

Leren programmeren met Python

Thomas More Kempen
Alain Bex
september 2023

Inhoud

1	Introductie	1
	<i>Over de inhoud</i>	<i>1</i>
	<i>Belangrijke begrippen.....</i>	<i>2</i>
	<i>Aanpak van cursus.....</i>	<i>3</i>
	<i>Oefeningen.....</i>	<i>4</i>
	<i>Multiple-Choice Quiz.....</i>	<i>4</i>
	<i>PowerPoint.....</i>	<i>4</i>
	<i>Locatie van je programma's</i>	<i>4</i>
	<i>Na dit hoofdstuk kan je:.....</i>	<i>5</i>
2	Python gebruiken	6
	<i>Inleiding</i>	<i>6</i>
	<i>Python installeren</i>	<i>6</i>
	<i>Een IDE installeren.....</i>	<i>6</i>
	<i>Namen van voorbeelden en oefeningen.....</i>	<i>6</i>
	<i>Je eerste programma's maken en uitvoeren</i>	<i>7</i>
	<i>Debuggen.....</i>	<i>8</i>
	<i>Case-sensitive</i>	<i>8</i>
	<i>Een Python instructie apart uitvoeren</i>	<i>8</i>
	<i>Na dit hoofdstuk kan je:.....</i>	<i>9</i>
3	Bewerkingen en Output	10
	<i>Inleiding</i>	<i>10</i>
	<i>Cursor.....</i>	<i>10</i>
	<i>Op het scherm zetten : Print</i>	<i>10</i>
	<i>Data types</i>	<i>12</i>
	<i>Bewerkingen.....</i>	<i>12</i>
	<i>Na dit hoofdstuk kan je:.....</i>	<i>14</i>
4	Variabelen en Input	15
	<i>Inleiding</i>	<i>15</i>
	<i>Wat is een variabele?</i>	<i>15</i>
	<i>Een variabele veranderen van inhoud</i>	<i>16</i>
	<i>De naam van een variabele.....</i>	<i>17</i>
	<i>Verkorte operatoren</i>	<i>18</i>
	<i>Gegevens vragen aan de gebruiker met input.....</i>	<i>18</i>
	<i>String omvormen naar een getal.....</i>	<i>19</i>
	<i>Variabelen en de debugger</i>	<i>21</i>
	<i>Commentaar in je programma</i>	<i>21</i>
	<i>Na dit hoofdstuk kan je:.....</i>	<i>22</i>
5	Eenvoudige Functies	23
	<i>Inleiding</i>	<i>23</i>
	<i>Wat is een functie?.....</i>	<i>23</i>
	<i>Parameters van een functie</i>	<i>23</i>
	<i>Basis functies</i>	<i>24</i>
	<i>ASCII-code van een karakter.....</i>	<i>25</i>

	Geneste functies.....	26
	Modules	27
	Na dit hoofdstuk kan je:.....	29
6	Conditie	30
	Inleiding	30
	if-instructie	30
	Soorten condities.....	31
	Boolean variabele : True or False	32
	if ... else.....	33
	if ... elif ... else.....	34
	Geneste if...elif...else	35
	Logische operatoren: and, or en not	35
	Na dit hoofdstuk kan je:.....	36
7	Meerdere keren uitvoeren: loops	37
	Inleiding	37
	while-loop	37
	for loop.....	40
	range(n,m) en range(n,m,k).....	44
	Geneste while, for, if.....	44
	Na dit hoofdstuk kan je:.....	46
8	Functies zelf maken	47
	Inleiding	47
	Een functie definiëren.....	47
	Een functie debuggen.....	48
	Functies met een parameter.....	48
	Functies met meerdere parameters	49
	Functies met parameters met default waarden	49
	Functies die iets "teruggeven" : return	50
	Functies met parameters die iets "teruggeven"	51
	Structuur van een functie.....	52
	De naam van een functie	52
	Lokale en globale variabelen	52
	Na dit hoofdstuk kan je:.....	53
9	Recurisie	54
	Inleiding	54
	Klassiek voorbeeld : faculteit bereken	54
	Klassiek voorbeeld : de rij van Fibonacci.....	55
	Voorbeeld : mappen op je computer	55
	Na dit hoofdstuk kan je:.....	55
10	Strings	56
	Inleiding	56
	Rekenen met strings.....	56
	Strings doorlopen letter per letter.....	56
	String : de index	57
	Methodes/methods : iets doen met strings	58
	Speciale karakter \ voor " , ' , \n	59

	<i>Strings over meerdere regels</i>	59
	<i>Na dit hoofdstuk kan je:</i>	61
11	Lists	62
	<i>Inleiding</i>	62
	<i>Basisbegrippen en stukken van lijsten</i>	62
	<i>Nuttige functies voor lists</i>	64
	<i>Controle op aanwezigheid in de list</i>	64
	<i>Doorlopen van een list</i>	64
	<i>Een list wijzigen via de index of indexen</i>	65
	<i>List operatoren</i>	66
	<i>List methodes</i>	67
	<i>Copy van een list</i>	68
	<i>Geneste lists : oxo</i>	70
	<i>Na dit hoofdstuk kan je:</i>	70
12	Tuples	71
	<i>Inleiding</i>	71
	<i>Basisbegrippen</i>	71
	<i>Tuples bruikbaar in functies</i>	72
	<i>Na dit hoofdstuk kan je:</i>	72
13	Dictionaries	73
	<i>Inleiding</i>	73
	<i>Wat is een dictionary?</i>	73
	<i>Werken met een dictionary:</i>	74
	<i>Na dit hoofdstuk kan je:</i>	76
14	Sets	77
	<i>Inleiding</i>	77
	<i>Basisbegrippen</i>	77
	<i>Een paar nuttige methods</i>	77
	<i>Typische handelingen met verzamelingen</i>	79
	<i>Na dit hoofdstuk kan je:</i>	80
15	De os-module : besturingssysteem commando's	81
	<i>Inleiding</i>	81
	<i>Bestanden en directories</i>	81
	<i>De os-sub-module path</i>	82
	<i>Alle bestanden doorlopen met os.walk</i>	83
	<i>Na dit hoofdstuk kan je:</i>	84
16	Tekstbestanden	84
	<i>Inleiding</i>	84
	<i>Platte tekstbestanden</i>	84
	<i>Testbestand</i>	84
	<i>Bestand : handle en pointer</i>	85
	<i>Bestand volledig lezen met read en sluiten met close</i>	87
	<i>Regels één voor één lezen met readline()</i>	88
	<i>Alle regels lezen in een list met readlines()</i>	88
	<i>Wat gebruiken : read, readline of readlines?</i>	89
	<i>Schrijven in tekstbestanden met write</i>	89

	Schrijven met writelines()	90
	Encoding	90
	CSV bestanden	91
	Na dit hoofdstuk kan je:	93
17	Tijd	94
	Inleiding	94
	Datums	94
	Tijden	95
	Plaatselijke tijdsbenamingen	95
	Duurtijd laten berekenen	96
	Kalender	97
	Na dit hoofdstuk kan je:	97
18	Python uitvoeren vanaf de Command Line	98
	Inleiding	98
	Eenvoudig voorbeeld	99
	Voorbeeld met extras	100
	Na dit hoofdstuk kan je:	101
19	Exceptions	102
	Inleiding	102
	Exception handling	102
	Veel gebruikte exceptions :	104
	Na dit hoofdstuk kan je:	104
20	Object orientatie	105
	Inleiding	105
	Klassen en objecten	105
	Encapsulation	106
	Overloading	106
	Relaties tussen objecten	108
	Polymorfisme en inheritance	109
	Na dit hoofdstuk kan je:	110
21	SQLite en andere DBMSsen	111
	Inleiding	111
	Een databank maken	111
	Records toevoegen in een tabel	112
	Andere DBMSsen	113
	Na dit hoofdstuk kan je:	113
22	GUI applicaties maken	114
	Inleiding	114
	Voorbeeld	114
	Na dit hoofdstuk kan je:	116
23	XML	117
	Inleiding	117
	Wat is XML	117
	Inhoud van een XML bestand:	117
	XML regels	118

	<i>Lezen van XML</i>	119
	<i>Elementen met attributen</i>	120
	<i>Gegevens aanpassen</i>	121
	<i>Elementen maken en verwijderen</i>	121
	<i>Een bestand nieuw aanmaken</i>	122
	<i>XML en Office</i>	123
	<i>Na dit hoofdstuk kan je:.....</i>	123
24	RegEx.....	124
	<i>Inleiding</i>	124
	<i>De module re.....</i>	124
	<i>Na dit hoofdstuk kan je:.....</i>	125
25	Web-sites maken met Python.....	126
26	Besluit	127

1 INTRODUCTIE

Deze cursus is bedoeld voor mensen die willen leren programmeren en die nog niet geprogrammeerd hebben.

De anderen kunnen de cursus gebruiken als middel om hetgeen ze kennen te herhalen. Er is geen enkele voorkennis nodig. Enkel werklust en enthousiasme!

Programmeren is creatief werken!

Eigenlijk is iedere programmeertaal een doos met bouwblokjes. Met die blokjes kan je iets maken. Sommige talen hebben veel blokjes : er zijn heel veel kant en klare bouwstenen. Sommige talen hebben er weinig : waardoor je vele kleinere blokjes moet gebruiken om een grotere te maken.

Doel van deze cursus is vooral om te leren hoe de basisblokjes werken en je hoe je die best combineert om een voor de eindgebruiker werkend programma te bouwen.

Over de inhoud

De onderwerpen zijn zodanig gekozen dat je in theorie enkel Python moet installeren op eender welke computer. Alle modules die je gaat tegenkomen in deze cursus worden standaard samen met Python geïnstalleerd, tenzij het duidelijk wordt aangegeven dat je iets extra moet installeren.

Hoofdstukken 1 tot en met 11 zijn fundamenteel. Ze behandelen begrippen die in iedere programmeertaal gebruikt worden. Enkel het hoofdstuk 9 over recursie kan overgeslagen worden.

Hoofdstukken 12 tot en met 14 gaan over iets complexere datastructuren. Veel externe modules (niet standaard meegeleverd met Python) maken hiervan gebruik.

Hoofdstuk 15 behandelt een aantal operating system specifieke commando's die zeker nuttig van pas komen vanaf het ogenblik dat je met bestanden gaat werken.

Hoofdstuk 16 over tekstbestanden is ook een must en onontbeerlijk voor de meeste nuttige toepassingen en ook vrij algemeen herbruikbaar in andere programmeertalen.

Hoofdstuk 17 gaat over het werken met tijd, zeker nuttig in proces- en planningstoepassingen.

Hoofdstuk 18 : De command-line gebruiken. Deze kennis kan nodig zijn. Zeker als je binnen een bedrijf zaken wil automatiseren en programma's wil schrijven die autonoom hun ding kunnen doen.

Hoofdstuk 19 : 80% van alle geschreven code dient om er voor te zorgen dat een eindgebruiker geen fouten kan maken. Hoe doe je dat in Python. Hoe doe je exception handling in Python?

Hoofdstuk 20 over object oriëntatie moet je zeker ook bekijken. Object oriëntatie, alhoewel niet nodig voor kleinere projecten, absoluut nodig als je programma een zekere omvang gaat aannemen. Bovendien kennen de meeste programmeertalen dezelfde begrippen die hierin voorkomen.

De overige hoofdstukken bevatten informatie over de vele domeinen waar je Python kan gebruiken. De belangrijkste nuttige modules passeren de revue: Databanken, GUI applicaties, XML, RegEx, ...

Deze cursustekst bevat de basiselementen om te starten. Hiermee kan je een werkend programma maken en krijgt je voldoende achtergrond om je verder te verdiepen in de gigantische hoeveelheid materiaal die beschikbaar is op het web. (google python).

Bovendien legt het de basis om een volgende programmeertaal te leren.

Belangrijke begrippen

Scripting : een programma schrijven in een scripting taal.

Een **scripting taal** is een soort programmeertaal.

Een programmeertaal is een formele taal waarmee je opdrachten kan schrijven die een computer kan uitvoeren.

Er zijn heel veel computertalen.

Iedere programmeertaal heeft zijn eigen regels.

Een tekst die met een programmeertaal geschreven is noemen we **code**.

Code die in een programmeertaal geschreven is, kan maar op één manier worden 'begrepen' door de computer.

Een programma schrijven :

je maakt een tekstbestand (of meerdere tekstbestanden) waarin instructies staan die geldig zijn in een programmeertaal.

Je kan dus een programma schrijven met de meest simpele tekstverwerker, met de kladblok, maar er zijn speciale programma's gemaakt om op een gemakkelijke manier programma's in een bepaalde computertaal te maken : **IDE's**.

IDE : Integrated Development Environment.

Een goede IDE :

- laat bepaalde fouten dadelijk zien
- laat zien wat de correcte instructies zijn
- gebruikt kleur om duidelijk informatie te geven
- en nog veel meer.

Een programma uitvoeren (runnen) : we laten de computer de instructies die we geschreven hebben uitvoeren.

Een programma testen : je laat het programma runnen en kijk of het programma doet wat er gevraagd is. Testen is heel belangrijk en vraagt dikwijls veel tijd en geduld. Je moet immers alle mogelijke gevallen controleren.

Een programma debuggen : fouten zoeken en verbeteren

Een programma werkt zelden van de eerste keer juist, er zullen hier en daar wat foutjes inzitten. Die fouten moeten er uit. De fouten er uithalen = **debuggen**.

Waarom moet je leren programmeren:

- programmeren is heel dikwijls een probleem opkappen in kleinere stukjes, in hapklare brokken die je wel kan oplossen.
En dat is nu net een vaardigheid die dikwijls van pas kan komen in het dagelijkse en beroeps-leven: **problemen oplossen** door ze systematisch op te delen in eenvoudiger problemen.
- het is enorm **creatief**: je start met niets en je maakt iets nieuw, iets dat daarvoor niet bestond en jij hebt dat gemaakt en dat doet bovendien nog iets. Prachtig, toch!
- je leert beter **communiceren**, je leert de juiste vragen stellen want je moet iets maken dat iets doet, je moet dus goed begrijpen wat het moet doen en goede vragen stellen aan de opdrachtgever.
- je kan zaken **automatiseren** : moeilijke vervelende repeterende zaken die anders door een mens moeten gedaan worden laat je door een machine doen
 - moeilijke berekeningen
 - zaken die 's nachts moeten gebeuren
 - ...
- je traint de **grijze massa** en dat kan niet slecht zijn

Aanpak van cursus

Ieder hoofdstuk behandelt bepaalde onderwerpen.

Alle onderwerpen worden aangebracht met behulp van voorbeelden.

Tijdens de les worden alle instructies in deze voorbeelden doorlopen en uitgelegd.

Dit gebeurt door stap voor stap de instructies te doorlopen.

In dit document worden de belangrijkste instructies overgenomen samen met de nodige uitleg en achtergrond.

Voor de verschillende programmalijnen wordt een nummer gezet en na het programma krijg je een uitleg voor die lijnen waarvoor bijkomende uitleg nuttig kan zijn.

Een nuttige oefening om je zelf te testen is om een voorbeeld te nemen en het uit te voeren stap voor stap (in debug mode, zie later) terwijl je telkens probeert te voorspellen wat er gaat gebeuren.

Soms vind je bestanden waarvan de naam eindigt met "extra". Deze bevatten dan nog wat meer voorbeelden dan in de cursus en zijn dan ook interessant om te oefenen.

Oefeningen

Bij ieder hoofdstuk horen oefeningen.

Maak er zo veel mogelijk.

Leren programmeren is ervaring opbouwen. Hoe meer oefeningen je maakt, hoe meer fouten je maakt en hoe meer je de volgende keer die fouten herkent en niet meer gaat maken. Alle oefeningen staan in een apart pdf-bestand.

De oefeningen met een * zijn de oefeningen die je minimaal moet kunnen maken.

Dikwijls kan je een voorbeeldbestand gebruiken als uitgangspunt om een oefening op te lossen.

De oplossingen van de oefeningen staan in een apart zip-bestand per hoofdstuk.

Multiple-Choice Quiz

Bij een aantal hoofdstukken hoort een quiz. Het zijn multiple-choice vragen waarin jouw kennis wordt getoetst. Het zijn meestal kleine programma's die iets op het scherm zetten. Jij moet aangeven wat er op het scherm gaat komen.

PowerPoint

Bij een aantal hoofdstukken hoort een PowerPoint.

De PowerPoint bevat een overzicht van de belangrijkste theorie.

Maar deze theorie kan je ook terugvinden in de cursustekst.

De cursustekst is hetgeen wat je moet kennen.

Locatie van je programma's

Programma's zijn bestanden.

Je gaat de programma's die je maakt in deze cursus moeten opslaan.

Je zal ergens een plaats moeten vastleggen waar je al je programma's gaat bewaren.

Dat kan eender waar zijn, je documenten drive, een externe drive, een stick,

Bij de cursus hoort een zip-bestand Cursus_Python_Vxx.zip.

Deze bevat :

Een hoofddirectory met hierin :

- Een pdf bestand met de cursustekst
- Een pdf bestand met de programmeeroefeningen gegroepeerd per hoofdstuk
- Voor alle hoofdstukken uit de cursus een map :
In deze map staan alle voorbeelden die je tijdens de cursus zal zien

Zet ook de oplossingen van de oefeningen in de juiste map zodat je ze later nog terugvind.

Als je deze zipfile ergens unzipped dan heb je onmiddellijk de goede structuur.

Na dit hoofdstuk kan je:

- De volgende begrippen uitleggen
 - scripting
 - scripting taal
 - programmeertaal
 - code
 - een programma schrijven
 - IDE
 - een programma runnen
 - een programma testen
 - een programma debuggen
- 5 redenen geven waarom leren programmeren nuttig is
- 5 voorbeelden geven van nuttige programma's en waarom ze nuttig zijn
- uitleggen welk studiemateriaal er is en het gebruiken
- op je computer een plaats maken waar je het cursusmateriaal kan opslaan en waar je later je oplossingen van je oefeningen kan zetten

2 PYTHON GEBRUIKEN

Inleiding

Om te kunnen programmeren in Python hebben we nodig :

- Python
- een IDE

Python installeren

Ga hiervoor naar: <https://www.python.org/downloads/>

en installeer python op je computer.

Afhankelijk van de cursus kan je hier specifieke instructies voor krijgen.

Een IDE installeren

Er zijn heel veel IDE's.

Afhankelijk van welke cursus je volgt zal er een bepaalde IDE gebruikt worden.

Heel dikwijls is dat Visual Studio Code (VSC) : een gratis IDE die je kan gebruiken voor bijna alle programmeertalen. Link: <https://code.visualstudio.com/>

Een andere is PyCharm : een IDE die specifiek voor Python gemaakt is. Hier is ook een gratis versie van beschikbaar.

Je vindt die zeker terug op het web samen met de nodige installatie instructies.

Ik gebruik meestal VSC omdat je er in verschillende programmeertalen kan programmeren. Je hoeft dus alleen maar die IDE te leren gebruiken.

Namen van voorbeelden en oefeningen

Op gemakkelijk voorbeelden en oefeningen terug te vinden kan je ze een duidelijke naam geven.

In deze cursus hebben alle voorbeelden een naam zoals:

- H01_Voorbeeld_01_hello
- H05_Voorbeeld_01_oppervlakte_cirkel

Deze namen vind je ook terug in de pdf met de cursustekst.

De oefeningen hebben een naam zoals:

- H05_Oefening_03_rekenen
- H10_Oefening_01_list_wissel

Deze namen vind je terug bij iedere oefening in de pdf met oefeningen.

Je eerste programma's maken en uitvoeren

Je gaat nu je eerste Python programma's schrijven en uitvoeren.

Je gaat leren hoe je een programma stap voor stap kunnen laten "runnen" in "debug" mode.

Het doel is niet om exact te weten hoe het programma werkt.

Het doel is dat je kan werken met de IDE.

En om te zien dat Python en de IDE goed geïnstalleerd zijn.

Hello world

Je gaat met de IDE een nieuw bestand maken.

In dat bestand ga je een programma intypen.

Om te beginnen maak je een programma met 1 regel:

```
print("Hello world")
```

Je bewaart dat bestand met de naam : H02_Voorbeeld_01_hello_world.py

In de juiste folder.

Belangrijk : bestanden met een Python programma hebben een extensie py.

Dit is belangrijk voor vele IDE's om te weten met welke programmertaal ze bezig zijn.

Je "runt" dit programma → Je gaat de tekst **Hello world** op het scherm zien komen.

Dit is ons eerste programma, met 1 instructie.

Het zet een tekst op het scherm.

Belangrijk is dat je de verschillende stappen herkent en kan uitvoeren

1. start je IDE
2. maak een nieuw bestand (met de juiste extensie)
3. bewaar het met een goede naam
4. type instructies in het bestand (het programma)
5. bewaar het programma (kan soms automatisch bewaard worden)
6. voer het programma uit

Je kan ook starten met een bestaand programma en dat dan opslaan onder een nieuwe naam. Dat is dikwijls handiger.

Berekening

Maak een nieuw programma met de lijn:

```
print(111111111 * 111111111)
```

en bewaar het onder de naam H02_Voorbeeld_02_berekening.py

Run het. En je ziet op het scherm :

```
1234567899987654321
```

Welkom

Maak een nieuw programma met de volgende 3 lijnen

```
voornaam = input("Geef je voornaam en duw <enter> : ")
boodschap = "We gaan Python leren, " + voornaam + "!"
print(boodschap)
```

en bewaar het onder de naam H02_Voorbeeld_03_welkom.py

Run het.

Debuggen

Iedere IDE moet de mogelijkheid geven om een programma in stapjes te laten uitvoeren: debuggen of runnen in debug mode.

Je vindt dit in de documentatie van de gebruikte IDE.

Ook de term "breakpoint" is belangrijk.

Een breakpoint is een punt in je programma waar het programma stopt.

Als het programma dan stopt dan je de programma-instructies één voor één, lijn voor lijn laten uitvoeren. Zo kan je goed zien of het programma doet wat het moet doen.

Bij de meeste IDE's kan je de programmalijnen aanduiden waar bij het debuggen gestopt moeten worden, zodat je niet het gans programma moet doorlopen als je maar een klein stukje wil testen.

Case-sensitive

Python is case-sensitive. Hoofdletters of kleine letters speelt een rol. Er zijn programmeertalen waarbij dat geen rol speelt.

Een Python instructie apart uitvoeren

In iedere IDE heb je de mogelijkheid om een Python instructie apart uit te voeren.

Dat kan nuttig zijn als je even iets wilt uittesten.

Je vindt dit in de documentatie van de gebruikte IDE.

Probeer dat zeker uit. Het kan handig van pas komen.

Na dit hoofdstuk kan je:

- Python installeren op je computer
- Een IDE installeren op je computer
- python programma's schrijven, saven, runnen
- een programma stap voor stap laten runnen (debug mode)
- breakpoints zetten
- apart een python instructie uitvoeren in de IDE
- ...

3 BEWERKINGEN EN OUTPUT

Inleiding

Je gaat leren

- hoe je gegevens op het scherm kan zetten
- dat er verschillende soorten gegevens zijn
- hoe je kan "rekenen" met verschillende soorten gegevens
- hoe en waarom je commentaar zet in een programma

Cursor

De cursor is de aanwijzer op het beeldscherm van een computer. Er zijn twee cursors:

- De tekstcursor wijst aan waar gegevens van het toetsenbord worden ingevoerd. Hij wordt met de pijltjestoetsen verplaatst. Is er een muis of touchpad, dan kan hij ook verplaatst worden door daarmee in de tekst te klikken.
- De muiscursor of muisaanwijzer geeft aan waar het effect heeft als er geklikt wordt. Wanneer er met de muis of touchpad bewogen wordt, beweegt de cursor mee.

In dit deel van de cursus zullen de Python programma's **enkel de tekstcursor** beïnvloeden.

Op het scherm zetten : Print

H03_Voorbeeld_01_print

```
001 # Voorbeelden van schermoutput
002 print("Hello world")
003 print(11111111)
004 print("choco", "gelei")
005 print("Er zijn", 2, "getallen")
006 print("Een lege tekst ", "", "kan ook!")
007 print("Er zijn", 2, "getallen ! ", end="")
008 print("Maar wat een verschil!")
009 print("Er zijn", 2, "getallen ", end="@")
010 print(" en nu iets speciaals met een @ !")
```

Geeft op het scherm:

```
Hello world
11111111
choco gelei
Er zijn 2 getallen
Een lege tekst   kan ook!
Er zijn 2 getallen ! Maar wat een verschil!
Er zijn 2 getallen @ en nu iets speciaals met een @ !
```


Lijn	Uitleg
001	Als Python een # tegenkomt dat gaat hij alles wat op die lijn achter de # staat niet uitvoeren. # is het commentaar symbool. Je kan het nuttig gebruiken om uitleg te geven over het programma. Uitleg die je later nodig hebt om te begrijpen hoe en waarom je programma werkt. En in een bedrijf vooral om het iemand anders gemakkelijk te maken om te begrijpen hoe en waarom je programma werkt.
002	<p>print is een functie.</p> <p>Een functie is herkenbaar aan een naam (in dit geval print) gevolgd door een ((rond haakje open) dat later gevolgd moet worden door een) (rond haakje sluiten).</p> <p>Tussen die ronde haakjes staat er dan dingen. Die dingen zijn de parameters. Een functie gaat iets doen met de parameters.</p> <p>Hier gaat de functie print de parameters op het scherm zetten op de plaats waar de cursor staat. Als er geen speciale parameter is, gaat de cursor naar de volgende schermregel.</p> <p>Hetgeen geprint moet worden staat hier tussen aanhalingstekens. Dit wil zeggen dat het een tekst is. Die tekst komt dan op het scherm.</p>
003	print zet een getal op het scherm op de plaats van de cursor en de cursor gaat naar de volgende regel
004	<p>print kan meerdere dingen op het scherm zetten, die dingen worden gescheiden door een komma. De komma zorgt er voor dat er een spatie tussen die dingen geplaatst wordt.</p> <p>Hier worden er dus 2 dingen op het scherm gezet gescheiden door een komma.</p>
005	Hier worden er 3 dingen geprint, een tekst, een getal en een tekst, gescheiden door een komma.
006	Je ziet een "" . Dit is een lege tekst. Dit is anders dan " " : een spatie.
007	Als je niet wil dat de cursor bij een print naar de volgende lijn gaat, dan kan aan print een parameter meegeven: end="" zorgt er voor de cursor blijft staan, dat hetgeen achter de end= staat geprint wordt, en dat de cursor opnieuw blijft staan. Dus de volgende print gaat daar vlak achter gebeuren zonder dat er naar een nieuwe regel gegaan wordt.
009	Zoals lijn 009 maar de cursor blijft staan en er wordt een @ geprint

Data types

Je hebt in het vorige voorbeeld al gezien dat er verschillende soorten gegevens zijn.

Python herkent volgende soorten gegeven :

Data type	Uitleg
string	Tekst bestaande uit nul of meerdere tekens omsloten met aanhalingstekens (dubbel of enkel) Voorbeelden: "choco", "New York", "", " ", "\$@#" Belangrijk: Een spatie is ook een teken!
integer	Een integer : een geheel getal (kan ook negatief zijn) Voorbeelden: 7, -23, 0, 666
float	(komt van floating point) Een decimaal getal herkenbaar aan een punt. Voorbeelden: 7.25, -12.5, 0.0, 3.14159 Belangrijk : in Python gebruik je altijd een punt als decimaalteken
boolean	Een "logisch" gegeven waarvan de waarde enkel True of False kan zijn. Je zal die terug tegenkomen in het hoofdstuk over condities.

Bewerkingen

Bewerkingen met getallen

H03_Voorbeeld_02_Bewerkingen

Lijn	Instructie	Op het scherm
001	<code>print(5+3)</code>	8
002	<code>print(5-3)</code>	2
003	<code>print(3-5)</code>	-2
004	<code>print(2*4);print(12345)</code>	8 12345
005	<code>print(2*3*4)</code>	24
006	<code>print(4**3)</code>	64
007	<code>print(5/3)</code>	1.6666666666666667
008	<code>print(5//3)</code>	1
009	<code>print(22%5)</code>	2
010	<code>print(2+3 * 4+5)</code>	19

Lijn	Uitleg
001	Je ziet dat de print functie eerst gaat uitrekenen wat er tussen de ronde haakjes staat. In dit geval een optelling : +
002	Bewerking : aftrekking : -
003	Je ziet dat het resultaat ook negatief kan zijn.

