Fred Vellinga BI Services

Python Introductie

Dit is een samenvatting van:
'De Programmeursleerling', Leren coderen met Python 3, van Pieter Spronck, Version 1.0.16, 24–10–2017. De meest recente versie van dat boek staat hier: www.spronck.net/pythonbook/. Voorbeelden heb ik aangepast. Hoofdstuk 22, Events, Hoofdstuk 29, Pandas, Hoofdstuk 30, PostgreSQL, en Hoofdstuk 31, fraude regels, zijn eigen brouwsels.

Code is getest in Visual Studio 2017/2019.

Code	: C:\Users\Fred\source\repos\P1\P1\P1.py
Python interpreter pad	: C:\Program Files\Python36\Python
IDLE shell	: C:\Program Files\Python36\Lib\idlelib\idle.pyw
Help 1	: C:\Program Files\Python36\Doc\python362.chm
Help 2	: C:\Python\python-3.7.2rc1-docs-html\library\index.html
Help 3	: C:\Python\docs-pdf
Built-in type methodes	Hierbij **link** naar zogenaamde built-in type methodes.

Opmerking: Je kunt ook de Anaconda Python distributie installeren: **www.anaconda.com**. Deze Python distributie komt met veel extra modules. Je krijgt dan Spyder als je ontwikkel IDE. Anaconda levert bijvoorbeeld wel standaard de BeautifulSoup module, waaraan gerefereerd word in paragraaf 27.4. Aan de andere kant, met Visual Studio kun je ook makkelijk modules toevoegen.

De macro Ctr+L maakt een streepjes lijst.
De macro Ctr+G maakt een grid/tabel.

1 Introductie 5
1.1 Python2 of Python3? 5
1.2 Python3 wetenwaardigheden 5
1.3 IDLE shell 5
2 Expressies 5
2.1 Data types 5
2.2 Strings 5
2.3 Integers en floats 6
2.4 Operator berekeningen 6
2.5 Type casting 6
2.6 Commentaar 6
2.7 Code Fouten 6
3 Variabelen en waardes 6
3.1 Constanten 7
3.2 Soft typing versus hard typing 7
3.3 Verkorte operatoren 7
4 Functies I 7
4.1 None 8
4.2 Alle ingebouwde functies 9
4.3 Modules 9
4.4 Random module 9
5 Condities 10
5.1 Boolean expressies 10
5.2 Logische operatoren 10
5.3 Conditionele statements 11
5.4 Meer-weg beslissingen 11
5.5 Case statement 12
5.6 Vroegtijdig afbreken 12
6 Iteraties 12
6.1 while loop 12
6.2 Eindeloze loops 13
6.3 for loop 13
6.4 for loop met strings 13
6.5 for loop met een getallenreeks 14
6.6 for loop met een handmatige collectie 14
6.7 Loop controle 15
6.8 Geneste loops 15
6.9 Opvangen invoer 15
7 Functies II 15
7.1 Het maken van functies 16
7.2 Parameter variabele data types 16
7.3 Default parameter waardes 16
7.4 Meerdere retourwaardes 17
7.5 Lokale/Global variabelen. Scope en levensduur 17
7.6 Globale variabelen 17
7.7 Modules 18
7.8 Anonieme functies 19
8 Recursie 19

9 Strings 20
9.1 Strings over meerdere regels 20
9.2 Speciale tekens 20
9.3 String manipulatie 20
9.4 Strings zijn onveranderbaar 21
9.5 String methodes 21
9.6 Codering van tekens (ord() en chr()) 21
9.7 strings met parameters 22
9.8 Slicing 22
9.9 ASCII tabel 23
10 Tuples 23
10.1 Toepassing voorbeeld met tuples 24
10.2 Enumerate (=opsomming) tuples/lists 24
11 Lists 25
11.1 List operaties 25
11.2 List methodes 26
11.3 Key in sort() 27
11.4 `is` operator 28
11.5 Lists als functie argumenten 28
11.6 Geneste lists 29
11.7 List casting 29
11.8 List comprehensions 29
12 Dictionaries 30
12.1 Dictionary methodes (een sub-set) 30
12.2 Keys 31
12.3 Opslaan van complexe waardes 31
12.4 Snelheid 32
13 Sets 32
13.1 Set methodes (een sub-set) 32
13.2 Frozensets 33
14 Besturingssysteem 34
14.1 Bestandssysteem 34
14.2 os functies (sub-set) 34
14.3 Voorbeelden 34
15 Tekstbestanden 35
15.1 Handles, pointers en buffers 35
15.2 Lezen van tekstbestanden 36
15.3 Welke leesmethode gebruiken? 37
15.4 Schrijven in tekstbestanden 37
15.5 Toevoegen aan tekstbestanden 38
15.6 `os.path` methodes 38
15.7 Encoding 39
16 Exceptions 39
16.1 Exception codes 40
16.2 Voorbeeld 40
16.3 Genereren van exceptions 40
17 Binaire Bestanden 40
17.1 Voorbeelden 40
17.2 Pointer Positie 42
17.3 Hexadecimaal schrijven 42
18 Bitsgewijze Operatoren 43
18.1 Codering van tekst 43
18.2 Coderen van getallen 43
18.3 Manipulatie van bits 44
18.4 Het nut van bitsgewijze operaties 45
19 Object Oriëntatie 46
19.1 Klassen, objecten, en hiërarchieën 47
19.2 Object oriëntatie 47
19.3 Methodes 49
19.4 Nesten van klassen 49

19.5 Kopieën en referenties 50
19.6 Geheugenbeheer 51
20 Operator Overloading 51
20.1 Sequenties 53
21 Function overloading 54
22 Events 55
23 Overerving 57
23.1 Meervoudige overerving 58
23.2 Uitbreiden en overschrijven 58
23.3 Interfaces 59
24 Iteratoren en Generatoren 60
24.1 Iteratoren 60
24.2 Iterabel object 60
24.3 Gedelegeerde iteratie 62
24.4 Functies 62
24.5 Generatoren 63
24.6 Generator expressies 63
24.7 itertools module 64
25 Command Line Verwerking 65
25.1 Command line argumenten 65
26 Reguliere Expressies 66
26.1 De re module 66
26.2 Match objecten 67
26.3 Reguliere expressies schrijven 67
26.4 Groeperen 68
26.5 Refereren binnen een reguliere expressie 69
26.6 Vervangen 70
27 Bestandsformaten 70
27.1 CSV 70
27.2 Pickling 72
27.3 JavaScript Object Notation (JSON) 73
27.4 HTML en XML 73
28 Diverse Nuttige Modules 74
28.1 datetime 74
28.2 collections 75
28.3 urllib 76
28.4 glob 76
28.5 statistics 76
29 Pandas 77
29.1 Opvullen van lege waarden 77
30 PostgreSQL 78
31 Fraude regels 84
32 Terminologie 89

1 Introductie

1.1 Python2 of Python3?

Python2 is niet volledig compatibel met Python3. In Python2 is 7/4 is 1, en niet 1.75. Dit omdat 7 en 4 gehele getallen zijn *('integers'*), en daarom is de uitkomst van de deling ook een geheel getal. Als je 1.75 als uitkomst wilt hebben, moet minstens een van de twee getallen een gebroken getal zijn. 7.0/4 heeft als uitkomst 1.75. Python3 doet de conversie naar gebroken getallen automatisch. In Python3 is de uitkomst van 7/4 dus 1.75. Veel Python2 programma's zijn echter geschreven onder de aanname dat een integer-deling naar beneden afrondt, wat betekent dat deze programma's niet meer correct functioneren als je ze uitvoert als Python 3 programma's. Dus zijn Python2 en Python3 programma's niet compatibel. Deze uitleg behandelt Python3. Ondersteuning voor Python2 stopt per 01-jan-2020, de zogenaamde End-Of-Life datum.

1.2 Python3 wetenwaardigheden

- In Python3 zijn alle (data) types geïmplementeerd als klassen, classes.
- Python3 is gebaseerd op Unicode. Python3 strings zijn Unicode strings. Je merkt geen verschil zolang de strings alleen ASCII-tekens bevatten.
- Python3 ondersteunt ook Unicode in namen voor variabelen, functies, classes, en methodes.
- Python3 is veel meer gebaseerd op iteratoren en generatoren dan Python2, wat een hoop voordelen heeft, vooral waar het snelheid en geheugengebruik betreft.

1.3 IDLE shell

Over het algemeen wordt voor Python bestanden de extensie **`.py`** gebruikt. Windows omgevingen hebben meestal ook de *IDLE* command shell tot beschikking. De *IDLE* shell heeft een menubalk en staat hier: C:\Program Files\Python36\Lib\idlelib\idle.pyw (Als je 3.6 Anaconda hebt, trouwens, wat ik eerst had.)

2 Expressies

Expressies of *uitdrukkingen* zijn code eenheden die één resultaat opleveren; hoe druk je de code uit om een resultaat te krijgen. Een expressie is een combinatie van één of meerdere waardes (zoals strings, integers, of floats) met behulp van operatoren, die dan een nieuwe waarde oplevert. Expressies kun je voorstellen als berekeningen. Verwar expressies niet met een *scalar* functies die ook één resultaat opleveren. Expressies kun je zo lang maken als je zelf wilt, zolang het syntactisch maar klopt.

2.1 Data types

Elke waarde (of expressie of functie resultaat) heeft een data type. Op basis van het data type weet het programma hoe er met de waardes gerekend moet worden. 30-12-2018 is een datum data type. Als je daar het getal 6 bij optelt dan wordt het antwoord 05-01-2019. Dus de **+** (en **-**) operator wordt ondersteunt voor datum data types als je intervallen wilt optellen/aftrekken. Maar 30-12-2018 vermenigvuldigen met 2 kan niet. Je kan ook geen datums van elkaar optellen/aftrekken. Wat zou dat moeten opleveren? De meest voorkomende data types zijn *strings* (tekst), *integers* (gehele getallen), en *floats* (gebroken getallen).

2.2 Strings

Een *string* is een tekst, omsloten door dubbele of enkele aanhalingstekens. "*appel*" is gelijk aan '*appel*' maar '*appel*" of "*appel*' mag niet. Als de tekst een enkel aanhalingsteken bevat, moet je deze omsluiten met dubbele aanhalingstekens; "*mango*'*s*" is een correcte string, maar '*mango*'*s*' niet. Hetzelfde geldt voor een dubbel aanhalingsteken in een string; die moet dan omsloten worden door enkele aanhalingstekens. Dus '*mango*"*s*' is tekstueel fout, maar syntactisch niet.

Het is handiger om de '*backslash*' (****) voor ieder dubbel- of enkel aanhalingsteken te plaatsen als zo een teken midden in een string staat. Dit heet ook wel een *literal*; letterlijk nemen van wat er staat, of beter 'het volgende karakter letterlijk nemen'; **`print('mango\'s')`**
Je kunt de **** ook beschouwen als beschermer. Het beschermt het volgende karakter van zijn interpretatie. Want er zijn veel karakter combinaties die een speciale interpretatie betekenis hebben. Zo is **`\n`** een nieuwe regel. **`print('mango\'s\nnieuwe regel')`** maar **`print('mango\'s\\nnieuwe regel'`** niet.
Om de backslash zelf te beschermen zet je er een backslash voor: **`print('\\')`**

```
print('Dit', 'is een test', 34, 'om getallen af te drukken:', 100)
```

2.3 Integers en floats

Integers zijn gehele getallen en *floats* zijn gebroken getallen. Integer berekeningen gaan veel sneller dan float berekeningen. Een integer data type neemt ook minder geheugen in beslag.
Dit is allemaal geldig: `print(5); print(+5); print(-5); print(0.01); print(-0.01);`
De '`;`' is een scheidingsteken en scheidt statements van elkaar.

2.4 Operator berekeningen

Eenvoudige berekeningen worden gemaakt door twee waardes te combineren met een operator ertussenin. Een aantal operatoren zijn:

`+` Optelling. Kan ook gebruikt worden om strings aan elkaar te plakken; string concatenatie.
```
print('tot ' + 'ziens')
```
`-` Aftrekking.

`*` Vermenigvuldiging. Kan ook gebruikt worden om strings te dupliceren.
```
print(3 * 'tot ziens '); print('tot ziens ' * 3)
```
Het resultaat is dus "tot ziens tot ziens tot ziens"

`/` Deling.

`//` Integer deling. Heet ook wel '*floor division*' en rond af naar beneden op een geheel getal. Als er floats in de berekening zitten, is het resultaat nog steeds een geheel getal maar wel van het data type float. Er vindt geen data type conversie plaats.

`**` Machtsverheffing.

`%` Modulo. Deze operator produceert de rest die overblijft na deling. Bijvoorbeeld: 14 gedeeld door 5 is 2.8, maar de modulo is 4, er gaat maar 2x5 in 14. `print(14%5)`

2.5 Type casting

Casting wordt gebruikt om waardes van data type te laten veranderen. Je *cast*(=*giet*) de waarde naar een ander tdata ype. Een aantal belangrijke casting functies zijn:

`int()` Zet de waarde om naar een integer en indien nodig afgerond naar beneden.

`float()` Zet de waarde om naar een float, waarbij `.0`, als het een geheel getal is, wordt toegevoegd.

`str()` Zet de waarde om naar een string.

`tuple()` Zet de waarde om naar een tuple.

`list()` Zet de waarde om naar een list.

2.6 Commentaar

Gebruik de '*hash mark*' (`#`) als commentaar karakter. Alles wat na de `#` volgt op een regel is commentaar. Wil je commentaar spreiden over meerdere regels gebruik dan `"""` om mee te beginnen en `"""` om af te sluiten. Alles daartussen is commentaar. (Het paartje `'''` mag ook).

2.7 Code Fouten

Je hebt twee soorten code fouten.

`Syntax errors` De code bevat een syntax fout en wil niet compileren en de code kan dan niet worden uitgevoerd. Een syntax fout: `print("Kees')`

`Runtime errors` De code is syntactisch goed, compileert en wordt uitgevoerd, maar tijdens het runnen treedt er een fout op; de runtime error. Bijvoorbeeld een `ZeroDivisionError`, die aangeeft dat je probeerde te delen door nul. Een runtime error: `print('Kees', 5/0)`

3 Variabelen en waardes

Een variabele is een plaats in het geheugen van de computer die een naam heeft gekregen, en waarin je een waarde kunt opslaan en wordt de '*variabele naam*' genoemd. Om een variabele te maken moet je een waarde '*toekennen*' aan een gekozen naam middels het is-gelijk `(=)` symbool, de '*assignment operator*'. Aan de linkerkant van het is-gelijk symbool zet je de variabele naam, en aan de rechterkant de waarde. Voorbeeld:

```
x = 10
y = x*5
print(x, y)   # er komt wel een spatie tussen de waarde 10 en 50
```

De rechterkant van een assignment wordt altijd geheel geëvalueerd voordat de toekenning plaatsvindt.

Variabele namen moeten voldoen aan de volgende conventies:
- Mag slechts bestaan uit een combinatie van letters, cijfers, en '*underscores*' (_).
- Moet beginnen met een letter of een underscore.
- Mag geen gereserveerd woord zijn.
- Hoofd- en kleine letters mogen door elkaar gebruikt worden.
- Python is '*case sensitive*', dus gevoelig voor de verschillen tussen hoofd- en klein letters. De naam '*variabele_een*' is niet hetzelfde als '*Variabele_een*'.

3.1 Constanten

Veel programmeertalen geven je de mogelijkheid om '*constanten*' te definiëren. Dat zijn variabelen die eenmalig een waarde toegekend krijgen die daarna niet meer veranderd kunnen worden. Python ondersteunt **geen** constanten.

3.2 Soft typing versus hard typing

Alle variabelen hebben een data type. Bij sommige programmeertalen moet je per variabele eerst het data type bepalen (declare), bij anderen is dat niet nodig en wordt het data type automatisch bepaalt.

`Hard typing` Variabele declaratie is verplicht. Voordeel is dat je nooit een waarde van het verkeerde data type in een variabele kunt stoppen. Tijdens compileren krijg je dan een syntax error. Wordt het daar niet ontdekt, dan krijg je een runtime error.

`Soft typing` Het data type wordt bepaald aan de hand van de waarde die je erin stopt. Dus het data type van een variabele kan veranderen. Python doet '*Soft typing*'. Dit werkt sneller, maar geeft grotere kans op runtime errors of helemaal geen fouten met onbedoelde uitvoer.

Hier zie je een voorbeeldje. De functie **`type()`** laat het data type zien.

```
a = 3;     print(type(a))     # geeft <class 'int'>
a = 3.0;   print(type(a))     # geeft <class 'float'>
a = "3.0"; print(type(a))     # geeft <class 'str'>
```

Omdat variabelen een data type hebben, past het effect van operatoren zich aan de types van de variabelen aan. Als de **+** operator gebruikt wordt op getal data types zal het een optelling worden, maar betreft het variabelen met tekst, dan zal er een string concatenatie ('*plakken*') plaatsvinden.

3.3 Verkorte operatoren

Python bevat een aantal '*verkorte notaties*' om de inhoud van variabelen aan te passen.

+= **`a += b`**; Is verkorte notatie voor **`a=a+b`**
-= **`a -= b`**; Is verkorte notatie voor **`a=a-b`**
*= **`a *= b`**; Is verkorte notatie voor **`a=a*b`**
/= **`a /= b`**; Is verkorte notatie voor **`a=a/b`**
= **`a ** b`; Is verkorte notatie voor **`a=a**b`**

De lijst is niet compleet. Deze wordt vaak gebruikt **`; i+=1`**. Je krijgt een index teller.

4 Functies I

Net als variabele namen, mogen functienamen alleen bestaan uit letters, cijfers, en underscores, en mogen ze niet starten met een cijfer. Functie parameters kunnen op twee manieren worden doorgegeven:

`Passed by value` De functie heeft geen toegang tot de variabelen die als parameters gebruikt worden. Er worden kopieën van de waardes doorgegeven. Deze functies heten ook wel '***pure functies***' omdat ze **geen** invloed hebben op de variabelen.

`Passed by reference` De functie variabelen kunnen wel worden wel aangepast. Deze functies heten '***modifier functies***' omdat ze **wel** invloed hebben op de variabelen. **Let op:** Je kunt dit niet zelf bepalen zoals bij andere talen. Het data type bepaalt of je de variabele binnen de functie mag aanpassen. Een globale string kun je bijvoorbeeld niet aanpassen binnen een functie. Dan kan wel als je er een *Class* van maakt.

- Een functie kan een mix hebben van beide typen parameters.
- Een functie kan één of meer retourwaardes hebben, de meesten, of geen retourwaarde, de **`print()`** functie bijvoorbeeld. Er wordt dan een **`None`** teruggegeven. Dit heet ook wel **`void`**(=leeg) functie.

4.1 None

Binnen Python is **None** wat in andere talen wel als **NULL** wordt gezien. **None** is een *niets* waarde. Let op, Python is case sensitive, dus **None** is een gereserveerd keyword, terwijl **none** een variabele is.

```
none = 2
None = 2                # dit mag niet
print(print("Test"))    # print eerst Test, dan None op volgende regel
print(None)             # print ook None
print(none)             # foutmelding als none niet is geïnitialiseerd
```

Omdat de functie **print()** geen retourwaarde heeft, vult Python daar zelf **None** voor in. Dat wordt dan afgedrukt.

Wat functies, anders dan **int()**, **float()** en **str()** die al eerder zijn genoemd.

abs() Geeft de absolute waarde van een getal. Dus een negatief getal wordt een positief getal.

max() Geeft de grootste of maximale waarde van alle opgegeven argumenten.

min() Geeft de kleinste of minimale waarde van alle opgegeven argumenten.

pow() Power of machtsverheffen. Optioneel mag je een derde parameter meegeven, en dan wordt de berekening als volgt: **x**y%z.** Dus de modulo wordt toegepast na het machtsverheffen. Maar alle drie de parameters moeten dan wel een integer zijn.

round() Afronding op gehele getalen. Optioneel mag je een tweede parameter meegeven die aangeeft hoeveel cijfers achter de komma behouden moeten worden.

len() Geeft de lengte van de parameter waarde terug. Meest gebuikt voor string lengte bepaling.

input() Interactief gebeuren. Vraagt om input en geeft dat terug.

```
x=input("hallo: ")
print(x)
```

print() Print is wat geavanceerder dan op het eerste gezicht lijkt. Dit is de syntax:
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
Normaal gaat een print naar de volgend regel **als alle objecten zijn afgedrukt** en worden ze afgedrukt zoals de object waardes ze aangeven. Maar je kunt ze ook naar een stream sturen en je kunt een separator opgegeven en bepalen wat er met het einde van de regel gebeurt. Dat kan ook een tab worden; '**\t**'

format() Een functie die het mogelijk maakt een geformatteerde string te bouwen, dus een string waarin bepaalde waardes op een specifieke geformatteerde manier worden afgedrukt Dit is de syntax:
format(*value*[, *format_spec*])
En het gaat dan om *format_spec*. De volgende twee code regels geven hetzelfde resultaat.

```
print(7/11)                  # 0.6363636363636364
print("{:.1f}".format(7/11))  # 0.6
print(format(7/11,".1f"))     # 0.6
```

De tweede format is een functie, de eerste is het een (string) methode. Omdat alle data typen binnen Python een klasse zijn, zijn de waardes een instantie van die klasse en wordt het een object. Op een object kun je methodes (zijn functies) loslaten.

4.2 Alle ingebouwde functies

Hieronder alle ingebouwde functies die zonder importeren van een module zouden moeten werken. Het zijn links naar de 3.7 documentatie.

abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Alle zogenaamde *built-in* types of built-in functie methodes staan **hier**.

4.3 Modules

Modules bevatten een verzameling functies gegroepeerd rondom een bepaald thema. Om die functies aan te roepen moet je die module importeren. Er zijn heel veel modules. Verwar een **`module`** niet met een **`class`**, alhoewel een module wel een verzameling classes (kunnen) zijn. (Python3 is object georiënteerd). Er zijn drie methodes om een functie uit een module aan te roepen. Als voorbeeld nemen we de **`math`** module.

```
import math
print(math.degrees(1))
```
Alle functies in module **`math`** zijn nu beschikbaar maar je moet altijd de module naam ervoor zetten om ze aan te roepen.

```
import math as mt
print(mt.degrees(1))
```
Zelfde als hierboven maar hier *alias* je de module naam naar een andere naam die je dan gebruikt.

```
from math import sqrt
print(sqrt(4))
```
Hier importeer je alleen de functie **`sqrt`** die je dan ook als zodanig mag aanroepen.

```
from math import sqrt, cosh
```
Hier importeer je dus twee functies. Je kunt zoveel functies importeren als je wilt. Functies zijn komma gescheiden.

```
from math import sqrt as squareroot
print(squareroot(16))
```
Hier *alias* je de functienaam naar een andere functienaam. Het zal vaak voorkomen dat module functies dezelfde naam hebben.

4.4 Random module

De **`random`** module bevat functies die '*pseudotoevalsgetallen*' genereren. Het zijn *pseudotoevalsgetallen* omdat het voor digitale computers onmogelijk is om echt toevalsgetallen te genereren.

```
from random import random, randint, seed
seed()                  # initialiseren van de random generator
print(random())
print('Een toevalsgetal tussen 1 en 10 is:', randint(1,10))
print('Een ander is:', randint(1,10))
seed(0)                 # genereert steeds dezelfde getallen, voor test doeleinden
print('3 toevalsgetallen zijn:', random(), random(), random())
seed(0)
print('Dezelfde 3 zijn:', random(), random(), random())
```

`seed()` initialiseert de toevalsgetal generator van Python. Als je een lijst van toevalsgetallen wilt hebben die iedere keer hetzelfde is voor je programma, roep dan **`seed()`** aan met een vast getal, bijvoorbeeld **`seed(0)`**. Reset de generator door **`seed`** zonder parameters, **`seed()`**, aan te roepen.

5 Condities

'*Conditionele statements*' of '*condities*' zijn statements die aftakkingen in de code mogelijk maken en stukken code wel of niet uitvoeren.

5.1 Boolean expressies

Een expressie die evalueert als **`True`** of **`False`** heet een '*boolean expressie*'. De meest gebruikte boolean expressies zijn vergelijkingen met een vergelijkingsoperator.

`<`	Kleiner dan.
`>`	Groter dan.
`<=`	Kleiner dan of gelijk aan.
`>=`	Groter dan of gelijk aan.
`==`	Gelijk aan.
`!=`	Niet gelijk aan.
`IN`	De **`IN`** operator test of een waarde voorkomt in een collectie, als de waarde links van de **`IN`** staat, en de collectie rechts van de **`IN`**. Werkt op meer data types dan alleen strings.
`NOT IN`	De inverse van de **`IN`** operator.

Vergelijksoperatoren kunnen gebruikt worden voor zowel getallen als strings. Voor strings; hoofdletters zijn kleiner dan kleine letters, en cijfers kleiner dan alle letters, en de '8' is kleiner dan de '9'.

```
print("1.", 2 < 5)                  # geeft True
print("2.", 3 > 3)                  # geeft False
print("3.", 3 >= 3)                 # geeft True
print("4.", 3 == 3.0)               # geeft True
print("5.", 3 == "3")               # geeft false
print("6.", 3 <= "3")               # deze operator mag je hier niet gebruiken; Fout!
print("7.", "syntax" == "syntax")   # geeft True
print("8.", "syntax" == "semantiek") # geeft False
print("10.", "Python" != "rotzooi")  # geeft True
print("11.", "Python" > "Perl")     # geeft True
print("12.", "banaan" < "mango")    # geeft True
print("13.", "banaan" < "Mango")    # geeft False
```

Voorbeelden met de **`IN`** operator:

```
print('y' in "Python")                    # geeft True
print('i,j,t' in "Pijthon")               # geeft False
print('i,j,t' in "Pi,j,thon")             # geeft True
print('ijn' in "Python, Pithon, Pijthon") # geeft False
print('jth' in "Python, Pithon, Pijthon") # geeft True
```

5.2 Logische operatoren

Boolean expressies kunnen gecombineerd worden middels logische operatoren. Er zijn drie logische operatoren: **`and`**, **`or`**, en **`not`**. Voorbeelden:

```
t = True
f = False
print(t and t)  # geeft True
print(t and f)  # geeft False
print(f and t)  # geeft False
print(f and f)  # geeft False
print(t or t)   # geeft True
print(t or f)   # geeft True
print(f or t)   # geeft True
print(f or f)   # geeft False
print(not t)    # geeft False
print(not f)    # geeft True
```

Kijk uit met het gebruik van logische operatoren, want een combinatie van **`and`**s en **`or`**s kan leiden tot onverwachte resultaten. Gebruik haakjes om te zorgen dat ze in de gewenste volgorde geëvalueerd worden.

5.3 Conditionele statements

Conditionele statements worden ook wel '*condities*' of '*if statements*' genoemd. De syntax is:

```
if <boolean_expressie>:
  <acties>
else:
  <acties>
```

- De dubbele punt (:) achter de *boolean_expressie* en de **`else`** is verplicht.
- De **`<acties>`** moeten inspringen, dat is verplicht. Zonder correcte inspringing kan Python niet zien welke regels code een codeblok vormen, en kan daarom niet de code uitvoeren zoals bedoelt.
- Het woord **`else`** moet uitgelijnd zijn met het woord **`if`** waar het bij hoort. Anders krijg je ofwel een tabulatie/inspring/indent fout, ofwel de code doet niet wat verwacht wordt.

```
x = input('Geef waarde: ')
if x == 5:
  print('De THEN tak')
  print("x is 5")
  x = 1
else:
  print('De ELSE tak')
  print('x is niet 5')
  x = 0
print(x) # Dit statement hoort niet bij een conditie blok.
```

Python beschouwt statements die elkaar opvolgen en die hetzelfde niveau van inspringen (*indenting*) hebben als één codeblok. Inspringen is meestal twee of vier spaties. De volgende code geeft dus fouten.

```
x = input('Geef waarde: ')
if x == 5:
  print('De THEN tak')
print("x is 5")       # hier gaat het fout
else:
  print('De ELSE tak')
print('x is niet 5')  # Deze gaat niet fout, want hoort niet langer bij de else tak
```

5.4 Meer-weg beslissingen

Gebruik het **`elif`** (else if) statement om meer-weg beslissingen te implementeren. Hieronder staat een wat uitgebreider voorbeeld. Het zet een oneindige *loop* of *lus* en maakt gebruik van het **`break`** statement om uit de code te breken. De code gaat dan ergens anders verder.

```
i = 0                                          # initialiseer de loop counter
while True:                                    # een oneindige loop
  i+=1                                         # verhoog met 1
  leeftijd = int(input('Geef leeftijd: '))     # input geeft string, omzetten naar integer
  if leeftijd < 12:                            # anders werkt deze conditie niet
    print("Je bent een kind!")
    print("Ga maar lekker buiten spelen!")
    if leeftijd <= 1:
      print("Wat nou, je Je bent nog een baby!")
  elif leeftijd < 18:
    print("Je bent een teenager!")
  elif leeftijd < 30:
    print("Je bent nog jong!")
  elif leeftijd < 50:
    print("Beginnen grijze haren te komen?")
  else:
    print("Wegen de jaren zwaar?")
    print("Valt wel mee hoor!")
  if i == 10 or leeftijd == 0:
    break                                      # hier verlaat je de oneindige loop
```

5.5 Case statement

Het blijkt dat Python geen **`case`** statement of **`switch`** statement kent. Apart. Je kunt daarvoor '*dictionaries*' gebruiken. Dat is een manier om ongeordende data te structureren. Hieronder staat een voorbeeld die een dictionary definieert in een functie.

```
def numbers_to_strings(number):
    case = {0: "zero",       # hier definieer je de dictionary, de key is dan 0, 1 of 2
            1: "one",
            2: "two"
           }
    return case.get(number)  # hier haal je de key waarde op
print(numbers_to_strings(1)) # geeft one
print(numbers_to_strings(9)) # geeft None, omdat komt niet voor in de dictionary lijst
```

Als de *key* of *sleutel* niet voorkomt kun je een andere waarde dan **`None`** geven door dit te doen:
`return case.get(number, "de default waarde")`
Een **`case`** statement heeft altijd een default waarde, dat is de '*laatste*' waarde.

5.6 Vroegtijdig afbreken

Gebruikt de **`exit()`** functie om een programma vroegtijdig beëindigen. Het programma stopt.

```
from sys import exit       # pak de exit uit deze module
i = 0
while True:                # oneindige loop
  i+=1
  print(i)
  input('om te pauzeren, anders zie je het effect niet goed')
  if i == 5:
    exit(0)                # hiermee stopt het programma
```

In dit voorbeeld wordt de **`exit()`** retourwaarde niet opgevangen. Dat kan wel. Hieronder een eenvoudig opzetje.

```
import sys
try:
  sys.exit(12)                 # Geen nut om te assignen want code vliegt er meteen uit
except SystemExit as err:
  # Hier kan exit code afhandeling gebeuren
  print(err.args)              # (12,)
  input("Press any key . . .") # Nodig om te laten zien wat er gebeurt
finally: exit()
```

6 Iteraties

De klasse programmeerconstructies die herhalingen mogelijk maken heten '*iteraties*', ook wel '*loops*' of '*lussen*' genoemd. Python kent twee looptechnieken:

- **`While`** loop.
- **`For`** loop.

De *until loop*, een variant op de *while loop*, kent Python niet. Het verschil zit hem in waar de conditie zit. Bij de *while loop* aan het begin van de loop. Bij de *until loop* aan het einde van de loop.

6.1 while loop

De syntax van de *while loop*:

```
while <boolean_expressie>:
  <acties>
else:
  <else_acties>
```

De **`while`** statement test een *boolean_expressie*. Als de expressie **`True`** oplevert wordt het codeblok onder de **`while`** uitgevoerd. Daarna gaat de *programcounter* terug naar de **`while`** statement en wordt de *expressie* opnieuw getest. En dus zit je in een loop. Je komt alleen maar uit de loop als de *expressie* **`False`** test, of je zet ergens een **`break`** statement. De **`else`** tak wordt alleen uitgevoerd als de *expressie* **`False`** oplevert.

Bij het voorbeeld hieronder wordt de code zes keer uitgevoerd. Tenzij je als gebruiker ingrijpt.

```
i = 0                                          # initialiseer de loop counter
while (i <= 5):                                # exit criteria zetten
  i+=1                                         # verhoog met 1
  print(i)
  exit_criteria = input("Input to escape: ")   # Voer iets in om te ontsnappen
  if i == 6:                                   # geen 5 !
    print("De laatste iteratie")
  if len(exit_criteria) != 0:                  # een lege string heeft altijd lengte 0
    break
print('i waarde: ', i)                         # waarde is 6 als volledig doorlopen
```

Hier hetzelfde voorbeeld maar dan met de **`else`** tak erin.

```
i = 0                                          # initialiseer de loop counter
while (i <= 5):                                # exit criteria zetten
  i+=1                                         # verhoog met 1
  print(i)
  exit_criteria = input("Input to escape: ")   # Voer iets in om te ontsnappen
  if len(exit_criteria) != 0:                  # een lege string heeft altijd lengte 0
    break                                      # ontsnap uit de loop
else:
  print("Einde while loop. i:", i)             # Als de while loop conditie False is. i=6
```

Als de code het **`break`** statement raakt, dan wordt de **`else`** tak **niet** uitgevoerd. De **`else`** tak maakt ook onderdeel uit van het gehele **`while`** statement en wordt dus in zijn geheel overgeslagen.

6.2 Eindeloze loops

While loops kunnen resulteren in zogenaamde *oneindige loops*. De *boolean_expressie* wordt nooit **`False`**. Dat komt doordat je de zogenaamde *exit_criteria* niet goed hebt gedefinieerd. Je merkt dat doordat het programma onverwacht lang blijft lopen. Het programma *hangt* dan.

6.3 for loop

For loops zijn gemakkelijker en veiliger te gebruiken dan *while loops*, maar kunnen niet voor alle iteratie problemen gebruikt worden. Alles wat een *for loop* kan doen, kan een *while loop* ook doen, maar niet andersom. Bij een *for loop* hoef je de *variabele* niet te initialiseren.

De syntax voor de *for loop* is:

```
for <variabele> in <collectie>:
  <acties>
else:
  <else_acties>
```

Een *for loop* verwerkt een *collectie* van items één voor één in de volgorde waarin ze worden aangeboden. Iedere cyclus verwerkt één item door het in de variabele te stoppen die naast de **`for`** staat. Die variabele kan dan gebruikt worden in het codeblok van de loop maar bestaat ook nog als de *for loop* is afgelopen. De **`else`** tak wordt alleen uitgevoerd als de gehele *collectie* is doorlopen, dus aan het eind.

Een '*collectie*' kan van alles zijn; collectie van woorden, collectie van karakters, collectie van getallen etc.

