Dictaat College
Python voor Natuur- en Sterrenkundigen

K. F. D. Rietveld

10 december 2015

`http://liacs.leidenuniv.nl/~rietveldkfd/courses/pmpy2015/`

De laatste maand van het vak Programmeermethoden voor studenten Natuurkunde en Sterrenkunde staat in het teken van de programmeertaal Python. Dit dictaat is een beknopte introductie tot Python aan de hand van hetgeen dat is behandeld in het college Programmeermethoden. De onderdelen van C++ die in Programmeermethoden zijn behandeld worden dus als bekend verondersteld. Naast het introduceren van de taal zelf zullen we ook oefenen in de context waarin Python in de Natuurkunde en Sterrenkunde wordt gebruikt: het doen van numeriek rekenwerk, dataverwerking en analyse, en het maken van mooie plots van hoge kwaliteit.

Een aantal secties in dit dictaat zijn gemarkeerd met het teken **(+)**. Deze secties zijn geavanceerde onderwerpen die niet uitgebreid in het hoorcollege zullen worden behandeld. Tevens is deze kennis niet nodig om de eindopdracht te kunnen volbrengen.

1 Wat is Python & waarom Python?

Python is een programmeertaal ontworpen door de Nederlander Guido van Rossum eind jaren '80 / begin jaren '90. In de laatste tien jaar heeft Python extreem aan populariteit gewonnen. Dit komt omdat het een simpele taal is, eenvoudig is in het gebruik en je complexe bewerkingen kunt opschrijven in enkele regels code. Daarnaast is de taal te gebruiken op alle gangbare besturingssystemen. Wat ook een grote rol speelt is het feit dat Python een zeer uitgebreide standaard bibliotheek heeft en dat het eenvoudig is om uitbreidingen (modules en *packages*) te schrijven.

Er zijn voor Python vele modules ontwikkeld voor het doen van numeriek rekenwerk en het genereren van plots van hoge kwaliteit. Door het grote gebruiksgemak van Python en de hoge kwaliteit van deze modules is Python zeer populair geworden in de Natuurkunde, Sterrenkunde, Biologie, enz.

In deze collegereeks zullen we kennismaken met Python en leren hoe simpele programma's te schrijven in Python. Er zal in het bijzonder ook aandacht zijn voor Python modules die in de natuur- en sterrenkunde worden gebruikt: numeriek rekenwerk met *NumPy* en plotten met *matplotlib*.

Je zult zien dat Python een simpele taal is om te leren gebruiken en dat beheersing van Python je veel tijd zal schelen in de toekomst! Python is een ultiem gereedschap om snel een programma te schrijven voor zaken die je anders met de hand zou doen. In plaats van gefrustreerd te zoeken naar functionaliteiten in Excel, zul je voortaan Python scripts schrijven.

2 De Python Interpreter

Bij het programmeren in C++ moesten we gebruik maken van een "compiler" om de source code om te zetten in een uitvoerbaar bestand. We zeggen dat C++ een "gecompileerde taal" is.

```
        gedit programma.cc
        g++ -Wall -o programma programma.cc
        ./programma
```

Python is een "geïnterpreteerde taal" Er hoeft niet apart een compiler te worden gedraaid. In plaats daarvan starten we de interpreter op met de vraag een specifiek programma uit te voeren. De interpreter zal het gewenste programma direct uitvoeren. Python wordt ook wel een *scripttaal* genoemd, net als bijvoorbeeld Perl, Ruby en PHP. Python programma's worden daarom ook vaak "Python scripts" of "scripts" genoemd. De meest gebruikte extensie is `.py`:

```
        gedit programma.py
        python programma.py
```

Een groot voordeel van een "geïnterpreteerde taal" is dat je je programma sneller kunt testen, je hoeft immers niet steeds een "compilatieslag" te doen. Een nadeel is dat sommige fouten in het programma alleen worden ontdekt als dat deel van het programma ook daadwerkelijk wordt uitgevoerd. Er is in dit geval geen compiler die van te voren de gehele code verwerkt. Wel kan dit (tot op zekere hoogte) worden tegengegaan met tools als *pylint* en *pyflakes*.

De interpreter is ook op te starten zonder een programma te specificeren. In dat geval start Python op in een interactieve modus er wordt een *prompt* (>>>) gepresenteerd:

```
Python 2.7.3 (default, Jun 22 2015, 19:33:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "Hello World"
Hello World
>>> a = 3
>>> b = 4
>>> print "%d + %d = %d" % (a, b, a + b)
3 + 4 = 7
>>>
```

Zoals in het voorbeeld is te zien kan er achter de prompt direct Python code worden geschreven. De ingegeven code wordt na een druk op de Return toets meteen uitgevoerd. De interactieve modus leent zich goed voor het uitproberen van kleine programmafragmenten en voor het krijgen van hulp. We zullen hier later op terugkomen. Je kunt de interactieve modus afsluiten met het statement `exit()`.

In het dictaat zullen we veel gebruik maken van de interactieve modus om functionaliteiten van Python uit te leggen. Als je zelf deze voorbeelden wilt uitproberen neem je alleen de code **na** de prompt over. Alles wat je achter de prompt typt, kun je ook gebruiken in een Python programma dat je in een editor schrijft.

3 Python installeren

Python is volledig open source en is gratis te verkrijgen. Als je een Linux of Mac machine gebruikt is Python met een aantal basis modules al standaard geïnstalleerd. We zullen in dit college ook gebruik maken van NumPy, SciPy, matplotlib en iPython. Als je zelf je Linux machine beheert zijn deze uitbreidingen misschien nog niet geïnstalleerd. Met de package manager (bijv. "apt-get" of "yum") kan je de extra benodigde pakketten erbij zetten. Voorbeeld voor een Ubuntu systeem:

```
sudo apt-get install python-numpy python-scipy python-matplotlib ipython
```

Mac OS 10.9 (Mavericks) en hoger komen standaard met NumPy, SciPy en matplotlib, maar niet iPython. Je kunt zonder problemen dit college volgen en de eindopdracht maken zonder iPython

te installeren. Als je toch met iPython wilt experimenteren kun je het installeren via bijvoorbeeld MacPorts (`http://www.macports.org/`) of door de Anaconda distributie te installeren (`https://www.continuum.io/download`, kies voor de Python 2.7 versie).

Op Windows-systemen moet je zelf Python downloaden en installeren. De kale interpreter is te vinden op `http://www.python.org/`. Echter, het is sterk aan te raden om een Python distributie te installeren waarin NumPy en matplotlib al zijn inbegrepen. Voorbeelden van dergelijke distributies zijn Enthought (`https://store.enthought.com/downloads/`) en Python(x, y) (`http://python-xy.github.io/`).

4 Een eerste Python programma

Laten we eens kennis maken met Python door Python-versies te bekijken van de eerste C++-programma's die zijn geïntroduceerd in het college Programmeermethoden.

```
print "Dit komt op het scherm."
exit(0)
```

En een tweede programma:

```
# dit is een simpel programma
getal = 42  # een variabele declareren en initialiseren
print "Geef een geheel getal ..",
getal = int(raw_input())
print "Kwadraat is:", getal * getal
exit(0)
```

En een derde programma:

```
# Dit is een regel met commentaar ...
import math  # voor de "pi" constante
print "Geef straal, daarna Enter ..",
straal = float(raw_input())
if straal > 0:
    print "Oppervlakte:",
    print math.pi * straal * straal
else:
    print "Niet zo negatief ..."
print "Einde van dit programma."
exit(0)
```

Een aantal karakteristieken vallen direct op:

- Commentaarregels beginnen met `#` en niet met `//`.
- Net als in C++ zijn er in Python *keywords* gedefinieerd, bijvoorbeeld: `import`, `print`, `if`, `else`.
- Regels worden niet beëindigd met een puntkomma.
- Er hoeft niet per se een *main* functie te worden gedeclareerd. Alle regels die niet in een functie staan, worden altijd uitgevoerd.
- Het `print` statement voegt standaard een newline toe (merk op dat `cout` dat niet doet). Als je aan het einde van het print statement een komma plaatst, dan wordt de newline **niet** toegevoegd.

- Er wordt geen gebruik gemaakt van accolades om functies of if-blokken te maken. In plaats van het gebruik van accolades of haakjes om blokken te maken, moet er in Python correct worden ingesprongen. Opeenvolgende regels die met hetzelfde aantal spaties (of tabs) zijn ingesprongen vormen samen een blok. In het voorbeeld is te zien dat het if-blok en else-blok op een dergelijke manier zijn gevormd. Merk ook op dat het aantal spaties in het if-blok toeneemt ten opzichte van het if-statement. We komen later terug op de exacte regels omtrent het inspringen.

5 Variabelen in Python

In C++ moesten variabelen vooraf worden gedeclareerd als een bepaald type en kan dit type niet meer veranderen. In Python is dit niet nodig. Variabelen kunnen worden gemaakt met een toekenningsstatement (*assignment statement*):

```
>>> a = 4
>>> b = "testje!"
>>> a = "overschrijven"    # oude waarde van variabele a wordt overschreven
```

Als een toekenning wordt toegepast op een naam die nog niet bestaat, dan wordt die naam automatisch aangemaakt. Bestaat de naam al wel? Dan wordt de oude waarde overschreven. In het geval de naam van een variabele wordt gebruikt in een expressie (bijvoorbeeld aan de rechterkant van een toekenningsstatement), dan moet de naam wel bestaan. Als dat niet zo is, volgt een foutmelding.

```
>>> a, b = 3, 4    # je kunt ook meerdere variabelen tegelijk toekennen
>>> c = a + b
>>> d = a + g      # Fout! De variabele g bestaat niet.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'g' is not defined
```

De foutmelding die we hierboven krijgen is een `NameError`. Python vertelt ons dat de variabele `g` niet bestaat. Met de *Traceback* geeft Python ook aan waar in het programma de fout is opgetreden. In dit geval is dat *stdin*, de standard input, omdat we de programmaregel in de interactieve modus hebben ingevoerd. Als we een programma in de editor schrijven en daarna draaien, dan zal Python de bestandsnaam en het precieze regelnummer aangeven waar de fout optreedt, bijvoorbeeld:

```
Traceback (most recent call last):
  File "programma.py", line 3, in <module>
    d = a + g
NameError: name 'g' is not defined
```

Elke variabele in Python heeft een type. Een type wordt in Python niet vooraf vastgelegd in een declaratie. In plaats daarvan wordt het type bepaald aan de hand van het soort data dat aan de naam wordt toegekend. Met de functie `type` kunnen we zien dat variabelen verschillende typen hebben:

```
>>> a, b, c = 9, 3.14, "strrrr"
>>> type(a)
<type 'int'>
>>> type(b)
<type 'float'>
>>> type(c)
<type 'str'>
```

```
>>> a = "strrr2"      # oude waarde van a wordt overschreven
>>> type(a)
<type 'str'>
```

Merk ook op dat wanneer we een andere waarde (en ook een ander soort waarde) aan a toekennen, dat de oude waarde van a wordt overschreven en ook het type van a verandert.

6 Getallen in Python

We kwamen in C++ verschillende typen tegen voor het opslaan van getallen: `int`, `long`, `bool`, `float`, `double`. Python kent ook verschillende typen getallen:

- Integers (type `int`), net als in C++ in de meeste gevallen 4 bytes groot. Hierdoor is het bereik -2^{31} tot $2^{31} - 1$, ofwel van ongeveer -2 miljard tot 2 miljard zoals we ook in de slides van Programmeermethoden zagen.
- Long integers (type `long`). Deze getallen hebben een bereik dat alleen maar wordt gelimiteerd door de hoeveelheid geheugen beschikbaar in de computer. Er kunnen hiermee dus **zeer** grote getallen worden opgeslagen.
- Boolean waarden (type `bool`). Variabelen van dit type kunnen de waarde `True` of `False` aannemen.
- Floating-point getallen (type `float`) om benaderingen van reële getallen op te slaan. In Python zijn floating-point getallen altijd double precision, dit komt overeen met `double` in C++.
- Complexe getallen (type `complex`). Python heeft ingebouwde ondersteuning voor complexe getallen! Voorbeeld:

```
>>> z = 6+9j    # "j" is de imaginaire eenheid, in de wiskunde i geheten
>>> type(z)
<type 'complex'>
>>> z.real
6.0
>>> z.imag
9.0
# een complex getal baseren op bestaande variabelen moet via de constructor
>>> a, b = 3, 4
>>> z = complex(a, b)
>>> z
(3+4j)
```

Op getallen kunnen we uiteraard allerlei operaties toepassen. Laten we weer eens kijken hoe de voorbeelden uit Programmeermethoden zich in Python verhouden:

```
a, b = 3, -5
getal = a + b    # getal wordt -2
a = a + 7        # a wordt met 7 opgehoogd naar 10
b += 1           # hoog b met 1 op, LET OP: Python kent geen ++ operator
a -= 1
getal += a
a = 19 / 7       # Integer deling: a wordt 2
b = 19 % 7       # Rest bij deling (modulo): b wordt 5
q = (6+9j) + (4+2j)  # Optelling complexe getallen: resultaat is (10+11j)
q = (6+9j) * 2       # Resultaat: (12+18j)
```

Let op dat een deling met gehele getallen altijd weer resulteert in een geheel getal. Als je een reëel getal als antwoord wilt toestaan, dan moet je zorgen dat één van de operanden een reëel getal is.

```
i = 9 / 5          # Geeft 1, i wordt een integer
x = 9 / 5.0        # Geeft 1.8, x wordt een float
x = float(9 / 5)   # Geeft 1.0, 9 / 5 geeft een integer resultaat dat wordt
                   # omgezet naar een float.
x = 9 / float(5)   # Geeft 1.8, x wordt een float
x = 9.0 // 5.0     # Geeft 1.0, // is delen met integer-afronding
m = 3 ** 4         # Python heeft een ingebouwde operator voor
                   # machtsverheffing, het resultaat is 81
```

Hierboven zien we dat `float()` wordt gebruikt om een getal van het type `float` te maken. `float()` is in feite een functie die kan worden gebruikt om getallen of zelfs strings om te zetten naar een float. Ook `float("3.14")` is correct Python en resulteert in een float type met de waarde 3.14. We moeten dus voorzichtig zijn om een dergelijke conversiefunctie te vergelijken met een C++ cast, een C++ cast kan bijvoorbeeld niet zomaar een string naar een float omzetten!

Ook voor de andere numerieke typen zijn er ingebouwde conversiefuncties: `int()`, `long()`, `complex()`. Variabelen kunnen ook worden omgezet naar een string met de conversiefunctie `str()`:

```
>>> a = 3.1412345567
>>> a
3.1412345567       # Een floating-point waarde
>>> str(a)
'3.1412345567'     # Een string
>>> type(str(a))
<type 'str'>       # zeker weten ...
```

Wat gebeurt er nu als er een operatie wordt gespecificeerd op twee verschillende typen, bijvoorbeeld `3 + 6.31`? In zo'n geval zal er impliciet een typeconversie plaatsvinden. In dit specifieke voorbeeld zal 3 worden geconverteerd naar een float type. De vakliteratuur noemt dit *coercion*. Voor numerieke typen wordt in het algemeen het type met het kleinere bereik geconverteerd. Bijvoorbeeld bij een optelling tussen een int en een long, zal de int worden geconverteerd naar een long. Soms is een conversie niet vanzelf mogelijk, bijvoorbeeld als je een int en een string bij elkaar probeert op te tellen. Zulke gevallen leiden tot een foutmelding (een `TypeError`).

7 Het print statement

Als we in de Python interpreter alleen de naam van een variabele intypen en op Enter drukken, krijgen we netjes de waarde van die variabele te zien (althans, als die variabele inderdaad bestaat). Hoe kunnen we nu in een Python programma geschreven in een editor variabelen afdrukken? Hier kunnen we het `print` statement voor gebruiken, bijvoorbeeld:

```
a = 110
print a
```

Een print statement in Python bestaat in feite uit het keyword `print` gevolgd door een lijst van expressies. Na het evalueren van de expressies worden de eindresultaten geconverteerd naar strings (zie hierboven hoe getallen naar strings werden geconverteerd) en vervolgens naar de terminal geschreven. Tussen de uitvoeren van de verschillende expressies worden spaties gevoegd.

```
>>> a = 110
>>> b = 12
>>> print "Test:", "a =", a, "b =", b, "en samen maakt dat", a + b
Test: a = 110 b = 12 en samen maakt dat 122
```

In het college Programmeermethoden is ook besproken hoe getallen, met name floating-point getallen, netjes kunnen worden afgedrukt. Met hulp van "iomanip" konden we instellen hoe breed de uitvoer moest zijn en hoeveel cijfers na de komma er moesten volgen. Ook in Python is dit mogelijk als we gebruik maken van uitvoerformattering ("output formatting"). Python kent twee manieren van uitvoerformattering, hier wordt vaak naar gerefereerd als oude stijl en nieuwe stijl. Omdat beide in Python codes voorkomen, zullen we beide manieren kort behandelen.