004	Vermenigvuldiging : *. Je ziet hier ook een ; (punt-komma). Je kan 2 instructies op dezelfde lijn zetten als je er een ; tussen zet. Het is niet aangeraden om dat te doen; het maakt de code weer wat moeilijker leesbaar. Maar soms wordt het in deze cursus gebruikt om plaats te sparen.
005	Je kan ook meerdere getallen vermenigvuldigen
006	Machtsverheffing : **. 4 ** 3 is dus 4 * 4 * 4
007	deling : / . Je ziet in het resultaat dat er afgerond wordt. Je ziet dat we 2 integers door elkaar delen en dat het resultaat een float is.
008	de gehele deling : // . Je deelt gewoon en laat alle cijfers na de komma weg. Je kijkt hoeveel keer het tweede getal in het eerste kan.
009	modulo : % . 22 modulo 5. Je zoekt de rest na een gehele deling. 5 gaat 4 keer in 22 en wat blijft er over? 2.
010	als je geen haakjes gebruikt gaat Python de bewerkingen uitrekenen in de wiskundig afgesproken volgorde. Advies : gebruik steeds haakjes indien twijfel mogelijk.

Bewerkingen met strings

Voorbeeld_03_Bewerkingen_strings

Lijn	Instructie	Op het scherm
001	<code>print("Choco" + "pasta")</code>	Chocopasta
002	<code>print("Choco" + " pasta")</code>	Choco pasta
003	<code>print("Choco" + "!" + "pasta")</code>	Choco!pasta
004	<code>print("Choco" + " " + "pasta")</code>	Choco pasta
005	<code>print("Choco");print("pasta")</code>	Choco pasta
006	<code>print("12"+"24")</code>	1224
007	<code>print(3 * "Choco")</code>	ChocoChocoChoco
008	<code>print("pasta" * 3)</code>	pastapastapasta
009	<code>print("choco",end="")</code>	
010	<code>print("pasta")</code>	chocopasta
011	<code>print("choco",end="-")</code>	
012	<code>print("pasta")</code>	choco-pasta
013	<code>print("choco\npasta")</code>	choco pasta
014	<code>print("\n\n\nnn")</code>	n n nn

Lijn	Uitleg
001	Strings optellen met + : de strings worden samengevoegd. (tegen elkaar)
002	Merk de blanco op juist voor pasta. Blanco is ook een teken, een karakter.
006	Laat je niet verleiden tot het antwoord 36. Er worden hier strings opgeteld. Geen getallen!

007	Een string vermenigvuldigen met een getal : zoveel keer de string achter elkaar zetten.
008	Eerst de string en dan het getal mag ook.
009	Als je niets speciaal doet, zal print op het scherm zetten wat tussen de haakjes staat en de cursor zal naar een nieuwe regel gaan. Met de end= parameter zeg je tegen de print-functie : print wat tussen aanhalingstekens staat en laat de cursor staan. Ga dus NIET naar een nieuwe regel op het scherm. Achter de end= staat er niets tussen de aanhalingstekens. De cursor blijft dus gewoon staan.
011	Zoals in lijn 009 , maar hier gaat er eerst nog een streepje geprint worden voor de cursor blijft staan.
013	Als de print functie in een string een \ tegenkomt gaat hij kijken wat achter de \ staat en iets doen afhankelijk van wat daar staat. In dit voorbeeld staar er een n achter: \n. De n van newline : de cursor gaat naar het begin van de volgende regel.

Na dit hoofdstuk kan je:

- print gebruiken om iets op het scherm te zetten
- uitleggen hoe de print functie werkt: wat doet de komma, meerdere dingen printen, ...
- uitleggen wat de cursor is en wat print doet met de cursor
- uitleggen hoe je kan vermijden dat na een print de cursor naar de volgende regel gaat
- opsommen welke verschillende data types er zijn
- uitleggen wat elk data type (string, integer en float) is, met voorbeelden
- de operatoren +, -, *, **, /, //, % uitleggen en gebruiken
- haakjes oordeelkundig gebruiken
- uitleggen welke bewerkingen je met strings kan doen
- uitleggen wat de speciale karakter \n doet in een string bij het printen
-

4 VARIABELEN EN INPUT

Inleiding

in dit hoofdstuk ga je leren wat variabelen zijn.

Een zeer belangrijke bouwsteen in programma's.

Variabelen zijn de containers waarin je gegevens, data kan plaatsen.

Iedere programmeertaal gebruikt variabelen.

Je gaat ook leren hoe je data kan vragen aan de gebruiker en hoe je die data dan verder kan gebruiken in je programma.

Wat is een variabele?

Een variabele is een plaats in het interne computergeheugen (met een naam) waar je iets in kan steken en waar je iets uit kan halen. (een doos met een naam waar je iets kan insteken en terug uithalen)

H04_Voorbeeld_01_variabelen

```
001  getal_1 = 3
002  print("Het eerste getal is", getal_1)
003  getal_2 = 5
004  print("Het tweede getal is", getal_2)
005  produkt = getal_1 * getal_2
006  print(getal_1, "x", getal_2, "is", produkt)
```

Geeft op het scherm:

Het eerste getal is 3

Het tweede getal is 5

3 x 5 is 15

Lijn	Uitleg
001	Links van de = : maak een variabele (een plaats in het geheugen), geef die variabele de naam getal_1. Rechts van de = : wat moet ik in de variabele steken? In dit geval het getal 3.
002	print gaat iets op het scherm zetten. 2 dingen. Eerst een string : "Het eerste getal is ", dan een spatie (van de komma) en dan getal_1 : de inhoud van de variabele getal_1. Als Python een naam tegenkomt van een variabele die hij al kent dan haalt hij op wat in die variabele zit.
005	Links van de = : maak een variabele (een plaats in het geheugen), geef die variabele de naam product. Rechts van de = : wat moet ik in de variabele product steken? In dit geval moet ik een bewerking uitvoeren. Ophalen wat er in getal_1 zit, ophalen wat er in getal_2 zit, en die getallen vermenigvuldigen. En dat resultaat komt in de gereserveerde plaats in het geheugen

009 print gaat iets op het scherm zetten. 5 dingen: eerst de waarde van getal_1, dan een string, dan de waarde van getal_2, dan een string en tenslotte de waarde van product

Onthou:

Met behulp van = maak je een variabele met een naam aan de linkerkant van de = en geef je er een waarde aan met behulp van hetgeen rechts van de = staat. Als je de waarde van een variabele wil ophalen, dan doe je dat via de naam van die variabele.

H04_Voorbeeld_02_variabelen_strings

```
001 woord = "choco"
002 print("Het woord is", woord)
003 aantal_keer = 3
004 print("Aantal keer:", aantal_keer)
005 print("Resultaat:", aantal_keer * woord)
```

Geeft op het scherm:

```
Het woord is choco
Aantal keer: 3
Resultaat: chocochochocho
```

Een variabele veranderen van inhoud

Tijdens de loop van een programma kan de inhoud van een variabele veranderen.

H04_Voorbeeld_03_variabele_veranderen

```
001 woord = "choco"
002 print("Het woord is", woord)
003 woord = "banaan"
004 print("Het woord is nu", woord)
005 woord = woord + woord
006 print("Het woord is nu", woord)
007 woord = 3 * woord
008 print("Het woord is nu", woord)
```

Geeft op het scherm:

```
Het woord is choco
Het woord is nu banaan
Het woord is nu banaanbanaan
Het woord is nu banaanbanaanbanaanbanaanbanaanbanaan
```

Lijn	Uitleg
001	Links van de = : maak een variabele (een plaats in het geheugen), geef die variabele de naam woord. Rechts van de = : wat moet Python in de variabele steken? In dit geval de string "choco".
003	Links van de = : maak een variabele (een plaats in het geheugen), geef die variabele de naam woord. Rechts van de = : wat moet Python in de variabele steken? In dit geval de string "banaan". Maar dat wil wel zeggen dat Python de waarde "choco" van vorige keer niet meer kent. In de variabele woord zit nu een nieuwe waarde!
005	Links van de = : maak een variabele (een plaats in het geheugen), geef die variabele de naam woord. Rechts van de = : wat moet Python in de variabele woord steken? In dit geval moet ik een bewerking uitvoeren. Ophalen wat er momenteel in de variabele woord zit en optellen bij hetgeen momenteel in de variabele woord zit. Dat levert op : "banaanbanaan" En dat resultaat komt terecht in de variabele woord, waardoor de vorige waarde van de variabele woord weer weg is.

Onthou : meestal ga je aan de rechterkant van het = teken een bewerking, een berekening uitvoeren. Het resultaat van die bewerking, die berekening komt dan terecht in de variabele waarvan de naam aan de linkerkant van het = teken staat.

De naam van een variabele

Het is belangrijk dat je goede namen geeft aan variabelen.

Onderstaand programma is niet goed.

H04_Voorbeeld_04_namen_niet_goed

```
001 a= 3.1415
002 b = 5.27
003 c = b * b * ai
004 print(c)
```

Het volgende programma is beter:

```
001 pi = 3.1415
002 straal = 5.27
003 oppervlakte = straal * straal * Pi
004 print("De oppervlakte van een cirkel met straal",straal,"is",oppervlakte)
```

Bij het tweede programma zie je dadelijk dat het programma de oppervlakte van een cirkel berekent. Je ziet duidelijk welke formule gebruikt wordt.

Goede namen geven informatie over de inhoud van de variabele.

Dit maakt een programma leesbaarder en dus gemakkelijker te begrijpen.

Je kan afspraken maken rond de naamgeving van variabelen.

In deze cursus is zo veel mogelijk conform de 'Google Style Guide' voor Python:

- namen van variabelen beginnen met een letter
- namen van variabelen bestaan uitsluitend uit kleine letters, cijfers en underscores (_)
- er worden geen afkortingen gebruikt : vb23 is geen goede naam
- underscores worden gebruikt om woorden te scheiden
- "De vlag dekt de lading", gebruik dus (lange) betekenisvolle namen

Verkorte operatoren

Als je een bewerking wil doen met een variabele en het resultaat ervan terug in diezelfde variabele willen steken dan kan je gebruik maken van de verkorte notatie van de bewerking.

H04_Voorbeeld_06_verkort

Lijn	Instructie	Op het scherm
001	<code>getal = 3</code>	
002	<code>print("Getal :", getal)</code>	Getal : 3
003	<code>getal +=1</code>	
004	<code>print("Getal :", getal)</code>	Getal : 4
005	<code>getal *=2</code>	
006	<code>print("Getal :", getal)</code>	Getal : 8
007	<code>getal -=6</code>	
008	<code>print("Getal :", getal)</code>	Getal : 2
009	<code>getal **=5</code>	
010	<code>print("Getal :", getal)</code>	Getal : 32
011	<code>getal //=4</code>	
012	<code>print("Getal :", getal)</code>	Getal : 8
013	<code>getal %=3</code>	
014	<code>print("Getal :", getal)</code>	Getal : 2

Lijn	Uitleg
003	dit is hetzelfde als de instructie : <code>getal = getal + 1</code>
005	hetzelfde als <code>getal = getal * 2</code>
007	hetzelfde als <code>getal = getal - 6</code>
009	hetzelfde als <code>getal = getal ** 5</code>

Gegevens vragen aan de gebruiker met input

In de eerste voorbeelden krijgen de variabelen een waarde door een instructie in het programma. Meestal met een `=`.

Voor de meeste programmeerproblemen zullen die waarden van buiten het programma moeten komen.

Een belangrijke bron van gegevens is de gebruiker. Hij zal iets moeten kunnen intypen. En hetgeen hij intypt mag niet verloren gaan, dus zullen we het in een variabele moeten steken.

H04_Voorbeeld_07_input

```
001  voornaam = input("Geef je voornaam en duw <enter> : ")
002  boodschap = "We gaan Python leren, " + voornaam + "!"
003  print(boodschap)
```

Geeft op het scherm: (als je Marie zelf intypt en dan op <enter> duwt)

Geef je voornaam en duw <enter> : Marie
We gaan Python leren, Marie!

Lijn	Uitleg
001	Je ziet aan de rechterkant van de = input . Input is ook weer een functie (je ziet de ronde haakjes) input gaat hetgeen tussen de ronde haakjes staat op het scherm zetten en gaat dan wachten. Tot je een aantal karakters hebt ingetypt (een string) en dan op <enter> gedrukt hebt. Input geeft als resultaat de string die je ingetypt hebt. En dat resultaat gaat dan terecht komen in de variabele die links van de = staat. In de variabele voornaam.
002	Maakt een variabele en zet daar een string in waarvan een deel de string is die je hebt ingegeven.

String omvormen naar een getal

H04_Voorbeeld_08_zonder_float

```
001  eerste_getal = input("Geef een getal en duw op <enter> : ")
002  tweede_getal = input("Geef een getal en duw op <enter> : ")
003  product = eerste_getal * tweede_getal
004  print(getal_1, "x", getal_2, "is", product)
```

Geeft op het scherm iets in de aarde van:

```
Geef een getal en duw op <enter> : 33
Geef een getal en duw op <enter> : 24
Traceback (most recent call last):
  File "d:\ThomasMore\OneDrive - Thomas More\Documenten\00. Python
WIP\Python_V09\H04. Variabelen en
Input\H04_Voorbeeld_08_zonder_float.py", line 3, in <module>
    product = eerste_getal * tweede_getal
              ~~~~~^~~~~~
TypeError: can't multiply sequence by non-int of type 'str'
```

In het voorbeeld in de vorige paragraaf heb je geleerd dat de functie `input` steeds een string oplevert. Namelijk de string die je ingetypt hebt voor dat je op <enter> duwde. In lijn 003 van het programma probeer je dus 2 strings te vermenigvuldigen. Dat kan Python niet en dus geeft hij een foutboodschap.

Als je de gebruiker met `input` een getal wil laten ingeven, waarmee je iets wil gaan bereken dan moet je die "input" omvormen naar een getal. Want met een string kan je geen berekeningen uitvoeren.

H04_Voorbeeld_09_met_float

```
001 eerste_getal = input("Geef een getal en duw op <enter> : ")
002 tweede_getal = input("Geef een getal en duw op <enter> : ")
003 getal_1 = float(eerste_getal)
004 getal_2 = float(tweede_getal)
005 product = getal_1 * getal_2
006 print(getal_1, "x", getal_2, "is", product)
```

Geeft op het scherm:

```
Geef een getal en duw op <enter> : 5.2
Geef een getal en duw op <enter> : 3.4
5.2 x 3.4 is 17.68
```

Lijn	Uitleg
001	vraagt iets aan de gebruiker en zet dat in de variabele <code>eerste_getal</code> . <code>Eerste_getal</code> is dus een string
003	de functie <code>float</code> zet hetgeen tussen de ronde haakjes staat om naar een float. In de variabele <code>getal_1</code> komt dus een float. iets waar je mee kan rekenen. Als je weet dat het gehele getallen zijn die ingegeven worden, kan je de functie <code>int</code> gebruiken.
005	hier gebeurt de berekening met de 2 floats; als we de input niet hadden omgevormd naar getallen, dan hadden we op deze lijn een foutboodschap gekregen.
006	het resultaat komt op het scherm

Meestal gebeurt deze omvorming samen met de input:

H04_Voorbeeld_10_input_float

```
001 eerste_getal = float(input("Geef een getal en duw op <enter> : "))
002 tweede_getal = float(input("Geef een getal en duw op <enter> : "))
003 product = eerste_getal * tweede_getal
004 print(eerste_getal, "x", tweede_getal, "is", product)
```

Lijn	Uitleg
001	Je kan dadelijk de float functie laten werken op de input. Dus eerst gebeurt de
002	vraag naar input, dan die input omvormen. Deze manier van werken wordt meer gebruikt.

Variabelen en de debugger

Iedere goede IDE voorziet de mogelijkheid om op ieder ogenblik te zien wat de waarde van de variabelen zijn. Dit is onmisbaar om goed te kunnen zien wat je programma doet.

Zoek dus zeker op hoe je dat met de IDE die jij gebruikt kan doen en doorloop de verschillende voorbeelden met de debugger en zie hoe de variabelen veranderen en zie welke extra informatie de debugger nog laat zien.

Commentaar in je programma

Commentaar in een programma is tekst waar de computer geen rekening mee houdt. Hij gaat niets uitvoeren als hij commentaar tegenkomt.

Je schrijft commentaar in je programma om uit te leggen hoe je programma werkt. Dit is niet alleen nuttig voor andere mensen die je programma moeten kunnen begrijpen.

Maar ook voor jezelf: als je een paar maanden later je programma terugziet, is de kans groot dat je niet meer exact weet waarom je bepaalde dingen op bepaalde manieren geprogrammeerd hebt. Commentaar helpt je dan.

Nog een andere reden is dat je bij het testen van het programma soms even niet wil dat bepaalde instructies niet uitgevoerd worden. Dan maak je even van deze instructies commentaar. En later maak je er terug instructies van.

Er zijn een paar manieren om commentaar aan te duiden:

- Als je een # teken zet in een lijn , dan zal Python **alles negeren** wat op die lijn rechts van dat teken staat.
- Python negeert alles wat tussen `"""` (3 keer ") en de volgende `"""` staat.
- Python negeert alles wat tussen `'''` (3 keer ') en de volgende `'''` staat.

H04_Voorbeeld_11_commentaar

```

001 #-----
002 # H04_Voorbeeld_10_commentaar |
003 #-----
004 # Zo kan je een volledige lijn als commentaar vastleggen
005 # Soms ook handig als je de lijn even niet wil laten uitvoeren
006 print("Hello World") # Ook gewoon achter een instructie
007 """
008 Als je veel commentaar hebt,
009 kan je die spreiden over meerdere regels met behulp

```

010	van 3 dubbele aanhalingstekens.
011	"""
Lijn	Uitleg
001-	Iedere lijn begint met een #
005	Dus iedere lijn is commentaar. Python doet hier niets mee.
006	De lijn start met een instructie die door Python wordt uitgevoerd. De # achter die instructie geeft aan dat hetgeen volgt commentaar is.
007	Python weet nu dat alles wat volgt tot de volgende """ commentaar is en dus niet moet uitgevoerd worden.

Na dit hoofdstuk kan je:

- uitleggen wat een variabele is
- een variabele een goede naam geven
- uitleggen hoe je een variabele een waarde geeft
- uitleggen hoe je de waarde van een variabele ophaalt
- rekenen met variabelen
- de verkorte operatoren += -= *= /* uitleggen, lezen en gebruiken
- instructies schrijven die input vragen van de gebruiker
- uitleggen hoe de input functie werkt
- alle voorbeelden doorlopen met de debugger en zien hoe de variabelen gemaakt worden en veranderen van inhoud
- 2 manieren geven waarmee je commentaar kan schrijven in je programma
- Commentaar herkennen in een programma
- 3 redenen opgeven waarom je commentaar zou schrijven in een programma

5 EENVOUDIGE FUNCTIES

Inleiding

Je kan ondertussen een programma schrijven.

In dit hoofdstuk ga je zien hoe je met een aantal kant en klare '**functies**' je programmeermogelijkheden aanzienlijk kan uitbreiden.

Deze functies zijn beschikbaar vanaf het ogenblik dat je python installeert.

Sommige zijn dadelijk bruikbaar. Voor sommige heb je extra instructies nodig.

Wat is een functie?

- Een functie is een blokje programmacode dat je kan oproepen. (waarvan je de code zelfs soms niet kan zien)
- Een functie heeft **een naam**.
- Met die naam kan je de functie **oproepen** met die naam
- Achter de naam staan ronde haakjes
- Soms staat er iets tussen die haakjes : parameters
- Als je die functie oproept dan **doet** die functie iets
- Meestal doet die functie iets met die parameters
- Soms geeft de functie **iets terug**. (**returns** iets)

Parameters van een functie

Soms kan je iets meegeven aan de functie : de **parameters** van de functie.

Die parameters zet je **tussen ronde haakjes** achter de naam van de functie.

Als er meerdere parameters zijn, zet je er een **komma** tussen.

In de volgende voorbeelden wordt het resultaat van functies ge"print".

print is zelf een functie en gaat eerst hetgeen tussen de ronde haakjes staat uitrekenen en het resultaat op het scherm zetten.

H05_Voorbeeld_01_functies

```
001 getal_1 = 25
002 getal_2 = 2.73
003 beleg = "choco"
004 print(max(getal_1, getal_2))
005 print(min(getal_1, getal_2))
006 print(round(getal_2))
007 print(len(beleg))
008 print(float("3.14"))
009 print(int("3"))
010 boodschap = str(getal_1) + " " + beleg + "-potten"
011 print(boodschap)
```

Geeft op het scherm:

```
25
2.73
3
5
3.14
3
25 choco-potten
```

Basis functies

Hier volgt een lijst van basis functies. Sommige functies kunnen meer dan dat je hier vindt, maar deze worden herhaald en uitgebreid in de volgende hoofdstukken.

Naam	Parameters	Returns	Uitleg
type	iets	het data type van de parameter	returns het data type van de parameter
float	iets	een float	returns de parameter maar omgevormd naar float
int	iets	een integer	returns de parameter maar omgevormd naar integer
str	iets	een string	returns de parameter maar omgevormd naar het string type
abs	een getal	een numerieke waarde	de parameter als hij positief is en – de parameter als hij negatief is
max	1 of meerdere getallen	een getal	het maximum van de parameters
min	1 of meerdere getallen	een getal	het minimum van de parameters
pow	2 getallen	een getal	de eerste parameter verheven tot de macht gelijk aan de tweede parameter
round	getal	een getal	de parameter wordt afgerond
len	string of ...	een getal	het aantal karakters van de string
input	string	string	zet de parameter op het scherm en wacht tot de gebruiker iets intypt en ENTER duwt. Input returns hetgeen de gebruiker heeft ingetypt
print	1 of meerdere	niets	zet de parameters op het scherm

De functies float, int en str zijn **type-casting functies** : ze vormen een bepaalde soort gegeven om naar een ander soort gegeven.

ASCII-code van een karakter

Alles in een computer wordt opgeslagen in geheugen met behulp van nullen en enen: BITS. Of in groepjes van 8 bits : BYTES.

Codering : afspraken hoe je tekens, karakters gaat op slaan in het interne of externe geheugen van de computer.

Een codering die door (vrijwel) ieder systeem ondersteund wordt is de ASCII codering. Ieder karakter wordt opgeslagen als een getal : de ASCII-code van dat getal.

Je ziet er een aantal in volgende tabel: in het blauwe vak de code en rechts ervan de karakter die er mee overeenkomt.

40	(50	2	60	<	70	F	80	P	90	Z	100	d	110	n	120	x
41)	51	3	61	=	71	G	81	Q	91	[101	e	111	o	121	y
42	*	52	4	62	>	72	H	82	R	92	\	102	f	112	p	122	z
43	+	53	5	63	?	73	I	83	S	93]	103	g	113	q	123	{
44	,	54	6	64	@	74	J	84	T	94	^	104	h	114	r	124	
45	-	55	7	65	A	75	K	85	U	95	_	105	i	115	s	125	}
46	.	56	8	66	B	76	L	86	V	96	`	106	j	116	t		
47	/	57	9	67	C	77	M	87	W	97	a	107	k	117	u		
48	0	58	:	68	D	78	N	88	X	98	b	108	l	118	v		
49	1	59	;	69	E	79	O	89	Y	99	c	109	m	119	w		

Het is deze codering waarmee je op een toetsenbord toch een karakter kan intypen zonder dat je die toets ziet op het toetsenbord. Als je ALT ingedrukt houdt en dan een getal intypt, dan zal je het teken zien verschijnen dat overeenkomt met dat getal. (probeert eens met ALT 125)

De ASCII-code is zeer belangrijk. In het hoofdstuk over condities ga je strings moeten vergelijken. Je zal zien dat strings vergeleken worden op basis van de ASCII-codes van de karakters van die strings.

Er zijn ook andere coderingen. Een hele bekende is UTF-8. UTF-8 is een tekencodering met variabele lengte: niet elk teken gebruikt evenveel bytes. Afhankelijk van het teken worden 1 tot 4 bytes gebruikt. Voor de 128 ASCII-teken is slechts één byte nodig, waarvan de numerieke waarde gelijk is aan de ASCII-code. Voor alle andere tekens zijn twee of meer bytes nodig.

Python heeft een paar functies waarmee je kan werken met ASCII-waarden

Naam	Parameters	Returns	Uitleg
ord	een karakter	een integer	de ASCII-code van het parameter
chr	integer	een karakter	de karakter met ASCII-code = de parameter

H05_Voorbeeld_02_ascii

Lijn	Instructie	Op het scherm
001	<code>print(ord("a"))</code>	97
002	<code>print(ord("A"))</code>	65
003	<code>print(ord(" "))</code>	124
004	<code>print(chr(66))</code>	B
005	<code>print(chr(98))</code>	b
006	<code>print(chr(126))</code>	~
007	<code>print(ord(chr(67)))</code>	67
008	<code>print(chr(ord("X")))</code>	X

Geneste functies

Een functie kan ook binnen een functie gebruikt worden: je kan functies "nesten".

H05_Voorbeeld_03_nest

001	<code>getal_1 = 25;getal_2 = 2.5</code>
002	<code>beleg_1 = "choco";beleg_2 = "gelei"</code>
003	<code>print(type(max(beleg_1,beleg_2)))</code>
004	<code>print(str(min(getal_1,getal_2)))</code>
005	<code>print(len(str(min(getal_1,getal_2))))</code>
006	<code>print(type(len(str(min(getal_1,getal_2)))))</code>
007	<code>print(round(float("3.14")))</code>
008	<code>print(round(float("3.14"),1))</code>

Geeft op het scherm:

```
<class 'str'>
2.5
3
<class 'int'>
3
3.1
```

Lijn	Uitleg
003	Als functies genest worden, worden ze afgewerkt van binnen naar buiten : eerst gaan we max laten werken, op dat resultaat laten we type werken en dat resultaat laten we printen.
008	je ziet dat je bij de round functie nog een extra parameter kan meegeven, deze extra parameter geeft aan tot op hoeveel cijfers na de komma er moet afgerond worden

Modules

Python bevat een aantal functies die je onmiddellijk kant en klaar kan oproepen. Voorbeelden hiervan heb je zojuist gezien.

Maar als je Python installeert, installeer je ook een aantal zogenaamde "modules".

Een module is een verzameling functies.

Om zo'n module te kunnen gebruiken moet je die eerst **importeren** met het commando `import`.

Als de module ge-importeerd is, kan je de functies in die module oproepen door eerst de naam van de module te schrijven, daarna een punt en daarna de naam van de functie.

de module math

Deze module bevat een aantal wiskundige functies.

H05_Voorbeeld_04_math

```
001 import math
002 getal = float(input("Geef een geheel getal : "))
003 vierkantswortel_getal = math.sqrt(getal)
004 boodschap = "De vierkantswortel van " + str(getal) + " is " \
005           + str(vierkantswortel_getal)
006 print(boodschap)
007 print(math.pi)
```

Geeft op het scherm:

```
Geef een geheel getal : 18
De vierkantswortel van 18.0 is 4.242640687119285
3.141592653589793
```

Lijn	Uitleg
001	Je moet eerst een import doen van een module om ze te kunnen gebruiken.
003	Je wil de vierkantswortel berekenen, hiervoor heb je de functie <code>sqrt</code> nodig. Deze functie kan je oproepen door <code>math.sqrt</code> te gebruiken. Je gebruikt de naam van de module gevolgd door een <code>.</code> en daarna de functie die je wil gebruiken.
004	Je ziet op het einde van deze regel een <code>\</code> . Je kan die gebruiken als je vindt dat de programmaregel wat te lang wordt. Als Python op het einde van een regel een <code>\</code> tegenkomt, dan zal Python de volgende regel in plaats van die <code>\</code> zetten. Het is dus een middel om lange regels te spreiden over meerdere regels.
007	De <code>math</code> module bevat dus ook constanten. Zoals hier het getal <code>pi</code>

de module random

Deze module bevat functies die heel nuttig kunnen zijn als je spelletjes wil programmeren waarbij je onvoorspelbare resultaten wil laten genereren. Onder andere : willekeurige getallen generen.