6.4 for loop met strings

Hier zie je dat de collectie een verzameling letters of karakters (characters) van een woord of string is. Elk karakter wordt hier afzonderlijk afgedrukt. Van het woord "banaan" maakt Python een *iterable* aan. Dat is een object waar je door heen kunt lopen.

```
for char in "banaan":
  print(char)
  if char == "n": break                # ook aan een for loop kun je ontsnappen
print('Klaar. Char waarde: ', char)
```

Dit werkt ook:

```
col = "Banaan"
for char in col:
  print(char)
print('Klaar. Char waarde: ', char)
```

Hier zie je dat als je de *collectie* aanpast dat geen invloed heeft op de *for loop* statement. De *collectie* wordt ergens in het geheugen gezet, het object, en heet dus de *iterable* en vandaar benaderd. Dat object is niet veranderbaar. De variabele *col* wel. Hoewel de namen hetzelfde zijn, zijn ze niet gelijk. (Je kunt ze ook als kopieën van elkaar beschouwen).

```
col = "Banaan"
for char in col:
  col = "Appel"                    # dit heeft geen effect op char variabele waarde
  print(char)
print('Klaar. col waarde: ', col)   # col zelf is wel Appel geworden
```

Hier dan een voorbeeld met **`else`** tak en **`break`** statement. Als het **`break`** statement wordt geraakt, dan wordt de **`else`** tak **niet** uitgevoerd.

```
for char in "qwerty":
  print(char)
  if char == "x": break                # ook een for loop kun je ontsnappen
else:
  print("Einde for loop. char:", char)  # het hele iterator object is doorlopen. char=y
```

6.5 for loop met een getallenreeks

De functie **`range()`** genereert een serie opeenvolgende (integer) getallen. Twee syntaxis bestaan:

`range(stop)` Een serie waardes tussen **`0`** en **`stop-1`**. Voorbeelden:

```
print(list(range(5)))     # [0, 1, 2, 3, 4]
print(list(range(-5)))    # [], werkt blijkbaar niet
```

`range(start, stop [,step])` Een serie waardes tussen start en **`stop-1`**. Als je step specificeert is dat de interval- of stapgrote. Voorbeelden:

```
print(list(range(-5,2)))   # [-5, -4, -3, -2, -1, 0, 1]
print(list(range(5,9)))    # [5, 6, 7, 8]
print(list(range(1,10,2))) # [1, 3, 5, 7, 9]
```

`List()` is een klasse (object) met een aantal methodes.

Hier een tweetal voorbeelden:

```
for var in range(10):       # tien getallen van 0 -9
  print(var)

for var in range(1, 11, 3): # vier getallen; 1,4,7,10
  print(var)
```

6.6 for loop met een handmatige collectie

Hier een aantal voorbeelden met hard gecodeerde collecties. Hier zie je dat je getallen en strings etc. door elkaar kunt gebruiken. Een serie items tussen haakjes is een '*tuple*'.

```
for var in (1,2,3,3):
  print(var)
for var in ("aap", "noot", "mies"):
  print(var)
for var in ("aap", 2,3, "noot", 1, "mies"):
  print(var)
```

Hier dan een voorbeeld met een *lijst* of *list* object met een mix van integers, floats en strings.

```
ObjList = list()            # initialiseer een lijst object van klasse list
ObjList.append(1)           # 4 items voeg je toe
ObjList.append('aap')
ObjList.append('noot')
ObjList.append(-12.2)
for var in ObjList:         # en hier print je ze één voor één af
  print(var)
```

6.7 Loop controle

Er zijn drie statements die je controle geven over de wijze waarop een loop uitgevoerd wordt:

Else De **else** tak van de **while** en **for** loop statement.

Break Het ontsnappen of afbreken van een loop. Het is altijd de binnenste loop. Als je geneste loops hebt, dan wordt alleen die loop afgebroken waar de **break** staat. Als bij geneste loops de **break** in de buitenste loop staat, dan wordt dus alles afgebroken.

Continue Dit is een variant op het **break** statement. Het huidige codeblok wordt gestopt en je keert meteen terug naar het begin van het **while** of **for** statement.

- **while** De boolean expressie wordt opnieuw geëvalueerd.
- **for** Het volgende item van de collectie wordt gepakt.

Het **continue** statement kan in het geval van de **while** loop tot een oneindige loop leiden.

6.8 Geneste loops

Je kunt een loop in een andere loop stoppen en die weer in een andere loop etc. Dat heet een geneste loop. Hoeveel lagen diep Python gaat weet ik niet. Hieronder een voorbeeldje.

```
i = 0; c = 0
while (i < 5):
  i+=1
  j = 0;                                          # de while j loop opnieuw initialiseren
  while (j < 3):
    j+=1
    for k in range(2):
      for l in "qwerty":
        c+=1                                      # 5x3x2x6=180 iteraties
        temp_var = "({}:{}:{}:{}:{})".format(i,j,k,l,c)
        print(temp_var)
```

6.9 Opvangen invoer

Hier een loopje om tekst altijd op dezelfde locatie op te vangen. De *curses* module is niet standaard. Zie hier de methodiek om een externe module te installeren.

```
import curses
scr = curses.initscr()                            # initaliseren van scherm of screen
curses.echo()                                     # onduidelijk of echt nodig
while True:
  scr.addstr(2, 2, "Invoer (0=ontsnappen)")       # op regel (2,2) zet je tekst neer
  scr.refresh()                                   # refresh nodig, anders zie je niks
  # getstr() geeft een byte object. Zet om naar tekst
  s = scr.getstr(2, 24, 48).decode(encoding="utf-8")
  if (s == "0"):
    break
  for i in range(len(s)):
    scr.addstr(2,24+i, " ")                       # Maak invoer regel leeg
# Je bent uit de loop. Maak alles schoon.
for i in range(23):
  scr.addstr(2, 2+i, " ")
scr.addstr(5, 5, "Je bent uit de loop")           # hier zit cursor einde regel
scr.move(5,5)                                     # hiet zit cursor op (5,5)
scr.refresh()
curses.beep()
scr.getch()                                       # wachten op een karakter
curses.endwin()                                   # de-initialiseren
```

7 Functies II

Voordelen van functies:

Encapsulatie	Het '*inpakken*' van code zonder kennis van de specifieke werking van de code.
Generalisatie	Het geschikt maken voor diverse situaties door gebruik te maken van parameters.
Beheersbaarheid	Het verdelen van een complex programma in gemakkelijk te bevatten delen.
Onderhoudbaarheid	Het gebruik maken van betekenisvolle functienamen en logische opdelingen om een programma beter lees- en begrijpbaar te maken.
Herbruikbaarheid	Het faciliteren van de overdraagbaarheid van code tussen programma's.
Recursie	Het beschikbaar maken van een techniek die '*recursie*' heet. Zie hoofdstuk 8.

7.1 Het maken van functies

Syntax van een functie is als volgt:

```
def <functie_naam>(<parameter_lijst>):
  <acties>
  return return_waardes
```

- Functies mogen alleen bestaan uit letters, cijfers, en underscores.
- Functies mogen niet beginnen met een cijfer.
- De **`return`** statement is optioneel. Het geeft minimaal een waarde terug. De **`print()`** functie geeft geen retourwaarde. Er wordt dan een **`None`** teruggegeven. Dit heet ook wel **`void`**(=leeg) functie. Niet helemaal waar; elke functie retourneert een waarde. The default waarde is **`None`**.
- Het **`return`** statement beëindigd ook de functie.
- Meerder **`return`** waardes zijn komma gescheiden.
- De parameter lijst bestaat uit nul of meer variabele namen, komma gescheiden.
- Voordat je een functie kunt aanroepen moet Python hem hebben gedefinieerd, dus kennen.
- De waarde van een parameter uit de *parameter_lijst* heet ook wel argument.
- Parameter zijn waardes die je kunt gebruiken binnen de functie en zijn lokaal aan de functie. Je kunt wel dezelfde naam buiten de functie gebruiken, maar dan is het een andere variabele.
- Parameters kunnen op twee manieren worden doorgegeven:

  `Passed by value` De functie heeft geen toegang heeft tot de variabelen die als parameters gebruikt worden. Er worden kopieën van de waardes doorgegeven. Deze functies heten ook wel '***pure functies***' omdat ze **geen** invloed hebben op de variabelen.

  `Passed by reference` De functie variabelen kunnen wel worden wel aangepast. Deze functies heten '***modifier functies***' omdat ze **wel** invloed hebben op de variabelen. **Let op:** Je kunt dit niet zelf bepalen zoals bij andere talen. Het data type bepaalt of je de variabele binnen de functie mag aanpassen. Een globale string kun je bijvoorbeeld niet aanpassen binnen een functie. Dan kan wel als je er een Class van maakt.
- Een functie kan een mix hebben van beide typen parameters.

Hier zie je hoe *passed by value* werkt. Beide stukjes code doen hetzelfde.

```
def func_multiply(a,b):
  a*=a
  b*=b
  return a*b
x = 2; y = 3
print(func_multiply(x,y), x, y) # 36 2 3
```

```
def func_multiply(a,b):
  a*=a
  b*=b
  return a*b
a = 2; b = 3
print(func_multiply(a,b), a, b) # 36 2 3
```

7.2 Parameter variabele data types

Variabele definitie is niet nodig in Python. Gebruik **`isinstance()`** om op data type te testen.

```
x = 12
if not isinstance(a,str): print('Geen string')
```

7.3 Default parameter waardes

Default parameter waarde(n) geeft je op door de **=** operator te gebruiken. De parameter is dan ook meteen optioneel. Zie voorbeeld hieronder.

```
def func_multiply(a=2,b=3):
  print(a,b)                  # 2, 3
  a*=a
  b*=b
  return a*b
print(func_multiply())        # 36
print(func_multiply(2,3))     # 36
print(func_multiply(b=3,a=2)) # 36
```

Je kan ook in de functieaanroep de parameter naam opgeven; dan maakt parameter volgorde niet meer uit. Hieronder zie je dat als je geen **`return`** statement gebruikt, een functie altijd **`None`** teruggeeft.

```
def func_add(a,b):
  print(a+b)
print(func_add(1,99)) # None wordt teruggeven
```

7.4 Meerdere retourwaardes

Je kunt meerdere waardes in één keer retourneren door er komma's tussen te zetten. Als je deze waardes wilt gebruiken in je programma na aanroep van de functies, moet je ze toekennen aan meerdere variabelen.

```
import datetime
def func_nieuwe_datum(y,m,d,increment):
  start = datetime.datetime(y,m,d)
  eind = start + datetime.timedelta(days=increment)
  return eind.year, eind.month, eind.day
y, m, d = func_nieuwe_datum(2018,12,30,565)
print("{}/{}/{}".format(y,m,d))                    # 2020/7/17
```

Je kunt het ook zo doen, en dan staat alles in een tuple. Bij meerdere retourwaardes wordt het een tuple.

```
ymd = func_nieuwe_datum(2018,12,30,565)
print(ymd)                                  # (2020, 7, 17)
```

7.5 Lokale/Global variabelen. Scope en levensduur

- Globale variabelen zijn geldig voor het gehele programma, maar niet binnen een functie.
- Lokale variabelen zijn geldig binnen de functie waar je ze definieert.
- Functie parameters zijn lokale variabelen.
- De scope (bereik) van een variabele geeft aan waar in de code de variabele bestaat/geldig is.
- De levensduur van de variabele geeft aan hoe lang een variabele bestaat.

7.6 Globale variabelen

Een globale variabele kun je alleen veranderen binnen een functie als je de variabele als zodanig bekent laat worden binnen een functie met het **`global`** statement. Het eerste code voorbeeld laat zien dat je **`gv1`** niet kunt veranderen, het tweede wel.

```
# hier zie je dat een globale variabele niet binnen een functie kunt veranderen
# de gv1 binnen de functie is een lokale variabele,
gv1 = "Globaal"
def func_gv1(a):
  gv1 = "Anders"           # dit is een lokale variabele
  a+="Lokaal"
  return a, gv1
print(func_gv1(gv1), gv1)  # ('GlobaalLokaal', 'Anders') Globaal
```

```
# hier zie je dat een globale variabele wel binnen een functie kunt veranderen
gv1 = "Globaal"
def func_gv2(a):
  global gv1              # je maakt bekent dat gv1 een globale variabele is
  gv1+="Anders"           # dit is dan de globale variabele
  a+="Lokaal"
  return a, gv1
print(func_gv2(gv1), gv1) # ('GlobaalLokaal', 'GlobaalAnders') GlobaalAnders
```

Maar in feite is dit niet *passed by reference*. Daar wil je een parameter meegeven die je kunt veranderen. Dat is afhankelijk van het data type. Het data type dat altijd veranderbaar is, is een Class. Het voorbeeld hieronder definieer een Class met slecht één variabele.

```
# Een Class, of klasse met een property, dat is een variabele
class ClsGv:
  gv = ''

def func_gv(a):
  a.gv = 'Lokaal' + a.gv
  return a.gv
ObjGv1 = ClsGv()                    # hier maak je instance van de Class, een object
ObjGv1.gv = 'Globaal1'              # hier geef je Class property gv een waarde
ObjGv2 = ClsGv()                    # hier maak je instance van de Class, een object
ObjGv2.gv = 'Globaal2'              # hier geef je Class property gv een waarde
# Retourneer: LokaalGlobaal1 LokaalGlobaal2 LokaalGlobaal1 LokaalGlobaal2
print(func_gv(ObjGv1), func_gv(ObjGv2), ObjGv1.gv, ObjGv2.gv)
```

Hieronder zelfde code als hierboven maar dan met een constructor, dat is de **`__init__()`** functie.

```
# Een Class, of klasse met een property, dat is een variabele. Hier met een constructor.
class ClsGv():
```

```
  def __init__(self, p_gv = None):
    if p_gv == None:
      self.gv = ''
    else:
      self.gv = p_gv

def func_gv(a):
  a.gv = 'Lokaal' + a.gv
  return a.gv
ObjGv1 = ClsGv()                        # hier maar je instance van de Class, een object
ObjGv1.gv = 'Globaal1'                  # hier geef je Class property gv een waarde
ObjGv2 = ClsGv('Globaal2')              # Instance en tegelijk Class property gv zetten
# Retourneer: LokaalGlobaal1 LokaalGlobaal2 LokaalGlobaal1 LokaalGlobaal2
print(func_gv(ObjGv1), func_gv(ObjGv2), ObjGv1.gv, ObjGv2.gv)
```

7.7 Modules

Je maakt een Python bestand met extensie **.py**, en plaatst er de functies in. Via het **import** statement importeer je de module. Je importeert of de hele module, of alleen de functies de je wilt.

Iets over de volgende constructie in een module, of in je eigen code, wat in essentie ook een module is.

```
if __name__ == '__main__':
  main()
```

De functie **main()** bevat feitelijk het hoofdprogramma dat andere functies kan aanroepen. Het Python bestand bevat code die als programma kan draaien, of functies die geïmporteerd kunnen worden in andere programma's. De bovenstaande constructie zorgt ervoor dat de code in **main()** alleen wordt uitgevoerd als het programma als een **separaat** programma is gestart, en niet als een module in een ander programma geladen is. Als het programma als module is geladen, kunnen alleen de functies in het programma worden geïmporteerd, en wordt de code voor **main()** genegeerd. Een Python bestand dat een dergelijke constructie bevat en dat voornamelijk als module wordt gebruikt, heeft vaak code in **main()** die de functies test. Dat kan nuttig zijn tijdens de ontwikkeling van de module. (Volgens mij kun je elke willekeurige functienaam voor **main()** gebruiken. Het gaat om **if __name__ == '__main__'**).

Hier een voorbeeld. Eerst de module.

```
'''
Een module met drie klassen, String, Integer en Float.
Die kun je gebruiken in functies om passed by reference te doen
'''
# Een string Class
class ClsStr():
  def __init__(self, p_gv:str = None):
    if p_gv == None:
      self.gv = ''
    else:
      self.gv = str(p_gv)
# Een integer class
class ClsInt():
  def __init__(self, p_gv:int = None):
    if p_gv == None:
      self.gv = 0
    else:
      self.gv = int(p_gv)

# Een float class
class ClsFlt():
  def __init__(self, p_gv:float = None):
    if p_gv == None:
      self.gv = 0.0
    else:
      self.gv = float(p_gv)

def main():
  def f_1():
    Obj1 = ClsStr()
    Obj2 = ClsInt()
    Obj3 = ClsFlt()
    print(Obj1.gv, Obj2.gv, Obj3.gv) # leeg 0 0.0
  f_1()
```

```
# De volgende code wordt niet uitgevoerd als je laadt als module
# wordt alleen uitgevoerd als je module als stand-alone programma uitvoert
# Volgens mij kun je elk functienaam hiervoor gebruiken.
if __name__ == '__main__':
  main()
```

De module staat hier: C:\Users\Fred\source\repos\P2\P2\module_byref.py en is onderdeel van een project (om testen makkelijker te maken). De code die de module aanroept, weer een ander project, staat hieronder. Hier zie je dat je de **`main()`** functie ook afzonderlijk kan aanroepen, wat niet verwonderlijk is. Het is best een gedoe om het pad naar de module te zetten. Je kunt het niet hard gecodeerd in het **`import`** statement zetten. (Voor Visual Studio heb je *Search Paths* onder de *Solutions Explorer* window).

```
import sys
sys.path.append("C:\\Users\\Fred\\source\\repos\\P2\\P2") # nodig om pad te zetten
import module_byref as Mdl

ObjStr = Mdl.ClsStr("Test")
ObjInt = Mdl.ClsInt(11)
ObjFlt = Mdl.ClsFlt(12)
print(ObjStr.gv, ObjInt.gv, ObjFlt.gv) # Test 11 12.0
Mdl.main()                             # Leeg 0 0.0
```

7.8 Anonieme functies

Python staat het toe om functies te maken die geen naam hebben. De functie kan aan een variabele toegekend worden, en je kunt die variabele dan gebruiken alsof het een functie is. Syntax:
`lambda <parameters>: <actie>`
`lambda` is een gereserveerd woord. **`<parameters>`** is een serie parameter namen, van elkaar gescheiden door komma's als er meer dan één is. **`<actie>`** is één commando. De anonieme functie heeft geen **`return`**, maar de waarde van **`<actie>`** wordt gebruikt als retourwaarde. Anonieme functies zijn niets anders dan reguliere functies, maar bestaan uit één enkele regel code. De volgende code maakt een anonieme functie die het kwadraat van de parameter berekent.

```
f = lambda x: x * x
g = 10*f(12)
print(g)
```

Hier een voorbeeld met een conditioneel statement, en je kunt diepe **`if-then-else`** constructies doen.

```
ld1 = lambda x: True if x>=1 else False
ld2 = lambda x, y: (True if y>=1 else False) if x>=1 else (True if y>=1 else False)
ld3 = lambda x, y: ('a' if y>=1 else 'b') if x>1 else ('c' if y>=1 else 'd')
a = ld1(1)     # True
b = ld2(1,0)   # False
c = ld3(0,0)   # 'd'
print(a, b, c)
```

8 Recursie

Recursie is een techniek waarbij een functie zichzelf aanroept. Iets algemener gesteld, refereert het aan een situatie waarin een functie andere functies aanroept op zo'n manier dat de uitvoering van de eerste functie nog steeds bezig is wanneer deze functie zelf nogmaals wordt aangeroepen (bijvoorbeeld, functie A() roept functie B() aan, die dan functie A() weer aanroept). Er zijn veel problemen waarvoor recursie een elegante oplossing biedt. Hier wat voorbeelden:

```
def faculteit(n):
  if n <= 1:
    return 1
  return n * faculteit(n-1)
print(faculteit(5))
```

```
def faculteit(n):
  if n <= 1:
    return 1
  print(n)               # om het effect te laten zien
  x = n*faculteit(n-1)
  print(n, x)            # om het effect te laten zien
  return x
print(faculteit(5))
```

De **print** statements laten het volgende zien:

De **print(n, x)** geeft het volgende:

```
2 2
3 6
4 24
5 120
```

De **print(n)** geeft het volgende:

```
5
4
3
2
```

Elke keer als de functie wordt aangeroepen wordt die apart in het geheugen geplaatst. Dus recursie is een geheugenslurper. In paragraaf 14.3 staat ook een recursie voorbeeld.

9 Strings

Hier zie je dat je de **+**, ***** en **in** operator kunt gebruiken op strings.

```
s1 = "appel"
s2 = " banaan "
print(s1)           # appel
print(s2)           # banaan
print(s1+s2)        # appel banaan
print(3*s1)         # appelappelappel
print(s2*3)         # banaan  banaan  banaan
print(2*s1 + 2*s2)  # appelappel banaan  banaan
for letter in s1:
  if letter in s2:
    print(s1, "en", s2, "bevatten beide de letter", letter)
```

9.1 Strings over meerdere regels

Gebruik het **** teken (backslash) om strings over meerdere regels te plaatsen. Dit werk ook voor code statements. Die kun je ze over meerder regels verdelen. De string wordt als één lange string gezien. Je kunt ook **'''** of **"""** gebruiken maar dan komt er steeds een *newline* karakter tussen.

```
x = 'String \
over \
meerdere \
regels. 1 regel.'
print(x)
```

Geeft:

```
String over meerdere regels. 1 regel.
```

```
x = '''String
over
meerdere
regels. 4 regels.'''
print(x)
```

Geeft:

```
String
over
meerdere
regels. 4 regels.
```

```
x = 'String \n\
over \n\
meerdere \n\
regels. 4 regels.'
print(x)
```

Geeft:

```
String
over
meerdere
regels. 4 regels
```

Je kunt het **\n** karakter er ook zelf inzetten, zie derde voorbeeld.

9.2 Speciale tekens

Er zijn een aantal speciale tekens of '*escape sequences*'. Deze tekens worden niet letterlijk genomen maar hebben een speciale betekenis.

****	Beschermen van het **** teken en print de backslash.
\n	*Newline* of nieuwe regel.
\t	Tabulatie of inspringen.
\x<nn>	Hexadecimaal getal. **\x20** is het decimale getal 32, en is een spatie.

Voorbeeldje:

```
x = '-\\\t\\\n\x20\x20\x20-'        -\      \
print(x)                              -
```

9.3 String manipulatie

Een string is een verzameling van tekens in een bepaalde volgorde. Een string heeft een karakter positie, een index die begin bij 0, en een lengte. Je kunt er allerlei bewerkingen op loslaten. Als een string een lengte van 5 karakters heeft dan is de eerste positie index 0 en de laatste 4. Zie voorbeelden hieronder.

```
X = 'qwerty'
for char in X:
  print(char, end='-')     # q-w-e-r-t-y-
print()
```

```
X = 'qwerty'
for i in range(0,len(X)):
  print(X[i], end='-')     # q-w-e-r-t-y-
print()
```

Onderstaande werkt ook, en dan begint het op een *array* te lijken. (Maar is het niet!)

```
print("qwerty"[3]) # r. je begint bij 0!
print(X[3])        # r. je begint bij 0!
```

Een index **moet** altijd een integer zijn. Je kunt ook negatieve indices gebruiken, die starten met -1 bij de laatste letter van de string, en die aflopen totdat de eerste letter van de string bereikt is.

```
X = '0123'
print(X[-4]) # 0
print(X[-3]) # 1
print(X[-2]) # 2
print(X[-1]) # 3
print(X[0])  # 0
```

Je kunt ook met indices een sub-string nemen maar omdat een string een klasse is, kun je ook de methodes (wat functies zijn) van de string klasse gebruiken. Er schijnt geen **substr** methode te zijn. Wat hieronder gebeurt heet ook wel '*slicing*', en is ook van toepassing op andere objecten. Zie paragraaf 9.8.

```
X = '123456QWERTY'
print(X[:])     #  123456QWERTY
print(X[:-1])   #  123456QWERT
print(X[::-1])  #  YTREWQ654321
print(X[::-4])  #  YW4 (zie uitleg)
print(X[:3])    #  123
print(X[::3])   #  14QR (zie uitleg)
print(X[2:4])   #  34
print(X.lower()) # 123456qwerty
```

Sub-strings kunnen ook een derde argument krijgen; de stapgrootte. Dit argument werkt equivalent aan het derde argument voor de **range()** functie. Syntax:
substrings = <string>[*<begin>*:*<einde>*:*<stap>*].
Gebruik een negatieve waarde voor de stapgrootte om een string om te draaien.
De ***begin*** waarde begint op positie 0 te tellen, maar de ***einde*** niet. Dat is dus ***einde-1***.

9.4 Strings zijn onveranderbaar

Strings zijn onveranderbaar ('*immutable*') en dus kun je ze niet wijzigen. Wil je er iets aan wijzigen dan moet je een nieuwe copy maken van de string. Dus **NewString = OldString**.

9.5 String methodes

De string is een klasse en dus zijn er methodes die een operatie uitvoeren op een string. Omdat strings onveranderbaar zijn retourneren ze een gewijzigde versie van de string. Ik ga hier niet in op alle methodes. De string methodes staan **hier**. Ook **integers** en **floats** hebben methodes,

9.6 Codering van tekens (ord() en chr())

De basis codering is ASCII. Dit is een 7-bits code, die 128 verschillende tekens kan weergeven. Unicode ondersteunt veel meer tekens. De meest bekende is UTF-8, die één byte gebruikt voor ieder van de ASCII tekens, maar meerdere bytes (2-4) voor alle andere tekens. Python ondersteunt UTF-8.
Hieronder wat voorbeelden om karakters op nummer te benaderen.

```
# de functies chr en ord werken op decimale tekens.
# \x<nn> en \u<nnnn> op hexadecimale getallen
print(chr(65), '\x41', '\u0041', chr(916), '\u0394') # A A A Δ Δ (griekse delta)
print(ord('A'),ord('Δ'))                             # 65 916

# Hier zie je het gehele griekse alfabat, grote letter en kleine letters
a = "\u0391"
for i in range(57):
  if i <= 24 or i >= 32:
    if i == 32: print()
    print(chr(ord(a)+i), end=" ")
print()
```

De vier coderingsfuncties zijn:

ord()	Ordinal functie. Geeft het decimale coderingsnummer van een karakter. ASCII en Unicode.
chr()	Character functie. Geeft het karakter van het decimale coderingsnummer. ASCII en Unicode.
\x<nn>	Alleen voor ASCII. Geeft het karakter op basis van het hexadecimale coderingsnummer.
\u<nnnn>	Unicode en ASCII. Geeft het karakter op basis van het hexadecimale coderingsnummer.

De volgende code geeft alle karakters tussen 0 en 511. Om alles mooi uit te lijnen, 50 tekens per regel, moeten een aantal controle karakters afgevlagd worden.

```
# 0=null, 7=bell, 8=backspace, 9=horizontal tab, 10=line feed, 13=carriage return
for i in range(512):
  if i not in (7,8,9,10,13):
    print(chr(i), end=' ')
  else:
    print(' ', end=' ')
  if  ((i+1) % 50 == 0):print()
print()
```

9.7 strings met parameters

Hier een viertal voorbeelden om variabelen als parameters door te geven om een string te maken.

```
def f_exist(pTable, pSchema='public'):
  sql_str = """select table_schema, table_name, count(*) as cnt
               from {0}
               where table_schema = {1} and table_name = {2}
               group by 1, 2""".format('information_schema.tables',
                                       f_string(pSchema),
                                       f_string(pTable))

  sql_str = """select table_schema, table_name, count(*) as cnt
               from %s
               where table_schema = %s and table_name = %s
               group by 1, 2""" %('information_schema.tables',
                                       f_string(pSchema),
                                       f_string(pTable))

  sql_str = """select table_schema, table_name, count(*) as cnt
               from {var1}
               where table_schema = {var2} and table_name = {var3}
               group by 1, 2""".format(var1='information_schema.tables',
                                       var2=f_string(pSchema),
                                       var3=f_string(pTable))

  sql_str = """select table_schema, table_name, count(*) as cnt
               from %(var1)s
               where table_schema = %(var2)s and table_name = %(var3)s
               group by 1, 2""" %{"var1": 'information_schema.tables',
                                       "var2": f_string(pSchema),
                                       "var3": f_string(pTable)}
  print(sql_str)
```

9.8 Slicing

'*Slicing*' is een techniek om een object in stukken op te hakken. Het werkt niet alleen op strings maar op anddere objecten, zoals '*Tuples*' en '*Lists*'.
subObject = <Object>[*<begin>*:*<einde>*:*<stap>*].
Er is ook een apart functie voor, **slice(begin, einde, stap)**, die je op twee manieren kunt gebruiken. Hieronder een Tuple en List voorbeeld.

```
T = (1,2,3,4,5,6,7,8)
sO = slice(3,5)              # slice object
sT1 = T[3:5]
sT2 = T[slice(3,5)]
sT3 = T[sO]
print(sT1, sT2, sT3, sep="\n") # 3 x (4, 5)
```

```
L = [1,2,3,4,5,6,7,8]
sO = slice(3,5)
sL1 = L[3:5]
sL2 = L[slice(3,5)]
sL3 = L[sO]
print(sL1, sL2, sL3, sep="\n") # 3 x [4, 5]
```

9.9 ASCII tabel

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

10 Tuples

- Een '*tuple*' is een groep van één of meer waardes die als een geheel behandeld worden.
- Tuples zijn onveranderbaar; je kunt geen nieuwe waarde toekennen aan een element van een tuple.
- Tuples worden niet vaak gebruikt, behalve als retourwaardes van functies.
- Een tuple bestaat uit een aantal waardes die van elkaar gescheiden zijn met komma's.
- Meestal worden tuples geschreven met haakjes eromheen, maar dat is niet verplicht.
- Je kunt in een tuple verschillende data types mixen.

Om elementen in een tuple te benaderen gebruik je indices.

```
T1 = ("een", 3, 1.4, ("twee", 5))     # vier waardes, waarvan eentje een tuple
T2 = T1[3]
print(type(T2), type(T1[3]))          # zelfde output: <class 'tuple'>
print(len(T1))                        # aantal elementen: 4
print(T1[3],T2[1])                    # ('twee', 5) 5
print(T1[3][1])                       # 5
```

Hier een voorbeeldje om een tuple volledig door te lopen. Doen we recursief.

```
T1 = ("een", 3, ('a', 'b'), 1.4, ("twee", 5, (1,2,3),'x'), 991, 992)
def func_show_tuple_elements(pTuple,level):
  for element in pTuple:
    # test of element een tuple is. Het lukt mij niet om op
    # de string <class 'tuple'> te testen
    if (isinstance(element, tuple) == False):
      print(level, element, sep=": ")
    else:
      func_show_tuple_elements(element,level+1)

func_show_tuple_elements(T1,1)
```

```
1: een
1: 3
2: a
2: b
1: 1.4
2: twee
2: 5
3: 1
3: 2
3: 3
2: x
1: 991
1: 992
```

De waarde *level* geeft aan hoe diep je in de tuple zit.

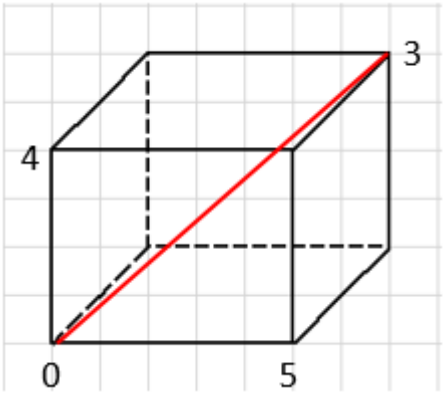
Nog wat wetenswaardigheden:

- Gebruik **max()**, **min()**, **sum()** om aggregaat operaties te doen op tuples die alleen bestaat uit getallen.
```
T1 = (327, 0.1, 419, 101, 667, 925, 225)
print(max(T1), min(T1), sum(T1), (sum(T1)/len(T1))) # de laatste is het gemiddelde
```
- Testen of een element onderdeel van een tuple is met behulp van de **in** operator.
```
print(101 in T1, 201 in T1)                    # True False
```
- Om een tuple met één element te maken doe je dit:
```
T1 = ("Waarde",)       # Dit is een tuple met 1 element
T2 = "Waarde",         # Dit is een tuple met 1 element
T3 = ("Waarde")        # Dit is een string
print(type(T1), type(T2), type(T3), T3)
```
- Tuples hebben ook indices, net zoals strings en werken ook zo, maar hebben geen stapgrootte.
```
T1 = ("a", "b", "c", "d", "e")
print(T1[:2], T1[0:2])          # hetzelfde; ('a', 'b')
```

Vergelijk tuples met elkaar met behulp van de reguliere vergelijkingsoperatoren, **==, !=, >, <** etc. Er zit een vergelijkingslogica achter, maar die laat ik voor wat het is.

10.1 Toepassing voorbeeld met tuples

Tuples zijn handig om als coördinaten te gebruiken. Neem een driedimensionaal vierkant met afmetingen (5,4,3) x-as-punt=5, y-as-punt=4 en z-as-punt=3, en dat je de lengte van de rode lijn wilt berekenen.



De volgende, algemene code, doet het. Je hebt twee driedimensionale coördinaten: (0,0,0) en (5,4,3). De wiskunde die zegt dat het steeds de wortel van de kwadraten is, is dan een gegeven.

```
from math import sqrt
def afstand(p1,p2,totaal=0):
  for i in range(len(p1)):
    totaal += (p1[i] - p2[i]) **2
  return sqrt(totaal)
# 1-dimensionale ruimte
punt1 = (0,)
punt2 = (5,)
print("1D:", punt1, "en", punt2, "is", afstand(punt1, punt2))
# 2-dimensionale ruimte
punt1 = (0,0)
punt2 = (5,4)
print("2D:", punt1, "en", punt2, "is", afstand(punt1, punt2))
# 3-dimensionale ruimte
punt1 = (0,0,0)
punt2 = (5,4,3)
print("3D:", punt1, "en", punt2, "is", afstand(punt1, punt2))
```

10.2 Enumerate (=opsomming) tuples/lists

Om de index te weten te komen van een list item als je daar doorheen loopt, gebruikt je de **enumerate()** functie. Hieronder een voorbeeldje. Het is een *List* met *Tuples*.

```
L = [('aap', 123), ('noot',1,2,3), ('mies', 'noot')]
for i, t in enumerate(L): # index in de lijst, en de tuple
  print('index: ', i, t)
```

11 Lists

- Een '*list*' of '*lijst*' is net als een tuple een geordende verzameling van data items.
- Een tuple is niet veranderbaar, een list wel. Dat heet '*mutable*'.
- Met *geordend* wordt niet gesorteerd bedoelt. Het zegt dat elk element afzonderlijk benaderbaar is.
- Bij een lists staan de list elementen tussen vierkante haken (**[]**).
- Via een **for** loop kun je de elementen van de list doorlopen.
- Je mag data types in een list mixen.
- De **len()** functie geeft het aantal elementen in een list.
- Je kunt de functies **max()**, **min()** en **sum()** gebruiken op een list.
- Testen of een element voorkomt in een list kan via **in** of **not in** operator.

