$ is 5 en niet 4, $6 + 4 / 2$ is 8 en niet 5.
- Operatoren met dezelfde rang worden van links naar rechts geëvalueerd. Dus in de expressie $graden / 2 * pi$ wordt de deling eerst uitgevoerd en het resultaat vermenigvuldigt met pi. Om te delen door 2π verwissel je de operand of gebruik je haakjes.

Deze tabel rangschikt alle operatoren samen op van hoog naar laag op.

Operator	Beschrijving
**	Macht
* / % //	Vermenigvuldiging, Deling, Modulo, Floor division
+ -	Optelling, aftelling
>> <<	Rechts/links Bits operatoren
&	Bit exclusieve And
^	Bit exclusieve Or, reguliere Or
<= < > >=	Vergelijkingsoperatoren
<> == !=	Gelijkheidsoperatoren
= %= /= //= -= += *= **=	Toekeningsoperatoren
is, is not	Identiteit operatoren
in, not in	Verzameling operatoren
not, or, and	Logische operatoren



NEW MEDIA &
COMMUNICATION
TECHNOLOGY

Selectiestructuren

Inleiding

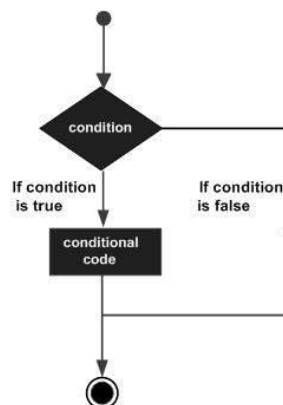
Tot op heden was alle geschreven code een opeenvolging van verschillende statements na elkaar waarbij alle statements telkens werden uitgevoerd. We spreken over een sequentie. Onderstaande code bevat een sequentie dat de som van twee ingetikte getallen verzorgt.

```
getal1 = int(input("Geef het eerste getal op: "))
getal2 = int(input("Geef tweede getal op: "))
som = getal1 + getal2
print("De som van {0} en {1} is {2}".format(getal1, getal2, som))
```

In deze routine zit er geen mogelijkheid om een andere bewerking uit te voeren. In veel gevallen mag bepaalde code maar uitgevoerd worden als er voldaan wordt aan een voorwaarde. We onderscheiden daarom enkelvoudige, tweevoudige en meervoudige keuzestructuren.

Enkelvoudige selectie

Enkel als de opgestelde voorwaarde WAAR is, wordt de sequentie binnenin uitgevoerd. Is de voorwaarde niet waar, dan wordt de sequentie binnenin niet uitgevoerd.



Syntax:

```
if (voorwaarde):
    sequentie_als_voorwaarde_waar_is
```

Opmerkingen:

- Het begin en einde van de sequentie wordt aangeduid via alignering. Heb daarom voldoende aandacht voor de alignering!
- De voorwaarde is een 'logisch statement' dat slechts twee oplossingen kan hebben: waar (true) of onwaar (false). In een voorwaarde kunnen vergelijkingsoperatoren alsook logische operatoren gebruikt worden.
- Als de voorwaarde waar is, zullen alle statements uitgevoerd worden. Indien de voorwaarde niet waar is, dan zullen de statements binnen de sequentie niet uitgevoerd worden en gaat men verder met de coderegels onder de sequentie.
- Python ziet elke niet-nul-waarde of niet-null-waarde (betekenis: zie later) als WAAR. Is de waarde 0 of null, dan is de voorwaarde NIET WAAR.
- Bestaat de code binnenin uit slechts één lijn, dan kunnen voorwaarde en sequentie op één lijn.

Enkele voorbeelden:

```
geboortejaar = 2000
if (geboortejaar > 1998):
    print("U bent nog geen 18!")
    print("Kom volgend jaar terug...")

if (geboortejaar == 2016) : print("Proficiat!")
```

Tweevoudige selectie

Bij deze keuzestructuur wordt er code uitgevoerd als aan de voorwaarde voldaan is, alsook (andere) code uitgevoerd als de voorwaarde niet voldaan is. Hierbij geldt volgende syntax:

```
if (voorwaarde):
    sequentie_als_voorwaarde_waar_is
else:
    sequentie_als_voorwaarde_niet_waar_is
```

Opmerking:

- Heb opnieuw voldoende aandacht voor de alignering!

Voorbeeld:

```
score = int(input("Geef uw score op: "))
if (score >= 10):
    print("U bent geslaagd!")
else:
    print("Helaas, volgende keer beter!")
```

Meervoudige selectie

Onder de meervoudige selectie groeperen we alle gevallen waar er meerdere voorwaarden toegelaten zijn. De syntax ziet er als volgt uit:

```
if (voorwaarde1):
    sequentie_als_voorwaarde1_waar_is
elif (voorwaarde2):
    sequentie_als_voorwaarde2_waar_is
elif (voorwaarde3):
    sequentie_als_voorwaarde3_waar_is
...
else:
    sequentie_als_geen_van_bovenstaande_voorwaarden_waar_zijn
```

Voorbeeld:

```
if (age < 0):
    print("Foutieve ingave!")
elif (age == 1):
    print("Deze leeftijd komt overeen met 14 mensenjaren.")
elif (age == 2):
    print("Deze leeftijd komt overeen met 22 mensenjaren.")
else:
    human = 22 + (age - 2) * 5
    print("Deze leeftijd komt overeen met {0} mensenjaren".format(human))
```

Switch in Python?

Binnen python bestaat er geen switch-commando zoals aanwezig in de programmeertalen C#, Java, ... Via if en elif-statements kan men gelukkig het zelfde resultaat bereiken. In het geval men een zeer grote verzameling moet overlopen, kan je gebruik maken van een dictionary om zo de juiste functie op te roepen. Beide zaken worden later uitvoerig toegelicht.

Deelproblemen

Inleiding

Het onderscheiden van verschillende deelproblemen binnen een applicatie is een cruciale stap in de opbouw van de applicatie. Vaak start men zeer vaag met de verschillende te realiseren taken, maar verfijnt men deze verder tot kleinere (niet-opdeelbare) eenheden. In het ontdekken van de verschillende deelproblemen, alsook in de vertaling naar methodes komt vaak ervaring bekijken. In deze cursus worden verschillende stijlregels afgedwongen die tot doel hebben het gebruik van deelproblemen (en onderliggende geschreven methodes en functiemethodes) te stimuleren.

Voordelen bij het gebruik van deelproblemen zijn:

- herhaling van code vermijden;
- code in meerdere programma's en toepassingen kunnen hergebruiken;
- om de complexiteit van een programma in kleinere, beter beheersbare delen kunnen opsplitsen;
- de leesbaarheid van een programma verbeteren;
- een deel van een programma te verbergen of beschermen.

Functies

In Python hanteert men de term 'functie' voor een groep van bij elkaar horende coderegels dat zo verantwoordelijk is voor één specifiek onderdeel/actie van jouw applicatie. Via de functienaam kan deze code op verschillende plaatsen meermaals aangeroepen worden. In andere programmeertalen hanteert men ook de termen 'subroutine' en 'procedure'.

Functies worden specifiek aangemaakt om herbruikbaar te zijn.

Syntax van een functie:

```
def function-name(Parameter list):  
    statements (function-body)
```

Een aantal opmerkingen hierbij zijn:

- 'def' is een sleutelwoord dat aangeeft dat dit een functiedefinitie is.
- de eerste regel van een functie wordt ook het '**functiehoofd**' genoemd.
- een functie identificeert zich via een betekenisvolle functienaam;
- een functienaam start met een kleine letter: bij een samenstelling van meerdere woorden worden er vaak underscores gebruikt;
- een functie methode kan bij de start specifieke hulpwaarden ('parameters') tussen haakjes meekrijgen. Een functie kan ook geen parameters hebben.
- Het functiehoofd eindigt met een dubbelpunt.
- het begin en einde van een functie wordt via alignering afgebakend: men spreekt over de 'body' van een functie.
- nadat de functie uitgevoerd is, keert het programma terug naar de plaats waar de functie opgeroepen werd.
- Je kan vanuit een functie een andere functie oproepen.
- In tegenstelling tot andere programmeertalen kan je in python het functiehoofd hergebruiken. Bij de functie-aanroep wordt de laatste versie gebruikt.

Opgelet: gezien python een script taal is, kan je enkel functies oproepen die vóór jouw huidige positie in het codebestand staan.

Functie zonder parameters

In onderstaand voorbeeld zijn 3 functies zonder parameters opgelijst.

```
def print_mijn_adres():
    print("Graaf Karel de Goedelaan 5")
    print("8500 Kortrijk")

def print_telefoonnummer():
    print("Tel.: 056/24.12.44")

def print_mijn_contactgegevens():
    print_mijn_adres()
    print_telefoonnummer()

#Test
print_mijn_contactgegevens()
```

Functie met parameter(s)

Aan een functie kunnen ook waarden doorgegeven worden. Hierdoor kan de functie meerdere malen aangeroepen worden, telkens met verschillende waarden. In de literatuur worden volgende termen gehanteerd (hoewel ze soms door elkaar gebruikt worden):

- Naast de functienaam staan tussen haakjes de *parameters*. Zij sommen precies op wat de functie nodig heeft.
- De doorgegeven waarden op de plaats waar de functie aangeroepen wordt, heten de *argumenten*.

Een functie kan op verschillende manieren van parameters voorzien worden:

- verplichte parameter(s)
- parameters zijn voorzien van keywords
- parameters hebben default waarde
- een variabel aantal argumenten

Verplichte door te geven argumenten/parameter(s)

Wanneer een functie één of meerdere parameters vermeldt, dan moeten deze ook in de juiste volgorde doorgegeven worden. Een voorbeeld hiervan is:

```
def print_naam_en_voornaam(naam, voornaam):
    print("%s, %s" % (naam, voornaam))

#test
print_naam_en_voornaam("Walcarius", "Stijn")
```

Vermelden van de parameternamen bij functie aanroeping

Bij het oproepen van een functie met parameters kan men ook de parameternamen vermelden. Dit laat jou toe de parameters in een andere volgorde door te geven (of zelfs niet te vermelden indien de functie het toelaat).

```
def print_naam_en_voornaam(naam, voornaam):
    print("%s, %s" % (naam, voornaam))

print_naam_en_voornaam(voornaam="Stijn", naam="Walcarius")
```

Default waarde bij parameter

Een default parameter is een parameter die in het functiehoofd over een default waarde beschikt. Hierdoor hoeft men de parameter niet meer verplicht door te geven. Indien er bij de functie-aanroep niets doorgegeven wordt, dan wordt de default waarde gebruikt. Bijvoorbeeld:

```
def print_naam_en_voornaam(naam, voornaam, geboortejaar=2000):
    print("%s, %s (%s)" % (naam, voornaam, geboortejaar))

print_naam_en_voornaam(voornaam="Stijn", naam="Walcarius")
print_naam_en_voornaam("Pinket", "Lies")
print_naam_en_voornaam("Duchi", "Frederik", 1985)
```

Resultaat:

Walcarius, Stijn (2000)
Pinket, Lies (2000)
Duchi, Frederik (1985)

Variabel aantal parameters

In sommige situaties weet men niet hoeveel parameters men zal doorgeven. In deze gevallen kan men opteren om met een '*' - aanduiding te werken. In deze situatie kan het aantal nul, één of meer zijn. Dit is ideaal om een lijst van bijvoorbeeld getallen door te geven.

Een * wordt geplaatst vóór de parameternaam.

In onderstaand voorbeeld vormt de parameter 'scores' een variabele aantal getallen.

```
def print_scores(modulenaam, *scores):
    print("Dit zijn alle scores van de module %s" % modulenaam)
    for score in scores:
        print(score)

#test 1
print_scores("Basic Programming", 10, 18, 5, 12, 11, 12)
#test 2
print_scores("Mobile App Development", 19)
#test 3
print_scores("Maths")
#print_scores() #uitvoeringsfout!
```

Return statement

Het returnstatement beëindigt de functie. Standaard wordt er niets door de aanroepende functie terug gegeven. Toch is dit ook mogelijk. In onderstaand voorbeeld wordt de som van de twee parameters bepaald en via het return-commando teruggegeven.

```
def som_getallen(getal1, getal2):
    som = getal1 + getal2
    print("De som is %d" % som)
    return som

resultaat = som_getallen(12, 36)
print("Het resultaat is %d" % resultaat)
```

Output:

De som is 48
Het resultaat is 48

Bereik van variabelen

Het bereik ('scope' is een meer gehanteerde informaticaterm) van een variabele drukt uit wat de plaats is waar deze variabele gekend is. Binnen Python onderscheidt men

- **lokale** variabele: de variabele krijgt een waarde in een functie. Deze nieuwe waarde is enkel in deze functie gekend en wijzigbaar. Zie voorbeeld hierboven.
- **globale** variabele (of modulevariabele): de variabele krijgt buiten de functies een waarde. Hierdoor is de variabele in elke functie benaderbaar.