7.1 Oude stijl (+)

De oude stijl van uitvoerformattering is erg vergelijkbaar met `printf` in C en C++. Wie bekend is met de %-notatie van `printf` voelt zich hierin direct thuis. Een voorbeeld:

```
>>> a = 123
>>> b = 62
>>> c = 3.1409134091023
>>> print "%d|%10d|%8.4f|" % (a, b, c)
123|        62|  3.1409|
```

Merk op dat de uitvoerformattering een combinatie is van een format string, het %-karakter en een reeks van argumenten voor de format string. Als de laatste twee worden weggelaten verdwijnt de speciale betekenis:

```
>>> print "%d|%10d|%8.4f|"
%d|%10d|%8.4f|
```

Laten we de format string nu bestuderen. Een %-teken geeft het begin aan van een specificatie van een conversie. De conversiespecificatie eindigt met een karakter. De eerste conversie is `%d`. Dit betekent dat het eerste argument wordt genomen, dus de variabele `a`. Dit argument wordt dan geformatteerd als een decimaal getal, aangeduid door het karakter "d". In de uitvoer wordt de conversie `%d` vervangen met het resultaat van de formattering, dus `123`.

De tweede conversie, `%10d` formatteert het tweede argument, `b`. Ook in dit geval wordt de uitvoer een decimaal getal, maar het getal 10 voorafgaand aan de "d" geeft aan dat het veld 10 karakters breed moet worden. Dit zien we terug in de uitvoer.

De derde conversie formatteert een floating-point getal. De conversie geeft aan dat het veld 8 karakters breed moet zijn en dat er 4 decimalen achter de komma moeten worden afgedrukt.

Dit zijn de meest gebruikte formatteringsmogelijkheden. Uiteraard zijn er vele malen meer mogelijkheden. Een volledig overzicht van de formatteringsmogelijkheden is te vinden op `https://docs.python.org/2.7/library/stdtypes.html#string-formatting`.

Tenslotte, het resultaat van de formattering kan ook in een string worden opgeslagen:

```
>>> test = "%10d %8.4f" % (a, b)
>>> print test
'       123  62.0000'
```

7.2 Nieuwe stijl

De nieuwe stijl van uitvoerformattering is conceptueel hetzelfde: er is weer sprake van een format string en een reeks van argumenten. De manier van het specificeren van conversies is echter anders, in plaats van %-notatie wordt er gebruik gemaakt van accolades. Een conversiespecificatie wordt in de documentatie een *format field* genoemd. Ook is de `%`-operator vervangen met een aanroep van `.format`. Laten we een eerste voorbeeld bekijken:

```
>>> a = 123
>>> b = 62
>>> c = 3.1409134091023
>>> print "{0} {1}".format(a, b)
123 62
>>> print "{0} {2}".format(a, b, c)
123 3.1409134091
```

Het eerste getal tussen de accolades geeft het hoeveelste element aan uit de reeks van argumenten dat moet worden afdrukt. In plaats van getallen, mag je hier ook gebruik maken van namen:

```
>>> print "{een} {twee}".format(een=a, twee=b)
123 62
```

Als we nu een dubbele punt toevoegen, dan kunnen we na de dubbele punt weer de breedte van het veld en ook een type aangeven:

```
>>> print "{0:6d} {0:6f} {1:8.4f} {2:20s}::".format(a, c, "testje")
   123 123.000000   3.1409 testje              ::
```

En ook in dit geval kunnen we de formattering opslaan in een string:

```
>>> test = "{0:6d} {0:6f} {1:8.4f}".format(a, c)
>>> print test
   123 123.000000   3.1409
```

Binnen deze formatteringstaal is er nog veel meer mogelijk, zoals links en rechts uitlijnen. Zie de documentatie: `https://docs.python.org/2/library/string.html#formatspec`.

8 Strings

In het college Programmeermethoden hebben we gezien dat er in C++ twee manieren zijn om te werken met strings: ouderwetse C-strings en met een string-klasse. Een ouderwetse C-string is een rijtje met `char`'s. In Python is er geen apart `char` type[1]. Strings worden altijd opgeslagen in een type `str`. Elke string is eigenlijk een object van type `str`[2] en net als in C++ kunnen er op dit object operaties worden uitgevoerd, zoals het bepalen van de lengte en het vergelijken van strings. Laten we eens een kijkje nemen:

```
>>> woord = "De."
>>> type(woord)
<type 'str'>
>>> len(woord)
3
>>> woord == "test"   # Vergelijken van strings kan gewoon met de == operator
False
>>> woord == "De."    # Merk op dat True en False met een hoofdletter worden geschreven
True
```

In een ouderwetse C-string kunnen we met een index een individueel array element, dus een karakter, uitlezen. In Python kunnen we objecten van het type `str` ook "indexen". Dit kan met een enkele index, maar ook door zowel een start als eind index op te geven zodat we substrings kunnen maken (dit noemen we "slicing"). In tegenstelling tot C en C++ zijn zelfs negatieve

[1] Wel kan je ASCII waarden opslaan in een `int`. Met de functie `chr()` kunnen deze worden omgezet naar een string, bijvoorbeeld `chr(83)` geeft `'S'`.

[2] In feite is alles in Python een object, zelfs `int`s en `float`s.

indexen toegestaan! Deze tellen terug vanaf het einde van de string. Uiteraard beginnen we nog steeds met 0 te tellen. Daarnaast is het zo dat je met indexing de string **alleen** kunt lezen! Het is niet mogelijk om de string te veranderen. Als je de string wilt aanpassen, maak je een nieuwe string waarin de aanpassing zit verwerkt, zie ook de + operator hieronder.

```
>>> s = "een lange test string"
>>> s[2]
'n'
>>> s[-4]
'r'
>>> s[3:8] # Merk op dat de eind-index niet meedoet: '8' wordt niet meegenomen.
' lang'
>>> s[6:]  # Einde reeks weggelaten, we gaan dan impliciet door tot het einde
'nge test string'
```

Andere operaties op strings zijn bijvoorbeeld kijken of de string een gegeven substring bevat (`in` operator), kijken of een string eindigt met een bepaalde string (`endswidth` methode) of strings aan elkaar plakken (+ operator).

```
>>> s = "aaa bbb ccc eee fff ggg"
>>> "aaa" in s
True
>>> "zzz" in s
False
>>> f = "testbestand.txt"
>>> f.endswith(".txt")
True
>>> s + f
'aaa bbb ccc eee fff gggtestbestand.txt'
>>> s + 12                          # Dit gaat niet goed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> s + str(12)                     # Wel als we expliciet converteren
'aaa bbb ccc eee fff ggg12'
```

Om strings aan elkaar te plakken gebruiken we de +-operator: merk op dat deze operator in dit geval niet optelt, maar aan elkaar plakt! Wat een operator dus precies doet, is afhankelijk van typen objecten waar deze op wordt toegepast. Bij getallen tellen we op, bij strings plakken we aan elkaar. Zoals we eerder al zagen, kunnen we de +-operator niet toepassen op een string en een getal. Bij de *-operator ligt dit anders: gegeven twee getallen dan wordt een vermenigvuldiging uitgevoerd, gegeven een string en een getal n, dan wordt dezelfde string n-maal herhaald. En we kunnen natuurlijk moeilijkere expressies bouwen door de operatoren te combineren:

```
>>> a = "aaa"
>>> b = "bbb"
>>> a * 3
'aaaaaaaaa'
>>> (a + b) * 3
'aaabbbaaabbbaaabbb'
>>> a * 3 + b * 3
'aaaaaaaaabbbbbbbbb'
```

9 Lijsten

In het college Programmeermethoden zijn arrays geïntroduceerd als een geordend rijtje van variabelen van hetzelfde type. Wat in Python in vele gevallen als een array wordt gebruikt is een

list. Een list is een geordende lijst van variabelen. In feite is een lijst een container van objecten, er wordt ook wel over gesproken als zijnde een *compound data type* of *sequence type*. De typen van de variabelen hoeven niet hetzelfde te zijn. Daarnaast staat de grootte van de lijst niet vast, je kan eenvoudig elementen toevoegen en verwijderen aan de lijst. Lijsten kunnen worden aangemaakt met blokhaken.

```
a = [1, 2, 3, 4, 5]
b = [1.0, 2.5, 3.4]
c = [1, "test", 4.5, False]     # Verschillende typen variabelen
```

Indexing en slicing zoals we hebben gezien bij strings, werken ook op lijsten.

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7]
>>> len(a)
8
>>> a[6]
6
>>> a[2:5]
[2, 3, 4]
>>> a[3:]           # begin of eind index mogen worden weggelaten.
[3, 4, 5, 6, 7]
>>> a[:6]
[0, 1, 2, 3, 4, 5]
>>> a[4] = 'ha!'    # Element van de lijst overschrijven
>>> a
[0, 1, 2, 3, 'ha!', 5, 6, 7]
```

We komen later nog op lijsten terug.

10 Controlestructuren

In deze sectie bekijken we de belangrijkste controlestructuren in Python: `if` voor het maken van keuzes, `for` voor een vast aantal herhalingen en `while` voor een onbekend aantal herhalingen.

10.1 If-statement

We beginnen met het `if`-statement, dit heeft in Python de volgende algemene vorm:

```
if test > 7:
    a = 13
    iets = "test is waar"
elif test < 7:          # in plaats van "else if" schrijven we "elif"
    a = 10
    iets = "we kwamen langs else if"
else:
    a = 4
    iets = "test is dus 7"
```

De algemene vorm verschilt maar weinig van C++. Toch vallen er een aantal dingen op:

- Anders dan C++ is het niet verplicht om haakjes rond de test te zetten. Als de testen uitgebreider worden is dit soms wel aan te raden, zie ook hieronder.
- Na de test volgt een dubbele punt, dit duidt op de komst van een ingesprongen blok.
- Let er op dat we een blok (of samengesteld statement) maken door in te springen en dus niet door accolades te plaatsen. Inspringen is **altijd** verplicht. Zorg ervoor dat alle statements in een blok met hetzelfde aantal spaties inspringen.

Predicaten zoals je bent gewend in C++ werken gewoon: ==, !=, <, >=, enzovoort. Wel opletten bij het maken van uitgebreidere Booleaanse expressies: in plaats van !, && en || gebruiken we not, and en or. Als je zowel and als or in een expressie gebruikt is het sterk aan te raden om haakjes te gebruiken, zodat je zeker weet dat de juiste expressies worden samengenomen. Een aantal voorbeelden:

```
if y >= 3 and y <= 7:  ...
if not (y < 3 or y > 7):   ...
if y >= 3 and (x == 4 or x == 5):  ...
if s == "hello": ...         # je kunt zonder problemen strings vergelijken
if len(s) == 5: ...
# Voor de leesbaarheid mag een if-statement meerdere regels beslaan,
# gebruik dan een backslash: \
if y >= 3 and (x == 4 or x == 5) and \
   z == 12 and (q >= 10 or q <= -10):  ...
```

Ook in Python wordt "short-circuiting" toegepast. In een test (x != 0 and y / x == 7) wordt de tweede test niet gedaan als x gelijk is aan 0.

10.2 For-statement

Het for-statement is in Python een stuk eenvoudiger dan in C++. In plaats van de soms ingewikkelde manipulatie van een iteratievariabele in C++, wordt er in Python simpelweg een iteratie van een lijst uitgedrukt. Na het for-keyword volgt de naam van de iteratievariabele. Deze iteratievariabele zal in elke iteratie van de loop opeenvolgend de verschillende waarden van de lijst aannemen.

```
for karakter in ['a', 'b', 'c', 'd', 'e']:
    print karakter,
# drukt af: a b c d e
for i in [1, 2, 3, 4, 5]:
    print i
```

Vaak willen we dat een variabele, bijvoorbeeld i, opeenvolgend de verschillende waarden uit een getallenreeks aanneemt. Voor lange getallenreeksen is het met de hand intypen van de reeks natuurlijk niet te doen. In Python hebben we daarvoor de functie range(), waarmee getallenreeksen kunnen worden gemaakt. Het resultaat van deze functie is weer een lijst, zodat het for statement weer gewoon een lijst afloopt. Het meeste simpele gebruik van range() geeft ons een lijst van getallen vanaf 0. Je kunt ook een ander startpunt opgegeven, of zelfs een stap-waarde: range(start, stop, step). **Let op:** het einde van de reeks is open, dus de gegeven stop-waarde zal geen deel uitmaken van de teruggegeven reeks.

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
>>> range(3,6)            # begin bij waarde 3
[3, 4, 5]
>>> range(0, 50, 5)       # maak steeds een stap van 5
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> range(20, 50, 5)
[20, 25, 30, 35, 40, 45]
>>> for i in range(10):   # itereer nu over een lijst gemaakt met range
...     print i,
...
0 1 2 3 4 5 6 7 8 9
```

In het voorbeeld zien we ook een verandering van de prompt. Omdat er na de dubbelepunt van het for-statement nog een ingesprongen blok moet volgen, verandert Python de prompt in ....

Je moet hier zelf inspringen (bijvoorbeeld met 4 spaties), voordat je de code schrijft. Als je klaar bent met het blok voer je een lege regel in (meteen op Return drukken) en de prompt verandert terug naar >>>. We weten nu genoeg om het voorbeeld uit het college Programmeermethoden te herschrijven in Python:

```
for ( i = 3; i <= 17; i = i + 2) cout << i << "-";

for i in range(3, 17+1, 2):
    print i, "-",
```

Merk nu ook het volgende subtiele verschil op: in C++ heeft de variabele `i` na afloop de waarde `19`. In Python is dit niet zo! Ga zelf na waarom dit het geval is.

Uiteraard kunnen we ook in Python dubbele `for`-lussen schrijven (loops nesten). Laten we nogmaals een voorbeeld uit Programmeermethoden herschrijven:

```
for i in range(1, 5+1):
    print "{0}:".format(i),
    for j in range(1, i+1):
        print i * j,
    print                    # print zonder argument geeft een newline
```

10.3 While-statement

Een `while`-statement in Python bestaat uit een testexpressie en een loop body. Anders dan het `for`-statement wordt er geen lijst afgelopen. Je zult in vele gevallen dus zelf een iteratorvariabele moeten bijhouden, bijvoorbeeld:

```
i = 1
n = 10
while i <= n:
    print i, "--", i * i
    i += 1
```

In tegenstelling tot C++ kent Python geen `do-while` statement.

11 Inspringregels

We hebben al gezien dat correct inspringen in Python een must is. Wanneer niet correct wordt ingesprongen zal een programma worden geweigerd door de interpreter (met een "Indentation-Error", een inspringfout), of erger nog: het programma doet iets anders dan je zou verwachten/willen.

Wanneer moet er worden ingesprongen? Houd je in achterhoofd dat je moet inspringen om blokken van statements te vormen. Dit gebeurt bijvoorbeeld bij `if`-statements, loops en het definiëren van functies zoals we later zullen zien. Merk op dat we in C++ in deze gevallen bijna altijd accolades moeten plaatsen! In C++ worden blokken van statements gemaakt door deze in accolades te plaatsen.

Binnen eenzelfde blok **moet** er in elke regel op dezelfde manier worden ingesprongen. Spring je voor de eerste regel van het blok in met 4 spaties, dan moeten alle volgende regels in dat blok ook worden ingesprongen met 4 spaties. Je mag overigens inspringen met zowel spaties als tabs. Echter, het is een goed gebruik om altijd in te springen met 4 spaties en om nooit tabs te gebruiken bij het schrijven van Python code. Tip: stel je editor in om te werken met een inspringing van 4 spaties. Vele editors kunnen ook automatisch voor je inspringen.

Het grote voordeel van het inspringen is dat bijvoorbeeld het "dangling else"-probleem wordt vermeden. Wellicht herinner je je nog het volgende voorbeeld uit het college Programmeermethoden:

```
if ( x > 0 )
  if ( y > 0 )
    cout << "Beide groter dan nul.";
  else                    // waar hoort deze bij?
    cout << " x positief, y negatief (of 0)  ";
```

Het is nu niet direct duidelijk waar de "else" bijhoort. Daarom is het in C++ sterk aan te raden om in dit soort gevallen gebruik te maken van accolades om verwarring te vermijden. In het college Programmeermethoden is hierbij opgemerkt: "Zorg ervoor dat de layout klopt – de compiler kijkt daar niet naar.". Het mooie is dat in het geval van Python de interpreter **juist wel** naar de layout kijkt. Er kan dus geen verwarring zijn: de layout (het inspringniveau) is leidend.