Wat voorbeeld code.

```
L1 = [327, 0.1, 419, 101, 667, 925, 225]
print(max(L1), min(L1), sum(L1)/len(L1))     # de laatste is het gemiddelde
L1 = ["een", 3, ['a', 'b'], 1.4, ["twee", 5, [1,2,3], 'x'], 991, 992]
L2 = L1[2]
print(type(L2), type(L1[2]))         # zelfde output: <class 'list'>
print(len(L1))                       # aantal elementen: 7
print(L1[3], L2[1])                  # 1,4 b
print(L1[2][1])                      # b
print(L1[:2])                        # ['een', 3]
```

De functie hieronder laat alle list elementen zien. Het is bijna identiek aan de tuple code.

```
def func_show_list_elements(pList,level):
  for element in pList:
    # test of element een tuple is. Het lukt mij niet om op
    # de string <class 'list'> te testen
    if (isinstance(element, list) == False):
      print(level, element, sep=": ")
    else:
      func_show_list_elements(element,level+1)

func_show_list_elements(L1,1)
```

```
1: een
1: 3
2: a
2: b
1: 1.4
2: twee
2: 5
3: 1
3: 2
3: 3
2: x
1: 991
1: 992
```

11.1 List operaties

Omdat lists veranderbaar zijn, mag je de inhoud van een list wijzigen. We gebruiken deze list als voorbeeld:

```
L1 = ["een", 3, ['a', 'b'], 1.4, ["twee", 5, [1,2,3], 'x'], 991, 992]

L1[0] = 1                  # hier wordt het eerste element een 1
print(L1)
L1[1] = ['X', 1, 2, [1,2]] # hier wordt het tweede element een geneste-list met een geneste-list
print(L1)
L1[1:2] = []               # hier verwijder je het tweede element compleet
L1[1] = []                 # Dit doet het ook, maar in de print statement zie je dit []
print(L1)
L1[0:len(L1)] = []         # En hier maak je de hele list leeg
print(L1)
```

Lists ondersteunen de **+** en ***** operatoren, en ze werken net zoals bij strings.

```
L2 = [-1, -2, -3]
L3 = L1+L2         # optellen, toevoegen van L2 aan L1
print(L3)
L1 = 5*L1          # hier dupliceer je L1 5x
print(L1)
L4 = 100*[-1]      # hier een lijst met 100 elementen allemaal -1
```

11.2 List methodes

Klassen hebben methodes (functies) om operaties op de data uit te voeren. Hier een aantal list methodes.

append()

Toevoegen van een element aan het einde van een lijst. Kan maar één argument bevatten.

```
L2 = [9, 8, 7]
L1.append([-1,-2, -3, ['a', 'b']])  # append met sub-list
L1.append(123)                      # append met getal 123
L1.append(L2)                       # L2 wordt toegevoegd als sub-list!
```

Equivalent met **<list>[len(<list>):] = [<element>]**, of **<list> += [<element>]**

extend()

Toevoegen van een element aan het einde van een lijst. Kan maar één argument bevatten. Het verschil met **append()** zit hem in toevoegen van **L2**. Bij de **append()** is dat een sub-list, bij de **extend()** een lijst van elementen.

```
L2 = [9, 8, 7]
L1.extend(123)    # !WERKT NIET!
L1.extend([123])  # werkt wel
L1.extend(L2)     # extended met 3 elementen
```

Equivalent met **<list>[len(<list>):] = <addlist>**

copy()

Een '*ondiepe*' ('*shallow*') copy van een lijst. Kopieert een lijst in een nieuwe lijst. (De assignment operator maakt een nieuwe referentie aan die naar hetzelfde object wijst). Echter, een ondiepe kopie kopieert alleen de elementen van het eerste niveau. De geneste-lists blijven wel dezelfde pointer houden.

```
L2 = L1                                     # L2 wijst naar L1 (pointer)
L3 = L1.copy()                              # een nieuwe onafhankelijk variabele
L2[0] = 3                                   # L1[0] is ook 3 geworden
L3[0] = -1                                  # Alleen L3 wijzigt
L3[2][0] = 1                                # Maar nu wijzigt L1, L2 en L3
print(L1, L2, L3, sep='\n')
print(id(L1), id(L2), id(L3), sep='\n')     # id() geeft variabele id nummer
print(id(L1[2]), id(L2[2]), id(L3[2]))      # geneste-list zijn hetzelde
```

Functie **id()** geeft het geheugenadres van een variabele. Zijn die gelijk dan heb je te maken met dezelfde variabele die toevallig een andere naam heeft. Echter, element 2 van **L1-L3** wijst wel weer naar hetzelfde geheugenadres, en is dus gedeeld. Gebruik functie **deepcopy()** om echt alles te kopiëren.

deepcopy()

Dit is geen methode maar een kopieer functie die generiek is en op meer objecten werkt dan lists alleen. Deze functie kopieert een list in zijn geheel, ongeacht hoeveel niveaus diep. Dit heet dan een '*diepe*' kopie.

```
from copy import deepcopy             # deepcopy werkt ook op andere objecten dan lists
L1 = ["een", 3, ['a', 'b'], 1.4, ["twee", 5, [1, 2, 3], 'x'], 991, 992]
L2 = deepcopy(L1)
L2[2][0] = 1                          # Alleen L2 wijzigt!
print(L1, L2, sep='\n')
print(id(L1), id(L2), sep='\n')       # verschillende adressen
print(id(L1[2]), id(L2[2]), sep='\n') # verschillende adressen
```

insert()

Toevoegen van een element op een specifieke positie. Eerste argument is de positie, het tweede het element.

```
L2 = [9, 8, 7]
L1.insert(0, L2)     # L2 wordt ingevoegd als sub-list
L1.insert(0, 7)      # zo kan je het als element doen
L1.insert(0, 8)
L1.insert(0, 9)
```

Equivalent met **<list>[<i>:<i>] = [<element>]**

remove()

Verwijder element uit een list, waarbij de element naam gebruikt wordt. Als het element vaker voorkomt wordt de eerste instantie (die met de laagste index) verwijderd.

```
L1.remove(["twee", 5, [1,2,3], 'x'])    # verwijder geneste-list
```

pop()
Verwijdert één element uit een list, waarbij de index gebruikt wordt. Index start met 0! De **pop()** methode retourneert het element wat je verwijderd.

```
L1.pop(4)        # verwijder het vijfde element
print(L1.pop(4)) # laat het vijfde element zien wat je verwijdert
```

del
del is een gereserveerd woord dat een list element verwijderd via de index. Syntax:

```
del <list>[<index>]
del <list>[<index1>:<index2>]
del L1[4]   # verwijder het vijfde element
del L1[1:4] # verwijder het tweede, derde en vierde element
```

index()
Retourneert de index van de eerste instantie in een list van het element dat als argument aan de methode is meegegeven. Begint te tellen vanaf 0!

```
i = L1.index(["twee", 5, [1,2,3], 'x'])    # het vijfde element, maar i=4!
```

count()
Geeft aan hoe vaak het element dat als argument is meegegeven voorkomt in de list. Doorzoekt niet de geneste-lists. Dus alleen het hoogste niveau.

```
x = L1.count("een")         # x en y zijn hetzelfde
y = L1.count(L1[0])
```

__len__()
De methode **__len__()** geeft de lengte, het aantal elementen, van een list. Is hetzelfde als **len(L1)**.

```
print(L1.__len__())     # zelfde als len(L1)
```

sort()
Sorteert de elementen van de list, van laag naar hoog. Als de elementen strings zijn, betreft het een alfabetische sortering. Als de elementen getallen zijn, betreft het een numerieke sortering. Als de elementen gemixt zijn, volgt een runtime error, tenzij bepaalde extra argumenten zijn meegegeven. Syntax:
sort(List, *key*=None, *reverse*=None)
key: Geeft extra sorteermogelijkheden aan. Bijvoorbeeld; **key=str.lower**. Dan wordt elk element als lowercase gesorteerd. Je kunt elke data type klasse gebruiken.
Reverse: Als je omgekeerd (descending) wilt sorteren.
Mijn L1 voorbeeld werkt niet goed, is gemixt. Vandaar een ander voorbeeld.

```
L1 = [1, 2, 6, 5, 88, 4, 4, 4, -4, -1]
L1.sort(key=int.__abs__) # getallen worden absoluut gesorteerd
L1.sort()
print(L1)
```

reverse()
Zet de elementen van de list in omgekeerde volgorde. Maar is wel iets geks mee aan de hand.

```
L1.reverse()
print(L1)
print(L1.reverse()) # dit werkt niet, geeft None
L1.reverse()        # en dan werkt terugzetten ineens ook niet meer?
print(L1)
```

Let op: Als je niet goed op let, voeg je heel snel een geneste list toe, in plaats van een element.

11.3 Key in sort()

De **key()** in **sort()** is nogal bewerkelijk. Je kunt ook eigen functies gebruiken. Twee voorbeelden.
Een list van strings wordt gesorteerd; als eerste op string lengte (korte strings vóór lange strings), en alleen bij gelijke lengte op alfabetische volgorde:

```
def len_alfabetisch(item):
  return len(item), item   # je sorteert eerst op lengte, en dan pas string
fruitlist = ["appel", "aardbei", "banaan", "framboos", "kers", "banaan", "doerian", "mango"]
fruitlist.sort(key=len_alfabetisch)
print(fruitlist)
```

Maar is hetzelfde (in dit geval dan) als:

```
fruitlist = ["appel", "aardbei", "banaan", "framboos", "kers", "banaan", "doerian", "mango"]
fruitlist.sort(key=str.__len__)
print(fruitlist)
```

Hier dan een voorbeeld waar strings en integers in zitten.

```
def mix_key_1(item):
  if isinstance(item, str):
    return 1, item
  return 0, item
def mix_key_2(item):
  if isinstance(item, str):
    return 1, len(item), item
  return 0, item
L1 = ["X", 0, "aardbei", 5, "banaan", 2, "Y", 9, "kers", "banaan", 7, 7, 6, "Z"]
L1.sort(key=mix_key_1)
print(L1)
L1.sort(key=mix_key_2)
print(L1)
```

Hoe de key precies werkt, begrijp ik niet helemaal, want elke element gaat door de functie. In ieder geval, hieronder de code om een list met geneste-lists te sorteren waarbij de inhoud van de geneste-list zelf niet gesorteerd wordt. De sorteer volgorde, dus ook waar de geneste-list komt te staan is afhankelijk of je op integer of string sorteert. Het maakt ook nogal wat uit als het eerste element in de list een geneste-list is.

```
def mix_key_3(item):
  # je sorteert de gehele list op eerste geneste-list in de list
  # en op basis daarvan wordt de geneste-list in de sorteerketen geplaatst
  # de geneste-list zelf wordt niet gesorteerd
  # Dus, dat is maar net wat er in de lists staat.
  if isinstance(item,(list)):
    # hier sorteer je dus op een getal en komt de eerste geneste_list vooraan te staan
    # dit is namelijk de eerste geneste list: ['x','a','b',1,2] en het 4e element is 1, een getal
    # zou de eerste geneste-list uit zeg twee elementen hebben bestaan dan krijg je foutmelding
    return mix_key_3(item[3])
  else:
    if isinstance(item, str):
      return 1, item    # blijkbaar is 1 een string?
  return 0, item        # en nul een integer?

L1 = ["een",3,['x','a','b',1,2],1.4,["twee",5, [1,'X',2,3],'x'],991,'a','x','b',992]
print(L1)
L1.sort(key=mix_key_3)
print(L1)
```

11.4 `is` operator

De **`is`** operator wordt gebruikt om de identiteiten van twee variabelen met elkaar te vergelijken. Dit is totaal iets anders dan de **==** operator die de inhoud vergelijkt. Het code voorbeeld hieronder maakt het duidelijk.

```
L1 = [1, 2, 3, ['a']]
L2 = L1
L3 = L1.copy()              # een ondiep copy volstaat hier om het effect te laten zien
if (L1 == L2): print(True)  # True: inhoud is hetzelfde
else: print (False)
if (L1 is L2): print(True)  # True: objecten zijn hetzelfde
else: print(False)
if (L1 is L3): print(True)
else: print(False)          # False: objecten zijn niet hetzelfde
```

11.5 Lists als functie argumenten

Als je een list aan een functie meegeeft als argument, dan is dit een zogeheten '*pass by reference*'. Dus de adres pointer wordt doorgegeven. Dat betekent dat de functie de inhoud van de list kan wijzigen. Structuur data typen worden vaak als *pass by reference* worden doorgegeven, alle andere als '*pass by value*'.

```
def func_list(pList):
  pList[0] = 999
L1 = [0]
func_list(L1)
print(L1)     # [999]
```

11.6 Geneste lists

Geneste-lists zijn elementen van een list die zelf ook weer een list zijn. Op deze manier kun je een matrix in je programma maken. Hieronder het boter-kaas-eieren bord.

```
def toon_bord(b):
  for rij in range(3):
    for kol in range(3):
      print(b[rij][kol], end=' ')
    print()
bord = [["-", "-", "-"], ["-", "-", "-"], ["-", "-", "-"]]
bord[1][1] = "X"
bord[0][2] = "O"
toon_bord(bord)
```

11.7 List casting

Van een collectie van elementen kun je een list maken door de type casting functie **`list()`** te gebruiken. Handig om een list van integers te maken met behulp van de range functie. Code hieronder doet hetzelfde. Een regel code versus drie regels code. **`<class 'list'> en [0,1,2,3,4,5,6,7,8,9] en 10`**

```
L1 = []
print(L1)
for i in range(10): L1.append(i)
print(type(L1), '\n', L1, L1.__len__())
```

```
L1 = list(range(10))
print(type(L1), '\n', L1, L1.__len__())
```

Je kunt een string ook casten als list. Dan krijg je een list waarin ieder teken in de string een element is.

```
S1 = 'QWERTY'
L1 = list(S1)
print(type(S1), type(L1), L1)        # ['Q','W','E','R','T','Y']
```

11.8 List comprehensions

Een '*list comprehension*' is een techniek waarbij je op een compacte manier lists maakt. Het is moeilijk leesbare code. Een *list comprehension* bestaat uit een expressie tussen vierkante haken, wat weer bestaat uit een **`for`** statement, gevolgd door nul of meer **`for`** en/of **`if`** statements. Het resultaat is een list die de elementen bevat die het resultaat zijn van de evaluatie van de combinatie van de **`for`** en **`if`** statements.

Voorbeeld; een list die de kwadraten van de getallen 1 t/m 25 bevat.

```
# De gewone manier
lc = []
for i in range(1,26): lc.append(i**2)
print(lc)
```

```
# De list comprehension manier
lc= [x**2 for x in range(1,26)]
print(lc)
```

Hier voorbeeldje waarbij je geen kwadraten wilt opnemen die eindigen op een 5.

```
lc = [x**2 for x in range(1,26) if x%10 != 5]
print(lc)
```

Hier maak je een list van tuples van drie integers, waarbij de drie integers alle drie verschillend zijn:

```
triolist = [(x,y,z) for x in range(1,5)
                    for y in range(1,5)
                    for z in range(1,5)
            if x != y if x != z if y != z
           ]
print(triolist)
```

12 Dictionaries

- Strings, tuples en lists zijn geordende datastructuren, wat inhoudt dat ze via indices benaderbaar zijn.
- Python biedt '***dictionaries***' als een manier om **ongeordende** data te structureren.
- Om een element te vinden, moet je de '*key*' ('*sleutel*') van het element kennen.
- Een dictionary is een verzameling van '*keys*' met geassocieerde waardes.
- Ieder **onveranderbaar** data type (bijv. string, int, float) mag gebruikt worden als key. Strings worden veel gebruikt. Maar ook een tuple is een onveranderbaar data type.
- Dictionaries zelf zijn veranderbaar.
- Dictionaries maak je middels accolades `{}`.
- Een lege dictionary maak je door een assignment aan een variabele te doen met `{}`.
- Je kunt een dictionary met inhoud maken met ieder element dat je erin wilt.
- De volgorde van een dictionary is willekeuring. Het is immer ongeordend.
- Je kunt geen '*geneste-dictionary*' maken door een key-bereik te definiëren. Maar de value van een key kan wel weer complex zijn, bijvoorbeeld een list.
- Je kunt een dictionary niet sorteren of inverteren.
- Dictionaries zijn een veranderbaar data type. Je geeft dus een pointer door.

Syntax van een dictionary is als volgt, waarbij de **`key:value`** paartjes komma gescheiden zijn.
`D = {<key>:<value>,}`
Om een waarde te vinden die hoort bij een specifieke sleutel, gebruik je dezelfde syntax als voor een list, waarbij de index dan de key is. Voorbeeldje, links de code, rechts de uitvoer.

```
D = {"appel":90, "banaan":80, "kers":150}
print(D)
for k in D:
  print("{}\t: {}".format(k, D[k]))
```

```
appel   : 90
banaan  : 80
kers    : 150
```

Je kunt het zien als voorraadbeheer van een fruithandelaar. Toevoegen van nieuw fruit doe je als volgt, waarbij key waardes overschrijfbaar zijn:

```
D["peer"] = 99
D["peer"] = 45  # hier zie je dat keys waardes elkaar overschrijven.
print(D)        # {'appel': 90, 'banaan': 80, 'kers': 150, 'peer': 45}
```

Verwijderen van een key-value paartje doe je met het **`del`** statement:

```
del D["appel"]
```

12.1 Dictionary methodes (een sub-set)

`clear()`
Legen van een dictionary.

`copy() en deepcopy()`
De assignment operator maakt een nieuwe variabele aan die naar hetzelfde object wijst. De **`copy ()`** methode maakt een ondiepe copy van een dictionary. De functie **`deepcopy()`** maakt een volledige copy van een dictionary. Een dictionary heeft dan wel geen geneste-dictionaries, maar kan wel lists als value hebben.

```
from copy import deepcopy   # deepcopy werkt ook op andere objecten dan dictionaries
D1 = {"appel":90, "banaan":80, "kers":150, "peer":0}
D2 = D1
D3 = D1.copy()
D4 = deepcopy(D1)
print(id(D1), id(D2), id(D3), id(D4), id(D1["banaan"]), sep="\n")
print(D1, D2, D3, D4, sep="\n")
```

Hier zie je dus dat een element een eigen geheugenadres heeft

`keys(), values() en items()`
`keys()` levert een iterator die alle keys van de dictionary genereert.
`values()` levert een iterator die alle waardes van een dictionary genereert.
`items()` levert een iterator die 2-tuples genereert die alle keys en waardes van de dictionary bevatten.
Je gebruikt iterators met name in **`for`** loops (maar je kunt ze ook gebruiken als argumenten voor de functies **`max()`**, **`min()`** en **`sum()`**). Er worden iterators geretourneerd (en niet lists) omdat die sneller werken dan lists. Wil je er een list van maken dan moet je casten.

```
D_k = D1.keys()
D_v = D1.values()
D_i = D1.items()
print(D_k)              # dict_keys(['appel', 'banaan', 'kers', 'peer'])
print(D_v)              # dict_values([90, 80, 150, 0])
print(D_i)              # dict_items([('appel', 90), ('banaan', 80), ('kers', 150), ('peer', 0)])
L_k = list(D_k)
L_v = list(D_v)
L_i = list(D_i)
print(L_k)                  # ['appel', 'banaan', 'kers', 'peer'])
print(L_v)                  # [90, 80, 150, 0])
print(L_i)                  # [('appel', 90), ('banaan', 80), ('kers', 150), ('peer', 0)]
print(type(D_i), type(L_i))   # <class 'dict_items'> <class 'list'>
```

get()

Haal een waarde uit de dictionary op basis van de key. Als de key niet bestaat krijg je **None** of een zelf gedefinieerde default waarde.

```
print(D1.get("perzik"))                  # None
print(D1.get("perzik", "Uitverkocht"))   # Uitverkocht
print(D1.get("appel"))                   # 90
```

12.2 Keys

Ieder onveranderbaar data type mag een dictionary key zijn. Tuples zijn dat ook. Een tuple kun je als key gebruiken om coördinaten informatie op te slaan. Hieronder een variant wat ook in paragraaf 10.1 staat.

```
from math import sqrt
def distance(p1,p2,total=0):
  for i in range(len(p1)-1):
    total += (p1[i] - p2[i]) **2
  return sqrt(total), p1[3], p2[3] # geeft een tuple terug

D = {(0,0,0):"Kuinre", (5,4,3):"Emmeloord"}
i = 0
for k in D:
  i+=1
  if (i == 1):
    p1 = list(k)
    p1.append(D.get(k))
  else:
    p2 = list(k)
    p2.append(D.get(k))
T = distance(p1,p2)
print("Afstand\t\t:", T[0], "\nPositie 1\t:", T[1], "\nPositie 2\t:", T[2])
```

12.3 Opslaan van complexe waardes

De value in een key-value paar kan elk willekeurig data type zijn. De key kan bijvoorbeeld een tuple zijn, de value een list. Hieronder een voorbeeldje.

```
D = {(0,0,0):["Kuinre", 800, "Overijssel"], (5,4,3):["Emmeloord", 15000, "Flevoland"]}
i = 0
for k in D:
  i+=1
  if (i == 1):
    p1 = list(k)
    p1.append(D.get(k))
  else:
    p2 = list(k)
    p2.append(D.get(k))
T = distance(p1,p2)
print("Afstand\t\t:"
      ,T[0]
      ,"\nPositie 1\t:"
        ,"Plaats\t:", T[1][0]
        ,"\n\t\t  Inwoners\t:", T[1][1]
        ,"\n\t\t  Provincie\t:", T[1][2]
      ,"\nPositie 2\t:"
        ,"Plaats\t:", T[2][0]
        ,"\n\t\t  Inwoners\t:", T[2][1]
        ,"\n\t\t  Provincie\t:", T[2][2]
```

```
    )
Afstand         : 7.0710678118654755
Positie 1       : Plaats        : Kuinre
                  Inwoners      : 800
                  Provincie     : Overijssel
Positie 2       : Plaats        : Emmeloord
                  Inwoners      : 15000
                  Provincie     : Flevoland
```

12.4 Snelheid

Lists en dictionaries zijn de twee meest-gebruikte datastructuren in Python. Om een keuze te maken:

- Lists nemen minder geheugen in beslag dan een dictionary.
- Een list is snel als je de elementen via hun index kunt benaderen.
- Een dictionary is de betere keuzes als de enige manier om iets te zoeken de elementen te scannen is.
- Dictionaries slaan de keys separaat op in een '*hash*' tabel.

13 Sets

- Sets zijn ongeordende datastructuren die alleen unieke elementen kunnen bevatten. Dit komt uit de relatietheorie en wordt o.a. gebruikt bij SQL-servers; elke rij in een tabel is uniek.
- Wiskundig gezien betreft het een verzameling van unieke elementen.
- De enige manier om elementen van een set te benaderen is via een **`for`** loop, of door met de **`in`** operator te testen of een element in de set bestaat.
- Dictionaries worden gebruikt om sets te implementeren. De elementen van een set zijn de keys van een dictionary. Dus alleen **onveranderbare** data types kunnen in een set worden opgeslagen. Een set zelf is wel veranderbaar.

Om een lege set te maken gebruik je de **`set()`** functie zonder argumenten. Je initialiseert een set door elementen als argument op te geven. Geef je een string als argument, dan wordt elk karakter een set element.

```
S1 = set()
print(type(S1))                 # <class 'set'>
S2 = set({1, 2, 3})
print(type(S2), S2)             # <class 'set'> {1, 2, 3}
S3 = {1,2,3}                    # zo kun je ook een set definieren
print(type(S3), S3)             # <class 'set'> {1, 2, 3}
S4 = set('QWERTY')
print(S4)                       # {'Y', 'E', 'T', 'R', 'W', 'Q'}
print(len(S4), S4.__len__())    # aantal elementen; beiden 6
```

Gebruik de **`for`** loop om een set te doorlopen. De variabele van de **`for`** loop krijgt toegang tot alle elementen van de set. De volgorde is niet gegarandeerd. Om te sorteren moet je casten naar list.

```
for e in S2: print(type(e), e) # integer en de waarde
for e in S4: print(type(e), e) # string en de waarde
```

13.1 Set methodes (een sub-set)

Een set is zowel een subset als een superset van zichzelf.

`add() en update()`

De **`add()`** methode voegt één nieuw element toe. Je kunt een tuple of string als argument gebruiken. Voor een string wordt elk karakter een nieuw element.
De **`update()`** methode voegt meerdere elementen toe.
Omdat een set alleen unieke elementen kan bevatten, worden duplicaten genegeerd.

```
S = {1, 2, 3}
for i in range(10): S.add(i)   # duplicaten worden overgeslagen
print(S)                       # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
S.update({10, 12, 13})
print(S)                       # {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 13}
```

remove(), discard() en clear()

De **remove()** methode verwijdert één element waarbij een fout wordt gegeven als het element niet bestaat.
De **discard()** methode verwijdert ook een element maar negeert niet bestaande elementen.
De **clear()** methode verwijdert alle elementen van de set in één keer.

```
S.remove(1)
S.discard(1)  # geen foutmelding
print(S)
S.clear()
print(S)      # set()
```

pop()

De **pop()** methode verwijdert één **willekeurig** element uit de set en retourneert het.

```
S = set('QWERTY')
RetVal = S.pop()         # T
print(S, '\n', RetVal)   # {'Q', 'E', 'W', 'Y', 'R'}
```

copy() en deepcopy()

De **copy()** methode doet een ondiepe copy van een set. Gebruik de **deepcopy()** functie om alles te kopiëren. Werkt verder net zoals bij lists.

union()

De vereniging ('*union*') van twee sets. Is een optelling. Maar **+** en ***** operator zijn niet gedefinieerd voor sets. Je kunt wel de **|** ('*pipeline*') gebruiken. Allen de unieke waardes worden toegevoegd.

```
S1 = set('QWERTY')
S2 = set('qwertyQWERTY')
S3 = S1.union(S2)
S4 = S1|S2
print(S1, S2, S3, S4, sep='\n') # S3 gelijk aan S4
```

intersection()

De doorsnede ('*intersection*') van twee sets is een set die alleen de elementen bevat die beide sets gemeen hebben. Je kunt ook de **&** ('*ampersand*') gebruiken. In SQL heet dit '*intersect*'

```
S1 = set('QWERTY1234')
S2 = set('12qwertyQWERTY')
S3 = S1.intersection(S2)
S4 = S1&S2
print(S1, S2, S3, S4,s ep='\n') # S3 gelijk aan S4
```

difference()

Het verschil ('*difference*') van twee sets is een set die de elementen van de eerste set bevat, met daaruit verwijderd de elementen die de eerste set gemeen heeft met de tweede set. In SQL heet dit '*except*' of '*minus*'. Je kunt ook de **–** operator gebruiken.

```
S1 = set('QWERTY1234')
S2 = set('12qwertyQWERTY')
S3 = S1.difference()
S4 = S1-S2
print(S1, S2, S3, S4, sep='\n') # S3 gelijk aan S4
```

isdisjoint(), issubset(), en issuperset()

isdisjoint() retourneert **True** als de twee sets geen elementen gemeen hebben.
issubset() retourneert **True** als alle elementen van de eerste set ook voorkomen in de tweede set.
issuperset() retourneert **True** als alle elementen van de tweede set ook voorkomen in de eerste set.

```
S1 = set('QWERTY1234')
S2 = set('12qwertyQWERTY')
b1 = S1.isdisjoint(S2)
b2 = S1.issubset(S2)
b3 = S1.issuperset(S2)
print(S1, S2, b1, b2, b3, sep='\n') # In dit geval is alles False
```

13.2 Frozensets

De elementen van een *frozenset* kunnen niet veranderd worden. Je kunt geen elementen toevoegen of veranderen. Je maakt een *frozenset* met de **frozenset()** functie.

```
S = frozenset('QWERTY')
S.add('X')              # kan niet, methode bestaat ook niet eens!
```

14 Besturingssysteem

Gebruik de **os** ('*operating system*' of '*besturingssysteem*') module voor interacties met het besturingssysteem. In dit geval gaan we dieper in op Windows. Stel het volgende:

Python interpreter	:	C:\Program Files\Python36\Python.exe
Python code	:	C:\Users\Fred\source\repos\P1\P1\P1.py
PATH variabele uitgebreid met	:	C:\Program Files\Python36

Als je in directory C:\Users\Fred\source\repos\P1\P1\ staat dan executeer je P1.py als volgt:
`Python P1.py`
Deze methode werkt overal waar je staat:
`"C:\Program Files\Python36\Python.exe" "C:\Users\Fred\source\repos\P1\P1\P1.py"`
Om de een of andere redenen werkt het niet binnen de Windows Powershell.

14.1 Bestandssysteem

Het bestandssysteem ('*file system*') van een computer kun je zien als een boomstructuur waarin directories en bestanden georganiseerd zijn. Gebruik de (/), Linux, of backslash (\), Windows, als pad scheidingsteken. Omdat Windows ook de forward slash ondersteunt, kun je die ook gebruiken, anders moet je **** gebruiken als scheidingsteken. Alleen in de code. Binnen de command prompt volg je de Windows conventie.

14.2 os functies (sub-set)

`getcwd()`
'*Get Current Working Directory*' retourneert de huidige directory naam als een string.

`chdir()`
Wijzigt de huidige directory. De nieuwe directory geef je mee als string argument.

`listdir()`
Retourneert een list die alle bestanden en directories bevat in de directory die als argument is meegegeven. De namen verschijnen in willekeurige volgorde. Ze bevatten niet het volledige pad. Geef je **`listdir()`** op zonder argument dan krijg je de inhoud van de current working directory.

`system()`
`system()` krijgt een string argument dat beschouwd wordt als systeemcommando, dat dan door Python wordt uitgevoerd in de command shell. Je kunt de functie gebruiken om alles te doen wat door het besturingssysteem ondersteund wordt, inclusief het opstarten van andere programma's. Er zijn echter betere manieren om andere programma's te starten; functies waarvan de naam begint met '*exec*'.
Maar het werkt wel makkelijk; start gevolgd door file locatie.

```
os.system("start C:\\Oudheid\\Test.txt")
```

14.3 Voorbeelden

Een drietal voorbeelden om directory inhoud te laten zien.

```
# Dit is denk de standaard manier om directory inhoud te laten zien.
import os
def func_dir_tree_1(pDir):
  RetVal = []
  for dirpath, dirnames, filenames in os.walk("C:\\Oudheid"):
    L = os.listdir(dirpath)
    for node in L:
      node = dirpath + '/' + node
      if (os.path.isfile(node) == True): RetVal.append(node)
  return RetVal
L = func_dir_tree_1("C:\\Oudheid")
L.sort()
for node in L: print(node)
```

```
# Hier een alternatief en oefening in recursief programmeren
import os
def func_dir_tree_2(pDir,pRetVal):
  L = os.listdir(pDir)                          # geeft een list van de gehele directory inhoud
  if (len(L) > 0):                              # lege directories hoeven niet te worden verwerkt
    # Wat je krijgt zijn subdirectories, de childs, voeg de parent directory toe
    # sorteer eerst, listdir() doet het niet standaard
    L.sort()
    for i in range(len(L)):
      if (len(pDir) == 3): L[i] = pDir + L[i]
      else: L[i] = pDir + "/" + L[i]
    # Ga nu alle nodes langs, is het een file dan print je die.
    # Zoniet, dan roep je deze functie opnieuw aan.
    # Niet alle nodes zijn toegankelijk, zoniet, dan afbreken en de volgende pakken
    for node in L:
      try:
        if (os.path.isdir(node) == True):
          func_dir_tree_2(node, pRetVal)
        else:
          pRetVal.append(node)
      except: None
  return pRetVal           # hier komt je dus echt terecht als allerlaatste

L= func_dir_tree_2("C:\Oudheid", [])
L.sort()
for node in L: print(node)

# Hier een methode om Windows drive letters te achterhalen.
# Geleend van het internet.
import os
def func_win_drives():
  RetVal = []
  # Als je ord('Z') doet wordt die niet meegenomen! Het is van x tot y en niet van x tot en met y
  for drive in range(ord('A'), ord('a')):
    if os.path.exists(chr(drive) + ':/'):
      RetVal.append(chr(drive))
  return RetVal
print("De volgende drives bestaan:", func_win_drives())
```

15 Tekstbestanden

- '*Tekstbestanden*' of '*platte tekstbestanden*' zijn bestanden waarin alle tekens leesbaar. Er zit geen opmaakcodes in zoals bij PDF of MS word. HTML bestanden zijn ook platte tekst bestanden.
- Aan het einde van iedere regel staat een '*newline*' symbool, in Python voorgesteld als **`\n`.**
- Sommige Windows programma's slaan het op als '*carriage return plus line feed*' (**`\r\n`**). Linux gebruikt altijd een enkele **`\n`**.
- Zolang je in Python een bestand benadert als een regulier tekstbestand, zal Python de tekens die het leest converteren naar de standaard **`\n`** en vice versa wanneer het tekens wegschrijft.

In feite heet dit ook wel **`I/O`** (Input/Output). Python kent drie typen **`I/O`**: *text I/O*, *binary I/O* and *raw I/O*. Alles staat in de **`io`** module. Het heet ook wel een '*stream*'; je leest een stroom data in, of schrijft het weg.

15.1 Handles, pointers en buffers

- Als je een bestand opent krijg je een '*handle*' of '*file handle*'. Dit is een variabele ('*handvat*') die toegang geeft tot een bestand. Een handle is een **Object**.
- Een handle bevat een '*pointer*' die wijst naar een specifieke plaats in het bestand; waar de pointer staat, daar lees of schrijf je.
- Afhankelijk van hoe je een bestand opent, komt de pointer aan het begin of einde van een bestand te staan.
- Om een nieuw bestand te maken, een bestand met een naam die nog niet bestaat, open je het bestand als een '*schrijven alleen*' bestand.
- Na het openen van het bestand is de handle het enige toegangspunt tot het bestand. Alle acties die je uitvoert op het bestand, voer je uit met behulp van de methodes van de handle.
- Sluit altijd een bestand als je klaar bent.

- Pointerverplaatsing gaat automatisch, ongeacht uit hoeveel bytes een karakter bestaat. Als je vijf UTF-8 karakters die gecodeerd zijn met één byte of vijf UTF-8 karakters leest die met twee bytes zijn gecodeeerd, de pointer springt naar het juiste volgende karakter. Binnen binaire bestanden is dat niet zo.
- Manipulatie van tekstbestanden vindt plaats in buffers. Tussentijds wegschrijven van een buffer naar het bestand op schijf heet '*flushing*'.

15.2 Lezen van tekstbestanden

Om een tekstbestand te lezen, open je het, leest er wat uit, en dat sluit je weer af. Er volgen nu een aantal methodes waarbij het volgende tekstbestand wordt gebruikt.

```
regel 1\r\n
regel 2
```

De eerste regel heeft een carriage return, de tweede niet!

open()

Open een bestand. Syntax:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None,
closefd=True, opener=None)
```

file Naam van het bestand, eventueel met pad naam. Dit heet ook wel de '*full file name*'.

mode Geeft aan hoe het bestand moet worden geopend:

- **r** open for reading (default).
- **w** open for writing, truncating (leeg maken) the file first. Oppassen met deze!
- **x** open for exclusive creation, failing if the file already exists.
- **a** open for writing, appending to the end of the file if it exists.
- **b** binary mode.
- **t** text mode (default).
- **+** open a disk file for updating (reading and writing).
- **U** universal newlines mode (deprecated).

Er zijn twee manieren om een file handle te krijgen:

`<handle> = open(<bestandsnaam>)`	Zo doen de meeste programmeertalen het.
`open(<bestandsnaam>) as <handle>`	Geen close statement nodig om bestand te sluiten.

Een voorbeeldje, waarbij blijkt dat in dit geval alleen de **r** en **U** mode werkt, als je wilt lezen.

```
dir = "C:/Users/Fred/source/repos/text/"    # we volgen de UNIX conventie
try:
  fp = open(dir + "tekst_1.txt", mode='r')  # fp staat voor FilePointer
  buffer = fp.read()                        # leest alles
  print(buffer,"\n",id(fp), sep="")         # id(fp) is de file handle
  fp.close()                                # flush en sluit de stream
except: None
```

read()

Lees een bestand. Syntax:

```
read(<size>)
```

size Geeft aan hoeveel karakters je wilt inlezen. De pointer komt te staan op het eerstvolgende karakter. Als geen waarde is gedefinieerd, of -1 of **None**, dan wordt de gehele file, of restant, geladen.

Het lezen tot einde bestand heet ook wel tot **EOF** (end of file). Je hebt ook lezen tot **EOL** (end of line).