Naam	Parameters	Returns	Uitleg
random	geen	een float	returns een willekeurig getal tussen 0 en 1
randint	2 integers	een integer	returns een willekeurige integer tussen de 2 parameters, inclusief de parameters

H05_Voorbeeld_05_random

```
001 import random
002 dobbelsteen = random.randint(1,6)
003 print("Ik gooi met de dobbelsteen een",doppelsteen, ".")
```

Geeft op het scherm:

Ik gooi met de dobbelsteen een 3 .

En dit resultaat zal dus onvoorspelbaar zijn.

Lijn	Uitleg
004	Je moet eerst een import doen van een module om ze te kunnen gebruiken.
005	Je wil een dobbelsteen nabootsen. Je wil dus een willekeurig getal tussen 1 en 6, 1 en 6 inbegrepen. Dus binnen de module random gebruik je de functie randint met 2 parameters.
006	Zet het resultaat op het scherm. Telkens je het programma runt kan er een andere waarde voor de dobbelsteen tevoorschijn komen.

Om je programma te kunnen testen is het dikwijls handig dat als er willekeurige getallen gegenereerd worden dat dat bij iedere run van het programma dezelfde getallen zijn.

Je kan Python een instructie geven waardoor hij dat doet.

H05_Voorbeeld_06_random_seed

```
001 import random
002 random.seed(666)
003 dobbelsteen = random.randint(1,6)
004 print("Ik gooi met de dobbelsteen een",doppelsteen, ".")
```

Lijn	Uitleg
002	Na deze instructie zal Python willekeurige getallen genereren op basis van het getal 666. Als je dit programma laat lopen dan zal de worp van de dobbelsteen steeds dezelfde zijn. Neem je in plaats van 666 een ander getal, dan krijg je een andere reeks willekeurige getallen.

Na dit hoofdstuk kan je:

- uitleggen wat een functie is
- uitleggen wat parameters van een functie zijn
- uitleggen dat een functie iets teruggeeft met een voorbeeld
- uitleggen waarom een functie nuttig is
- uitleggen met een voorbeeld wat type casting functies zijn en hoe ze werken
- uitleggen wat ASCII code is en de bijhorende functies ord en chr gebruiken
- uitleggen met een voorbeeld wat geneste functies zijn en hoe ze werken
- de werking uitleggen van een aantal functies en deze functies gebruiken
- uitleggen wat een module is en het nut ervan
- via een module wiskundige functies gebruiken
- via een module willekeurige getallen genereren

6 CONDITIES

Inleiding

Tot nu toe werden altijd alle lijnen van je programma uitgevoerd. Van boven naar beneden, lijn per lijn, iedere lijn.

Je kan er ook voor zorgen dat code alleen wordt uitgevoerd als aan **een conditie, een voorwaarde** voldaan is.

if-instructie

H06_Voorbeeld_01_geslaagd

```
001 score = int(input("Geef je score en duw <enter> "))
002 print("-----")
003 if score > 9 :
004     print("Je bent geslaagd.")
005     print("Proficiat !!!")
006 print("----- einde programma -----")
```

Lijn	Uitleg
001	Vraagt input en vormt de input string om naar int want er moet gerekend worden.
003	Tussen de if en de : staat een conditie. Als die conditie waar is : voer de hierna volgende inspringende lijnen uit (004 en 005) Als de conditie niet waar is : ga verder met lijn 006

Structuur van de if instructie, de if blok :

```
if <conditie> :
    <inspringende instructies>
<vervolg programma>
```

<conditie> kan maar 2 mogelijk waarden hebben: waar of niet waar.
Python heeft hier speciale waarden voor. True en False.

Als de waarde van de conditie True is worden alle <inspringende instructies> uitgevoerd

Als de waarde van de conditie False is gaan het programma verder bij <vervolg programma>

Je ziet dat lijnen **004** en **005 opgeschoven, ge-intendeert** zijn!

<inspringende instructies> : is specifiek voor de programmeertaal Python : het inspringen is **verplicht**.

Op die manier zie je ook onmiddellijk waar die instructies bijhoren.
 In vele andere programmeertalen mag je opschuiven, maar je moet niet.
 Je krijgt daar wel het advies om op te schuiven omdat dan de code veel leesbaarder worden.

Python verplicht het opschuiven waardoor automatisch je code leesbaarder wordt.

Belangrijk :

- vergeet de : niet na de conditie
- zorg dat de <inspringende instructies> "inspringen"
- <inspringende instructies> kunnen meerdere instructies op meerdere lijnen zijn.

Soorten condities

Dingen vergelijken met elkaar

Symbol	Uitleg
==	is gelijk aan
>	is groter dan
<	is kleiner dan
>=	is groter dan of gelijk aan
<=	is kleiner dan of gelijk aan
!=	is niet gelijk aan

Het resultaat van een vergelijking is ofwel **True** ofwel **False**.

H06_Voorbeeld_02_vergelijken

Lijn	Instructie	Op het scherm
001	<code>import random</code>	
002	<code>getal = 666;woord = "choco"</code>	
003	<code>print(3 > 4)</code>	False
004	<code>print(getal > 5)</code>	True
005	<code>print(3 != (8-5))</code>	False
006	<code>print(round(3.14159) > 3.14)</code>	False
007	<code>print(3 ** 2 >= 2 ** 3)</code>	True
008	<code>print(True);print(False)</code>	True False
009	<code>print("a" > "b")</code>	False
010	<code>print(woord <= "gelei")</code>	True
011	<code>print("choco" <= "chocopasta")</code>	True
012	<code>print(woord == "choco")</code>	True
013	<code>print(3 * woord > woord)</code>	True
014	<code>print("appelen">"peren")</code>	False
015	<code>print("kleiner">"groter")</code>	True
016	<code>print("10 m">"100 m")</code>	False
017	<code>print("9 m">"10 m")</code>	True
018	<code>a = random.randint(1,6)</code>	
019	<code>print(a > getal)</code>	False
020	<code>print("True" > "False")</code>	True

Lijn	Uitleg
005	Voor je gaat vergelijken moeten eerst alle bewerkingen uitgevoerd worden
009	strings met elkaar vergelijken : vergelijk de eerste karakter van beide strings, is de ene groter dan de andere (op basis van de ASCII code) dan is de ene string groter dan de andere.
011	strings met elkaar vergelijken : vergelijk de eerste karakter van beide strings, is de ene groter dan de andere (op basis van de ASCII code) dan is de ene string groter dan de andere. Zijn ze gelijk dan neem je de tweede karakter van beide strings en je vergelijkt, enz.. Als er geen karakters meer voorhanden zijn, is de langste string de "grootste". Zoals in het voorbeeld (" choco " <= " chocopasta ") Maar dit geldt dus enkel als je eerste alle karakters één voor één vergeleken hebt en ze gelijk waren in beide strings. Het is dus zeker niet zo dat de langste string altijd de "grootste" is.

Conditie : is aanwezig in : de in-operator

De **in-operator** test of iets binnen iets anders zit.

Voorlopig kan je dat gebruiken om te testen of een string te vinden is binnen een andere string. Later ga je soorten gegevens zien waar je dat ook kunt, zoals testen of iets aanwezig is in een lijst.

H06_Voorbeeld_03_is_aanwezig.py

Lijn	Instructie	Op het scherm
001	<code>print("t" in "Python")</code>	True
002	<code>print("to" in "Python")</code>	False
003	<code>print("th" in "Python")</code>	True
004	<code>print("p" in "Python")</code>	False
005	<code>print("P" in "Python")</code>	True
006	<code>print("ho" in woord)</code>	True
007	<code>test = "hoc"</code>	
008	<code>print(test in woord)</code>	True

Boolean variabele : True or False

Je kan een conditie ook in een variabele steken: dat is dan een variabele van het type **boolean**. Zo'n variabele kan enkel de waarden True of False hebben.

Deze variabele kan je dan gebruiken in een conditie.

H06_Voorbeeld_04_boolean

Lijn	Instructie	Op het scherm
001	<code>leeftijd = 23</code>	
002	<code>volwassen = (leeftijd > 18)</code>	
003	<code>if volwassen :</code>	
004	<code> print("OK, je bent volwassen!")</code>	OK, je bent volwassen!

Lijn	Uitleg
002	Volwassen is een boolean variabele
003	En op die manier wordt de conditie weer wat leesbaarder.

if ... else

H06_Voorbeeld_05_if_else

```

001 score = int(input("Geef je score en duw <enter> "))
002 print("-----")
003 if score > 9 :
004     print("Je bent geslaagd.")
005     print("Proficiat !!!")
006 else:
007     print("Je bent niet geslaagd.")
008     print("Spijtig !!!")
009 print("----- einde programma -----")

```

Lijn	Uitleg
003	Als <code>score > 9</code> dan voeren we lijnen 004 en 005 uit. Als dat NIET zo is voor we lijnen 007 en 008 uit.

Structuur

```

if <conditie> :
    <inspringende instructies>
else :
    <inspringende instructies>
<vervolg programma>

```

De if-else gebruik je als je **1 conditie** hebt.

Als die True is doe je het ene, anders doe je het andere.

Achter de else mag dus geen conditie staan.

Als de conditie True is worden alle **<inspringende instructies>** na de if: -lijn uitgevoerd en daarna gaat het programma verder bij <vervolg programma>

Als de conditie False is worden alle **ingesprongen** <instructies> na de else:-lijn uitgevoerd en daarna gaat het programma verder bij <vervolg programma>

Belangrijk :

- vergeet de : niet na de conditie
- vergeet de : niet na de else
- zorg dat de instructies "inspringen"
- instructies kunnen meerdere instructies op meerdere lijnen zijn

if ... elif ... else

H06_Voorbeeld_06_if_elif_else

```

001 score = int(input("Geef je score en duw <enter> "))
002 print("-----")
003 if score > 9 :
004     print("Je bent geslaagd.")
005     print("Proficiat !!!")
006 elif score == 9 :
007     print("Je mag herkansen.")
008     print("Succes !!!")
009 else:
010     print("Je bent niet geslaagd.")
011     print("Spijtig !!!")
012 print("----- einde programma -----")

```

Lijn	Uitleg
003	Als de conditie op lijn 006 false is, gaat het programma naar lijn 006 : een nieuwe conditie. Als die False is wordt lijn 012 uitgevoerd.

Structuur

```

if <conditie> :
    <inspringende instructies>
elif <conditie> :
    <inspringende instructies>
else :
    <inspringende instructies>
<vervolg programma>

```

Het programma controleert eerst de conditie achter de if.

Als die False is dan controleert het programma de conditie achter de eerste elif.

Als die False dan controleert het programma de conditie achter de tweede elif.

Enzovoort.

Als een conditie True is, worden de <inspringende instructies> uitgevoerd die er achter staan.

Tenslotte, als geen enkele conditie True is, voert het programma de <inspringende instructies> achter de else uit.

Na iedere uitvoering van de <inspringende instructies> na if, elif of else, gaat het programma verder met <vervolg programma>

De elif gebruik je dus als je meerdere condities hebt.

Geneste if...elif...else

H06_Voorbeeld_07_geneste_if

```

001 score = int(input("Geef je score en duw <enter> "))
002 print("-----")
003 if score > 9 :
004     print("Je bent geslaagd.")
005     print("Proficiat !!!")
006 elif score == 9:
007     herkansing_gedaan = input("Herkansing gehad <j> of <n> ? ")
008     if herkansing_gedaan == "n" :
009         print("Je mag herkansen.")
010         print("Succes !!!")
011     else:
012         print("Je mag niet meer herkansen.")
013         print("Je bent niet geslaagd.")
014         print("Spijtig !!!")
015 else:
016     print("Je bent niet geslaagd.")
017     print("Spijtig !!!")
018 print("----- einde programma -----")

```

Lijn	Uitleg
008	binnen deze elif staat opnieuw een if-else: een if binnen een if : een geneste if. Je ziet dat de tweede if opgeschoven is ten opzichte van de eerste. En dat lijnen 009 en 010 nog verder opgeschoven zijn. Op die manier kan je heel snel zien tot welke if welke lijn hoort.
012	de instructies op lijnen 012 en 013 spingen ook in, maar iets meer dan de instructies die horen bij de vorige if. Hoe dieper de if genest is, hoe verder de instructies inspringen. Dit wil ook zeggen dat je ook niet te diep mag nesten, want dan wordt het opnieuw onoverzichtelijk.

Logische operatoren: and, or en not

Je kan condities samenstellen. Hiervoor kan je logische operatoren gebruiken.

H06_Voorbeeld_08_logische operatoren

```

001 antwoord = input("Type een <j> of een <n> en duw <enter> ")
002 print("-----")
003 if (antwoord == "j") or (antwoord == "n") :
004     print("Dat heb je goed gedaan!")
005 else:
006     print("Ik vroeg een <j> of een <n> !")
007     print("Graag even opletten !!!")
008 print("----- einde programma -----")

```

Lijn	Uitleg
003	(antwoord == "j") or (antwoord == "n") is een samengestelde conditie en or is een logische operator. In dit voorbeeld moet er dus één van de condities True zijn om de samengestelde conditie True te maken;

Hieronder vindt je de meest gebruikte logische operatoren met hun werking.

	Uitleg
and	Als and tussen condities staat, is het resultaat True als alle condities True zijn; anders is het resultaat False.
or	Als or tussen condities staat, is het resultaat True als één van de condities True is ; het resultaat is alleen False als alle condities False zijn.
not	not kun je voor een conditie plaatsen om de conditie om te keren van True naar False en vice versa.

H06_Voorbeeld_09_logische_and

```

001 antwoord = input("Type een cijfer en duw <enter> ")
002 print("-----")
003 if (antwoord in "0123456789") and (len(antwoord) == 1):
004     print("Dat heb je goed gedaan!")
005 else:
006     print("Ik vroeg een cijfer !")
007     print("Graag even opletten !!!")
008 print("----- einde programma -----")

```

Na dit hoofdstuk kan je:

- Uitleggen hoe je instructies conditioneel uitvoert
- voorbeelden geven van condities
- condities herkennen
- uitleggen hoe de in-conditie werkt
- de in-conditie gebruiken
- vergelijkings-condities gebruiken
- uitleggen wat een boolean variable is
- een boolean variabele gebruiken
- uitleggen hoe de if-blok werkt
- uitleggen hoe de if-else-blok werkt
- uitleggen hoe de if-elif-else-blok werkt
- uitleggen wat een geneste if is
- de if instructie gebruiken in een programma
- de verschillende logische operatoren opsommen en gebruiken

7 MEERDERE KEREN UITVOEREN: LOOPS

Inleiding

Heel dikwijls moet je dezelfde instructies meerdere keren laten uitvoeren. Soms weet je exact het AANTAL KEER dat je dat moet doen, maar soms moet je iets blijven doen ZOLANG er aan een bepaalde voorwaarde, conditie voldaan is.

Python heeft 2 instructies om iets meerdere keren te laten uitvoeren :

Uitleg	
while	lets doen zolang aan een conditie voldaan is.
for	lets doen voor ieder 'ding' uit een reeks van 'dingen' Gebruik dit als je weet dat iets een bepaald aantal keer moet gebeuren.

while-loop

H07_Voorbeeld_01_while_toets_NOK

```

001 toets = input( "Type de letter j en duw <enter> : ")
002 aantal_pogingen = 1
003 while (toets != "j") :
004     toets = input( "Type de letter j en duw <enter> : ")
005     aantal_pogingen = aantal_pogingen +1
006 print("Je had ",aantal_pogingen," pogingen nodig!")
007 print( "----- einde programma ----- ")

```

Kan op het scherm geven:

```

Type de letter j en duw <enter> : n
Type de letter j en duw <enter> : J
Type de letter j en duw <enter> : j
Je had 3 pogingen nodig!
----- einde programma -----

```

Lijn	Uitleg
001	Vraagt aan de gebruiker om iets in te typen en <enter> te duwen. Hetgeen hij intypt komt in de string variabele toets.
002	Maakt een variabele om het aantal pogingen bij te houden.
003	Doe zolang : Achter de while (tot aan de :) staat de conditie (toets != "j"). Dus zolang de conditie (toets != "j") True is, zullen alle ingesprongen instructies (op lijn 004 en 005) uitgevoerd worden en keert Python terug naar lijn 003, waar hij dan opnieuw de conditie gaat checken. Enzovoort. Als de conditie False wordt gaat het programma verder met lijn 006
004	Vraag opnieuw om iets in te geven. Deze lijn is belangrijk. Immers als er niets gaat veranderen aan de variabele toets, dan blijft de while conditie hetzelfde en gaan we in een "oneindige loop"

H07_Voorbeeld_02_while_niet_nul

De opgave is om aan de gebruiker getallen te laten invoeren : als hij 0 invoert stopt het programma met getallen vragen en moet hij het aantal getallen kunnen zien dat hij ingegeven heeft en de som van al die getallen.

```

001 n_getallen = 0
002 som = 0
003 getal = float(input( "Type een getal en duw <enter> (0 = STOP): "))
004 while getal != 0 :
005     n_getallen = n_getallen + 1
006     som = som + getal
007     getal = float(input( "Type een getal en duw <enter> (0 = STOP): "))
008 print( "----- ")
009 print( "Resultaat:  ")
010 print( "----- ")
011 print( "Aantal getallen: ",n_getallen)
012 print( "Som :  ", som)
013 print( "----- einde programma ----- ")

```

Kan op het scherm geven:

```

Type een getal en duw <enter> (0 = STOP): 5
Type een getal en duw <enter> (0 = STOP): 666
Type een getal en duw <enter> (0 = STOP): 0
-----
Resultaat:
-----
Aantal getallen:  2
Som :  671.0
----- einde programma -----

```

Lijn	Uitleg
001	er wordt aan de gebruiker gevraagd om een getal in te typen en <enter> te duwen. Hetgeen hij intypt wordt omgevormd naar een float en dat resultaat komt in de variabele getal.
005	achter de while (tot aan de :) staat de conditie <code>getal != 0</code> Dus zolang de conditie <code>getal != 0</code> True is, zullen alle ingesprongen instructies (op lijn 005 , 006 en 007) uitgevoerd worden en daarna keert Python terug naar lijn 004 , waar hij dan opnieuw de conditie gaat checken. Enzovoort. Als de conditie False wordt gaat het programma verder met lijn 008

H07_Voorbeeld_03_while_som_kleiner

De opgave is om aan de gebruiker gehele getallen te laten invoeren tot de som van de getallen die hij ingevoerd heeft over een bepaalde grens gaat. In dit voorbeeld 100. Als dat gebeurt geeft het programma de som van de ingevoerde getallen.

Je heb dus al zeker een variabele som nodig.

som : de som van alle tot nog toe ingevoerde getallen.

```
001 som = 0
002 getal = int(input( "Geef een geheel getal en duw <enter> "))
003 som = som + getal
004 while (som < 100) :
005     print( "Je hebt ",getal, "ingegeven. ")
006     print( "De som is dus nu ", som, ", dus lager dan 100 ")
007     getal = int(input( "Geef een geheel getal en duw <enter> "))
008     som += getal
009 print( "De som is nu ", som, "en dus stopt het programma ")
```

Kan op het scherm geven:

```
Geef een geheel getal en duw <enter> 45
Je hebt 45 ingegeven.
De som is dus nu 45 , dus lager dan 100
Geef een geheel getal en duw <enter> 66
De som is nu 111 en dus stopt het programma
```

Lijn	Uitleg
001	Je zorgt voor een startwaarde voor de som van de getallen : in het begin is de waarde van som = 0.
002	Er wordt aan de gebruiker gevraagd om een getal in te typen en <enter> te duwen. Hetgeen hij intypt wordt omgevormd naar een integer en dat resultaat komt in de variabele getal.
003	Na de eerste input moet je de nieuwe som al gaan berekenen. Het kan immers zijn dat de gebruiker al dadelijk een getal ingeeft dat groter is dan 100.
004	Achter de while (tot aan de :) staat de conditie som < 100 Dus zolang die conditie True is, zullen alle ingesprongen instructies (op lijn 005-008) uitgevoerd worden. Daarna keert Python terug naar lijn 004, waar hij dan opnieuw de conditie gaat checken. Enzovoort. Als de conditie False wordt gaat het programma verder met lijn 009 .
007	De gebruiker moet het volgende getal invullen.
011	De nieuwe som wordt berekend. Lijnen 007 en 008 zijn identiek aan lijnen 002 en 003 . Dit is een patroon dat je vaak ziet als er aan de gebruiker minstens 1 keer iets moet gevraagd worden. Zijn invoer moet aan een voorwaarde voldoen en het programma blijft hetzelfde vragen tot aan die voorwaarde voldaan is.

Structuur van een while-blok

```
while <conditie> :
    <inspringende instructies>
<vervolg programma>
```

Werking :

- als Python de while tegenkomt, dan kijk hij achter de while en verwacht hij een <conditie> (tussen de while en het dubbele punt).
- Als die <conditie> True is, voert hij alle ingesprongen <instructies> uit en gaat hij terug naar de lijn met de while waar hij opnieuw de <conditie> gaat controleren
- Als die <conditie> False is, gaat hij verder met <vervolg programma>.

Veel voorkomende fouten:

- de dubbele punt vergeten
- de test op gelijkheid met = in plaats van ==
- in de ingesprongen instructies moet er iets gebeuren waardoor de conditie verandert want anders blijft de computer die instructies uitvoeren, wat we dan een **ONEINDIGE LUS** noemen

for loop

Dikwijls moet een programma een bepaald aantal keer iets doen, of iets doen met een **aantal** "dingen", met een **reeks van** dingen:

- je moet een aantal getallen optellen
- je moet alle binnengekomen facturen nummeren
- je moet alle accispunten in een netwerk scannen
- je moet voor alle cursisten een invulformulier maken
- je moet alle foto's taggen

Een veel gebruikt woord voor zo'n reeks dingen is een **collectie**

Zo'n collectie zie je op verschillende manieren verschijnen.

Met behulp van een for loop kan je een collectie **één voor één** alle dingen in de collectie aflopen.

een string als collectie

Een string is een collectie. Een collectie van letters.

Dus kan je een for gebruiken om één voor één alle letters van die string te doorlopen:

H07_Voorbeeld_04_for_letters_in_string

De opgave is om de gebruiker een woord te vragen en dan alle letters van dat woord onder elkaar te printen.

```

001 woord = input("Geef een woord en duw <enter> : ")
002 for letter in woord :
003     print(letter)
004 print("Einde programma.")

```

Kan op het scherm geven:

```

Geef een woord en duw <enter> : kaas
k
a
a
s
Einde programma.

```

Lijn	Uitleg
004	de gebruiker moet iets ingeven, hetgeen hij ingeeft komt in de string variabele woord
005	Hier is de collectie de string woord. Deze collectie bestaat uit de opeenvolgende letters in de variabele woord. Achter de for staat een variabele : letter. De for-loop gaat één voor één de letters uit de variabele woord doorlopen en ze één voor één in de variabele letter zetten. Dus de eerste keer zal in de variabele letter de eerste letter zijn.
003	Zet de inhoud van de variabele letter op het scherm. En ga daarna terug naar lijn 002 om het volgende ding uit de collectie te halen (in dit geval de volgende letter) Als lijn 002 ziet dat alle dingen uit de collectie doorlopen zijn gaat het programma naar lijn 004 .

een range als collectie

Als je in je programma een bepaald aantal keer iets moet uitvoeren dan gebruik je in Python een range.

Een range is dikwijls een reeks opeenvolgende gehele getallen.

Bijvoorbeeld in **range(5)** zitten gehele getallen startend met 0 en eindigend met 4.

Heel vervelend: beginnen bij 0 en niet bij 1, niet tot 5, maar eentje minder.

Maar hoeveel getallen zitten er in? 5 en dat is weer logisch.

Dus je kan range gebruiken in een for-blok. Een range is een collectie.

H07_Voorbeeld_05_for_aantal_keer_printen

De opgave is om de gebruiker een woord te vragen en dan 3 keer dat woord met een lijntje er onder af te printen.

```

001 woord = input("Geef een woord en duw <enter> : ")
002 for teller in range(3) :
003     print(woord)
004     print("-" * len(woord))
005 print('----- einde programma -----')
```

Kan op het scherm geven:

```

Geef een woord en duw <enter> : chpco
chpco
-----
chpco
-----
chpco
-----
----- einde programma -----
```

Lijn	Uitleg
002	Hier is de collectie range(3). De collectie bestaat uit de opeenvolgende getallen 0,1,2 . De for gaat één voor één deze getallen doorlopen en ze één voor één in de variabele teller zetten. En voor ieder van die getallen worden lijnen 003 en 004 (de ingesprongen lijnen) uitgevoerd. Na lijn 003 en 004 gaat het programma terug naar lijn 002 om het volgende getal uit de range in de variabele teller te steken. Als alle getallen van de collectie doorlopen zijn gaat het programma verder met lijn 005

Een kleine aanpassing aan het programma zorgt er voor dat de variabele teller ook op het scherm komt en dat die variabele ook gebruikt wordt :

H07_Voorbeeld_06_for_printen_met_teller

```

001 woord = input("Geef een woord en duw <enter> : ")
002 for teller in range(5) :
003     print(teller,")",woord * teller)
004 print("----- einde programma -----")
```

Kan op het scherm geven:

```

Geef een woord en duw <enter> : choco
0 )
1 ) choco
2 ) chocochoco
```



```

3 ) chocochochoco
4 ) chocochochocochocho
----- einde programma -----

```

Het volgende programma berekent de som van de eerste 100 gehele getallen.

H07_Voorbeeld_07_for_som_100_getallen

```

001  hoogste_getal = 100
002  som = 0
003  for teller in range(hoogste_getal + 1) :
004      som = som + teller
005      print(som)
006  print("De som van de eerste",hoogste_getal,"getallen is",som)
007  print("----- einde programma -----")

```

Het volgende programma vraagt aan de gebruiker 10 getallen en print daarna de som van die 10 getallen af.

Stel dat je 5 getallen wil vragen aan de gebruiker en dat je dan de som van die getallen op het scherm moeten zetten. Dat kan met het volgende programma.

H07_Voorbeeld_08_meerdere_getallen_inputten

```

001  som = 0
002  for teller in range(5) :
003      getal = int(input("Geef een getal en duw <enter> : "))
004      som = som + getal
005  print("De som van alle ingevoerde getallen is",som)
006  print("----- einde programma -----")

```

Geeft op het scherm:

Structuur van een for-blok

```

for <variabele> in <collectie> :
    <inspringende instructies>
<vervolg programma>

```

Werking:

Als Python een for tegenkomt dan verwacht hij achter de for een <variabele>.

Achter die variabele verwacht Python een in

En achter de in verwacht Python een <collectie> (een hoop dingen).

En daarachter de klassieke `:`.

Python neemt het eerste "ding" uit de <collectie> en steekt het in <variabele> en voert de <inspringende instructies> uit.

Daarna keert Python terugkeert naar de lijn met `for`.

Dan haalt Python het tweede ding uit de <collectie> en steekt het in <variabele> en voert de < inspringende instructies> uit en dan het volgende ding tot en met het laatste ding uit de <collectie> waarna voor de laatste keer < inspringende instructies> uitgevoerd worden. Daarna gaat Python verder met <vervolg programma>

Dus hoeveel keer gaat Python de < inspringende instructies> uitvoeren? Evenveel keer als het aantal 'dingen' in de <collectie>.

Gaat Python telkens dezelfde < inspringende instructies> uitvoeren? Jazeker! Maar het resultaat zal soms anders zijn omdat heel dikwijls de waarde van de <variabele> gebruikt wordt in de < inspringende instructies>. En die <variabele> verandert telkens.

range(n,m) en range(n,m,k)

Je kan range ook gebruiken met 2 parameters : range(5,10) is de collectie van alle gehele getallen startend bij 5 en eindigend bij 9.

En zelfs met 3 parameters, de derde parameter geeft de "stapgrootte" aan : range(5, 20, 3) : je start bij 5 : het volgende getal is 5+3=8, het volgende 8+3=11, enzovoort, tot 20, maar zoals zo dikwijls 20 er niet bij.