Tenslotte nog een opmerking over lege blokken. In C++ is het mogelijk om een leeg statement blok te maken met een accolade openen en sluiten, zonder statements ertussen. Hoe kunnen we dit nu voor elkaar krijgen met inspringen? Python heeft hier een speciaal keyword voor: `pass`. `pass` is een statement dat niets doet, eigenlijk een soort tijdelijke opvulling (een "placeholder"). Het wordt vaak gebruikt bij code die nog niet af is, bijvoorbeeld een `if`-statement, loop of functie die later nog zal worden ingevuld.

```
x = 10
if x > 0:         # ONGELDIG Python! Er volgt geen ingesprongen statement!
    # niets
print "test"
if x > 0:
    pass          # gebruik van pass, dit is prima
print "test"
```

12 Meer over lijsten

Tot nu toe hebben we alleen maar lijsten bekeken met een vaste lengte en hebben we geen elementen toegevoegd of verwijderd. Uiteraard is er met het type *list* veel meer mogelijk. Voorheen hebben we lijsten direct met data geïnitialiseerd. Het is ook mogelijk om te beginnen met een lege lijst. Let op dat je een lege lijst niet zomaar kunt indexeren! Een lege lijst bevat geen elementen, dus indexering zal leiden tot een error die je vertelt dat het element achter de gegeven index niet bestaat (een "IndexError"). Dit is te vergelijken met het benaderen van array-elementen buiten de gedeclareerde grootte in C++, maar een C++ compiler klaagt daar meestal niet over, de Python interpreter wel.

Je kunt waarden toevoegen aan een lijst met de methoden *append* of *insert*: *append* zet de waarde achteraan in de lijst, bij *insert* geef je een index op waarvoor het nieuwe element moet worden geplaatst. Elementen verwijderen kan met de *remove* methode of het *del* statement. Bij *remove* moet je de waarde opgeven van het object dat je wilt verwijderen. Wanneer je het *del* statement gebruikt, geef je de index van het element aan (een slice opgeven mag ook!). Tenslotte is er ook een *pop* methode, welke je kan gebruiken als je de lijst als een stapel ("stack") gebruikt. Standaard haalt *pop* het laatste object uit de lijst en geeft deze terug als returnwaarde. Ook is het mogelijk om een index te geven als argument.

```
>>> a = []                   # Een lege lijst.
>>> a = list()               # Hetzelfde als de regel hierboven.
>>> a[4] = "test!"           # FOUT: element met index 4 bestaat nog niet!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
>>> a.append("een")
>>> a.append("twee")
>>> a.append("drie")
```

```
>>> a.insert(0, "nul")
>>> a
['nul', 'een', 'twee', 'drie']
>>> a.remove("een")           # Verwijder element met waarde "een"
>>> a.pop()                   # Geef en verwijder laatste element
'drie'
>>> a
['nul', 'twee']
>>> a.pop()
'twee'
>>> b = range(5)
>>> b.pop(2)                  # Geef en verwijder element op index 2
2
>>> del b[1]                  # Verwijder element op index 1
>>> del b[2]                  # LET OP: dit was b[3]!
>>> b
[0, 3]
```

Eerder maakten we al kennis met "slicing". Het is nu tijd om de formele definitie te bekijken:

$$start : eind : stap$$

waar $start$ de begin index is, $eind$ de eind index, welke zoals we hebben gezien open is, en $stap$ is de stapgrootte. In feite worden alle indices i gekozen waarvoor geldt dat $i \leq start < eind$ en $(i - start) \mod stap = 0$. Het opgeven van $stap$ is optioneel en ook $start$ en $eind$ mogen worden weggelaten. Wat gebeurt er als zowel $start$ als $eind$ niet worden gespecificeerd? In dat geval wordt de gehele lijst gekozen. Ook handig is dat je toekenningen mag doen aan een slice, de waarde die wordt toegekend moet dan wel weer een lijst zijn (een enkele waarde, een *scalar*, mag niet). De aangegeven slice wordt dan vervangen door de nieuwe lijst. De lengte van de slice en nieuwe lijst hoeven **niet** overeen te komen.

```
>>> a = range(10)        # Lijst 0 t/m 9
>>> a[2:5]
[2, 3, 4]
>>> a[2:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> a[:5]
[0, 1, 2, 3, 4]
>>> a[2:8:2]
[2, 4, 6]
>>> a[::2]               # Gehele lijst, stapgrootte 2
[0, 2, 4, 6, 8]
>>> a[::3]
[0, 3, 6, 9]
>>> a = range(10)
>>> a[0:5] = ['a', 'b', 'c', 'd', 'e']    # Vervang eerste deel van de lijst
>>> a
['a', 'b', 'c', 'd', 'e', 5, 6, 7, 8, 9]
>>> a[5:5] = ['x', 'y', 'z']              # Toevoeging in het midden
>>> a
['a', 'b', 'c', 'd', 'e', 'x', 'y', 'z', 5, 6, 7, 8, 9]
>>> a[10:] = range(100, 110)
>>> a
['a', 'b', 'c', 'd', 'e', 'x', 'y', 'z', 5, 6, 100, 101, 102, 103, 104, 105, 106, 107,
108, 109]
>>> a[0:10] = []                 # Verwijder eerste 10 elementen
>>> a
[100, 101, 102, 103, 104, 105, 106, 107, 108, 109]
```

```
>>> a[:] = []                    # Leeg de gehele lijst
>>> a
[]
```

Er zijn ook faciliteiten om in lijsten te zoeken. Één daarvan is de operator `in`. Deze operator gaat na of een gegeven object in de container zit, resulterend in een Boolean waarde (wel/niet aanwezig in de container). Let op dat `in` in de context van een expressie een andere betekenis heeft dan wanneer het wordt gebruikt in een `for` loop. In het geval je de index van een bepaald element wilt weten, kan je gebruik maken van de methode `index`[3]. Het tellen hoe vaak een element voorkomt kan met `count`.

```
>>> a = ['een', 'lijst', 'met', 'een', 'aantal', 'woorden']
>>> 'lijst' in a
True
>>> 'blabla' in a
False
>>> a.index('aantal')
4
>>> a.count('een')
2
```

Een andere belangrijke techniek is het "nesten" van lijsten: we maken dan een lijst van lijsten. Dit is mogelijk omdat een lijst allerlei typen objecten kan bevatten, dus ook lijsten. Het is dan ook mogelijk om over meerdere niveaus te indexeren (`a[i][j][k]`). Maar let op! Geneste lijsten zijn **geen** multi-dimensionele arrays. De geneste lijsten mogen namelijk elke gewenste grootte aannemen. We komen later nog terug op het gebruik van echte arrays.

```
>>> a = [[1, 2, 3, 4, 5], ['a', 'b', 'c'], [], ['x']]
>>> for lijst in a:
...     print len(lijst),
...
5 3 0 1
>>> a[0][1]
2
>>> a[1][2]
'c'
>>> a[2][4]            # Op a[2] zit een lege lijst!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> b = [[1, 2, 3], 591243, ['a', 'b', 'c']]
>>> b[1][3]           # Op b[1] zit een getal, geen lijst!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object has no attribute '__getitem__'
```

Wanneer we lijsten combineren met `for`-loops kunnen we al een aantal simpele algoritmen implementeren. Hieronder geven we voorbeelden van een sommatie van een lijst van integers en het bepalen van het grootste element in een lijst van integers.

```
# Bereken som met array indexing
som = 0
for i in range(len(a)):
    som += a[i]
```

[3]In het geval een element meerdere keren voorkomt, geeft `index` je de index van het eerste element dat wordt gevonden.

```
# Dit is een meer "Python-esque" manier:
som = 0
for element in a:
    som += element

# Bepalen grootste element
grootste = 0
for element in a:
    if element > grootste:
        grootste = element
print "Het grootste element is: ", grootste
```

Alhoewel Python voor veel van dit soort problemen een ingebouwde functie heeft, is het implementeren van dergelijke algoritmen altijd een goede oefening. Bovenstaande kan ook worden bereikt met `sum(a)` en `max(a)`.

13 Tuples

Omdat tuples vaak voorkomen in Python code besteden we er kort aandacht aan. Een *tuple* is een geordende reeks van objecten. Anders dan lijsten kunnen tuples niet worden aangepast, eenmaal gemaakt dan is de tuple vastgeklonken. Tuples worden vaak gebruikt om objecten die aan elkaar zijn gerelateerd samen op te slaan. Bijvoorbeeld een coördinatenpaar!

We zullen zo zien dat we tuples kunnen gebruiken om in functies meerdere waarden terug te geven. Later zullen we zien dat we een lijst niet als key kunnen gebruiken in een dictionary, maar een tuple wel. Om coördinaatparen als dictionary keys te gebruiken, maken we gebruik van tuples. Net als op lijsten werken indexing, slicing en `in` ook op tuples.

Je maakt een tuple door een reeks van objecten te geven, gescheiden door komma's. Vaak moet deze reeks tussen haakjes staan, maar dat is niet altijd verplicht. Om verwarring te voorkomen kun je simpelweg altijd haakjes plaatsen als je twijfelt.

```
>>> a = (1, 2, 3, 4, 5, 6, 7, 's', 'a', 'b')
>>> a[4]
5
>>> a[4] = 100                    # Tuples kunnen niet worden aangepast
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> a[4:8]
(5, 6, 7, 's')
>>> 's' in a
True
>>> b = (a, 4, ('q', 'z'), 6)      # Nesting is geen probleem
>>> b
((1, 2, 3, 4, 5, 6, 7, 's', 'a', 'b'), 4, ('q', 'z'), 6)
```

14 Functies

Zodra programma's groter worden, wordt het belangrijk om de code goed te structureren en het dupliceren van code te vermijden. Dit doen we door gebruik te maken van functies. Net als in C++ hebben functies in Python een naam, argumenten (of parameters) en een resultaat ("return value"). Functies mogen doen wat je wilt: iets uitrekenen of juist een bepaalde taak verrichten. Zoals we hebben gezien worden in Python variabelen niet vooraf gedeclareerd met een bepaald type. Bij functie argumenten en het resultaat is dat net zo, bij het definiëren van de functie geven

we alleen de namen van argumenten op en niet de types. Het definiëren van een functie ziet er globaal als volgt uit:

```
def functienaam(arg1, arg2, ..., argn):
    blok van statements (met inspringing!)
```

Na de dubbele punt volgt, net als bij *for*-loops, een blok van statements die op dezelfde manier zijn ingesprongen. Het eerste statement dat niet meer is ingesprongen maakt geen deel meer uit van de functie, dus daar is de definitie van de functie afgelopen. Een functie zonder argumenten is overigens ook toegestaan. Het teruggeven van een resultaat ("return value") gaat met het keyword **return**. Een functie heeft altijd een resultaat, als **return** wordt weggelaten is het resultaat de waarde `None` (van het type "NoneType"). Als je `return` gebruikt zonder expressie er achter, dan wordt standaard ook `None` als resultaat gebruikt. `None` is geen waarde en evalueert naar False in een Boolean expressie. We bekijken nu een aantal functiedefinities:

```
def hallo():
    print "hello world"

def telop(a, b):
    c = a + b
    return c

def sommeer(lijst):
    som = 0
    for l in lijst:
        som += l
    return som

def paar(a, b, c):
    # We gebruiken bij return haakjes, maar dat is niet verplicht!
    return (a, a + b, a + b + c)

# en zo roepen we de functies aan
hallo()
resultaat = telop(10, 41)
de_som = sommeer([4, 45, 5, 23, 4])
t = paar(1, 2, 3)            # t wordt een tuple
x, y, z = paar(1, 2, 3)      # we kunnen ook direct de elementen van de
                             # tuple in aparte variabelen zetten
```

In het voorbeeld `sommeer` zien we dat het geen enkel probleem is om een lijst als argument te hebben. Ook dictionaries (zie later) en tuples kun je zonder problemen als functieargument gebruiken. Wel is belangrijk dat de argumenten geen type hebben en we geen compiler hebben die nakijkt of de argumenten in functie-aanroepen van het juiste type zijn. Bijvoorbeeld `telop(10, "hallo")` gaat fout!

Het voorbeeld `paar` demonstreert dat je tuples als returnwaarde kunt gebruiken. In de aanroep kun je ervoor kiezen om het resultaat ook op te vangen in een tuple (1 variabele) of direct de tuple uit te pakken in aparte variabelen.

Verdieping: call-by-reference, call-by-value, of iets anders? **(+)**

In C++ moesten we call-by-reference gebruiken om 2 of meer resultaatwaarden te hebben. Hoe zit dat eigenlijk met call-by-value en call-by-reference in Python? Eigenlijk is geen van beide een goede beschrijving van wat er in Python gebeurt. Om dit te begrijpen is het van belang in je achterhoofd te houden dat alles in Python een object is. Variabelen in Python zijn eigenlijk niets meer dan namen voor een object of links tussen een naam en een object. Het

is dus mogelijk dat één object meerdere namen heeft. Dit is precies wat we zien gebeuren bij het aanroepen van functies.

```
def telop(a, b):      # 'a' verbonden met zelfde object als 'x',
                      # en 'b' met 'y'
    tmp = a + b
    a = 10            # naam 'a' wordt nu gebonden aan een ander object,
    return tmp

x = 125
y = 23
resultaat = telop(x, y)
print x   # x is hier nog steeds 125
```

In de functie in dit voorbeeld zijn `a`, `b` en `tmp` lokale variabelen. Zoals we in het college Programmeermethoden hebben geleerd zijn `a` en `b` formele parameters. Als startwaarde krijgen deze formele parameters een link met hetzelfde object als de corresponderende actuele parameter (de actuele parameters in dit geval zijn `x` en `y`). De scope van de variabelen `a`, `b` en `tmp` is beperkt tot de functie `telop`. **Let op!** dit geldt alleen voor de variabelen, de namen, en **niet** voor de objecten waar deze mee zijn gekoppeld. We bekijken nu een iets ingewikkelder voorbeeld:

```
def voegtoe(lijst, x):    # Formele parameter 'lijst' wijst nu naar
                          # hetzelfde object (dezelfde lijst) als 'reeks'!
    lijst.append(x)       # Dus we voegen toe aan 'reeks'

    lijst = list()        # Hier wordt de naam 'lijst' gekoppeld aan een
                          # nieuwe, lege lijst. Maar 'reeks' verandert
                          # dus niet!

reeks = [1, 2, 3]
voegtoe(reeks, 10)
print reeks
# Resultaat:  1, 2, 3, 10
```

Als we een lijst meegeven als parameter, dan zal de formele parameter in eerste instantie wijzen naar diezelfde lijst. Dit lijkt op call-by-reference. Operaties op die lijst worden dus op dezelfde lijst uitgevoerd als de lijst waar de actuele parameter naar wijst. Als we nu echter een andere lijst toekennen aan de naam `lijst`, dan gebeurt er iets bijzonders. De naam `lijst` wordt nu gebonden aan de nieuwe lijst. Dit verandert echter helemaal niets aan de lijst waar `lijst` voorheen naar wees. Deze lijst blijft gewoon bestaan en `reeks` wijst er nog naar. Merk op dat wanneer er sprake zou zijn van call-by-reference, dan zou het veranderen van `lijst` in de functie ook `reeks` moeten veranderen, maar dat is dus niet zo. We bekijken nu nog een voorbeeld met een string object om in te zien dat er inderdaad geen sprake is van call-by-reference.

```
def verleng(s):
    s = s + "toevoeging"    # 's' wordt nu gebonden aan een nieuw object!

a = "hello world"
verleng(a)
print a                     # 'a' is dus niet veranderd!
```

In de functie wordt de naam `s` gebonden aan een nieuw object door het toekenningsstatement. We moeten dus erg oppassen bij het gebruik van strings! Bovenstaande code kan

worden gecorrigeerd door de nieuwe s als returnwaarde op te geven (return s) en de returnwaarde van verleng toe te kennen aan a: a = verleng(a).

We zien nu dus in dat Python niet call-by-value is, immers zelfs voor integers wordt niet de integer-waarde (de echte value) doorgeven aan de formele parameter maar een link naar een object dat die integer-waarde bevat. Python is ook niet call-by-reference want als we in een functie de referentie naar een object veranderen, verandert de actuele parameter niet mee. Een vaste naam voor het gedrag dat we in Python zien is er niet, maar als mogelijke omschrijving wordt bijvoorbeeld "call-by-object-reference" geopperd[a].

Als we willen en daar noodzaak voor is kunnen we call-by-reference wel emuleren door gebruik te maken van een lijst. In de functie passen we dan een waarde in het lijst-object aan en zoals we hebben gezien wijst de actuele parameter ook naar de lijst die hierdoor is veranderd.

```
def emulatie(l):
    l[0] += "toevoeging"
    l[1] = 64

a = ["hello world", 103]
emulatie(a)
print a[0], a[1]
# Resultaat: 'hello worldtoevoeging' 64
```

[a] Zie http://www.jeffknupp.com/blog/2012/11/13/is-python-callbyvalue-or-callbyreference-neither/.