```
fp = open(dir + "tekst_1.txt")
print("-", fp.read(5), "-", sep='')
print("-", fp.read(5), "-", sep='')
print("-", fp.read(), "-", sep='')
fp.close
```

Dit wordt afgedrukt:

```
-regel-
- 1
re-
-gel 2-
```

Je ziet dat de **\r\n** niet meetelt, maar wel wordt afgedrukt, of opgeslagen in een variabele als je daarvoor kiest. Kwijt ben je hem nimmer.

close()
Sluit een bestand, en alles wordt weggeschreven. De file handle gaat dan weg. Een bestand kun je ook sluiten zonder **close()**. Gebruik syntax: **open(<bestandsnaam>) as <handle>**.
Je moet dan wel de **with** modifier gebruiken.

```
with open(dir + "tekst_1.txt") as fp:
  print(fp.read())
```

readline() en readlines()
De methode **read()** leest een geheel bestand, of een aantal karakters. Om een tekstbestand regel voor regel te lezen, kun je de **readline()** of **readlines()** gebruiken.
readline(): Leest één regel per keer in en retourneert een string. Het **\n** teken wordt meegenomen en dus als je de string afdrukt krijg je een lege regel. In het voorbeeld bestand heeft regel 2 geen \n.

```
print("-", fp.readline(), "-", sep='')
print("-", fp.readline(), "-", sep='')
```

Dit wordt afgedrukt:

```
-regel 1
-
-regel 2-
```

Wil je geen lege regel dan moet je dit doen:

```
print("-", fp.readline(), "-", end='')
```

readlines(): Leest alle regels in en plaatst één regel als string element in een list. De code hieronder geeft steeds een lege regel. Dat komt omdat elk element het \n teken heeft. Dus als de buffer wordt afgedrukt, wordt ook het *newline* teken afgedrukt. En omdat de **print()** functie zelf ook naar een nieuwe regel gaat **na is uitgevoerd**, krijg je een lege regel te zien na iedere tekstregel die wordt afgedrukt.

```
fp = open(dir + "tekst_1.txt")
L = fp.readlines()
print(L)                                    # ['regel 1\n', 'regel 2']
for e in L: print(e)                        # regel voor regel met steeds een lege regel
for e in L: print(e, end="")                # Zelfde maar dan zonder lege regel
for e in fp.readlines(): print(e, end="")   # zelfde als hierboven
fp.close()
```

Om even terug te komen op die lege regel. Dit: **print('regel 1\n', 'regel 2\n')** is niet hetzelfde als dit:

```
print('regel 1\n')
print('regel 2\n')
```

De print geeft een nieuwe regel als het statement helemaal klaar is, dus als alle objecten zijn afgedrukt. Dan volgt een *newline* of iets anders wat je in de **end=** parameter hebt gespecificeerd.

15.3 Welke leesmethode gebruiken?

De **read()** en **readlines()** lezen beide een bestand als geheel in. Dat is prima bij kleine bestanden maar problematisch bij grote bestanden. Je kunt buffer overflow krijgen. Kies daarom voor de methode om per regel in te lezen, **readline()** of per karakter, **read(1)**.

15.4 Schrijven in tekstbestanden

Om te schrijven in een tekstbestand open je het bestand met **mode='w'**, schrijft er naar en sluit het weer. Als je een bestaand bestand opent voor schrijven dan wordt de inhoud van het bestand **leeg** gemaakt.

```
fp = open(dir + "tekst_1.txt", mode='w')
fp.close()
fp = open(dir + "tekst_1.txt")
print(fp.read())                            # er valt niets te lezen!
fp.close()
```

Bestaat het bestand niet, dan wordt het gemaakt.

write()
Schrijft een string naar een bestand. Om tekst op een nieuwe regel laten beginnen moet je zelf de *newline* karakter specificeren.

```
fp = open(dir + "tekst_2.txt", mode='w')
fp.write('Regel 1')
fp.write('Regel 2')
fp.write('Regel 3')
fp.write('\nRegel 4')
fp.close()
fp = open(dir + "tekst_2.txt")
print(fp.read())
fp.close()
```

De output is:
```
Regel 1Regel 2Regel 3
Regel 4
```

writelines()
Schrijft een list van strings in een keer naar een bestand. De list mag **geen** geneste-lists hebben en ook hier geld weer dat je de *newline* moet specificeren als je hem erin wilt hebben.
```
L = ['Regel 1\n', 'Regel 2\n', 'Regel 3\n']
fp = open(dir + "tekst_3.txt", mode='w')
fp.writelines(L)
fp.close()
fp = open(dir + "tekst_3.txt")
print(fp.read())
fp.close()
```

15.5 Toevoegen aan tekstbestanden

Je kunt een bestand in '*toevoegen*' of '*appending*' mode openen. Dan voeg je tekst toe aan het einde van het bestand. Je opent een bestand met **mode='a'**, schrijft iets naar het bestand en sluit het weer. Voor het voorbeeld hieronder gebruiken we de **writelines()** methode.

```
L=['\nRegel 3\n', 'Regel 4\n', 'Regel 5\n']
fp = open(dir + "tekst_1.txt", mode='a')
fp.writelines(L)
fp.close()
fp = open(dir + "tekst_1.txt")
print(fp.read())
fp.close()
```

15.6 os.path methodes

Een aantal handige en veel gebruikte functies vindt je in de **os.path** module. Je moet wel eerst de module importeren. Je kunt een alias gebruiken als je wilt.

```
import os.path          # zonder alias
import os.path as io    # met alias, en dan hoef je niet de volledig module naam gebruiken
```

Hierbij een aantal functies:

exists() Geeft aan of een pad of een bestand wel of niet bestaat.
```
import os.path
if os.path.exists(dir): print(True)
else: print(False)
if os.path.exists(dir + "teksts_1.txt"): print(True)
else: print(False)
```

isfile() Geeft aan of het argument wel of niet een file is.
```
if os.path.isfile(dir): print(True)
else: print(False)
```

isdir() Geeft aan of het argument wel of niet een directory is.
```
if io.isdir(dir): print(True)
else: print(False)
```

join() Probeert van het argument een geldig pad te maken. Dus het zorgt dat de '*slashes*' goed staan.
```
import os
import os.path as io   # met alias, en dan hoef je niet de volledig module naam gebruiken
os.chdir("C:\Program Files\Python36")
L = os.listdir()
for e in L:
  bestand = io.join(os.getcwd(), e)
  if (io.isfile(bestand)):
    t = "File\t\t"
  elif (io.isdir(bestand)): t = "Directory\t"
  else: t = "Unknown"
  print(t, ": ", bestand, sep='')
```

Basename() Haalt het laatste niveau uit een pad naam. Dat kan een file zijn, maar ook een directory.
```
print(io.basename("C:/Program Files/Python36/python.exe")) # python.exe
print(io.basename("C:\\Program Files\\Python36"))          # Python36
print(io.basename("C:/Program Files"))                     # Program Files
print(io.basename("C:\\"))                                 # leeg
print(io.basename(""))                                     # leeg
```

dirname() Retourneert het volledige pad minus het laatste niveau. Dat is altijd een directory.
```
print(io.dirname("C:/Program Files/Python36/python.exe")) # C:/Program Files/Python36
```

```
print(io.dirname("C:\\Program Files\\Python36"))          # C:\Program Files
print(io.dirname("C:/Program Files"))                     # C:/
print(io.dirname("C:\\"))                                 # C:\
print(io.dirname(""))                                     # leeg
```

getsize() Geeft de grote van een bestand in bytes. Alleen relevant als het een file betreft?

```
print(io.getsize("C:/Program Files/Python36/python.exe")) # 100504
print(io.getsize("C:\\Program Files\\Python36"))          # 4096
print(io.getsize("C:/Program Files"))                     # 12288
print(io.getsize("C:\\"))                                 # 12288
```

15.7 Encoding

Tekstfiles hebben een zogeheten '*encoding*' ('*versleuteling*') die zegt hoe de tekens in een bestand gecodeerd zijn. De standaard encoding van jouw systeem wat Python gebruikt is als volgt te bepalen:

```
from sys import getfilesystemencoding
print(getfilesystemencoding())          # utf-8
```

Gebruik de encoding optie als een tekstbestand een andere encoding heeft dan de standaard.

```
dir = "C:/Users/Fred/source/repos/text/"                    # we volgen de UNIX conventie
fp = open(dir + "tekst_1.txt", mode='r', encoding ='ascii')
print(fp.read())
fp.close
```

16 Exceptions

Een '*syntax error*' is een compileer fout en het programma kan dan niet runnen. Tijdens draaien of runnen kunnen er ook fouten optreden, de '*runtime errors*'. Het programma stopt dan. Maar deze foutmeldingen kun je ook laten afhandelen door de '*exception handling*', je '*catcht*' ('*vangt*') de fout op en kan de code laten bepalen wat er moet gebeuren. De bedoeling van exception handling is dat het programma blijft draaien. Om exception handling te doen gebruikt je de volgende constructie.

```
try:
  <TryCode>
except <ExceptionError> <as <Variabele>>
  <ExceptCode>
<else:>
  <ElseCode>
<finally:>
  <FinallyCode>
```

De **else** en **finally** zijn optioneel.

try:
  <TryCode>

Op alle code wat tussen **try** en **except** staat vind de exception handling plaats.

except <Error>
  <Except*Code*>

Als er een fout optreed genereert het systeem een errorcode. Als je die code hier op vangt, dus neerzet, wordt de *ExceptCode* uitgevoerd. Je mag zoveel **except** statements in een **try-except** blok zetten als je wilt.
Breidt je de **except** statement uit met de **as variabele** optie dan kun je extra informatie over de fout verkrijgen.

```
except IOError as ex:
  print(ex.args)
```

Je kunt ook de fout opvangen en er verder niets mee doen. Doe dan dit.

```
except: None
```

Je kunt ook geen foutcode meenemen. Doe dan dit

```
except:
  <ExceptCode>
```

<else:>
  <ElseCode>

Als er **geen** fout is opgetreden wordt de *ElseCode* uitgevoerd.

<finally:>
  <FinallyCode>

De *Finallycode* wordt altijd uitgevoerd, ongeacht of er fouten zijn of niet.

Exception handling kan genest worden. Het is tenslotte gewone code.

16.1 Exception codes

Dit zijn een aantal veel voorkomende exceptions:

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 (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 van een waarde met een data type dat niet ondersteund wordt door de operatie die je uitvoert.

16.2 Voorbeeld

Code met informatie over de fout

```
dir = "C:/Users/Fred/source/repos/text/"
while True:
  f = input("File naam: ")
  try:
    if (f == '0'): break
    fp = open(dir + f)
    print(fp.read(None))
    fp.close()
  except IOError as ex:
    print(ex.args)
  else:
    print("Bestand:", dir + f)
  finally:
    print("Altijd")
```

Code zonder informatie over de fout

```
dir = "C:/Users/Fred/source/repos/text/"
while True:
  f = input("File naam: ")
  try:
    if (f == '0'): break
    fp = open(dir + f)
    print(fp.read(None))
    fp.close()
  except: None
```

16.3 Genereren van exceptions

Gebruik het **raise** statement om zelf een exception te genereren. Dat kan om allerlei redenen handig zijn. Als je een klasse programmeert kun je die foutmeldingen laten generen via het **raise** statement zodat de aanroepende programmatuur er iets mee kan doen. Je kan een bestaande exception genereren, maar je kunt ook je eigen exceptions definiëren.

17 Binaire Bestanden

'*Binaire bestanden*' zijn alle bestanden die geen tekstbestanden zijn. Bestandsverwerking is globaal hetzelfde.

mode = 'rb'	Alleen lezen van een binair bestand. Pointer wordt op positie 0 gezet. Of je leest een aantal bytes of alles.
mode = 'wb'	Alleen schrijven naar een binair bestand. **Oppassen**, bestand wordt eerst leeggemaakt.
mode = 'rb+'	Lezen en schrijven van een binair bestand. Het bestand wordt **niet** eerst leeggemaakt.
mode = 'wb+'	Lezen en schrijven van een binair bestand. **Oppassen**, bestand wordt eerst leeggemaakt.

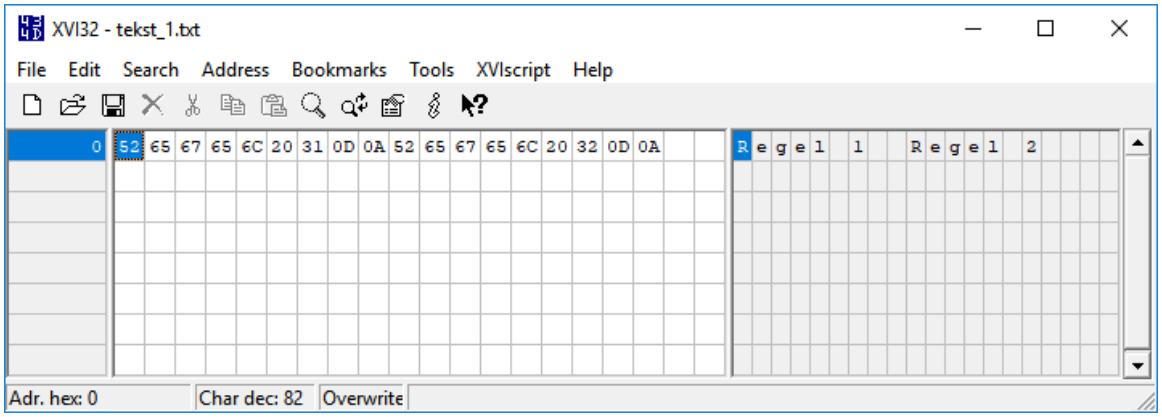
Een binair bestand is te zien als één continue stroom van bytes. Dus of je leest het in één keer in, of je leest het per byte in, of een aantal bytes. Karakter als **'\n'** worden niet geïnterpreteerd. Dus de bytes in een binair bestand kunnen van alles beteken. Je kunt alleen de functies **read()** en **write()** gebruiken. Strings kun je schrijven maar moet je prefixen met een b, dat betekent byte string; **b'Test'**. Zo weet Python dat het een byte string is. De **b** zelf maakt geen onderdeel uit van de variabele waarde zelf.

17.1 Voorbeelden

Nu een aantal voorbeelden. We gebruiken dezelfde tekstbestanden als hiervoor. Want een tekstbestand kun je ook als binair bestand inlezen. Dit is tekst_1.txt:

```
regel 1\r\n
regel 2\r\n
```

Om een binair bestand te lezen kun je gebruik maken van zogenaamde hex-editors. Niet dat tekst dan persé leesbaar wordt, maar alle bytes worden dan ingelezen. Hieronder de byte representatie van het test bestand.



Dit is leesbaar (rechter gedeelte) omdat het een tekstbestand betreft.

```
# Opent het bestand voor lezen alleen.
dir = "C:/Users/Fred/source/repos/text/"      # we volgen de UNIX conventie
fp = open(dir + "tekst_1.txt", mode='rb')
buffer = fp.read()
print(buffer)                                 # b'regel 1\r\nregel 2\r\n'
fp.close()
```

De **b** maakt geen onderdeel uit van de byte string inhoud. Het is een prefix die jou vertelt dat het geen gewone string is, maar een byte string.

```
# Open het bestand voor schrijven alleen. Bestand wordt eerst leeg gemaakt.
dir = "C:/Users/Fred/source/repos/text/"      # we volgen de UNIX conventie
fp = open(dir + "tekst_1.txt", mode='wb')
fp.write(b'Regel 1\tRegel 2\t\nRegel 3\t')
fp.close()
fp = open(dir + "tekst_1.txt", mode='rb')     # Hier zie je dan de nieuwe inhoud van de file
buffer = fp.read()
print(buffer)                                 # b'Regel 1\tRegel 2\t\nRegel 3\t'
fp.close()
```

```
# Opent het bestand voor lezen/schrijven en zet de schrijfpointer waar je maar wilt met de read()
dir = "C:/Users/Fred/source/repos/text/"      # we volgen de UNIX conventie
fp = open(dir + "tekst_1.txt", mode='rb+')
fp.read(8)                                    # zet de pointer op positie 8
fp.write(b'Hallo\n')
fp.close()
fp = open(dir + "tekst_1.txt", mode='rb')     # Hier zie je dan de nieuwe inhoud van de file
print(fp.read())                              # b'Regel 1\rHallo\n 2\r\n'
fp.close()
```

Hier zie je dus dat je gaat overschrijven daar waar de pointer staat.

```
# Opent het bestand voor lezen/schrijven. Bestand wordt eerst leeg gemaakt.
# Je kunt hier de read() functie gebruiken, maar doet hier niets.
dir = "C:/Users/Fred/source/repos/text/"      # we volgen de UNIX conventie
fp = open(dir + "tekst_1.txt", mode='wb+')
print(fp.read(8))                             # b''
fp.write(b'Hallo\n')
print(fp.read())                              # b''
fp.close()
fp = open(dir + "tekst_1.txt", mode='rb')     # Hier zie je dan de nieuwe inhoud van de file
print(fp.read())                              # b'Hallo\n'
fp.close()
```

Je ziet hier dus dat de **`read()`** niets doet.

17.2 Pointer Positie

Als je een bestand opent in **rb+/wb+** kun je de schrijfpointer op elke plaats in de file zetten waar je maar wilt via de **seek()** methode. Deze methode kan de pointer binnen een bestand verplaatsen. Syntax:
seek(*offset*[, *whence*])
Zet de stream position, pointer, naar de ***offset*** waarde die afhankelijk is van de ***whence*** parameter. De default waarde is **SEEK_SET**.

SEEK_SET of 0 Zet de pointer relatief ten opzichte van het begin van het bestand. Offset >= 0.
SEEK_CUR of 1 Zet de pointer relatief ten opzichte van de huidige positie.
SEEK_END of 2 Zet de pointer relatief ten opzichte van de huidige positie. Offset moet negatief getal zijn.

```
fp = open(dir + "tekst_1.txt", mode='wb+')
fp.write(b'Hallo\n')
fp.seek(2)                  # Relatief vanaf begin; je zet pointer op positie 3
print(fp.read(2))           # b'll'. Pointer komt op positie 5.
print(fp.read())            # b'o\n'. Pointer komt of EOF
fp.seek(0)                  # Relatief vanaf begin; je zet pointer op positie 1, begin van file
print(fp.read(2))           # b'Ha'. Pointer komt op 3
fp.seek(2,1)                # Relatief vanaf huidige positie. je zet pointer op positie 5
print(fp.read(2))           # b'o\n'. Pointer komt of EOF
fp.seek(-4, 2)              # Relatief vanaf einde; Je ze pointer op positie 3.
print(fp.read())            # b'llo\n'. Pointer kom tof EOF
fp.close()
```

17.3 Hexadecimaal schrijven

Je kunt elk karakter als hexadecimaal karakter naar een string of bestand schrijven met de **\x<nn>** notatie. Als het een zichtbaar teken is wordt het "goed" afgedrukt. (De **x** staat dus voor he**x**adecimaal).

```
x = "Hello,\x20world!"          # \x20 is hexadecimaal 20 en decimaal 32 en is een spatie
print(x)                        # Hello, world!
```

Maar **\x00**, **\x01**, **\x02** zijn control karakters. Wat gebeurt er als je die in een string stopt?

```
x = "Hello,\x20\x00\x01\x02world!"
print(x)                                # Hello,  world!
```

Je ziet een tweetal rare karakters, en **\x00**, de **NULL** geeft ook een spatie. Om te zien wat er precies staat moet je de string omzetten naar een bytestring. Je gaat casten.

```
x = "Hello,\x20\x00\x01\x02world!"
print(x)                                # Hello,  world!
y = bytes(x, 'utf-8')                   # omzetten naar een bytestring
print(y)                                # b'Hello, \x00\x01\x02world!'
```

Wat leesbaar is wordt als leesbaar teken afgedrukt, anders krijg je de hexadecimale code te zien. Beter is om de **decode()** en **encode()** methodes te gebruiken.

```
x = "Hello,\x20\x00\x01\x02world!
print(x)                                # Hello,  __world!
y = bytes(x, 'utf-8')                   # omzetten naar een bytestring
z = x.encode('utf-8')                   # Beter is dit.
print(y,z, sep='\n')                    # b'Hello, \x00\x01\x02world!'
a = z.decode('utf-8')                   # En hier maak je er weer een gewone string van
print(a)                                # en krijg je weer de rare karakters
```

encode() Is een methode van de string klasse en maakt er een byte string van.
decode() Is een methode van de byte string klasse en maakt er een string van.

Hier volgt dan een voorbeeldje van het schrijven naar een bestaand bestand.

```
dir = "C:/Users/Fred/source/repos/text/"      # we volgen de UNIX conventie
fp = open(dir + "tekst_9.txt", mode='wb')
# De volgende twee statements doen precies hetzelfde
fp.write('1:Regel 1\x0D\x0ARegel 2\x0D\x0A'.encode('utf-8'))
fp.write(b'2:Regel 1\x0D\x0ARegel 2\x0D\x0A')
fp.close()
fp = open(dir + "tekst_9.txt")
print(fp.read())
fp.close()
```

18 Bitsgewijze Operatoren

De '*bitsgewijze operatoren*' werken alleen maar op byte data types. Dat zijn ook byte strings. Een byte is 8 bit. Een bit is de kleinste data-eenheid die een computer kan manipuleren en heeft de waarde 0 of 1.

1 bit	0-1.
4 bits	Nibble (Hexadecimaal); 0000-1111 0-15.
8 bits	Byte; 0000 0000 – 1111 1111 (0 -65535).
16 bits	Word: 0000 0000 0000 0000 – 1111 1111 1111 1111.
32 bits	Long word.
64 bits	Big integer.

Hier zie je dus dat 4 bits een hexadecimaal getal vormen. Net zoals in het decimale stelsel zit rechts het cijfer met de '*laagste*' waarde. Dit heet ook wel '*least significant bit*'. Dus '0001' is 1 en '1111' is 15. Maar dit werkt tot op byte niveau. Computers slaan data per byte op en dit '11111111 00000000' kan hetzelfde zijn als '00000000 11111111'. Dit heet ook wel '*Endiannes*'. Welk byte is het '*least significant byte*'? Ga er voor het gemak vanuit dat dit het meest rechter byte is.

18.1 Codering van tekst

Python staat standaard voor tekst op UTF-8. Dat is een variabel decoderingsschema. Een karakter kan 1 byte, 2 bytes, 3 bytes of 4 bytes hebben. Dit is het algoritme:

- Een byte die een 0 heeft als meest linker bit is een ASCII teken.
- Een byte die een 1 heeft als meest linker bit is de start van een sequentie van meerdere bytes die samen één teken representeren. De sequentie bestaat uit een '*leidende byte*' (de meest linker byte) en één of meer '*volgende bytes*'.
- Bij een multi-byte sequentie leeft de '*leidende byte*', van links naar rechts, een aantal bits met waarde 1, gevolgd door een bit met waarde 0, gevolgd door een aantal resterende bits. De totale lengte van de multi-byte sequentie is het aantal bits met waarde 1 links van de meest linkse 0 in de leidende byte.
- Iedere '*volgende byte*' heeft 10 als de meeste linkse twee bits.
- De minimumlengte van de sequentie is twee bytes, de maximumlengte is zes bytes (de '*leidende byte*' is dan 1111110x). In de praktijk is UTF-8 codering beperkt tot een maximum van 4-byte sequenties.

18.2 Coderen van getallen

Een getal opgeslagen als string representatie is niet hetzelfde gecodeerd als een getal opgeslagen als integer representatie. Getallen worden ook als bit patronen opgeslagen maar dat is volkomen transparant zolang je maar niet '*bitsgewijs*' gaat rekenen.

- Positieve gehele getallen zijn altijd gecodeerd als multi-byte patronen, die een 0 als meest linker bit hebben. De meest linker bit is het '*sign bit*' of '*teken bit*'. De rest van het patroon is zoals hierboven uitgelegd.
- Negatieve getallen zijn op een andere manier gecodeerd en gebruiken het zogenaamde '*twee-complement systeem*'. Om een negatief geheel getal te coderen, neem je eerst de absolute waarde van dat getal en maak je het bit patroon. Dan inverteer je alle bits, een 0 wordt een 1, een 1 wordt en 0 en bij dat resultaat tel je een 1 bij op. Een negatief getal begint dan altijd met een 1.
- Gebroken getallen worden opgeslagen in de wetenschappelijke notatie, waarbij een deel van het multi-byte patroon als exponent gebruikt wordt.

Onderstaande plaatje (bron Wikipedia) laat het duidelijk zien.

0	1	1	1	1	1	1	1	= 127
0	0	0	0	0	0	1	0	= 2
0	0	0	0	0	0	0	1	= 1
0	0	0	0	0	0	0	0	= 0
1	1	1	1	1	1	1	1	= −1
1	1	1	1	1	1	1	0	= −2
1	0	0	0	0	0	0	1	= −127
1	0	0	0	0	0	0	0	= −128

Om een negatief getal te encoderen inverteer je alle bits, berekent het resultaat, telt daar weer 1 bij op en zet er dan een `'-'` voor. Gebruik de **`bin()`** functie om een binaire representatie van een integer te laten zien.

```
x = 0;  b_x = bin(x)
y = 2;  b_y = bin(y)
z = -2; b_z = bin(z)
print(b_x, b_y, b_z)  # 0b0 0b1 -0b1
x = 127
y =-127
print(bin(x), bin(y)) # 0b1111111 -0b1111111
```

Je ziet wel dat negatieve getallen dezelfde bit notering krijgen als positieve getallen maar dan met een **-0b** teken. Dus je kan niet goed zien hoe negatieve getallen intern zijn opgeslagen. (De **b** betekent hier *binair*, denk ik).

18.3 Manipulatie van bits

De volgende operatoren doen data manipulatie op bit niveau:

<< **Shift left:** Verschuift elke bit in de byte het opgegeven aantal naar links, waarbij het bit patroon aan de rechterkant met 0-bits wordt aangevuld. De bit lengte en dus het getal wordt groter.

```
x = 21
y = x << 2
print(x, y, bin(x), bin(y)) # 21 84 0b10101 0b1010100
```

Het verschuiven van een bit levert steeds een verdubbeling op.

>> **Shift right:** Verschuift elke bit in de byte het opgegeven aantal naar rechts waarbij het bit aan de linkerkant wegvalt. De bit lengte wordt dus kleiner en dus wordt het getal kleiner.

```
x = 4
y = x >> 2
print(x, y, bin(x), bin(y)) # 4 1 0b100 0b1
x = 5
y = x >> 2
print(x, y, bin(x), bin(y)) # 5 1 0b101 0b1
x = 6
y = x >> 2
print(x, y, bin(x), bin(y)) # 6 1 0b110 0b1
```

Hier zien we wat aparts. Shiften naar rechts kan hetzelfde resultaat opleveren.

& **Bitwise and:** Vergelijkt de bit positie van twee getallen met de **and** operator. Levert alleen een 1 op als beide posities een 1 zijn, anders een 0.

```
x = 12
y = 11
z = x & y
print(x,y,z, bin(x), bin(y), bin(z)) # 12 11 8 0b1100 0b1011 0b1000
```

| **Bitwise or:** Vergelijkt de bit positie van twee getallen met de **or** operator. Levert een 1 op als één van beide posities een 1 is, anders een 0.

```
x = 12
y = 11
z = x | y
print(x,y,z, bin(x), bin(y), bin(z)) # 12 11 15 0b1100 0b1011 0b1111
```

~ **Bitwise not:** Inverteert elke bit positie. Een 0 wordt een 1, een 1 wordt een 0. Dus een positief getal wordt een negatief getal en andersom.

```
x = 12
z = ~x
print(x, z, bin(x), bin(z)) # 12 -13 0b1100 -0b1101
x = -12
z = ~x
print(x, z, bin(x), bin(z)) # -12 11 -0b1100 0b1011
```

Het twee complements algoritme zorgt ervoor dat je er steeds 1 naast zit.

^ **Bitwise exclusive or:** Vergelijkt elke bit positie met elkaar. Zijn de aan elkaar gelijk dan wordt het een 0, zijn ze ongelijk dan wordt het een 1.

```
x = 12
y = 11
z = x^y
print(x, y, z, bin(x), bin(y), bin(z)) # 12 11 7 0b1100 0b1011 0b111
```

De bitsgewijze **xor** operator geeft een gemakkelijke manier om getallen te versleutelen. Neem een bit patroon en noem dat het '*masker*'. Pas dat masker toe op een getal via de **xor**. Dat geeft een nieuw getal, dat het versleutelde getal is. Iemand die het masker niet kent, kan het originele getal niet herleiden. Maar iemand die het masker kent, kan het originele getal eenvoudigweg terugkrijgen door het masker opnieuw toe te passen op het versleutelde getal.

```
x = 12
m = 11
c = x ^ m                                # encoded
d = z ^ m                                # decoded
print(x, c, d, bin(x), bin(c), bin(d))  # 12 7 12 0b1100 0b111 0b1100
```

18.4 Het nut van bitsgewijze operaties

Alles wat je met bitsgewijze operatoren kunt doen kun je ook met de gebruikelijke berekeningsmethodes doen, die ook meer kunnen. Maar bitsgewijze operatoren werken wel veel sneller dan de gebruikelijke methodes. Intern zal de Python interpreter/compiler er wel gebruik van maakt waar nodig.
Hier een voorbeeldje om kleurcodering in één getal op te slaan wat dan wel weer handig kan zijn. De RGB-code bestaat uit 3 bytes. Als je ze representeert als hexadecimaal dan kun je de drie bytes aan elkaar plakken en krijg je een totaal getal die je ook weer kunt uitpakken.
R = rechter byte, most significant byte.
G = middelste byte.
B = linker byte, least significant byte.

```
def getRGB(color):
  r = (color >> 16) & 255                    # hier pak je de most significant byte
  g = (color >> 8) & 255                     # hier pak je de middelste byte
  b = color & 255                            # hier pak je de least significant byte
  return r, g, b
r, g, b = getRGB(223567)
print("rood={}, groen={}, blauw={}".format(r,g,b))   # rood=3, groen=105, blauw=79
print(getRGB(49151))                                 # (0, 191, 255)
```

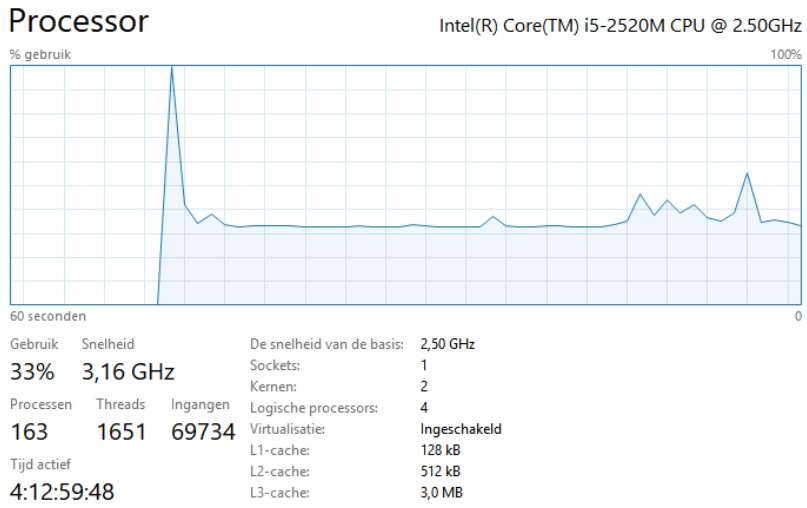
```
def setRGB(r,g,b):
  if (r > 0): r = r*65536       # dit 2**16.
  if (g > 0): g = g*256         # dit is 2**8
  RetVal = r + g + b
  return RetVal, hex(RetVal)
print(setRGB(3,105,79))         # ('0x3694f', 223567)
print(setRGB(0,191,255))        # ('0x0bfff', 49151)
```

Hieronder zie je een wat meer uitgewerkt geheel. Alle RGB-codes worden bepaald en weggeschreven naar een csv tekstbestand. Excel kan de file automatisch laden, maar kan maximaal maar 1 miljoen rijen aan. Het tekstbestand bestaat uit 255*255*255 = 16.581.375 rijen. De variabele **i** staat voor integer, **h** voor hexa.

```
from datetime import datetime      # nodig om de tijd af te drukken

# Functie om een hex getal te krijgen met altijd een leidende nul als < 16
def func_hex(i):
  h = hex(i).lstrip('x0')
  if (i < 16): h = '0'+h
  return h

# Bij r schuif je de bits 16 plaatsen naar rechts. Je houdt dan 8 bits over.
# Bij g schuif je dit bits 8 plaatsen naar rechts. Je houdt dan 16 bits over,
# maar de 8 bits die je moet hebben staan helemaal links; 1 t/m 8.
# Dus de bits 9 t/m 16 moet je kwijtraken, en dus AND je met 255 is 00000000 11111111
# bij de b staat de byte al op de juiste plaats, maar moet je moet de bits 9 t/m 24 kwijtraken.
# Is zelfde als bij g
def getRGB(color):
  r = (color >> 16)        # hier pak je de most significant byte
  g = (color >> 8) & 255   # hier pak je middelste byte
  b = color & 255          # hier pak je de least significant byte
  return r, g, b

# Methode 1 om de drie kleuren in één getal te stoppen.
def setRGB1(r,g,b):
  h = '0x'+func_hex(r) + func_hex(g) + func_hex(b)
  return h, int(h, 16)

# Methode 2 om de drie kleuren in één getal te stoppen.
def setRGB2(r,g,b):
  if (r > 0): r=r*65536  # dit 2**16
  if (g > 0): g=g*256    # dit is 2**8
  i = r + g + b
  return hex(i), i
```

```
print(datetime.now())
dir = "C:/Users/Fred/source/repos/text/"
fp = open(dir + "rgb.csv", mode='w')
sep=";"
for r in range(256):
  for g in range(256):
    for b in range(256):
      h1, i1 = setRGB1(r,g,b)
      h2, i2 = setRGB1(r,g,b)
      c1 = getRGB(i1)
      c2 = getRGB(i1)
      #print(r,g,b,h1,i1,h2,i2,c1,c2)
      fp.write(str(r)+sep+str(g)+sep+str(b)+sep)
      fp.write(str(h1)+sep+str(i1)+sep)
      fp.write(str(h2)+sep+str(i2)+sep)
      fp.write(str(c1)+sep)
      fp.write(str(c2)+'\n')
fp.close()
print(datetime.now())

# Hier open je de laatse regels
fp = open(dir + "rgb.csv", mode='r', encoding ='ascii')
fp.seek(1271172000)
print(fp.read())
fp.close
```

Het geheel draaide op een laptop met i5-2520M CPU, 8Gb geheugen, interne GPU, 256 GB SSD.
Het proces liep van: 12:02:44 tot 12:46:45, bestandsgroottte: 1.271.172.196 bytes. Alle CPU-tijd wordt door Python opgeslokt. Maar, CPU-gebruik ging niet naar 100%, bleef tussen 30-40% schommelen.



19 Object Oriëntatie

Hiervoor ging het over '*Gestructureerd Programmeren*' of '*Imperatief Programmeren*'. Een andere manier van programmeren is '*Object Georiënteerd Programmeren*' of '*Object Oriented Programming*'. Een *Object* kun zien als een pointer die wijst naar iets in het geheugen waar wat gebeurt. Een *Object* is een instantie van een *Klasse*. Een klasse is een data type, maar dan wel een heel bijzondere. Een class (Engels) is dus code. Een object is een variabele. Een class (code) bestaat uit vier componenten (binnen Python is de terminologie ietwat afwijkend):

Constructor	Initialiseren van object als je die definieert.
Properties	Rekwisieten; Lokale variabelen binnen het object en alleen bekend binnen het object. Een property is leesbaar, schrijfbaar of beide. Een property is dus een **klasse variabele**.
Methods	Methodes; functies van een object. Een methode is dus een **klasse functie**.
Events	Gebeurtenissen; een object kan events genereren en consumeren. Als een event binnenkomt kan het object in beweging komen. Functies worden dan uitgevoerd etc. **Let op:** Events zijn niet exclusief voor classes.
Inheritance	Overerving; dit is geen component van een object maar een programmeer concept om op basis van een bestaande class, een nieuwe class te maken.

Niet elke class hoeft al deze componenten te ondersteunen. Je kan bijvoorbeeld honderden objecten van dezelfde class hebben. Ze hebben dus allemaal dezelfde event handling. Maar als een ander object o.i.d. een event afvuurt gaan niet al die honderden objecten iets doen. Alleen het object waarvoor het event is bedoelt gaat aan de slag. Je moet dat wel als zodanig programmeren.
Opmerking 1: Hoewel een class net iets anders is dan een object, ontkom je er niet aan om ze zo af toe door elkaar te halen omdat het uiteindelijk, praktisch gezien, wel om hetzelfde gaat.
Opmerking 2: Een class kan ook eigen lokale variabelen hebben. Dit lijkt op een property en bij sommige talen kun je lokale variabelen ook publiekelijk (public) maken zodat ze ook buiten de class beschikbaar zijn.

19.1 Klassen, objecten, en hiërarchieën

In de object-georiënteerde wereld behoort iedere onderscheidbare entiteit tot een class. Een class is een generiek model voor een groep entiteiten. De class beschrijft alle attributen die de entiteiten gemeen hebben, en beschrijft de methodes die de class aanbiedt waarmee de wereld buiten de class invloed op de class kan uitoefenen. Classes bestaan in hiërarchieën. Een generieke, hoog-niveau class kan eigenschappen en methodes beschrijven die gedeeld worden door verschillende onder-classes.
Met onder-class bedoel ik hier een volledige zelfstandige class die wordt gebuikt door een andere, hoger niveau class. Je geeft een class als variabele door aan een andere class. Die *passed* variabele kun je zien als onder-class. We gaan straks nog zien dat je dit dan meestal als *passed by value* moet doen. (Paragraaf 19.5).
Een onder-class kan eigenschappen en methodes toevoegen, en kan zelfs eigenschappen en methodes wijzigen. Iedere onder-class kan zelf ook weer onder-class hebben. Een class hiërarchie wordt geïmplementeerd met behulp van een mechanisme dat '*Inheritance*' heet. **Let op;** dit is niet hetzelfde als geneste-classes. Een geneste class is een class binnen een class binnen een class etc.

19.2 Object oriëntatie

Om een nieuwe class te maken maak je gebruik van het gereserveerde woord **`class`**.

`class()`

Wanneer een **`class`** gedefinieerd is, kun je instanties van die class toekennen aan variabelen. Als voorbeeld een class die een punt in een 2-dimensionale ruimte beschrijft. Die class noemen we Punt.