H07_Voorbeeld_09_range_n_m_k

```
001 for getal in range(4, 14, 2) :
002     print(getal, end=" ")
003 print()
004 print("----- einde programma -----")
```

Geeft op het scherm:

```
4 6 8 10 12
----- einde programma -----
```

Geneste while, for, if

Meestal ga je de instructies met condities en loops moeten combineren om een werkend programma te maken. Je gaat instructies moeten nesten.

H07_Voorbeeld_10_for_in_for

```
001 for getal_1 in range(1,11) :
002     for getal_2 in range(1,11) :
003         print(getal_1 * getal_2, end=" ")
004     print()
005 print("----- einde programma -----")
```

H07_Voorbeeld_11_while_in_for

```

001 import random
002 aantal_pogingen = 10
003 for teller in range(aantal_pogingen):
004     dobbelsteen = random.randint(1,6)
005     print(dobbelsteen, end="")
006     while dobbelsteen != 6 :
007         print("-", end="")
008         dobbelsteen = random.randint(1,6)
009         print(dobbelsteen, end="")
010     print()
011 print("*** einde programma ***")

```

H07_Voorbeeld_12_if_in_for

```

001 import random
002 aantal_keer_gooien = 10
003 aantal_zessen = 0
004 for teller in range(aantal_keer_gooien):
005     dobbelsteen = random.randint(1,6)
006     print("Ik werp een : ",dobbelsteen)
007     if dobbelsteen == 6:
008         aantal_zessen += 1
009 print("Ik gooide",aantal_keer_gooien,"keer en er
010 waren",aantal_zessen,"zessen bij")
011 print("*** einde programma ***")

```

H07_Voorbeeld_13_if_in_for_in_while

```

001 woord = input( "Geef een woord en duw <enter> (niet ingeven = stoppen) : ")
002 klinkers = "aeouiy"
003 while (woord != "") :
004     nieuwe_woord = ""
005     for letter in woord :
006         if letter not in klinkers :
007             nieuwe_woord = nieuwe_woord + letter
008     print("Nieuwe woord = ", nieuwe_woord)
009     woord = input( "Geef een woord en duw <enter> (niet ingeven = stoppen)
010 : ")
011 print("----- einde programma -----")

```

Je merkt ook wel op dat je in het nesten niet te ver mag gaan. De code moet nog leesbaar blijven.

Na dit hoofdstuk kan je:

- uitleggen dat een string een collectie is
- de structuur van de while en de for instructie blokken uitleggen
- uitleggen wanneer je while en wanneer je for gebruikt
- uitleggen wat range is, wat de betekenis is van de verschillende parameters die je aan range kan geven
- de while en de for instructies gebruiken
- de range in combinatie met de for instructie gebruiken.
- de verschillende controle-instructies in elkaar nesten
- ...

8 FUNCTIES ZELF MAKEN

Inleiding

Als je in je programma meerdere keren dezelfde instructies aan het schrijven bent, is het dikwijls interessant om zelf een functie te maken met die instructies.

Een functie is een blokje code dat je een naam geeft. Je kan die functie oproepen via de naam van die functie. Het blokje code dat bij die naam hoort wordt dan uitgevoerd. Op die manier moet je dat stukje code slechts op één plaats hebben, wat het een stuk gemakkelijker maakt als je later aanpassingen moet doen.

Een functie definiëren

H08_Voorbeeld_01_functie

```
001 def groet() :
002     print(" Hallo, hoe is het er mee ? ")
003 groet()
004 print("--- einde programma ---")
```

Geeft op het scherm:

```
Hallo, hoe is het er mee ?
--- einde programma ---
```

Lijn	Uitleg
001	Een functie definitie start steeds met def gevolgd door de naam van de functie (in dit voorbeeld is de naam groet) met vlak daarachter een haakje open. In dit voorbeeld onmiddellijk gevolgd door een haakje dicht. (je ziet later dat er soms wel iets tussen die haakjes kan staan). De regel eindigt (zoals zo dikwijls) met een dubbelpunt.
002	Hier start de code die hoort bij de functie groet. Deze code is ingesprongen. Hiermee geef je aan dat die code hoort bij de definitie van de functie.
003	Hier start ons programma pas. Als je het programma uitvoert dan zal dit de eerste lijn zijn die uitgevoerd wordt. De lijn roept de functie op : je gebruikt de naam van de functie gevolgd door haakje open en haakje dicht. Het programma zal dus "springen" naar lijn 002 en de code uitvoeren die daar staat en daarna terug "springen" naar lijn 004 .

Dus eender waar in je programma waar je deze groet nodig hebt roep je de functie groet op. Bovendien als je een aanpassing wil doen aan de groet moet je dat maar op één plaats doen.

Een functie debuggen

Als je met de IDE je programma gaat debuggen, lijn per lijn, dan zullen de meeste IDE's een speciale knop voorzien om naar die functie te springen. Als je de "normale" knop gebruikt om verder te gaan, zal de functie onmiddellijk volledig uitgevoerd worden. Je ziet dus niet welke instructies de functie uitvoert. Als je weet wat de functie doet wat ze moet doen, dan hoeft dat ook niet. Maar als je, zeker in het begin, de instructies van de functie wil zijn in debug, dan kan je die speciale knop gebruiken om op het juiste ogenblik naar die functie te "springen". Je zal dan zien welke instructies er uitgevoerd worden en als alle instructies uitgevoerd zijn zal je zien dat er teruggesprongen wordt vlak achter de instructie die de functie heeft opgeroepen.

Functies met een parameter

H08_Voorbeeld_02_functie_parameter

```
001 def groet(naam) :
002     print(" Hallo, hoe is het er mee,", naam, "?")
003 groet("Jos")
004 groet("Marie")
005 groet(666)
006 print("--- einde programma ---")
```

Geeft op het scherm:

```
Hallo, hoe is het er mee, Jos ?
Hallo, hoe is het er mee, Marie ?
Hallo, hoe is het er mee, 666 ?
--- einde programma ---
```

Lijn	Uitleg
001	Het verschil met het vorige voorbeeld is dat er hier nu de naam van een variabele tussen de haakjes staat. Als je de functie oproept (zoals in lijn 003 en 004) moet je iets meegeven. De functie heeft 1 parameter, namelijk naam. Als de functie wordt opgeroepen wordt deze variabele vervangen door de waarde die meegegeven wordt in de oproep.
002	Hier start de code die hoort bij de functie groet. Deze code wordt uitgevoerd en de waarde van naam, die doorgegeven werd in de oproep, wordt gebruikt.
003	De functie wordt opgeroepen. Tussen de haakjes staat "Jos". Dus Python springt naar de functie groet en zet in de variabele naam van de functie de waarde "Jos".
004	De functie wordt opnieuw opgeroepen, maar nu met de waarde "Marie".

Functies met meerdere parameters

H08_Voorbeeld_03_functie_2_parameters

```

001 def groet(naam, lengte) :
002     if lengte == "Lang" :
003         print(" Hallo, hoe is het er mee,", naam, "?")
004     else:
005         print(" Hoeist,", naam, "?")
006 groet("Jos", "Lang")
007 groet("Marie", "Kort")
008 print("--- einde programma ---")

```

Geeft op het scherm:

```

Hallo, hoe is het er mee, Jos ?
Hoeist, Marie ?
--- einde programma ---

```

Lijn	Uitleg
001	Je ziet dat de functie 2 parameters heeft: naam en lengte. Als je de functie oproept moet je dus 2 dingen meegeven tussen de ronde haakjes.
006	De functie wordt opgeroepen met 2 parameters. Dus bij het springen naar de functie wordt de variabele naam opgevuld met "Jos" en de de variabele lengte met "Lang"

Functies met parameters met default waarden

H08_Voorbeeld_04_functie_default

```

001 def groet(naam, lengte="Lang") :
002     if lengte == "Lang" :
003         print(" Hallo, hoe is het er mee,", naam, "?")
004     else:
005         print(" Hoeist,", naam, "?")
006 groet("Jos")
007 groet("Marie", "Kort")
008 print("--- einde programma ---")

```

Geeft op het scherm:

```

Hallo, hoe is het er mee, Jos ?
Hoeist, Marie ?
--- einde programma ---

```

Lijn	Uitleg
001	De functie heeft ook weer 2 parameters. Maar tussen haakjes wordt er nu aan de variabele lengte een waarde gegeven. Door aan lengte te waarde "Lang" te geven geef je aan dat die waarde moet gebruikt worden als bij het oproepen geen waarde wordt opgegeven voor die parameter.
006	De functie wordt opgeroepen zonder parameter voor lengte. Dus de functie zal uitgevoerd worden "Lang" als waarde voor de variabele lengte.

Functionies die iets "teruggeven" : return

De functies die je tot hier hebt gezien, zijn functies die iets "doen". Ze zetten iets op het scherm. Je kan ze dus gewoon oproepen en dan doen ze wat ze moeten doen. Ze geven geen resultaat terug dat je verder in je programma kan gebruiken. Ze geven geen resultaat dat je in een variabele kan steken. Heel veel functies moeten dat wel doen.

H08_Voorbeeld_05_functie_return

```

001 import random
002 def gooi_dobbelsteen() :
003     dobbelsteen = random.randint(1,6)
004     return dobbelsteen
005 for teller in range(4) :
006     worp = gooi_dobbelsteen()
007     print("Ik werp een ", worp)
008 print("----- einde programma -----")

```

Kan op het scherm geven:

```

Ik werp een  1
Ik werp een  4
Ik werp een  5
Ik werp een  6
----- einde programma -----

```

Lijn	Uitleg
001	Importeert de module random, om functies te kunnen gebruiken om willekeurige getallen te genereren.
002	Start de definitie van de functie gooi_dobbelsteen. De functie heeft geen parameters.
004	De functie zal de waarde die achter return teruggeven aan de oproeper van de functie.
005	Hier begint het programma. Doe 4 keer.
006	Roept de functie op een zet hetgeen de functie teruggeeft in de variabele worp. De functie wordt hier gebruikt aan de rechterkant van het = teken en zal dus een waarde opleveren die terecht komt in de variabele die aan de linkerkant van het = teken staat.

Dit voorbeeld geeft ook mooi het nut weer van een functie.

Als je de functie hebt gemaakt, dan moet je niet meer nadenken over hoe je die module ook al weer moet gebruiken, de juiste naam van de randint functie en welke parameters je ook al moest opgeven.

Wil je een dobbelsteen gooien : roep gewoon `gooi_dobbelsteen` op.

Funcities met parameters die iets "teruggeven"

In dit voorbeeld kan je de functie oproepen met een parameter tussen de haakjes en geeft de functie iets terug.

Heel dikwijls wordt een functie hiervoor gebruikt. Je geeft iets aan de functie en de functie geeft iets terug.

H08_Voorbeeld_06_oppervlakte_cirkel

```

001 import math
002
003 def oppervlakte_cirkel(r) :
004     oppervlakte = r * r * math.pi
005     return oppervlakte
006
007 straal = float(input(\
008     "Geef de straal van het grasperk\n\
009     en duw <enter> : "))
010
011 oppervlakte_perk = oppervlakte_cirkel(straal)
012
013 print("De oppervlakte van een grasperk met straal",\
014     straal,"is",oppervlakte_perk)
015 print("----- einde programma -----")

```

Kan op het scherm geven:

Geef de straal van het grasperk

en duw <enter> : 5.4

De oppervlakte van een grasperk met straal 5.4 is 91.60884177867838

----- einde programma -----

Lijn	Uitleg
001	Importeer math, om later de waarde van pi op te halen.
003	Start de definitie van de functie oppervlakte_cirkel. De functie heeft 1 parameter : r.
005	Geeft de waarde van de variabele oppervlakte terug aan de oproeper van de functie.
011	Rechts van de = wordt de functie opgeroepen. De waarde van de variabele straal wordt meegegeven aan de functie. Hetgeen teruggegeven wordt door de functie komt in de variabele oppervlakte_perk terecht.

Structuur van een functie

```
def <functie_naam>( <parameters> ) :
    <ingeschoven instructies>
    return <functieresultaat>
```

waarbij er dus soms geen <parameters> zijn en soms geen return <functieresultaat>

De naam van een functie

Een functie moet een naam hebben. Volg dezelfde afspraken als de afspraken voor de namen van variabelen.

Geef een naam die duidelijk aangeeft wat de functie doet.

Geef als het kan een naam die begint met een actie.

Lokale en globale variabelen

Scope van een variabele: waar in het programma blijft die variabele zijn waarde behouden.

Als je in een functie een nieuwe variabele aanmaakt, dan zal deze variabele alleen maar zichtbaar zijn binnen die functie. Bij het terugkeren vanuit die functie naar het hoofdprogramma zal deze variabele niet meer gekend zijn. Zo'n variabele die je nieuw aanmaakt binnen een functie, om te gebruiken binnen die functie is een lokale variabele. De scope van zo'n variabele is de functie.

Een variabele die in het hoofdprogramma gemaakt wordt is een globale variabele (global). De scope van een global variabele is gans het programma.

Als je een functie oproept dan zal die functie deze variabele herkennen.

H08_Voorbeeld_07_global_local

```
001 def groet_1() :
002     print(tekst)
003 def groet_2() :
004     tekst = " Hallo, hoe is het er mee ? "
005     print(tekst)
006 tekst = "Hallo, Hallo"
007 groet_1()
008 groet_2()
009 groet_1()
```

Geeft op het scherm:

```
Hallo, Hallo
Hallo, hoe is het er mee ?
Hallo, Hallo
```

Lijn	Uitleg
006	De variabele tekst is global.
007	Als je dus functie groet1 oproept zal die functie de waarde kennen van de variable tekst.
008	Roept de functie groet2 op. Die functie begint met tekst= . Hier is tekst een lokale variabele.
009	Roept functie groet1 terug op. De variabele tekst bevat nog steeds de oorspronkelijke waarde.

Als je toch zou willen dat in de functie groet2 de waarde van de globale variabele tekst verandert, dan kan dat door in de functie de instructie

```
global tekst
```

te gebruiken.

Na dit hoofdstuk kan je:

- uitleggen wat een functie is
- een functie maken
- een functie oproepen in een programma
- uitleggen wat local en global variabelen zijn
- zelf een functie maken
 - met of zonder parameters
 - met en zonder default waarden
 - die iets teruggeeft aan de oproeper

9 RECURSIE

Inleiding

Recursie is een speciale techniek waarbij je binnen een functie dezelfde functie terug oproept, terwijl die functie nog aan het uitvoeren is. De functie roept zichzelf op. Om niet in een oneindige reeks van oproepen te geraken moet de functie zo gebouwd zijn dat ze op een bepaald ogenblik zichzelf niet meer hoeft op te roepen.

Klassiek voorbeeld : faculteit bereken

6! : 6.5.4.3.2.1

Je benoemt dit als : 6 faculteit : je start bij 6 en vermenigvuldigt met alle gehele getallen kleiner dan 6.

Als je hier even over nadenkt : $6! = 6 \cdot 5!$

Dus als je de faculteit van een getal wil berekenen neem je dat getal en je vermenigvuldigt dat met de faculteit van het getal min één.

Dat kan je vertalen naar een recursieve python functie :

H09_Voorbeeld_01_faculteit

```
001 def faculteit( n ):
002     if n <= 1:
003         return 1
004     else:
005         return n * faculteit( n-1 )
006
007 getal = int(input("Geef een getal en duw <enter> : "))
008 print(getal, "! = ",faculteit(getal))
```

Kan op het scherm geven:

```
Geef een getal en duw <enter> : 20
20 ! =  2432902008176640000
```

Lijn	Uitleg
005	Hier zie je dat n vermenigvuldigd wordt met de faculteit van n-1. De functie roept zichzelf op. Dat is recursiviteit.
002	Stop als n gelijk wordt aan 1 . $1! = 1$. Dit is bij recursie een voorwaarde. Je moet er voor zorgen dat de functie zichzelf niet blijft oproepen. Ergens moet er een voorwaarde zijn waarmee hieraan een einde komt.

Als je dit voorbeeld doorloopt met debug zie je duidelijk wat er gebeurt. Je ziet ook duidelijk hoe n verandert bij het oproepen en vooral dat als de functie stopt hoe ze dan terugkeert naar zichzelf.

Klassiek voorbeeld : de rij van Fibonacci

1-2-3-5-8-13-21-34-55-89

Dit zijn de 10 eerste getallen van de rij van Fibonacci. Je begint met 1 en 2 en dan is ieder volgend getal de som van de 2 vorige getallen.

H09_Voorbeeld_02_fibo

```

001 def fibo( n ):
002     if n == 1:
003         return 1
004     if n == 2:
005         return 2
006     else:
007         return fibo(n-2) + fibo(n-1)
008
009 getal = int(input("Geef een getal en duw <enter> : "))
010 for teller in range(1,getal) :
011     print(fibo(teller),end="-")
012 print(fibo(getal))

```

Lijn	Uitleg
007	Hier zie je in de Python code dat het n-de Fibonacci getal gelijk is aan het n-1-de plus het n-2-de. In de instructie zie je duidelijk dat de functie zichzelf oproept.
002 004	Beide lijnen zorgen ervoor dat de recursie stopt.

Voorbeeld : mappen op je computer

Een map op je computer is een recursieve structuur. Binnen een map kun je opnieuw een map hebben, en daar onder weer mappen. Een map is een boomstructuur. Iedere boomstructuur kan je doorlopen met behulp van recursiviteit.

In de oefeningen van het hoofdstuk over de OS-module vindt je voorbeelden waarbij je een recursieve functie moet maken die recursief een mappenstructuur doorloopt.

Na dit hoofdstuk kan je:

- uitleggen wat recursie is
- een paar voorbeelden geven van recursieve functies

10 STRINGS

Inleiding

Even herhalen :

Een string : 0 of meerdere tekens tussen quotes. (afpraak : dubbele quotes)

Vele programmeerproblemen zijn problemen waar je met tekstuele gegevens moet werken. Daarom zijn er in de meeste programmeertalen vele mogelijkheden om met tekst te werken. In dit hoofdstuk ga je een aantal belangrijke bewerkingen zien.

Rekenen met strings

Je hebt al leren "rekenen" met strings.

Voorbeeld H10_Voorbeeld_01_Strings_reken

Lijn	Instructie	Op het scherm
001	<code>beleg = "choco"</code>	
002	<code>extra = "pasta"</code>	
003	<code>print(beleg + extra)</code>	chocopasta
004	<code>print(3 * beleg)</code>	chocochocochoco
005	<code>print(extra * 2)</code>	pastapasta

Strings doorlopen letter per letter

Met een for loop kunnen we een string letter voor letter doorlopen:

H10_Voorbeeld_02_String_doorlopen

001	<code>beleg = "pasta"</code>
002	<code>extra = "kaas"</code>
003	<code>for letter in beleg :</code>
004	<code> if letter in extra :</code>
005	<code> print(beleg, "en", extra, "bevatten beide een", letter)</code>

Geeft op het scherm:

```
pasta en kaas bevatten beide een a
pasta en kaas bevatten beide een s
pasta en kaas bevatten beide een a
```

Lijn	Uitleg
003	De for gaat in de variabele letter één voor één alle letters van de string zetten. Voor iedere waarde van die variabele teller zullen lijn 004 en 005 uitgevoerd worden.

String : de index

Ieder teken in een string heeft een "positie-nummer" : een index.

Die start bij 0 (de eerste karakter) en eindigt bij het laatste teken met de waarde die juist één lager is dan de lengte van de string.

Hieronder bijvoorbeeld de string 'choco'.

c	h	o	c	o
0	1	2	3	4
-5	-4	-3	-2	-1

Je ziet op de tweede lijn de verschillende indexen van de karakters die er juist boven staan.

Op het schema kan je ook zien, op de derde lijn dat je ook van achter (-1) naar voor (-5) kan gaan met de indexen. Dat is handig als je achteraan wil beginnen. Ook handig als je enkel maar geïnteresseerd bent in de laatste karakter.

Je kan de karakters van een string ophalen via de index: je neemt de naam van de variabele waar de string in zit en tussen vierkante haakjes zet je de index van de karakter die je wil.

Je kan ook meerdere karakters in één keer ophalen. Hiervoor gebruik je vierkant haakjes en soms een dubbel punt.

H10_Voorbeeld_03_String_index

Lijn	Instructie	Op het scherm
001	<code>zin = "Choco is lekker!"</code>	
002	<code>print(zin[3])</code>	c
003	<code>print(zin[2:4])</code>	oc
004	<code>print(zin[3:])</code>	co is lekker!
005	<code>print(zin[:4])</code>	Choc
006	<code>print(zin[1:-1])</code>	hoco is lekker
007	<code>print(zin[-3])</code>	e
008	<code>positie = 3</code>	
009	<code>print(zin[positie:positie*2])</code>	co

Lijn	Uitleg
002	Neemt de letter met index 3 (opgepast, de eerste letter heeft index 0).
003	De : wil zeggen tot, niet tot en met, index 4 mag er niet bij.
004	Als er achter : niets staat wil dat zeggen : tot het eind.
005	Als er voor : niet staat wil dat zeggen : vanaf het begin.
006	Van positie 1 tot de laatste positie
007	Van achter terugtellen tot je -3 tegenkomt
009	In plaats van getallen kan je ook variabelen gebruiken om de indexen aan te duiden.

Methodes/methods : iets doen met strings

Met een method kan je iets doen met een string.

Hoe gebruiken :

- je schrijft de stringnaam
- gevolgd door een punt
- gevolgd door de naam van de method
- gevolgd door haakje open en achteraan haakje dicht (wordt dikwijls vergeten)
- waarbij er soms tussen die haakjes nog een parameter kan staan

Een method verandert niets aan de string waarop je de method toepast.

Wil je het resultaat van de methode gebruiken in je programma, dan zal je dat resultaat van de method in een variabele moeten zetten

Een paar nuttige methods:

Naam	Parameters	Resultaat	Uitleg
strip	geen	string	verwijdert spaties aan het begin en einde van een string
upper/ lower	geen	string	maakt van alle letters hoofdletters/kleine letters
find	strings	integer	de index van het eerste voorkomen van de parameter in de string
replace	2 strings	string	vervang alle voorkomens van de de eerste parameter door de tweede parameter
split	string indien afwezig = blanco	list	splitst de string op in stukken gebaseerd op de parameter als scheiding
join	lijst	string	plakt de strings uit de parameter aan elkaar gescheiden door de string voor de punt.

H10_Voorbeeld_04_Strings_methods

Lijn	Instructie	Op het scherm
001	<code>woord = "is"</code>	
002	<code>zin_1 = "Choco is lekker!"; print(zin_1)</code>	Choco is lekker!
003	<code>zin_2 = " Banaan is ook ok! " ;</code>	
004	<code>print(zin_2)</code>	Banaan is ook ok!
005	<code>zin_3 = zin_2.strip() ; print(zin_3)</code>	Banaan is ook ok!
006	<code>zin_3 = zin_3.upper() ; print(zin_3)</code>	BANAAN IS OOK OK!
007	<code>zin_3 = zin_3.lower() ; print(zin_3)</code>	banaan is ook ok!
008	<code>positie = zin_1.find(woord); print(positie)</code>	6
009	<code>zin_3 = zin_1.replace(woord,"was") ;</code>	
010	<code>print(zin_1)</code>	Choco is lekker!
011	<code>lijst = zin_1.split(" ") ; print(lijst)</code>	['Choco', 'is', 'lekker!']
012	<code>zin_3 = "-".join(lijst) ; print(zin_3)</code>	Choco-is-lekker!
013	<code>zin_3 = "".join(lijst) ; print(zin_3)</code>	Chocoislekker!

Speciale karakter \ voor ",',\n

Stel dat je de volgende tekst wil printen :

```
Jan zei: "Fatima zei 'Hallo' tegen me".
```

Je kan proberen met:

```
print("Jan zei: "Fatima zei 'Hallo' tegen me.")
```

Maar dat gaat niet lukken: de tweede " zal Python in war brengen.

Om een " of een ' te printen zet je er een \ voor.

Een werkende instructie is dus :

```
print("Jan zei: \"Fatima zei 'Hallo' tegen me.\")
```

Hoe een \ printen? Met \\.

Ook als je een nieuwe lijn wil nemen ergens in een string kan je \n (new line) gebruiken.

H10_Voorbeeld_05_speciale_karakters

```
001 print("Jan zei:\n \"Fatima zei 'Hallo' tegen me.\")
002 zin = "Choco is lekker! \n maar \" Banaan \n  \\ \\ is ook ok! \"
003 print(zin)
```

Geeft op het scherm:

```
Jan zei:
"Fatima zei 'Hallo' tegen me."
Choco is lekker!
maar " Banaan
  \ \ is ook ok! "
```

Strings over meerdere regels

H10_Voorbeeld_06_lange_string

```
001 eerste_lange_tekst = "Lorem ipsum dolor sit amet,\
002   consectetur adipiscing elit \
003   sed do eiusmod tempor incididunt \
004     ut labore et dolore magna aliqua."
005 print(eerste_lange_tekst)
006 print()
007 tweede_lange_tekst = """Lorem ipsum dolor sit amet,
008   consectetur adipiscing elit,
009   sed do eiusmod tempor incididunt
010   ut labore et dolore magna aliqua."""
011 print(tweede_lange_tekst)
```

Geeft op het scherm:

```

Lorem ipsum dolor sit amet,consectetur adipiscing elit sed do
eiusmod tempor incididunt      ut labore et dolore magna aliqua.

```

```

Lorem ipsum dolor sit amet,
    consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua.Een duidelijk verschil!

```

Lijn	Uitleg
004	Het \ symbool achteraan een lijn geeft aan dat Python precies moet doen alsof er geen nieuwe regel staat. Je kan \ ook gebruiken als je instructies te lang worden of om instructies wat overzichtelijker te maken.
010	Als de string start met """" dan kijk Python waar de tekst stopt en voegt hij achter die tekst een \n (new line) toe en daarachter de volgende regel toe en dat blijft hij doen tot hij een volgende keer """" tegenkomt.

Het \ symbool achteraan een lijn geeft aan dat Python precies moet doen alsof er geen nieuwe regel staat. Je kan \ ook gebruiken als je instructies te lang worden of om instructies wat overzichtelijker te maken. Hierna een voorbeeld waarbij de conditie van de if te lang wordt.

In de volgende if is een \ symbool gebruikt om er voor te zorgen vermijden dat de volledige conditie leesbaar bleef op het scherm

```

if (len(zin_1) < 18) and (len(zin_2) < 20) \
    or (woord >= ingegeven_ww) :
is identiek aan
if (len(zin_1) < 18) and (len(zin_2) < 20) or (woord >= ingegeven_ww) :

```

het \-symbool is de 'continuation character' ; de meeste programmeertalen hebben zo een teken.

Na dit hoofdstuk kan je:

- uitleggen wat een string is
- strings gebruiken in een programma
- delen van strings ophalen
- de werking uitleggen van volgende functies en deze functies gebruiken:
 - strip
 - upper
 - lower
 - find
 - replace
 - split
 - join
- een ", een ', een \ en een new line "printen"
- werken met lange strings
- uitleggen wat een continuation karakter en ze gepast gebruiken

11 LISTS

Inleiding

Tot nu heb je met variabelen gewerkt waar "één" ding in zat. Een getal, een string en logische waarde.

Hiermee geraak je al een heel eind, maar in de praktijk zal je dan snel de nood voelen om een samenhangende reeks van gegevens te gebruiken.

Je hebt manieren nodig om dingen in groepjes te plaatsen.

De getallen van de lotto, de namen van cursisten, de woorden in een tekst enzovoort.

En om dan zo'n groepje te doorlopen, stukken ervan te pakken om er iets mee te doen.

Lijsten en nog andere gegevensstructuren die dit toelaten, zijn fundamentele bouwstenen om te programmeren. Er zijn weinig praktische problemen die je zonder hulp van lijsten (of aanverwanten) op een goede manier kan oplossen.

Vele andere programmeertalen gebruiken "Arrays" in plaats van lists. Maar er zijn duidelijke verschillen tussen lists en arrays.

Basisbegrippen en stukken van lijsten

Een 'list' is een geordende verzameling van dingen.

Die dingen zijn de elementen van de lijst.

De volgorde speelt een rol.