Toch verschilt Python op dit vlak met de klassiekere programmeertalen zoals Java en C#. Onderstaande voorbeelden vertellen waarom.

Voorbeeld 1:

<pre>def print_boodschap(): print(boodschap) boodschap = "NMCT rules!" print_boodschap()</pre>	<p>Output: NMCT rules!</p> <p>Opm: in de functie print_boodschap wordt de globale variabele boodschap gebruikt. Dit levert geen probleem op aangezien deze een waarde krijgt voordat de functie opgeroepen wordt.</p>
<pre>def print_boodschap(): boodschap = "Howest is de max." print(boodschap) boodschap = "NMCT rules!" print_boodschap() print(boodschap)</pre>	<p>Output: Howest is de max. NMCT rules!</p> <p>Opm: in de functie print_boodschap krijgt de variabele boodschap een 'nieuwe' waarde. In werkelijkheid is de variabele boodschap in de functie een lokale variabele (naast de globale variabele)</p>
<pre>def print_boodschap(): print(boodschap) boodschap = "Howest is de max." print(boodschap) boodschap = "NMCT rules!" print_boodschap() print(boodschap)</pre>	<p>Output: we krijgen een error (UnboundLocalError: local variable 'boodschap' referenced before assignment). Doordat de variabele boodschap in de functie een waarde krijgt, is deze lokaal. De variabele mag pas gebruikt worden na de toekenning van een waarde aan de variabele. Het is hier een misvatting dat dan 'voorlopig de globale variabele zal gebruikt worden'</p>
<pre>def print_boodschap(boodschap): print(boodschap) boodschap = "Howest is de max." print(boodschap) boodschap = "NMCT rules!" print_boodschap(boodschap) print(boodschap)</pre>	<p>Output: NMCT rules! Howest is de max. NMCT rules!</p> <p>De parameter zorgt ervoor dat de fout uit voorgaande situatie hier niet voorkomt. De waarde van de parameter wordt op de eerste lijn afgeprint. Een parameter gedraagt zich als een lokale variabele.</p>

Toch bestaat er een manier die er kan voor zorgen dat binnen een functie ook steeds op de globale variabele gewerkt wordt. Dit gebeurt via het keyword 'global'.

<pre>def print_boodschap(): global boodschap boodschap = "Howest is de max." print(boodschap) boodschap = "NMCT rules!" print_boodschap() print(boodschap)</pre>	<p>Output: Howest is de max. Howest is de max.</p> <p>Opm: via het keyword global laten we weten dat we in deze functie gebruik maken van de globale variabele. Er wordt geen lokale variabele aangemaakt.</p>
---	--

Oefening: bepaal de output van onderstaande code:

```
def foo(x, y):  
    global a  
    a = 42  
    x, y = y, x  
    b = 33  
    b = 17  
    c = 100  
    print (a, b, x, y)  
  
a, b, x, y = 1, 15, 3, 4  
foo(17, 4)  
print (a, b, x, y)
```

Deelproblemen?

Een deelprobleem is een gedeelte van een complex probleem dat als afzonderlijk probleem kan aangemerkt worden. Problemen die uit de (beroeps)praktijk worden gekozen, zijn lang niet altijd enkelvoudig maar bestaan vaak uit een aantal deelproblemen. Een verstandige manier van aanpak om het eigenlijke probleem aan te pakken, is vooraf na te gaan welke deelproblemen eerst opgelost dienen te worden voordat aan het eigenlijke probleem gewerkt kan worden.

Om een complex probleem aan te kunnen pakken, dienen eerst de deelproblemen geïdentificeerd te worden. Daarvoor kan het volgende stappenplan gebruikt worden:

1. Maak een overzicht van de verschillende aspecten van het betreffende probleem. Dit mag uitgebreid zijn.
2. Selecteer die aspecten uit het overzicht die van essentieel belang zijn voor de oplossing van het probleem en controleer of deze bijdragen aan het ontstaan van het probleem. Indien zo, dan heb je een deelprobleem.
3. Volgorde aanbrengen: Ga na welke deelproblemen eerst opgelost moeten worden voordat het eigenlijke probleem aangepakt kan worden.

Het grote voordeel van het consequent gebruiken van deelproblemen is het verhogen van de leesbaarheid en de onderhoudbaarheid van de code.

Praktisch vertaald naar deze module: een 'goed' (niet meer opdeelbaar) deelprobleem vertaalt zich meestal in een afzonderlijke functie van +/- max 15 regels code die een op zich staande functionaliteit aflevert. Enkele voorbeelden kunnen zijn:

- het opslaan van wijzigingen naar een bestand;
- het tellen van het aantal lijnen uit een bestand;
- het gemiddelde berekenen van een doorgegeven reeks getallen;

String

Strings van tekens zijn erg belangrijk voor programmeertalen. Ze houden niet enkel een verzameling van tekens bij. De programmeertaal Python biedt daarnaast ook een uitgebreide groep verzameling van methodes aan om strings te kunnen bewerken.

Algemeen

Een string-variabele aanmaken is in Python heel eenvoudig door een tekstwaarde aan een variabele toe te kennen. Python handelt zowel enkele als dubbele aanhalingstekens af.

```
opleidingsnaam = "NMCT"
hogeschool = 'Howest'
print("De opleiding %s bevindt zich aan de hogeschool %s" % (opleidingsnaam, hogeschool))
```

Python heeft geen character datatype zoals andere programmeertalen. Eén enkel karakter wordt eveneens aanzien als een string.

Om een substring te benaderen maken we gebruik van vierkante haakjes. Het begin en einde van een substring wordt via [] en [:] aangeduid. Hou er mee rekening dat het eerste karakter op positie 0 staat.

Het aanpassen van de waarde van een string-variabele gebeurt eenvoudig door een nieuwe waarde aan de string-variabele toe te kennen.

```
favoriere_module = "Web"
favoriere_module = "Basic Programming" #nieuwe waarde
```

In een string kunnen ook 'niet-printbare' karakters voorkomen zoals een nieuwe lijn. In het vakjargon spreekt men over de zgn. 'escape karakters'. Welke output geeft onderstaande code?

```
favoriere_module = "\tBasic\n\tProgramming"
print(favoriere_module)
```

String operatoren

Onderstaande lijst somt alle bijzondere operatoren voor strings op:

Operator	Beschrijving	Voorbeeld
+	Concatenatie van 2 strings	
*	Herhalen van een string	
[]	Geeft het karakter op een welbepaalde positie terug	
[:]	Substring	
in	Te gebruiken in een voorwaarde waarmee men controleert of een karakter in een string voorkomt	
not in	Te gebruiken in een voorwaarde waarmee men controleert of een karakter in een string niet voorkomt	
%	Te gebruiken bij het afprinten van een variabelewaarde in een string (zie eerder)	

In de verschillende labo's zullen elk van deze operatoren verder onderzocht en besproken worden.

Weergave van Strings

Een uitvoerige beschrijving van hoe strings in combinatie met andere parameters kunnen weergegeven worden, vindt u eerder onder het onderdeel 'Output van variabelen'.

Voorbeeld:

```
print ("Mijn naam is %s. Mijn gewicht is %d kg en ik ben al %g m groot!" % ('Hanne', 21, 1.10))
```

Resultaat:

Mijn naam is Hanne. Mijn gewicht is 21 kg en ik ben al 1.1 m groot!

Op onderstaande link vindt u meer informatie alsook verdere mogelijkheden.

<https://docs.python.org/3.7/library/string.html>

Stringfuncties

We bekijken nu de belangrijkste methodes van de String. De tabel hieronder zijn telkens functies met een terugkeer waarde, m.a.w. ze geven steeds een resultaat terug, en laten de oorspronkelijke string ongewijzigd! Afhankelijk van de gebruikte methode moeten nul, één of meerdere argumenten tussen de haakjes meegegeven worden.

Onderstaande (onvolledige) tabel geeft een overzicht van de belangrijkste string-functies.

Stringfunctie	Beschrijving	Voorbeeld
capitalize()	Geeft een kopie van de string terug met de eerste letter in hoofdletter	naam = "jan" print(naam.capitalize())
center(totale_breedte, opvul karakter)	Geeft een kopie terug waarbij de string gecentreerd wordt en met een karakter vooraan/achteraan wordt opgevuld totdat de totale lengte bekomen is	naam = "jan" print(naam.center(20, '*'))
count(substring, start, end)	Telt het aantal keer dat de substring in de string voorkomt, vanaf de startpositie (default: start string) tem eindpositie (default: einde string)	naam="antwerpen" print(naam.count('an', 0))
endswith(suffix[,start[,end]])	Gaat na of de string eindigt met suffix, eventueel binnen een deel van de string kijken	str = "this is string example....wow!!!"; print(str.endswith('wow', 20))
find(substring, beg=0, end=len(string))	Gaat de positie na waar de substring in de string voorkomt; eventueel wordt extra zoekbereik opgegeven (default vanaf start, tot einde van string). Indien niet aanwezig wordt -1 terug gegeven	str = "this is string example....wow!!!"; print(str.find('is')) print(str.find('is', 10))
index(substring, beg=0, end=len(string))	Idem als find, maar indien substring niet voorkomt wordt er een fout (ValueError) gegenereerd	str = "this is string example....wow!!!"; print(str.index('is')) print(str.index('is', 10))
isalnum()	Geeft waar terug als string min 1 karakter bevat én enkel bestaat uit letters/cijfers	str = "this is string example....wow!!!"; print(str.isalnum())
isdigit()	Geeft true terug als de string enkel cijfers bevat. Anders false	
isalpha()	Geeft true terug als de string enkel letters bevat. Anders false.	
islower()/isupper()	Geeft true terug als de string enkel uit kleine/hoofd letters bestaat. Anders false.	
isnumeric()	Geeft true terug als een Unicode-string enkel uit cijfers bestaat. Anders false. (Om	test = u"this2009"; print (test.isnumeric())

	een string als Unicode te definiëren gebruik de letter 'u' voor het eerste aanhalingsteken)	
isspace()	Geeft true terug als de string enkel uit spaties bestaat. False in het andere geval	
len()	Geeft de lengte van een string terug	
lower()/upper()	Zet een string om in kleine of hoofdletters	
lstrip()/rstrip()	Geeft een kopie terug waarbij alle spaties links/rechts verwijderd zijn.	
strip()	Combinatie van lstrip en rstrip	
replace(old, new [,max])	Geeft een kopie terug waarbij <i>old</i> is vervangen door <i>new</i> . Optioneel kan men opgeven hoeveel keer max de vervanging mag gebeuren (startend vanaf begin string)	<pre>str = "this is string example....wow!!!"; print(str.replace('is', 'was',))</pre>
title()	Geeft een kopie terug waarbij elk woord start met hoofdletter	
swapcase()	Geeft kopie terug waarbij kleine letters hoofdletter worden, en omgekeerd.	

Meer string-functies zijn te vinden op:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

Herhalingen (lussen)

Inleiding

We hebben al gezien dat een applicatie een opeenvolging van instructies (sequentie) kan uitvoeren. We gaan nu bekijken hoe we in een applicatie een opeenvolging van instructies een aantal keer na elkaar kunnen laten herhalen. Een deel van de kracht van computers komt neer door het vermogen deze herhalingen razendsnel na elkaar te laten uitvoeren. In het programmeerjargon spreken we van een lus of iteratie. We onderscheiden volgende soorten:

- Doorlopen van een getallen reeks: For-lus
- Voorwaardelijk herhaling: While-lus
- Doorlopen van een lijst: For-lus (zie verder)
- Recursie

For-lus & range

De intern aanwezige functie `range()` laat toe om over reeks van getallen samen te stellen, die nadien met een for-lus kunnen overlopen worden.

`range(n)` genereert een opeenvolgende lijst van integer getallen, startend vanaf 0 en eindigend bij (n -1).

Bijvoorbeeld:

`range(10)` → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

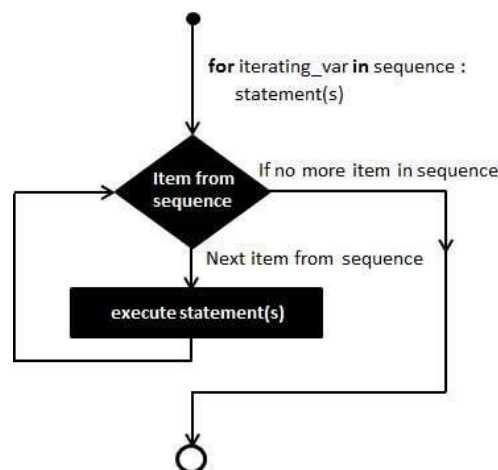
`range(min, max)` genereert een opeenvolgende lijst van integer getallen, startend vanaf het opgegeven min en eindigend bij (n -1). Bijvoorbeeld:

`range(4,10)` → [4, 5, 6, 7, 8, 9]

`range(min, max, step)` genereert een opeenvolgende lijst van integer getallen, startend vanaf het opgegeven min en eindigend bij (n -1) met een stapgrootte `step`. Bijvoorbeeld:

`range(4,50,5)` → [4, 9, 14, 19, 24, 29, 34, 39, 44, 49]

De `range()` functie is handig in combinatie met een for-lus. Bij een for-lus wordt een verzameling elementen één voor één overlopen. Voor elk element wordt de code binnenin de for-lus herhaald.