Python functies hebben nog een aantal leuke mogelijkheden om het gebruik van functies flexibeler te maken. Zo is het bij het definiëren van de functie mogelijk om standaardwaarden ("default values") te specificeren. Als een parameter niet expliciet als actuele parameter wordt opgegeven, dan krijgt de corresponderende formele parameter de standaardwaarde. Deze parameter wordt dus optioneel om op te geven als actuele parameter.

Daarnaast is het bij het aanroepen van de functie ook mogelijk om de naam van de formele parameter te gebruiken om een waarde aan toe te kennen. Dit is handig als je functies hebt met veel parameters en veel standaardwaarden omdat je dan niet alle parameters expliciet hoeft op te geven. De actuele parameters nemen dan de vorm aan naam=waarde en worden "keyword arguments" genoemd. Keyword arguments moeten altijd aan het einde van de reeks van actuele parameters worden geplaatst.

```
# Deze functie heeft een groot aantal standaardwaarden
def teken_lijn(p1, p2, kleur="zwart", dikte=1.0, pijl=None, stippel=False):
    # hier wordt de lijn getekend
    pass

# Gebruik alle standaardwaarden
teken_lijn( (10, 10), (100, 10) )
# Gebruik alleen de laatste twee standaardwaarden
teken_lijn( (10, 10), (100, 10), "rood", "10.0")
# Specificeer zelf alle parameters
teken_lijn( (10, 10), (100, 10), "rood", "10.0", "Gevuld", True)
# Gebruik van keyword arguments
teken_lijn( (10, 10), (100, 10), stippel=True)
```

Wanneer je veel functies gaat schrijven en je programma groter wordt, wordt het des te belangrijk om de code goed te documenteren. Zo is het een goed gebruik om voor elke functie goed uit te leggen wat deze precies doet. Je kan altijd netjes commentaar boven aan de functie zetten. Ook kan je gebruik maken van een mooie functionaliteit die Python ingebouwd heeft: "documentation strings". Hiermee koppel je de documentatie van je functie aan de functie zelf en kan je in de interactieve modus met de functie help de documentatie opvragen. Ook kent

Python verschillende tools voor het automatisch genereren van documentatie die gebruik maken van deze “docstrings”.

```
# Deze functie heeft een groot aantal standaardwaarden
def teken_lijn(p1, p2, kleur="zwart", dikte=1.0, pijl=None, stippel=False):
    '''Deze functie tekent een lijn van p1 naar p2 met de attributen
    kleur, dikte, pijl en stippel.
    '''

    # hier wordt de lijn getekend
    pass

# Hiermee kan je in de interpreter hulp krijgen over je functie
help(teken_lijn)
```

15 Globale structuur Python programma

Een eenvoudig Python programma bestaat uit een enkel `.py`-bestand. Dit bestand bevat de code die moet worden uitgevoerd door de interpreter en zal regel voor regel worden uitgevoerd. Code hoeft niet verplicht in een functie te staan. Code die niet in een functie staat zal altijd regel voor regel door de interpreter worden uitgevoerd. Code die wel in een functie staat wordt alleen uitgevoerd als die functie ook daadwerkelijk wordt aangeroepen. Net als in C++ kan een functie alleen worden aangeroepen **nadat** deze is gedefinieerd. Python kent geen prototypes, dus je zal altijd functie definities boven aan je programma moeten zetten.

Omdat het binnen Python niet verplicht is om alle code binnen een functie te plaatsen, kent Python ook geen main-functie. Het is natuurlijk wel netjes om een main-functie te maken en deze dan aan te roepen vanuit de globale code. Een nette manier om een Python programma te structuren is als volgt:

```
# Eerst alle import statements
import sys

# Dan alle hulpfuncties
def hulpfunctie(a):
    print "Hello world, a=", a

# De main-functie
def main():
    q = 10354
    hulpfunctie(q)

    return 0

# En tenslotte de "globale" code die main aanroept. Waarom we een dergelijk
# if-statement gebruiken zullen we later in het dictaat zien.
if __name__ == "__main__":
    # Let op: omdat we hier niet binnen een functie zitten, mag er geen
    # return worden gebruikt.
    sys.exit(main())
```

Het is een goed gebruik om alle `import`-statements bovenaan het programma te zetten. Zodra we met NumPy aan de slag gaan, zullen we altijd een `import`-statement voor NumPy nodig hebben. Plaats daarna alle functies die je zelf hebt geschreven in het bestand. Vervolgens de main-functie. En tenslotte een stukje globale code om je main-functie aan te roepen. We maken hierbij gebruik van een `if`-statement dat de waarde van `__name__` vergelijkt, waarom dit zo is heeft te maken met het schrijven van modules en daar komen we later op terug.

16 Dictionaries (+)

Lijsten kun je alleen indexeren met gehele getallen. Soms zou het goed uitkomen als je in plaats van een geheel getal een ander soort object kunt gebruiken om een container te indexeren. Relevante voorbeelden zijn bijvoorbeeld indexeren op strings of coördinaatparen. In Python is dit mogelijk met een zogenaamde *dictionary*. In feite wordt er in een dictionary een afbeelding opgeslagen van een object naar een ander object. Deze objecten worden vaak de *key* en de *value* genoemd: gegeven de key kun je toegang krijgen tot de bijbehorende value. Een dictionary is dus eigenlijk een collectie van *key-value paren*.

In andere programmeertalen wordt een dergelijke datastructuur vaak een associatieve array of hash table genoemd. Dictionaries worden door middel van een hash table geïmplementeerd. Wat er gebeurt is dat er voor de key een bepaalde hash-waarde wordt berekend. Equivalente objecten moeten afbeelden op dezelfde hash-waarde. Met deze hash-waarde, een geheel getal, kunnen we dus weer een lijst indexeren en zo krijgen we toegang tot de gezochte value[4].

Een ander belangrijk verschil ten opzichte van lijsten is dat het bij dictionaries **wel** is toegestaan om een toekenning te doen aan een nog niet-bestaande index (key). In dat geval zal er automatisch een nieuw key-value paar aan de dictionary worden toegevoegd. Let ook op het feit dat dictionaries **niet**-geordend zijn, de key-value paren worden ongeordend opgeslagen. Het gebruik van de dictionary laat zich weer het beste demonstreren middels een aantal voorbeelden.

```
>>> d = dict()
>>> d["walter"] = "071-5270000"
>>> d["kris"] = "06-12345678"
>>> d["joop"] = "0123-524513"
# Value ophalen uit de dictionary aan de hand van een key
>>> d["kris"]
'06-12345678'
# Waar je met [] een lege lijst maakt, maak je met {} een lege dictionary
>>> k = {}
>>> k[4,3] = "rood"           # We maken hier gebruik van een tuple!
>>> k[1,2] = "blauw"
>>> k[9,4] = "zwart"
>>> len(k)     # Hoeveel paren in deze dictionary?
3
>>> k          # Je kunt de gehele dictionary ook printen
{(1, 2): 'blauw', (9, 4): 'zwart', (4, 3): 'rood'}
>>> k[5,2]     # Deze key zit niet in de dictionary
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: (5, 2)
>>> k[1,2]     # Deze wel
'blauw'
```

Afsluitend bespreken we nog een aantal veel gebruikte operatoren en methoden van dictionaries. Met de *in* operator en de *has_key* methode is het mogelijk om te kijken of een gegeven key al in de dictionary aanwezig is. Verwijderen uit de dictionary kan met het *del* statement door het te verwijderen paar voluit te specificeren.

De methode *get* haalt de waarde op voor het paar met de gegeven key (voorbeeld: `d.get("joop")`). Een veelvoorkomend scenario bij dictionaries is het opslaan van gehele getallen die we steeds willen ophogen. Een goed voorbeeld is het maken van een histogram. We zouden dan eerst moeten nagaan of een gegeven key al bestaat, zo niet de key toevoegen, en dan pas kunnen we de waarde ophogen. Dat kan makkelijker! We kunnen in de *get* methode ook een

[4]Het komt natuurlijk voor dat verschillende objecten dezelfde hash-waarde hebben. In dat geval wordt er na de hash-indexering nog door een lijst gelopen van keys en de bijbehorende value, zodat de correcte value wordt gevonden. Bespreking van de exacte implementatiedetails van hash tables maakt geen deel uit van de inhoud van dit college.

"default"-waarde opgegeven die moet worden teruggeven in het geval de key nog niet bestaat. Hiermee kunnen we het uitschrijven van een *if*-statement vermijden.

```
>>> "walter" in d
True
>>> "test" in d
False
>>> d.has_key("kris")
True
>>> del d["kris"]
>>> d
{'joop': '0123-524513', 'walter': '071-5270000'}
>>> t = {}
>>> t["B"] += 1                 # Probeer element met 1 op te hogen
                                # Maar deze key bestaat nog niet!
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'B'
>>> t["B"] = t.get("B", 0) + 1  # Bestaat nog niet, dus gebruik defaultwaarde 0
>>> t["B"] = t.get("B", 0) + 1  # Nu wel de waarde van t["B"]
>>> t["B"]                      # Het resultaat is dus 2
2
```

17 Iteratietechnieken

Voornamelijk bij dataverwerking en -analyse speelt het bewandelen van datastructuren een belangrijke rol. We bekijken nu een aantal veelgebruikte iteratiepatronen om te leren hoe we op de makkelijkste manier iteraties kunnen uitdrukken. Zoals we zullen zien kan dit soms op een veel eenvoudigere en elegantere manier dan stug vast te houden aan de `for` loop in combinatie met de `range` functie[5].

Eerder in dit dictaat hebben we gezien dat `for` loops altijd een lijst afgelopen. Als we geen lijst bij de hand hebben, maar een loop willen over een subset van de gehele getallen, dan kunnen we daarvoor de `range` functie gebruiken. Stel nu dat we een lijst willen aflopen en in de loop body ook de index van elk element van de lijst nodig hebben. Twee naïeve manieren om dit te doen zijn:

```
for i in range(len(lijst)):
    print i, "-", lijst[i]
```

```
i = 0
for l in lijst:
    print i, "-", l
    i += 1
```

Een veel makkelijkere manier is om gebruik te maken van de functie `enumerate`. Deze maakt automatisch paren (tuples!) aan van index en element:

```
for i, l in enumerate(lijst):
    print i, "-", l
```

In dit voorbeeld zien we ook meteen een ander mooi iteratiepatroon: het aflopen van een lijst van tuples. Voor de netheid laten we vaak de haakjes om de tuple weg in het `for`-statement.

```
lijst = [(1, 'a'), (2, 'b'), (3, 'c')]   # een lijst van tuples
for getal, letter in lijst:
    print getal, ",", letter
# En dit is equivalent:
```

[5]Een belangrijke vuistregel in Python is de volgende: als je een loop schrijft met behulp van `range`, dan is de kans zeer groot dat er een makkelijkere, elegantere en/of snellere manier bestaat om hetzelfde te bereiken.

```
for (getal, letter) in lijst:
    print getal, ",", letter
```

Wil je een lijst afgelopen in omgekeerde volgorde of gesorteerd? Geen probleem! Daar zijn `reversed` en `sorted` voor.

```
lijst = [4, 13, 2, 8, 11, 5]
for l in reversed(lijst):
    print l,
# Geeft: 5 11 8 2 13 4
for l in sorted(lijst):
    print l,
# Geeft: 2 4 5 8 11 13
for l in reversed(sorted(lijst)):
    print l,
# Geeft: 13 11 8 5 4 2
```

(+) Bij het analyseren van data willen we ook vaak dictionaries aflopen. We kunnen loops maken over de lijst van alle keys van een dictionary of over alle key-value pairs (dit zijn weer tuples). Om dit te doen kunnen we gebruik maken van de methoden `keys`, `values` en `items` van de dictionary.

```
# Net als bij lijsten is er ook een verkorte manier om dictionaries te maken,
# we geven dan simpelweg alle key-value paren op.
voorraad = { "peren": 2, "appels": 8, "tomaten": 0, "witte bonen": 101 }
for k in voorraad.keys():
    print k,
# Geeft: tomaten peren witte bonen appels
# (Merk op: een dictionary is ongeordend!)
for k in sorted(voorraad.keys()):
    print k,
# Geeft: appels peren tomaten witte bonen (lexicografisch/alfabetisch gesorteerd)
for v in voorraad.values():
    print v,
# Geeft: 0 2 101 8
for k, v in voorraad.items():
    print "Er zijn {0} stuks {1}.".format(v, k)
# Geeft:
# Er zijn 0 stuks tomaten.
# Er zijn 2 stuks peren.
# Er zijn 101 stuks witte bonen.
# Er zijn 8 stuks appels.
```

18 Inlezen en wegschrijven van bestanden

Tot nu toe hebben we alleen maar uitvoer gestuurd naar het beeldscherm (of eigenlijk de terminal) met behulp van `print`. Vooral bij het doen van dataverwerking is het zeer belangrijk om data te kunnen lezen uit bestanden en naar andere bestanden te kunnen wegschrijven. Vaak heb je, bijvoorbeeld, de resultaten van een experiment in een bepaald bestand staan. Je wilt dan in je Python programma de inhoud van dit bestand inlezen, daar een aantal berekeningen mee uitvoeren en vervolgens de resultaten weer naar een nieuw bestand wegschrijven.

In C++ maakten we gebruik van de `ifstream` en `ofstream` klassen. In Python hebben we een object van het type `file`. Met de functie `open` kunnen we een bestand openen en een `file` object aanmaken. Deze functie heeft twee parameters: als eerste de bestandsnaam en als tweede de "modus" waarin de file moet worden geopend. Als je wilt lezen gebruik je als modus `"r"`, als

je wilt schrijven "w". Een aanroep aan de functie open resulteert in een file object, welke vervolgens methodes heeft waarmee we uit het geopende bestand kunnen lezen of naar het bestand data kunnen wegschrijven. Afsluitend moet het bestand worden gesloten, dit doen we met de methode close. Het volgende programma opent het bestand "test.txt" en schrijft de inhoud van dit bestand naar het beeldscherm.

```
f = open("test.txt", "r")       # "r", want we gaan lezen
line = f.readline()
while line != "":               # Een lege string duidt EOF aan
    print line,                 # We willen geen extra newline van print
    line = f.readline()
f.close()
```

Bovenstaand voorbeeld lijkt erg op de manier waarop we in C++ files inlezen. We kunnen het voorbeeld echter op een nog mooiere, meer Python-achtige, manier herschrijven:

```
f = open("test.txt", "r")
for line in f:
    print line,
f.close()
```

In dit geval let Python voor ons op de end-of-file. Deze constructie zorgt ervoor dat f een lijst van regels van het bestand oplevert, welke we dan één voor één aflopen. We gebruikten in C++ de methode get om een enkel karakter te lezen. We kunnen iets soortgelijks in Python doen met f.read(1), welke precies 1 byte (1 karakter) uit een bestand leest en het resultaat teruggeeft als string (Python heeft geen apart type voor het opslaan van losse karakters). Echter, in principe gaat het inlezen van data in Python altijd regel per regel. Vervolgens kun je uit elke regel specifieke data halen ("parsen"). Stel we hebben een bestand dat op elke regel 3 gehele getallen heeft gescheiden door spaties. Dan kunnen we dat op de volgende manier inlezen:

```
f = open("getallen.txt", "r")
for line in f:
    line = line.rstrip("\n")          # haal de regelovergang eraf
    a, b, c = line.split(" ")         # splits de string op spatie-karakter
    a, b, c = int(a), int(b), int(c)  # maak overal integers van
    som = a + b + c
    print "Som:", som
f.close()
```

Oplossing voor arbitrair aantal getallen **(+)**

Ook het bovenstaande programma kan op een mooiere manier worden geschreven. We slaan dan de getallen op in een lijst en op die manier kunnen we regels met verschillende aantallen getallen verwerken. Vervolgens gebruiken we de functie map om de conversiefunctie int op elk element van de lijst toe te passen en de functie sum om de lijst te sommeren.

```
f = open("getallen.txt", "r")
for line in f:
    line = line.rstrip("\n")
    getallen = line.split(" ")
    getallen = map(int, getallen)
    som = sum(getallen)
    # Of alles in 1 regel:
    # som = sum(map(int, line.split(" ")))
    print "Som:", som
f.close()
```

Schrijven naar bestanden kan met de methode `write` op het file-object en ook door het `print` statement te instrueren om de uitvoer naar een file-object te sturen in plaats van naar de terminal. Belangrijk: de `write` methode accepteert alleen strings, dus andere objecten dien je eerst zelf naar een string te converteren. Bijvoorbeeld `f.write(str(42))` om het getal 42 naar een bestand te schrijven. Bij het `print` statement gebruiken we de notatie >> om een file-object aan te duiden waar de uitvoer naartoe moet in plaats van de terminal.

```
f = open("uitvoer.txt", "w")
f.write("hello world\n")         # write voegt geen regelovergang toe
f.write(str(43) + "\n")
print >>f, "Met print is het eenvoudiger"
print >>f, "Geheel getal: {0}  Floating point: {1}.".format(51, 3.1412345)
f.close()
```

Aangezien een file-object gewoon een object is, kan je zonder problemen het object meegeven als parameter aan een functie. We kunnen dan functies schrijven om bijvoorbeeld nette tabellen naar een bestand te schrijven:

```
def maak_kopjes(f):
    print >>f, "{0:>4s} | {1:>4s} | {2:>4s}".format("a", "b", "c")
    print >>f, "-" * 5 + "|" + "-" * 6 + "|" + "-" * 5

def mooi_formatteren(f, a, b, c):
    print >>f, "{0:4d} | {1:4d} | {2:4d}".format(a, b, c)

f = open("tabel.txt", "w")
maak_kopjes(f)
mooi_formatteren(f, 12, 54, 50)
mooi_formatteren(f, 54, 34, 41)
mooi_formatteren(f, 6, 59, 35)
f.close()

# Deze code schrijft het volgende naar het bestand:
#    a |    b |    c
#-----|------|-----
#   12 |   54 |   50
#   54 |   34 |   41
#    6 |   59 |   35
```

19 Modules en packages

We beschikken nu over voldoende basisvaardigheden met Python om met uitbreidingen aan de slag te gaan. Tot nu toe hebben we alleen maar programma's geschreven die bestonden uit een enkel bestand. Voor grotere programma's is dit natuurlijk niet handig en zouden we graag de mogelijkheid willen hebben om onze code te modulariseren en over meerdere bestanden te

verspreiden. In C++ konden we dit doen door meerdere `.cc`-bestanden te maken en deze gezamenlijk te compileren. We maakten een “header”-file waarin de prototypes van de functies werde opgenomen die we vanuit andere bestanden wilden aanroepen. Het is in Python mogelijk om meerdere `.py`-bestanden te maken. Zoals we hebben gezien kun je in Python alleen maar een functie aanroepen **nadat** deze is gedefinieerd. Hoe zit dat dan met functies uit andere bestanden, welke we niet expliciet definiëren in het bestand waarin we de functie willen aanroepen? We gebruiken hiervoor het `import`-statement. Met `import` kunnen we een soort definities maken van functies die zich in andere bestanden bevinden.