```
class Punt:
  pass                # betekent doe niets; Placeholder voor latere code
```

Om een klasse aan een variabele toe te kennen maak je gebruik van de assignment, **=**, operator. Als je dat doet, heb je een object gemaakt. De **`pass`** statement is een code statement die niets doet, dit omdat er wel een code statement moet staan.

```
o_P = Punt()
print(type(o_P))      # <class '__main__.Punt'>
print(o_P)            # <__main__.Punt object at 0x0000021E5F639BE0>
```

Hier zie je dat het een object is op geheugenadres **`21E5F639BE0`**. Een punt heeft een x en een y coördinaat. Dan kun je zien als attributen van de class en zijn dus de properties van een class wat weer een lokale variabele binnen de class is. Omdat Python '*soft typing*' gebruikt, kun je waardes aan attributen, variabelen, toekennen op het moment dat je een property (variabele) maakt. Je maakt daarvoor gebruik van een speciale initialisatie methode, **`__init__()`**, in de class. In andere talen heet dat ook wel de '*constructor*'.

`__init__()`

Je gebruikt de **`__init__()`** methode om alles te initialiseren waarvan je wilt dat het bestaat bij het instantiëren van een class. In het geval van class Punt een x en een y coördinaat.

```
class Punt:
  def __init__(self):
    self.x = 12.12
    self.y = 99.01
o_P = Punt()
o_P.z = 11              # blijkbaar kun je variabelen van een klasse buiten de klasse definiëren
print("({}, {}, {})".format(o_P.x, o_P.y, o_P.z))                  # (12.12, 99.01, 11)
```

Eerst iets aparts: Binnen Python kun je klasse variabelen definiëren buiten de klasse definitie om. Die hangen dan aan het object. Dat is gevaarlijk, want nu kunnen objecten van dezelfde klasse anders zijn. Tevens schijn je ook variabelen te kunnen definiëren buiten de **`__init__()`** methode om. Misschien kan het handig zijn. Volgens mij kan het niet bij andere talen, zoals C#.

`__init__()` krijgt één parameter, die ***self*** genoemd wordt. Iedere methode die je definieert binnen een class krijgt **altijd minstens één parameter**, die gevuld wordt met een referentie aan het object waarvoor je de methode aanroept. Het is de gewoonte dat deze eerste parameter altijd ***self*** genoemd wordt. Dat is niet verplicht. Dit is ook afwijkend als bij bijvoorbeeld C#. Daar definieer je methodes (=functies) precies hetzelfde als in niet class code.

Als je een versie van een class wilt hebben met extra attributen, kun je '*inheritance*' gebruiken om een nieuwe class te maken op basis van de bestaande class. In feite maak je dan een andere class. Inheritance kun je zien als een programmeer aspect dat onderdeel is van object georiënteerd programmeren.

De **`__init__()`** methode kan, net als elke andere methode, argumenten krijgen. Je kunt die argumenten gebruiken om class variabelen een waarde te geven.

```
class Punt:
  def __init__(self, x, y):
    self.x = x
    self.y = y
o_P = Punt(12, 12)       # hier initialiseer je het object en zet meteen de properties x en y
print("({}, {})".format(o_P.x, o_P.y)) # (12, 12)
o_P.x = 100              # hier verander je ze weer van waarde
o_P.y = 200
print("({}, {})".format(o_P.x, o_P.y))  # (100, 200)
```

Overloading

Even een uitstapje. Wat hierboven gebeurde heet '*function overloading*'. Dat is een techniek om hoe om te gaan met optionele argumenten in de functie parameter string, en ook als ze van verschillende data types zijn. Dit is tamelijk krachtig. In het bovenstaande voorbeeld kan ik onderstaande **niet** doen:

```
o_P2 = Punt()
```

Je bent namelijk nu verplicht om twee argumenten waardes mee te geven. Wil je dat niet, dan moet je de functie parameters default waardes meegeven. Je krijgt dan dit:

```
class Punt:
  def __init__(self, x=1.0, y=1.0):
    self.x = x
    self.y = y
o_P1 = Punt(12, 12)       # hier initialiseer je het object en zet meteen de properties x en y
print("({}, {})".format(o_P1.x, o_P1.y)) # (12, 12)
o_P1.x = 100              # hier verander je ze weer van waarde
o_P1.y = 200
print("({}, {})".format(o_P1.x, o_P1.y))  # (100, 200)
o_P2 = Punt()
print("({}, {})".format(o_P2.x, o_P2.y))  # (1.0, 1.0)
```

Maar ik kan function overloading ook zo gebruiken, dat het wel kan. Zie hoofdstuk 21.

`__repr__()` en `__str()` methode

Als je de object variabele wilt printen dan krijg een geheugenadres te zien. Als je de **`__repr__()`** methode gebruikt kun je nuttige informatie krijgen. De afkorting *repr* staat voor: '*Return Printable Representation*' van het object. Het is een string. Je kunt maar één string waarde retourneren.

```
class Punt:
  def __init__(self, x=1.0, y=1.0):
    self.x = x
    self.y = y
  def __repr__(self, par='Ook wat'):
    return "({}, {}, {})".format(self.x, self.y, par)
o_P = Punt(12, 12)
print(o_P)                    # (12, 12, Ook wat)
r = o_P.__repr__('Hallo')     # Dit doet niets
print(o_P)                    # (12, 12, Ook wat)
o_P.__repr__('Hallo')         # Dit doet ook niets
print(o_P)                    # (12, 12, Ook wat)
```

Hier zie je dus dat je ook parameters kunt meegeven aan deze methode. Maar je kunt de variabele naderhand niet meer veranderen, blijkbaar. Je kan ook de **`__str()__`** methode gebruiken.

```
class Punt:
  l = "lokaal"                # Dit is een lokale of private variabel
  def __init__(self, x=1.0, y=1.0):
    self.x = x
    self.y = y
```

```
  def __str__(self, par='Dit is'):
    return "({}, {}, {}, {})".format(self.x, self.y, par, self.l)
o_P = Punt(12, 12)
print(o_P)                    # (12, 12, Dit is, lokaal))
r = o_P.__str__('Hallo')
print(r)                      # (12, 12, Hallo, lokaal))
print(o_P)                    # Doet niets; (12, 12, Dit is, lokaal))
```

Zelfde als hiervoor. Als beide methodes zijn gedefinieerd, en je print het object, dan wordt de **`__str()__`** methode genomen. Het schijnt dat als je het geheel in de command shell draait je wel een ander gedrag krijgt. Wat je ook ziet is dat je lokale variabelen kunt definiëren die alleen geldig zijn binnen het object. Die property waardes kun je ook buiten het object halen. Bij andere talen kun je een lokale variabele ook publiekelijk maken.

19.3 Methodes

De methode **`__init__()`**, **`__repr__()`**, en **`__str__()`** zijn voorgedefinieerde methodes die iedere class heeft. Daarom beginnen en eindigen ze met een dubbele underscore om dit zichtbaar te maken. Je kunt ook je eigen methodes definiëren voor een class. De functienamen volgen een bepaalde conventie:

- is_func_a: Methodes die een True/False genereren
- set_func_a: Om een waarde in het object te krijgen
- get_func_a: Om een waarde uit het object te krijgen.

Bij andere talen, C#, lijkt dit op hoe je property waardes zet.

Hier definiëren we twee methodes. Eentje om de lijn lengte te berekenen vanaf de oorsprong, en om de x en y te zetten. Kan ook met de **`__init__()`** methode. Maar gaat om het idee.

```
import math
class Punt:
  def __init__(self, x=1.0, y=1.0):
    self.x = x
    self.y = y
  def __str__(self, par=''):
    return "({}, {}, {})".format(self.x, self.y, par)
  def set_x_y_value(self, x, y):
    self.x = x
    self.y = y
  def get_line_length_origin(self):
    return math.sqrt(self.x**2 + self.y**2)
o_P = Punt(2, 2)
ll = o_P.get_line_length_origin()
print(ll, o_P.x, o_P.y)                              # 2.8284271247461903 2 2
o_P.set_x_y_value(8,8)
print(o_P.get_line_length_origin(), o_P.x, o_P.y)    # 11.313708498984761 8 8
```

19.4 Nesten van klassen

Niets staat je tegen om classes als argumenten mee te geven met classes. Je neemt een object op in een ander object. Dit is zeer gangbaar. Maar je kunt classes ook nesten. Je definieert een class binnen een class. Je kan dat een inner class of geneste class noemen.

Hieronder (volgende pagina) een voorbeeldje. **Let op:** Je ziet dat je in de **`init`** van de buitenste class, een binnen class definieert.

```
import math
class Punt:
  def __init__(self, x=1.0, y=1.0):
    self.x = x
    self.y = y
    self.Vorm = self.Vorm()  # hier definieer je binnen de buitenste class een binnen class
  def __str__(self, par=''):
    return "({}, {}, {})".format(self.x, self.y, par)
  def set_x_y_value(self, x, y):
    self.x = x
    self.y = y
  def get_line_length_origin(self):
    return math.sqrt(self.x**2 + self.y**2)
  # Hier heb je je geneste class; kan je ook zien als inner class
  class Vorm:
    def __init__(self, v=''):
      self.v= v
    def set_line_shape(self, v):
      self.v = v
    def get_line_shape(self):
      return self.v
o_P = Punt(2, 2)
ll = o_P.get_line_length_origin()
print(ll, o_P.x, o_P.y)                              # 2.8284271247461903 2 2
o_P.set_x_y_value(8,8)
print(o_P.get_line_length_origin(), o_P.x, o_P.y)    # 11.313708498984761 8 8
o_P.Vorm.set_line_shape('StippelLijn')
print(o_P.Vorm.get_line_shape())                     # StippelLijn
```

Wat je hier ziet is de opbouw van classes.

19.5 Kopieën en referenties

Omdat objecten altijd pointers zijn, en als je een object in ander object gebruikt, en je het object daar verandert, dan heeft dat effect op alle andere objecten die dat object gebruiken. Dit omdat pointers worden doorgegeven. Het is dan vaak handiger om een object niet door te geven als *pass by reference* maar als *pass by value*. Omdat Python dit soort dingen automatisch voor je doet, gebeurt het in dit geval meestal niet op de gewenste manier. In feite is *pass by value* meestal wat je wilt als het gaat om object overdracht. *Pass by value* doe je binnen python met de **`copy()`** of **`deepcopy()`** functie.
Let op: Dit doet zich niet voor binnen geneste-classes.

Hierbij een compleet (abstract) voorbeeld, alleen om duidelijk te maken dat als je een object doorgeeft als argument aan een ander object, je dit moet doen: `def __init__(self, o_A): self.oA = copy(o_A)`

```
import math
from copy import copy

class clsA:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def get_x_y(self):
     return self.x, self.y

# Hier wordt Object o_A doorgegeven als pass by reference. Automatisch.
class clsB1:
  z = 0
  def __init__(self, o_A):
    self.oA = o_A    # hier hang je het object aan een variabele, die heet dus oA binnen de class
    self.z = math.sqrt(self.oA.x**2 + self.oA.y**2)
    self.oA.x = self.z              # hier verander je object oA en dat heeft impact op
    self.oA.y = self.z              # variabele o_A buiten deze klasse.
  def get_z_line(self):
      return self.z
```

```
# Hier wordt Object o_A doorgegeven als pass by value. Gebruik copy() of deepcopy() functie.
class clsB2:
  z = 0
  def __init__(self, o_A):
    self.oA = copy(o_A)              # hier maak je een copy aan van het o_A object
    self.z = math.sqrt(self.oA.x**2 + self.oA.y**2)
    self.oA.x = self.z               # hier verander je object oA en dat heeft impact op
    self.oA.y = self.z               # variabele o_A buiten deze klasse.
  def get_z_line(self):
      return self.z

class clsC:
 def __init__(self, o_A):
   self.oA = o_A
   self.z = math.sqrt(self.oA.x**2 + self.oA.y**2)
 def get_z_line(self):
     return self.z

# 1: Hier geeft je o_A variabele door als argument, waarbij ClsB1 hem
#    als pass by reference opneemt, de standaard behandelmethode
o_A = clsA(3,4)
print(o_A.get_x_y())             # (3, 4)
# Hier geeft je o_A variabele door als argument
# En dan is o_A.x = o_B.y = 5 geworden. Dat is niet de bedoeling
print(clsB1(o_A).get_z_line())   # 5.0
print(o_A.get_x_y())             # (5.0, 5.0), dit is niet de bedoeling
print(clsC(o_A).get_z_line())    # 7.071067811865475

# 2: Hier geeft je o_A variabele door als argument, waarbij ClsB2 hem
#    als pass by value opneemt. Dat is de betere methode in dit geval
o_A = clsA(3,4)
print(o_A.get_x_y())             # (3, 4)
# Hier geeft je o_A variabele door als argument, waarby ClsB2 hem als pass by value opneemt
# o_A.x en o_B.y verandert niet
print(clsB2(o_A).get_z_line())   # 5.0
print(o_A.get_x_y())             # (3, 4) en dat is de bedoeling
print(clsC(o_A).get_z_line())    # 5.0
```

19.6 Geheugenbeheer

Objecten kunnen veel geheugen in beslag nemen met name als ze complex zijn, en je er veel van nodig hebt. Wat ook kan voorkomen is data objecten rondzweven in het geheugen en niet langer gebruikt worden. Ze nemen dan onnodig geheugen in beslag. In C reserveer je geheugen met **`alloc()`** en geef je het vrij met **`free()`**. Dus je bent als programmeur zelf verantwoordelijk voor het opruimen van ongebruikt geheugen. Binnen Python en andere talen wordt dit automatisch voor je gedaan en heet het proces die dat doet '*Garbage Collection*. ' Kortom, je hoeft je niet echt druk te maken over geheugenbeheer van objecten.

20 Operator Overloading

'*Operator overloading*' is een techniek waarbij je aan een bestaande operator een eigen gedrag kunt toekennen. Je kunt aan de **+** operator het gedrag van de **–** operator toekennen. Dat nieuw gedrag is dan geldig op de variabele waarvoor je het hebt gedefinieerd. Hieronder zie je hoe je dat doet.

```
class clsInt:                # Je maakt een nieuwe integer class
  def __init__(self, a):
    self.a = a
  def __str__(self):         # nodig om de integer af te drukken, wel geretourneerd als string
    return str(self.a)
  def __add__(self, other):  # hier herdefinieer je de + operator
     return self.a - other.a
v1 = clsInt(1)
v2 = clsInt(3)
x = v1 + v2
p = 1; q = 3; r = p + q
print(v1, v2, x)                    # 1 3 -2
print(p, q, r)                      # 1 3 4
print(type(v1), type(v2), type (x))  # <class '__main__.clsInt'> (3x)
print(type(p), type(q), type (r))    # <class 'int'> <class 'int'> <class 'int'>
```

Het werkt dus alleen op de nieuwe integer class. Ik weet niet of het mogelijk is om de bestaande integer class aan te passen. Straks ga ik dit nog een keer gebruiken om de hele integer class te overerven.
Wat er intern gebeurt, is dat als je dit doet: **`v1 + v2`**, Python de **`__add__()`** methode pakt en uitvoert wat daar staat. Dan heeft dit even uitleg nodig: **`def __add__(self, other):`**
`Self` en **`other`** zijn pointers naar een object. De namen staan vrij, maar **`self`** wordt altijd gebruikt voor het eigen object. Dan is **`other`** de pointer naar het andere object die **van dezelfde class** is als het eigen object. Dat andere object heeft dus ook een property **`a`**. Dus je trekt de **`a`** waarde van object **`other`** van het eigen object af. Als je **`v1-v2`** doet krijg je een fout, omdat de **`-`** operator (nog) niet gedefinieerd is door deze class. Dus wil je operaties uitvoeren op een zelf gedefinieerde class, dan zul je ze zelf moeten programmeren.
(**Voor de duidelijkheid:** **`other`** is een variabele naam. Als je er een class variabele van wilt maken, dan kan dat elke class zijn. In dit geval moet het dezelfde class zijn want anders kun je er geen operaties op uitvoeren. Je kunt alleen maar rekenkundige operatie op dezelfde data types uitvoeren. Of je moet casten).

De **`__add__()`** methode is een class eigen methode, net zoals **`__init__()`**. Dit betekent dat er mapping is van bestaande operators naar een class methode om een operator gedrag te herdefiniëren:

Operator	Expressie	Methode
Addition	`+`	`__add__`
Subtraction	`-`	`__sub__`
Multiplication	`*`	`__mul__`
Power	`**`	`__pow__`
Division	`/`	`__truediv__`
Floor Division	`//`	`__floordiv__`
Remainder (modulo)	`%`	`__mod__`
Bitwise Left Shift	`<<`	`__lshift__`
Bitwise Right Shift	`>>`	`__rshift__`
Bitwise AND	`&`	`__and__`
Bitwise OR	`|`	`__or__`
Bitwise XOR	`^`	`__xor__`
Bitwise NOT	`~`	`__invert__`
Negate a value	`-`	`__neg__`
	`+=`	`__iadd_`
	`-=`	`__isub__`
	`-=`	`__imul__`
	`*=`	`__idiv__`
	`/=`	`__ifloordiv__`
	`//=`	`__imod__`
	`%=`	`__ipow__`
	`**=`	`__iadd__`
	`<<=`	`__ilshift__`
	`>>=`	`__irshift__`
	`&=`	`__iand__`
	`^=`	`__ixor__`
	`|=`	`__ior__`
Negate a value	`-`	`__neg__`
Plus sign		`__pos__`
Absolute numer		`__abs__`
Conver to integer		`__int__`
Convert to float		`__float__`
Round a value		`__round__`
complex()		`__complex__`
Convert to octal		`__oct__`
Convert to hexadecima;		`__hex__`
Returns a byte string		`__bytes__ (same as __str__)`
Less than	`<`	`__lt__`
Less than or equal to	`<=`	`__le__`
Equal to	`==`	`__eq__`
Not equal to	`!=`	`__ne__`
Greater than	`>`	`__gt__`
Greater than or equal to	`>=`	`__ge__`

Operator overloading is een typisch voorbeeld van '*polymorphisme*', een concept dat een functie toestaat verschillende resultaten te produceren op basis van de types van de argumenten. *Polymorphisme* wordt vaak genoemd als een van de krachtige eigenschappen van object oriëntatie.

Een voorbeeld met alle vergelijkingsoperatoren. Je ziet dat je binnen een class de eigen class functies kunt aanroepen en zie hoe **`self`** (het *eigen-object*) en **`other`** (het *andere-object*) met elkaar interacteren.

```
import math
class Punt:
  z = 0.0                           # niet nodig, maar ging eerst voor een andere uitwerking
  def __init__(self, x, y):
    self.x = x
    self.y = y
    self.__line_length___()
  def __str__(self):
    self.__line_length___()
    return str(self.z)
  # Deze functie doet al het werk
  def __line_length___(self):
    self.z = math.sqrt(self.x**2 + self.y**2)
  def get_line_length_origin(self):
    self.__line_length___()
    return self.z
  def __add__(self, other):
    return self.z + other.z
  def __sub__(self, other):
    return self.z - other.z
  def __eq__(self, other):
    if self.z == other.z: return True
    else: return False
  def __ne__(self, other):          # ne is de de inverse van eq
    return not(self.__eq__(other))  # je hoef niet het eigen object door te geven, automatisch
  def __ge__(self, other):
    if self.z >= other.z: return True
    else: return False
  def __lt__(self, other):          # lt is de inverse van ge
    return not(self.__ge__(other))  # je hoef niet het eigen object door te geven, automatisch
  def __le__(self, other):
    if self.z <= other.z: return True
    else: return False
  def __gt__(self, other):          # gt is de inverse van le
    return not(self.__le__(other))  # je hoef niet het eigen object door te geven, automatisch

o_P1 = Punt(4, 2)
o_P2 = Punt(12,12)
# Onder geeft: 4.47213595499958 16.97056274847714 21.44269870347672 -12.49842679347756
print(o_P1, o_P2, o_P1+o_P2, o_P1-o_P2)
# Onder geeft: False True False True True False
print(o_P1 == o_P2, o_P1 != o_P2, o_P1 >= o_P2, o_P1 < o_P2, o_P1 <= o_P2, o_P1 > o_P2)
```

20.1 Sequenties

Een speciale class is de sequentie. Tuples, lists, dictionaries, en sets zijn sequenties. Zulke classes bevatten een serie elementen, die je kunt benaderen middels een index of een key. Je kunt zelf ook een sequentie class maken, door een aantal methodes die informatie over elementen van de class geven te overloaden.

`__len__()`	Implementeert de **`len()`** functie, die aangeeft hoeveel elementen het object bevat.
`__getitem__()`	Implementeert het retourneren van een element. **`IndexError`** en **`KeyError`** moeten ook geïmplementeerd worden. Als key een index is, dan moet voor een complete implementatie ook zogenaamde '*slice objects*' ondersteund worden. Voorbeeld: **`Waarde = Obj[<key|index>]`**
`__setitem__()`	Implementeert het toekennen van een waarde aan een element van het object. Voorbeeld: **`Obj[<key|index>] = Waarde`**
`__delitem__()`	Implementeert het verwijderen van een element uit het object als het gereserveerde woord **`del`** wordt gebruikt. Voorbeeld: **`del Obj[<key|index>]`**
`__contains__()`	Bepaalt of het element bestaat als je het gereserveerde woord **`in`** gebruikt. Voorbeeld: **`in Obj["Element Naam"]`**

`__missing__()` Wordt aangeroepen door **`__getitem__()`**, met de key of index als argument, als deze key of index niet verwijst naar een element dat in het object bestaat. Deze methode is vooral bedoeld voor geneste classes van de Python dictionary.

Hieronder een eenvoudig voorbeeld. Een lijst met letters; een alfabet. Je ziet dat er ook een paar niet sequentiële methodes worden gedefinieerd, append en remove.

```
class clsAlfabet:
  def __init__(self):
    self.l = []
  def __str__(self):
    return str(self.l)
  def append(self, item):
    self.l.append(item)
  def remove(self, item):
    self.l.remove(item)
  def sort(self):
    self.l.sort()
  def __setitem__(self, i, item):  # i = index, item = item value
    self.l[i] = item
  def __delitem__(self, i):
    del self.l[i]
  def __getitem__(self, i):
    return self.l[i]
  def __len__(self):
    return len(self.l)
  def __contains__(self, item):
    return item in self.l

o_A = clsAlfabet()
o_A.append("A"); o_A.append("B"); o_A.append("C")
print("a" in o_A, "A" in o_A)                       # False True
print(o_A.l, o_A[2], len(o_A), o_A.__len__())       # ['A', 'B', 'C'] C 3 3
del o_A[0]
o_A.remove("B")
print(o_A.l)                                        # ['C']
o_A.append("A")
o_A[0] = "Z"
print(o_A.l)                                        # ['Z', 'A']
o_A.append("B")
print(o_A.l)                                        # ['Z', 'A']
o_A.sort()
print(o_A.l)                                        # ['Z', 'A', 'B']
for c in o_A:                                       # hier print je alles
  print(c)
```

21 Function overloading

Omdat Python '*soft typing*' doet speelt '*function overloading*' hier minder dan bij talen die aan '*hard typing*' doen. Hieronder een voorbeeld uit C# om te laten zien wat wordt bedoeld.

```
private Int32 OverloadFunctie() {
  return 0;
}
private Int32 OverloadFunctie(Int32 pA) {
  return pA;
}
private Int32 OverloadFunctie(string pA, string pB) {
  return Convert.ToInt32(pA) + Convert.ToInt32(pA);
}
private void OmTeLatenZien() {
  Int32 a,b,c;
  a = OverloadFunctie();
  b = OverloadFunctie(10);
  c = OverloadFunctie("10", "10");
}
```

Op zich werkt dit prettig. De Python oplossing is zoiets als dit:

```
def OverloadFunctie(pA=0, pB=0):
  if (isinstance(pA, int) == True and isinstance(pB, int) == True):
    return pA+pB;
  elif (isinstance(pA, str) == True and isinstance(pB, str) == True):
    return int(pA)+int(pA)
  else: None

a = OverloadFunctie()
b = OverloadFunctie(10)
c = OverloadFunctie("10", "10")
print(a, b, c)                          # 0 10 20
```

Maar wat als je een object mee geeft? Dat komt er **None** uit. Niet persé fout, maar binnen C# kun je deze voorbeeldfunctie met een object als argument niet aanroepen, je krijg een foutmelding in je IDE, en als je die laat zitten, als je gaat compileren. De methodes hebben elk zo hun voor- en nadelen.

22 Events

Met events kun je (asynchroon) communiceren tussen classes. Meer in zijn algemeenheid kun je een functie aanroepen die wordt getriggerd door wat je maar wilt. Het is niet exclusief voor classes. Omdat Python events niet standaard ondersteunt, lijkt het, moet je een module installeren.

1. Hier staat een events module: https://pypi.org/project/Events/
2. Download Events-0.3.tar.gz naar C:\Python
3. Open een command window en ga naar C:\Python
4. Voer het volgende commando uit: "C:\Program Files\Python36\Scripts\pip" install --user events
   (In essentie moet **pip install events** voldoende zijn).

(Dit stappenplan is heel afhankelijk van wat je Python distributie is. Bij Visual Studio staat het hele pip gebeuren onder: Tools -> Python -> Python Environmenst).

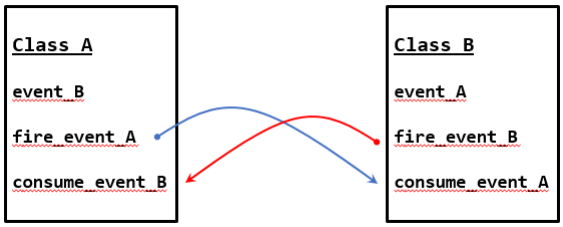
Hier een eenvoudig voorbeeldje. Staat in de documentatie van de betreffende **events** module.

```
import events
print(events.__all__)

def event_something(reason):
  print("changed: %s" %reason)

def raise_event(e, s):
  e.on_something(s)

e = events.Events()
e.on_something += event_something      # event subscribing
raise_event(e, "Zo kan het!")          # event firing methode 1. changed: Zo kan het
e.on_something("Maar zo kan het ook!") # event firing methode 2. changed: Maar zo kan het ook!
e.on_something -= event_something      # even de-subscribing
e.on_something("Maar zo kan het ook!") # gebeurt niets! Wat correct is!
```

Uitleg:

1. **on_something** is de naam van een event die is verbonden met functie **event_something()**. Laat je niet van de wijs brengen door de benaming. Want dit werkt ook **e.a += b**. Dan is **a** de event naam en **b** de event functie. Je hebt nu een '*eventhandler*' gemaakt. Dit heet ook wel een callback functie.
2. Dan vuur je de gebeurtenis af; '*event firing*'. Ik heb een tussenstap, maar rechtstreeks is gewoon **e.on_something()**. De functie die dan is verbonden met het event wordt uitgevoerd.
3. Je kunt ook '*de-subscribing*' doen. Je koppelt de event los van de functie. De-subscribing is tamelijk courant. Als je wilt dat jouw applicatie niet op muis-clicks moet reageren dan zet je die tijdelijk uit. (Dit is niet hetzelfde als muis-clicks voor je systeem uitzetten).

Je kunt dus een oneindig aantal events maken die allemaal zijn verbonden met dezelfde eventhandler. In de praktijk maak je verschillende events die verbonden zijn met allemaal verschillende eventhandlers. Nu volgt een wat uitgebreider voorbeeld om te zien hoe je events tussen classes kan gebruiken.



Class_A praat met **Class_B** en vice versa. Het is een vorm van '*state*' programming. Het event wat **Class_A** afvuurt, moet **Class_B** opvangen. Het event wat **Class_B** afvuurt, moet **Class_A** opvangen.
Dat betekent dat elk object zicht bewust moet zijn van de ander. Omdat event namen al snel een warboel wordt, wat vuurt wat af, en wie consumeert wie, moet je de namen zo strak mogelijk houden. In ieder geval, in **Class_A** definieer je de eventhandling die komt van **B**. Dus daar maak je **event_B**. De class methode **work()** doet dan het feitelijke werk. Het is handig om eventhandling zo kort mogelijk te houden, alleen afvuren en opvangen en de feitelijke werkzaamheden door iets anders te laten plaatsvinden.

```
class ClsA():
  def __init__(self, n):
    self.n = n
    self.e = events.Events()
    self.e.eventB += self.consume_event_B
  def class_b(self,other):
    self.other = other
  def fire_event_A(self):
    self.other.e.eventA()
  def consume_event_B(self):
    self.work()
  def work(self):
    while True:
      x = input('%s: Geef waarde: ' %self.n)
      if (x == '0'): break
      if (x == '1'):
        self.fire_event_A()
        print("A")
        break;
```

```
class ClsB():
  def __init__(self, n):
    self.n = n
    self.e = events.Events()
    self.e.eventA += self.consume_event_A
  def class_a(self, other):
    self.other = other
  def fire_event_B(self):
    self.other.e.eventB()
  def consume_event_A(self):
    self.work()
  def work(self):
    while True:
      x = input('%s: Geef tekst: ' %self.n)
      if (x == '0'): break
      if (x == '1'):
        self.fire_event_B()
        print("B")
        break
```

```
o_A = ClsA("A Class A")
o_B = ClsB("B Class B")
o_A.class_b(o_B)
o_B.class_a(o_A)
o_A.work()
```

Probleem met deze code is dat nadat je de event afvuurt, alle code daarna wel in de wachtrij, op de stack, komt. Dat wil je niet echt. Het komt door de **while** loop, denk ik. Ik bedoel dit stukje code:

```
self.fire_event_A()
print("A")
break;
```

Als het event is afgevuurd en afgerond dan kom je niet meteen bij **print("A")** uit. Je krijgt dit:

```
A Class A: Geef waarde: 1
B Class B: Geef tekst: 1
A Class A: Geef waarde: 1
B Class B: Geef tekst: 1
A Class A: Geef waarde: 1
B Class B: Geef tekst: 1
A Class A: Geef waarde: 0
B
A
B
A
B
A
```

Als ik volgende code neem heb ik nog steeds hetzelfde probleem.

```
class ClsA():
  def __init__(self, n):
    self.n = n
    self.e = events.Events()
    self.e.eventB += self.consume_event_B
  def class_b(self,other):
    self.other = other
  def fire_event_A(self):
    self.other.e.eventA()
  def consume_event_B(self):
    self.work()
  def work(self):
    x = input('%s: Geef waarde: ' %self.n)
    if (x == '0'):
      self.e.eventB -= self.consume_event_B
    elif (x == '1'):
      self.fire_event_A()
      print("A")
    else:
      self.e.eventB()
```

```
class ClsB():
  def __init__(self, n):
    self.n = n
    self.e = events.Events()
    self.e.eventA += self.consume_event_A
  def class_a(self, other):
    self.other = other
  def fire_event_B(self):
    self.other.e.eventB()
  def consume_event_A(self):
    self.work()
  def work(self):
    x = input('%s: Geef tekst: ' %self.n)
    if (x == '0'):
      self.e.eventA -= self.consume_event_A
    elif (x == '1'):
      self.fire_event_B()
      print("B")
    else:
      self.e.eventA()
```

Het is wel zo dat ik in de class blijf hangen waar ik kom te zitten, ik zit in een loop, lus, ook om een beetje duidelijk te maken wat er gebeurt. Maar op zich lijkt dit wel ok. De code moet wel doorgang vinden. Hieronder iets anders gecodeerd om duidelijk te laten zien wat er gebeurt:

```
class ClsA():
  def __init__(self, n):
    self.n = n
    self.e = events.Events()
    self.e.eventB += self.consume_event_B
  def class_b(self,other):
    self.other = other
  def fire_event_A(self):
    self.other.e.eventA()
  def consume_event_B(self):
    self.work()
  def work(self):
    x = input('%s: Geef waarde: ' %self.n)
    self.e.eventB -= self.consume_event_B
    self.fire_event_A()
    print("A")
```

```
class ClsB():
  def __init__(self, n):
    self.n = n
    self.e = events.Events()
    self.e.eventA += self.consume_event_A
  def class_a(self, other):
    self.other = other
  def fire_event_B(self):
    self.other.e.eventB()
  def consume_event_A(self):
    self.work()
  def work(self):
    x = input('%s: Geef tekst: ' %self.n)
    self.e.eventA -= self.consume_event_A
    self.fire_event_B()
    print("B")
```

Het gaat met name om dit:

```
def work(self):
  x = input('%s: Geef waarde: ' %self.n)
  self.e.eventB -= self.consume_event_B
  self.fire_event_A()
  print("A")
```

In dit geval moet je de-subscriben, want anders kom je in een oneindige loop te zitten. Ook moet je het doen voordat je **`fire_event_A()`** doet, want je code komt nooit voorbij **`self.fire_event_A()`** als je geen event de-subscribing doet.
Als laatste: Je geeft hier een class door als *passed by reference*. Zie hier voor *pass by value*.

Hoofdstuk 30, PostgreSQL, laat ook een class zien met event handling.

23 Overerving

Overerving ('*inheritance*') is een mechanisme om een nieuwe class te baseren op een bestaande class, door alleen de verschillen tussen de twee aan te geven. Bij de definitie van een nieuwe class, zet je tussen haakjes de naam van een andere class. De nieuwe class erft dan alle attributen en methodes van de andere class; ze zijn automatisch opgenomen in de nieuwe class.

Hier een voorbeeldje op basis van de eerder gebruikt **`clsInt`** class. We maken nu een float, clsFloat, class.