In onderstaand voorbeeld wordt de som van een reeks getallen berekend.

```

som = 0
for i in range(1,101):
    som = som + i
print ("De som van de eerste 100 getallen is %d" % som)
  
```

Voorwaardelijk herhaling: while

Onder deze categorie plaatsen we alle lusstructuren waar de herhaling van een voorwaarde afhangt. Hierdoor is het aantal keer dat de lus herhaald wordt, niet vooraf gekend. In de programmeertaal Python beschikken we over de while-lus. Vertaald naar het Nederlands luidt dit als volgt: “zolang de voorwaarde geldig is, herhaal je telkens opnieuw volgende opdracht(en)”.

De syntax ziet er als volgt uit:

```
while voorwaarde:
    statement(s)
```

Hierbij kunnen de statement(s) zowel één of meerdere statements zijn. De voorwaarde kan gelijk welke expressie zijn. De statements worden herhaald zolang de conditie waar is. Opgelet, alle statements die dezelfde alignering hebben, behoren tot de while-lus. Wanneer de conditie onwaar wordt, wordt de while-lus beëindigd en gaat hij verder met de code onder de while-lus.

Voorbeeld:

Code

```
count = 0
while (count < 9):
    print ('The count is: %d' % count)
    count = count + 1

print ("Einde!")

end = 100
som = 0
index = 1

while index <= end:
    som = som + index
    index += 1

print("Sum of 1 until %d: %d" % (end, som))
```

Uitvoering

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Einde!

Sum of 1 until 100: 5050
```

Oneindige lus

Bij deze lusstructuren is het opstellen van een goede voorwaarde cruciaal. In het slechtste geval verkrijgt men een oneindige lus (m.a.w., de lus kan niet meer verlaten worden). Toch kan een oneindige lus in bijvoorbeeld een cliënt/server toepassing nuttig zijn: de server blijft continue draaien, waardoor de cliënten met de server verbinding kunnen maken en kunnen communiceren.

Break

Het break-statement beëindigt de lus en herneemt de uitvoering bij het eerstvolgend statement na de lus. De break kan zowel in een while-lus als een for-lus gebruikt worden.

Voorbeeld:

Code	Uitvoering
------	------------

<pre>var = 10 while var > 0: print('Huidige waarde van variabele is: %d' % var) var = var - 1 if var == 5: break print("Ten einde!")</pre>	<p>Huidige waarde van variabele is: 10 Huidige waarde van variabele is: 9 Huidige waarde van variabele is: 8 Huidige waarde van variabele is: 7 Huidige waarde van variabele is: 6 Ten einde!</p>
--	--

Continue

Het continue-statement zorgt ervoor dat de actuele lus gestopt wordt, en er onmiddellijk naar de voorwaarde van de lus teruggekeerd wordt. 'Continue' kan zowel in een while- als een for-lus gebruikt worden.

Voorbeeld:

Code	Uitvoering
<pre>var = 6 while var > 0: var = var - 1 if var == 2: continue print('Huidige waarde van variabele is: %d' % var) print("Ten einde!")</pre>	<p>Huidige waarde van variabele is: 5 Huidige waarde van variabele is: 4 Huidige waarde van variabele is: 3 Huidige waarde van variabele is: 1 Huidige waarde van variabele is: 0 Ten einde!</p>

Datastructuren

Tot nu toe hebben we variabelen gedefinieerd die individueel zijn en op zich zelf staan. Bijvoorbeeld:

```
modulenaam = "Basic Programming"
jaartal = 2016
```

Deze variabelen leiden hun eigen leven. We kunnen ze voorstellen als individuele geheugenplaatsen waarbij individuele namen bij horen. In tegenstelling hiermee komen we in de praktijk vaak gegevens tegen die niet op zichzelf staan, maar thuis horen in een verzameling. Voorbeelden hiervan zijn een telefoonboek, een klas studenten,... In de informatica-wereld spreken we van gegevensstructuren of datastructuren.

De basis datastructuur in Python is de **sequentie**. Elk element in een sequentie heeft een positie of index. Het eerste element zit op positie 0, de volgende op 1, enz. Python beschikt over deze verschillende soorten sequenties:

- Lists
- Tuples
- Strings
- Range-objecten

Er is gemeenschappelijke functionaliteit over deze verschillende sequenties: 'indexing, slicing, toevoegen, vermenigvuldigen, controleren op aanwezigheid van een element. Bovendien heeft Python functionaliteit voor het opvragen van de lengte van een sequentie, het opvragen van grootste en kleinste element.

In bovenstaande lijst staat ook Strings als sequentie vermeld. Dit is terecht aangezien elk karakter binnen een string een welbepaalde positie heeft en dus ook op deze manier kan opgevraagd worden.

Code	Resultaat
<pre>test_woord = "Basic Programming" print(test_woord[0]) print(test_woord[6:]) print(test_woord[6:9])</pre>	<pre>B Programming Pro</pre>

Naast de sequenties worden ook de datastructuur set en dictionary toegelicht. In tegenstelling tot de sequenties hebben de elementen geen unieke index.

List

List is de meest veelzijdige datastructuur aanwezig in Python. Deze wordt gecodeerd als een lijst van elementen tussen vierkante haakjes waarbij de elementen door komma's van elkaar gescheiden worden. Belangrijk om hierbij op te merken, is dat de elementen niet van hetzelfde datatype hoeven te zijn. Ook hier staat het eerste element op positie 0, enz. Een list is daarmee een geordende datastructuur.

Voorbeeld:

```
list1 = ['1NMCT1', '1NMCT2', 1997, 2000]
list2 = [1, 2, 3, 4, 5]
list3 = ["a", "b", "c", "d"]
list4 = [True, False, False, True]
```

Het opvragen van één of meerdere elementen gebeurt via de positie(s).

Code	Resultaat
------	-----------

```
print("list1[0]: ", list1[0])          list1[0]: 1NMCT1
print("list2[1:5]: ", list2[1:5])    list2[1:5]: [2, 3, 4, 5]
```

Het aanpassen en verwijderen van een element in een list gebeurt eveneens via de positie.

```
list4[2] = True           #update
del list4[3]              #delete
print(list4)
```

Basis operatoren

Lists kunnen overweg met de + en * operator. De + operator komt overeen met een samenvoeging van 2 lists, de * operator staat voor een herhaling van een list.

De *in* operator wordt gebruikt in een controlestructuur om na te gaan of een element in de list aanwezig is. De *in* operator kan ook gebruikt worden in een for-lus om elk element binnen een list te overlopen.

Enkele voorbeelden:

Code

```
print(list1 + list2)
print(list2 * 2)
if (7 in list2): print("element 7 is aanwezig")
for x in list3:
    print(str.upper(x))
```

Resultaat

```
['1NMCT1', '1NMCT2', 1997, 2000, 1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
A
B
C
D
```

Ingebouwde functies & methodes

Ingebouwde list-functies in Python	Beschrijving
len(list)	Geeft het aantal elementen in de list terug
max(list)	Geeft het maximum uit een list terug
min(list)	Geeft het minimum uit een list terug
list(seq)	Zet een sequentie om in een list

Ingebouwde list-methodes in Python	Beschrijving
list.append(obj)	Voegt object toe aan list
list.count(obj)	Telt hoeveel keer een object in de list voorkomt
list.extend(seq)	Breidt de list uit met de sequentie
list.index(obj)	Geeft het eerste voorkomen van een obj in de list terug.
list.insert(index,obj)	Voegt een object toe op doorgegeven positie
list.pop([index])	Verwijdert het laatste element (of element op doorgegeven positie) uit de list
list.remove(obj)	Verwijdert een object uit de list
list.reverse()	Keert de list om
List.sort([func])	Sorteert de list (eventueel via doorgegeven sorteerfunctionaliteit)

Tuple

Een tuple is een geordende datastructuur met 'onwijzigbare' objecten. Net zoals lists zijn tuples sequenties. Het grootste verschil met tuples is dat na creatie deze laatste niet meer kunnen gewijzigd worden.

Om een tuple te definiëren gebruiken we ronde haakjes. Ook hier worden de elementen door komma's gescheiden. Een lege tuple wordt gecodeerd als twee haakjes. Bij een tuple met slechts één element moet er toch een komma gebruikt worden.

Voorbeelden zijn:

```
t1 = ("1NMCT", "2NMCT", "3NMCT")
t2 = ("1Devine", "2Devine", "3Devine")
```

Tuples zijn onwijzigbaar wat inhoudt dat de interne elementen niet meer kunnen aangepast, gewijzigd of verwijderd worden. Een nieuwe tuple aanmaken als een samenvoeging van twee bestaande tuples kan wel.

```
totaal = t1 + t2
print(totaal)
```

Basis operatoren

De basisoperatoren zijn identiek als Lists.

Ingebouwde functies & methodes

Ingebouwde tuple-functies in Python	Beschrijving
cmp(tuple1, tuple2)	Vergelijkt de elementen van beide tuples
len(tuple)	Geeft het aantal elementen in de tuple terug
max(tuple)	Geeft het maximum binnen een tuple terug
min(tuple)	Geeft het minimum in een tuple terug
tuple(seq)	Zet een sequentie om in een tuple

Set

Een set is een ongeordende datastructuur met **unieke** elementen. Duplicaten worden niet toegelaten. Doordat de sequentie geen ordening kent, hebben de verschillende elementen ook geen unieke index. Hierdoor valt de set niet onder de sequenties.

Om een set te definiëren gebruiken we accolades. Ook hier worden de elementen door komma's van elkaar gescheiden. Een lege set wordt gecodeerd als twee accolades. Merk op dat bij de opsomming van verschillende elementen dubbels mogen aanwezig zijn. Deze dubbels worden gedetecteerd en tot één element terug gebracht.

Enkele voorbeelden zijn:

```
geboortejaren = {1998, 1997, 1995, 1996, 1999, 2000, 1997, 1996, 2001}
print("De %d unieke geboortejaren zijn: " % len(geboortejaren))
print(geboortejaren)
```

levert als output:

De 7 unieke geboortejaren zijn:
{1995, 1996, 1997, 1998, 1999, 2000, 2001}

```
unieke_letters = set("stijn walcarius")
print(unieke_letters)
```

levert als output:

```
{'c', 's', 'w', 'u', 'r', 'j', 'n', 'i', 't', 'a', ' ', 'l'}
```

```
print(unieke_letters[0])
```

levert een uitvoeringsfout op:

TypeError: 'set' object does not support indexing

Set heeft interessante wiskundige operaties zoals unie, doorsnede, verschillen, etc. Meer informatie is te vinden op

<https://docs.python.org/3.5/tutorial/datastructures.html?highlight=set#sets>

Basis operatoren

De basisoperatoren zijn identiek als bij de List.

Dictionary

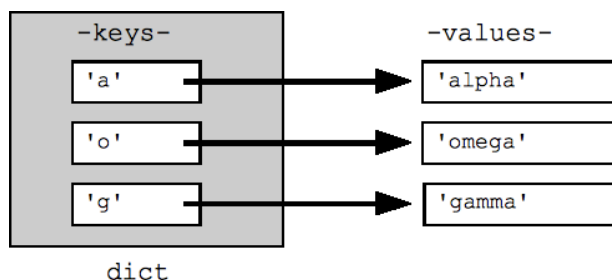
Een zeer populaire datastructuur binnen python is de **dictionary**. De elementen uit een dictionary hebben geen unieke positie, maar worden opgevraagd met hun key. Naast de key heeft elk element ook een value. Wanneer men een nieuw element aan de dictionary wenst toe te voegen, dan kan dit enkel met een nog niet-gebruikte key.

Een voorbeeld:

```
mijn_dict = {"a": "alpha", "o": "omega", "g": "gamma"}
print(mijn_dict)
print("Opvragen van een waarde uit de dictionary ahv zijn key:")
print(mijn_dict.get("a", "niets gevonden voor deze key"))
print(mijn_dict.get("z", "niets gevonden voor deze key"))
```

geeft volgende output:

```
{'g': 'gamma', 'o': 'omega', 'a': 'alpha'}
Opvragen van een waarde uit de dictionary ahv zijn key:
alpha
niets gevonden voor deze key
```



Belangrijke operaties zoals het toevoegen, verwijderen, controleren op aanwezigheid van een key worden hieronder geïllustreerd:

```
mijn_dict["t"] = "theta"           #toevoegen van key & value
del mijn_dict["o"]                 #verwijderen van value mbv key
print(mijn_dict)

#list(d.keys()) on a dictionary returns a list of all the keys used in the
dictionary
if ("t" in mijn_dict.keys()):      #controle op aanwezigheid van key
    print("key is aanwezig")
else:
    print("key is niet aanwezig")
```

levert als output:

```
{'a': 'alpha', 't': 'theta', 'g': 'gamma'}
key is aanwezig
```

Ondanks dat een dictionary een ongeordende collectie is, kunnen de elementen toch met een for-lus overlopen worden. Hierbij kan je zowel de keys afzonderlijk, de values afzonderlijk of de keys en values samen overlopen worden. Voorbeeld:

```
print("Dit zijn de keys die in gebruik zijn: ")
for key in mijn_dict.keys():
    print("key: %s " % key)

print("Dit zijn de values uit de dictionary: ")
for key in mijn_dict.values():
    print("Value: %s " % key)

print("Dit zijn de keys én values uit de dictionary: ")
for key, val in mijn_dict.items():
    print("key: %s -> value: %s" % (key, val))
```

Input/output

Dit hoofdstuk staat stil bij de verschillende input/output functies in Python. Dit is slechts een beperkt overzicht. Voor meer functies verwijzen we door naar de officiële documentatie:

<https://docs.python.org/3/tutorial/inputoutput.html>

Printen naar het scherm

De meest eenvoudige manier om output te genereren is door gebruik te maken van print-statements naar het scherm. De print-functie zet de expressie binnenin om naar een string waarna deze getoond wordt.