Stel we hebben een bestand `handig.py` met daarin de functies `hallo`, `telop` en `vermenigvuldig`. We noemen `handig.py` een *module*. Als we vanuit ons `programma.py` modules willen aanroepen moeten we deze eerst importeren. We kunnen een gehele module importeren, maar ook een specifieke functie uit een module:

```
# importeer de gehele module, let op we laten ".py" weg!
import handig

handig.hallo()
c = handig.telop(a, b)
c = handig.vermenigvuldig(a, b)
```

```
# importeer een specifieke functie uit een module
from handig import telop

# We hoeven nu niet de prefix "handig." te gebruiken
c = telop(a, b)
```

```
# importeer de gehele module, maar onder een afgekorte naam
import handig as h

h.hallo()
c = h.telop(a, b)
```

Met de functie `dir()` kunnen we bekijken wat er allemaal in een module zit. Dit is erg handig vanuit de interactieve interpreter. Uiteraard kun je met `help()` hulp krijgen over de module.

```
>>> import handig
>>> dir(handig)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'hallo', 'telop',
'vermenigvuldig']
>>> help(handig)
...
>>> help(handig.hallo)
...
```

Hele grote Python-uitbreidingen bestaan natuurlijk vaak uit meerdere modules. Die modules worden dan samen in één directory gezet en we noemen het dan een *package*. We hoeven deze losse modules in de meeste gevallen niet allemaal apart te importeren, de package is zo ingericht dat bij het importeren van de package automatisch meerdere modules worden geïmporteerd. Voor ons is het belangrijkste om te weten dat we met `import` ook packages importeren, bijvoorbeeld: `import numpy as np`.

We hebben nu gezien hoe we gebruik kunnen maken van modules. Hoe kunnen we nu onze eigen modules maken? Dit is helemaal niet moeilijk: je maakt gewoon een `.py` bestand en daarin definieer je een aantal functies. Je mag ook variabelen definieren en importeren, handig voor fysische constanten! Let wel goed op het volgende: **(1)** de zelfgeschreven module die je wilt importeren moet in dezelfde directory staan als het programma[6], **(2)** de bestandsnaam mag geen

[6]Natuurlijk kan dit flexibeler: daarvoor moet je de module globaal installeren of het zoekpad aanpassen, we zullen dat in dit dictaat niet behandelen.

streepjes ("-") bevatten. De module `handig.py` zoals we hierboven hebben gebruikt ziet er als volgt uit:

```
def hallo():
    print "hello world"

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

def vermenigvuldig(a, b):
    return a * b
```

Wanneer `handig.py` wordt geïmporteerd voert de interpreter het hele bestand uit. Als het bestand alleen maar bestaat uit functiedefinities, dan worden alleen maar functies gedefinieerd. Als er globale code tussen de functiedefinities staat, dan zal deze code worden uitgevoerd. Stel er zou onderaan `handig.py` code staan (als een soort main functie), dan wordt deze ook bij het importeren van de module uitgevoerd. Vaak willen we dit niet! Om dit te voorkomen maken we gebruik van `if __name__ == '__main__':` zoals we eerder zagen. Deze if-conditie is alleen waar voor het programma dat het hoofdprogramma is en dus niet wordt geïmporteerd. Op deze manier kun je ervoor zorgen dat bepaalde globale code (zoals een aanroep van een main functie) alleen maar voor het hoofdprogramma wordt uitgevoerd.

20 NumPy

NumPy is een Python "package" dat zeer veel wordt gebruikt voor numeriek rekenwerk. Het belangrijkste onderdeel van NumPy is een multidimensionale array datastructuur, waar we uitgebreid kennis mee zullen maken. Vele wiskundige operaties zijn allemaal in NumPy ingebouwd en klaar voor gebruik. De NumPy array is zeer snel (deze is eigenlijk geïmplementeerd in C++) en dus geschikt voor het verwerken van grote hoeveelheden data.

Om NumPy te kunnen gebruiken in een programma moeten we het NumPy package eerst importeren: `import numpy as np`. Via `np.` kunnen we nu alle NumPy functies en objecten gebruiken. In de tekst gaan we er in alle voorbeelden met de interactieve prompt vanuit dat de NumPy package is geïmporteerd.

20.1 NumPy array datastructuur

De NumPy array datastructuur is een multidimensionale array zoals je deze ook in C++ hebt leren kennen. Er zijn dus een aantal belangrijke verschillende ten opzichte van de Python 'list' datastructuur:

- Voor NumPy arrays moet van te voren een aantal dimensies en een grootte worden opgegeven (net als arrays in C++). Aan de hand hiervan wordt het aantal elementen bepaald dat de array zal bevatten en je kunt dit later niet meer uitbreiden.
- Alle elementen van de array zijn van hetzelfde type. Dit is bij lijsten niet zo.
- Zoals we zullen gaan zien hebben operatoren op NumPy arrays een andere werking dan bij lijsten. De werking ligt veel dichter bij wat je vanuit de wiskunde zou verwachten. Geef dus wanneer je gaat rekenen altijd de voorkeur aan NumPy arrays!

Bij het creëren van de array moeten we het aantal dimensies en de grootte van de dimensies meteen opgeven. We beperken ons voor nu tot een enkele dimensie. Een array bevat dus direct na het maken het aantal gewenste elementen. Het is daarom van belang om te bepalen wat de initiële waarde wordt van deze elementen: de elementen moeten worden geïnitialiseerd. Er zijn verschillende initialisaties mogelijk en daarom ook verschillende manieren om een array te maken. We zitten er nu een aantal op een rijtje.

- `np.array(lijst)`: initialiseer de array aan de hand van de gegeven lijst. De elementen van de lijst worden allen in de array geplaatst.
- `np.zeros(n)`: initialiseer een array ter grootte van `n` elementen met nullen.
- `np.ones(n)`: initialiseer een array ter grootte van `n` elementen met enen.
- `np.tile(v, n)`: initialiseer een array ter grootte van `n` elementen met de waarde `v`.
- `np.arange(start, stop, stap)`: initialiseer een array met een getallenreeks. Deze functie werkt precies zoals `range()`, maar in tegenstelling tot `range()` mag er ook met floating-point getallen worden gewerkt.
- `np.linspace(start, stop, n)`: initialiseer een array met `n` elementen, gelijkmatig verdeeld tussen `start` en `stop`. **Let op:** de waarde `stop` telt in dit geval **wel** mee en zal worden opgenomen als laatste element van de array.

En deze functies kunnen als volgt worden gebruikt:

```
>>> np.array([1, 2, 3, 4, 5, 6])
array([1, 2, 3, 4, 5, 6])
>>> np.zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> np.ones(6)
array([ 1.,  1.,  1.,  1.,  1.,  1.])
>>> np.tile(39., 6)
array([ 39.,  39.,  39.,  39.,  39.,  39.])
>>> np.arange(0, 10, 2)
array([0, 2, 4, 6, 8])
>>> np.linspace(1, 5, 10)
array([ 1.        ,  1.44444444,  1.88888889,  2.33333333,  2.77777778,
        3.22222222,  3.66666667,  4.11111111,  4.55555556,  5.        ])
```

Ook met `print` kun je NumPy arrays afdrukken. Als je het aantal cijfers achter de komma dat wordt afgedrukt wilt aanpassen, dan kan dat met `np.set_printoptions`:

```
>>> A = np.linspace(1, 5, 10)
>>> print A
[ 1.          1.44444444  1.88888889  2.33333333  2.77777778  3.22222222
  3.66666667  4.11111111  4.55555556  5.        ]
>>> np.set_printoptions(precision=3)
>>> print A
[ 1.     1.444  1.889  2.333  2.778  3.222  3.667  4.111  4.556  5.   ]
```

Eigenschappen van een NumPy array, zoals aantal dimensies, kunnen te allen tijde worden opgevraagd. Het volgende voorbeeld laat zien hoe de belangrijkste eigenschappen kunnen worden opgevraagd:

```
>>> A = np.zeros(6)     # 6 elementen, waarde nul.
>>> A.ndim              # Aantal dimensies.
1
>>> A.shape             # De grootte van elke dimensie (zie ook later).
(6,)
>>> A.size              # Het aantal elementen in de array.
6
>>> A.dtype             # Het datatype van elk element (zie ook hieronder)
dtype('float64')
```

20.2 Datatypen in NumPy

Elk element in een NumPy array heeft hetzelfde datatype. NumPy probeert zelf een geschikt datatype te kiezen aan de hand van hoe de array wordt geïnitialiseerd. Hierboven zagen we al dat wanneer een array wordt geïnitialiseerd met een lijst, het datatype van de elementen in de lijst wordt overgenomen. Bij initialisatie met bijvoorbeeld enen of nullen, wordt standaard een float type gebruikt.

Bij het bekijken van het datatype van een array, zagen we het "float64" type. Dit is geen Python object-type, maar een NumPy datatype. NumPy arrays worden in het geheugen opgeslagen als C arrays. Om de data zo efficiënt mogelijk op te slaan kan er worden gekozen uit meerdere verschillende datatypen. De belangrijkste typen zijn: `np.bool_`, `np.int32`, `np.float64` en `np.complex128`. Voor een compleet overzicht verwijzen we de lezer naar de documentatie[7].

Bij het aanmaken van een array kan het te gebruiken datatype worden gespecificeerd met de toevoeging `dtype=`:

```
>>> np.ones(10)
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
>>> np.ones(10, dtype=np.int32)
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1], dtype=int32)
```

20.3 Rekenen met NumPy arrays

Wanneer we rekenen met een array, willen we vaak dat deze zich gedraagt als een vector of matrix. Een optelling of vermenigvuldiging met een scalair getal moet worden uitgevoerd op alle elementen. Als we een optelling of vermenigvuldiging uitvoeren met een lijst, observeren we echter een vreemd gedrag:

```
>>> l = [1, 2, 3, 4]
>>> l * 4
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> l + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "int") to list
>>> l * l
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'list'
```

We zien dat vermenigvuldiging van een lijst met een scalair leidt tot het n-maal herhalen van deze lijst. De +-operator toegepast op lijsten is geen optelling, maar een concatenatie en kan niet worden toegepast op een lijst en een getal. Tenslotte is het vermenigvuldigen van een lijst en een lijst niet mogelijk.

Het toepassen van wiskundige operaties op lijsten geeft dus helemaal niet het resultaat wat we zouden verwachten. In dit soort gevallen moeten we dus altijd NumPy arrays gebruiken, welke zich wel als een vector gedragen zoals is te zien in het volgende voorbeeld:

```
>>> a = np.array([1, 2, 3, 4])
>>> a * 4
array([ 4,  8, 12, 16])
>>> a + 4
array([5, 6, 7, 8])
>>> a * a
array([ 1,  4,  9, 16])
```

[7] `http://docs.scipy.org/doc/numpy-1.10.1/user/basics.types.html`

Operaties worden elementgewijs op alle elementen toegepast. We kunnen hier gebruik van maken wanneer we een bepaalde formule $f(x)$ willen berekenen voor meerdere x-waarden. We zetten eerst de x-waarden klaar in een array en vervolgens maken we een array met de resultaten. Het is dus niet nodig om een for-loop te schrijven! We kunnen toe met een enkele regel Python welke werkt voor een arbitrair aantal elementen. Het volgende voorbeeld doet dit voor $f_1(x) = x^2$ en $f_2(x) = x^3 + 2x^2 - 3$.

```
>>> x = np.arange(0, 10)
>>> print x
[0 1 2 3 4 5 6 7 8 9]
>>> f1 = x ** 2
>>> f1
array([ 0,  1,  4,  9, 16, 25, 36, 49, 64, 81])
>>> f2 = x ** 3 + 2 * x**2 - 3
>>> f2
array([ -3,   0,  13,  42,  93, 172, 285, 438, 637, 888])
```

We gaan nu een ander soort functies bekijken welke wanneer toegepast op een array maar een enkele resultaatwaarde oplevert. Een array wordt in feite gereduceerd tot een enkele waarde. We noemen dit "reductieoperatoren". Veel gebruikte reductieoperatoren zijn sommatie, gemiddelde en minimum/maximum. Deze operatoren kunnen als volgt worden toegepast:

```
>>> a = np.array([31, 16, 68, 40, 44, 52, 4, 28, 33, 14])
>>> np.sum(a)
330
>>> np.amin(a)
4
>>> np.amax(a)
68
>>> a.sum()
330
```

Het laatste voorbeeld laat zien dat veel van deze functies ook als methode op een array object mogen worden toegepast. Een kort overzicht van de namen van de belangrijkste reductieoperatoren is als volgt:

- `np.sum`: sommeer de elementen van de array.
- `np.prod`: product van de elementen van de array.
- `np.amin`: bepaal minimum element van de array en `np.amax` het maximum element.
- `np.mean`: gemiddelde van de elementen van de array.
- `np.std`: bepaal de standaard deviatie van de elementen van de array.

Uiteraard kunnen we ook met indexing en slicing werken. Het uitlezen van de array met indexing en slicing werkt precies zoals je bent gewend (inclusief stapgroottes). Met een index krijg je één array-element terug en wanneer je gebruik maakt van een slice krijg je een subarray terug. Het toekennen aan een slice wijkt iets af van hoe dat gaat met lijsten. Als je een scalair toekent aan een slice, dan krijgt elk element van de slice die waarde (anders dan bij lijsten!). Als je een reeks van waarden wilt toekennen aan een slice, dan moet die reeks hetzelfde aantal elementen bevatten als de slice. We kunnen immers het aantal elementen van de array niet meer veranderen, anders dan bij lijsten!

```
>>> A = np.arange(0, 10)
>>> A[1:4] = 10
>>> print A
[ 0 10 10 10  4  5  6  7  8  9]
>>> A[8:] = [20, 21, 22, 23]    # Reeks om toe te kennen groter dan slice
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot copy sequence with size 4 to array axis with dimension 2
>>> A[8:] = [20, 21]
>>> print A
[ 0 10 10 10  4  5  6  7 20 21]
```

20.4 Wiskundige functies en constanten

Belangrijke wiskundige functies zoals de goniometrische functies, *log* en wortel zijn in NumPy gedefinieerd als functies. Deze functies kun je aanroepen met een enkel getal, maar uiteraard ook met NumPy arrays. Wanneer aangeroepen voor een array, dan wordt de functie op elk element van de array toegepast. Een overzicht van belangrijke functies:

- Goniometrische functies: `np.sin`, `np.cos`, `np.tan`. Let op dat de functies argumenten verwachten in radialen. Gebruik `np.deg2rad` om graden naar radialen om te zetten.
- Logaritmische functies: `np.log`, `np.log10`.
- Exponentiële functie: `np.exp`.
- `np.floor` om naar beneden af te ronden, `np.ceil` om naar boven af te ronden.
- Vierkantswortel: `np.sqrt`.