H11_Voorbeeld_01_list_index

```
001  lotto = [3, 14, 25, 27, 33, 47]
002  soorten_beleg = ["choco", "boter", "gelei", "salami"]
003  soorten_beleg_x = ["choco", "boter", "gelei", \
004      ["hagelslag", "banaan"], "salami"]
005  raar = ["choco", 666, True]
006  print(soorten_beleg[2])
007  print(raar[2])
008  print(soorten_beleg_x[3])
009  print(soorten_beleg_x[3][1])
010  print(soorten_beleg[1:3])
011  print(lotto[2:5])
012  print(lotto[0:6:2])
```

Geef op het scherm:

```
gelei
True
['hagelslag', 'banaan']
banaan
['boter', 'gelei']
[25, 27, 33]
[3, 25, 33]
```

Lijn	Uitleg
001	lotto is een lijst. Een list start dus met een vierkant haakje en eindigt met een vierkant haakje. Daartussen staan de elementen van de list. De elementen zijn gescheiden van elkaar door een komma. De elementen kunnen eender welk gegeven zijn. Ze kunnen dus ook van eender welk type zijn. lotto is een list met 6 elementen, 6 integers
002	soorten_beleg is een list met 4 strings
003	soorten_beleg_x is een list met 5 elementen. Een element van een list kan ook weer een list zijn. Een list met lists is een geneste list.
005	Een lijst kan ook elementen van verschillende data types hebben.
006	Zoals de letters van een string hebben de elementen van een list ook een index. Die start bij 0 (het eerste element) en eindigt bij het laatste element met de waarde die juist één lager is dan de lengte van de list. Met behulp van de index kan je een element ophalen uit een list. Je neem de naam van de lijst en zet tussen vierkante haakjes de index. In dit voorbeeld : haal uit de list soorten_beleg het element met index 2. (het derde element dus)
008	Het element met index 3 van de list soorten_beleg_x is zelf een list.
009	Neem het element met index 1 van het element met index 3 van de list soorten_beleg_x
010	Je kan ook meerdere opeenvolgende elementen uit een lijst halen : slicing. Je moet dan opgeven bij welke index je wil starten en tot waar je wil gaan. Je kan opnieuw een : gebruiken tussen de boven- en ondergrens. Ook hier weer: de : wil zeggen tot en niet tot en met. Het resultaat is opnieuw een list.
012	Er staat nu een 3 ^{de} getal tussen de []. Zoals mij range is dit derde getal ook weer de stapgrootte. In dit voorbeeld : je start bij index 0, dan naar index 2, dan naar 4 en dan stopt het want het is opnieuw tot en niet tot en met.

Onthou:

[2] : het derde element

[2:4] : het derde en vierde element

[2:5] : vanaf index 2 tot index 5 (niet tot en met)

[0:6:2] : vanaf ... tot .. in stappen van ...

Als bij volgende voorbeelden de 5 eerste lijnen ontbreken, dan zijn ze :

001	lotto = [3, 14, 25, 27, 33, 47]
002	soorten_beleg = ["choco", "boter", "gelei", "salami"]
003	soorten_beleg_x = ["choco", "boter", "gelei", \
004	["hagelslag", "banaan"], "salami"]
005	raar = ["choco", 666, True]

Nuttige functies voor lists

Naam	Parameters	Resultaat	Uitleg
len	list	integer	de lengte van de parameter
max min	list	een element van de list	het grootste/kleinste element van de parameter
sum	list		de som van de elementen van de parameter

H11_Voorbeeld_02_list_functies

Lijn	Instructie	Op het scherm
006	<code>print(len(lotto))</code>	6
007	<code>print(len(soorten_beleg_x))</code>	5
008	<code>print(max(lotto))</code>	47
009	<code>print(min(soorten_beleg))</code>	boter
010	<code>print(sum(lotto))</code>	149

Controle op aanwezigheid in de list

Om te controleren of iets aanwezig is in de lijst, gebruik je, net zoals bij strings in.

H11_Voorbeeld_03_list_in

Lijn	Instructie	Op het scherm
006	<code>print("o" in "choco")</code>	True
007	<code>print("py" in "Python")</code>	False
008	<code>print(3 in lotto)</code>	True
009	<code>print(4 in lotto)</code>	False
010	<code>print("gelei" in soorten_beleg)</code>	True
011	<code>print("banaan" in soorten_beleg_x)</code>	False
012	<code>print("banaan" in soorten_beleg_x[3])</code>	True

Lijn	Uitleg
006	In kan je ook gebruiken in strings.
007	De twee opeenvolgende letters moeten voorkomen in de dezelfde volgorde.
008	3 is een element in de lijst lotto.
009	4 is geen element in de lijst lotto.
011	banaan is wel zichtbaar in de lijst, maar niet als element.
012	banaan is wel een element van de lijst binnen de lijst.

Doorlopen van een list

Met een for kan je een lijst van voor naar achter doorlopen:

H11_Voorbeeld_05_list_doorlopen

```

006 for soort_beleg in soorten_beleg:
007     print(soort_beleg, end= " - ")
008 print()
009 for teller in range(len(lotto)):
010     print(str(teller+1) + ")"+ \
011           str(lotto[teller]), end=" ")
012 print()

```

Geeft op het scherm:

```

choco - boter - gelei - salami -
1)3   2)14   3)25   4)27   5)33   6)47

```

Lijn	Uitleg
006	soort_beleg doorloopt alle elementen van de lijst soorten_beleg (van index 0 tot op het einde van de lijst met index 3).
007	drukt soort_beleg af en gaat niet naar een volgende regel; zet een streepje.
008	Brengt de cursor naar een nieuwe regel.
009	Laat teller lopen van 0 tot en met 5
010	Drukt de teller af met een haakje achter en daarna het element van de lotto list met als index teller. Dit is een klassieke manier om mooi door een lijst te lopen en mooi een nummer er bij te zetten. Blijft ook op dezelfde regel verder printen.
012	Brengt de cursor naar een nieuwe regel.

Een list wijzigen via de index of indexen

Via de indexen kan jij elementen van een lijst wijzigen, toevoegen, verwijderen.

H11_Voorbeeld_06_list_wijzig_met_index

```

001 soorten_beleg = ["choco", "boter", "gelei", "salami"]
002 soorten_beleg[1] = "pindakaas"
003 print(soorten_beleg)
004 soorten_beleg = ["choco", "boter", "gelei", "salami"]
005 soorten_beleg[1] = ["pindakaas", "biscoff"]
006 print(soorten_beleg)
007 soorten_beleg = ["choco", "boter", "gelei", "salami"]
008 soorten_beleg[1:1] = ["pindakaas", "biscoff"]
009 print(soorten_beleg)
010 soorten_beleg = ["choco", "boter", "gelei", "salami"]
011 soorten_beleg[1:4] = ["pindakaas", "biscoff"]
012 print(soorten_beleg)
013 soorten_beleg = ["choco", "boter", "gelei", "salami"]
014 soorten_beleg[1:3] = []
015 print(soorten_beleg)

```

Geeft op het scherm:

```
['choco', 'pindakaas', 'gelei', 'salami']
['choco', ['pindakaas', 'biscoff'], 'gelei', 'salami']
['choco', 'pindakaas', 'biscoff', 'boter', 'gelei', 'salami']
['choco', 'pindakaas', 'biscoff']
['choco', 'boter', 'gelei', 'pindakaas', 'biscoff', 'salami']
['choco', 'salami']
```

Lijn	Uitleg
002	Vervangt in de list soorten_beleg het element met index 1 door het rechterlid.
005	Vervangt het element met index 1 door het rechterlid. En het rechterlid is hier een lijst.
008	Door expliciet een range te geven 1:1 geef je aan dat het de elementen zijn die op plaats 1 moeten komen en niet de lijst. 1:1 schuift dingen tussen de elementen van de lijst
011	Vervangt de elementen met index 1 tot index 4 door de elementen uit de lijst in het rechterlid
014	Vervangt de elementen met index 1 tot index 3 door de elementen uit de lijst uit het rechterlid die in dit voorbeeld leeg is. De elementen worden dus verwijderd.

List operatoren

Je kan lijsten optellen en vermenigvuldigen met een integer.

H11_Voorbeeld_07_list_operatoren

```
001 soorten_beleg = ["choco", "boter", "gelei", "salami"]
002 print(soorten_beleg)
003 soorten_beleg += ["pindakaas", "biscoff"]
004 print(soorten_beleg)
005 soorten_beleg = ["choco", "boter"] + ["gelei", "salami"]
006 print(soorten_beleg)
007 soorten_beleg = 3 * ["choco", "boter"]
008 print(soorten_beleg)
```

Geeft op het scherm:

```
['choco', 'boter', 'gelei', 'salami']
['choco', 'boter', 'gelei', 'salami', 'pindakaas', 'biscoff']
['choco', 'boter', 'gelei', 'salami']
['choco', 'boter', 'choco', 'boter', 'choco', 'boter']
```


List methodes

Een list method doet meestal iets met een lijst.

Je moet dus oppassen : de oorspronkelijke lijst ben je dan kwijt.

Hoe gebruiken : je schrijft de listnaam gevolgd door een punt gevolgd door de naam van de method gevolgd door haakje open en haakje dicht waarbij er soms tussen die haakjes nog een parameter kan staan.

De voorbeelden maken het duidelijk.

Een paar nuttige methods

Naam	Parameters	Resultaat	Uitleg
append	iets		voegt iets toe achter aan de list
extend	list		voegt de elementen van de parameter toe achter aan de list
insert	integer en iets		voegt iets toe op index aangegeven door de integer parameter en schuift alle elementen daar achter op
remove	iets		verwijdert uit de list het element dat gelijk is aan de parameter
pop	niets		verwijdert het laatste element uit de lijst
pop	integer		verwijdert het element met index gelijk aan de parameter uit de lijst
del			gevolgd door een stuk van een list aangegeven door de indexen (één of meerdere elementen) : verwijdert de aangewezen elementen uit de list.
index	iets	integer	geeft de index waar het iets in de list staat
count	iets	integer	geeft het aantal keer dat het iets voorkomt in de lijst
sort			sorteert de lijst (er kunnen nog parameters bij opgegeven worden om de manier van sorteren te beïnvloeden
reverse			keert de volgorde van de elementen van de list om

H11_Voorbeeld_08_methods

```

001 soorten_beleg = ["choco", "boter", "gelei", "salami"]
002 soorten_beleg.append("biscoff"); print("Na append: ", soorten_beleg)
003 soorten_beleg = ["choco", "boter", "gelei", "salami"]
004 soorten_beleg.extend(["pindakaas", "kop"]);
005 print("Na extend: ", soorten_beleg)
006 soorten_beleg = ["choco", "boter", "gelei", "salami"]
007 soorten_beleg.insert(2, "pindakaas");print("Na insert: ", soorten_beleg)
008 soorten_beleg = ["choco", "boter", "gelei", "salami"]
009 soorten_beleg.remove("boter");print("Na remove: ", soorten_beleg)
010 soorten_beleg = ["choco", "boter", "gelei", "salami"]
011 soorten_beleg.pop();print("Na pop() :",soorten_beleg)

```

```

012 soorten_beleg.pop(1);print("Na pop: ", soorten_beleg)
013 soorten_beleg = ["choco", "boter", "gelei", "salami"]
014 del soorten_beleg[2];print("Na del : ", soorten_beleg)
015 del soorten_beleg[1:2];print("Na del: ", soorten_beleg)
016 soorten_beleg = ["choco", "boter", "gelei", "salami"]
017 print(soorten_beleg);print(soorten_beleg.index("gelei"))
018 soorten_beleg = ["choco", "boter", "gelei", "salami"]
019 soorten_beleg.extend(["pindakaas", "kop", "choco"]);
020 print("Na extend: ", soorten_beleg)
021 print(soorten_beleg.count("choco"))
022 soorten_beleg = ["choco", "boter", "gelei", "salami"]
023 soorten_beleg.sort();print("Na sort: ", soorten_beleg)
024 soorten_beleg = ["choco", "boter", "gelei", "salami"]
025 soorten_beleg.reverse();print("Na reverse: ", soorten_beleg)

```

Geeft op het scherm:

```

Na append: ['choco', 'boter', 'gelei', 'salami', 'biscoff']
Na extend: ['choco', 'boter', 'gelei', 'salami', 'pindakaas', 'kop']
Na insert: ['choco', 'boter', 'pindakaas', 'gelei', 'salami']
Na remove: ['choco', 'gelei', 'salami']
Na pop() : ['choco', 'boter', 'gelei']
Na pop: ['choco', 'gelei']
Na del : ['choco', 'boter', 'salami']
Na del: ['choco', 'salami']
['choco', 'boter', 'gelei', 'salami']
2
Na extend: ['choco', 'boter', 'gelei', 'salami', 'pindakaas', 'kop', 'choco']
2
Na sort: ['boter', 'choco', 'gelei', 'salami']
Na reverse: ['salami', 'gelei', 'boter', 'choco']

```

Copy van een list

Je moet weten dat de list methodes de lijst waar je mee werkt veranderen. Waardoor je de oorspronkelijke lijst kwijt bent.

Daarom maak je dikwijls een copy van die lijst, zodat je achteraf nog weet met welke lijst je vertrokken bent.

Een list kopiëren is echter niet zo eenvoudig. Je hebt er zelfs een aparte module voor nodig.

H11_Voorbeeld_09_list_copy

```

001 soorten_beleg = ["choco", "boter", "gelei", "salami"]
002 soorten_beleg_nieuw = soorten_beleg
003 print(soorten_beleg)
004 print(soorten_beleg_nieuw)
005 soorten_beleg_nieuw[2] = "biscoff"
006 print(soorten_beleg)
007 print(soorten_beleg_nieuw)

```

```

008 import copy
009 soorten_beleg = ["choco", "boter", "gelei", "salami"]
010 soorten_beleg_nieuw = copy.deepcopy(soorten_beleg)
011 print(soorten_beleg)
012 print(soorten_beleg_nieuw)
013 soorten_beleg_nieuw[2] = "biscoff"
014 print(soorten_beleg)
015 print(soorten_beleg_nieuw)

```

Geeft op het scherm:

```

['choco', 'boter', 'gelei', 'salami']
['choco', 'boter', 'gelei', 'salami']
['choco', 'boter', 'biscoff', 'salami']
['choco', 'boter', 'biscoff', 'salami']
['choco', 'boter', 'gelei', 'salami']
['choco', 'boter', 'gelei', 'salami']
['choco', 'boter', 'gelei', 'salami']
['choco', 'boter', 'biscoff', 'salami']

```

Lijn	Uitleg
002	Maakt een nieuwe variabele en stelt die gelijk aan een reeds bestaande. Je zou kunnen denken dat dan 2 verschillende plaatsen in het computergeheugen zijn.
003 004	Dit lijkt zo te zijn.
005	Verandert iets aan de nieuwe list
006 007	In het resultaat van de print zie je dat de reeds bestaande lijst ook veranderd is. De oorzaak hier van is dat de instructie op lijn 002 geen copy doet, maar het adres van een nieuwe variabele gelijkstelt aan het adres van een vroeger aangemaakte variabele : enkel de geheugenadressen worden gelijkgesteld, niet de inhoud van het geheugen.
010	Maakt een echte copy, een deepcopy van de lijst. Hiervoor heb je op lijn 008 een import moeten doen van de module copy.
014 015	In het resultaat van de print zie je duidelijk dat beide lijsten nu verschillend zijn.

Geneste lists : oxo

Bestudeer aandacht volgend voorbeeld: het bevat een heleboel zaken die je geleerd hebt: functies, geneste lijsten, printen Het kan de start zijn van je eigen versie van het spel.

H11_Voorbeeld_10_list_nest

```

001  oxo_bord = [ ["-", "x", "-"], \
002               ["o", "o", "-"], \
003               ["x", "-", "o"] ]
004  def print_oxo_bord(b) :
005      print("  1  2  3 ")
006      for rij in range(3) :
007          print( rij + 1, end= " ")
008          for kolom in range(3) :
009              print(b[rij][kolom], end=" ")
010          print()
011  print_oxo_bord(oxo_bord)
012  oxo_bord[1][2] = "x"
013  oxo_bord[2][1] = "o"
014  print_oxo_bord(oxo_bord)

```

Na dit hoofdstuk kan je:

- uitleggen wat een list is
- een list doorlopen
- stukken uit lists halen
- bewerkingen uitvoeren met lists
- lists gebruiken in een programma
- de werking uitleggen van volgende functies en deze functies gebruiken:
 - append
 - extend
 - insert
 - remove
 - pop
 - index
 - count
 - sort
 - reverse
- del gebruiken met lists
- uitleggen wat een geneste list is
- geneste lists gebruiken

12 TUPLES

Inleiding

Een tuple is ook een verzameling van dingen die je gegroepeerd kan behandelen. Tuples lijken op lijsten maar je kan de elementen van een tuple niet veranderen zoals je dat met lijsten kan doen.

Tuples worden niet zo veel gebruikt.

Basisbegrippen

Een tuple is een geordende verzameling van dingen.

Je zet die dingen tussen **ronde** haakjes, gescheiden door komma's.

Die dingen zijn de **elementen** van de lijst.

De volgorde speelt een rol.

H12_Voorbeeld_01_tuple_voorbeelden

```

001 lotto = (3, 14, 25, 27, 33, 47)
002 soorten_beleg = ("choco", "boter", "gelei", "salami")
003 soorten_beleg_x = ("choco", "boter", "gelei", \
004     ("hagelslag", "banaan"), "salami")
005 raar = ("choco", 666, True)
006 print(soorten_beleg[2])
007 print(raar[2])
008 print(soorten_beleg_x[3])
009 print(soorten_beleg_x[3][1])
010 print(soorten_beleg[1:3])
011 print(lotto[2:5])
012 print(lotto[0:6:2])
013 teller = 0
014 while teller < len(soorten_beleg):
015     print(soorten_beleg[teller],end=",")
016     teller +=1
017 print()
018 print( ("choco", "banaan") == ("choco", "banaan"))
019 print( ("choco", "banaan") == ("banaan", "choco"))
020 print( ("choco", "gelei") >= ("choco", "banaan"))

```

Geeft op het scherm:

```

gelei
True
('hagelslag', 'banaan')
banaan
('boter', 'gelei')
(25, 27, 33)
(3, 25, 33)

```

```
choco,boter,gelei,salami,
True
False
True
```

Lijn	Uitleg
001-005	Maakt tuples aan. Tussen ronde haakjes
	Alle andere instructies zijn vergelijkbaar met de instructies die je tegengekomen bent bij lists.

Tuples bruikbaar in functies.

Soms kan het nuttig zijn dat een functie in plaats van één waarde, meerdere waarden teruggeeft.

H12_Voorbeeld_02_tuple_functie

```
001 def bereken_som_en_gemiddelde(lijt):
002     som = sum(lijt)
003     gemiddelde = som / len(lijt)
004     return som, gemiddelde
005 getallen = (3, 14, 25, 27, 33, 47)
006 som_en_gemiddelde = bereken_som_en_gemiddelde(getallen)
007 print(som_en_gemiddelde)
```

Geeft op het scherm:

```
(149, 24.833333333333332)
```

Lijn	Uitleg
001	Definieer een functie die de som en het gemiddelde van de elementen van een lijst berekent. Parameter is een lijst.
004	De functie geeft 2 dingen terug. Dit doe je met een return van de 2 dingen gescheiden door een komma.
006	Roept de functie op en zet het resultaat in de variabele som_en_gemiddelde.
007	Print de variabele. Hier zie je duidelijk op het scherm dat de variabele een tuple is: ronde haakjes en elementen gescheiden door een komma.

Dus : als een functie meerdere waarden teruggeeft, geeft ze die waarden terug in een tuple.

Na dit hoofdstuk kan je:

- uitleggen wat een tuple is en waar het soms nuttig voor is.
- bewerkingen uitvoeren met tuples
- tuples gebruiken om, in de definitie van een functie, meerdere variabelen terug te geven
- begrijpen dat je dikwijls in plaats van tuples ook lists kan gebruiken

13 DICTIONARIES

Inleiding

Een lijst is een krachtige datastructuur om gegevens in op te slaan, maar soms schieten ze nog te kort. Stel dat je bijvoorbeeld informatie wil bijhouden over cursisten. Een cursist heeft een naam en een score.

Met je huidige kennis van datastructuren maak je dan 2 lijsten:

- een lijst met namen
- een lijst met op de overeenkomstige indexpositie de score van de cursist.

Wil je nu van een bepaalde cursist de score weten, dan moet je zijn naam gaan opzoeken in de lijst van namen om de indexpositie te kennen van die naam. Met die indexpositie ga je dan de score ophalen in de lijst van namen.

H13_Voorbeeld_01_cursist_score_met_list

```
001 cursisten = ["Julie", "Mila", "Lucas", "Sem"]
002 scores = [10, 7, 16, 19]
003 cursist = input("Geef de naam van de cursist >>> ")
004 cursist_index = cursisten.index(cursist)
005 cursist_score = scores[cursist_index]
006 print("De score van",cursist,"is",cursist_score)
```

Geeft op het scherm:

```
Geef de naam van de cursist >>> Mila
De score van Mila is 7
```

Dit werkt, maar de koppeling tussen de cursist en zijn score is heel los. Als er iets zou gebeuren met de volgorde in één van de lijsten klopt de informatie niet meer.

Python kent een datastructuur waarbij de gegevens wel echt gekoppeld blijven :

de **dictionary**. Het woord zegt het zelf : een woordenboek. Je kan er in gaan opzoeken.

Wat is een dictionary?

In het voorbeeld uit de inleiding zou een bruikbare dictionary zijn:

```
scores = {"Julie":10, "Mila" : 7, "Lucas" : 16, "Sem" : 19}
```

Een dictionary is een collectie, tussen **accollades**, niet geordend, wijzigbaar, bestaande uit paren, paren gescheiden door een komma.

Een paar bestaat uit :

- een sleutel : een **key**
- een dubbelpunt gevolgd door
- een waarde : een **value**

Een dictionary is dus een collectie key-value-combinaties.

Volgend programma doet hetzelfde als het voorbeeld uit de inleiding :

H13_Voorbeeld_02_cursist_score_met_dictionary

```
001 scores = {"Julie":10, "Mila" : 7, "Lucas" : 16, "Sem" : 19}
002 cursist = input("Geef de naam van de cursist >>> ")
003 cursist_score = scores[cursist]
004 print("De score van",cursist,"is",cursist_score)
```

Kan op het scherm geven:

```
Geef de naam van de cursist >>> Lucas
De score van Lucas is 16
```

Lijn	Uitleg
001	Maak een dictionary. Je ziet de accolades, je ziet de verschillende koppels, ieder koppel met een naam, gevolgd door een dubbelpunt, gevolgd door de score. Alle gegevens die in het eerste voorbeeld in 2 lijsten weredn opgeslagen staan nu netjes in 1 datastructuur.
003	scores[cursist]: je zoekt in de dictionary naar het koppel met key = de waarde van cursist en je vindt de corresponderende value terug. Je zet de key om op te zoekn steeds tussen de vierkante haakjes.

Als je gegevens hebt die gekoppeld zijn aan een "key", waarmee je informatie wil gaan **opzoeken**, dan zijn dictionaries meestal het beste datatype.

Werken met een dictionary:

H13_Voorbeeld_03_gebruik_dict_scores

```
001 cursisten = ["Julie", "Mila", "Lucas"]
002 scores_input = [10, 7, 16]
003 scores = {}
004 scores["Daan"] = 8;print(scores)
005 for teller in range(len(cursisten)):
006     scores[cursisten[teller]]=scores_input[teller]
007 print(scores)
008 print("Score Mila;",scores["Mila"])
009 test_naam = "Jan"
010 test = scores.get(test_naam, test_naam + " niet gevonden!");
011 print(test)
012 scores["Julie"]= 12; print(scores)
013 scores.pop("Mila") ; print("Na pop:",scores)
014 test = scores.pop(test_naam,test_naam + " niet gevonden!");
015 print(test)
```


Geeft op het scherm:

```
{'Daan': 8}
{'Daan': 8, 'Julie': 10, 'Mila': 7, 'Lucas': 16}
Score Mila; 7
Jan niet gevonden!
{'Daan': 8, 'Julie': 12, 'Mila': 7, 'Lucas': 16}
Na pop: {'Daan': 8, 'Julie': 12, 'Lucas': 16}
Jan niet gevonden!
```

Lijn	Uitleg
001	Start met 2 listen waarmee we de dictionary gaan vullen. De eerste lijst met de namen
002	van de cursisten. De tweede met de corresponderende scores.
003	Maak een lege dictionary.
004	Voeg een eerste koppel toe met key="Daan" en value=8
005	Start een for loop die de elementen van de beide lijsten in de dictionary zet.
008	Zoekt de score van Mila.
010	method get : zoekt met een key en als die wordt gevonden geeft de method de value voor die key terug. Als de key er niet is, wordt de string teruggegeven die als tweede parameter opgegeven is.
012	Doe een update : overschrijf de huidige value voor de key "Julie" met de value 12.
013	method pop : verwijdert het koppel met key "Mila".
014	method pop met extra parameter : als de key niet bestaat, wordt de string die als 2 ^{de} parameter opgegeven is teruggegeven.

H13_Voorbeeld_04_gebruik_dict_scores

```
001 import operator
002 scores = {'Daan': 8, 'Julie': 12, 'Mila': 7, 'Lucas': 16}
003 scores.popitem(); print("Na popitem:", scores)
004 print(scores.keys())
005 print(scores.values())
006 gemiddelde = round(sum(scores.values())/len(scores), 2)
007 print("Gemiddelde score =", gemiddelde)
008 for cursist in sorted(scores):
009     print(cursist, scores[cursist], end=" - ")
010 print(); print(scores.items())
011 print(sorted(scores.items()))
012 print(sorted(scores.items(), \
013             key=operator.itemgetter(1), reverse=True))
014 scores.clear() ; print(scores)
015 print("Einde programma.")
```

Geeft op het scherm:

```
Na popitem: {'Daan': 8, 'Julie': 12, 'Mila': 7}
dict_keys(['Daan', 'Julie', 'Mila'])
dict_values([8, 12, 7])
Gemiddelde score = 9.0
Daan 8 - Julie 12 - Mila 7 -
dict_items([('Daan', 8), ('Julie', 12), ('Mila', 7)])
[('Daan', 8), ('Julie', 12), ('Mila', 7)]
[('Julie', 12), ('Daan', 8), ('Mila', 7)]
{}
Einde programma.
```

Lijn	Uitleg
001	Importeert module operator om later de itemgetter functie te kunnen gebruiken in lijn 013
002	Maakt een dictionary.
003	Method popitem : verwijdert het laatste toegevoegde key-value-paar.
004	Zet enkel de keys op het scherm
005	Zet enkel de values op het scherm
006	Gebruikt functies sum en len om het gemiddelde van de values te berekenen.
008	Doorloop alle elementen van de dictionary in volgorde van de keys.
010	Print de inhoud van de dictionary.
015	Print <code>sorted(scores.items())</code> : dit is de lijst bestaande uit tuples die overeenkomen met de key-value combinaties, gesorteerd op de key
016	Gebruik de <code>itemgetter(1)</code> om in de key parameter van de <code>sorted</code> functie op te geven dat de lijst moet gesorteerd worden op het item in de kolom met index 1. De parameter <code>reverse</code> geef nog aan of van klein naar groot of omgekeerd wordt gesorteerd.
017	Method <code>clear</code> : maakt de dictionary volledig leeg.

Na dit hoofdstuk kan je:

- uitleggen wat een dictionary is
- uitleggen wanneer het handig is om een dictionary te gebruiken
- bewerkingen uitvoeren met dictionaries
- ...

14SETS

Inleiding

Set = (wiskundige) verzameling.

Set is een datastructuur die vooral voor wiskundige toepassingen interessant kan zijn. Het grote verschil met bijvoorbeeld lists is dat een set niet geordend is en dat een set geen "dubbels" kan bevatten.

Hierdoor wordt het mogelijk om alle wiskundige set-operaties op deze data uit te voeren.

Basisbegrippen

Een set is een ongeordende verzameling van verschillende dingen.

Die dingen zijn de elementen van de set.

Je vormt de set door accolades rond de dingen te zetten.

De volgorde speelt geen rol.

Je kan de elementen van de set niet bereiken met een index. Je kan ze wel één voor één bereiken via een for loop.

Je kan de elementen van een lijst niet veranderen, je kan wel elementen toevoegen en verwijderen.