```
print("NMCT is great!")
```

Input-functie

Tegenover het print-commando staat de input-functie. Deze functie leest data uit het keyboard en levert deze af als een string. Via conversie-methodes kan deze string naar een ander datatype eventueel omgezet worden.

```
avator = input("Geef de naam van jouw avator op: ")
leeftijd = int(input("Wat is zijn leeftijd?"))
print("Jouw avator heet {0} en is {1} oud".format(avator, leeftijd))
```

Opmerking: in het input-commando kan een volledige expressie verwerkt zitten:

```
favoriete_kleur = input("Kies uw favoriete kleur uit %s" % str(["red", "green", "blue"]))
print("Jouw favoriete kleur is: " + favoriete_kleur)
```

Openen van een bestand

Vooraleer een bestand te kunnen inlezen, moet het bestand geopend worden. Dit gebeurt via de ingebouwde methode `open()`. Deze methode maakt een **file-object** aan waarop talrijke andere methodes op losgelaten kunnen worden.

De syntax van de open-methode ziet er als volgt uit:

```
file object = open(file_name [, access_mode][, buffering])
```

Waarbij

- `file_name`: bestandsnaam
- `access_mode` (optioneel): verduidelijkt de toegangsstatus van het bestand na openen
- `buffering` (optioneel): 0 betekent geen buffering; 1 betekent line buffering, bij groter dan 1 wordt de doorgegeven waarde de buffergrootte

Voorbeeld:

```
mijn_bestand = open("resultaten.txt", "w+")
```

Onderstaande tabel geeft de verschillende `access_modes` weer:

Mode	Beschrijving
------	--------------

r	Open het bestand enkel om te lezen. De cursor wordt vooraan geplaatst. Dit is de default mode.
rb	Open het bestand om binair in te lezen. De cursor wordt vooraan geplaatst.
r+	Open het bestand om te lezen en te schrijven. Cursor wordt vooraan geplaatst.
rb+	Open het bestand om te lezen en te schrijven in binair formaat. Cursor wordt vooraan geplaatst
w	Open het bestand enkel om te schrijven. Als het bestand reeds bestaat, dan wordt het overschreven. Bestaat het nog niet, dan wordt het bestand eerst aangemaakt.
wb	Open het bestand om te schrijven in binair formaat. Als het bestand reeds bestaat, dan wordt het overschreven. Bestaat het nog niet, dan wordt het bestand eerst aangemaakt.
w+	Open het bestand om in te lezen en naar toe te schrijven. Als het bestand reeds bestaat, dan wordt het overschreven. Bestaat het nog niet, dan wordt het bestand eerst aangemaakt.
wb+	Open het bestand om in te lezen en naar toe te schrijven in binair formaat. Als het bestand reeds bestaat, dan wordt het overschreven. Bestaat het nog niet, dan wordt het bestand eerst aangemaakt.
a	Open het bestand om achteraan data toe te voegen. Als het bestand nog niet bestaat, dan wordt het bestand eerst aangemaakt.
ab	Open het bestand om achteraan data in binair formaat toe te voegen. Als het bestand nog niet bestaat, dan wordt het bestand eerst aangemaakt
a+	Open het bestand om achteraan data toe te voegen of om het bestand in te lezen. Als het bestand nog niet bestaat, dan wordt het bestand eerst aangemaakt.
ab+	Open het bestand om achteraan data toe te voegen of om het bestand in te lezen in binair formaat. Als het bestand nog niet bestaat, dan wordt het bestand eerst aangemaakt.

Status van een file-object

Eenmaal het bestand geopend is, kan je informatie over het bestand gaan inwinnen via onderstaande methodes:

- file.closed: geeft true terug als het bestand gesloten is; false in het andere geval;
- file.mode: geeft de toegangsstatus terug;
- file.name: geeft bestandsnaam terug.

Voorbeeld:

```
mijn_bestand = open("resultaten.txt", "w+")
print("Bestandsnaam: ", mijn_bestand.name)
print("Bestand gesloten of niet : ", mijn_bestand.closed)
print("Bestand geopend of niet : ", mijn_bestand.mode)
mijn_bestand.close()
```

geeft als output:

```
Bestandsnaam:  resultaten.txt
Bestand gesloten of niet :  False
Bestand geopend of niet :  w+
```

Close()-methode

De close()-methode slaat alle data op, waarna het bestand gesloten wordt. Het is aanbevolen steeds op het einde het bestand te sluiten.

Inlezen van en schrijven naar bestanden

Nadat een bestand geopend werd, kan het eigenlijke inlezen of schrijven starten. Hiervoor zijn verschillende methodes beschikbaar.

Write()-methode

De write()-methode slaat een string op in een geopende file. Deze string hoeft niet enkel louter tekst te zijn, maar kan evengoed binaire data (bv een afbeelding) opslaan. Opgelet: de write-methode zorgt er niet voor dat er op een nieuwe lijn gesprongen wordt. Dit dien je zelf te verzorgen met het newline-karakter.

Voorbeeld:

```
mijn_bestand.write("Auteur: Stijn Walcarius\n")
mijn_bestand.write("Jaartal: 2016\n" )
jaargang = 12
mijn_bestand.write(str(jaargang))
```

read()-methode

De read()-methode leest tekst in van een geopende file. Het is belangrijk om op te merken dat deze strings ook binaire data kan bevatten. De syntax van de read-methode ziet er als volgt uit:

```
data = fileObject.read([aantal bytes])
```

Bij de read()-methode kan je optioneel het aantal te lezen bytes opgeven. Indien niet opgegeven, probeert hij zoveel mogelijk in te lezen.

Voorbeeld 1: we printen de inhoud van een bestand af.

```
def print_inhoud_bestand_1(bestandsnaam):
    fo = open(bestandsnaam, "r") # opgelet: r+
    data = fo.read()
    print(data)
    fo.close()
```

```
print_inhoud_bestand_1("NMCT.txt")
```

Voorbeeld 2: een bestand bestaat uit verschillende lijnen waarbij op elke lijn een getal staat. We lezen elke lijn in en berekenen het dubbel om dit uiteindelijk ook af te printen.

```
mijn_bestand = open("resultaten.txt", "r")
for lijn in mijn_bestand.readlines():
    getal = int(lijn)
    print(getal * getal)
mijn_bestand.close()
```

Alternatieve oplossing: de open()-methode geeft onmiddellijk een file af waar je met een for-lus kan over lopen. Zo heb je de read()-methode zelfs niet nodig:

```
mijn_bestand = open("getallen.txt", "r")
for lijn in mijn_bestand:
    getal = int(lijn)
```



```
print(getal * getal)
mijn_bestand.close()
```

Voorbeeld 3: lees de lijnen uit een bestand in, en sla deze op in een list. Via de string-functie 'rstrip' verwijderen we het newline-karakter.

```
mijn_list = []
mijn_bestand = open("jaartallen.txt", "r")
for lijn in mijn_bestand:
    mijn_list.append(lijn.rstrip('\n'))
mijn_bestand.close()
print(mijn_list)
```

readline()-methode

Deze methode leest tekens uit het bestand, beginnend bij de cursorpositie, tot aan en inclusief het volgende "newline" teken. Deze tekens worden als een string geretourneerd. Als je aan het einde van het bestand bent en je probeert een nieuwe regel te lezen, krijg je een lege string terug.

Via een while-lus () kan lijn per lijn verwerkt worden.

```
def print_inhoud_bestand_2(bestandsnaam):
    fo = open(bestandsnaam, "r")
    line = fo.readline()
    while (line != ""):
        print(line)
        line = fo.readline()
    fo.close()
```

```
print_inhoud_bestand_2("NMCT.txt")
```

readlines()-methode

Vergelijkbaar met de readline() methode is de readlines() methode. readlines() leest alle regels in een tekstbestand, en retourneert ze als een list van strings. De strings bevatten de newline tekens.

```
def print_inhoud_bestand_3(bestandsnaam):
    fo = open(bestandsnaam, "r") # opgelet: r+
    lines = fo.readlines()
    print(lines)
    fo.close()
```

```
print_inhoud_bestand_3("NMCT.txt")
```

Resultaat: ['Python is a great language.\n', 'Yeah it's great!!\n']

Tot slot, het is belangrijk om op te merken dat deze strings ook binaire data kan bevatten. De syntax van de read-methode ziet er als volgt uit:

```
data = fileObject.read([aantal bytes])
```

Object georiënteerd programmeren

In de voorbije hoofdstukken bespraken we een manier van programmeren die vaak ‘sequentieel’ of ook wel ‘imperatief’ programmeren genoemd wordt. Deze applicaties bestaan uit een sequentie van statements, selectiestructuren, lussen, etc.

Tegenwoordig ondersteunen praktisch alle programmeertalen het concept van ‘object oriëntatie’. Ook Python is een objectgeoriënteerde programmeertaal. Hoewel object oriëntatie een natuurlijke manier aanlevert om problemen en oplossingen te bezien, is het behoorlijk lastig het concept goed onder de knie te krijgen. Je moet eerst het probleem in al zijn aspecten volledig doorgronden voordat je kan programmeren.

Voor kleinere problemen kan dit omslachtig lijken, zeker als de beschikbare tijd beperkt is. Toch zal je voor grote projecten eerst veel tijd spenderen aan het analyseren van jouw oplossing. De object georiënteerde aanpak kan dan net zeer hulpzaam zijn in het creëren van de oplossing. Het is bovendien een vrij universeel concept dat ook in andere programmeertalen aanwezig is.

Object oriëntatie is een vrij breed onderwerp, waardoor er meerdere hoofdstukken aan besteed zullen worden. In deze cursus beschouwen we de basis van object oriëntatie.

“Object”-voorbeeld uit de realiteit

Veel programma’s moeten omgaan met personen. De studentenadministratie werkt met studenten. Deze studenten volgen cursussen, die door docenten gedoceerd worden. Je mag veronderstellen dat de programmeur die de software voor de studentenadministratie geïmplementeerd heeft slim genoeg was om één enkele userinterface te gebruiken om de gegevens van zowel studenten als docenten in te voeren.

Alle personen hebben immers een voornaam en een familienaam. Ze hebben ook een vast verblijfadres, een leeftijd en een geslacht. Om ze uniek te maken, gebruikt de studentenadministratie van iedere persoon het rijksregisternummer. Al deze elementen zijn “**eigenschappen**” (**properties**) van personen die verwijzen naar specifieke bewaarde data (**attributen**).

Leeftijd wordt berekend m.b.v. de geboortedatum en de huidige datum. Je kunt leeftijd voorstellen als een eigenschap, maar het is een eigenschap die elke keer opnieuw berekend moet worden als je hem nodig hebt. Je kunt hem niet als een vaste waarde opslaan, omdat hij morgen anders kan zijn dan vandaag, zonder dat er iets verandert aan de geboortedatum. Daarom wordt “leeftijd” als **methode** toegevoegd.

Studenten “schrijven zich in” voor cursussen. Het lijkt er dus op dat er een methode inschrijven_cursus() moet aanwezig zijn. Enz.

Objecten uit de programmeerwereld

Zoals eerder reeds vermeld is, is ook Python een **object georiënteerde taal**. Object georiënteerd programmeren werkt in de eerste plaats met **objecten**. Een eerste voorbeeld zagen we al in vorig hoofdstuk, nl. een File-object. Elke object heeft zijn eigenschappen en zijn gedrag. Op een object kunnen ook gebeurtenissen plaats vinden waarna een bepaalde actie kan ondernomen worden.

In het vakjargon gebruikt men in het bijzonder over de properties, de methodes en de events van een object. Deze drie belangrijke begrippen worden één voor één in dit en volgende hoofdstukken toegelicht.

Laat ons bovenstaande eerst even vertalen naar een concreet voorbeeld uit de realiteit. Elke auto is een object.

- De kleur, het aantal passagiersplaatsen, de grootte van de kofferruimte, de cilinderinhoud van zijn verbrandingsmotor, ... zijn specifieke kenmerken waarmee ze onderling van elkaar verschillen. Dit zijn de **properties**.

- Een property verwijst naar concrete opgeslagen data. Deze data wordt ook wel de **attributen** van een klasse genoemd.
- Elke auto kan opgestart worden, kan vooruit rijden, kan de koplampen aansteken... Dit zijn de **'methodes'** van een auto.
- Tot slot kan een auto onderweg aangereden worden, plots stilvallen, in een file staan... We spreken over **'events'** (gebeurtenissen) die op een auto kunnen plaats vinden.

Verwar een property niet met een methode. Een property beheert de interne data van een object, terwijl methodes instaan voor het gedrag van een object.

Merk op dat sommige objecten op hun beurt andere objecten kunnen bevatten.

Klassen & objecten

Hoe kunnen we nu een groep van analoge entiteiten (enkel hun eigenschappen zijn verschillend) generiek gaan omschrijven? We introduceren het begrip **'klasse'**.

In de object-georiënteerde wereld behoort iedere onderscheidbare entiteit tot een klasse (Engels: "class"). Een klasse is een generiek model voor een groep entiteiten. De klasse beschrijft alle attributen die de entiteiten gemeen hebben, en beschrijft de methodes die de klasse aanbiedt waarmee de wereld buiten de klasse invloed op de klasse kan uitoefenen.

Op zichzelf is een klasse geen entiteit. Een entiteit die behoort tot een klasse, wordt een **"object"** genoemd. De terminologie is dat een object een "instantie" is van een bepaalde klasse.

Een klasse somt de attributen op. Een object dat een instantie van de klasse is, geeft waarden aan de attributen. En hoewel een klasse de methodes beschrijft die hij ondersteunt, kun je een methode alleen uitvoeren voor een object dat een instantie is van de klasse.

Klassen & datatypes in Python

Hoe vertaalt een klasse zich naar variabelen? Een klasse is een data type, een object is een waarde. Op een object kan een methode via de syntax **<variabele>.<methode>()** aangeroepen worden.

Voorbeeld van een File-object dat via de variabelenaam 'mijn_bestand' aangesproken wordt. Na het aanmaken van een file-object kan je via de methode read() de inhoud van het bestand gaan inlezen. De methode close() zorgt ervoor dat het file-object het bestand correct sluit.

De properties van het object zijn opvraagbaar via de syntax **<variabele>.<property>** waarbij er dus géén haakjes nodig zijn. Op een File-object bespraken we in vorig hoofdstuk de properties name, closed en mode.

Voorbeeld:

```
mijn_bestand = open("resultaten.txt", "w+")
print("Bestandsnaam: ", mijn_bestand.name)
print("Bestand gesloten of niet : ", mijn_bestand.closed)
print("Bestand geopend of niet : ", mijn_bestand.mode)
inhoud = mijn_bestand.read()
mijn_bestand.close()
```

Eerste eigen klasse in Python

Na in vorig hoofdstuk de filosofie van object georiënteerd programmeren bekeken hebben, richten we in dit hoofdstuk tot het zelf schrijven van een eerste klasse. Het start allemaal met het gereserveerde woord “**class**”

class

Een “class” kan je beschouwen als een nieuw data type. Wanneer een class gecreëerd is, kun je instanties van de class toekennen aan variabelen. Om eenvoudig te beginnen, zal ik een class Persoon creëren dat de minimale gegevens van een persoon zal bijhouden. Het is een conventie binnen Python dat elke klassenaam met een hoofdletter start.

Het creëren van de eerste versie van de class Persoon in Python is vrij eenvoudig:

```
class Persoon:
    pass
```

Het gereserveerde woord pass in de class definitie betekent “doe niks.” Dit gereserveerde woord kan je overal gebruiken waar je een commando moet plaatsen, maar waar je nog niks hebt om er neer te zetten. Je mag het niet leeg laten, of alleen een commentaar regel schrijven.

Maar zodra je statements toevoegt, kan je het woord pass verwijderen. Om een object te creëren dat een instantie van de class is, ken ik aan een variabele de naam van de class toe, met haakjes erachter, alsof het de aanroep van een functie is (later zien we hoe je extra informatie via de haakjes kan doorgeven).

```
class Persoon:
    pass
```

```
p1 = Persoon()
print(type(p1))
```

Elke persoon heeft nu een naam, voornaam en geboortjaar. Python ondersteunt ‘soft types’ waarbij je waardes aan attributen kan toekennen. We doen dit voor de eerste maal in de `__init__` methode.

`__init__()`

De initialisatie methode van een class heeft de naam `__init__` (twee “underscores,” gevolgd door het woord init, gevolgd door nog twee “underscores”). Zelfs als je de methode `__init__()` niet expliciet in een klasse opneemt, bestaat hij toch. Je kunt de `__init__()` methode gebruiken om alles te initialiseren waarvan je wilt dat het bestaat bij het instantiëren van de klasse.

Opmerking: achter de schermen wordt een dictionary aangelegd, waarbij de keys en values resp. de attributnamen en de bijhorende waardes zijn. Deze dictionary kan je zelfs opvragen via de methode `__dict__()`

Voorbeeld:

```
class Persoon:

    def __init__(self):
        self.voornaam = ""
        self.naam = ""
        self.geboortjaar = 2016
```

```
p1 = Persoon()
print(type(p1))
print(p1.__dict__)
```

Uitvoering:

```
<class '__main__.Persoon'>
{'voornaam': '', 'naam': '', 'geboortjaar': 2016}
```

Enkele verduidelijkingen bij de methode `__init__()`:

- de init-methode krijgt één parameter 'self'. Dit geldt voor elke methode die je in de klasse opneemt. Het is een verwijzing (referentie) naar het object waar je mee bezig bent. Het is een goede gewoonte om deze parameter steeds 'self' te noemen.
- In de methode krijgen drie attributen een waarde. Ze vormen alle drie een onderdeel van het object zelf, vandaar dat we ze aanspreken via het self-keyword
- Om deze attributen een waarde te geven, gebruiken we de punt-notatie `<object>.<attribuut>`

Opmerking 1: kan je enkel attributen aanmaken in de methode `__init__()`? Het antwoord hierop is neen! Je kan ook in andere methodes extra attributen aanmaken. Zelfs buiten de klasse kunnen nog steeds extra attributen aan de klasse toegevoegd worden:

```
class Persoon:
```

```
    def __init__(self):
        self.voornaam = ""
        self.naam = ""
        self.geboortejaar = 2016
```

```
p1 = Persoon()
print(type(p1))
p1.geslacht = "M"
print(p1.__dict__)
```

Toch is deze werkwijze niet aan te bevelen. Het is een goed gebruik om alle attributen die je aan een object wilt toekennen uitsluitend te creëren in de `__init__()` methode (hoewel je hun waardes elders kunt wijzigen), zodat je weet dat iedere instantie van de class deze attributen heeft, en geen instantie méér dan deze heeft.

Opmerking 2: Net als andere methodes kan de `__init__()` methode parameters krijgen. Je kunt die parameters gebruiken om (sommige van) de attributen een waarde te geven. Bijvoorbeeld, als ik een instantie van Student onmiddellijk waardes wil geven voor de naam, voornaam en geboortejaar kan ik de volgende class definitie gebruiken:

```
class Persoon:
```

```
    def __init__(self, naam, voornaam, geboortejaar=2016):
        self.voornaam = voornaam
        self.naam = naam
        self.geboortejaar = geboortejaar
```

```
p1 = Persoon("Walcarius", "Stijn")
p2 = Persoon("Duchi", "Frederik", 1979)
print("Het geboortejaar van %s %s is %d " % (p1.voornaam, p1.naam, p1.geboortejaar))
print("Het geboortejaar van %s %s is %d " % (p2.voornaam, p2.naam, p2.geboortejaar))
```

`__init__()` heeft nu vier parameters. De eerste is nog steeds self, aangezien dat altijd de eerste parameter moet zijn. De tweede, derde en vierde heten respectievelijk naam, voornaam en geboortejaar. Je mag ze ook anders noemen. Vervolgens wordt naam toegekend aan `self.naam`, enz.

Als je het meegeven van bepaalde argumenten optioneel wilt maken, kan je ook hier aan de parameters default waardes geven.

`__str__()` en `__repr__()`

Wat gebeurt er als je een variabele van jouw klasse gewoon wenst te printen?

```
p1 = Persoon("Walcarius", "Stijn")
print(p1)
p2 = Persoon("Duchi", "Frederik", 1979)
print(p2)
```

Resultaat:

```
<__main__.Persoon object at 0x01C0E350>  
<__main__.Persoon object at 0x01C0E650>
```

Het resultaat is weinig informatief. Python biedt een voorgedefinieerde methode waarmee je dat kan regelen, namelijk de methode `__repr__()`. `__repr__()` moet een string retourneren, en die string wordt getoond als geprobeerd wordt het object te tonen.

Python kent nog een tweede methode om een string versie van een object te creëren, namelijk `__str__()`. `__str__()` is hetzelfde als `__repr__()`, maar wordt alleen gebruikt als het object geprint wordt, of als argument gebruikt wordt in de `format()` methode. Als `__str__()` niet gedefinieerd is, wordt `__repr__()` in plaats ervan gebruikt (maar niet vice versa). Als `__str__()` wel gedefinieerd is, dan kun je ervoor zorgen dat iets anders gebeurt wanneer het object getoond wordt middels de `print()` methode, en wanneer het getoond wordt op andere manieren.

Het is de gewoonte dat de `__repr__()` methode een string retourneert die ieder detail van een object bevat, zodat je (indien nodig) aan de hand van deze string het object volledig opnieuw zou kunnen instantiëren, terwijl `__str__()` een string retourneert die een netjes geformatteerde, goed leesbare versie van de meest belangrijke informatie van een object bevat. In veel gevallen kunnen deze strings hetzelfde zijn.

Veel programmeurs negeren `__repr__()` en definiëren alleen `__str__()`.

Voorbeeld:

```
class Persoon:
```

```
    def __init__(self, naam, voornaam, geboortejahr=2016):  
        self.voornaam = voornaam  
        self.naam = naam  
        self.geboortejahr = geboortejahr  
  
    def __repr__(self):  
        return self.naam + ", " + self.voornaam + " (" + str(self.geboortejahr) + ") "  
  
    def __str__(self):  
        return self.voornaam + " " + self.naam
```

```
p1 = Persoon("Walcarius", "Stijn")  
print(p1)  
print(repr(p1))  
p2 = Persoon("Duchi", "Frederik", 1979)  
print(p2)  
print(repr(p2))
```

Eigen methodes

Je kan ook eigen methodes in een klasse opnemen. Zulke methodes krijgen namen die lijken op de namen van functies, en volgen dezelfde conventies: ze beginnen met een kleine letter, en als er meerdere woorden zijn, hebben ze underscores tussen of hoofdletters voor de eerste letter van ieder tweede en volgende woord.

```
class Persoon:
```

```
    def __init__(self, naam, voornaam, geboortejahr=2016):  
        self.voornaam = voornaam  
        self.naam = naam  
        self.geboortejahr = geboortejahr  
  
    def __repr__(self):  
        return self.naam + ", " + self.voornaam + " (" + str(self.geboortejahr) + ") "  
  
    def __str__(self):  
        return self.voornaam + " " + self.naam
```

```
def verjaardag(self):  
    self.geboortejaar = self.geboortejaar + 1
```

Information hiding

In vorig hoofdstuk werd uitgelegd hoe een klasse binnen Python tot stand komt. In deze situatie bestaat onze klasse uit:

- Methode `__init__()`: aangeroepen na de creatie van een object waarin verschillende attributen in de klasse gedefiniëerd worden
- Methodes `__repr__()` en `__str__()` om informatie over het object zelf op een gerichte manier terug te kunnen geven.
- Methode `__del__()`: vlak voor de destructie van object aangeroepen
- Eigen aanvullende methodes

Tot nu toe zijn alle aangemaakte attributen van buitenuit de klasse vrij toegankelijk en dus ook wijzigbaar.