Opmerking: als je over deze functies documentatie probeert te lezen met `help(np.sin)` krijg je een standaardpagina over "ufuncs". Dit komt door de manier waarop deze functies in NumPy zijn geïmplementeerd. Om specifieke informatie te krijgen over een functie, gebruik dan `np.info(np.sin)`.

Ook zijn er verschillende wiskundige constanten beschikbaar: `np.pi`, `np.e` [8]. In het volgende voorbeeld is te zien hoe de sinus kan worden berekend voor een enkele waarde en voor een array.

```
>>> np.sin(np.deg2rad(90))
1.0
>>> x = np.linspace(0, np.pi / 2., 10)
>>> y = np.sin(x)
>>> print y
[ 0.          0.17364818  0.34202014  0.5         0.64278761  0.76604444
  0.8660254   0.93969262  0.98480775  1.        ]
```

20.5 Floating-point getallen

Reële getallen worden in computers opgeslagen als "floating-point" getallen. Omdat de opslagcapaciteit niet onbeperkt is, is het niet mogelijk om alle mogelijke reële getallen op te slaan. Een getal zal dus worden afgerond naar het dichtstbijzijnde getal dat wel als floating-point getal kan

[8] Mocht je in de toekomst ook gebruik gaan maken van SciPy, in deze package zijn ook verschillende natuurkundige constanten gedefinieerd: `http://docs.scipy.org/doc/scipy/reference/constants.html`.

worden opgeslagen. Hierdoor worden reële getallen dus bij benadering opgeslagen en het is zeer belangrijk om hier rekening mee te houden bij het schrijven van numerieke programma's. Ook irrationele getallen, zoals π en $\sqrt{2}$, met een oneindig aantal cijfers achter de komma, kunnen dus niet exact worden gerepresenteerd als een floating-point getal.

Moderne hardware slaat floating-point getallen op volgens de IEEE 754 standaard. In Python worden alle floating-point getallen opgeslagen met "double precision". Voor elk getal zijn 64 bits beschikbaar. Deze worden verdeeld over een "exponent", "significant" (of "mantissa") en een "sign". De sign geeft aan of het getal positief of negatief is. Stel we hebben een significant van 1.11011 en een exponent van -3, dan krijgen we $1.11011 \times 2^{-3} = 0.23046875$. Er zijn 53 bits beschikbaar voor de significant (waarvan 1 bit voor de sign) en 11 voor de exponent. We kunnen hiermee 16 significante cijfers opslaan. Alle getallen met meer dan 16 significante cijfers zullen moeten worden afgerond. Een double precision getal heeft een bereik van ongeveer -10^{308} en 10^{308} met 16 significante cijfers.

In veel gevallen zal er dus sprake zijn van een afrondfout. Bij iteratieve berekeningen wordt er dus met een afrondfout doorgerekend, waardoor de fout alsmaar groter kan worden. Wees hiervan bewust! Wanneer je een operatie uitvoert op een klein en zeer groot getal, dan zal de significantie van het kleine getal verdwijnen.

Ook bij het vergelijken van floating-point getallen moeten we erg voorzichtig zijn. Bijvoorbeeld 6.1 wordt opgeslagen als 6.0999999999999996. Een statement als `x == 6.1` kan dus iets anders doen dan verwacht! Maak in dit soort gevallen altijd gebruik van de functie `np.allclose(a, b)` met `a` en `b` losse getallen of arrays. Deze functie doet geen exacte vergelijking, maar kijkt of `a` en `b` binnen een zeer kleine tolerantie hetzelfde zijn.

20.6 Werken in meerdere dimensies

Tot nu toe hebben we alleen gewerkt met arrays met één dimensie. De NumPy array kan echter een arbitrair aantal dimensies aan. Dimensies worden in NumPy ook wel "assen" (axes) genoemd. Elke as heeft een bepaalde lengte. Een NumPy array kan dus worden omschreven door de lengtes van alle assen te benoemen. We noemen dit de vorm van de array. De lengtes van de assen worden vaak weergeven als een tuple. Bijvoorbeeld: `(2, 3, 5)` is een 3-dimensionale array met dimensies 2, 3 en 5.

Wanneer we multidimensionale arrays willen maken, geven we een tuple op dat de vorm van de array specificeert. Op deze manier kunnen we gebruik blijven maken van `np.ones`, `np.zeros` en `np.tile` waarmee we al kennis hebben gemaakt. In plaats van het aantal elementen, geven we een tuple dat de vorm specificeert als argument. Een andere handige functie om arrays te maken is `np.eye` waarmee identiteitsmatrices kunnen worden gemaakt (`np.eye` accepteert geen vorm-tuple als argument en maakt altijd een 2-dimensionale array).

```
>>> A = np.tile(6, (3, 4))    # 3 rijen, 4 kolommen
>>> print A
[[6 6 6 6]
 [6 6 6 6]
 [6 6 6 6]]
>>> B = np.zeros((2, 4, 3))
>>> print B
[[[ 0.  0.  0.]
  [ 0.  0.  0.]
  [ 0.  0.  0.]
  [ 0.  0.  0.]]

 [[ 0.  0.  0.]
  [ 0.  0.  0.]
  [ 0.  0.  0.]
  [ 0.  0.  0.]]]
>>> I = np.eye(3)    # Een 3x3 identiteitsmatrix
```

```
>>> print I
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Let bij de uitvoer van het array op het aantal blokhaken. Voor elke dimensie wordt er een blokhaak geopend en gesloten. Array `A` is 2-dimensionaal en merk op dat er twee blokhaken openen worden afgedrukt. Bij de 3-dimensionale array `B` zijn dit er 3. Het aantal elementen tussen twee blokhaken komt overeen met de lengte van de laatste (binnenste) dimensie.

Ook kunnen arrays worden aangemaakt gebaseerd op een geneste lijst. Matrices kunnen worden gemaakt op basis van een string (vergelijk MatLab) waarbij de rijen worden gescheiden door een puntkomma:

```
>>> C = np.array([[1, 2, 3], [6, 7, 4]])
>>> print C
[[1 2 3]
 [6 7 4]]
>>> print C.shape
(2, 3)
>>> D = np.array(np.mat("1 2 3; 6 7 1"))
>>> print D
[[1 2 3]
 [6 7 1]]
```

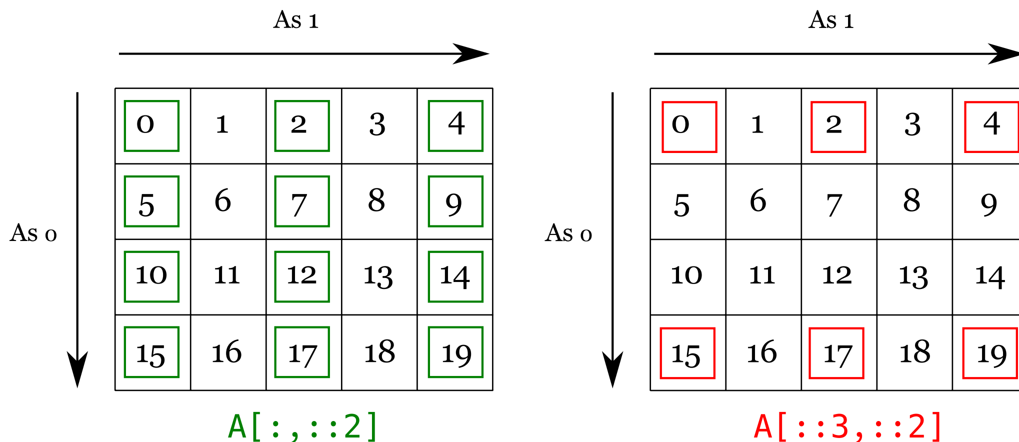
Let ook op dat wanneer je een kopie wilt maken van een array, je dit **expliciet** moet aangeven!

```
>>> A = np.eye(3)
>>> B = A             # Kopieert niet, maar legt een extra referentie aan.
>>> B[0,2] = 9        # Indexeren komen we later op
>>> print A           # A is dus ook aangepast!
[[ 1.  0.  9.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
>>> B = np.copy(A)    # De correcte manier om een kopie te maken.
```

Tenslotte bekijken we het veranderen van de vorm. In tegenstelling tot het aantal elementen van een array, ligt de vorm van een array niet vast. We kunnen de vorm van de array veranderen, zonder dat de waarden van de elementen van de array worden veranderd[9]. De methode `.ravel` maakt een array 1-dimensionaal, met `.reshape` mogen we zelf een vorm opgegeven.

```
>>> print np.eye(3).ravel()
[ 1.  0.  0.  0.  1.  0.  0.  0.  1.]
>>> print np.arange(10, 20).reshape((2, 5))
[[10 11 12 13 14]
 [15 16 17 18 19]]
>>> print np.arange(10, 20).reshape((5, 2))
[[10 11]
 [12 13]
 [14 15]
 [16 17]
 [18 19]]
# Let op: we kunnen geen 10 elementen kwijt in een 3x3 array!
>>> print np.arange(10, 20).reshape((3, 3))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

[9] Technisch gezien wordt er een extra “view” gemaakt naar dezelfde array.

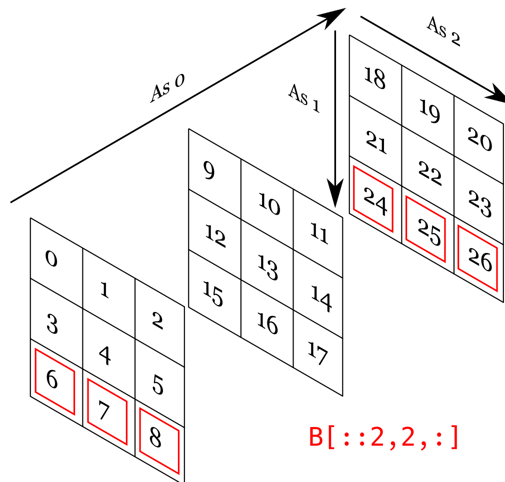


Figuur 1: Representatie van een 2-d array met assen en twee voorbeeld slices.

20.7 Slicing en indexing met multidimensionale arrays

Om slicing en indexing toe te passen op multidimensionale arrays geef je een index of slice op per as (dimensie) van de array, gescheiden door komma's. Als je een slice zonder indexen opgeeft (`:`), dan wordt de gehele as geselecteerd. We bekijken nu een aantal voorbeelden. Onthoud dat bij 2-dimensionale arrays de eerste as (as 0) de rij aanduidt en de tweede as (as 1) de kolom, zie ook Figuur 1.

```
>>> A = np.arange(20).reshape( (4, 5) )
>>> A[:,:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> A[2,1]        # Selecteer een enkel element
11
>>> A[2,:]        # Selecteer de derde rij.
array([10, 11, 12, 13, 14])
>>> A[2]          # Slices aan het einde mag je weglaten (zie hieronder).
array([10, 11, 12, 13, 14])
>>> A[:,3]        # Selecteer de vierde kolom
array([ 3,  8, 13, 18])
>>> A[:,3:]       # Selecteer de vierde en volgende kolom
array([[ 3,  4],
       [ 8,  9],
       [13, 14],
       [18, 19]])
>>> A[:,::2]     # Selecteer kolom 0, 2, 4, ...
array([[ 0,  2,  4],
       [ 5,  7,  9],
       [10, 12, 14],
       [15, 17, 19]])
>>> A[::3,::2]  # Selecteer rij 0, 3, ... en kolom 0, 2, 4, ...
array([[ 0,  2,  4],
       [15, 17, 19]])
>>> A[::3,::2] = 99    # Je mag toekennen aan dit soort slices!
>>> A
array([[99,  1, 99,  3, 99],
       [ 5,  6,  7,  8,  9],
```



Figuur 2: Representatie van een 3-d array met assen en voorbeeldslice.

```
       [10, 11, 12, 13, 14],
       [99, 16, 99, 18, 99]])
# Bij het toekennen moet de array de juiste vorm hebben!
>>> A[::3,::2] = np.arange(100, 106).reshape((2,3))
>>> A
array([[100,   1, 101,   3, 102],
       [  5,   6,   7,   8,   9],
       [ 10,  11,  12,  13,  14],
       [103,  16, 104,  18, 105]])
```

Na het opgeven van een index of slice voor de eerste as, mogen de andere assen worden weggelaten. Wanneer dit gebeurt wordt er impliciet een : gelezen.

Voor de volledigheid bekijken we ook een klein voorbeeld met een 3-dimensionale array. Hier kiest de eerste as een "vlak", de tweede as een rij en de derde weer een kolom, zie ook Figuur 2.

```
>>> B = np.arange(27).reshape( (3,3,3) )
>>> B[0,:,:]         # Kies alleen "voorste" vlak; B[0] is equivalent.
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> B[0,:,2]         # Kies uit het voorste vlak de derde kolom.
array([2, 5, 8])
>>> B[::2,2,:]       # Uit vlakken 0, 2, ... kies de derde rij.
array([[ 6,  7,  8],
       [24, 25, 26]])
>>> B[:,::2,::2]     # Uit alle vlakken, selecteer rij/kolom 0, 2, ...
array([[[ 0,  2],
        [ 6,  8]],

       [[ 9, 11],
        [15, 17]],

       [[18, 20],
        [24, 26]]])
```

20.8 Wiskundige operaties op multidimensionale arrays

Veel wiskundige operaties op multidimensionale arrays werken zoals je zou verwachten. Wel is vereist dat de vormen van de twee arrays compatibel zijn. Als de twee vormen gelijk zijn is er natuurlijk aan deze eis voldaan. Ook mag een scalair getal als operand worden opgegeven.

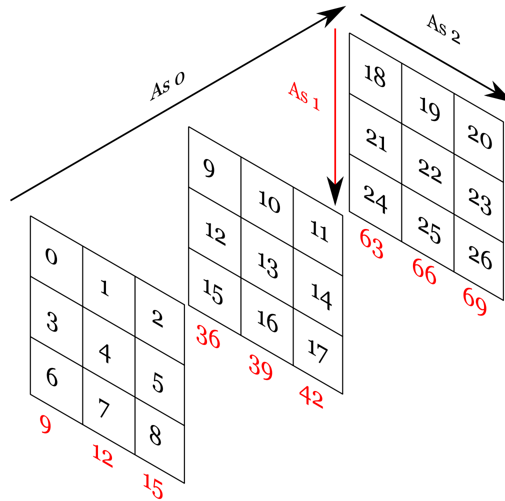
```
>>> np.ones( (3, 3)) + np.tile(10, (3, 3) )
array([[ 11.,  11.,  11.],
       [ 11.,  11.,  11.],
       [ 11.,  11.,  11.]])
>>> np.eye(4) * 9
array([[ 9.,  0.,  0.,  0.],
       [ 0.,  9.,  0.,  0.],
       [ 0.,  0.,  9.,  0.],
       [ 0.,  0.,  0.,  9.]])
>>> np.arange(9).reshape( (3, 3) ) * 2
array([[ 0,  2,  4],
       [ 6,  8, 10],
       [12, 14, 16]])
```

Belangrijk! Het vermenigvuldigen van twee arrays gebeurt standaard elementgewijs, dit is anders dan het dot product (matrixvermenigvuldiging)! Om een dot product uit te voeren gebruik je de functie `np.dot(A, B)`.

```
>>> A = np.eye(3)            # Identiteitsmatrix
>>> B = np.tile(4, (3, 3))   # Alle elementen 4.
>>> A * B
array([[ 4.,  0.,  0.],
       [ 0.,  4.,  0.],
       [ 0.,  0.,  4.]])
>>> np.dot(A, B)
array([[ 4.,  4.,  4.],
       [ 4.,  4.,  4.],
       [ 4.,  4.,  4.]])
```

Als twee arrays niet dezelfde vorm hebben is het soms toch mogelijk om een operatie uit te voeren. Er wordt dan gekeken of de lengtes van dimensies overeenkomen. Als er dimensies zijn weggelaten wordt er gekeken of er een dimensie kan worden toegevoegd aan de buitenkant (dus vooraan in de vorm-tuple): dus bijvoorbeeld $(3,)$ mag $(1,3)$ worden. Stel we hebben een array met vorm $(5,6)$ en een array met vorm $(6,)$. NumPy maakt van de tweede array een array met vorm $(1,6)$. Nu komen de dimensies van de tweede as (de rij) overeen. NumPy zal nu de tweede array (een rij) optellen bij elke rij van de eerste array. Zie ook het volgende voorbeeld:

```
>>> A = np.ones( (4, 3) )
>>> B = np.array( [1, 2, 3] )     # shape: (3, )
>>> A + B                         # B wordt opgeteld bij elke rij van A
array([[ 2.,  3.,  4.],
       [ 2.,  3.,  4.],
       [ 2.,  3.,  4.],
       [ 2.,  3.,  4.]])
>>> A + B.reshape( (1, 3) )       # Dezelfde operatie
array([[ 2.,  3.,  4.],
       [ 2.,  3.,  4.],
       [ 2.,  3.,  4.],
       [ 2.,  3.,  4.]])
>>> A + np.array([[1, 2, 3]])     # Dezelfde operatie, dubbele blokhaken!!
array([[ 2.,  3.,  4.],
```