Een zeer belangrijk kenmerk van een set is dat er **geen dubbels** kunnen in voor komen. Python zorgt hier voor. Wat je ook doet met de set, ieder element zal slechts één keer voorkomen.

Een paar nuttige methods

Naam	Parameters	Resultaat	Uitleg
add	iets		voegt de parameter toe aan de set (ergens in de set, niet noodzakelijk achteraan, het speelt geen rol)
update	een set		voeg de elementen van de parameter toe aan de set
discard	iets		verwijdert de parameter uit de set
remove	iets		verwijdert de parameter uit de set en geeft een fout als het element niet aanwezig is in de set
copy		set	maakt een nieuwe set = de oorspronkelijke set
clear			verwijdert alle elementen van de set

H14_Voorbeeld_01_set_voorbeelden

```

001 soorten_beleg = {"choco", "boter", "gelei", "salami"}
002 for beleg in soorten_beleg :
003     print(beleg,end=" ")
004 print()
005 print("gelei" in soorten_beleg)
006 print("ham" in soorten_beleg)
007 soorten_beleg.add("ham")
008 print(soorten_beleg)
009 soorten_beleg.add("ham");print(soorten_beleg)
010 nieuw_beleg = {"biscoff", "ham", "kaas"}
011 soorten_beleg.update(nieuw_beleg)
012 print(soorten_beleg)
013 soorten_beleg.discard("biscoff")
014 print(soorten_beleg)
015 x = soorten_beleg.copy();print(x)
016 soorten_beleg.clear();print(soorten_beleg)

```

Geeft op het scherm:

```

salami choco gelei boter
True
False
{'gelei', 'boter', 'choco', 'salami', 'ham'}
{'gelei', 'boter', 'choco', 'salami', 'ham'}
{'gelei', 'boter', 'kaas', 'choco', 'salami', 'biscoff', 'ham'}
{'gelei', 'boter', 'kaas', 'choco', 'salami', 'ham'}
{'choco', 'gelei', 'ham', 'boter', 'salami', 'kaas'}
set()

```

Lijn	Uitleg
001	Maakt een set aan. Je herkent de accolades.
002	Doorloopt alle elementen van de set.
003	Print ieder element
005	Controleer de aanwezigheid van iets in de set.
006	
007	Voegt iets toe aan de set.
009	Voegt hetzelfde toe aan de set en je ziet na het printen dat er niets veranderd is aan de set. Een set kan geen gelijke elementen bevatten.
011	Voegt alle elementen van de set nieuw_beleg toe aan de set soorten_beleg.
013	Verwijdert een element uit de set.
015	Kopieert een set.
016	Verwijdert alle elementen uit de set. De set bestaat nog wel, maar is leeg.

Typische handelingen met verzamelingen

Python voorziet (als één van de weinige programmeertalen) methods die typisch zijn voor verzamelingen. Als je wiskunde minder leuk vindt dan kan je deze paragraaf overslaan.

Naam	Parameters	Resultaat	Uitleg
union	een set	een set	levert de set op die de unie is van beide sets
intersection	een set	een set	levert de set op die de doorsnede is van beide sets
difference	een set	een set	levert de set op die het verschil is van beide sets
is_disjoint	een set	een boolean	True : de doorsnede van de sets is leeg False : de doorsnede is niet leeg
is_subset	een set	een boolean	True : de set is een deelverzameling van de parameter False: geen deelverzameling
is_superset	een set	een boolean	True: de set bevat de parameter False : de set bevat de parameter niet

H14_Voorbeeld_02_set_verzamelingen

```

001 soorten_beleg_pa = {"choco", "boter", "gelei", "salami"}
002 soorten_beleg_ma = {"biscoff", "ham", "kaas", "gelei"}
003 soorten_beleg = soorten_beleg_pa.union(soorten_beleg_ma)
004 print(soorten_beleg)
005 soorten_gemeen = soorten_beleg_ma.intersection(soorten_beleg_pa)
006 print(soorten_gemeen)
007 soorten_alleen_pa = soorten_beleg_pa.difference(soorten_beleg_ma)
008 print(soorten_alleen_pa)
009 print(soorten_beleg_ma.isdisjoint(soorten_beleg_pa))
010 print(soorten_beleg.issuperset(soorten_alleen_pa))
011 print(soorten_alleen_pa.issubset(soorten_beleg_pa))

```

Geeft op het scherm:

```

{'ham', 'choco', 'boter', 'biscoff', 'kaas', 'gelei', 'salami'}
{'gelei'}
{'boter', 'salami', 'choco'}
False
True
True

```

Lijn	Uitleg
001	Maakt 2 sets aan.
002	
003	Maakt de union van de 2 sets. Merk op dat in het resultaat het element "gelei" slechts één keer voor komt.
005	Maakt de intersection en de difference.
007	
009	Test of sets een lege doorsnede hebben
010	Check of een set een deelverzameling is van een andere
011	

Na dit hoofdstuk kan je:

- uitleggen wat een set is
- bewerkingen uitvoeren met sets
- een aantal methods gebruiken
- ...

15 DE OS-MODULE : BESTURINGSSYSTEEM COMMANDO'S

Inleiding

In de volgende hoofdstukken kom je regelmatig bestanden tegen. Je komt dan op een terrein waar het besturingssysteem waarmee je werkt een rol begint de spelen. Bestanden, directories zijn hierin zeer belangrijke objecten en het is dan ook handig dat je er met Python handig mee kan omspringen.

De os module maakt dat mogelijk. (os = operating system)

De os-module geeft ons een gemakkelijke manier om te werken met het besturingssysteem.

Hier zijn ook risico's aan verbonden : omdat je met Python commando's gemakkelijk bestanden en directories kan manipuleren, kan er ook wel eens wat fout gaan. Dingen kunnen foutief verwijderd worden, foutief van naam veranderd worden, enz.... Dus zorg er steeds voor dat hetgeen je programma doet ook om een gemakkelijke manier kan "teruggedraaid" worden. Voorzie steeds een "undo".

In dit hoofdstuk beperken zie je een aantal basis-instructies zodat je net genoeg weet hoe je in de volgende hoofdstukken kan werken met bestanden en directories.

Bestanden en directories

H15_Voorbeeld_01_os_voorbeelden

```
001 import os
002 print("Current Working Directory:")
003 print(os.getcwd());print
004 lijst_dirs_en_files = os.listdir()
005 print("List met de inhoud van de Current Working Directory:")
006 print(lijst_dirs_en_files)
007 print("Maak een directory met naam data:")
008 os.mkdir("data")
009 print("Verander van directory:")
010 os.chdir("data")
011 print("Current Working Directory:")
012 print(os.getcwd());print
013 os.chdir("..")
014 print("List met de inhoud van de Current Working Directory:")
015 lijst_dirs_en_files = os.listdir()
016 print(lijst_dirs_en_files)
```

Geeft op het scherm: (afhankelijk waar je het programma laat lopen)

```

Current Working Directory:
C:\Users\Gebruiker\OneDrive - Thomas More\Documenten\00.
Lesgeven\Cevora Python\01. Voorbeelden\H15. OS module
List met de inhoud van de Current Working Directory:
['H00_Format without #.py', 'H15_Voorbeeld_01_os_voorbeelden.py',
'H15_Voorbeeld_01_os_voorbeelden_c.py']
Maak een directory met naam data:
Verander van directory:
Current Working Directory:
C:\Users\Gebruiker\OneDrive - Thomas More\Documenten\00.
Lesgeven\Cevora Python\01. Voorbeelden\H15. OS module\data
List met de inhoud van de Current Working Directory:
['data', 'H00_Format without #.py',
'H15_Voorbeeld_01_os_voorbeelden.py',
'H15_Voorbeeld_01_os_voorbeelden_c.py']

```

Lijn	Uitleg
001	Deze import is nodig om de os functies te kunnen gebruiken.
002	Print de naam van de huidige directory.
004	Maak een list met daarin de inhoud van de huidige directory : de cwd : de current working directory.
008	Maak een directory.
009	Ga naar de directory.
013	Ga naar de bovenliggende directory.
015	Haal opnieuw de inhoud op waarbij je zal zien dat de directory data er bij gekomen is

De os-sub-module path

De os-module heeft een submodule : path. Deze module bevat een aantal nuttige functies die te maken hebben met path-namen en met bestanden en directories. Als je die functies wil gebruiken moet je er dus os.path. voor zetten.

Naam	Parameters	Resultaat	Uitleg
isfile	iets	een boolean	True als het bestand een bestand is, anders False
isdir	iets	een boolean	True als het bestand een directory is, anders False
getsize	iets	integer	als het iets een bestand is krijg je de grootte in bytes, als het een directory is krijg je de grootte van het bestand dat de directory beschrijft. (dus niet de som van de grootte van alle bestanden in de directory)
exists	iets	boolean	True als iets (het bestand of directory) bestaat, anders False

H15_Voorbeeld_02_os_path_module

```

001 import os
002 huidige_dir = os.getcwd()
003 inhoud_dir = os.listdir(huidige_dir)
004 for bestand_of_dir in inhoud_dir :
005     if os.path.isfile(bestand_of_dir) :
006         print(bestand_of_dir, \
007             os.path.getsize(bestand_of_dir))
008     else:
009         print(bestand_of_dir, "is een directory")
010 if os.path.exists("testbestand.txt") :
011     print("Het bestand is er")
012 else :
013     print("Het bestand is er niet")

```

Geeft op het scherm iets in de aard van:

```

data is een directory
H16_Voorbeelden_Bestanden_V01.py 6590
H16_Voorbeeld_01_read.py 130
H16_Voorbeeld_01_read_c.py 308
H16_Voorbeeld_02_readline.py 221
H16_Voorbeeld_02_readline_c.py 470
...
H16_Voorbeeld_05_writelines.py 345
H16_Voorbeeld_05_writelines_c.py 733
H16_Voorbeeld_06_os_module.py 411
Het bestand is er niet

```

Lijn	Uitleg
002	Haalt de current working directory op.
003	Haalt alle items op in die directory en zet die in een list. (bestanden en directories)
004	Doorloopt ieder ding van de list
005	als het ding een bestand is :
006	Print de naam en de grootte in bytes.
010	Check of het bestand bestaat.

Alle bestanden doorlopen met os.walk

Na dit hoofdstuk kan je:

- weet je dat de `os`-module een handig hulpmiddel is om met bestanden en directories te werken
- zeggen welke nuttige informatie de `os`-module kan geven over bestanden en directories.

16 TEKSTBESTANDEN

Inleiding

Tot nu kwam de input steeds van de gebruiker en kwam de output op het scherm. De meeste toepassingen die geschreven worden hebben bestanden als input en bestanden als output.

In de voorbeelden wordt een bestand gebruikt `beleg.txt`. Het bevat volgende 5 lijnen:

```
kaas  
boter  
choco
```

```
gelei
```

De 4^{de} lijn is leeg.

Platte tekstbestanden

Tekstbestanden of 'flat files' bestaan uit regels tekst, opgebouwd uitsluitend uit tekens.

Elke regel wordt op het einde afgesloten met het speciale symbool newline `\n`.

Je kan zo'n bestand lezen met bijvoorbeeld de Kladblok applicatie in Windows.

Je ziet de newline niet omdat als Kladblok zo'n symbool ziet, hij naar een nieuwe regel gaat.

Je ziet meestal aan de extensie van het bestand of het een tekstbestand is of niet.

Voorbeelden van tekstbestanden : `txt`-bestanden, `csv`-bestanden, `py`-bestanden, `html`-bestanden, ...

Word-bestanden en afbeeldingen zijn bijvoorbeeld geen flat-files, maar binaire bestanden. Deze bestanden moeten op andere manieren benaderd worden.

Testbestand

In de voorbeelden in dit hoofdstuk maken we gebruik van een bestand met 5 lijnen.

```
kaas  
boter  
choco
```

```
gelei
```

De 4^{de} lijn is leeg.

Bestand : handle en pointer

Voor je iets kan doen met een bestand moet je het eerst openen.
Hiervoor heeft Python de open-functie.

Voorbeeld:

```
file_beleg = open(r"H16_Data_beleg.txt")
```

De open functie heeft als parameter de naam van het bestand waarmee je iets mee wil doen en heeft soms ook parameters die aangeven wat je met het bestand wil doen.

Als je enkel de bestandsnaam opgeeft, veronderstelt Python dat het bestand met die naam in de "huidige" directory staat; de "current working directory". De current working directory is meestal de directory waar het python programma wordt uitgevoerd. Als het bestand niet in de huidige directory, staat, moet je het complete pad naar het bestand opgeven zodat Python het kan vinden.

Handle

De open instructie maakt een zogeheten "handle" of "file handle" aan. In

```
file_beleg = open(r"H16_Data_beleg.txt")
```

is file_beleg de variabele waar die handle in zit.

Vanaf het ogenblik dat je die handle hebt, kan en moet je in de rest van je programma via deze handle alle nodige operaties doen op het bestand dat je geopend hebt.

Je kan natuurlijk ook een variabele opgeven als parameter van de open functie. Dit maakt het een stuk gemakkelijker als je meerdere keren iets moet doen met hetzelfde bestand.

De volgende 2 lijnen hebben hetzelfde effect als de eerste lijn van het voorbeeld.

```
file_naam_beleg = r"H16_Data_beleg.txt"
file_beleg = open(file_naam_beleg)
```

Pointer

Na het openen van een bestand wordt er een zogenoemde 'pointer' gemaakt. Deze pointer is een plaats in het bestand en zal veranderen afhankelijk van wat het programma doet.

Als je het bestand opent staat de 'pointer' van het bestand in het begin van het bestand. Telkens je iets leest zal die pointer verschuiven vlak achter hetgeen je gelezen hebt. Iedere lees-operatie start vanaf de positie van de pointer.

Als je alles van het bestand gelezen hebt staat de pointer op het einde van het bestand.
Als je dan nog iets zou lezen, lees je een lege string .

Bestand volledig lezen met read en sluiten met close

H16_Voorbeeld_01_read

```
001 file_beleg = open(r"H16_Data_beleg.txt")
002 soorten_beleg = file_beleg.read()
003 print(soorten_beleg)
004 file_beleg.close()
```

Geeft op het scherm:

```
kaas
boter
choco

gelei
```

Lijn	Uitleg
001	Opent het bestand met de opgegeven naam. Je ziet een r voor de naam van het bestand : als je niet wil dat de \ als speciale karakter wordt geïnterpreteerd en dat Python de karakter exact moet nemen zoals ze tussen aanhalingstekens staan. file_beleg is de handle die overeenkomt met het bestand. Vanaf nu kan je met die handle het bestand behandelen.
002	<code>file_beleg.read()</code> : lees het bestand vanaf de pointer (dus vanaf het begin want je hebt nog niets gelezen) tot het einde. Zet het resultaat in variabele soorten_beleg. Als je in debug mode gaat kijken in die variabele dan zie je de string "kaas\nboter\nchoco\n\ngelei" . Je ziet hier de \n terug komen: de new line, neem een nieuwe regel. Op het einde van het bestand staat geen \n.
003	Door de \n's die in het bestand zit zal er dus op de juiste plaats een nieuwe regel genomen worden.
004	Sluit het bestand. Advies : zorg ervoor dat op het einde van je programma alle gebruikte bestanden gesloten zijn. Zeker als je schrijft naar een bestand: dan ben je zeker dat alles weggeschreven is. In sommige situaties wordt er immers niet dadelijk naar het bestand geschreven, maar naar een buffer die pas weggeschreven wordt op het ogenblik van de close.

Regels één voor één lezen met readline()

Je kan ook lijn per lijn een bestand lezen :

H16_Voorbeeld_02_readline

```
001 file_naam_beleg = r"H16_Data_beleg.txt"
002 file_beleg = open(file_naam_beleg)
003 lijn = file_beleg.readline()
004 while lijn :
005     lijn = lijn.rstrip("\n")
006     print(lijn)
007     lijn = file_beleg.readline()
008 file_beleg.close()
```

Lijn	Uitleg
003	Leest vanaf de pointer tot en met de eerstvolgende \n en zet hetgeen je leest in variabele lijn en zet de pointer achter die eerstvolgende \n.
004	Een while kan ook gebruikt worden met enkel een variabele erachter. Als die variabele iets bevat interpreteert while dit als TRUE en gaat dus verder met de loop. Als de variabele "" (niets) bevat interpreteert while dit als FALSE en stopt de loop. Dit is een handige methode om een bestand te lezen tot het einde van dat bestand. Immers als de pointer op het einde staat en je leest nog eens dan is het resultaat ""
008	De ingelezen tekstlijn wordt aan de rechterkant gestript van de \n. (rstrip) Als je dat niet zou doen, dan zal de print op lijn 009 een 2 new lines geven : 1 van de gelezen lijn en 1 van de print.
009	leest de volgende lijn

Alle regels lezen in een list met readlines()

H16_Voorbeeld_03_readlines

```
001 file_naam_beleg = r"H16_Data_beleg.txt"
002 file_beleg = open(file_naam_beleg)
003 lijst_beleg = file_beleg.readlines()
004 for beleg in lijst_beleg :
005     beleg = beleg.rstrip('\n')
006     print(beleg)
007 file_beleg.close()
```

Geeft op het scherm:

```
kaas
boter
choco

gelei
```

Lijn	Uitleg
003	Leest alle lijnen van het bestand en zet ze in de list met naam lijst_beleg. Dus de eerste lijn van het bestand komt terecht op index 0, de volgende op index 1, enzovoort.
004	Omdat alle informatie nu in een list zit kan je die list gemakkelijk doorlopen met een for loop.
005	Er moet nog steeds gestript worden.

Wat gebruiken : read, readline of readlines?

Een paar bedenkingen:

- Het nadeel van read en readlines is dat het ganse bestand in het geheugen gelezen wordt, wat bij heel grote bestanden een probleem kan zijn.
- Het voordeel van readlines is dat alle informatie in een list komt. Je kan dus alle nuttige methods voor lists gebruiken.
- Het voordeel van readline is dat je je programma gemakkelijker kan testen door bijvoorbeeld maar een paar lijnen van het eventueel heel grote bestand te verwerken. Als de verwerking dan foutvrij is, kan je de logica op het ganse bestand laten werken.

Dus je gebruikt het afhankelijk van de situatie.

Schrijven in tekstbestanden met write

H16_Voorbeeld_04_writeline

001	<code>bestand_met_woorden = open(r"H16_Out_Data_woorden.txt", "w")</code>
002	<code>woord = input("Geef een woord en duw <enter>")</code>
003	<code>while woord != "" :</code>
004	<code> bestand_met_woorden.write(woord + "\n")</code>
005	<code> woord = input("Geef een woord en duw <enter>")</code>
006	<code>bestand_met_woorden.close()</code>

Lijn	Uitleg
001	Om een bestand te kunnen schrijven doe je ook een open, maar je geeft ook de parameter "w" mee. Opgepast : als je een bestand opent met "w" wordt er een nieuw bestand gemaakt. Als er al een bestand is met dezelfde naam, wordt dat bestand overschreven . Als je dat niet wil, of soms wel wil, moet je zelf instructies in je programma schrijven om dat al dan niet toe te laten. Als je lijnen wil toevoegen aan een bestand, dan gebruik je in plaats van de parameter "w" de parameter "a". "a" van append. Als bij append het bestand niet bestaat, wordt het nieuw aangemaakt.
003	Loop zolang er iets ingetypt is
004	Schrijf weg naar het bestand. Schrijven kan je doen met write, maar vergeet niet een \n toe te voegen als je wil dat ieder woord op een nieuwe lijn begint.

Dit voorbeeld is niet helemaal OK. Wat scheelt er aan en hoe kan je het aanpassen om het wel 100% in orde te krijgen?

Schrijven met writelines()

Zoals je een bestand kan inlezen in een lijst, zo kan je ook een ganse lijst met één instructie wegschrijven naar een bestand.

Hier kan je writelines() voor gebruiken.

H16_Voorbeeld_05_writelines

```
001 file_naam_beleg = r"H16_Data_beleg.txt"
002 file_beleg = open(file_naam_beleg)
003 file_filtered_beleg = open(r"H16_Out_Data_beleg_f.txt", "w")
004 lijst_beleg = file_beleg.readlines()
005 for beleg in lijst_beleg :
006     if beleg > "choco":
007         lijst_beleg.remove(beleg)
008 file_filtered_beleg.writelines(lijst_beleg)
009 file_beleg.close()
010 file_filtered_beleg.close()
```

Lijn	Uitleg
004	Lees alle lijnen en zet ze in een list
005	Doorloop gans de list
006	Als het element groter is dan "choco"
007	Verwijder het uit de list
008	Schrijf alle elementen van de list weg naar het bestand

Encoding

Soms krijg je bij het lezen van een bestand een UnicodeDecodeError.

Een voorbeeld :

```
UnicodeDecodeError: 'utf8' codec can't decode byte 0xa5 in position
0: invalid start byte
```

De oorzaak is dikwijls dat dat bestand op een bepaalde manier 'ge-encodeerd' is. en dat je het op een andere manier wil lezen waardoor er "vertaal" problemen ontstaan.

Dikwijls zie je dan in het bestand dingen zoals:

```
2020-03-16, "Liège", "Wallonia", 12, 24, 2, 2, 0, 2, 1
```

Hier is de è op een andere manier vertaald.

encoderen : omzetten van karakters naar codes. Dit kan op verschillende manieren. Een manier is de ASCII-codering die je al tegengekomen bent in het hoofdstuk over condities bij strings.

Als je een tekstbestand opent met de Kladblok, dan zie je rechts beneden de encoding van dat bestand. Iets in de aard van:

100%	Windows (CRLF)	UTF-8
------	----------------	-------

Hier is encoding dus UTF-8

Als je een bestand wil openen met specifieke codering, geef dan de encoding parameter mee bij het openen van dat bestand.

Bijvoorbeeld:

```
bestand_met_woorden = open("Woorden2.txt", encoding='latin-1')
```

Controleer ook of je IDE in een goede encoding staat. Dit is meestal duidelijk zichtbaar.

CSV bestanden

CSV bestanden zijn speciale tekstbestanden.

CSV = Comma Separated Values.

Iedere lijn (record) bestaat uit verschillende stukken die gescheiden zijn door hetgeen men noemt een delimiter.

Deze delimiter kan een komma zijn maar is ook dikwijls een punt-komma.

Een voorbeeld van de typische inhoud van een csv-bestand:

```
"beleg";"aantal";"prijs"
"kaas";"6";"8.45"
"boter";"12";"12.45"
"choco";"0";"21.50"
"gelei";"9";"10.05"
```

In dit voorbeeld is ieder element van de lijn afgebakend met dubbele quotes. Soms is dat niet zo.

csv-bestanden worden dikwijls gebruikt om gegevens tussen bedrijven uit te wisselen of om gegevens aan een gebruiker te bezorgen. Bijvoorbeeld bij een aantal banken kan je je rekeninguittreksels krijgen in csv formaat. Even googlen met "csv overheid" en je zal zien dat er vele gegevens beschikbaar worden gesteld met csv-bestanden.

Om een csv bestanden te verwerken kan je het bestand lijn per lijn lezen.

Uit iedere lijn kan je de stukken halen met behulp van de functie split.

Hiermee kan je dan verder werken.

Dikwijls bevat de eerste lijn van een csv bestand de "header" waar je de verschillende namen vindt van de stukken informatie in een record. Indien de eerste lijn die informatie niet bevat dan geeft de instantie die de csv publiceert je de nodige informatie.

Je moet immers weten welke informatie waar in het bestand staat.

Met hetgeen je tot nog toe geleerd hebt kan je dus csv-bestanden volledige zelf gaan behandelen.

Python heeft echter ook standaard een module met een aantal handige methods.

Stel in het csv-bestand zit :

```
"beleg";"aantal";"prijs"
"kaas";"6";"8.45"
"boter";"12";"12.45"
"choco";"0";"21.50"
"gelei";"9";"10.05"
```

H16_Voorbeeld_06_read_csv

```
001 import csv
002 file_beleg = open(r"H16_Data_beleg.csv")
003 lijst_beleg = csv.reader(file_beleg, delimiter = ";")
004 for beleg in lijst_beleg :
005     print(beleg)
006 file_beleg.close()
```

Geeft op het scherm :

```
['beleg', 'aantal', 'prijs']
['kaas', '6', '8.45']
['boter', '12', '12.45']
['choco', '0', '21.50']
['gelei', '9', '10.05']
```

Lijn	Uitleg
001	Importeer csv module
003	Gebruik de reader method om de data "in te lezen". Handig is hierbij dat je kan opgeven wat de delimiter is. Hierdoor zal de lijn mooi opgesplitst worden in de juiste stukken.
004	Ga lijn per lijn door hetgeen je ingelezen hebt.
005	Print het uit. Hier zie je dat iedere lijn een lijst is. Met deze lijst kan je weer iets doen.

Je kan ook zelf csv bestanden aanmaken:

Bekijk volgende voorbeeld en ga na wat er gebeurt.

H16_Voorbeeld_07_write_csv

```

001 import csv
002 file_beleg = open(r"H16_Data_beleg.csv")
003 file_beleg_update = open(r"H16_Out_Data_beleg_update.csv", "w", newline='')
004 # newline want anders komt een newline achter iedere gelezen lijn
005 lijst_beleg = list(csv.reader(file_beleg, delimiter = ";"))
006 lijst_beleg_update = csv.writer(file_beleg_update, delimiter = ";",
007     quoting=csv.QUOTE_NONNUMERIC)
008 print(lijst_beleg)
009 nieuwe_lijst_beleg = []
010 nieuwe_lijst_beleg.append(lijst_beleg[0])
011 for teller in range(1, len(lijst_beleg)):
012     beleg = lijst_beleg[teller]
013     print(beleg)
014     if beleg[1] != "0" :
015         nieuwe_lijst_beleg.append(beleg)
016 lijst_beleg_update.writerows(nieuwe_lijst_beleg)
017 file_beleg.close()
018 file_beleg_update.close()

```

Dit waren eenvoudige voorbeelden van hetgeen de csv module kan.
 Raadpleeg de officiële Python documentatie voor verdere details.
 Of google "python csv" voor talloze mogelijkheden.

Na dit hoofdstuk kan je:

- uitleggen wat een tekstbestand is
- de begrippen handle en pointer uitleggen
- tekstbestanden lezen :
 - in één keer
 - lijn per lijn
 - alle lijnen
- tekstbestanden schrijven
 - lijn per lijn
 - meerdere lijnen
- tekstbestanden wijzigen
- uitleggen wat een csv bestand is
- csv bestanden lezen en schrijven
- uitleggen wat encoding is en typische problemen geven die hieruit kunnen voortvloeien

17 TIJD

Inleiding

Dikwijls moet je werken met tijd.

In dit hoofdstuk ga je een aantal bewerkingen met tijd uitvoeren.

Datums

H17_Voorbeeld_01_datums

```
001 import datetime
002 vandaag = datetime.date.today();
003 print("Het is vandaag: ", vandaag)
004 print(vandaag.day, "/", vandaag.month, "/", vandaag.year)
005 print(vandaag.strftime("%A %d %B in het jaar %Y"))
006 periode = datetime.timedelta(days=100)
007 print("Vandaag +", periode.days, "dagen=", vandaag + periode)
008 dubbel = periode * 2
009 print("Vandaag +", dubbel.days, "dagen=", vandaag + dubbel)
010 nieuwjaar = datetime.date(2023,1,1)
011 tot_nieuwjaar = nieuwjaar - vandaag
012 print("Nog", tot_nieuwjaar.days, "dagen tot nieuwjaar!")
```

Geeft op het scherm:

```
Het is vandaag: 2022-05-21
21 / 5 / 2022
Saturday 21 May in het jaar 2022
Vandaag + 100 dagen= 2022-08-29
Vandaag + 200 dagen= 2022-12-07
Nog 225 dagen tot nieuwjaar!
Datum en tijd
```

Lijn	Uitleg
001	Import de module datetime
002	Maakt een datetime object voor de datum van vandaag.
003	Print informatie van het object vandaag (day, month, year)
005	Zorg met de methode strftime dat de datum van vandaag op een bepaalde manier op het scherm komt. Die manier wordt bepaald door de %A, %d, %B. Er is een lange lijst van mogelijke codes om eender welk datumformaat op het scherm te brengen.
006	Maakt een datetime.timedelta object met 100 dagen. Hiermee ga je rekenen.
007	Telt het datetime.timedelta object op bij het vandaag object om de datum 100 dagen vanaf nu te bekomen.
008	Je kan rekenen met timedelta objecten.
011	Zet nieuwjaar in een datetime object.
014	Je kan het verschil berekenen tussen datetime objecten.