```
p1 = Persoon("Walcarius", "Stijn")
print(p1)
p1.voornaam = "1252"
print(p1)
```

Enkele kritische bedenkingen hierbij zijn:

- soms wensen we bepaalde attributen we helemaal niet aan de buitenwereld te tonen
- soms wensen we toe te laten dat bepaalde attributen vrij opvraagbaar zijn, maar wensen we het wijzigen ervan nauwer te controleren.
- soms wensen we bepaalde attributen enkel voor afgeleide klassen aanspreekbaar zijn (zie verder)

In de wereld van OO-design worden dan ook verschillende termen (soms door elkaar) gehanteerd. We vermelden onmiddellijk de engelstalige vaktermen gezien deze het meest gebruikelijk zijn.

- **Data encapsulation** slaat op de combinatie van data met specifieke methodes om deze data toegankelijk te maken.
- **Data hiding** slaat op het verbergen van data voor de buitenwereld zodat deze niet per ongeluk kunnen gewijzigd worden.
- **Data abstraction** is de combinatie van data encapsulation en data hiding. De interne data is verborgen voor de buitenwereld én is enkel toegankelijk via de methodes.

In onderstaande paragrafen wordt de versie van Python verder toegelicht.

Data hiding: public vs private attributen

De toegang tot aangemaakte attributen kunnen we drie categoriën opsplitsen:

- public attributen: zijn zichtbaar voor iedereen (dus ook buiten de klasse)
- private attributen: zijn onzichtbaar buiten de klasse, en dus enkel zichtbaar (en opvraagbaar) binnen de klasse.
- Protected attributen zijn enkel zichtbaar in de klasse alsook in een afgeleide klasse (zie later). Daarbuiten zijn deze onzichtbaar.

In Python wordt de zichtbaarheid via het aantal underscores duidelijk gemaakt:

syntax	Type	Betekenis
naam	public	Deze attributen zijn zowel binnenin als buiten de klasse vrij toegankelijk
<code>_naam</code>	protected	Deze attributen zijn enkel binnenin de klasse én in afgeleide klassen toegankelijk.
<code>__naam</code>	private	Deze attributen zijn enkel binnenin de klasse zichtbaar. Van buiten uit zijn ze onzichtbaar en dus ook niet toegankelijk.

Voorbeeld: in de klasse Persoon werden de drie attributen private geplaatst.

```
class Persoon:

    def __init__(self, naam, voornaam, geboortejaar=2016):
        self.__voornaam = voornaam
        self.__naam = naam
        self.__geboortejaar = geboortejaar

    def __repr__(self):
        return self.__naam + ", " + self.__voornaam + " (" + str(self.__geboortejaar) + ") "

    def __str__(self):
        return self.__voornaam + " " + self.__naam

    def verjaardag(self):
        self.__geboortejaar = self.__geboortejaar + 1
```

Gevolg: onderstaande lijn geeft nu een uitvoeringsfout:

```
p1 = Persoon("Walcarius", "Stijn")
print(p1.naam)
```

We kunnen buiten de klasse niet meer aan onze attributen!

Data encapsulation

Via specifieke methodes wensen we deze data opnieuw voor de buitenwereld toegankelijk maken. Python verschilt hierin sterk van andere programmeertalen. We presenteren eerst de methode uit andere programmeertalen (vertaald in python-syntax). Daarna wordt de aanbevolen werkwijze uit Python toegelicht.

Get- en set- methodes

Java, C# en talrijke andere programmeertalen voorzien een methodiek van get- en set-methodes. Afhankelijk van de programmeertaal ziet de syntax er iets anders uit. Toch is het achterliggend idee identiek. Hierbij geldt dat:

- Een get-methode de waarde van een attribuut terug geeft
- Een set-methode de waarde van een attribuut wijzigt met een nieuwe doorgegeven waarde. Eventueel kunnen hier extra controles ingebouwd worden om zo na te gaan of de nieuwe doorgegeven waarde weldegelijk voldoet aan onze voorwaarden.

In Python kan deze methodiek als volgt geïmplementeerd worden. Dit is niet de aanbevolen werkwijze (zie verder).

Voorbeeld:

```
class Persoon:

    def __init__(self, naam, voornaam, geboortejaar=2016):
        self.__voornaam = voornaam
        self.__naam = naam
        self.__geboortejaar = geboortejaar

    def __repr__(self):
        return self.__naam + ", " + self.__voornaam + " (" + str(self.__geboortejaar) + ") "

    def __str__(self):
        return self.__voornaam + " " + self.__naam

    def verjaardag(self):
        self.__geboortejaar = self.__geboortejaar + 1
```

```
def set_naam(self, nieuwe_naam):
    self.__naam = nieuwe_naam

def get_naam(self):
    return self.__naam

def set_voornaam(self, nieuwe_voornaam):
    self.__voornaam = nieuwe_voornaam

def get_voornaam(self):
    return self.__voornaam

def set_geboortejaar(self, nieuw_geboortejaar):
    self.__geboortejaar = nieuw_geboortejaar

def get_geboortejaar(self):
    return self.__geboortejaar
```

De familienaam afprinten kan nu als volgt:

```
p1 = Persoon("Walcarius", "Stijn")
# print(p1.naam)
print(p1.get_naam())
```

Indien we het geboortejaar wensen te wijzigen, gebruiken we onderstaande code:

```
p1 = Persoon("Walcarius", "Stijn")
print(p1.get_naam())
p1.set_geboortejaar(2000)
```

Python's versie van get- en set-methodes

Enkele kritische bemerkingen bij voorgaand voorbeeld: hoewel datahiding en data encapsulation duidelijk geïntegreerd werden, is deze werkwijze atypisch voor Python. Python streeft immers naar een duidelijke, heldere, eenvoudig leesbare codestijl. Bovenstaande code zou in Python eerder als volgt moeten gebruikt worden:

```
p1 = Persoon("Walcarius", "Stijn")
print(p1.naam)
p1.geboortejaar = 2000
```

Om deze werkwijze wél toe te laten, kunnen we misschien de attributen opnieuw public maken? Deze filosofie staat haaks op de werkwijze uit andere programmeertalen. Hierbij wordt standaard elk attribuut private geplaatst, en worden onmiddellijk de nodige get- en set-methodes gegenereerd.

Het opnieuw public plaatsen van de verschillende attributen gebeurt in de methode `__init__()`:

```
class Persoon:
    def __init__(self, naam, voornaam, geboortejaar=2016):
        self.voornaam = voornaam
        self.naam = naam
        self.geboortejaar = geboortejaar
```

Maar... hierdoor gaat de datahiding verloren!? Inderdaad, datahiding verdwijnt. Maar... data-encapsulation verdwijnt dan ook? Klopt, zonder voorzorgen is elk attribuut opnieuw vrij benaderbaar en is dus elke waarde opnieuw mogelijk... Dit is niet wenselijk!

```
p1 = Persoon("Walcarius", "Stijn")
print(p1.naam)
p1.geboortejaar = -201
```

Toch heeft Python op bovenstaande opmerkingen een antwoord: **properties**! De oplossing voor het attribuut naam ziet er nu als volgt uit:

```
@property
def naam(self):
    return self.__naam          #hier wordt de waarde van een private attribuut teruggegeven

@naam.setter
def naam(self, nieuwe_naam):
    if (nieuwe_naam != ""):
        self.__naam = nieuwe_naam          #hier wordt een private attribuut aangemaakt
    else:
        self.__naam = "onbekend"          #hier wordt een private attribuut aangemaakt
```

Een property bestaat uit twee delen:

- Een methode die gebruikt wordt om een waarde terug te geven. Deze methode wordt vooraf gegaan door een annotatie “@property”. Deze komt vóór het hoofd van de methode. We spreken over de *property-methode* van een attribuut. Andere gangbare term is de ‘getter-property’ of kortweg ‘getter’.
- Een methode die verantwoordelijk is om een nieuwe waarde in te stellen. Deze wordt vooraf gegaan door de annotatie “@<name>.setter” waarbij name een link is naar de eerste methode. In deze methode kan je controles inbouwen. Uiteindelijk wordt in deze methode het private attribuut aangemaakt. We spreken over de *setter-methode* van een attribuut.

Deze annotaties geven als voordeel dat we beide methodes op een kortere manier kunnen aanroepen:

```
p1 = Persoon("Walcarius", "Stijn")
print(p1.naam)
p1.naam = "Vanacker"
print(p1.naam)
print(p1.__dict__)
```

Uitvoering hiervan levert volgend resultaat:

```
Walcarius
Vanacker
{'_Persoon__voornaam': 'Stijn', '_Persoon__geboortejaar': 2016, '_Persoon__naam': 'Vanacker'}
```

Wanneer we de init-methode terug nemen, zien we dat ook daar van de setter gebruik wordt gemaakt.

```
class Persoon:
    def __init__(self, naam, voornaam, geboortejaar=2016):
        self.voornaam = voornaam
        self.naam = naam
        self.geboortejaar = geboortejaar

    @property
    def naam(self):
        return self.__naam

    @naam.setter
    def naam(self, nieuwe_naam):
        if (nieuwe_naam != ""):
            self.__naam = nieuwe_naam
        else:
            self.__naam = "onbekend"
```

De klasse Persoon ziet er finaal als volgt uit:

```
class Persoon:
    def __init__(self, naam, voornaam, geboortejaar=2016):
        self.voornaam = voornaam
        self.naam = naam
        self.geboortejaar = geboortejaar

    @property
    def naam(self):
        return self.__naam

    @naam.setter
    def naam(self, nieuwe_naam):
        if (nieuwe_naam != ""):
            self.__naam = nieuwe_naam
```

```

else:
    self.__naam = "onbekend"

@property
def voornaam(self):
    return self.__voornaam

@voornaam.setter
def voornaam(self, nieuwe_voornaam):
    if (nieuwe_voornaam != ""):
        self.__voornaam = nieuwe_voornaam
    else:
        self.__voornaam = "onbekend"

@property
def geboortejaar(self):
    return self.__geboortejaar

@geboortejaar.setter
def geboortejaar(self, nieuw_geboortejaar):
    if (nieuw_geboortejaar < 1900):
        self.__geboortejaar = 2016
    else:
        self.__geboortejaar = nieuw_geboortejaar

def repr (self):
    return self.naam + ", " + self.voornaam + " (" + str(self.geboortejaar) + ") "

def __str__(self):
    return self.voornaam + " " + self.naam

def verjaardag(self):
    self.geboortejaar = self.geboortejaar + 1

```

Hoeven we steeds voor elk attribuut op deze werkwijze werken? Helemaal niet. Bovenstaande python-werkwijze is enkel nuttig als we controles op nieuwe waarden wensen in te bouwen, of als we de output van een attribuut wensen te manipuleren. In dat laatste geval werken we de property-methode verder uit. Bijvoorbeeld:

```

@property
def leeftijd(self):
    currentYear = datetime.now().year
    return "%d jaar" % (currentYear - self.geboortejaar)

```

Conclusie

Het gebruik van public versus private attributen kunnen we nu als volgt samenvatten. We vertrekken van een situatie waarbij we een nieuwe klasse wensen te ontwikkelen met daarin een attribuut “mijn_vooropleiding”

- Is het attribuut buiten de klasse nodig?
- Is het antwoord hierop neen: maak het attribuut private. Maak géén property-methode en setter-methode
- Is het antwoord hierop ja: maak het toegankelijk als een publiek attribuut.
- Wens je controles in te bouwen zodat niet elke waarde aan het attribuut kan gegeven worden, of wens je de waarde op een andere wijze teruggeven: gebruik dan een private attribuut gecombineerd met een property-methode en setter-methode.

Opmerking: In het vervolg van deze cursus alsook in de verschillende labolessen zal voor de eenvoud voor elk attribuut gekozen worden om te werken met een resp. property-methode en setter-methode.

Klasse attributen & methodes

Klasse- versus instantie-attributen

Tot nu toe beschrijven de attributen van een klasse hoe elk object individueel kan verschillen. De klasse Persoon had de attributen naam, voornaam en geboortjaar. Twee objecten zullen dan ook verschillen van elkaar doordat ze voor deze attributen andere waarden hebben. We spreken ook over instantie-attributen ('Instance Attributes').

```
p1 = Persoon("Walcarius", "Stijn")
p2 = Persoon("Di Marco", "Mileto")
```

Toch bestaat er ook een manier om data over alle objecten heen te delen. Dit concept is meer gekend als "class attributes" of "klasse-variabelen/klasse-attributen". Wanneer een object de waarde ervan wijzigt, wijzigt dit mee over alle objecten heen. Deze klasse attributen worden immers door iedereen gedeeld.

Klasse-attributen zijn binnen de python-syntax eenvoudig herkenbaar:

- Zij worden niet in de init-methode of andere methode aangemaakt
- Hun definitie staat buiten de methodes.
- Meestal worden deze bovenaan de klasse vermeld

Start-voorbeeld:

```
class Persoon:
    vereniging = "Howest"

    def __init__(self, naam, voornaam, geboortjaar=2016):
        self.voornaam = voornaam
        self.naam = naam
        self.geboortjaar = geboortjaar

    @property
    def naam(self):
```

Gebruik van deze klasse-attributen:

```
p1 = Persoon("Walcarius", "Stijn")
p2 = Persoon("Di Marco", "Mileto")
print("Onze vereniging: %s " % Persoon.vereniging)
print("Onze vereniging: %s " % p1.vereniging)
Persoon.vereniging = "NMCT" #wijzigen class-attribute
print("Onze vereniging: %s " % p2.vereniging)
```

Geeft volgende uitvoering:

```
Onze vereniging: Howest
Onze vereniging: Howest
Onze vereniging: NMCT
```

Een klasse-attribuut is van buitenuit opvraagbaar via:

- Ofwel de klassenaam: <class-name>.<class-attribute>
- Ofwel elk object van de klasse zelf: <name_object>.<class-attribute>

De waarde van een klasse-attribuut wijzigen kan enkel via de naam van de klasse:
<class-name>.<class-attribute> = <new-value>

Opgelet: onderstaande code werkt niet op de klasse variabele! In de tweede lijn wordt een extra instantie-attribuut aangemaakt.