```
class clsInt:                   # Je maakt een nieuwe integer class
  def __init__(self, a):
    self.a = a
  def __str__(self):            # nodig om de integer af te drukken, wel geretourneerd als string
    return str(self.a)
  def __add__(self, other):     # hier herdefinieer je de + operator
     return self.a - other.a

class clsFloat(clsInt):         # Inheritance
  pass

v1 = clsInt(1); v2 = clsInt(3)
v3 = clsFloat(1.0); v4 = clsFloat(3.0)
print(v1, v2, v1+v2)                    # 1 3 -2
print(v3, v4, v3+v4)                    # 1.0 3.0 -2.0
```

De nieuwe class (clsFloat) wordt de '*subclass*' genoemd, en de class waarvan de subclass wordt afgeleid (clsInt) heet de '*superclass*' (soms '*ouder-class*' genoemd).

23.1 Meervoudige overerving

Je kunt meer dan één class doorgeven bij *inheritance*. Dat heet '*meervoudige overerving*'. Zie hieronder hoe dat moet. Gewoon alle superclasses tussen haakjes zetten.

```
class clsA:
  pass
class clsB:
  pass
class clsC:
  pass
class ClsZ(clsA, clsB, clsC):
  pass
```

Meervoudige overerving maakt de boel complex omdat Python moet uitzoeken uit welke class de methodes komen. Ook kan de programmeer het overzicht kwijtraken. Veel object georiënteerde talen ondersteunen meervoudige overerving niet eens, en die dat wel doen zetten er vaak waarschuwingen bij.

23.2 Uitbreiden en overschrijven

Een subclass breidt je uit met nieuwe methodes door die in de subclass te definiëren. Als de methodes in de *superclass* al bestaan worden ze overgeschreven. Hier een aanpassing op de clsFloat class.

```
class clsFloat(clsInt):
  def __add__(self, other):     # overschrijven
    return self.a + other.a
  def __mul__(self, other):     # nieuwe methode
    return self.a * other.a

v1 = clsFloat(2.0)
v2 = clsFloat(3.0)
print(v1, v2, v1+v2, v1*v2)     # 2.0 3.0 5.0 6.0
```

Er zijn twee manieren om de methode van een andere class aan te roepen. Als een methode wordt overschreven, kun je de oorspronkelijke (*superclass*) methode altijd nog expliciet aanroepen via:

`'class call' methode`	Je roept een methode uit een andere superclass aan met de syntax: **`<classnaam>.<methode>()`**
`super() methode`	Niet nodig om de *superclass* te definiëren. Je roept meteen de methode uit de *superclass* aan. Syntax: **`Super().<methode>()`** Bij meervoudige overerving wordt de methode uit de eerste superclass gepakt.

Hieronder een voorbeeld met meervoudige overerving om te laten zien wat de **`super()`** functie pakt.

```
# Drie classes die allemaal een add functie hebben die verschillend zijn.
class clsA:
  def f_add(self, a, b):
    self.a = a; self.b = b
    return self.a - self.b
```

```
class clsB:
  def f_add(self, a, b):
    self.a = a; self.b = b
    return self.a * self.b
class clsC:
  def f_add(self, a, b):
    self.a = a; self.b = b
    return self.a ** self.b

# Hier een class met meervoudige overerving. Dus clsZ heeft drie superclasses
class clsZ(clsA, clsB, clsC):
  def f_add(self, a, b):
    self.a = a;  self.b = b
    return self.a + self.b
  def fA(self, a, b):
    self.a = a; self.b = b
    return super().f_add(a, b)       # hier moet je self, je eigen object weglaten
  def fB(self, a, b):
    self.a = a; self.b = b
    return clsB.f_add(self, a, b)
  def fC(self, a, b):
    self.a = a; self.b = b
    return clsC.f_add(self, a, b)

o = clsZ()
print(o.f_add(3,4));  #  7 (3+4)
print(o.fA(3,4));     # -1 (3-4), dit is dus de class A functie
print(o.fB(3,4));     # 12 (3*4)
print(o.fC(3,4));     # 81 (3**4)
```

Als dit doet: `class clsZ(clsC, clsB, clsC):` en de rest is hetzelfde, dan levert `print(o.fA(3,4))` waarde 81 op. Nu wordt immers de `clsC` methode gepakt. Omdat de *class call* geen reguliere aanroep van een methode is, moet `self` als argument meegegeven worden. Bij `super()` is dat niet nodig.

23.3 Interfaces

Een interface is een class die specifieke attributen en methodes vastlegt zonder een implementatie te geven van die methodes. Het idee is dat je gedwongen bent een subclass af te leiden die de methodes implementeert. Je kunt dan, zelfs als je nog niet weet welke subclasses allemaal gemaakt gaan worden, toch al functies bouwen die methodes aanroepen van de interface class, met als aanname dat deze functies worden aangeroepen met instanties van subclasses waarvoor de methodes ingevuld zijn.

```
class clsInterfaceA:
  def __init__(self, pVal1 = None, pVal2 = None):
    self.a = pVal1
    self.b = pVal2
  def DoeIets(self):
    NotImplemented
  def DoeNiets(self):
    NotImplemented

# Hier zie je dat je interface class wel kunt gebruiken
o_I = clsInterfaceA()
print(o_I.DoeIets(),o_I.DoeNiets()) # None None (heef niets te maken met __ini__)

# Hier implementeer je de interface class tot iets.
class clsImplementatie(clsInterfaceA):
  def DoeIets(self):
    return self.a + self.b
  def DoeNiets(self):
    self.b = self.a
    return self.a - self.b

o_I = clsImplementatie(5,9)
print(o_I.DoeIets(),o_I.DoeNiets()) # 14  0
```

De interface class is dus een *superclass* geworden. Volgens mij is er niets op tegen om in de interface class uitgewerkte methodes op te nemen.

24 Iteratoren en Generatoren

Iteratoren maken het mogelijk dat je een zelf-gedefinieerde class kunt gebruiken in **`for ... in ...`** statements. Generatoren zijn een eenvoudige manier om iteratoren te maken.

24.1 Iteratoren

Het **`for ... in ...`** commando kan op allerlei verschillende manieren gebruikt worden. Een overzicht.

```
for i in [1,2,3,4]:                 # lists
  print(i, end=" ")
  print()
for i in ("pi", 3.14, 22/7):        # tuples
  print(i, end=" ")
  print()
for i in range(3, 11, 2):           # integer range
  print(i, end=" ")
  print()
for c in "Hallo":                   # karakter uit string
  print(c, end=" ")
  print()
for key in {"appel":1, "banaan":3}: # dictionaries
  print(key, end=" ")
```

Lists, strings, en dictionaries zijn allen '*iterabele*' ('*iterable*'), wat betekent dat ze gebruikt mogen worden in **`for ... in ...`** statements. Vele andere objecten kunnen ook als iterabelen gebruikt worden. Zoals instanties van je eigen classes. Maar die moet je dan wel als zodanig coderen.

Een '*iterator*' is een object dat een nieuw element retourneert iedere keer dat je de standaardfunctie **`next()`** aanroept met het object als argument. Als het object niks meer heeft dat geretourneerd kan worden, genereert het een **`StopIteration`** exception. Het optioneel tweede argument van **`next()`** onderdrukt deze exception. Je kunt van iedere iterabele een *iterator* object maken met de standaardfunctie '**`iter()`**'.
Hier een voorbeeldje om te werking van **`next()`** aan te geven.

Zonder iter()

```
L = ["A", "B", "C"]
print("n:", next(L))  # geeft een foutmelding
for item in L:        # allemaal afgedrukt
  print(item)
for item in L:        # allemaal afgedrukt
  print(item)
```

Met iter()

```
L = iter(["A", "B", "C"])
print("n:", next(L)) # A wordt afgedrukt
for item in L:       # B en C geprint
  print(item)
for item in L:       # hier gebeurt niets!
  print(item)
```

Nu is een list van zichzelf iterabel, maar dan zonder de **`next()`**. Daar blijkt je dus niet zoveel aan te hebben. Want je kunt de list daarna niet meer benaderen als je er helemaal doorheen bent gegaan. Je moet blijkbaar de index resetten. Maar hoe kon ik niet zo snel achterhalen. Onderstaande werkt, ook een vorm van resetten.

```
L = ["A", "B", "C"]
Li = iter(L)
for item in Li: print(item)
Li = iter(L)                    # opnieuw initialiseren, resetten
for item in Li: print(item)
```

24.2 Iterabel object

Een iterabel object moet de volgende twee methodes bevatten:
`__iter__()` Methode die de iterabele (meestal het object zelf) retourneert.
`__next__()` Methode die toegang geeft tot alle elementen die het object bevat, en een **`StopIteration`** exception genereert als er geen meer zijn. Voor een **`for ... in ...`** loop stopt de loop.

Er zijn drie manieren om iterabele objecten te maken:
1. Iedere keer als **`__next__()`** wordt aangeroepen wordt het geretourneerde element verwijdert. De iterabele wordt dus kleiner. Of dit handig is?
2. Je houdt een index bij. Iedere keer als **`__next__()`** wordt aangeroepen wordt het element geretourneerd en de index verhoogt. Als de index buiten de grenzen van de sequentie komt, wordt **`StopIteration`** gegenereerd. Op deze manier maak je een herbruikbare iterabele. Je moet wel een methode toevoegen die de index weer op nul zet. Maar kan ook automatisch.

3. Maakt een object dat het volgende element berekent wanneer **`__next__()`** wordt aangeroepen. Een dergelijke iterabele kan eindig zijn, maar kan ook een oneindige aantal elementen retourneren. Je kunt de iterabele ook opnieuw laten beginnen als je een methode toevoegt om de berekening opnieuw te initialiseren. Van deze methode heb ik geen voorbeeld opgenomen. Zie oorspronkelijk boek.

In de onderstaande twee voorbeelden worden ook wat technieken toegepast eerder beschreven. Het is wel een kwestie van goed kijken wat er precies gebeurt; **`l`** is de list variabele, **`i`** de index en **`item`** de list-waarde.

Manier 1

```
class clsAlfabet:
  def __init__(self):
    self.l = []
  def __str__(self):
    return str(self.l)
  def append(self, item):
    self.l.append(item)
  def remove(self, item):
    self.l.remove(item)
  def sort(self):
    self.l.sort()
  def __setitem__(self, i, item):
    self.l[i] = item
  def __delitem__(self, i):
    del self.l[i]
  def __getitem__(self, i):
    return self.l[i]
  def __len__(self):
    return len(self.l)
  def __contains__(self, item):
    return item in self.l
  def __iter__(self):
    return self
  def __next__(self):
    i=len(self.l)
    if (i > 0):
      RetVal = self.l[0]
      del self.l[0]
      return RetVal
    raise StopIteration()

o_A = clsAlfabet()
o_A.append("A"); o_A.append("B");
o_A.append("C")
print("n :", next(o_A))  # A
for c in o_A:
  print("1:", c)         # B C
for c in o_A:     # Niets; Iterabele is leeg
  print("2:", c)
```

Manier 2

```
class clsAlfabet:
  def __init__(self):
    self.l = []
    self.i = -1 # index telling begint bij 0
  def __str__(self):
    return str(self.l)
  def append(self, item):
    self.l.append(item)
  def remove(self, item):
    self.l.remove(item)
  def sort(self):
    self.l.sort()
  def index(self, item = None):
    if (item == None): return self.i
    else: return self.l.index(item)
  def __setitem__(self, i, item):
    self.l[i] = item
  def __delitem__(self, i):
    del self.l[i]
  def __getitem__(self, i):
    return self.l[i]
  def __len__(self):
    return len(self.l)
  def __contains__(self, item):
    return item in self.l
  def __iter__(self):
    return self
  def __next__(self):
    if self.i < len(self.l)-1:
      self.i += 1
      return self.l[self.i]
    self.i = -1   # hier reset je automatisch
    raise StopIteration ()
  def reset(self):
    self.i = -1

o_A = clsAlfabet()
o_A.append("A"); o_A.append("B"); o_A.append("C")
# Als itereert op index is alles OK
for i in range(0,len(o_A)):
  print("3:", o_A[i])
for i in range(0,len(o_A)):
  print("4:", o_A[i])

print("n1 :", next(o_A))  # A
print("n2 :", next(o_A))  # B
print("n3 :", next(o_A))  # C
print("n4 :", next(o_A, "EINDE"))  # EINDE.
print(o_A.index())        # 2
print(o_A.index("A"))     # 0
for c in o_A:             # leeg, als n4 niet gedaan
  print("5:", c)
o_A.reset()               # reset
for c in o_A:             # Alle drie
  print("6: ", o_A.index(), c)
for c in o_A:             # Alle drie
  print("7: ", o_A.index(), c)
```

Nog iets over de **`next()`**. Als je bij het eind bent, **`n4`** in het voorbeeld, en je geeft geen eigen "EINDE" op, dan klapt het programma. Het statement **`print("5:"..)`** is niet leeg in het geval uitvoeren van **`print("n4"...)`**. De index wordt immers automatisch gereset.

24.3 Gedelegeerde iteratie

Een iterabele wordt dus gemaakt door het aanroepen van de **`__iter__()`** methode voor een object, dat zichzelf retourneert. Het kan ook anders. Een iterabele mag de iteratie delegeren aan een ander object, dat wordt aangemaakt door de iterabele en geretourneerd wordt als de **`__iter__()`** methode wordt aangeroepen. Dat heet '*delegate an iterable*'. (Ik kreeg dit trouwens niet goed aan de praat).

```
class FiboIterable:
  def __init__(self, seq):
    self.seq = seq
  def __next__(self):
    if len(self.seq) > 0:
      return self.seq.pop(0)
    raise StopIteration ()

class Fibo:
  def __init__(self, maxnum=1000):
    self.maxnum = maxnum
  def __iter__(self):
    nr1 = 0
    nr2 = 1
    seq = []
    while nr2 <= self.maxnum:
      nr3 = nr1 + nr2
      nr1 = nr2
      nr2 = nr3
      seq.append(nr1)
    return FiboIterable(seq)

fseq = Fibo()
print("n :", next(fseq))  # werkt niet!
for n in fseq:
  print(n, end=" ")
print()
for n in fseq:
  print(n, end=" ")
print()
```

Deze aanpak heeft een aantal voordelen:

- Je kunt meerdere instanties van de iterabele parallel aan elkaar uitvoeren zonder er expliciet meer dan één te maken. (Omdat ze automatisch worden gemaakt wanneer dat nodig is, dus als **`for ... in ...`** gebruikt wordt).
- Je hoeft geen methode **`reset()`** aan te roepen om weer van voor af aan te beginnen; iedere nieuwe aanroep van de iterabele begint weer van voor af aan.
- De gedelegeerde iterabele wordt automatisch uit het geheugen verwijderd wanneer er geen elementen meer in zitten; *garbage collection*

24.4 Functies

Een drietal functies; **`zip()`**, **`reversed()`** en **`sorted()`**.

`zip()` functie

Je kunt tuples maken die de elementen bevatten van meerdere iterabelen middels de standaardfunctie **`zip()`**. Een zip-object is een iterator, dat wil zeggen, je kunt het zip-object zelf niet printen, maar je kunt de elementen van het object doorlopen via een **`for ... in ...`** constructie. Het n^{th} element van het zip-object bestaat uit de n^{th} elementen van ieder van de iterabelen die als argumenten gebruikt worden. Als deze iterabelen van ongelijke lengte zijn, dan is de lengte van het zip-object gelijk aan de kortste lengte van de argumenten. Om een eenvoudig voorbeeld te geven:

```
z = zip([1,2,3], [4,5,6], [7,8,9])
for x in z:
  print(x)
Output:
(1, 4, 7)
(2, 5, 8)
(3, 6, 9)
```

```
z = zip([1,2,3,4,5],[['a', 'b'], [1,2,3,4,5], 99])
for x in z:
  print(x)
Output:
(1, ['a', 'b'])
(2, [1, 2, 3, 4, 5])
(3, 99)
```

Als je geneste-lists gebruikt krijg je toch wel een wat aparte uitvoer. Omdat de kortste iterabele uit drie (geneste) elementen bestaat, krijg je drie tuples terug en vervalt 4 en 5 uit de eerste iterabele. Hieronder wordt een range, een iterator, en een list comprehension gezipt.

```
class Dubbel:
  def __init__(self):
    self.seq = [2*x for x in range(1, 11)]
  def __iter__(self):
    return self
  def __next__(self):
    return self.seq.pop(0)

seq = zip(range(1,11), Dubbel(), [3*x for x in range(1,11)])
for x in seq:
  print(x)
```

reversed() functie

Maakt vanuit een iterabele een iterator die de elementen van de iterator in omgekeerde volgorde verwerkt. Niet alle iterabelen kunnen omgekeerd worden, maar de iterabelen die onderdeel zijn van standaard Python (zoals lists) kunnen het in ieder geval.

```
L = ["A", "B", "C", "D"]
for item in reversed(L):
  print(item, end=" ")  # D C B A
```

sorted()

Maakt vanuit een iterabele een iterator die de elementen van de iterator gesorteerd verwerkt.

```
L = ["A", "X", "C", "D"]
for item in sorted(L):
  print(item, end=" ")  # A C D X
```

24.5 Generatoren

Een generator is een functie die het gedrag van een iterabel object emuleert. Het is makkelijker om een generator te bouwen dan het is om een iterabele te bouwen. Generatoren zijn gebaseerd op het gereserveerde woord **yield**. Als **__next__()** wordt aangeroepen voor een generator, wordt de functie uitgevoerd totdat **yield** wordt aangetroffen, en dan wordt de waarde geretourneerd die met de **yield** geassocieerd is. Daar aangekomen, "wacht" de functie totdat **__next__()** opnieuw wordt aangeroepen, waarna de functie verder gaat waar hij gebleven was totdat **yield** weer gevonden wordt. **StopIteration** wordt automatisch gegenereerd wanneer de functie eindigt. Je hoeft niet expliciet **__next__()** en/of **__iter__()** te definiëren voor een generator. Een generator bestaat bij gratie van het feit dat de functie het gereserveerde woord **yield** bevat, en het geassocieerde iterabele object wordt automatisch door Python gemaakt, inclusief correcte implementaties voor **__next__()** en **__iter__()**.

```
def func_range(c):
  for i in range(c):
    yield i                 # dit maakt blijkbaar een iterator

gen_A = func_range(10)
print(next(gen_A,"EINDE"))  # 0
print(next(gen_A,"EINDE"))  # 1
for item in gen_A:
  print(item, end="-")      # 2-3-4-5-6-7-8-9-
print()
gen_A = func_range(10)      # je moet wel opnieuw initialiseren
for item in gen_A:
  print(item, end="-")      # 0-1-2-3-4-5-6-7-8-9-
```

24.6 Generator expressies

In paragraaf 11.8 wordt het concept '*list comprehension*' beschreven. Omdat een list in een iterator veranderd kan worden, en daarom in een generator, heeft Python een gelijksoortig concept geïntroduceerd voor generatoren; '*generator expressies*'. De syntax van een *generator expressie* is gelijk aan de syntax van een *list comprehension*, behalve dat er ronde in plaats van vierkante haken omheen staan. De volgende *generator expressie* retourneert alle kwadraten tot en met 100.

```
ge_A = (x**2 for x in range(11))
for x in ge_A:
  print(x, end=" ")           # 0-1-2-3-4-5-6-7-8-9-0 1 4 9 16 25 36 49 64 81 100
```

Als je de twee buitenste haakjes in de generator expressie wijzigt in vierkante haken, wordt de code uitgevoerd met **ge_A** als resultaat van een list comprehension.

```
ge_A = [x**2 for x in range(11)]
for x in ge_A:
  print(x, end=" ")           # 0-1-2-3-4-5-6-7-8-9-0 1 4 9 16 25 36 49 64 81 100
print()
```

Met list comprehension wordt de hele list in één keer gegenereerd, terwijl met een generator expressie de elementen pas gegenereerd worden wanneer ze nodig zijn. Een generator expressie neemt dus minder geheugen in beslag

24.7 itertools module

De **itertools** module bevat een verzameling functies die geavanceerde manipulatie van iteratoren mogelijk maken. Hier worden een aantal besproken. Wel eerst dit doen: **import itertools as it**

chain()

Neemt twee of meer iterabelen as argumenten en functioneert als een iterabele die de samenstellende iterabelen één voor één afwerkt.

```
ge = it.chain([1,2,3], [11,12,13,14], [x**2 for x in range(1,6)]) # drie iterabelen
for item in ge:
  print(item, end=" ") # 1 2 3 11 12 13 14, en hier dan de kwadraten 1 4 9 16 25
```

zip_longest()

Zelfde als **zip()**, maar maakt een iterabele met een lengte gelijk aan die van de langste van de samenstellende iterabelen. Je kunt een **fillvalue=argument** opgeven om aan te geven welke waarde er op de lege plekken moet komen te staan.

```
ge = it.zip_longest("appel", "framboos", "banaan", fillvalue=" ")
for item in ge:
  print(item)
```

```
('a', 'f', 'b')
('p', 'r', 'a')
('p', 'a', 'n')
('e', 'm', 'a')
('l', 'b', 'a')
(' ', 'o', 'n')
(' ', 'o', ' ')
(' ', 's', ' ')
```

product()

Maakt een iterabele die alle elementen produceert van het Cartesisch product van de iterabelen die als argumenten zijn meegegeven. (Een Cartesisch product is het vermenigvuldigen van twee tabellen waarbij alle combinaties worden gemaakt).

```
ge = it.product([1,2,3], "AB", ["appel", "banaan"])
for item in ge:
  print(item)
```

```
(1, 'A', 'appel')
(1, 'A', 'banaan')
(1, 'B', 'appel')
(1, 'B', 'banaan')
(2, 'A', 'appel')
(2, 'A', 'banaan')
(2, 'B', 'appel')
(2, 'B', 'banaan')
(3, 'A', 'appel')
(3, 'A', 'banaan')
(3, 'B', 'appel')
(3, 'B', 'banaan')
```

permutations()

Krijgt een iterabele als argument, en een optioneel tweede argument dat een lengte aangeeft. Het maakt een iterabele die alle permutaties produceert die combinaties zijn van elementen van de meegegeven iterabele, met de gegeven lengte. Als geen lengte wordt meegegeven, genereert het alle permutaties van alle elementen. Als de meegegeven iterabele elementen dubbels bevat, zullen er kopieën van permutaties zijn.

```
ge = it.permutations([1,2,3], 2)
for item in ge:
 print(item)
```

```
(1, 2)
(1, 3)
(2, 1)
(2, 3)
(3, 1)
(3, 2)
```

combinations()
Krijgt een iterabele als argument, en een tweede argument dat een lengte aangeeft. Het maakt een iterabele die combinaties produceert van elementen van het eerste argument, met de gegeven lengte. De elementen van de combinaties staan in de volgorde zoals ze in de originele iterabele stonden. Dubbels kunnen ontstaan.

```
ge = it.combinations([1,2,3,1,2], 2)
for item in ge:
  print(item)

# Dit levert alleen zichzelf op:
ge = it.combinations([1,2,3,1,2], 5)
for item in ge:
  print(item)
```

```
(1, 2)
(1, 3)
(1, 1)
(1, 2)
(2, 3)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
(1, 2)
```

combinations_with_replacement()
Zelfde als **combinations()**, maar ieder element van de iterabele mag meerdere keren gebruikt worden.

```
ge = it.combinations_with_replacement([1,2,3], 2)
for item in ge:
  print(item)
print()
```

```
(1, 1)
(1, 2)
(1, 3)
(2, 2)
(2, 3)
(3, 3)
```

25 Command Line Verwerking

Programma's die in de command shell draaien worden ook wel batch programma's genoemd. Een batch programma is een lijst van statements, een command line, die uitgevoerd wordt. Zo een command line statement kun je ook één voor één intypen en afzonderlijk, achter elkaar uitvoeren, maar je kunt ze ook in een (batch) programma stoppen. Door een batch programma argumenten mee te geven kun je bepaalde logica erin stoppen. De bedoeling van een batch programma is dat deze minimaal communiceert met de gebruiker. Windows kent de zogenaamde Windows Powershell, die krachtiger is dan de normale Windows command shell, en een eigen object georiënteerde programmeeromgeving heeft. Unix is altijd command line georiënteerd geweest, en is van zichzelf al krachtig.

25.1 Command line argumenten

Op de command line kun je een Python programma starten met een aantal argumenten. Syntax:
python <programmanaam>.py <argument_1> <argument_2> ... <argument_n>
De argumenten zijn van elkaar gescheiden door middel van spaties en mogen van alles zijn. Als je een argument hebt dat zelf een spatie bevat, moet je het tussen dubbele aanhalingstekens zetten. Als het argument zelf ook een dubbele spatie heeft is het command shell afhankelijk hoe te beschermen. Het programma verwerkt de argumenten en voert op basis daarvan code stappen uit.

sys.argv
De argumenten worden opgeslagen in een voor gedefinieerde list die **sys.argv** heet. De **sys** module moet wel zijn geïmporteerd. Het is een list van strings, waarbij iedere string één van de command line argumenten is. De lijst bevat altijd minimaal één argument, namelijk de volledige bestandsnaam.

```
import sys
print(len(sys.argv), sys.argv[0]) # 1 C:\Users\Fred\source\repos\P1\P1\P1.py
for item in sys.argv:
  print(item)
```

Testen van een batchprogramma kan in je eigen IDE. Alle argumenten simuleer je of met globale variabelen of een test versie van de **argv** list. Je '*toggle*t' dan tussen de test **argv** list en de echte.

In het voorbeeld hieronder is **run_mode** de '*toggle switch*', tuimelschakelaar. Als het geheel naar productie gaat zet je **run_mode** op 'P'.

```
import sys
run_mode = 'O'                    # O A P straat, Ontwikkeling, Acceptatie, Productie

def func_test_argv():             # hier maak je je test argumenten command line
  pArgv = []
  pArgv.append("C:\\Batch\\b1.py")
  pArgv.append("-i Ja")
  pArgv.append("-x Buiten")
  return pArgv

def Main():
  if run_mode == 'O':
    argv = func_test_argv()     # hier je test argumenten
  else:
    argv = sys.argv             # hier de 'echte'
  for item in argv:
    print(item)

Main()
```

In dit voorbeeld zijn **-i** en **-x** dus opties, en **Ja** en **Buiten** de optie waardes. Op deze manier maakt het niet uit waar je de opties specificeert in de commando regel. Dus in je code moet je de opties aflopen etc.

sys.exit()
Met **sys.exit()** breek je het programma af. Je kunt een integer meegeven (0-255). Het batchprogramma kan daarop reageren. Een exit code van 0 staat meestal voor succesvol doorlopen. (Zie hier voor afbreken).

Argparse module
Het verwerken van alle command-line opties heet ook wel '*parsen*' ('*ontleden*'). De module **argparse** ondersteunt command-line verwerking.

26 Reguliere Expressies

Reguliere expressies zijn tekst strings die een '*patroon*' beschrijven dat je kunt/moet vinden in tekstuele data. Bijvoorbeeld, de reguliere expressie **a+** beschrijft een patroon dat bestaat uit een serie van één of meer keer de letter 'a'. In de string 'aardvarken' vind je dit patroon twee keer; de 'aa' aan het begin van het woord, en de 'a' verder op. Sommige tekens zijn '*meta-tekens*' die een speciale betekenis hebben in reguliere expressies. De meta-tekens zijn: **. ^ $ * + ? { } [ ] \ | ( )**

26.1 De re module

Om reguliere expressies, gewoon een stuk code, te gebruiken, moet je de **re** module importeren.
Die code kun je '*compileren*' via de **re** module om een '*patroon object*' te produceren. Dat patroon object gebruik je dan om te zoeken naar het patroon in de data. Hieronder een voorbeeld. Hier zoek je op elke sub-string die uit een **a** of meerder **a's** bestaat. Vandaar het + teken. Laat je die weg dan zoek je op **a** alleen.

```
import re
O_se = re.compile(r"a+")        # O_se= Object search_expression
r = O_se.findall("aardvarken")  # r=result, een list met ['aa', 'a']
print(r)
for item in r:
  print(item)
```

Je kunt de compilatie-stap over te slaan, en het zoeken van het patroon aan te roepen met een *class call* in de **re** module; de '*verkorte methode*'. Het compileren gebeurt onderhuids. De code wordt dan:

```
r = re.findall(r"a+", "aardvarken")  # r=result, een list met ['aa', 'a']
print(r)
for item in r:
  print(item)
```

Als je het patroon vaker gaat gebruiken in de code heeft het compileren naar een *patroon object* de voorkeur. Hoeft maar één keer gemaakt worden.
De **r"<xxx>** staat voor regular expression. Doe dit als je de patroontekst in variabelen wilt stoppen:

```
se="a+"
O_se = re.compile(se)
```

26.2 Match objecten

De **`findall()`** methode retourneert alle instanties van het gezochte patroon. Een '*match object*' geeft behalve het resultaat ook de index positie waar het patroon is gevonden. De **`search()`** methode retourneert een match object voor de **eerste** instantie waar het patroon in de string voorkomt.

```
O_se = re.compile(r"a+")                              # O_se = Object search_expression
O_m = O_se.search("Het aardvarken is ontsnapt!")      # O_m  = Object match
if (O_m != None):
  print("Gevonden: {}, begin: {}, eind: {}".format(O_m.group(), O_m.start(), O_m.end()))
  # resultaat: Gevonden: aa, begin: 4, eind: 6
else: print("Niets gevonden")
```

Je kunt de **`search()`** methode ook aanroepen via de *verkorte methode*. In ieder geval, je krijgt begin- en eindpositie terug van de **eerste** match. Als het patroon niet voorkomt geeft het object **`None`** terug. Test daarop. (Er wordt niets gevonden als je dit doet: **`O_se = re.compile(r"aaa+")`**).
De **`group()`** methode heeft een aantal handige toepassingen die je via argumenten kunt benaderen.

Om alle matches te doorlopen maak je gebruik van de **`finditer()`** methode. Deze maakt een iterator van de list van matches.

```
O_se = re.compile(r"a+")                              # O_se = Object search_expression
O_m = O_se.finditer("Het aardvarken is ontsnapt!")    # O_m  = Object match
if (O_m != None):
  for m in O_m:
    print("Gevonden: {}, begin: {}, eind: {}".format(m.group(), m.start(), m.end()))
else: print("Niets gevonden")
# Gevonden: aa, begin: 4, eind: 6
# Gevonden: a, begin: 9, eind: 10
# Gevonden: a, begin: 23, eind: 24
```

26.3 Reguliere expressies schrijven

De eenvoudigste reguliere expressie is een string van tekens. Je mag ook een verzameling tekens beschrijven tussen vierkante haken, **`[`** en **`]`**. Dat acteert als een **`or`** zoek methode. Dit **`r"b[aeo]ll"`** vindt alleen ball, bell of boll. Zie hieronder.

```
print(re.findall(r"b[aeo]ll", "Een bal, een bel, een bol"))     # [] is dus niets gevonden
print(re.findall(r"b[aeo]ll", "Een ball, een bell, een boll"))  # ['ball', 'bell', 'boll']
```

Het backslash teken (\) wordt gebruikt om aan te geven dat het volgende teken een speciale betekenis heeft. Hier zijn een aantal.

`\b`	Woord begrenzing.
`\B`	Geen woord begrenzing.
`\d`	Cijfer [0-9]. **`\d{1,2}`** verwacht 1 of 2 cijfers. **`\d{4}`** verwacht vier cijfers.
`\D`	Geen cijfer [^0-9].
`\n`	Nieuwe regel (newline).
`\r`	Return.
`\s`	Spatie (inclusief tabulatie).
`\S`	Geen spatie.
`\t`	Tabulatie.
`\w`	Alfanumeriek teken [A-Za-z0-9_]
`\W`	Geen alfanumeriek teken [^A-Za-z0-9_]
`\<x>`	Groep referentie teken; x staat naar welke groep je refereert.
`\/`	Voorwaartse slash.
`\\`	Backslash.
`\"`	Dubbel aanhalingsteken.
`\'`	Enkel aanhalingsteken.
`^`	Start van een string.
`$`	Einde van een string.
`.`	Ieder of elk teken.
`()`	Groepeer teken.
`|`	Pipeline teken (**`or`** functie).
`[]`	Bereik teken [abc] betekent of a, of b of c. [a-f] betekent; moet tussen a of f liggen.
`{}`	Een herhalingsoperator; de te verwachten aantal tekens, zie voorbeeld bij **`\d`**

Bijvoorbeeld de **^A** representeert een string die start met de letter 'A'. Je kunt tekens of sub-strings tussen haakjes zetten, wat ze groepeert. Binnen een groep kun je een keuze maken tussen meerdere (groepjes) tekens met het pipe-line (**|**) teken. **Let op:** Speciale tekens (zeker die zonder backslash) werken niet zoals aangegeven als ze tussen vierkante haken staan; de punt is dan niet "elk teken", maar een echte punt.

```
print(re.findall(r"(ball|bell|boll)", "Een ball, een bell, een boll"))  #['ball', 'bell', 'boll']
```

Er zijn ook herhalingsoperatoren om aan te geven dat (een deel van) een reguliere expressie meerdere keren herhaald wordt. Een aantal veelgebruikte zijn:

*****	Nul of meer keer.
+<n>	+ 1 of meer keer.
?	Nul of 1 keer.
{p,q}	Minstens **p** en hoogstens **q** keer.
{p,}	Minstens **p** keer.
{p}	Precies **p** keer.

De operator komt achter het deel dat herhaald moet worden. Bijvoorbeeld, ab*c betekent de letter 'a', gevolgd door nul of meer keer de letter 'b' en dan één keer de 'c'.

```
print(re.findall(r"ab*c", "abbbaxcabcaaabbbbcazcl"))     # ['abc', 'abbbbc']
```

Het matchen van herhalingen gebeurt '*gulzig*' ('*greedy*'). Er wordt altijd geprobeerd het patroon zo vroeg mogelijk in de tekst te matchen, en de herhaling zo breed mogelijk uit te strekken. Zie hieronder.

```
O_m = re.finditer(r"ba+","Schaap zegt 'baaaaah' tot Ali Baba.")      # baaaaa gevonden op: 13.
for m in O_m:                                                       # ba gevonden op: 32.
  print("{} gevonden op: {}.".format(m.group(), m.start()))
```

26.4 Groeperen

Groepen maakt je met haakjes; **(\d{1,2})-(\d{1,2})-(\d{4})** kan bijvoorbeeld een datum beschrijven: dd-mm-jjjj. Let ook op de **{}** (accolades) teken. Die geeft een te verwachten aantal aan. De expressie bevat dus drie groepen. De code hieronder zoekt naar dit patroon in een string.