Figuur 3: Grafische weergave van `B.sum(axis=1)`.

```
       [ 2.,  3.,  4.],
       [ 2.,  3.,  4.],
       [ 2.,  3.,  4.]])
# Stel nu, C heeft lengte vier.
>>> C = np.array( [1, 2, 3, 4] )
# Dit lukt nu niet, want NumPy probeert met (1, 4) en dan komen de dimensies
# van de tweede as niet overeen (3 != 4).
>>> A + C
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (4,3) (4)
```

Maar wat nu als we een rij van lengte 4 willen optellen bij de kolommen van A? We moeten dan de rij een andere vorm geven zodanig dat de operatie kan worden uitgevoerd. De kleine array moet dus van vorm 4 worden omgezet naar $(4, 1)$ (4 rijen, 1 kolom). Dit kan met de methode `.reshape`, maar ook door `np.newaxis` op te geven als index waardoor er automatisch een nieuwe as wordt gemaakt.

```
>>> A + C.reshape( (4, 1) )
array([[ 2.,  2.,  2.],
       [ 3.,  3.,  3.],
       [ 4.,  4.,  4.],
       [ 5.,  5.,  5.]])
# Of de volgende kortere notatie, waarbij we alle elementen van C
# selecteren en een nieuwe as toevoegen aan het einde.
>>> A + C[:,np.newaxis]
array([[ 2.,  2.,  2.],
       [ 3.,  3.,  3.],
       [ 4.,  4.,  4.],
       [ 5.,  5.,  5.]])
```

Hoe gedragen reductieoperatoren zich nu op multidimensionale arrays? Standaard worden alle elementen in alle dimensies gereduceerd tot een scalair. Je kunt ook specificeren dat een reductieoperator langs een bepaalde as (dimensie) van de array moet werken. Dit laat zich weer het beste illustreren met een voorbeeld. Laten we als voorbeeld een 3-dimensionale array nemen en via elke as de som bepalen. In Figuur 3 is de berekening van `np.sum(axis=1)` grafisch weergegeven.

```
>>> B = np.arange(27).reshape( (3,3,3) )
>>> B.sum()
351
>>> s = B.sum(axis=2)  # "Binnenste" as: sommeer elke rij (dus loop kolom-as af)
>>> s
array([[ 3, 12, 21],
       [30, 39, 48],
       [57, 66, 75]])
>>> s[0, 1]            # Eerste "vlak", tweede rij.
12
>>> s[2, 0]            # Derde "vlak", eerste rij.
57
>>> B.sum(axis=1)      # Sommeer elke kolom (dus langs de rij-as)
array([[ 9, 12, 15],
       [36, 39, 42],
       [63, 66, 69]])
>>> B.sum(axis=0)      # Sommeer langs de diepte-as.
array([[27, 30, 33],
       [36, 39, 42],
       [45, 48, 51]])
```

Je kunt ook cumulatieve sommen laten berekenen langs een bepaalde as. De richting waarin wordt gesommeerd komt dan ook duidelijk naar voren. Bijvoorbeeld langs de kolom-as (merk op dat dit geen reductieoperatie is en het aantal elementen en dimensies dus niet afneemt):

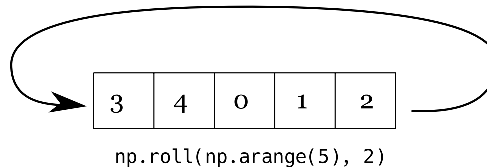
```
>>> B.cumsum(axis=2)
array([[[ 0,  1,  3],
        [ 3,  7, 12],
        [ 6, 13, 21]],

       [[ 9, 19, 30],
        [12, 25, 39],
        [15, 31, 48]],

       [[18, 37, 57],
        [21, 43, 66],
        [24, 49, 75]]])
```

Je ziet dat NumPy heel veel bewerkingen al ingebouwd heeft. Je zou natuurlijk ook zelf loops kunnen schrijven om een rij bij alle rijen van een array op te tellen of om een som te bepalen, maar het is veel beter om de ingebouwde NumPy functies te gebruiken. Niet alleen is dit veel eenvoudiger, de ingebouwde functies zijn ook vele malen sneller omdat deze eigenlijk in C zijn geïmplementeerd. De vuistregel bij het gebruik van NumPy is om zo weinig mogelijk loops te gebruiken!

20.9 Rollen en roteren

De data in een array kan worden "gerold" en "geroteerd". Bij rollen schuiven we de elementen in een array n plaatsen op. Elementen die uit de array worden geschoven, worden aan de voorkant van de array er weer in geschoven, zie ook Figuur 4. In het geval van multidimensionale arrays geven we een as aan als richting waarin moet worden gerold.

```
>>> A = np.arange(9)
>>> np.roll(A, 3)           # Schuif de elementen 3 plaatsen door.
array([6, 7, 8, 0, 1, 2, 3, 4, 5])
>>> A = A.reshape( (3, 3) )
>>> A
```



Figuur 4: Grafische weergave van `np.roll(np.arange(5), 2)`.

```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
>>> np.roll(A, 1, axis=0)   # Schuif 1 plaats door in de kolom-richting.
array([[6, 7, 8],
       [0, 1, 2],
       [3, 4, 5]])
>>> np.roll(A, 2, axis=0)   # Schuif 2 plaatsen door in de kolom-richting.
array([[3, 4, 5],
       [6, 7, 8],
       [0, 1, 2]])
```

Met de functie `np.rot90` kunnen de eerste twee dimensies van een multidimensionale array worden geroteerd. Standaard is de rotatie 90 graden tegen de klok in. Er zijn ook functies om data te spiegelen, `np.fliplr` om horizontaal te spiegelen en `np.flipud` voor verticaal.

```
>>> np.rot90(A)         # Draai 90 graden tegen de klok in
array([[2, 5, 8],
       [1, 4, 7],
       [0, 3, 6]])
>>> np.fliplr(A)        # Horizontaal spiegelen.
array([[2, 1, 0],
       [5, 4, 3],
       [8, 7, 6]])
```

20.10 Selectie van elementen en maskers

We kunnen ook Boolean expressies evalueren voor een NumPy array. Bijvoorbeeld om te kijken of alle elementen van een array aan een bepaalde conditie voldoen. Of om na te gaan of er ten minste één element bestaat dat aan een conditie voldoet. We gebruiken hiervoor respectievelijk `np.all` en `np.any`.

```
>>> A = np.arange(10, 19).reshape( (3, 3) )
>>> np.all(A >= 15)      # Zijn alle elmenten >= 15?
False
>>> np.all(A >= 10)      # >= 10?
True
>>> np.any(A == 14)      # Is er tenminste een element gelijk aan 14?
True
>>> np.any(A == 4)       # En aan 4?
False
>>> np.any(A < 10)       # Tenminste een element kleiner dan 10?
False
```

Wat gebeurt hier nu eigenlijk? Hoe toepassen van een Boolean expressie op een array resulteert eigenlijk in een Boolean array. `np.all` kijkt dan vervolgens of alle elementen van die Boolean array true zijn. We kunnen ook het aantal keer true tellen in de Boolean array. Tevens kan de

Boolean array worden gebruikt als een soort "slice" om alleen die elementen uit de array te kiezen waarvoor de Boolean waarde true is (we zeggen dan dat we de Boolean array als masker ("mask") gebruiken). Merk op dat we met maskers ingewikkelde (en ook onregelmatige) patronen van elementen kunnen selecteren welke niet mogelijk zijn met een enkele slice. We kunnen zelfs een operatie uitvoeren op deze gekozen subset van de array!

```
>>> A = np.arange(10, 19).reshape((3, 3))
>>> A >= 15
array([[False, False, False],
       [False, False,  True],
       [ True,  True,  True]], dtype=bool)
>>> np.sum(A >= 15)       # Tel aantal keer true in de Boolean array
4
>>> mask = A >= 15
>>> print A[mask]         # Druk elementen >= 15 af. (Let op: 1-d view)
[15 16 17 18]
>>> A[mask] += 100        # Tel 100 op bij elementen >= 15
>>> print A
[[ 10  11  12]
 [ 13  14 115]
 [116 117 118]]
```

20.11 Random numbers

NumPy kent ook methoden voor het genereren van random getallen. Zo is het eenvoudig om een random array te initialiseren. Standaard maakt NumPy gebruik van een pseudo-random number generator: de getallen zijn dus niet echt willekeurig, maar worden volgens een bepaalde procedure gegenereerd. Voor de meeste toepassingen is een pseudo-random generator echter goed genoeg.

Met `np.random.random()` kunnen we random getallen en arrays maken. De elementen zullen zitten in het interval $[0.0, 1.0)$. Als je getallen in een integer range wilt kun je gebruik maken van `np.random.random_integers()`:

```
>>> np.random.random()
0.9420733512975746
>>> np.random.random( (3, 3) )
array([[ 0.85083159,  0.28587965,  0.69833045],
       [ 0.98522151,  0.93762675,  0.29451167],
       [ 0.17332978,  0.87714118,  0.36772117]])
# Integers tussen 0 t/m 10, shape (3, 3)
>>> np.random.random_integers(0, 10, (3, 3) )
array([[6, 9, 4],
       [8, 0, 5],
       [8, 7, 1]])
```

Een andere handige functie is `np.random.choice`[10], waarmee een trekking wordt gedaan uit een gegeven array. Standaard wordt er één waarde gekozen.

```
>>> np.random.choice(np.arange(100, 200))
117
>>> np.random.choice(np.arange(100, 200), 5)  # Trekking van 5 elementen
array([190, 135, 176, 112, 101])
```

NumPy kent ook vele kansverdelingen ("probability distributions") waaronder de veel gebruikte normaalverdeling. Met de functie `np.random.normal()` kunnen getallen worden getrokken uit de normaalverdeling. Je mag waarden voor μ en σ opgegeven als argumenten:

[10] Alleen beschikbaar in NumPy versie 1.7.0 en hoger.

```
>>> np.random.normal(0.0, 0.4)
0.18566812216203574
>>> np.random.normal(0.0, 0.4)
-0.719179308621786
>>> np.random.normal(0.0, 0.4)
-0.3502191433586541
>>> np.random.normal(0.0, 0.4, 10)    # Meteen 10 waarden
array([-0.08439069,  0.37264074, -0.40323753,  0.50495263,  0.13786892,
       -0.02567837, -0.88560125, -0.22346038,  0.04929721, -0.12100758])
```

20.12 Data lezen uit een bestand

NumPy heeft een ingebouwde functie om data uit tekstbestanden te laden die bestaan uit regels met getallen. Er zijn ook geavanceerde methoden om data in te lezen en weg te schrijven met NumPy, maar we zullen deze in dit college niet behandelen. Stel we hebben een tekstbestand `test1.txt` met de volgende inhoud:

```
1 2 3
4 5 6
0 9 1
9 3 4
```

dan kunnen we dit als volgt inlezen:

```
>>> A = np.loadtxt("test1.txt")
>>> print A
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 0.  9.  1.]
 [ 9.  3.  4.]]
```

In het geval de getallen op een regel worden gescheiden met komma's (dit is het geval in `test2.txt`), moeten we het scheidingsteken aangeven:

```
>>> A = np.loadtxt("test2.txt", delimiter=",")
>>> print A
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 0.  9.  1.]
 [ 9.  3.  4.]]
```

Tenslotte nog een voorbeeld met kolomkoppen:

```
kol1 kol2 kol3
4.5  34.3 35.3
50.3 23.  2.4
```

`np.loadtxt` kan de strings in de kopjes niet lezen, we moeten dan aangeven om de eerste regel over te slaan. We geven als argument het aantal regels dat moet worden overslagen, in dit geval 1:

```
>>> A = np.loadtxt("test3.txt", skiprows=1)
>>> print A
[[  4.5  34.3  35.3]
 [ 50.3  23.    2.4]]
```

21 Matplotlib

Matplotlib is een Python package waarmee plots kunnen worden gemaakt van een zeer hoge kwaliteit. De kwaliteit van de plots is hoog genoeg om deze op te kunnen nemen in wetenschappelijke publicaties. Matplotlib ondersteunt een zeer groot aantal verschillende plots, neem eens een kijkje in de "plot gallery"[11]. Matplotlib wordt normaliter gebruikt in combinatie met NumPy. Met NumPy kun je je data voorbereiden en verwerken. Vervolgens geef je NumPy arrays als parameters aan Matplotlib functies om deze te plotten.

Vanwege de grote hoeveelheid aan verschillende plots die met Matplotlib kunnen worden gemaakt, kunnen we Matplotlib niet in zijn totaliteit bespreken. We beperken ons tot het maken van een aantal simpele plots met behulp van "pyplot", een eenvoudig raamwerk om plots mee te maken.

21.1 Een eerste plot

Laten we beginnen met het maken van een plot van een simpele lineaire functie. Eerst importeren we de benodigde packages. Vervolgens berekenen we onze coördinaatparen. Tenslotte plotten we deze coördinaten en zetten we de plot op het scherm (we zullen later zien hoe we de plot naar een bestand kunnen laten schrijven).

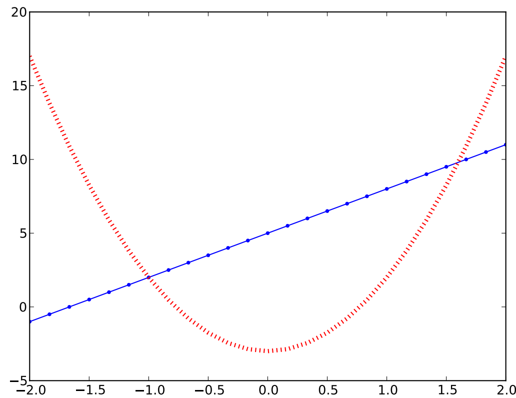
```
import numpy as np
import matplotlib.pyplot as plt

# Bepaal de x-coordinaten die we willen plot.
x = np.arange(0, 10, 0.5)
# Bereken nu voor elk x-coordinaat de y-waarde
# Functie: y = 3x + 5
y = 3 * x + 5

# Geef de x- en y-arrays als parameters aan de plot functie.
plt.plot(x, y)

# Zet de plot op het scherm
plt.show()

exit(0)
```

21.2 Kleuren en markers

De functie `plt.plot` accepteert een groot aantal argumenten om eigenschappen van de te plotten lijn aan te passen. Omdat de meeste van deze argumenten allemaal standaardwaarden hebben, hoeven wij deze niet expliciet te specificeren. Om eigenschappen van de lijn aan te passen kunnen we met behulp van keyword arguments alleen die eigenschappen opgeven die we willen aanpassen. Eigenschappen die je vaak zult aanpassen zijn bijvoorbeeld:

- `color` - de kleur van de lijn. Matplotlib kent een hoop namen van kleuren zoals `red`, `black`, `blue`, `yellow` enzovoort[12]. Als je favoriete kleur er niet bij zit mag je ook een kleur specificeren als hexadecimaal getal: `#E9967A`.
- `marker` - met marker kun je ervoor kiezen om alle geplotte punten te markeren. Standaard staat dit uit. Markers die je kunt kiezen zijn bijvoorbeeld `"."` (punt), `"o"` (cirkel), `"x"` (kruis) of `"*"` (ster). Voor een volledig overzicht zie[13].

[11] http://matplotlib.org/gallery.html
[12] Matplotlib kent alle HTML kleuren, zie hier voor een lijst: http://www.w3schools.com/html/html_colornames.asp.
[13] http://matplotlib.org/api/markers_api.html#module-matplotlib.markers



Figuur 5: Plot van twee lijnen met verschillende eigenschappen.

- `linewidth` - om de dikte van de lijn te bepalen. De parameter is een floating-point getal.
- `linestyle` - om te kiezen voor een doorgetrokken lijn of stippel lijn. Probeer eens `"solid"`, `"dashed"`, `"dashdot"` en `"dotted"`.
- `label` - om een labelstring op te geven voor deze lijn, welke in de legenda zal verschijnen (zie hieronder).