```
print("p2 wijzigt vereniging?")
p2.vereniging = "IPO" #!p2 maakt nieuw instance-attribute aan
print("Onze vereniging: %s " % Persoon.vereniging)
```

```
print("Onze vereniging: %s " % p1.vereniging)
print("Onze vereniging: %s " % p2.vereniging)
```

Uitvoering levert:

```
p2 wijzigt vereniging?
Onze vereniging: NMCT
Onze vereniging: NMCT
Onze vereniging: IPO
```

Voorbeeld klasse attribuut

In onderstaand voorbeeld wordt via een klasse-attribuut bijgehouden hoeveel objecten van een klasse aangemaakt werden. De werkwijze hiervoor bestaat als volgt:

- Maak een klasse-attribuut `aantal_personen` aan. Zet deze bij aanvang op 0.
- In de `__init__()`-methode wordt het klasse-attribuut verhoogd met 1
- In de `__del__()`-methode wordt het klasse-attribuut verlaagd met 1

Om vanuit een interne methode een klasse-attribuut aan te spreken, gebruiken we de syntax:

```
type(self).<class-attribue>
```

De relevante code uit de klasse hiervoor is:

```
class Persoon:

    vereniging = "Howest"
    aantal_personen = 0

    def __init__(self, naam, voornaam, geboortejaar=2016):
        self.voornaam = voornaam
        self.naam = naam
        self.geboortejaar = geboortejaar
        type(self).aantal_personen +=1

    def __del__(self):
        type(self).aantal_personen -=1
```

Uitvoering van onderstaand code:

```
p1 = Persoon("Walcarius", "Stijn")
p2 = Persoon("Di Marco", "Mileto")
print("Aantal personen: %d" % Persoon.aantal_personen)
del (p2)
print("Aantal personen: %d" % Persoon.aantal_personen)
```

levert volgend resultaat:

```
Aantal personen: 2
Aantal personen: 1
```

Static methodes

In voorgaande voorbeelden waren de klasse-attributen publiek. Uiteraard kunnen deze ook private gedeclareerd worden. De werkwijze is analoog als bij gewone instantie-attributen, nl. een dubbele underscore voor de variabelenaam.

Voorbeeld:

```
class Persoon:

    __aantal_personen = 0
```

Toch moeten we nu ook een manier hebben om deze private klasse attributen van buitenuit op te vragen en te wijzigen zonder eerst een object van de klasse te hoeven aanmaken.

De oplossing hiervoor bestaat uit het creëren van zgn. ‘static-methodes’. Een static-methode kan opgeroepen worden zonder eerst een object van de klasse aan te maken. Static-methodes worden in Python via de annotatie ‘@staticmethod’ aangeduid.

```
class Persoon:

    __aantal_personen = 0

    @staticmethod
    def AantalPersonen():
        return Persoon.__aantal_personen

    def __init__(self, naam, voornaam, geboortejaar=2016):
        self.voornaam = voornaam
        self.naam = naam
        self.geboortejaar = geboortejaar
        type(self).__aantal_personen += 1

    def __del__(self):
        type(self).__aantal_personen -= 1
```

Uitvoering van onderstaande code:

```
p1 = Persoon("Walcarius", "Stijn")
p2 = Persoon("Di Marco", "Mileto")
print("Aantal personen: %d" % Persoon.AantalPersonen())
del (p2)
print("Aantal personen: %d" % Persoon.AantalPersonen())
```

levert volgend resultaat op:

```
Aantal personen: 2
Aantal personen: 1
```

Klasse methodes

Klasse methodes mogen niet verward worden met static methodes. Net zoals static methodes zijn klassemethodes niet gebonden aan een object. Klasse-methodes hebben als eerste parameter een referentie naar de klasse waarin ze zitten. Een klassemethode wordt vooraf gegaan door de annotatie ‘@classmethod’. Het oproepen gebeurt op identieke wijze als een static-methode.

```
class Persoon:

    __aantal_personen = 0

    @classmethod
    def AantalPersonen(cls):
        return Persoon.__aantal_personen
```

Waar worden klassemethodes gebruikt?

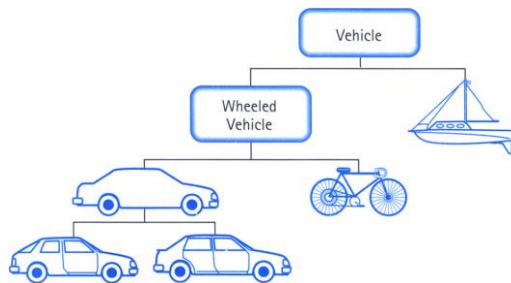
- factory-methods: methodes die objecten van een klasse kunnen genereren (valt buiten bereik van de cursus)
- bij overerving kan in een klasse-methode nagegaan worden welke subklasse precies actief is. (overerving komt in volgend hoofdstuk aan bod).

Overerving

Overerving (“inheritance”) is een mechanisme om een nieuwe klasse te baseren op een bestaande klasse, door enkel en alleen de verschillen tussen de twee aan te geven. Dit is een bijzonder krachtig concept dat het mogelijk maakt hoogst flexibele en gemakkelijk onderhoudbare programma’s te schrijven.

Net zoals in andere object-georiënteerde programmeertalen ondersteunt Python ook overerving. Hierbij erft een klasse over van een andere klasse: een klasse (‘sub-klasse/child-class’) erft dan niet alleen de attributen, maar ook de methodes van de andere klasse (‘basis-klasse/super-class’).

Op deze manier ontstaat er een hiërarchie tussen de klassen. Dit is sterk vergelijkbaar met voorbeelden uit de reële wereld: personenwagens, fietsen, boten,... zijn allemaal voertuigen. Toch kunnen we fietsen en voertuigen onderbrengen onder de noemer ‘voertuigen met banden’. Onder de personenwagens kunnen nog verschillende categorieën plaatsen. Enz.



Basis syntax overerving

Overerving is erg eenvoudig. Bij de definitie van een nieuwe klasse, zet je tussen haakjes de naam van een andere klasse. De nieuwe klasse erft dan alle attributen en methodes van de andere klasse, wat wil zeggen dat ze automatisch zijn opgenomen in de nieuwe klasse. De syntax ziet er als volgt uit:

```
class <naam subklasse>(<naam basisklasse>):
```

Als illustratie werken we verder op het voorbeeld van vorig hoofdstuk. We vertrekken van de klasse Persoon met de instantie-attributen naam, voornaam en geboortjaar. De klasse Persoon is dan ook de basisklasse. De klasse Student laten we erven van de klasse Persoon. Deze klasse wordt zo een subklasse van de klasse Persoon. Hierdoor erft de klasse Student alle attributen en methodes van de klasse Persoon. Overerving wordt dan ook vaak verwoord als “... is een...” relatie: Een Student-object is een Persoon-object.

```
class Student(Persoon):
    pass
```

```
s1 = Student("Pinket", "Lies", 1980)
s2 = Student("Daels", "Jef", 1976)
print(s1)
print(s2)
```

Uitvoering:

```
Lies Pinket
Jef Daels
```


Subklasse uitbreiden

Een subklasse kan steeds uitgebreid worden met extra attributen en extra methodes.

Voorbeeld: bij het aanmaken van een nieuwe Student wensen we naast de naam, voornaam en geboortjaar ook een list van namen van ingeschreven modules. We voegen alvast de `__init__()`-methode toe:

```
class Student(Persoon):
    def __init__(self, naam, voornaam, geboortjaar, modules):
        self.ingeschreven_modules = modules
        # naam, voornaam & geboortjaar???
```

Evenwel, bij het aanmaken van een nieuwe student merken we op dat:

- De attributen naam, voornaam en geboortjaar in de klasse Persoon aanwezig zijn. Het instellen gebeurt in de methode `__init__()` van de klasse Persoon.
- List is nieuw in de klasse Student.

Let nu op: indien de methode reeds in de basisklasse bestaat, dan overschrijft de methode van de subklasse de versie uit de basisklasse! Dit staat beter bekend als ‘**method overriding**’.

In bovenstaand voorbeeld dienen we vanuit de `__init__`-methode() van de klasse Student ook deze van de klasse Persoon kunnen aanspreken. Er zijn twee werkwijzes om een methode uit de basisklasse aan te roepen:

- met een zgn ‘class call’. De syntax ziet er als volgt uit:
`<class-name>.<methode> (...)`
- via de `super()`-methode. De syntax ziet er als volgt uit:
`super().<methode> (...)`

De eerste methode toegepast op bovenstaand voorbeeld: om de `init`-methode van de klasse Persoon aan te roepen, schrijven we `Persoon.__init__(...)`. Opgelet, bij de aanroep moet ook keyword `self` als eerste parameter vermeld worden! `Self` kan hier perfect gebruikt worden aangezien elk Student-object ook een Persoon-object is.

```
class Student(Persoon):
    def __init__(self, naam, voornaam, geboortjaar, modules):
        self.ingeschreven_modules = modules
        Persoon.__init__(self, naam, voornaam, geboortjaar)
```

De tweede methode werkt met het keyword `super`: via dit keyword verwijst men onmiddellijk naar de basisklasse (superklasse). In dit geval hoeven we het keyword `self` niet als eerste parameter te gebruiken. Vorig voorbeeld wordt dan:

```
class Student(Persoon):
    def __init__(self, naam, voornaam, geboortjaar, modules):
        super().__init__(naam, voornaam, geboortjaar)
        self.ingeschreven_modules = modules
```

De tweede werkwijze wordt ook in andere programmeertalen gebruikt, en geniet dan ook onze voorkeur.

Beide werkwijzes kunnen ook in andere methodes gehanteerd worden: zo kan men op elk moment steeds refereren naar attributen of methodes uit de basisklasse.

```
class Student(Persoon):
    def init (self, naam, voornaam, geboortjaar, modules):
        super().__init__(naam, voornaam, geboortjaar)
        self.ingeschreven_modules = modules

    def __repr__(self):
        return super().__repr__() + " is ingeschreven in " +
            str(len(self.ingeschreven_modules)) + " modules"

    def __str__(self):
        return "Student " + super().__str__()

    def inschrijven module(self, nieuwe module):
        self.ingeschreven_modules.append(nieuwe_module)
```

Tenslotte, hoe gaat Python nu te werk wanneer een methode op een object van een subklasse aangeroepen wordt?

- hij gaat eerst op zoek in de subklasse zelf: vindt hij hier de methode, dan wordt deze uitgevoerd.
- Wanneer de methode niet in de subklasse terug gevonden wordt, wordt in de basisklasse verder gezocht. Indien hier aanwezig, dan wordt deze uitgevoerd. Indien ook hier afwezig, dan wordt een uitvoeringsfout gegenereerd.

Meervoudige overerving

In voorgaande paragrafen werd geïllustreerd hoe een klasse kan erven van een andere klasse. Binnen de programmeertaal Python kan je een klasse ook laten erven van meerdere andere klassen. We spreken hier over meervoudige overerving. Andere programmeertalen laten dit niet toe (Java, C#), de programmeertaal C++ dan opnieuw wel.

De syntax hiervoor ziet er als volgt uit:

```
class SubclassName(BaseClass1, BaseClass2, BaseClass3, ...):  
    pass
```

Opmerking: het is perfect mogelijk dat bv. BaseClass1 ook erft van meerdere andere klassen. Enz. We krijgen een ‘overervingsboom’.

Als een methode van een instantie van de nieuwe class wordt aangeroepen, moet Python beslissen welke methode gebruikt wordt als deze voorkomt in meerdere van de samenstellende classes. Python controleert daarvoor eerst de subclass zelf. Als de methode daar niet gevonden wordt, worden alle superclasses gecontroleerd, van links naar rechts. Zodra een specificatie van de methode gevonden wordt, wordt die uitgevoerd.

Wenst men vanuit de subklasse een methode uit één van de basisklassen aan te roepen, dan maak je het best gebruik van een ‘class-call’. Zo geef je precies aan welke basisklasse je wenst te gebruiken.

Aangezien deze programmeertechniek niet in elke programmeertaal aanwezig is, zullen we dan ook in deze cursus er niet verder over uitweiden. Ook de verschillende labo’s zullen hier niet verder op inspelen.

Operator overloading

Is het mogelijk om klassieke bewerkingsoperatoren (zoals + en *) in jouw eigen klasse op te nemen? Ja! In sommige gevallen kan dit zelfs een logische aanvulling zijn. We lichten dit toe met volgend voorbeeld: je maakt een klasse 'Winkelkar' waarin via een list producten worden bijgehouden.

Wanneer twee winkelkarretjes bij elkaar opgeteld worden, veronderstellen we dat de som zowel de producten uit het eerste als het tweede winkelkarretje bevatten.