```
O_se = re.compile(r"(\d{1,2})-(\d{1,2})-(\d{4})")
O_m = O_se.search("Gedateerd op: 25-1-2019, blablabla")
if (O_m != None):
  # Onderstaande geeft: D:25-1-2019; d:25; m:1; j:2019
  print("D:{}; d:{}; m:{}; j:{}".format(O_m.group(0), O_m.group(1), O_m.group(2), O_m.group(3)))
  print(O_m.group())           # dit is zelfde als group(0): 25-1-2019
  print(O_m.groups())          # tuple met: ('25', '1', '2019')
  for item in O_m.groups():
    print(item)                # hier druk je ze één voor één af
else: print("Niets gevonden")
```

Je ziet hier het effect van de methode **group()** en **groups()**. **Groups()** geeft een tuple met alle groepen.

De **findall()** methode retourneert een list van patroon objecten. Als er meerdere groepen zijn, is een patroon object een tuple waarin alle groepen zitten. Zie voorbeeld hieronder.

```
O_se = re.compile(r"(\d{1,2})-(\d{1,2})-(\d{4})")              Geeft de tuples:
O_m = O_se.findall("Eerst:25-1-2019, Twee:1-3-2019, blabla")   ('25', '1', '2019')
for datum in O_m:                                              ('1', '3', '2019')
  print(datum)
```

Via de de constructie **?P<naam>** kun je iedere groep een naam te geven. Je kunt dan aan de groepnamen refereren, i.p.v. de index. (De **P** betekent denk ik *parameter*).
Let op: dit keer maakt **<** en **>** wel onderdeel uit van de syntax.

```
O_se = re.compile(r"(?P<dag>\d{1,2})-(?P<maand>\d{1,2})-(?P<jaar>\d{4})")
O_m = O_se.finditer("Eerst:25-1-2019, Twee: 1-3-2019, blabla")
for m in O_m:
  print("{} begin: {} eind: {}".format(m.group(), m.start(), m.end()))
  print("dag  : {} begin: {}".format(m.group('dag'), m.start('dag')))
  print("maand: {} begin: {}".format(m.group('maand'), m.start('maand')))
  print("jaar : {} begin: {}".format(m.group('jaar'), m.start('jaar')))
Resultaat:
25-1-2019 begin:6 eind:15
dag  : 25 begin: 6
maand: 1 begin: 9
```

```
jaar : 2019 begin: 11
1-3-2019 begin:23 eind:31
dag  : 1 begin: 23
maand: 3 begin: 25
jaar : 2019 begin: 27
```

Hier nog een voorbeeldje. Een bedrag met een valuta symbool ervoor, mag er ook achter staan, en waarbij je alleen een getal wilt overhouden.

```
r = re.sub('\D', '','€ 0.00') # 000
print(r)
m = re.search(r"(\d+(.|,)(\d+)(.|,)(\d+))|(\d+(.|,)(\d+)|(\d+))", '€ 2.500,15')
if m != None :
  s = str(m.group())
print(s)             # 2.500,15
```

De string 2.500.500,15 wordt niet gevonden, je moet dan dat formaat nog een keer helemaal vooraan definiëren. Het getal 2500500,15 wel weer, en ook het getal 250050015. Dat doet namelijk de laatste groep.

26.5 Refereren binnen een reguliere expressie

Reguliere expressies die strings representeren waarin een willekeurig teken (dat geen spatie is) twee keer voorkomt kun je oplossen met groepen, en speciale referenties binnen reguliere expressies. De string 'gasoven' geeft geen match, maar 'magnetron' wel; de 'n' komt twee keer voor

Gebruikt het speciale teken **\x**, waarbij x een cijfer is, waarmee je dan refereert aan de groep met index **x** in het patroon. Dus de gevraagde reguliere expressie is: **(\S).*\1**
Uitleg: De **\S** is een speciaal teken dat een non-spatie voorstelt. Door er haakjes omheen te zetten, wordt het een groep, en omdat het de eerste (en enige) groep is, is de index 1. De **.*** stelt een serie van nul of meer tekens voor die alles kunnen zijn. De **\1** een referentie aan de eerste groep, en stelt dat je hier exact moet hebben wat de eerste groep is. Het is niet nodig om te beschrijven wat er vóór de **\S** of na de **\1** komt, omdat deze reguliere expressie niet de string als geheel representeert. Dus zolang dit maar ergens voorkomt in de string, wordt het patroon gevonden. Hieronder twee voorbeelden.

```
O_se = re.compile(r"(\S).*\1")
O_m  = O_se.search("Monty Python Flying Circus")
print(O_m)               # <_sre.SRE_Match object; span=(1, 11), match='onty Pytho'>
# o komt 2 keer voor in de string
print("{} komt twee keer voor in de string".format(O_m.group(1)))
```

```
O_m = O_se.finditer("Monty Python Flying Circus")
for m in O_m:
  print(m.group(0))
  print("{} komt 2 keer voor in de string".format(m.group(1)))
# onty Pytho
# o komt 2 keer voor in de string
# n s Flyin
# n komt 2 keer voor in de string
```

Waarom krijg je alleen maar een 'o' en een 'n'. Dit zijn namelijk de dubbels: 'o' 2x, 'n' 3x, 't' 2x, 'y' 3x en 'i' 2x. Dit is wat er gebeurt. Er wordt begonnen te zoeken vanaf de 'M' als daar nog eentje van wordt gevonden is dat de eerste groep. Is er geen tweede 'M' dan wordt de 'o' gepakt. Is dat het ook niet etc. Maar in dit geval is er wel een tweede 'o'. Beiden zijn gevonden in deze string: **"onty Pytho"**. Dan gaat de expressie verder met zoeken waar de pointer stond en dan wordt de 'n' gevonden in deze string: **"n Flyin". Dan** rest er alleen nog deze string **"g Circus"** om door te zoeken. En daar zitten geen dubbels in. Tja, is dat wat je wilt? Je zou zeggen; ik wil alle karakters die meer dan 1x voorkomen. Hieronder twee voorbeelden:

List

```
s = "Monty Python Flying Circus"  # search string
p = "(\S).*\\1"                   # pattern
L = []
O_se = re.compile(p)
for i in range(len(s)):
  s1 = s[i:]        #  de voorkant wordt afgekapt
  O_m = O_se.search(s1)
  if (O_m != None):
    v = O_m.group(1)
    if (L.__contains__(v) == False):
      L.append(v)
      L.append(s.count(v))
```

Dictionary

```
s = "Monty Python Flying Circus"  # search string
p = "(\S).*\\1"                   # pattern
D = {}
for i in range(len(s)):
  s1 = s[i:]
  O_m = O_se.search(s1)
  if (O_m != None):
    k = O_m.group(1)
    if (D.get(k) == None):
      D[k] = s.count(v)
print(D)
```

```
print(L)
# ['o', 2, 'n', 3, 't', 2, 'y', 3, 'i', 2]          # {'o': 2, 'n': 3, 't': 2, 'y': 3, 'i': 2}
```

Je gaat gewoon de hele string door. Door elke match in een list of dictionary te zetten alleen als die er nog niet is, pak je de dubbels. Dan tel je hoeveel van die karakters in de string zitten. Wil je weten welke karakters 3x voorkomen dan doe je bijvoorbeeld dit:

```
for k in D:                        # k =key
  if (D.get(k) == 2): print(k)
```

26.6 Vervangen

Gebruik de **sub()** methode om sub-strings binnen de string te vervangen. De methode heeft drie argumenten: Het patroon dat je wilt vervangen, het patroon waarmee je wilt vervangen, en de string. Bij vervanging kun je ook weer groepen gebruiken. In de volgende string zit een typefout.
`"zie zo: Of je nu categorizeert, rationalizeert, of analyzeert, je moet een s gebruiken!"`
Je wilt de 'z' vervangen door een 's'. Als je dit doet vervang je alle:

```
s  = "zie zo: Of je nu categorizeert, rationalizeert, of analyzeert, je moet een s gebruiken!"
p1 = "z"
p2 = "s"
r = re.sub(p1,p2,s)
print(r)
# sie so: Of je nu categoriseert, rationaliseert, of analyseert, je moet een s gebruiken!
```

Bij deze vervang je de juiste 's'.

```
p1 = "([iy])z(eert)"
p2 = "\g<1>s\g<2>"
r = re.sub(p1,p2,s)
print(r)
# zie zo: Of je nu categoriseert, rationaliseert, of analyseert, je moet een s gebruiken!
```

Je moet werken met groepen. Hier zijn er twee gebruikt om aan te geven dat je 'eert' ook kunt vervangen. In dit geval niet maar als je er dit van maakt **p2 = "\g<1>s\g<2>te"** dan plak je achter groep 2 de string 'te' en wordt het 'rationaliseertte'. Niet dat je daar veel aan hebt. Deze code is het kortst:

```
p1 = "([iy])z"
p2 = "\g<1>s"
r = re.sub(p1,p2,s)
```

Om 'eert' te vervangen door 'eerde 'doe je trouwens (twee mogelijkheden) dit:

```
p1 = "([iy])z([e])ert"        p1 = "([iy])z(e)ert"
p2 = "\g<1>s\g<2>erde"        p2 = "\g<1>s\g<2>erde"
```

De tweede is te prefereren omdat je toch maar op één beginkarakter van een groep zoekt.

27 Bestandsformaten

De bestandsformaten, csv, json, html en xml hebben allemaal hun eigen modules.

27.1 CSV

CSV staat voor '*Comma-Separated Values*'. Het is een tekstbestand waarbij de velden zijn gescheiden door komma's. Maar dat hoeft niet. Elk scheidingsteken volstaat. Zie ook hoofdstuk 15.

```
Dir = "C:/Users/Fred/source/repos/text/"    # we volgen de UNIX conventie
fil = "csv_1.csv"                                                    a,b,c,d,e
try:                                                                    f,g,h,i,j
  fp = open(dir + fil)          # fp staat voor FilePointer            klm,op,
  buffer = fp.read().strip()    # leest alles en verwijdert leading/trailing spaces   q,r,s,t
  print(buffer)                 # van het bestand. dus niet van elke regel!          u,v,w,x,y,z
  fp.close()                    # flush en sluit de stream
except: None
```

Het CSV-formaat is niet gestandaardiseerd. Python heeft een csv module die een standaard ondersteund die vrij veel gebruikt wordt. Een aantal methodes volgen nu.

CSV reader()

De **reader()** functie geeft toegang tot een CSV-bestand en retourneert een iterator die per regel (record) een list geeft met de velden (columns) van een regel.

```
import csv
dir = "C:/Users/Fred/source/repos/text/"    # we volgen de UNIX conventie
fil = "csv_1.csv"
try:
  fp = open(dir + fil)          # fp staat voor FilePointer
  csvreader = csv.reader(fp)
  for row in csvreader:         # de regel
    print(row)
    for col in row:             # de column of veld
      print("-",col.strip(),"-",sep="")
  fp.close()
except: None
# De eerste regel.
# ['   a', 'b', 'c', 'd', 'e     ']
# -a-
# -b-
# -c-
# -d-
# -e-
```

De **reader()** functie heeft twee belangrijke opties:

delimiter=<teken> Het veldscheidingsteken, default de komma.

quotechar=<teken> Het veldomsluitingsteken, default de **"**, dubbele aanhalingstekens.

Het omsluiten van een veld met **quotechar** is alleen nodig als de string buitengewone tekens bevat, zoals newline tekens, of tekens die ook als **delimiter** worden gebruikt.

CSV writer()

Met de **writer()** functie schrijf je naar een CSV-bestand. Je opent een bestand in de '**w**' of de '**a**' modus. Zelfde als bij tekstbestanden. **Let op.** Met **mode='w'** wordt je bestand eerst **leeggemaakt**.

De **writer()** functie heeft dan een **writerow()** of **writerows()** methode om velden, in een list, weg te schrijven naar het bestand. De **writer()** functie heeft drie belangrijke opties:

delimiter=<teken>	Het veldscheidingsteken, default de komma.	
quotechar=<teken>	Het veldomsluitingsteken, default de **"**, dubbele aanhalingstekens.	
quoting=<quotemethode>	Geeft aan welke type velden door **quotechar** omsloten worden:	
	csv.QUOTE_ALL	Elk veld.
	csv.QUOTE_MINIMAL	Alleen velden waar het noodzakelijk is. (Default).
	csv.QUOTE_NONNUMERIC	Alle velden die geen integers of floats zijn
	csv.QUOTE_NONE	Geen enkel veld.

```
import csv
dir = "C:/Users/Fred/source/repos/text/"    # we volgen de UNIX conventie
fil = "csv_1.csv"
try:
  fp = open(dir + fil, mode="a", newline="") # a=append, toevoegen
  csvwriter = csv.writer(fp)
  csvwriter.writerow([1,2,3])
  csvwriter.writerow(["1", "2", "3"])
  fp.close()
except: None
fp = open(dir + fil)
buffer = fp.read()
print(buffer)
fp.close()
```

Het schijnt belangrijk te zijn om de optie newline op **newline =''** te zetten. Dit is wat de documentatie zegt: "*If **newline = ''** is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use **\r\n** linendings on write an extra **\r** will be added. It should always be safe to specify **newline = ''**, since the csv module does its own newline handling.*"

Op Windows heb je **\r\n**. Laat je het weg in het voorbeeld hierboven, dan krijg je een lege regel.

27.2 Pickling

Gebruik de '*pickling*' ('*conservering*') module om (complexe) datastructuren op te slaan. Je conserveert, bewaart, een datastructuur in een bestand. Je gebruikt de **`dump()`** functie om data te schrijven, en de **`load()`** functie om te lezen en alles weer terug te zetten in het originele formaat. Zie het als '*inpakken*' en '*uitpakken*' van data. Hieronder een voorbeeld. Een list wordt opgeslagen, ingepakt, en daarna weer uitgepakt. Een tussenstap laat zie wat er kaal in de file staat.

```
import pickle
dir = "C:/Users/Fred/source/repos/text/"
fil = "pickle_1.pck"
data = [("A1", 12, 15.23), ("B1", 25, 19.02), ("C1", 5, 0.67), [12, [1,2], 13, 14]]
fp = open(dir + fil, "wb")
pickle.dump(data, fp)          # binair inpakken
fp.close()
fp = open(dir + fil,mode="rb") # binair lezen
buffer = fp.read()
print(buffer)
fp.close()
fp = open(dir + fil, "rb")     # weer uitpakken
data = pickle.load(fp)
fp.close()
print(type(data))              # je krijgt een list terug
print(data)
# b'\x80\x03]q\x00(X\x02\x00\x00\x00A1q\x01K\x0cG@.u\xc2\x8f\\(\xf6\x87q\x02X\x02\x00\x00\
x00B1q\x03K\x19G@3\x05\x1e\xb8Q\xeb\x85\x87q\x04X\x02\x00\x00\x00C1q\x05K\x05G?\xe5p\xa3\xd7\n=q\
x87q\x06]q\x07(K\x0c]q\x08(K\x01K\x02eK\rK\x0eee.'
# <class 'list'>
# [('A1', 12, 15.23), ('B1', 25, 19.02), ('C1', 5, 0.67), [12, [1, 2], 13, 14]]
```

Je ziet dat de **`load()`** functie de data structuur opnieuw opbouwt. Het werkt zelfs voor eigen classes:

```
class Punt:
  def __init__(self, x, y):
    self.x = x
    self.y = y
  def __repr__(self):
    return "({},{})".format(self.x, self.y)

import pickle
dir = "C:/Users/Fred/source/repos/text/"
fil = "pickle_2.pck"
fp = open(dir + fil, mode="wb+")
for x in range(4):
  for y in range(3):
    o_P = Punt(x, y)
    pickle.dump(o_P, fp)
fp.close()
fp = open(dir + fil, mode="rb")        # binair lezen
buffer = fp.read()
print(buffer)
fp.close()
fp = open(dir + fil, mode="rb")        # binair uitpakken
# Een loopje om alle punten uit het bestand te lezen
try:
  L =[]
  while True:
    L.append(pickle.load(fp))
except EOFError: None
print(L)   # [(0,0), (0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1), (2,2), (3,0), (3,1), (3,2)]
fp.close()
```

27.3 JavaScript Object Notation (JSON)

JavaScript Object Notation (JSON) is een bestandsformaat dat gebruikt wordt door applicaties die via webservices communiceren. JSON lijkt op pickling, maar daar wordt data binair opgeslagen. JSON slaat objecten op in tekst formaat, dus leesbaar. Gebruik de **`json`** module om JSON-files te maken/laden.

```
import json
dir = "C:/Users/Fred/source/repos/text/"
fil = "json_1.json"
data = [("A1", 12, 15.23), ("B1", 25, 19.02), ("C1", 5, 0.67), [12, [1,2], 13, 14]]
fp = open(dir + fil, "w")
json.dump(data, fp)          # omzetten naa json formaat
fp.close()
fp = open(dir + fil)         # tekst  lezen
buffer = fp.read()
print(buffer)
fp.close()
fp = open(dir + fil, "r")    # json weer uitpakken
data = json.load(fp)
print(data)
fp.close()
# [["A1", 12, 15.23], ["B1", 25, 19.02], ["C1", 5, 0.67], [12, [1, 2], 13, 14]]
# [['A1', 12, 15.23], ['B1', 25, 19.02], ['C1', 5, 0.67], [12, [1, 2], 13, 14]]
```

Alternatieven voor **`dump()`** en **`load()`** zijn de functies **`dumps()`** en **`loads()`** die geen handle als argument krijgen. In plaats daarvan kun je heel veel opties specificeren. Hieronder recht-toe-recht-aan voorbeeld.

```
dir = "C:/Users/Fred/source/repos/text/"
fil = "json_2.json"
data = [("A1", 12, 15.23), ("B1", 25, 19.02), ("C1", 5, 0.67), [12, [1,2], 13, 14]]
fp = open(dir + fil, "w")
j = json.dumps(data)         # omzetten naar json formaat
fp.write(j)                  # en hier schrijf je dan weg naar file
fp.close()
fp = open(dir + fil, "r")    # json weer uitpakken
data = fp.read()             # hier laad je json data uit file
j = json.loads(data)         # en hier
fp.close()
```

De **`json`** module ondersteunt alleen standaard datastructuren. Voor eigen classes moet je ze eerst omzetten naar standaard Python structuren. Gebruik de **`JSONencoder`** en **`JSONdecoder`** classes uit de **`json`** module.

27.4 HTML en XML

HTML en XML is een tekstformaat om informatie op webpagina's te tonen. Het is leesbare tekst met allemaal zogenaamde HTML-tags. Je kunt HTML pagina's verwerken via reguliere expressies, maar als de pagina's redelijk fatsoenlijk geformatteerd zijn, kun je de '*Beautiful Soup*' module gebruiken. Deze wordt in Python **`bs4`** genoemd. De module bevat de **`BeautifulSoup`** class die je kunt gebruiken om HTML en XML-bestanden te laden. Maar **`bs4`** wordt niet standaard door Python ondersteund; je moet het apart installeren. Een alternatief voor *Beautiful Soup* is de **`xml`** module.

In de voorbeelden in de officiële documentatie worden er eigen classes gemaakt die gebaseerd zijn op bijvoorbeeld **`html.parser.HTMLparser`**. De documentatie zegt dat je moet overriden: '*The user should subclass HTMLParser and **override** its methods to implement the desired behavior. This parser does not check that end tags match start tags or call the end-tag handler for elements which are closed implicitly by closing an outer element.*'

Hieronder twee voorbeelden, eentje die naar scherm print, en eentje naar een file. Om de een of andere reden lukte het mij niet om binnen de class een file te openen, of een file handle pointer door te geven.

```
import html.parser
class ClsHtmlPaser(html.parser.HTMLParser):
  def handle_starttag(self, tag, attrs):
    # Een tag bestaat meestal uit een of meer name/value pairs
    print("\nStart tag\t:", tag)
    if (len(attrs) != 0):
      for attribute in attrs:
        for i in range(2):
          if (i==0): print("name\t\t:", attribute[0])
          else: print("value\t\t:", attribute[1])
      print
```

```
  def handle_endtag(self, tag):
    print("End tag\t\t:", tag)

  def handle_data(self, data):
    if (data.strip().__len__() != 0):
      print("data\t\t:", data.strip())

  def feed(self, data):  # dit hoeft niet, kan achterwege worden gelaten
    super().feed(data)

dir_in  = "C:/Python/python-3.7.2rc1-docs-html/library/"
file_in = "index.html"
parser = ClsHtmlPaser()
fp = open(dir_in + file_in, encoding = "utf-8") # je moet de encoding goed zetten, anders fout
data = fp.read(250)
fp.close()
```

Hier de versie voor de output file. Je output nu alles naar een file.

```
import html.parser
class ClsHtmlPaser(html.parser.HTMLParser):
  def handle_starttag(self, tag, attrs):
    # Een tag bestaat meestal uit een of meer name/value pairs
    fpw.write("\nStart tag\t:" + tag + "\r\n")
    if (len(attrs) != 0):
      for attribute in attrs:
        for i in range(2):
          if (i==0): fpw.write("name\t\t:" + attribute[0] + "\r\n")
          else: fpw.write("value\t\t:" + attribute[1] + "\r\n")

  def handle_endtag(self, tag):
    fpw.write("End tag\t\t:" + tag + "\r\n")

  def handle_data(self, data):
    if (data.strip().__len__() != 0):
      fpw.write("data\t\t:" + data.strip() + "\r\n")

fpw  = open("C:/Users/Fred/source/repos/text/html_1.txt", mode="w", encoding = "utf-8")
parser = ClsHtmlPaser()
dir_in  = "C:/Python/python-3.7.2rc1-docs-html/library/"
file_in = "index.html"
fp = open(dir_in + file_in, encoding = "utf-8") # je moet de encoding goed zetten, anders fout
data = fp.read()
fp.close()
parser.feed(str(data))
fpw.close()
```

28 Diverse Nuttige Modules

Python is een verzameling classes die beschikbaar gesteld worden via het importeren van modules. Hier volgen een aantal handige modules.

28.1 datetime

De **`datetime`** module bevat functies om operaties op datum en tijd te doen. De belangrijkste classes zijn: **`datetime`**, **`timedelta`**, **`date`**, en **`time`**. De **`datetime`** bevat attributen **`year`** (jaar), **`month`** (maand), **`day`** (dag), **`hour`** (uur), **`minute`** (minuut), **`second`** (seconde), **`microsecond`** (duizendste seconde), en **`tzinfo`** (tijdzone). Objecten die instanties zijn van deze classes zijn onveranderbaar, net zoals strings, dus bewerkte data komt dan in een ander veld doormiddel van de assignment, **`=`**, operator.

```
import datetime as dt
print(dt.date.today())    # 2019-01-26
print(dt.datetime.now())  # 2019-01-26 11:29:04.559435
dt_str = now.strftime("%m/%d/%Y, %H:%M:%S")  # huidige tijd in string notatie
```

Output formaat (yyyy-mm-dd of dd-mm-yyyy etc.) kun je zelf instellen. Om met **`datetime`** objecten te rekenen, heb je **`timedelta`** object nodig. Dat object specificeert het verschil tussen twee **`datetime`** objecten en bevat **`days`** (dagen), **`seconds`** (seconden), en **`microseconds`** (duizendste seconden). De operaties die je uit kunt voeren zijn bijvoorbeeld optellen en aftrekken.

Een overzicht van alle datetime methodes etc. staat **hier**.

```
y = dt.datetime.now().year
d_xmas = dt.datetime(y, 12, 25, 23, 59, 59, 999999)  # Een datum; 25-12-y
d = dt.datetime.now()
print(d_xmas, d)                        # 2019-12-25 23:59:59.999999 2019-01-26 15:39:32.072203
o_Dagen = d_xmas - d                    # Object dagen
if o_Dagen.days < 0:
 print("Kerst komt volgend jaar weer.")
elif o_Dagen.days == 0:
 print("Het is Kerst!")
else:
  print("Nog", o_Dagen.days, "dagen tot Kerst!")     # Nog 333 dagen tot Kerst!
```

Hier een voorbeeldje met **`timedelta`** object. Zo te zien kun je dat ook allemaal doen met **`datetime`** object. Beiden geven zelfde antwoord.

```
d1 = dt.datetime(2018, 12, 12, 23, 59, 59, 999999)
d2 = dt.datetime(2019, 1, 15, 22, 11, 12)
print(d1, d2, d2-d1, sep="\n")       # 33 days, 22:11:12.000001
d3 = dt.timedelta(days=0, seconds=59, microseconds=999999, milliseconds=0, minutes=59, hours=23,
weeks=0)
d4 = dt.timedelta(days=34, seconds=12, microseconds=0, milliseconds=0, minutes=11, hours=22,
weeks=0)
print(d4-d3, sep="\n")               # 33 days, 22:11:12.000001
```

```
# Hier voorbeelden om timestamp om te zetten naar dates en interval in dagen etc. te berekenen.
a = dt.datetime.now()                                    # 2019-02-23 16:45:18.438401
b = dt.datetime.today() - dt.timedelta(12)               # 2019-02-23 16:45:18.438401
c = a.replace(hour=0, minute=0, second=0, microsecond=0)  # 2019-02-23 00:00:00
d = dt.datetime.today().date()                           # 2019-02-23
e = b.date()                                             # 2019-02-11
f = int((d-e).total_seconds()/(60*60*24))                # interval in dagen: 12
print(a, b, c, d, e, f, sep='\n')
```

Hier eenvoudige code om jaar, maand, dag, uur, minuten en seconden uit een datetime object te halen.

```
dt = datetime(2019, 5, 2, 18, 45, 44)
print(dt, dt.year, dt.month, dt.day, dt.hour, dt.minute, dt.second)
print("Enter any key")
```

Hier wat string naar date omzettingen.

```
dt_str = "12/31/99 15:01:55"
dt = datetime.strptime(dt_str, '%m/%d/%y %H:%M:%S')   # datetime type 1999-12-31 15:01:55
dt_str_1 = dt.strftime('%y-%m-%d %p%I:%M:%S')         # string: 99-12-31 PM03:01:55
dt_str_2 = dt.strftime('%Y-%m-%d %H:%M:%S')           # string: 1999-12-31 15:01:55
print(dt, dt_str_1, dt_str_2, sep="\n")
print(dt, dt.year, dt.month, dt.day, dt.hour, dt.minute, dt.second)
```

28.2 collections

De **`collections`** module bevat classes die helpen om iterabelen te manipuleren, zoals strings, tuples, lists, dictionaries, en sets. Twee handige zijn de **`Counter`** class en de **`Deque`** class. Een **`Counter`** object lijkt op een dictionary, die elementen bevat in de vorm van keys, en voor ieder van de elementen een 'telling' als waarde. Je maakt een **`Counter`** object door bij aanmaken als argument een sequentie te geven waarvan je de elementen wilt tellen.

Het voorbeeld hieronder telt dus het aantal unieke elementen in de list. De methode **`most_common()`** kan de eerste '*most common*', de eerste twee most common of allemaal laten zien. Als je de **`update()`** methode gebruikt voeg je meer elementen toe aan de list en wordt de telling anders.

```
data = ["appel", "banaan", "appel", "banaan", "appel", "kers" ]
o_c = coll.Counter(data)
print(o_c)                      #  Counter({'appel': 3, 'banaan': 2, 'kers': 1})
print(o_c.most_common(1))       #  [('appel', 3)]
data_2 = ["mango", "kers", "kers", "kers", "kers"]
o_c.update(data_2)
print(o_c)                      # Counter({'kers': 5, 'appel': 3, 'banaan': 2, 'mango': 1})
print(o_c.most_common(1))       # [('kers', 5)]
print(o_c.most_common())        # [('kers', 5), ('appel', 3), ('banaan', 2), ('mango', 1)]
```

Een **deque** object is een list die je gebruikt als een '*queue*' ('*wachtrij*'). De queue operaties werken alleen op de uiteinden van de queue. Je verandert iets aan de *voorkant* of aan de *achterkant* De voorkant is dan de *left* side. Voor de rest zijn het de standaard list methodes, die op de rechterkant werken. Hieronder een voorbeeld:

```
dq = coll.deque([1, 2, 3])
dq.appendleft(4)            # werkt op de voorkant
dq.extendleft([5,6])
dq.append(0)                # werkt op de acherkant
dq.extend([9,99])
print(dq)                   # deque([6, 5, 4, 1, 2, 3, 0, 9, 99])
```

28.3 urllib

Gebruik de **urllib** module om webpagina's te benaderen zoals je bestanden benadert. Twee interessante modules zijn: **urllib.request** om informatie op het Internet te benaderen, en **urllib.error** die http error definities heeft. Gebruik de **urllib.parse** module om te communiceren met websites. Als een webpagina is geopend heb je een file handle waar de reguliere tekstfile methodes van toepassing zijn.

```
from urllib.request import urlopen
from urllib.error import HTTPError, URLError
import sys
try:
  u = urlopen("https://docs.python.org/3")  # geeft een file handle
  print(type(u))                # <class 'http.client.HTTPResponse'>
except HTTPError as e:          # gebruik deze url: urlopen("https://docs.python.org/3/1")
  print("HTTP Error", e)
  input("Press any key to continue . . .")
  sys.exit()
except URLError as e:           # gebruik deze url: urlopen("https://docss.python.org/3")
  print("URL error", e)
  input("Press any key to continue . . .")
  sys.exit()
try:
  while True:
    tekst = u.readline()
    if (len(tekst) == 0): break
    print(tekst)
except EOFError: None
```

28.4 glob

De **glob** module heeft een functie **glob()** waarmee je een list van bestandsnamen kunt produceren, gebaseerd op een zoek-patroon dat als argument wordt meegegeven. Het patroon **"A[0-9]?B.*"** zoekt alle bestanden die beginnen met de letter 'A', gevolgd door een cijfer, gevolgd door een willekeurig teken, gevolgd door een 'B', met een willekeurige extensie. De functie **iglob()** geeft een iterator, i.p.v. een list.
Let op: Het patroon heeft niets van doen met reguliere expressies.

```
import glob
# File die string"_1" in de file naam hebben staan
glist = glob.glob("C:/Users/Fred/source/repos/text/*_[1]*.*")
for name in glist:
  print(name)
```

28.5 statistics

De **statistics** module geeft toegang tot statistische functies. Al deze functies krijgen als argument een sequentie of iterator met getallen (integers of floats). Om een paar functies te noemen:

mean()	Het gemiddelde.
median()	De mediaan, is het middelste getal van een reeks. Bij een reeks met een even aantal getallen wordt het gemiddelde van de twee middelste waardes genomen. Maar hier bestaan varianten op en die worden ondersteunt.
mode()	De modus, is het getal wat het meest voorkomt in de reeks. Python verwacht ook nog dat het getal uniek moet zijn. Dus als er meerdere waardes zijn die het even vaakst voorkomen wordt er een fout gegenereerd. Dat is niet echt handig.
stdev()	De standaarddeviatie.
variance()	De variantie.

Deze functies kunnen een **`StatisticsError`** genereren. Dit is relevant voor de **`mode()`** functie, omdat deze exceptie gegenereerd wordt als er geen unieke modus is.

```
import statistics as stat
data = [4, 5, 1, 1, 2, 2, 2, 3, 3, 3]
print("gemiddelde:", stat.mean(data))  # gemiddelde: 2.6
print("mediaan:", stat.median(data))   # mediaan: 2.5
try:
  print("modus:", stat.mode(data))
except stat.StatisticsError as e:
  print(e)                             # no unique mode; found 2 equally common values (2 en 3)
print("st.dev.: {:.3f}".format(stat.stdev(data)))      # st.dev.: 1.265
print("variantie: {:.3f}".format(stat.variance(data))) # variantie: 1.600
```

29 Pandas

Pandas is een module bedoelt voor data-analyse en modelering. Het is een tweedimensionale datastructuur waar je allemaal data handelingen op kunt verrichten. *Pandas* is afgeleid van '*panel data*', een term uit de financiële statistiek die rijen data bevat over meerdere tijd perioden voor dezelfde individuen of eenheden. Dus het zal zoiets betekenen als **Pan**el **D**ata **A**nalysi**s**.
In dit hoofdstuk komen met name praktische oplossing voor problemen waar ik tegen aan ben gelopen om zo'n *panel*, die trouwens een *dataframe* heet, te vullen. (Je kunt ook driedimensionale dataframes maken.)

29.1 Opvullen van lege waarden

Binnen Pandas heet een dataset een dataframe en is op te vatten als een tabel. Je definieert een kolomschema en vult die per rij met waarden. Cellen die leeg zijn krijgen de waarde '**`NaN`**' (Not Available Number, wat een float is.). **Let op;** de kolomvolgorde is automatisch gesorteerd. Dus als je een dataframe print, krijg je niet persé de kolom volgorde zoals gedefinieerd. Een dataframe is mutable.

```
import pandas as pd

df = pd.DataFrame(columns=['D1', 'C1', 'B1', 'A1'])
df1 = pd.DataFrame([[1, 2], [3, 4], [5, 6], [7 ,8], [9,10]], columns=['D1', 'C1'])
df = df.append(df1, ignore_index=True)
print(df)                          # de kolomvolgorde is A1 t/m D1
df = df[['D1', 'B1', 'C1', 'A1']] # hier forceer je de kolomvolgorde willekeurig
df.head()                          # in feite hier
df = df.fillna(0)                  # hier vul je alle NaN's met een 0
print(df)
a=input("1: Press any key ...")

df = pd.DataFrame(columns=['D1', 'C1', 'B1', 'A1'])
df1 = pd.DataFrame([[1, 2], [3, 4], [5, 6], [7 ,8], [9,10]], columns=['D1', 'C1'])
df = df.append(df1, ignore_index=True)
if not(df.empty) :   # hier controleer je of een dataframe rijen heeft
  df = df.fillna(0)
  print(df)
a=input("2: Press any key ...")

df = pd.DataFrame(columns=['D1', 'C1', 'B1', 'A1'])
df1 = pd.DataFrame([[1, 2], [3, 4], [5, 6], [7 ,8], [9,10]], columns=['D1', 'C1'])
df = df.append(df1, ignore_index=True)
NaN_values = {'A1': '-een', 'B1': 12} # hier kun je verschillende default waarden
df = df.fillna(value=NaN_values)      # voor NaN kolommen opgeven.
print(df)
a=input("3: Press any key ...")

df = pd.DataFrame(columns=['D1', 'C1', 'B1', 'A1'])
df1 = pd.DataFrame([[1, 2], [3, 4], [5, 6], [7 ,8], [9,10]], columns=['D1', 'C1'])
df = df.append(df1, ignore_index=True)
for r in range(df.shape[0]):          # hier loop je per cell door het het dataframe heet
  for c in range(df.shape[1]):
    if pd.isna(df.iat[r,c]) == True:  # als een cel NaN is, dan geef
      df.iat[r,c] = 0                 # je het hier een waarde
print(df)
a=input("4: Press any key ...")
```

30 PostgreSQL

PostgreSQL is een opensource transactionele database. De database wordt vrij veel gebruikt door commerciële bedrijven binnen eigen applicaties of om eigen database implementaties te maken. De Amazon Web Services Redshift database is bijvoorbeeld een PostgreSQL implementatie. Er zijn verschillende data adapters die kunnen connecteren met PostgreSQL. *IronPython.net* is interessant omdat die een **ADO.NET** framework implementatie bevat. Maar, IronPython werkt nog niet voor Python3. De *Psycopg* module zegt van zichzelf dat het de meest populaire PostgreSQL database adapter voor Python is, en dat Psycopg 2 zowel Unicode als Python3 'vriendelijk' is.

Hieronder volgt een eenvoudige Class, met optionele event handling, om te connecteren met een PostgreSQL database. De connection string parameters staan in *Windows Environment Variabelen*.