Tijden

H17_Voorbeeld_02_tijden

```

001 import datetime
002 nu = datetime.datetime.now()
003 print(type(nu))
004 print("Het is nu: ", nu)
005 print(nu.hour, "/", nu.minute, "/", nu.second, "/", nu.microsecond)
006 print(nu.strftime("%H %M %S stipt!"))
007 periode = datetime.timedelta(seconds=10000)
008 print("nu +", periode.seconds, "seconden=", nu + periode)
009 dubbel = periode * 2
010 print("nu +", dubbel.seconds, "seconden=", nu + dubbel)

```

Geeft op het scherm:

```

<class 'datetime.datetime'>
Het is nu: 2022-05-21 21:56:51.248429
21 / 56 / 51 / 248429
21 56 51 stipt!
nu + 10000 seconden= 2022-05-22 00:43:31.248429
nu + 20000 seconden= 2022-05-22 03:30:11.248429

```

Lijn	Uitleg
002	Haalt de huidige tijd op en zet die in een datetime object.
003	Nu is dus een object van het soort datetime.datetime
005	Print verschillende informatie van het object.
006	Met strftime (string format time) zet je de tijd in een bepaald formaat om het scherm. Er zijn talloze formaten beschikbaar die je via de % codes gebruiken.
007	Maakt een timedelta object van 10000 seconden.
008	Tel het op bij het huidige tijdstip.
009	Rekent met timedelta objecten.

Plaatselijke tijdsbenamingen

Soms is het nodig de tijdsbenamingen te gebruiken van een bepaald geografisch gebied. Probeer dit evenwel te vermijden.

H17_Voorbeeld_03_locale

```

001 import datetime
002 import locale
003 nu = datetime.datetime.now()
004 print("Het is nu: ", nu)
005 print(nu.strftime("%A %d %B in het jaar %Y"))
006 locale.setlocale(locale.LC_TIME, 'nl_BE.UTF-8')
007 print(nu.strftime("%A %d %B in het jaar %Y"))

```

```

008 locale.setlocale(locale.LC_TIME, 'fr_BE.UTF-8')
009 print(nu.strftime("%A %d %B in het jaar %Y"))
010 datum_string = "dimanche 19 juin in het jaar 2022"
011 datum_object = datetime.datetime.strptime(\
012     datum_string, "%A %d %B in het jaar %Y")
013 print(datum_object)

```

Geeft op het scherm:

```

Het is nu: 2022-05-21 22:06:40.929527
Saturday 21 May in het jaar 2022
zaterdag 21 mei in het jaar 2022
samedi 21 mai in het jaar 2022
2022-06-19 00:00:00
De module datetime

```

Lijn	Uitleg
002	Importeert de locale module die zorgt voor omvormingen van namen naar de lokaal gebruikte namen.
006	Zet de uitvoer naar België Nederlands.
008	Zet de uitvoer naar België Frans.
010	Het kan ook andersom : starten met een string in de lokale taal
011	en die string omzetten naar een datetime object gebruik maken van strptime en de juist % code.

Duurtijd laten berekenen

H17_Voorbeeld_04_hoe_lang

```

001 import time
002 start_tijd = time.time()
003 print(start_tijd)
004 a=0
005 for i in range(1000000):
006     a=a+1
007     time.sleep(1)
008 eind_tijd = time.time()
009 print("Duur:",eind_tijd - start_tijd)

```

Geeft het aantal seconden na 1 januari 1970.

Lijn	Uitleg
001	Importeert de module time : dit is een handige module om tussentijden te tonen.
002	Steekt het aantal seconden tussen dit ogenblik en 1 januari 1970 middernacht in een variabele.

003	Print die variabele. Op het scherm komt het aantal seconden sinds 1 januari 1970 middernacht. Niet zo interessant, maar het doel is om het aantal seconden tussen twee ogenblikken te meten.
005-006	Doet 100.000 optellingen.
007	doet 1 seconde niets.
008	Steekt het aantal seconden tussen dit ogenblik en 1 januari 1970 middernacht in een andere variabele.
009	Print het verschil tussen die twee aantallen seconden om zo de duurtijd van de 100.000 optellingen te weten plus die 1 seconde dat er niets gebeurde.

Kalender

H17_Voorbeeld_05_kalender

001	<code>import calendar</code>
002	<code>maand = calendar.monthcalendar(2022, 5)</code>
003	<code>print(calendar.month(2022, 5))</code>
004	<code>print (calendar.calendar(2022, 2, 1, 7))</code>

Lijn	Uitleg
002	Berekent de kalender voor de opgegeven maand mei 2022. Zet het resultaat in een lijst.
003	Print de kalender voor de opgegeven maand.
004	Print de kalender voor het opgegeven jaar.

Na dit hoofdstuk kan je:

- de juiste modules en functies gebruiken om met tijd te werken
- werken met datums en tijden.
- meten hoelang je programma duurt
- kalenders ophalen en afdrukken

18 PYTHON UITVOEREN VANAF DE COMMAND LINE

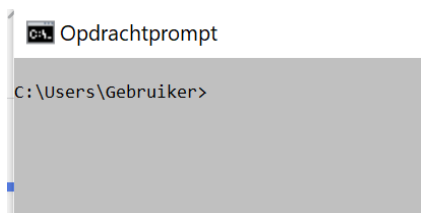
Inleiding

Tot nu toe heb je programma's laten lopen in de IDE.

Voor een gebruiker is dat niet nodig. Hij wordt liefst niet geconfronteerd met een complex gereedschap.

Dikwijls is het handiger dat hij een commando kan intypen en dat commando doet dan hetgeen wat gevraagd is.

Een commando intypen zal je moeten doen op de command line. Voor Windows zal dat er uit zien zoals:



Het is soms ook belangrijk dat je het programma kan uitvoeren zonder dat er een gebruiker bij te pas komt. Je wil bepaalde zaken laten uitvoeren buiten de kantooruren, 's nachts, op een bepaald uur, zodat je 's morgens ziet wat er uitgevoerd is en wat het resultaat. Je kan dus de uitvoering van je programma laten opnemen in een script dat iedere dag op een bepaald uur uitgevoerd wordt.

In het uitgebreide voorbeeld zie je dat je in je programma informatie kan zetten die een gebruiker kan helpen te begrijpen wat het programma doet.

Om de gebruiker nog meer te helpen kan je in je programma instructies gebruiken die bijkomende informatie tonen terwijl het programma aan het lopen is.

Dit kan heel nuttig zijn, zeker als sommige operaties wat langer duren.

Informeer de gebruiker wat er aan het gebeuren is, of als het kan hoe lang hij moet wachten op een volgende resultaat.

Hetgeen volgt is slechts een greep uit de vele mogelijkheden van commandline verwerking. Voor voorbeelden : google op "python command line" .

Eenvoudig voorbeeld

Het voorbeeld is zo eenvoudig om duidelijk de werking te laten zien.
Stel dat je de gebruiker via een commando 2 getallen wil laten delen door elkaar.
In ons voorbeeld zal hij kunnen intypen achter de command prompt :

```
python H18_Voorbeeld_01_deel.py 5 2
```

en zal er op het scherm een 2 komen.

Lijn	Uitleg
python	Je moet python uitvoeren om een python programma te laten runnen. Vandaar dat je moet starten met python. Bij sommige installatie is het voldoende op py te grebuiken.
deel.py	Dit is de naam van het bestand waarin het python programma staat. Dit moet je dus zeker testen, want soms moet je de volledige directory meegeven, je moet genoeg toelatingen hebben. Hier moet je goed kijken wat het operating systeem allemaal nodig heeft om exact het bestand met het programma te localiseren.
5 2	Het programma heeft 2 parameters 5 en 2. Deze worden door elkaar gedeeld en het resultaat wordt afgerond zodat er 2 op het scherm komt.

Het programma waarmee je dit laat doen:

H18_Voorbeeld_01_deel

001	<code>import argparse</code>
002	<code>parser = argparse.ArgumentParser()</code>
003	<code>parser.add_argument('getal1', type=float)</code>
004	<code>parser.add_argument('getal2', type=float)</code>
005	<code>args = parser.parse_args()</code>
006	<code>resultaat = round(args.getal1 / args.getal2)</code>
007	<code>print(resultaat)</code>

Lijn	Uitleg
001	Om een programma te maken dat je aan de commandline kan uitvoeren met parameters, heb je de module argparse nodig.
002	Start de parser op.
003	Zeg dat er een eerste parameter is, met naam getal1 van het type float.
004	Zeg dat er een tweede parameter is, met naam getal2 van het type float.
005	Haal die parameters op.
006	Doe de berekening.
017	Zet het resultaat op het scherm

De naam die je aan dit programma geeft bepaalt welk commando je moet ingeven om het te laten uitvoeren.

Als je denkt aan de gebruiker probeer je een zo kort mogelijke maar toch duidelijke naam te bedenken.

Voorbeeld met extras

H18_Voorbeeld_02_deel_extra

```

001 import argparse
002 parser = argparse.ArgumentParser(\
003     description='Bepaal het quotient van twee getallen')
004 parser.add_argument('getal1', type=float, help='Het eerste getal.')
005 parser.add_argument('getal2', type=float, help='Het tweede getal.')
006 parser.add_argument("-d", "--decimalen", default=2,\
007     type=int, \
008     help="Het aantal decimalen dat getoond wordt in het resultaat.")
009 parser.add_argument("-v", "--verbose", action="store_true", \
010     help="Print extra informatie uit tijdens het uitvoeren van het script.")
011 args = parser.parse_args()
012 if args.verbose:
013     print("Berekenen van het quotient...")
014     print("Afronden naar", args.decimalen, " decimalen...")
015 resultaat = round(args.getal1 / args.getal2, args.decimalen)
016 print(resultaat)
017 if args.verbose:
018     print("Script succesvol uitgevoerd!")

```

Als je aan de command line zou intypen:

```
python H18_deel_extra_c.py -h
```

Krijg je :

```
usage: H18_deel_extra_c.py [-h] [-d DECIMALEN] [-v] getal1 getal2
```

Bepaal het quotient van twee getallen

positional arguments:

getal1	Het eerste getal.
getal2	Het tweede getal.

optional arguments:

-h, --help	show this help message and exit
-d DECIMALEN, --decimalen DECIMALEN	Het aantal decimalen dat getoond wordt in het resultaat.

-v, --verbose	Print extra informatie uit tijdens het uitvoeren van het script.
---------------	--

Je kan dus in je programma nuttige help informatie stoppen.

Lijn	Uitleg
003	description : zorgt voor uitleg als de gebruiker help vraagt.
006	Geeft de mogelijkheid om een bijkomende parameter mee te geven voorafgegaan door -d of --decimalen. De waarde van die opgeven parameter komt in args.decimalen. Als de gebruiker die parameter niet geeft, is hij gelijk aan 2. (default).
008	Extra help voor de gebruiker.
009	Voegt de verbose parameter toe die terecht komt in args.verbose. Als de parameter aanwezig is, is args.verbose True anders False. Verbose is een typisch programmeerwoord en zegt eigenlijk of er tijdens de loop van het programma extra uitleg wordt gegeven op het scherm.
011	Zet de ingegeven parameters in de args variabelen.
012	Als je -v gebruikt hebt op de command line, geef dan meer uitleg tijdens het runnen.

Als je in met dit voorbeeld op de command zou intypen :

```
python H18_deel_extra_c.py -d 4 -v 3 7
```

dan krijg je als output op het scherm:

```
Berekenen van het quotient...
Afronden naar 4 decimalen...
0.4286
Script succesvol uitgevoerd!
```

Na dit hoofdstuk kan je:

- uitleggen wat het nut kan zijn van het uitvoeren van een Python programma op de command line
- een eenvoudig programma schrijven met parameters dat kan uitgevoerd worden op de command
- uitleggen hoe je de gebruiker van je programma nog meer informatie kan geven over de werking van het programma

19 EXCEPTIONS

Inleiding

Heel dikwijls zal een programma plots stoppen omdat het een fout tegenkomt. Python stopt het programma en geeft een niet zo gebruikersvriendelijke boodschap.

Zolang jij de enige persoon bent die het programma gebruikt is dat niet zo erg. Jij weet immers waarom die boodschap er staat en jij kan dikwijls snel de nodige aanpassingen doen om er voor te zorgen dat de fout niet meer voorkomt.

Maar als iemand anders je programma gebruikt kan je beter doen. Je kan op voorhand nadenken over de mogelijke fouten die zich kunnen voordoen en in je programma code schrijven zodat als de fout zich voordoet, de reactie van het programma een stuk gebruiksvriendelijker wordt.

Het probleem is dat je soms niet op voorhand alle fouten kan voorzien.

Om dit probleem op te lossen kan je aan "exception handling" doen : code schrijven die wordt uitgevoerd als er zich eender welke fout voordoet.

Exception handling

H19_Voorbeeld_01_zonder

```
001 getal = int(input("Geef een getal en duw enter: " ))
002 print( "1 gedeeld door",getal,"is", 1/getal, "." )
003 print( "Einde programma!" )
```

Geeft op het scherm (als je 0 in geeft):

```
Geef een getal en duw enter: 0
Traceback (most recent call last):
  File "c:\Users\Gebruiker\OneDrive - Thomas More\Documenten\00.
  Lesgeven\Cevora Python\01. Voorbeelden\H17.
  Exceptions\H17_Voorbeeld_01_zonder_c.py", line 5, in <module>
    print( "1 gedeeld door",getal,"is", 1/getal, "." )      # 005
ZeroDivisionError: division by zero
```

Lijn	Uitleg
001	Vraagt om iets in te voeren.
002	Als je op lijn 001 0 hebt ingegeven, dan zal 1/getal niet lukken. (delen door 0 gaat niet). Het programma stopt en geeft de boodschap.
003	Je komt dus nooit op lijn 003 als je een 0 ingeeft bij de uitvoering van lijn 002

Maar ook als je bijvoorbeeld iets anders invoert :

Geef een getal en duw enter: choco

Traceback (most recent call last):

File "c:\Users\Gebruiker\OneDrive - Thomas More\Documenten\00. Lesgeven\Cevora Python\01. Voorbeelden\H17. Exceptions\H17_Voorbeeld_01_zonder_c.py", line 4, in <module>

```
    getal = int(input("Geef een getal en duw enter: " )) # 004
ValueError: invalid literal for int() with base 10: 'choco'
```

Lijn	Uitleg
001	Vraagt om iets in te voeren.
002	Als je op lijn 001 "choco" ingegeven, dan reclameert de int functie. Die aanvaard enkel strings die kunnen omgezet worden naar een integer.

Python geeft wel aan wat het probleem is. In het eerste geval zie je ZeroDivisionError en in het tweede geval ValueError. Deze informatie kan je nuttig gebruiken in je programma. Python bevat instructies om deze fouten op te vangen. En dat is nodig als je de gebruiker niet wil confronteren met bovenstaande moeilijk begrijpbare foutboodschappen.

H19_Voorbeeld_01_try

```
001 try:
002     getal = int(input("Geef een getal en duw enter: " ))
003     print("1 gedeeld door",getal,"is", 1/getal, " ." )
004 except ZeroDivisionError :
005     print( "Je gaf een 0 in en ik kan niet delen door nul" )
006 except ValueError:
007     print( "Je gaf geen getal in!" )
008 except:
009     print("Er is iets onverwacht gebeurd!")
010 print( "Einde programma!" )
```

Lijn	Uitleg
002	Vraagt om iets in te voeren.
004	Als de error ZeroDivisionError voorvalt dan print ...
006	Als je geen getal zou ingegeven hebben zal de int functie de conversie naar int niet kunnen doen en zal er een fout optreden.
008	Als er zich toch nog een fout voordoet die niet binnen de toe nog toe opgesomde mogelijke fouten hoort, dan print ...

Veel gebruikte exceptions :

Exception	Uitleg
ZeroDivisionError	Bij het delen door nul
IndexError	Bij het benaderen van een list of tuple met een index die niet binnen het legale bereik valt.
KeyError	Bij het benaderen van een dictionary met een key die onbekend is.
IOError	Bij iedere fout die kan optreden als je een bestand benadert.
FileNotFoundError	Bij het proberen te openen van een niet-bestaand bestand om eruitte lezen
ValueError	Bij "casten" van een waarde naar een andere waarde.

Al deze exceptions kan je nuttig gebruiken als je wil vermijden dat een gebruiker cryptische foutboodschappen krijgt.

Na dit hoofdstuk kan je:

- een paar voorbeelden geven van situaties waarbij je programma een error zou geven die jij zou kunnen opvangen met programma-instructies
- de try: except gebruiken om fouten af te handelen

20 OBJECT ORIENTATIE

Inleiding

Object Oriented Programmeren : **OOP**

Is een manier van programmeren die voortvloeit uit Object Oriented Ontwerpen. Je maakt een ontwerp : je kijkt welke objecten er in je probleem voorkomen. Je kijkt welke informatie er over die objecten moet bijgehouden worden. (de eigenschappen van het object) en je bepaalt welke acties je wil uitvoeren op dat object (de methods van dat object). Al de eigenschappen en methods worden vastgelegd in de definitie van een klasse.

Klassen en objecten

Een klasse is een blauwdruk, een sjabloon waarmee je in je programma objecten kan maken. In die klasse leg je niet alleen vast welke eigenschappen het object heeft, maar ook wat je met dat object kan doen : de methods. Je hebt in vorige hoofdstukken vele kant een klare methods gebruikt. Je gaat nu zelf voor de klassen die je zelf maakt zelf methodes kunnen programmeren.

H20_Voorbeeld_01_cursist

```

001 class Cursist:
002     def __init__(self, naam, voornaam, geboortedatum):
003         self.naam = naam
004         self.voornaam = voornaam
005         self.geboortedatum = geboortedatum
006         self.aanwezig = False
007
008     def komt_binnen(self):
009         self.aanwezig = True
010     def gaat_buiten(self):
011         self.aanwezig = False
012
013 cursist_1 = Cursist("Peeters", "Jan", "1990-12-13")
014 cursist_2 = Cursist("Hermans", "Marie", "1993-08-22")
015 print(cursist_1.naam, cursist_1.aanwezig)
016 cursist_1.komt_binnen()
017 print(cursist_1.naam, cursist_1.aanwezig)

```

Geeft op het scherm :

```

Peeters False
Peeters True

```

Lijn	Uitleg
001	Start van de definitie van de klasse Cursist.
002	Definieer de constructor, de <code>__init__</code> method. Een eerste methode waar je best altijd voor zorgt is de <code>__init__</code> method. Het is de method die er voor zorgt dat een object wordt aangemaakt. Telkens je een object aanmaakt in je programma, worden deze instructies uitgevoerd. Tussen de ronde haakjes zie je de parameters die je opgeeft bij het aanmaken van het object. De eerste parameter wordt meestal self genoemd en moet er steeds zijn.
003-006	Vult de data op die bij het object hoort. Gebruik steeds self. gevolgd door de eigenschap van het object.
008	Definieert een method <code>komt_binnen</code> .
010	Definieert een method <code>gaat_buiten</code> .
013-014	Maakt 2 objecten van het type Cursist en geef het de eigenschappen die tussen de haakjes staan.
015	Print de eigenschappen van een object. Een eigenschap kan je opvragen met de naam van het object gevolgd door een punt gevolgd door de naam van de eigenschap. Zoals je vroeger al deed bij methods voor strings en lijsten.
016	Roept de method <code>komt_binnen</code> op voor het object <code>cursist_1</code> . Zoals gewoonlijk : de objectnaam, een punt er achter en dan de naam van de zelfgemaakte method.
017	Print opnieuw de eigenschappen van het object <code>cursist_1</code> . Je ziet dat de method opgeroepen in lijn 016 de eigenschap aanwezig heeft veranderd.

Samengevat :

De definitie van een klasse start steeds met `class` gevolgd door de naam van de klasse gevolgd door een dubbelpunt.

Voorzie steeds een constructor; een `__init__` methode.

De constructor wordt steeds uitgevoerd als het object gemaakt wordt.

Encapsulation

Een woord dat je dikwijls hoort bij object orientation. Inkapseling, je pakt een stukje logica in en zet het bij de klasse waar het bij hoort. Je hoeft er niet meer naar om te kijken, als je het nodig hebt roep je het op. Je hoeft zelfs niet meer te weten hoe het intern in elkaar zit. Alle logica zit verborgen achter de klasselogica.

In het eerste voorbeeldje zie je een method `komt_binnen` die hoort bij de klasse Cursist. Alle instructies die moeten uitgevoerd worden als een cursist binnenkomt kunnen hier "ingekapseld" worden. Een buitenstaander hoeft zelf niet meer te weten wat er juist gebeurt; hij roept gewoon de method op en de juiste dingen gebeuren.

Overloading

Een andere zeer krachtige techniek die bij het object oriented programmeren gebruikt kan worden is "overloading".

Het volgende voorbeeld maakt het duidelijk.

Als Python de print functie gebruikt, dan vormt Python hetgeen tussen de haakjes staat achter de print om naar een stringvoorstelling. Intern doet hij dat met de `__str__` methode.

In het voorbeeld wordt die `__str__` methode "overloaden" voor de klasse `Cursist`.

H20_Voorbeeld_02_cursist_overloading_str

```

001 class Cursist:
002     def __init__(self, naam, voornaam):
003         self.naam = naam
004         self.voornaam = voornaam
005         self.aanwezig = False
006
007     def __str__(self):
008         text = "Cursist " + self.voornaam + " " + self.naam + " ("
009         if self.aanwezig :
010             text = text + "aanwezig)"
011         else:
012             text = text + "afwezig)"
013         return text
014
015     def komt_binnen(self):
016         self.aanwezig = True
017
018 cursist_1 = Cursist("Peeters", "Jan")
019 cursist_2 = Cursist("Hermans", "Marie")
020 print(cursist_1)
021 print(cursist_2)
022 cursist_2.komt_binnen()
023 print(cursist_2)

```

Geeft op het scherm :

```

Cursist Jan Peeters (afwezig)
Cursist Marie Hermans (afwezig)
Cursist Marie Hermans (aanwezig)

```

Lijn	Uitleg
007	Start van de overloading van <code>__str__</code> . Hier legt je vast wat je wil zien gebeuren als er een object van klasse <code>Cursist</code> "geprint" wordt.
020	Hier zie je het effect van de overloading. Python moet een object van klasse <code>Cursist</code> printen, dus hij gaat op zoek naar een method <code>__str__</code> bij die klasse en voert die uit voor het object en het resultaat zet hij op het scherm.

In dit voorbeeld werd methode `__str__` overload. Maar op identiek dezelfde manier kan je een hele reeks logica overladen :

`__add__` : als je deze methode definieert bij je klasse dan zeg je tegen Python wat hij moet doen als hij objecten van de klasse moet optellen. In ons voorbeeld zou je dan in je programma `cursist_1 + cursist_2` kunnen uitvoeren.

Hetzelfde principe geldt voor vele bewerkingen, vergelijkingsoperatoren, functies. Dit is een zeer krachtig instrument. Het laat je toe om naast zelf nieuwe objecten aan te maken en om zelf bewerkingen, specifieke voor deze objecten, te programmeren.

Relaties tussen objecten

Je kan verschillende klassen maken en voor iedere klasse objecten aanmaken. Je moet natuurlijk ook objecten met elkaar kunnen verbinden. Dat kan je ook vastleggen op niveau van de klasse definitie.

Onderstaand voorbeeld maakt dat duidelijk.

H20_Voorbeeld_03_cursist_cursus

```

001 class Cursus:
002     def __init__(self, naam, id):
003         self.naam = naam
004         self.id = id
005
006 class Cursist:
007     def __init__(self, naam, voornaam):
008         self.naam = naam
009         self.voornaam = voornaam
010         self.cursus = ""
011
012     def schrijf_in(self, cursus):
013         self.cursus = cursus
014
015 cursus_1 = Cursus("Python", "PY100")
016 cursist_1 = Cursist("Peeters", "Jan")
017 cursist_1.schrijf_in(cursus_1)

```

Lijn	Uitleg
001	Start van de definitie van de klasse Cursus.
003	Start van de definitie van de klasse Cursist.
012	Maakt een methode aan om het kenmerk cursus aan de cursist toe te kennen.
015	Maakt een object cursus_1.
016	Maakt een object cursist_1.
017	Roept de method schrijf_in met als parameter cursus_1 (het object). De method aangemaakt op lijn 012 geeft de cursist_1 het kenmerk cursus_1. Op die manier leg je een relatie tussen beide objecten.

Polymorfisme en inheritance

Een andere zeer krachtige techniek die bij het object oriented programmeren gebruikt kan worden is "polymorfisme". (vele vormen).

Dikwijls komt dit samen voor met inheritance. (overerving)

Onderstaand voorbeeld maakt dat duidelijk.

H20_Voorbeeld_04_poly

```

001 class Persoon:
002     def __init__(self, naam, voornaam):
003         self.naam = naam
004         self.voornaam = voornaam
005
006     def werk(self):
007         return("Ik doe nog niet veel")
008
009 class Cursist(Persoon):
010     def __init__(self, naam, voornaam, nummer):
011         self.nummer = nummer
012         super().__init__(naam, voornaam)
013
014     def werk(self):
015         print("Ik studeer!")
016
017 class Docent(Persoon):
018     def __init__(self, naam, voornaam, id):
019         self.id = id
020         super().__init__(naam, voornaam)
021     def werk(self):
022         print ("Ik geef les!")
023
024 cursist_1 = Cursist("Peeters", "Jan", "C01")
025 docent_1 = Docent("Hermans", "Marie", "X01")
026 cursist_1.werk()
027 docent_1.werk()

```

Geeft op het scherm :

```
Ik studeer!
Ik geef les!
```

Lijn	Uitleg
001	Start van de definitie van de klasse Persoon.
006	Definieer de method werk voor een object van klasse Persoon.
009	Start van de definitie van de klasse Cursist. Er staat: Cursist(Persoon). Python weet nu dat de klasse Cursist een subklasse is van de klasse Persoon. Hierdoor zullen alle methods die gedefinieerd zijn voor de (super)klasse Persoon overgeërfd kunnen worden door de subklassen Cursist. Tenzij je er voor kiest om zelf een specifieke method voor dat soort object te maken.

012	Gebruik de constructor van de superklasse. (overerving!)
014	Definieer de method werk voor de klasse Cursist. Polymorfisme! De method werk was er ook al voor de klasse Persoon. Maar ze krijgt hier nu een andere "vorm".
017	Start van de definitie van de klasse Persoon.
024	Definieer de method werk voor de klasse Docent. Polymorfisme!
029	Python weet met welk object hij bezig is en afhankelijk van het soort object
030	gaat de method uitvoeren.

Na dit hoofdstuk kan je:

- uitleggen wat klassen en objecten zijn
- werken met klassen en objecten
- voorbeelden geven van
 - overloading
 - encapsulation
 - inheritance
 - polymorfisme

21 SQLITE EN ANDERE DBMSSEN

Inleiding

In het hoofdstuk over tekstbestanden heb je geleerd hoe je gegevens kan opslaan in externe bestanden. Een stap verder is om die gegevens op te slaan in een databank. Vele moderne applicaties slaan hun gegevens op in relationele databanken : databanken die bestaan uit tabellen en die toegankelijk zijn via een data base management systeem (DBMS). Heel dikwijls zijn deze databanken toegankelijk met behulp van SQL : Structured Query Language.

SQL is een universele databank query-taal die in de meeste DBMSsen kan gebruikt worden.

Python bevat standaard een heel eenvoudig DBMS, maar evengoed toegankelijk met behulp van SQL : SQLite.

Het grote voordeel hiervan is dat je geen andere producten nodig hebt om je gegevens op een gestructureerd manier op te slaan.

Er zijn ook verschillende gratis programma's beschikbaar om deze databanken te beheren, zodat je niet tekens een Python programma moet schrijven om iets te doen met die databank. Bijvoorbeeld SQLiteStudio.