```
w1 = Winkelkar()
w1.voeg_product_toe("cd1")
w1.voeg_product_toe("cd2")
w2 = Winkelkar()
w2.voeg_product_toe("boek1")
w2.voeg_product_toe("boek3")
print("Winkelkar 1: %s" % w1)
print("Winkelkar 2: %s" % w2)
w3 = w1 + w2
print("Winkelkar 3: %s" % w3)
```

Indien we niet aan onze klasse duidelijk maken wat de +-operator precies inhoudt, krijgen we op deze regel een uitvoeringsfout terug.

We lossen dit op door de methode `__add__(self, other)` aan de klasse Winkelkar toe te voegen. Hierbij geldt dat:

- De eerste parameter het object aan de linkerkzijde van de operator + voorstelt (w1)
- De tweede parameter (other) het object aan de rechterzijde van de operator + voorstelt (w2)
- het is de linkerparameter die de methode uitvoert

We kiezen ervoor om een nieuwe winkelkar aan te maken waarbij zijn productenlist bestaat uit de som van de list uit w1 (self) en de list uit w2 (other).

Voorbeeld:

```
class Winkelkar:
    def __init__(self):
        self.producten = []

    def __str__(self):
        s = "producten zijn "
        for product in self.producten:
            s = s + product + " "
        return s

    def voeg_product_toe(self, nieuw_product):
        self.producten.append(nieuw_product)

    def __add__(self, other):
        w = Winkelkar()
        w.producten = self.producten + other.producten
        return w

    def __iadd__(self, other):
        self.producten += other.producten
        return self
```

```
w1 = Winkelkar()
w1.voeg_product_toe("cd1")
w1.voeg_product_toe("cd2")
w2 = Winkelkar()
w2.voeg_product_toe("boek1")
w2.voeg_product_toe("boek3")
print("Winkelkar 1: %s" % w1)
print("Winkelkar 2: %s" % w2)
w3 = w1 + w2
print("Winkelkar 3: %s" % w3)
w1 += w2
print("Winkelkar 1: %s" % w1)
```

Onderstaande lijst geeft een overzicht van de operatoren met hun overeenkomstige methodes.

+	object.__add__(self, other)
-	object.__sub__(self, other)
*	object.__mul__(self, other)
//	object.__floordiv__(self, other)
/	object.__div__(self, other)
%	object.__mod__(self, other)
**	object.__pow__(self, other[, modulo])
<<	object.__lshift__(self, other)
>>	object.__rshift__(self, other)
&	object.__and__(self, other)
^	object.__xor__(self, other)
	object.__or__(self, other)
+=	object.__iadd__(self, other)
-=	object.__isub__(self, other)
*=	object.__imul__(self, other)
/=	object.__idiv__(self, other)
//=	object.__ifloordiv__(self, other)
%=	object.__imod__(self, other)
**=	object.__ipow__(self, other[, modulo])
<<=	object.__ilshift__(self, other)
>>=	object.__irshift__(self, other)
&=	object.__iand__(self, other)
^=	object.__ixor__(self, other)
=	object.__ior__(self, other)
<	object.__lt__(self, other)
<=	object.__le__(self, other)
==	object.__eq__(self, other)
!=	object.__ne__(self, other)
>=	object.__ge__(self, other)
>	object.__gt__(self, other)

Exception handling

Soms treden runtime errors op niet omdat je een programmeerfout hebt gemaakt, maar omdat er een probleem optreedt dat je niet kon voorzien toen je het programma schreef. Dit is buitengewoon relevant als je met bestanden werkt: bijvoorbeeld, als je een bestand verwerkt dat op een USB-stick staat, en de gebruiker verwijdert de USB-stick tijdens de verwerking, krijg je uiteraard een fout die je niet echt zou kunnen voorzien in je code.

Iedere runtime error genereert in de code een zogenaamde “exception” (“uitzondering”) die je kunt “afvangen.” Het afvangen van een exception betekent dat je in je programma code opneemt die ervoor zorgt dat de opgetreden fout zoveel mogelijk netjes wordt afgehandeld, in plaats van je programma abrupt af te breken.

Errors en exceptions

Als je een Python programma start, controleert Python eerst of alle statements in je programma voldoen aan de syntax-eisen die Python stelt. Als dat niet het geval is, genereert Python een zogeheten “syntax error” en wordt het programma niet uitgevoerd. Als Python geen syntax errors tegenkomt, wordt het programma uitgevoerd, maar er kunnen dan nog steeds statements gevonden worden die fouten genereren als ze worden uitgevoerd. Zulke statements veroorzaken een “runtime error.”

Over het algemeen zul je proberen runtime errors op te lossen door je code uit te breiden of te wijzigen. Bijvoorbeeld, het volgende programma geeft een runtime error als je geen getal ingeeft:

```
def demo1():
    list_getallen = []
    print("Geef een reeks van 10 getallen op.")
    while (len(list_getallen) < 10):
        getal = int(input("Geef nu een getal op: "))
        list_getallen.append(getal)
    print("De opgegeven getallen zijn: %s" % str(list_getallen))
demo1()
```

Om die op te lossen, kun je het programma wijzigen en vooraf controleren of de invoer wel degelijk getal is. [Thuisopgave: zoek even op!]

Het gebeurt echter wel eens dat je zult moeten accepteren dat exceptions kunnen optreden omdat je gewoonweg niet alle omstandigheden waarin je programma wordt uitgevoerd kunt voorzien. Dat is specifiek het geval als je programma afhangt van zaken die je niet onder controle kunt hebben, zoals bij het werken met bestanden en gebruikershandelingen...

Afhandelen van exceptions

Om exception expliciet in je programma af te handelen, gebruik je de try ... except constructie. Er zijn verschillende manieren om deze constructie toe te passen.

De basis vorm van de try ... except constructie heeft de volgende syntax:

```
try:
    <acties>
except:
    <exception afhandeling>
```

Als de <acties> tussen try: en except: worden uitgevoerd en er een exception wordt gegenereerd, springt Python onmiddellijk naar de <exception afhandeling> en voert die uit, waarna het programma vervolgt met de regels code onder de <exception afhandeling>. Als er geen exceptions optreden gedurende de uitvoering van <acties>,

wordt de <exception afhandeling> overgeslagen.

Gebruik makend van exception afhandeling, kan de code aan het begin van dit hoofdstuk als volgt geschreven worden om runtime errors te vermijden:

```
def demolbis():
    list_getallen = []
    print("Geef een reeks van 10 getallen op.")
    while (len(list_getallen) < 10):
        try:
            getal = int(input("Geef nu een getal op: "))
            list_getallen.append(getal)
        except:
            print("Geen geldige waarde. Probeer opnieuw.")
    print("De opgegeven getallen zijn: %s" % str(list_getallen))
```

Bij het except-blok kan je ook specifiek vermelden welke soort exception je opvangt:

```
def demolbis():
    list_getallen = []
    print("Geef een reeks van 10 getallen op.")
    while (len(list_getallen) < 10):
        try:
            getal = int(input("Geef nu een getal op: "))
            list_getallen.append(getal)
        except ValueError:
            print("Geen geldige waarde. Probeer opnieuw.")
    print("De opgegeven getallen zijn: %s" % str(list_getallen))
```

Je kunt extra informatie krijgen over een exception door een as toe te voegen aan een except, middels de syntax except <exception> as <variabele>. De variabele bevat dan een exception "object," dat meer informatie bevat over de exception. Helaas is er geen standaard manier waarop je de informatie eruit kunt. Tenslotte kan je de fout zelf ook benaderen door deze met een variabelenaam te benoemen:

```
def demolbis():
    list_getallen = []
    print("Geef een reeks van 10 getallen op.")
    while (len(list_getallen) < 10):
        try:
            getal = int(input("Geef nu een getal op: "))
            list_getallen.append(getal)
        except ValueError as ex:
            print("Geen geldige waarde. Probeer opnieuw.")
            print("Detail foutmelding: %s" % str(ex))
    print("De opgegeven getallen zijn: %s" % str(list_getallen))
```

Afhandelen van specifieke exceptions

Bekijk de code hieronder. Er kunnen minstens twee exceptions optreden als deze code wordt uitgevoerd. Welke?

```
print( 3 / int( input( "Geef een getal op: " ) ) )
```

De twee exceptions die in deze code kunnen optreden zijn de ZeroDivisionError als je een nul ingeeft, en de ValueError als je iets ingeeft dat geen integer is. Probeer even!

Je kunt beide exceptions afhandelen met een enkele try ... except constructie, maar je kunt ze ook van elkaar onderscheiden door meerdere excepts te gebruiken. Iedere except kan gevolgd worden door één van de specifieke exceptions, en de code die bij die except hoort wordt alleen uitgevoerd als die specifieke exception wordt gegenereerd.

```
def meerdere_except_blokken():  
    try:  
        print(3 / int(input("Geef een getal: ")))  
    except ZeroDivisionError:  
        print("Je kunt niet delen door nul")  
    except ValueError:  
        print("Je gaf geen getal op")
```

Als je “alle overige exceptions” wilt afhandelen, kun je een except zonder specifieke exception aan het einde toevoegen. Slechts één van de excepts zal worden uitgevoerd, namelijk de eerste die wordt aangetroffen die van toepassing is. Dit werkt dus ongeveer als een if ... elif ... elif ... else constructie.

```
def meerdere_except_blokken():  
    try:  
        print(8 / int(input("Geef een getal: ")))  
    except ZeroDivisionError:  
        print("Je kunt niet delen door nul")  
    except ValueError:  
        print("Je gaf geen getal op")  
    except:  
        print("Iets onverwachts ging fout")
```

Hier zijn een aantal specifieke exceptions die vaak optreden:

- ZeroDivisionError: Delen door nul
- IndexError: Het benaderen van een list of tuple met een index die niet binnen het legale bereik valt
- KeyError: Het benaderen van een dictionary met een key die onbekend is
- IOError: Iedere fout die kan optreden als je een bestand benadert (deze exception is een alias voor OSError)
- FileNotFoundError: Het proberen te openen van een niet-bestaand bestand om eruit te lezen
- ValueError: Het optreden van een fout bij het “casten” van een waarde naar een andere waarde
- TypeError: Het gebruiken bij een operatie van een waarde met een data type dat niet ondersteund wordt door de operatie

Toevoegen van een finally

Je kunt nog een extra tak toevoegen aan een try constructie, namelijk finally. Bij finally kun je een serie statements opnemen die worden uitgevoerd ongeacht de wijze waarop de try constructie wordt verlaten. Als alles normaal verloopt, worden de statements bij de finally uitgevoerd, maar ook als je een runtime error krijgt, worden ze uitgevoerd. Je kunt bijvoorbeeld finally gebruiken om er zeker van te zijn dat een bestand dat je geopend hebt, wordt gesloten.

```
try:
    fp = open( "nmct.txt" )
    print( "Bestand geopend" )
    print( fp.read() )
finally:
    fp.close()
    print( "Bestand gesloten" )
```

Genereren van exceptions

Je mag zelf in je code ook exceptions genereren. Daarvoor is het gereserveerde woord **raise** bestemd. Je laat het volgen door één van de bekende exceptions (je mag eventueel ook je eigen exceptions definiëren). Je mag een exception die je genereert willekeurige argumenten meegeven, en die zijn dan weer via het args attribuut te benaderen.

```
def demo3():
    try:
        demo4()
    except ZeroDivisionError:
        print("Feedback vanuit demo 3: Delen door 0 kan natuurlijk niet...")

def demo4():
    try:
        getal = 4
        print(getal / 0)
    except ZeroDivisionError as zde:
        print("Feedback vanuit demo 4: Delen door 0 kan niet")
        raise zde
```

Je vraagt je misschien af waarom je exceptions zou willen genereren. Het antwoord is dat als je een module programmeert, en er een fout kan optreden in een van de functies in de module (bijvoorbeeld omdat het hoofdprogramma de verkeerde parameters meegeeft), het eigenlijk niet de bedoeling is dat je foutmeldingen print. Het is veel netter om een exception te genereren, en het hoofdprogramma deze exception af te laten handelen. Hier is een voorbeeld waarbij een foutieve waarde in de setter-property via een exception terug wordt gegeven aan de aanroepende methode. Daar kan opnieuw exception handling toegepast worden.

```
@geboortejjaar.setter
def geboortejjaar(self, value):
    # controle of de nieuwe waarde wel degelijk een int is
    if isinstance(value, int) and value >= 1950 and value < datetime.now().year:
        self.__geboortejjaar = value
    else:
        # self.__geboortejjaar = 2017
        raise ValueError("Geen geldige geboortedatum!")
```