```
import psycopg2 as pg

class ClsPg:
  """
  PostgreSQL connection class.
  Provides a few methods the read and write data to a postgreSQL database.
  The prefix __ means that the variables and/or functions are private.

  The class has one event that fires in case of errors. Event handling is
  optional. Even when it is turned on, it is up to the user of the class to
  capture that event.

  All variables are private. Use the properties to read/write:
  p_diag_msg   : set/return the diagnostics flag. True or False
  p_error_msg  : set/return any error message generated by this class
  p_connection : return the connection object (__CNN)
  p_cursor     : return the cursor object (__CUR)
  p_user       : set/return the user name (__user)
  p_pwd        : set the passwordt (__pwd)
  p_host       : set/return the host name, IP address (__host)
  p_port       : set/return the port number (__port)
  p_database   : set/return the database (__database)
  p_cnn_string : set/return connection string. When you set the connection string
                 the internal connection string is created.
  When you change one of the connection string parameter values you have to run
  method connection_string() to make a connection string. But only
  if no connection string has been defined. Use the re_connect() method when you
  want to change an existing connection string.
  A connection string consists of two part that both must exists:
  - The complete internal connection string.
  - All individual connection parameter name/value pair. Those are used to generate
    the internal connection string.

  Events:
  event_handler   : Defines the callback function that must be executed when the event is fired.
                    When now callback function defined, then no event handling.
  __fire_pg_event : Fires the event and executed the callback function. (It is a local method).

  Local methods:
  __make_connection_string()    : Makes the connections string. All parameter values must exist.
  __display_connection_string() : Display the connection string, the password is greyed
  __parse_cnn_string()          : Parse the connection string and put all parameter values in the
                                  correct parameters.
                                  user, password, host, port, dbname are supported.

  Public methods:
  quoted_string     : Put single quotes around a string
  connection_string : make the internal connection string when not yet defined
  connect           : make the connnection to the server at return CONNECTION and CURSOR object.
                      when internal connection string not set, it is set here
  close             : closes connection to the server
  re-connect        : close server connection and re-connect to server definition
                      as specified in the internal connection string. So, can be
                      a different server.
  execute           : Execute a DDL SQL statement and does auto-commit.
  cursor            : Executes a DML SQL statement and return the CURSOR object.
```

```
  table_drop        : Drops or removes a table.
  table_truncate    : Empties a table using the TRUNCATE statement. No ROLLBACK possible.
  table_delete      : Empties a table using the DELETE statement. ROLLBACK possible.
  table_exist       : Checks if a table exists.
  row_count()       : Returns the row count of a table.
  """

  # Constructor section
  # region

  def __init__(self, pUser=None, pPwd=None, pHost=None, pDb=None, pPort=5432):
    """
    The constructor. When all optional parameters are defined the connection string is made.
    The default port number of PostgreSQL is 5432
    """
    self.__user = pUser
    self.__pwd = pPwd
    self.__host = pHost
    self.__database = pDb
    self.__port = pPort
    self.__CNN = None
    self.__CUR = None
    self.__cnn_string = None
    self.__diag_msg_flag = False
    self.__error_msg = None
    self.__e_hdl = None          # the event handler or callback function
    if pUser != None and pPwd != None and pHost != None and pPort != None and pDb != None :
      self.__make_connection_string()

  #endregion

  # Event section
  #region

  def event_handler(self, e_hdl):
    """ Defines the event handler function (callback function) that must be called/executed """
    self.__e_hdl = e_hdl

  def __fire_pg_event(self):
    """
    Fire the event and execute the callback function. Testing on existence of callback function
    is mandatory. When you omit it, your code won't run, regardless if you use events or not
    """
    if self.__e_hdl != None :
      self.__e_hdl()                   # executes the callback function

  #endregion

  # Properties section
  #region

  @property
  def p_connection(self):
      """ Return the CONNECTION object """
      return self.__CNN

  @property
  def p_cursor(self):
    """ Returns the CURSOR object """
    return self.__CUR

  @property
  def p_diag_msg(self):
    """ True or False indicate if diagnostics message must be displayed in the error message
string """
    return self.__diag_msg_flag
  @p_diag_msg.setter
  def p_diag_msg(self, pDiag):
    self.__diag_msg_flag = pDiag

  @property
```

```
  def p_error_msg(self):
    """ Returns the error message string """
    return self.__error_msg

  @property
  def p_user(self):
    """ Read/Write of the database user account name """
    return self.__user
  @p_user.setter
  def p_user(self, pUser):
    self.__user = pUser

  @property
  def p_pwd(self):
    """ Returns the grayed database user account password """
    return 'XXXX'

  @property
  def p_host(self):
    """ Read/Write of the host name, IP adress, of the database server """
    return self.__host
  @p_host.setter
  def p_host(self, pHost):
    self.__host = pHost

  @property
  def p_port(self):
    """ Read/Write of the port number """
    return self.__port
  @p_port.setter
  def p_port(self, pPort):
    self.__port = pPort

  @property
  def p_database(self):
    """ Read/Write of the database name that holds the tables etc. """
    return self.__database
  @p_database.setter
  def p_database(self, pDB):
    self.__database = pDB

  @property
  def p_cnn_string(self):
    """ Read/Write of the connection string """
    return self.__display_connection_string()
  @p_cnn_string.setter
  def p_cnn_string(self, pcnn_string):
    self.__parse_cnn_string(pcnn_string)
    self.__make_connection_string()

  #endregion

  # Local functions/methods section
  #region

  def __make_error_msg(self, args, diag):
    """ Makes the error string an fires the event """
    self.__error_msg = ''
    for i in args:
      self.__error_msg += i + "\n"
    if self.__diag_msg_flag:
      if diag.column_name != None :        self.__error_msg += "Column name       : " +
diag.column_name + "\n"
      if diag.constraint_name != None :    self.__error_msg += "constraint_name   : " +
diag.constraint_name + "\n"
      if diag.context != None :            self.__error_msg += "context           : " +
diag.context + "\n"
      if diag.datatype_name != None :      self.__error_msg += "datatype_name     : " +
diag.datatype_name + "\n"
      if diag.internal_position != None :  self.__error_msg += "internal_position : " +
diag.internal_position + "\n"
```

```
      if diag.internal_query != None :     self.__error_msg += "internal_query    : " +
diag.internal_query + "\n"
      if diag.message_detail != None :     self.__error_msg += "message_detail    : " +
diag.message_detail + "\n"
      if diag.message_hint != None :       self.__error_msg += "message_hint      : " +
diag.message_hint + "\n"
      if diag.message_primary != None :    self.__error_msg += "message_primary   : " +
diag.message_primary + "\n"
      if diag.schema_name != None :        self.__error_msg += "schema_name       : " +
diag.schema_name + "\n"
      if diag.severity != None :           self.__error_msg += "severity          : " +
diag.severity + "\n"
      if diag.source_file != None :        self.__error_msg += "source_file       : " +
diag.source_file + "\n"
      if diag.source_function != None :    self.__error_msg += "source_function   : " +
diag.source_function + "\n"
      if diag.source_line != None :        self.__error_msg += "source_line       : " +
diag.source_line + "\n"
      if diag.sqlstate != None :           self.__error_msg += "sqlstate          : " +
diag.sqlstate + "\n"
      if diag.statement_position != None : self.__error_msg += "statement_position: " +
diag.statement_position + "\n"
      if diag.table_name != None :         self.__error_msg += "table_name        : " +
diag.table_name + "\n"
    self.__fire_pg_event()

  def __make_connection_string(self):
    """ Makes the connection string """
    self.__cnn_string ="user={} password={} host={} port={} dbname={}".format(self.__user,
self.__pwd, self.__host, self.__port, self.__database)

  def __display_connection_string(self):
    """ Displays the connection string. The password is greyed """
    return "user={} password={} host={} port={} dbname={}".format(self.__user, 'XXX',
self.__host, self.__port, self.__database)

  def __parse_cnn_string(self, pcnn_string):
    """ Simply connection string parser. Puts the pairs into local variables """
    par_d = False  # parameter detected flag
    val_d = False  # value detected flag
    D = {}         # put the found parameter/value pair into a dictionary
    for c in pcnn_string.strip():
      if par_d == True :
        if c == "=" :
          val = ''
          par_d = False
          val_d = True
        else :
          if c != " " : par += c
      else :
        if val_d == True :
          if c == " " :
            D[par] = val
            par = ''
            par_d = True
            val_d = False
          else :
            val += c
        else :
          par = c
          par_d = True
          val_d = False
    D[par] = val    # the last parameter-value pair
    # Assign to the connection parameters
    for k in D:
      if k == 'user':
        self.__user = D[k]
      elif k == 'password' :
        self.__pwd = D[k]
      elif k == 'host':
        self.__host = D[k]
```

```
    elif k == 'port':
      self.__port = D[k]
    elif k == 'dbname':
      self.__database = D[k]

# endregion

# Public methods
#region

def quoted_string(self, pString):
  """ Put single quotes around a string """
  return "'" + pString + "'"

def connection_string(self):
  """ Make the connection string when not yet defined """
  if self.__cnn_string == None:
    self.__make_connection_string()

def connect(self):
  """  Makes the connection with the database and set the CONNECTION and CURSOR object """
  if self.__CNN == None:
    if self.__cnn_string == None:       # unlikely condition to happen, but you never know
      self.__make_connection_string()
    try:
      self.__CNN = pg.connect(self.__cnn_string)
      self.__CUR = self.__CNN.cursor()
      return self.__CNN
    except pg.Error as err:
      self.__make_error_msg(err.args, err.diag)
      return None
  else:
    return None

def close(self):
  """ Close the connection to the database """
  try:
    self.__CUR.close()
    self.__CNN.close()
  except pg.Error as err:
    self.__make_error_msg(err.args, err.diag)

def re_connect(self):
  """ Re-connect to a database """
  if self.__CUR != None: self.__CUR.close()
  if self.__CNN != None: self.__CNN.close()
  self.__make_connection_string()
  return self.connect()

def execute(self, pSQL):
  """ Executes a DDL statement using auto-commit """
  try:
    with self.__CNN:              # this does auto-commit
      self.__CUR.execute(pSQL)
  except pg.Error as err:
    self.__make_error_msg(err.args, err.diag)
    return None

def cursor(self, pSQL):
  """ Executes a DML statement and returns the cursor object """
  try:
    self.__CUR.execute(pSQL)
    return self.__CUR
  except pg.Error as err:
    self.__make_error_msg(err.args, err.diag)
    return None

def table_drop(self, pTable, pSchema='public'):
  """  Drop table. The default schema is public """
  sql_str = "drop table if exists {var1}.{var2}".format(var1=pSchema, var2=pTable)
  self.execute(sql_str)
```

```
  def table_truncate(self, pTable, pSchema='public'):
    """  Truncate (empties) a table. No ROLLBACK possible. The default schema is public """
    sql_str = "truncate table {var1}.{var2}".format(var1=pSchema, var2=pTable)
    self.execute(sql_str)

  def table_delete(self, pTable, pSchema='public'):
    """  Delete all rows from a table. ROLLBACK possible. The default schema is public """
    sql_str = "delete from table {var1}.{var2}".format(var1=pSchema, var2=pTable)
    self.execute(sql_str)

  def table_exist(self, pTable, pSchema='public'):
    """  Check if a table exists. The default schema is public """
    sql_str = """select table_schema, table_name, count(*) as cnt
                 from {var1}
                 where table_schema = {var2} and table_name = {var3}
                 group by 1, 2""".format(var1='information_schema.tables',
                                         var2=self.quoted_string(pSchema),
                                         var3=self.quoted_string(pTable))
    self.__CUR.execute(sql_str)
    if self.__CUR != None :
      l = self.__CUR.fetchall()            # returns a list of tuples
      if len(l) == 1 and l[0][2] == 1:
        return True
      else: return False
    else: return False

  def row_count(self, pTable, pSchema='public'):
    """ Count the number of rows in a table. The default schema is public """
    sql_str = "select count(*) as cnt from {var1}.{var2}".format(var1=pSchema,
                                                                 var2=pTable)
    self.__CUR.execute(sql_str)
    if self.__CUR != None :
      t = self.__CUR.fetchone()            # returns one tuple
      return t[0]
    else : return None

  #endregion

# Module debug section
#region

if __name__ == '__main__':

  from events import Events
  from  os import environ as env

  def consume_pg_event():
    """ The callback function """
    print(Opg.p_error_msg)


  # pg_event is the event name, consume_pg_event is the callback function that is
  # called and executed. The name pg_event is rather random, the callback function not,
  # that function must exist.
  e = Events()
  e.pg_event += consume_pg_event

  l_user = env.get('pg_user')
  l_pwd = env.get('pg_pwd')
  l_host = env.get('pg_host')          # localhost
  l_port = env.get('pg_port')          # 5432
  l_database = env.get('pg_database' ) # python

  Opg = ClsPg(pUser=l_user, pPwd=l_pwd, pHost=l_host, pDb=l_database)
  # Set the callback function. When not sets, events are not fired
  Opg.event_handler(consume_pg_event)

  # It is not mandatory to assign the connection object to a variable.
  CNN = Opg.connect()
  if CNN != None:
```

```
    print(CNN.dsn, CNN.closed, Opg.p_cursor, sep="\n") # closed = 0 means ok
    print(Opg.table_exist("t1"), Opg.row_count("t1"), sep="\n")
    Opg.table_truncate('t1')
    for i in range(10):
      sql_str = 'insert into {0} values ({1}, {2})'.format('t1', i, Opg.quoted_string('AA_' +
str(i)))
      Opg.execute(sql_str)

    sql_str = "select * from {var1}.{var2}".format(var1='public', var2='t1')
    Ocur = Opg.cursor(sql_str)
    if Ocur != None:
      for t in Ocur:     # iterates through a list of tuples
        print(t)
    Opg.close()

#endregion
```

31 Fraude regels

Een transactie fraude regel is een software module die iets doet. Er zijn heel veel soorten fraude regels. Bijvoorbeeld; binnen 30 minuten vindt er een pintransactie plaats met dezelfde kaart op een locatie in Amsterdam en Rotterdam. Dat kan fysiek niet.

In Python kun je gebruik maken van pandas en NumPy, maar dat zorgt voor een extra laag, met eigen syntax. Tevens krijg je te maken van hun specifieke data types. Met reguliere Python kom je net zo ver. Het is eenvoudiger te lezen, en sneller.

Het doel is om door een groep van data heen te glijden, daar een subset van nemen, een snee, en die dan analyseert. Zie het ook letterlijk als een broodsnee, die ene keer dunner, de andere keer dikker. Abstract; er is een continue stroom aan deeg (de data) die hak je in broden (de groep) en die hak je weer in sneden (de te analyseren data). De sneedikte (aantal rijen in de data) is nooit constant.

Neem deze dataset; 1 Groep met 5 rijen. Je wilt zoeken naar twee transacties minimaal, binnen drie minuten.

```
Groep minuut
------------
G1    1
G1    2
G1    3
G1    6
G1    9
```

Er zijn drie manieren om een snee te retourneren.
Algoritme_1: Je gaat één voor één door de dataset. Je retourneert de eerste rij en analyseert die. Dan vul je die aan met de volgende rij en analyseert dat, dat blijf je doen zolang de volgende rij binnen de drie minuten blijft van de eerste rij. Past het niet meer, dan pak je rij 2 etc. Je krijgt dan dit.

```
1 1 1 2 2 3 3 6 6 9
  2 2   3   6   9
    3
```

Er worden hier 10 sneden geanalyseerd waarvan 5 voldoen aan je criteria dat er twee transacties zijn.

Algoritme_2: Je gaat één voor één door de dataset. Je retourneert een dataset om te analyseren als het niet meer past. Je krijgt dan dit:

```
1 2 3 6
2 3 6 9
3
```

Er worden hier 4 sneden geanalyseerd waarvan alle 4 voldoen aan je criteria dat er twee transacties zijn.

Algoritme_3: Het algoritme is gelijk aan ***algoritme_2***, maar je kijkt of de nieuwe snee, niet dezelfde einde heeft als de vorige snee. Als dat zo is, geen noodzaak om te analysen. In een kleinere snee zit niet meer informatie dan in een grotere snee.

```
1 3 6
2 6 9
3
```

Er worden hier 3 sneden geanalyseerd waarvan alle 3 voldoen aan je criteria dat er twee transacties zijn. Het ene algoritme is niet beter of slechter dan de andere, want het is maar net wat je wilt retourneren en uiteindelijk wilt bewaren. De TimeWindowModule.py doet ***algoritme_3***.

Wil je een snee vergelijken tegen historische data van een transactiedrager (pinpas, creditcard, je mobiel, etc.), dan moet je in de **`analyze_window()`** functie de database opnieuw bevragen en vergelijken met het historische gedrag van de transactiedrager, of als dat kan met het onderliggende banknummer.

```
"""
Module: TimeWindowModule.py
- Chops a stream of data in pieces based on a key.
- Chops the key data in pieces based on a time interval.
In other words:  A group of data in a group.
Note: The input must be sorted by the key and the date column. It can be done
      in Python, but handier is to do that in the SQL that feeds the input data set.
Note: The explanation mentions one output list. In fact the output list is a nested list.
      You can combine different rules that share the same DateTime Window.

The module consists of two functions:
group_by_current(): A helper function.
time_window()     : The worker function that chops everything in pieces.


            k1  k2  D  c1  c2  c3
  ================================
    ^     ^   1   2  1   a   b   c
    |     |   1   2  2   d   e   f
    |  Dw |   1   2  3   g   h   i
    |     |   1   2  4   j   k   l
    |     V   1   2  5   m   n   o
GBw |   -------------------------
    |     ^   1   2  6   p   q   r
    |  Dw |   1   2  7   s   t   u
    |     |   1   2  8   v   w   x
    V     V   1   2  9   y   z   z
  ================================
Terminology:
k1 = key_1, forms the GROUP BY clause and makes a GroupBy Window
k2 = key_2, forms the GROUP BY clause and makes a GroupBy Window
D  = DateTime column, makes a DateTime Window within a GroupBy Window
c1 = Other columns.

GBw: GroupBy window. A slice of records that belongs to a GROUP BY clause.
Dw : DateTime Window. A slice of records that belongs to a DateTime interval,
     in seconds, within a GroupBy Window.

The picture above is a bit misleading regarding the Dw. In fact it returns the
maximum number of records (Dw) for a time range within the GroupBy Window. In
the example below that is: A, D and E. All the others can be ignored.

     A b c D E f g h i j
  1  *
  2  * *
  3  * * *
  4  * * * *
  5  * * * * *
  6  * * * * * *
  7        * * * *
  8        * * * * *
  9          * * * * *
 10          * * * * * *
Thus, one slice per end boundary is outputted. The boundaries never overlap.
The records 1-10 forms a GroupBy Window, and 4-8 forms a DateTime Window. (D)
There are two windows or slices:
- The GroupBy Window Slice
- The DateTime Window Slice.
"""

from datetime import datetime, timedelta
from collections import Counter
```

```
def group_by_current(pL1c, pLgb):
  """
  Get the current GROUP BY clause values from the current row.
  pL1c: A list with the current row of L1
  pLgb: A list containing the GROUP BY column indexes.
  """
  L = []
  for i in range(len(pLgb)):
    L.append(pL1c[pLgb[i]])
  return L

def time_window(pL1, pL2, pLgb, pD, pWs, pF, pLcols=None, pDebug=False):
  """
  Input parameters:
  pL1   : The input list, containing the data. A 2-dimensional structure.
  pL2   : The output list. The output list is populated at the pF function.
          The list variable must be declared somewhere. A 2-dimensional structure.
  pLgb  : A list containing the indexes that forms the GROUP BY clause. A
          1-dimensional structure. This gives the Groupby Window Slice.
          A list of column names is allowed too.
  pD    : The DateTime column index. An integer. Column name is allowed too.
  pWs   : The window size in seconds. An Integer. This gives the DateTime
          Window Slice, which is a slice within the GroupBy Window Slice.
  pF    : The calling function that operates on the DateTime Window Slice.
          The pF function returns pL2, the output list, a 2-dimensional
          structure. All logic on the DateTime Window Slice is done there.
          This calling functions lets pL2 grow incremental.
          Name this function as follows; analyze_window(plw). This is because
          you have to define this function yourself, outside this module.
  pLcols: When pLgb and pD contains column names, which is a subset of the
          columns in pL1, you have to specify all pL1 column names here, and
          in the right order. Needed for column to index mapping.

  Local variables:
  Ls : List containing tuples whereby each tuple contains the start and end
       index of a GroupBy Window Slice.
  s1 : slice index begin.
  s2 : slice index end. Note: this is an exclusive value.
  T  : Tuple containing the GroupBy Window Slice index values.
  s  : A slice that forms a GroupBy Window Slice.
  Lp : A list containing the values of the previous GROUP BY clause columns.
  Lc : A list containing the values of the current GROUP BY clause columns.
  Lg : ListGroup; contains the rows given by s. The GroupBy Window Slice.
       Thus, Lg is one particular subset (GroupBy Window) of L1. Indexing
       all starts at zero.
  Ldw: ListDateTimeWindow. A list that contains the rows belonging to the
       DateTime Window. Thus, Ldw is one particular subset (DateTime Window)
       of Lg. It is Ldw that is analysed by the pF function.
  There are two windows or slices:
  - The GroupBy Window Slice
  - The DateTime Window Slice.

  In this implementation everything is done sequentially. This can be done faster.
  When a GroupBy Window is detected, you can immediately go to the DateTime Window
  section and find a subset of rows there and feed that to the pF function. You skip
  the GroupBy Window slicing saving part. This speed up things. For a later version.
  """

  # Check if column name to colum index mapping must be done.
  if pLcols != None:
    Ltmp = []                              # temporary list, contains the column indexes
    for i in range(len(pLgb)):
      Ltmp.append(pLcols.index(pLgb[i]))
    pLgb = Ltmp
    pD = pLcols.index(pD[0])

  # Here starts the GroupBy Window Slice detection.
  Ls = []   # list with tuples. Each tuple contains the start/end index of a GroupBy Window Slice
  s1 = 0    # slice index begin
  s2 = 0    # slice index end, watch out; this is an exclusive value
```

```
  # Get all the GroupBy Window slices. They are put in Ls.
  # Lp contains the previous GROUP BY clause values.
  # Lc contains the current GROUP BY clause values.
  Lp = group_by_current(pL1[0], pLgb)     # initialize Lp. The first row
  for i in range(1,len(pL1)) :
    Lc = group_by_current(pL1[i], pLgb)   # the current GROUP BY clause values
    if Lc != Lp :
      s2 = i
      Ls.append((s1,s2))
      s1 = s2                             # start of next slice is end of previous one
      Lp = Lc
  Ls.append((s1,i+1))                     # the slice of the last group

  # Iterate through the GroupBy Window slices. It returns a tuple containing
  # the slice boundary.
  for T in Ls:
    s = slice(T[0],T[1])     # the slice
    Lg = pL1[s]              # ListGroup; contains the GroupBy Window subset

    # We are now going to determine the DateTime Window Slice in Lg.
    cnt   = len(Lg)     # the count, number of rows in the GroupBy Window
    first = 0           # first index of subset in the slice
    last  = 0           # last index of subset in the slice.
    ptr  = -1           # pointer to keep track of the end boundary of the slice

    while last < cnt:
      diff = int((Lg[last][pD] - Lg[first][pD]).total_seconds())
      if diff <= pWs:
        # As long as you are still inside the time range, increase the end slice index.
        last += 1
      else:
        # Ok, you are no longer within the time-range. Now check what your
        # end boundary of the new slice will be. When it has the same
        # end boundary as your previous slice, then skip it. No added value.
        if ptr != last:
          ptr = last
          Ldw = Lg[slice(first, last)]
          pF(Ldw)                         # here you go to the function to analyse your dataset
        # Here you start with a brand new slice within your GroupBy Window Slice
        first += 1
        last = first

    # Get the last one or the only one.
    if ptr != cnt:
      Ldw = Lg[slice(first, last)]
      pF(Ldw)

  # A debug section. Return the GroupBy window slice range and the output list.
  # Note: You can leave this out, and the now need to pass through pL2.
  if pDebug == True:
    for x in Ls: print(x)
    for i, l in  enumerate(pL2):
      if len(l) != 0:
        for x in l: print(i, ': ', x, x[pD])
      else:
        print(i, ': ', l)

def debug_section():
  """
  Debug function. Runs only when this module is executed in stand-alone mode.
  Here you can set up your test data and other testing.
  In this case we want to detect a key that has at least three or more transactions
  within 30 minutes and at at least two different location codes and the sum of
  the transactions most be more than 1000 euro.
  """
  # The input data set.
  # field names: 'k1' - 'k2' - 'amt' - 'd1' - 'loc' - 'ptr'
  # field index:  0      1       2      3       4       5
  L1 = [
       ['G1','Ga', 100, datetime(2019,11,15,20,12,12), 111, 0],
```

```
        ['G1','Ga', 110, datetime(2019,11,15,21,13,12), 112, 1],
        ['G1','Ga', 120, datetime(2019,11,15,22,14,12), 213, 2],
        ['G1','Ga', 600, datetime(2019,11,15,22,15,12), 114, 3],

        ['G1','Gb', 100, datetime(2019,11,15,22,12,12), 22, 4],
        ['G1','Gb', 110, datetime(2019,11,15,22,13,12), 22, 5],
        ['G1','Gb', 120, datetime(2019,11,15,22,14,12), 22, 6],
        ['G1','Gb', 200, datetime(2019,11,15,22,15,12), 22, 7],
        ['G1','Gb', 220, datetime(2019,11,15,22,16,12), 22, 8],
        ['G1','Gb', 600, datetime(2019,11,15,22,17,12), 22, 9],
        ['G1','Gb', 700, datetime(2019,11,15,22,18,12), 22, 10],

        ['G1','Gc', 100, datetime(2019,11,15,22,12,12), 431, 11],
        ['G1','Gc', 150, datetime(2019,11,15,22,13,12), 432, 12],
        ['G1','Gc', 300, datetime(2019,11,15,22,14,12), 433, 13],

        ['G2','Ga', 100, datetime(2019,11,15,22,12,12), 441, 14],
        ['G2','Ga', 200, datetime(2019,11,15,22,13,12), 442, 15],
        ['G2','Ga', 300, datetime(2019,11,15,22,14,12), 443, 16],
        ['G2','Ga', 900, datetime(2019,11,15,22,15,12), 444, 17],

        ['G2','Gb', 10, datetime(2019,11,15,22,12,12), 551, 18],
        ['G2','Gb', 20, datetime(2019,11,15,22,13,12), 552, 29],
        ['G2','Gb', 30, datetime(2019,11,15,22,14,12), 553, 20],
        ['G2','Gb', 40, datetime(2019,11,15,22,15,12), 554, 21],
        ['G2','Gb', 50, datetime(2019,11,15,22,16,12), 555, 22],
        ['G2','Gb', 60, datetime(2019,11,15,22,17,12), 556, 23],
        ['G2','Gb', 70, datetime(2019,11,15,22,18,12), 557, 24],

        ['G2','Gc', 10, datetime(2019,11,15,22,12,12), 661, 25],
        ['G2','Gc', 20, datetime(2019,11,15,22,13,12), 662, 26],
        ['G2','Gc', 50, datetime(2019,11,15,22,14,12), 663, 27],
      ]

  # The DateTime Window size for all rules
  v_dtw  = 30*60     # The DateTime Window. 30 minutes, expressed in seconds

  # Rule 1
  r1_min_txs   = 3
  r1_min_loc   = 2
  r1_max_amount = 1000

  # Rule 2
  r2_min_txs   = 2
  r2_max_amount = 100000

  # Rule 3
  r3_min_txs   = 1
  r3_max_amount = 1000

  """
  The analyze window function. Must be different for each rule, but you can combine
  rules having the same filter criteria, and same DateTime Window. Saves processing time.
  The last record of List DateTime Window is outputted. But that is a choice.
  """
  def analyze_window(pLw):
    len_window = len(pLw)
    if len_window >= r1_min_txs:
      L =  [pLw[i][4] for i in range(len_window)]    # transpose column values into one row, list
comprehension
      cc_locations = Counter(L)                     # cc=counter class
      if len(cc_locations) >= r1_min_loc:           # counts number of unique location codes
        L = [pLw[i][2] for i in range(len_window)]
        if sum(L) >= r1_max_amount:
          L2[0].append(pLw[len(pLw)-1])
    # Rule 2
    if len_window >= r2_min_txs:
      L =  [pLw[i][4] for i in range(len_window)]    # transpose column values into one row, list
comprehension
      if sum(L) >= r2_max_amount:
        L2[1].append(pLw[len(pLw)-1])
```

```
    # Rule 3
    if len_window >= r3_min_txs:
      L =  [pLw[i][4] for i in range(len_window)]    # transpose column values into one row, list
comprehension
      if sum(L) >= r3_max_amount:
        L2[2].append(pLw[len(pLw)-1])

  # Use indexes. In the analyze_window function you also use indexes.
  L2 = []
  for i in range(3):
      L2.append([])       # The output data set; function analyze_window determines the content
  time_window(L1, L2, [0,1], 3, v_dtw, analyze_window, pDebug=True)

  # Uses column names. In the analyze_window function you still have to use indexes.
  L2 = []
  for i in range(3):
      L2.appen            # The output data set; function analyze_window determines the content
  v_cols = ['k1', 'k2', 'amt', 'd1', 'loc', 'ptr']
  v_gb   = ['k1',  'k2']
  v_date = ['d1']
  time_window(L1, L2, v_gb, v_date, v_dtw, analyze_window, pLcols=v_cols, pDebug=True)

# Here the debug_section is executed, when in stand-alone mode.
if __name__ == '__main__':
  debug_section()
```

32 Terminologie

Append	Toevoegen aan het einde van iets.
Assignment	Toekennen.
batch	Een groep commando's die achter elkaar worden uitgevoerd zonder handmatig ingrijpen.
case-insensitive	Hoofdletters worden behandeld als kleine letters; hoofdletterongevoeligheid.
cast	Het omzetten van een item met een bepaald data type naar een item met een ander data type. '*Cast*' is '*gieten*' in het Nederlands. Je giet het in een andere vorm.
class	Een klasse, dat wil zeggen, een beschrijving van de attributen en methodes die een bepaalde groep objecten gemeen hebben.
class call	De aanroep van een methode van een andere class.
command line	De plek waar je in de command shell commando's kunt intypen.
command shell	Een programma dat je toestaat om rechtstreeks met het besturingssysteem van je computer te communiceren door commando's in te typen.
crash	Het plotseling eindigen van een programma doordat er een fout optreedt.
dictionary	Woordenboek. In Python is het een data type dat bestaat uit een verzameling sleutels met bijbehorende waardes, die gevonden kunnen worden als je de sleutel kent. Heet ook wel '*hash table*'; je hakt iets op in stukken en geeft elke stukje een code.
encoding	Versleuteling. Het verwijst ofwel naar de wijze waarop tekens in een bestand zijn opgeslagen (ASCII of Unicode), ofwel naar het vertalen van tekens naar een van die manieren van opslaan.
error	Fout.
exception	Uitzondering. Een exception wordt gegenereerd door een programma tijdens de uitvoering als er iets fout gaat. Je kunt exceptions afvangen in je code, wat '*exception handling*' wordt genoemd.
float	Float is de afkorting voor '*floating-point number*', oftewel een '*gebroken getal*'. Het is tevens de benaming van het data type dat gebroken getallen omvat.
garbage collection	Vuilnisophalen. Het beschrijft het proces dat er automatisch voor zorgt dat geheugen dat niet meer nodig is, wordt vrijgegeven voor hergebruik.
handle	Handvat. Een handle is een variabele die toegang geeft tot een bestand.
inheritance	Overerving. Het is de benaming voor het mechanisme waarbij subclasses de eigenschappen en methodes krijgen van de class waarvan ze zijn afgeleid.

input	De invoer die de gebruiker aan een programma verstrekt door op het toetsenbord te typen, of die uit een bestand wordt ingelezen.
integer	Een geheel getal.
interface	In object georiënteerde programmeertalen: Een class die gebruikt wordt om een definitie vast te leggen, maar die niet gebruikt kan worden om direct instanties van te maken.
iterator	Een functie die items één voor één genereert.
key	Sleutel. Een key wordt gebruikt om een waarde in een dictionary te vinden.
keyword	Een woord dat in Python gereserveerd is voor specifieke taken.
list	Een list is een lijst. Het is een data type.
list comprehension	Lijst begrip. List comprehension is een techniek waarbij je een list maakt door een functionele beschrijving van de list te geven. Bijvoorbeeld, je schrijft niet [1,4,9,16], maar in plaats daarvan schrijf je een Python uitdrukking die je kunt vertalen als: 'alle kwadraten van gehele getallen tussen 1 en 20'.
loop	Lus. Een loop is een iteratie in een programma, meestal via een **while** of een **for**.
newline	Nieuwe regel. In Python voorgesteld als '**\n**'. Geeft als het voorkomt in een string aan dat de tekst vanaf dat punt verder moet gaan op een nieuwe regel.
output	De uitvoer van een programma die op het scherm wordt getoond, of die in een bestand wordt weggeschreven.
overloading	Een deel van de term '*operator overloading*' die inhoudt dat je een betekenis geeft aan een operator voor nieuwe data types. Wordt ook gebruikt bij functies. Je hebt dan '*function overloading*'. De functienaam is hetzelfde, maar de parameters kunnen verschillen.
path	Pad. Een bestandsnaam inclusief de directorystructuur die precies aangeeft waar in het bestandssysteem het bestand zich bevindt.
pickling	Het opslaan in een bestand van een hele datastructuur via de '*pickle*' module.
pointer	Aanwijzer. Een pointer in een bestand is een variabele die refereert aan een specifieke positie in dat bestand.
print	Afdrukken, of druk af. Kan zijn '*afdrukken op papier*' of '*laten zien op het scherm*'.
queue	Een queue is een specifiek soort rij, namelijk een rij (in Python meestal geïmplementeerd als list) waarbij nieuwe elementen aan het einde van de rij worden toegevoegd, en elementen alleen verwijderd mogen worden vanaf het begin van de rij. (Kun je ook zien als een '*wachtrij*').
raise	'*Raise an exception*' betekent het actief genereren van een exception in een programma. Om een exception te genereren, gebruik je het gereserveerde woord **raise**.
root	De wortel van een boomstructuur, ofwel, de hoogstgelegen knoop in een boomstructuur, van waaruit de rest van de boom benaderd kan worden.
runtime error	Een fout die optreedt tijdens het uitvoeren van een programma. Meestal hoort bij een runtime error een specifieke foutmelding van het programma.
statement	Opdracht. Een statement is één programma-regel, bijvoorbeeld een commando of een berekening.
tuple	Tupel. Het is een verzameling van één of meerdere waardes die bij elkaar horen en in een vaste volgorde staan.
underscore	Het '*laag liggende brede streepje*' dat zich op de meeste toetsenborden bevindt op dezelfde toets als het minteken.