Een databank maken

H21_Voorbeeld_01_maak_DB

```
001 import sqlite3
002 connectie = sqlite3.connect("testschool.db")
003 c = connectie.cursor()
004 c.execute(""" CREATE TABLE cursist (
005     naam text,
006     voornaam text,
007     open_saldo real) """)
008 )
009 connectie.close
```

Kan op het scherm geven:

Geef een woord en duw <enter> : choco

```
0 )
1 ) choco
2 ) chocochocho
3 ) chocochochochocho
4 ) chocochochochochochocho
----- einde programma -----
```

```
#-----
# H20_Voorbeeld_01_maak_DB |
#-----
```

```

import sqlite3                # 004
conn = sqlite3.connect("school.db") # 005
c = conn.cursor()             # 006
c.execute(""" CREATE TABLE cursist (
    naam text,
    voornaam text,
    open_saldo real) """)      # 010
                                # 011
)                               # 012
conn.close

```

Lijn	Uitleg
001	Om SQLite kunnen gebruiken moet je de module sqlite3 importeren.
002	Om de databank te kunnen gebruiken moet je je connecteren aan die databank. Je geeft aan de functie connect de naam mee van het bestand met die databank. Als de databank niet bestaat dan zal ze aangemaakt worden.
003	Maakt een cursor. Voor iedere handeling met de databank heb je een "cursor" nodig. Met die cursor zal je SQL instructies kunnen uitvoeren.
004	Voert een SQL instructie uit. Hier wordt een SQL instructie uitgevoerd (execute) om een nieuwe tabel te maken.
012	Sluit de databank.

Records toevoegen in een tabel

Een relationele databank bestaat uit tabellen. In een tabel zitten records.

H21_Voorbeeld_02_add_DB_records

```

001 import sqlite3
002 conn = sqlite3.connect("school.db")
003 c = conn.cursor()
004 c.execute(""" INSERT INTO cursist VALUES ("Peeters", "Jan", 12.00) """)
005 c.execute(""" INSERT INTO cursist VALUES ("Hermans", "Marie", 67.50) """)
006 conn.commit
007 c.execute(""" SELECT * FROM cursist WHERE naam = "Hermans" """)
008 print(c.fetchall())
009 conn.close

```

Lijn	Uitleg
004	Voert SQL instructies uit om 2 lijnen toe te voegen aan de tabel cursist
005	
006	Voert een commit uit. Zolang commit niet is uitgevoerd worden de gegevens nog niet weggeschreven.
007	SQL om gegevens op te halen.
008	de gegevens uit de cursor worden op het scherm gezet

Andere DBMSsen

Naast SQLite dat standaard in Python zit, zijn er nog andere relationele database systemen. Een greep: Microsoft SQL Server, Oracle Database, MySQL and IBM DB2

Om deze systemen te kunnen gebruiken zul je bijkomende modules moeten installeren. Maar na die installatie en na de nodige import instructies in je Python programma, is de manier van werken met deze databanken vrijwel analoog als het werken met SQLite. Door via een connect en een cursor SQL uit te voeren kan je alle tabellen en gegevens manipuleren.

Als je met andere databanken wil werken zal jij hiervoor specifieke bijkomende software moeten installeren. Google eerst even om te zien of er ondertussen geen verbeterde of andere software ter beschikking gekomen is.

Wat ook zeer handig is dat je dan tegelijkertijd software installeert waarmee je naast Python die databanken kan benaderen. Dat is meestal een soort "workbench" waarmee je alle mogelijk operaties op databanken kan uitvoeren.

Na dit hoofdstuk kan je:

- uitleggen wat SQLITE is
- een programma maken dat een aantal elementaire operaties uitvoert op een SQLITE databank.
- uitleggen dat er nog andere DBMSsen zijn die je kan gebruiken

22 GUI APPLICATIES MAKEN

Inleiding

Het grootste deel van deze cursus bevat voorbeelden waarbij invoer en uitvoeren via de terminal gebeurt. De gebruiker voert "karakters" in en het programma zet "karakters" op het scherm.

Dit is een uitstekende manier om te leren programmeren.

Echter, in het dagelijkse leven gebruik je de computer niet op die manier. Je bent gewoon om te werken met een GUI : een Graphical User Interface.

In Python kan je ook met een GUI gaan programmeren. Er zijn standaard modules voorzien waarmee je dat kan doen.

Voorbeeld

H22_Voorbeeld_01_GUI

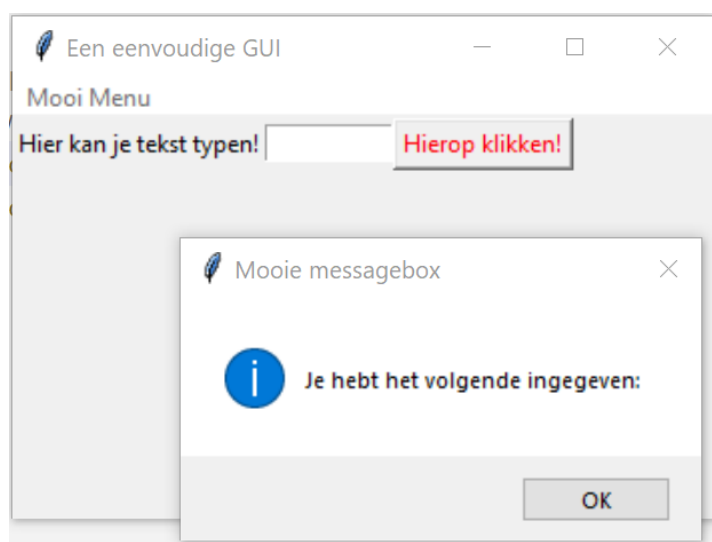
```

001 from tkinter import *
002 from tkinter import messagebox
003 root = Tk()
004 root.title("Een eenvoudige GUI")
005 root.geometry('350x200')
006 menu = Menu(root)
007 item = Menu(menu)
008 menu.add_cascade(label='Mooi Menu', menu=item)
009 item.add_command(label='Eerste keuze')
010 root.config(menu=menu)
011
012 lbl = Label(root, text = "Hier kan je tekst typen!")
013 lbl.grid()
014 tekst = Entry(root, width=10)
015 tekst.grid(column =1, row =0)
016 def geklikt():
017     resultaat = "Je hebt het volgende ingegeven:"\
018         + tekst.get()
019     messagebox.showinfo("Mooie messagebox", resultaat)
020     lbl.configure(text = "Knap, hé")
021 Knop = Button(root, text = "Hierop klikken!" ,
022             fg = "red", command=geklikt)
023 Knop.grid(column=2, row=0)
024 root.mainloop()

```


Lijn	Uitleg
001	Import de nodige modules. tkinter is de basis. raadpleeg de documentatie om
002	te weten wat je soms nog bijkomend moet importeren.
003	Start het window op.
004	Geeft een titel aan het window.
005	Geef de grootte op van het window.
006	Start een menu definitie.
007	Maakt een menu-tab met de opgegeven titel en voegt er één
008	keuzemogelijkheid aan toe.
009	
012	Maakt een label met de opgegeven tekst en plaatst op in het window.
013	
014	Maakt een veld waar je iets kan ingeven en plaatst het op het window.
015	
016	Definieert een functie die wordt uitgevoerd als er op de knop (aangemaakt vanaf lijn 021) geklikt wordt.
018	Haal de tekst op die de gebruiker heeft ingevoerd (<code>tekst.get()</code>) en zet die in een variabele.
019	Toont een messagebox met informatie.
021	Maakt een knop (command button) aan, maat die op en bepaalt wat er moet
022	gebeuren als de gebruiker op de knop klikt. Zet die knop op het window.
023	
024	Zorgt voor de "oneindige" loop : zolang de gebruiker het window niet dicht klikt blijft het programma draaien

Als je dit programma laat lopen, krijgt je een window te zien waarmee je een aantal dingen kan doen. Hierachter een screenshot:



Zoals je ziet is dit een heel ander soort programmeren : de logica is "event driven" : het reageert enkel op "events".

Events :

- een handeling van de gebruiker: een muishandeling, een toetsaanslag, ...
- een event gegenereerd door de computer : een tijdstip, een ander programma ...

In een GUI applicatie werk je veel met objecten. Objecten die kant en klaar geprogrammeerd zijn in de tkinter modules.

Een selectie uit de vele objecten :

Object	Uitleg
Window	
Menu	
Label	Om tekst op het window te plaatsen.
Entry	Om tekst te kunnen invoeren
MessageBox	
Scrollbar	Om te schuiven
Checkbox	Om vinkjes te kunnen zetten
...	

Na dit hoofdstuk kan je:

- uitleggen wat je nodig hebt om met Python aan GUI programmatie te doen.
- een eenvoudig voorbeeld lezen en uitleggen hoe het werkt
- ...

23 XML

Inleiding

In de vorige hoofdstukken heb je met bestanden gewerkt.

- Met gewone tekstbestanden bestaande uit lijnen (eindigend op de fameuze \n).
- Met csv-bestanden die al een zeer duidelijke structuur hadden met rijen en kolommen.

Deze bestandsformaten worden zeer veel gebruikt en Python voorziet standaard vele mogelijkheden om met deze bestanden te werken.

Een ander formaat dat ook veel gebruikt wordt is het XML-formaat.

In dit hoofdstuk ga je zien hoe een XML-bestand er uit ziet.

En dat Python een standaard module heeft waarmee je XML_bestanden kan lezen en schrijven.

Wat is XML

XML = eXtensible Markup Language

Het is dus een taal.

Met die taal kan je data beschrijven.

De beschrijving gebeurt grotendeels met tags (vandaar de term Markup) die je zelf kan definiëren. (vandaar de term eXtensible)

Inhoud van een XML bestand:

Voorbeeld van de inhoud van een XML bestand.

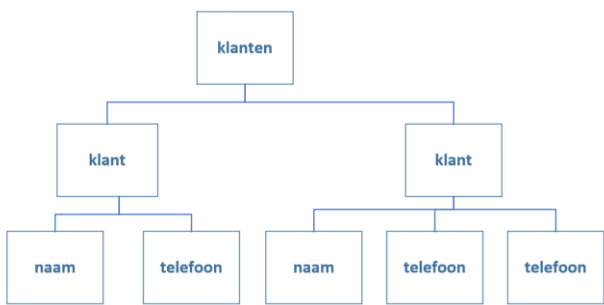
H23_Data_klanten.xml

```

001 <?xml version="1.0" encoding="utf-8"?>
002 <klanten>
003   <klant>
004     <naam>An Andriesen</naam>
005     <telefoon>014/230998</telefoon>
006   </klant>
007   <klant>
008     <naam>Bart Bols</naam>
009     <telefoon>016/238974</telefoon>
010     <telefoon>0477/182569</telefoon>
011   </klant>
012 </klanten>

```

Het is dus duidelijk een tekstbestand. Je kan het open met een standaard teksteditor.

Lijn	Uitleg
001	Geef de versie en de encoding.
002	<p>Vanaf hier staan gegevens. Je herkent in deze tekst heel duidelijk de onderstaande boomstructuur.</p>  <pre> graph TD klanten[klanten] --> klant1[klant] klanten --> klant2[klant] klant1 --> naam1[naam] klant1 --> telefoon1[telefoon] klant2 --> naam2[naam] klant2 --> telefoon2[telefoon] klant2 --> telefoon3[telefoon] </pre> <p>Dit herkennen werd je gemakkelijk gemaakt omdat de gegevens op gepaste ogenblikken geïntendeerd werden. Bij Python is dat een verplichting, bij XML helaas niet. Alles mag gewoon achter elkaar staan waardoor het natuurlijk bijna onmogelijk wordt de structuur te herkennen. Maar gelukkig heeft Python een paar methods om een xml beter leesbaar te maken.</p> <p>Op het schema : boven aan staat de "root" , in ons voorbeeld is de root : klanten.</p> <p>Alle blokjes eronder zijn de elementen.</p> <p>Ieder element heeft naam, en heeft een starttag met een bijhorende eindtag. Een starttag begint steeds met een < (kleiner dan teken) gevolgd door de naam van het element en achter de naam een > (groter dan teken)</p> <p>De bijhorende eindtag is identiek behalve dat die start met </ of eindigt met >/.</p> <p>Tussen starttag en eindtag staat de tekst die hoort bij de naam van het element.</p> <p>Een element onder de root noemen we een child-element.</p> <p>De root heeft in ons voorbeeld 2 child-elementen.</p> <p>Beide child-elementen hebben op hun beurt weer 2 child-elementen.</p> <p>Je ziet heel duidelijk dat binnen ieder element opnieuw elementen zitten die op dezelfde manier zijn opgebouwd: een elementen-boom.</p>

XML regels

- Er moet een root zijn
- voor iedere starttag moet er een correcte eindtag zijn
- namen van elementen zijn hoofdlettergevoelig
- Elementen moeten correct genest worden:
 - <zln><deel>dit is fout</zln> hoor</deel>
 - <zln><deel>dit is juist</deel> hoor</zln>
- Namen beginnen met een letter, gevolgd door een letter, cijfer, ".", "_", "-" of ":"
- Namen mogen geen spaties bevatten en mogen niet starten met de 3 letters XML
- 5 speciale tekens : <, >, ', ", & mag je niet gebruiken in tekst.
In plaats daarvan moet je < , > , ' , " , & gebruiken
- Commentaar schrijf je als volgt: <!-- commentaar -->

Lezen van XML

Met het bestand H23_Data_klanten.xml:

H23_Voorbeeld_01_lees_klanten

```

001 import xml.etree.ElementTree as ET
002 xmldoc = ET.parse("H23_Data_klanten.xml")
003 root = xmldoc.getroot()
004 for child in root: print(child.tag)
005 for klant in root:
006     for info in klant :
007         print("Tag:", info.tag,":",info.text, end=" ")
008     print()
009     nr = klant.find("telefoon");print("Met find:",nr.text)
010     for nr in klant.findall("telefoon"):
011         print("Met findall:",nr.text,end=" ")
012     print()
013 for naam in root.iter("naam"):
014     print("Met iter:",naam.tag,":",naam.text,end=" ")

```

Geeft op het scherm:

```

klant
klant
Tag: naam : An Andriesen Tag: telefoon : 014/230998
Met find: 014/230998
Met findall: 014/230998
Tag: naam : Bart Bols Tag: telefoon : 016/238974 Tag: telefoon :
0477/182569
Met find: 016/238974
Met findall: 016/238974 Met findall: 0477/182569
Met iter: naam : An Andriesen Met iter: naam : Bart Bols

```

Lijn	Uitleg
001	Importeert de module xml.etree.ElementTree.
002	Leest het ganse bestand in en vorm de boom en zet die in xmldoc..
003	Haalt de top van de boom op met getroot.
004	Voor ieder child-element dat je dadelijk onder de root vindt: print de tag. Dus wordt er 2 keer klant geprint.
005	Voor ieder child-element dat je dadelijk onder de root vindt :
006	Haal alle child-element hiervan op (je zit dan al 2 niveaus diep)
007	print de tag en de bijhorende text
009	Zoek het eerste child-element van klant met tag telefoon.
010	Maak een lijst met alle child-elementen van klant met tag telefoon.
013	Zoek alle elementen onder de root (eender welk niveau) met tag "naam".

Elementen met attributen

H23_Data_klanten_met_attr.xml

```

001 <?xml version="1.0" encoding="utf-8"?>
002 <klanten>
003   <klant korting="ja" taal="NL">
004     <naam>An Andriesen</naam>
005     <telefoon>014/230998</telefoon>
006   </klant>
007   <klant korting="nee" taal="FR">
008     <naam>Bart Bols</naam>
009     <telefoon>016/238974</telefoon>
010     <telefoon>0477/182569</telefoon>
011   </klant>
012 </klanten>

```

In dit voorbeeld zie je dat er bij het element klant informatie is bijgekomen. Achter de klant tag staan "attributen" van het klant element. Deze attributen kan je ook ophalen.

H23_Voorbeeld_02_lees_klanten_met_attr

```

001 import xml.etree.ElementTree as ET
002 xmldoc = ET.parse("H23_Data_klanten_met_attr.xml")
003 root = xmldoc.getroot()
004 for klant in root:
005     attributen = klant.attrib
006     for key, value in attributen.items():
007         print(key, ":", value, end=" , ")
008     print()
009 for klant in root:
010     print(klant.find("naam").text, end=":")
011     print(klant.get('taal'))

```

Geeft op het scherm:

```

korting : ja , taal : NL ,
korting : nee , taal : FR ,
An Andriesen:NL
Bart Bols:FR

```

Lijn	Uitleg
004	Doorloopt alle "klant"-elementen.
005	Haalt alle attributen op. Attributen zitten in een dictionary.
006	Doorloopt alle attributen en zet ze samen met hun waarde op het scherm.
009	Doorloopt alle "klant"-elementen.
010	Zet de waarde van het "naam" element op het scherm.

011 Zet de waarde van het "taal" attribuut het scherm.

Advies : vermijdt attributen. Je kan dezelfde informatie ook plaatsen in elementen, waardoor je meer uniformiteit creëert.

Elementen geven bovendien veel meer mogelijkheden om aanpassingen en uitbreidingen te doen.

Maar je kan natuurlijk van andere bronnen informatie krijgen waarin attributen wel gebruikt worden.

Gegevens aanpassen

H23_Voorbeeld_03_update_klanten

```
001 import xml.etree.ElementTree as ET
002 xmldoc = ET.parse("H23_Data_klanten_met_attr.xml")
003 root = xmldoc.getroot()
004 for klant in root:
005     for naam in klant.iter("naam"):
006         naam.text = naam.text.upper()
007     klant.set("aangepast", "yes")
008     klant.set("korting", klant.get("korting").upper())
009 xmldoc.write('klanten_aangepast.xml')
```

Lijn	Uitleg
004	Doorloopt alle "klant"-elementen.
005	Doorloopt alle "naam" segmenten horende bij de klant
006	Verandert de tekst van die "naam".
007	Voegt een attribuut "aangepast" met waarde "yes" toe aan het "klant" element.
008	Verandert het attribuut "korting" van het "klant" segment.
009	Schrijft de ganse elementen-boom weg naar een bestand met de opgegeven naam.

Elementen maken en verwijderen

H23_Voorbeeld_04_maak_verwijder_element

```
001 import xml.etree.ElementTree as ET
002 xmlDoc = ET.parse("H23_Data_klanten_met_attr.xml")
003 root = xmlDoc.getroot()
004 klant = ET.Element("klant")
005 klant.set("korting", "nee"); klant.set("taal", "FR")
006 naam = ET.SubElement(klant, "naam")
007 naam.text = "Alain Bex"
008 telefoon = ET.SubElement(klant, "telefoon")
009 telefoon.text = "0477"
010 root.append(klant)
```

```

011 print(ET.tostring(root, encoding='unicode'))
012 for klant in root:
013     naam = klant.find("naam")
014     if "An" in naam.text :
015         root.remove(klant)
016 print(ET.tostring(root, encoding='unicode'))
017 ET.indent(root)
018 print(ET.tostring(root, encoding='unicode'))
019 xmlDoc.write("klanten_met_AB_zonder_An.xml", encoding="unicode")

```

Lijn	Uitleg
004	Maak een "klant" element. Voorlopig los.
005	Voegt attributen toe
006	Maakt onder klant een element "naam"
008	Maakt onder klant een element "telefoon"
010	Hang het "klant" element onder de root.
012	Doorloop alle "klant" - elementen
013	Neemt het eerste "naam" element onder "klant"
014	Als in tekst horende bij "naam" de string "An" staat
015	Verwijder dan het "klant"-element.
016	Print een string-voorstelling van de boom. Merk op dat die niet zo ordelijk is.
017	Maakt de boom ordelijk.
018	Dat zie je als je hem terug print.
019	Schrijft de ordelijke boom weg naar een bestand.

Een bestand nieuw aanmaken

H23_Voorbeeld_05_compleet_nieuw

```

001 import xml.etree.ElementTree as ET
002 root = ET.Element("klanten")
003 klant = ET.Element("klant")
004 klant.set("korting","nee");klant.set("taal","FR")
005 naam = ET.Element("naam");naam.text = "Hans Haverals"
006 telefoon = ET.Element("telefoon")
007 telefoon.text = "017/212656"
008 klant.append(naam);klant.append(telefoon)
009 root.append(klant)
010 xmlDoc = ET.ElementTree(root)
011 print(ET.tostring(root, encoding='unicode'))
012 ET.indent(root)
013 print(ET.tostring(root, encoding='unicode'))
014 xmlDoc.write('H23_Data_klanten_nieuw.xml', encoding='unicode')

```


Lijn	Uitleg
001	Maakt de root.
003	Maakt een "klant"-element met attributen.
004	
005	Maakt een "naam"-element.
006	Maakt een "telefoon"-element.
008	Hangt beide elementen aan het "klant"-element.
009	Hangt het "klant"-element aan de root.
010	Maakt het xml document met deze root.
011	Print een string-voorstelling van de boom. Merk op dat die niet zo ordelijk is.
012	Maakt de boom ordelijk.
013	Dat zie je als je hem terug print.
014	Schrijft de ordelijke boom weg naar een bestand.

XML en Office

H23_Voorbeeld_06_unzip_office

```

001 import zipfile
002 te_zippen_bestand = zipfile.ZipFile('H23_Data_Test PPTX.pptx', 'r')
003 te_zippen_bestand.extractall('temp')

```

Lijn	Uitleg
001	Importeert een module die het mogelijk maakt om met zip-bestanden te werken.
002	Maakt een "handle" naar een PowerPoint (in zip formaat)
003	Unzipt het PowerPoint bestand naar de opgegeven directory "temp". In deze directory zie je hoe alle informatie van de PowerPoint wordt opgeslaan in XML-bestanden. Dit is zo voor alle Office-bestanden.

Na dit hoofdstuk kan je:

- uitleggen hoe een XML-bestand er uit ziet
- een XML-bestand aanmaken, lezen, wijzigen en verwijderen
- uitleggen dat Office-bestanden eigenlijk zip-bestanden zijn en hoe die office zip-bestanden unzipt met Python.

24 REGEX

Inleiding

Een reguliere expressie (uit het Engels, regular expression, afgekort tot "regex", "regex" of RE) is een manier om patronen te beschrijven waardoor een computer softwarematig tekst kan herkennen. Er bestaat hiervoor een formele syntaxis, die deels is gestandaardiseerd.

Reguliere expressies worden bijvoorbeeld in teksteditors gebruikt om stukken tekst te doorzoeken of te veranderen; in andere programma's worden ze gebruikt om te controleren dat bepaalde patronen voorkomen. Veel programmeertalen ondersteunen reguliere expressies voor tekstmanipulatie.

Zonder al te diep in te gaan op "regex" zelf ga je in dit hoofdstuk zien hoe je op een gemakkelijk manier regex gebruikt in Python. En hoe kan het ook anders Python bevat een module die het je gemakkelijk maakt regex te gebruiken.

Dit hoofdstuk is vooral bedoeld om een introductie te geven. Regex op zich is uitgebreid genoeg om het onderwerp zijn van een aparte cursus. Maar heel regelmatig ga je regex tegenkomen in programma's.

De module re

Zonder al te diep in te gaan op "regex" zelf volgen hier een paar voorbeelden hoe je dit nuttig gereedschap kan gebruiken in Python.

regex is vooral nuttig als er een controle op geldigheid moet gebeuren om een string. Bijvoorbeeld een geldige maand, jaar, email, telefoonnummer, enz.. Het volstaat dan om aan regex een patroon aan te bieden waaraan die string moet voldoen.

Een reguliere expressie (regex) beschrijft strings zonder ze alle afzonderlijk op te noemen. De drie strings Handel, Händel en Haendel kan kunnen bijvoorbeeld beschreven worden met het patroon "H(a|ä|ae)ndel".

Denk ook aan de wildcards die je in verschillende operating systems tegenkomt.

H24_Voorbeeld_01_regex

```
001 import re
002 tekst = "Kaas, Boter, Choco, Gelei zijn soorten beleg"
003 zoek = re.search("e",tekst);print(zoek)
004 zoek = re.search("e[rln]",tekst);print(zoek)
005 zoek = re.findall("e[rln]",tekst);print(zoek)
006 email_regex = re.compile \
007 (r"([A-Za-z0-9]+[._-])*[A-Za-z0-9]+@[A-Za-z0-9-]+(\.[A-Z|a-z]{2,})+")
008 print(re.fullmatch(email_regex, "alainbex2@gmail.com"))
009 print(re.fullmatch(email_regex, "alainbex2@gmail.com"))
```

Geeft op het scherm:

```
<re.Match object; span=(21, 23), match='el'>
<re.Match object; span=(9, 11), match='er'>
['er', 'el', 'en', 'el']
None
<re.Match object; span=(0, 19), match='alainbex2@gmail.com'>
```

Lijn	Uitleg
001	Import de module re die je toelaat om met regex te werken.
002	Maakt de string waarin we gaan zoeken.
003	Zoekt de string "el" en geeft een Match object terug met de positie waarop het gevonden is.
004	Je ziet zoeken op " e[rln] ". Als er karakters staan tussen vierkante haakjes dan wil dat zeggen eender welke van die karakters. Zo zijn er verschillende symbolen in regex waarmee je de meest ingewikkelde zoekoperatie kan uitvoeren.
005	Zoek alle voorkomens van " e[rln] ". Je ziet in de output dat er 4 gevonden waren.
006	Hier zie je een veel voorkomende toepassing van regex. In vele toepassingen moet een email-adres ingegeven worden en dat moet een correct email adres zijn. Het regex patroon dat je op lijn 007 ziet is het patroon waar een email-adres aan moet voldoen. Zo kan je op het web talloze voorbeelden vinden van patronen voor allerlei data die moet gecontroleerd worden op geldigheid: telefoonnummers (uit verschillende landen), postcodes, etc.. Zonder hier zelf vele programmalijnen voor te moeten schrijven kan je het regex patroon overnemen.

Na dit hoofdstuk kan je:

- hoe je regex kan gebruiken in een Python programma
- uitleggen waar je regex voor kan gebruiken

25 WEB-SITES MAKEN MET PYTHON

Naast puur met tekst werken, naast een GUI die draait om de lokale computer zal je natuurlijk ook veel op het web werken en websites gebruiken.

Je kan een website maken met Python, maar je hoeft niet vanaf nul te beginnen.

Er zijn frameworks ontwikkeld die je toelaten om vrij snel een website te maken.

Een framework is een verzameling van kant en klare modules, data, settings, etc. die je toelaten om iets te maken zonder dat je de details van het framework hoeft te kennen. (denk aan het woord encapsulation dat je in het hoofdstuk over object orientatie bent tegengekomen)

Om websites te maken zijn er een aantal frameworks.

Google "python web framework"

Iedere framework heeft voor- en nadelen.

In het ene framework zitten honderden kant en klare functionaliteiten, maar is er een lange leercurve.

In het andere ben je dadelijk mee qua kennis, maar zijn er minder bruikbare methods beschikbaar.

Twee belangrijke frameworks (dd. 2022)

- Django
- Flask

Wil je een website maken, dan moet je je eerste de vraag stellen, wil ik die maken met Python. Onderzoek eerst wat de alternatieven zijn.

Een website leren maken met Django of Flask is een cursus op zich.

26 BESLUIT

Na het doorlopen van deze cursus heb je een basiskennis opgebouwd.

Niet alleen over Python, maar ook over programmeren.

Je zou nu in staat moeten zijn om eender welke taal snel onder de knie te krijgen. Begrippen zoals variabelen, datatypes, if-then, while- en forloops, functies, lijsten, bestanden, object oriëntatie komen immers in iedere programmeertaal terug.

In de laatste hoofdstukken van de cursus is het vooral de bedoeling om mee te geven wat de extra modules, die samen met Python beschikbaar worden gesteld, kunnen doen. Zodat je niet zelf gaat programmeren wat standaard beschikbaar is.

Natuurlijk zijn er nog vele modules die niet standaard beschikbaar zijn en waarbij je dus eerst die modules moet installeren op je computer vooraleer je in je programma een import kan doen. Voor iedere probleem dat je wil oplossen levert google "python <wat keywords van je probleem> " meestal talloze hits.

Ik hoop dat je na het volgen van deze cursus ook "goesting" gekregen hebt in het programmeren. Het is immers een enorm creatieve bezigheid, je kan er ontzettend nuttige en onnuttige dingen mee doen. Je start met "niets", je typt een hoop lijnen en je "runt" die. En als dat dan doet wat het moet doen : zalig!