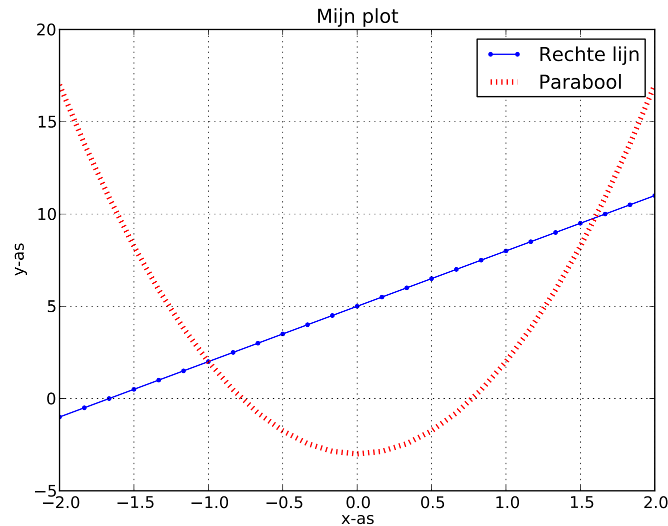
Het volgende programma plot twee lijnen in verschillende stijlen, zoals in Figuur 5 is te zien.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2, 2, 25)
y1 = 3 * x + 5
y2 = 5 * x ** 2 - 3

plt.plot(x, y1, color="blue", lw=1.0, linestyle="solid", marker=".")
plt.plot(x, y2, color="red", lw=4.0, linestyle="dotted")

plt.show()

exit(0)
```

21.3 De plot opmaken

Zonder titel en as-labels is een plot natuurlijk niet af. Om de plot op te maken kun je de volgende functies gebruiken:

- `plt.title(s)` stelt de string `s` in als titel in voor de gehele plot.
- `plt.xlabel(s)` stelt de string `s` in als label voor de x-as.
- `plt.ylabel(s)` stelt de string `s` in als label voor de y-as.
- `plt.grid(True)` zet een “grid” aan.



Figuur 6: Nette plot met titel en as-labels

- `plt.legend(loc=locstr)` voegt een legenda toe, op basis van de labels ingesteld bij het aanroepen van `plt.plot`. Met `locstr` kan de locatie van de legenda worden gekozen: `upper right`, `lower left`, `center`, enzovoort.

Nog een leuk detail: je mag gebruik maken van TeX-markup voor wiskundige tekens en formules in de titel- en labelstrings.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(-2, 2, 25)
y1 = 3 * x + 5
y2 = 5 * x ** 2 - 3

plt.plot(x, y1, color="blue", lw=1.0, linestyle="solid", marker=".", label="Rechte lijn")
plt.plot(x, y2, color="red", lw=4.0, linestyle="dotted", label="Parabool")

plt.title("Mijn plot")
plt.xlabel("x-as")
plt.ylabel("y-as")

plt.grid(True)

plt.legend(loc="upper right")

plt.show()

exit(0)
```

21.4 Assen instellen

Ook aan de assen kan het een en ander worden ingesteld. Zo kunnen we de intervallen van de assen aangeven die zichtbaar moeten zijn: `plt.ylim(-2, 10)` en `plt.xlim(0, 100)`. Als alternatief kun je ook `plt.axis` gebruiken met keyword arguments:

```
plt.axis(xmin=0, xmax=20., ymin=-10, ymax=100.)
```

Met behulp van `plt.xscale()` en `plt.yscale()` kan de schaal van elke as worden ingesteld. Als parameter geef je bijvoorbeeld `"linear"` of `"log"`.

21.5 Meerdere plots uit één array

Het is ook mogelijk om voor een enkele reeks van x-waarden meerdere functies te plotten en om de data voor deze functies in één array op te slaan. Dit is natuurlijk netter dan het gebruik van een aparte `y1` en `y2` zoals we eerder deden. Daarnaast kunnen we die verschillende functies dan ook met één functieaanroep plotten! Als een multidimensionale array als argument aan `plt.plot` wordt gegeven, dan zal `plt.plot` elke kolom als te plotten getallenreeks behandelen. Er wordt dus kolom-voor-kolom geplot. Zie het volgende voorbeeld:

```
x = np.linspace(-10, 10, 200)
y = np.zeros( (x.shape[0], 3) )
y[:,0] = x ** 2
y[:,1] = 2 * x ** 2
y[:,2] = 4 * x ** 2
h = plot(x, y)
plt.legend(h, ("1", "2", "3"))
plt.show()
```

Een klein nadeel is dat je niet meer de labels voor de legenda kunt opgeven in de `plt.plot()` functieaanroep. In plaats daarvan vang je de returnwaarde van `plt.plot()` op. Deze bevat een lijst van “handles” (verwijzingen) naar de lijnen die op het scherm zijn gezet. In de `plt.legend()` aanroep koppelen we nu aan elk van deze handles een legendastring.

21.6 Plots opslaan naar een bestand

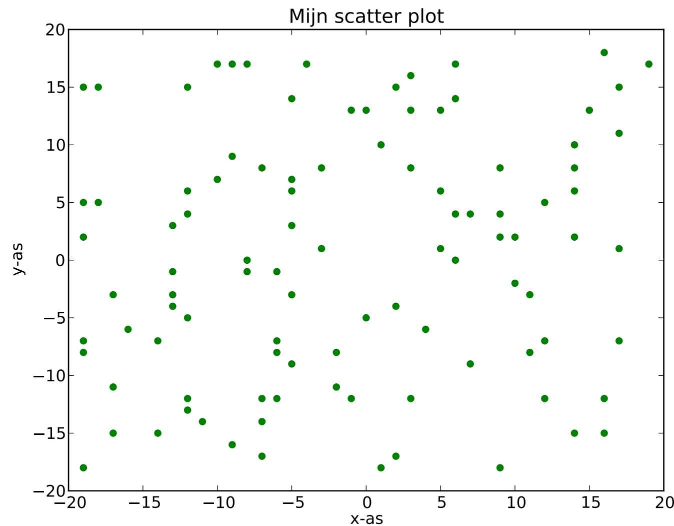
Een plot opslaan naar een bestand is eigenlijk heel eenvoudig. In plaats van `plt.show()` roepen we `plt.savefig()` aan. Als parameter moeten we een bestandsnaam opgeven waarin de plot moet worden opgeslagen. Er kan worden gekozen uit meerdere formaten. Vaak wordt er gebruik gemaakt van PDF bestanden. Bijvoorbeeld `plt.savefig("mijnplot.pdf")` slaat de huidige plot op als `mijnplot.pdf` in PDF-formaat.

21.7 Omgaan met meerdere plots in één programma

Het komt vaak voor dat je met één programma meerdere plots maakt. Normaal gesproken komen herhaalde aanroepen van `plt.plot` terecht in dezelfde figuur. We hebben dus een functie nodig om een nieuw figuur aan te maken. Deze functie is `plt.figure()`. Programma’s zullen er dus vaak als volgt uitzien:

1. `plt.figure()`
2. Één of meerdere aanroepen `plt.plot()`.
3. Plot opmaken door assen in te stellen, titel te zetten. enz.
4. `plt.show()` of `plt.savefig()`.
5. Vervolgens kun je terug naar de eerste stap om aan een nieuwe plot te beginnen.

Je kunt ook tegelijk werken aan meerdere plots: met `plt.figure(1)` en `plt.figure(2)` kun je steeds wisselen tussen twee plots.



Figuur 7: Een voorbeeld van een scatter plot.

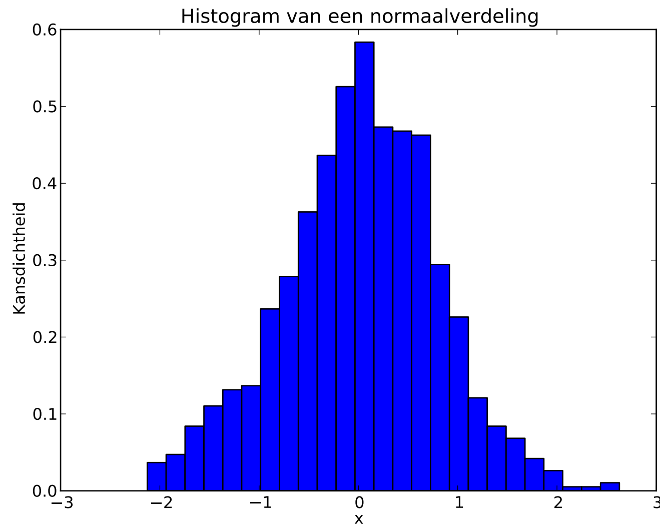
21.8 Scatter plots

Met behulp van `plt.scatter()` kun je scatter plots maken. Het volgende voorbeeldprogramma maakt een scatter plot van 100 willekeurig gekozen coördinaten. Het resultaat is te zien in Figuur 7.

```
import numpy as np
import matplotlib.pyplot as plt

x = np.random.random_integers(-19, 19, 100)
y = np.random.random_integers(-19, 19, 100)

plt.scatter(x, y, color="green")

plt.title("Mijn scatter plot")
plt.xlabel("x-as")
plt.xlim(-20, 20)

plt.ylabel("y-as")
plt.ylim(-20, 20)

plt.savefig("scatterplot.pdf")

exit(0)
```

21.9 Histogrammen maken

Matplotlib kent ook een speciale functie voor het plotten van histogrammen: `plt.hist()`. Laten we bijvoorbeeld een plot maken van de normaalverdeling met behulp van `np.random.normal()` welke we eerder in dit dictaat tegenkwamen. Je kunt ook zelf histogrammen maken van een array met behulp van `np.histogram()`.



Figuur 8: Een histogram van de normaalverdeling.

```
import numpy as np
import matplotlib.pyplot as plt

s = np.random.normal(0.0, 0.8, 1000)

# Histogram met 25 "bins" en normaliseer het histogram
plt.hist(s, bins=25, normed=True)

plt.title("Histogram van een normaalverdeling")
plt.xlabel("x")
plt.xlim(-3, 3)
plt.ylabel("Kansdichtheid")

plt.savefig("plot5.pdf")

exit(0)
```

Het resultaat is te zien in Figuur 8. Uiteraard geldt hoe meer samples er worden genomen, hoe beter de curve van de normaalverdeling zal worden benaderd.

22 iPython

We hebben tot nu toe gebruik gemaakt van de ingebouwde interactieve interpreter van Python. Deze interactieve modus werkt goed, maar is erg basaal. Er bestaan ook uitgebreidere omgevingen om interactief gebruik te maken van Python. Een van de populairste is "iPython". iPython wordt vaak in combinatie met NumPy en matplotlib gebruikt en verhoogt de productiviteit aanzienlijk.

Zodra je met iPython gaat werken zul je zien dat alle resultaten worden opgeslagen in een dictionary `Out`. Je kunt deze waarden dan makkelijk hergebruiken in nieuwe statements:

```
In [1]: 3 + 7
Out[1]: 10
```

```
In [2]: Out[1] * 9
Out[2]: 90
```

Met het commando `hist` krijg je een geschiedenis te zien van ingevoerde commando's. Je kunt commando's ook opnieuw uitvoeren, bijvoorbeeld om `In[5]` nog een keer te draaien schrijf je `%rerun 5`.

iPython functioneert ook als simpele shell! Je kunt gebruik maken van `ls`, `cd` en `cat` om door je bestanden te navigeren. Daarbovenop heeft iPython een zeer goede "tab completion" functionaliteit. Deze werkt voor namen van variabelen, functies, methoden en bestandsnamen. Als je de naam van iets niet meer weet, druk op "Tab" en iPython laat zien wat de mogelijkheden zijn.

Als je iPython opstart in de zogenaamde "pylab" modus (`ipython --pylab`) dan zijn NumPy en matplotlib al geïmporteerd en kun je direct aan de slag. Tenslotte is het mogelijk om iPython in notebook modus op te starten, waarin grafieken direct tussen je code verschijnen. Zie ook [14] en probeer het vooral eens uit!

23 Hoe nu verder?

Dit dictaat is een beknopte introductie tot Python en het gebruik van NumPy en Matplotlib. Binnen de taal Python zijn er veel meer mogelijkheden. Zo is Python volledig object georiënteerd en kun je zelf ook klassen schrijven, operators overloaden, generators implementeren, werken met list comprehensions, lambda functies gebruiken, exception gooien, enzovoort. We geven twee kleine voorbeelden. Een generator is een functie die dienst kan doen als een "iterator" en kan worden gebruikt samen met een `for`-loop. Als de generator het `yield` statement uitvoert, zal de loop met de gegeven waarde voor de iteratorvariabele worden uitgevoerd. De loop wordt uitgevoerd totdat de generator-functie een normale `return` doet. Een eenvoudig voorbeeld is het volgende:

```
def graaf_tel():
    reeks = range(1, 7)
    for getal in reeks:
        yield getal

# Generator gebruiken in een for-loop
for i in graaf_tel():
    print i
```

In dit voorbeeld loopt de generator de getallenreeks 1 t/m 6 af. De `for`-loop zal dus de getallen 1 t/m 6 afdrukken, elk op een nieuwe regel. Een generator komt goed van pas wanneer je een bepaalde datastructuur wilt aflopen of wanneer je een reeks van objecten met een bepaalde karakteristiek wilt aflopen (bijvoorbeeld een reeks matrices met een bepaalde eigenschap).

Een list comprehension is een handige manier om een lijst te genereren. Bijvoorbeeld een lijst bestaande uit n nullen of n lege lijsten. Merk op dat we met NumPy arrays hiervoor allerlei handige initialisatiefuncties hebben gebruikt. Voor lijsten bestaan deze niet en wordt er vaak gebruik gemaakt van list comprehensions. Een list comprehension heeft overigens veel weg van de manier om in de wiskunde de elementen van een verzameling te noteren, bijvoorbeeld

$$\{x \mid x \in \mathbb{N} \wedge x > 3 \wedge x < 10\}$$

We zouden in Python een lijst met dezelfde elementen kunnen aanmaken met

[14] http://ipython.org/notebook.html

```
[x for x in N if x > 3 and x < 10]
```

als `N` een lijst zou zijn met alle natuurlijke getallen. Zo'n begrip hebben we in Python niet, dus we gebruiken `range` om zo'n getallenreeks te genereren:

```
[x for x in range(3, 10)]
```

Je kunt ook een lijst initialiseren met (bijvoorbeeld) 5 lege lijsten als volgt:

```
[[] for i in range(5)]
```

Tenslotte nog twee laatste voorbeelden waarbij we ook het gebruik van een generator combineren. De eerste zal alle integers gegenereerd door de generator omzetten naar een string met behulp van de `map` functie. De tweede sommeert alle elementen gegenereerd door de generator.

```
strings = map(str, [i for i in graaf_tel()])
som = sum(i for i in graaf_tel())
```

Om meer te leren over de taal Python kun je het beste terecht bij "The Python Tutorial" (zie de bronnenlijst).

Daarnaast zijn er vele handige Python modules en packages die je kunt leren gebruiken. Bekijk eens de "The Python Standard Library Reference"[15] waarin de gehele Python standaardbibliotheek staat beschreven. Voor elke module is er een documentatiepagina. Hier vind je eerst een omschrijving van de inhoud en doel van de module. Daarna worden in detail alle klassen en functies in de module beschreven. Alle functieparameters worden benoemd en uitgelegd. Vaak kun je aan het einde van de documentatie enkele voorbeelden van het gebruik van de module terugvinden.

Ook buiten de standaardbibliotheek zijn er veel modules beschikbaar, je kunt ernaar zoeken in de PyPI, the Python Package Index[16]. Om extra modules te installeren gebruik je op Linux of Mac in eerste instantie de package manager van het besturingssysteem ("apt-get", "yum" of "ports"), of op Windows en Mac de package manager van de Python distributie. Probeer anders eens Python pip[17].

Om Python goed onder de knie te krijgen is het beste advies om vooral veel met Python te werken. Gebruik iPython als rekenmachine en probeer simpele taken te automatiseren door een Python programma te schrijven. Je zal merken dat hoe beter je Python gaat beheersen, hoe meer tijdwinst je ermee kunt behalen! Veel succes en plezier!

Bronnen

Walter Kosters. Sheets college Programmeermethoden.
`http://liacs.leidenuniv.nl/~kosterswa/pm/` Geraadpleegd op: 17 november 2015.

Python Software Foundation. The Python Tutorial.
`https://docs.python.org/2/tutorial/` Geraadpleegd op: 15 november 2015.

Python Software Foundation. The Python Language Reference.
`https://docs.python.org/2/reference/`. Geraadpleegd op: 15 november 2015.

Jan-Willem van de Meent, Merlijn van Deen, Bastiaan Florijn en Geert Wortel. Werkcollege Diffusie: Introductie Python en IPython Notebook.

[15] `https://docs.python.org/2/library/index.html`
[16] `https://pypi.python.org/pypi`
[17] `https://docs.python.org/2/installing/`

The Scipy community. NumPy v1.10 Manual.
http://docs.scipy.org/doc/numpy/. Geraadpleegd op: 1 december 2015.

M. Newman. Computational Physics with Python, Chapter 4: Accuracy and speed.
Online beschikbaar: http://www.umich.edu/~mejn/cp/chapters/errors.pdf

Wikipedia. Double-precision floating-point format.
https://en.wikipedia.org/wiki/Double-precision_floating-point_format. Geraadpleegd op: 1 december 2015.