Dictaat College Programmeermethoden NA

Programmeren in Python voor Natuur- en Sterrenkundigen

K. F. D. Rietveld

November 2017

http://liacs.leidenuniv.nl/~rietveldkfd/courses/prna/

1 Introductie

In het college Programmeermethoden NA wordt de programmeertaal Python uitgebreid, maar niet geheel, behandeld. Het vak is speciaal ontwikkeld voor studenten natuurkunde en sterrenkunde en daarom zal ook het werken met numerieke data en het vervaardigen van plots met behulp van Python aan de orde komen. Het doel van het vak is om vaardig genoeg te worden in Python om de kennis van deze programmeertaal te gaan gebruiken als gereedschap bij de andere vakken in de bacheloropleidingen natuur- en sterrenkunde.

Dit dictaat is een beknopte introductie tot de programmeertaal Python en is deels gebaseerd op het collegedictaat van het vak Programmeermethoden waarin wordt gewerkt met de programmeertaal C++. De concepten die tijdens de hoorcolleges aan bod komen worden in dictaat uitgelegd aan de hand van eenvoudige voorbeeldprogramma's. 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 programmeeropdrachten te kunnen volbrengen.

1.1 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.

1.2 Het uitvoeren van een Python-programma

We herhalen kort hoe een Python-programma kan worden uitgevoerd en hoe de interpreter in de interactieve modus kan worden gestart. Voor een samenvatting hoe Python te installeren, zie het kader "Python installeren", voor meer details over installatie en achtergrond over computers en programmeren, zie het dictaat "Computers en programmeren"[1].

Python is een zogenaamde "geïnterpreteerde taal". Om een Python-programma uit te voeren wordt een "interpreter" opgestart. De interpreter is een programma dat een Python-programma leest en dit vervolgens stap voor stap uitvoert. Het programma is een tekstbestand dat de Python-code bevat, geschreven volgens de (syntax)regels van de programmeertaal. Een dergelijk bestand dat direct door een interpreter wordt uitgevoerd wordt ook wel een "script" genoemd en Python een *scripttaal*. Andere bekende scripttalen zijn Perl, Ruby, JavaScript en PHP. Tekstbestanden die Python-code bevatten krijgen een bestandsnaam met de extensie `.py`.

Om een Python-programma te editen en daarna te draaien gebruiken we de volgende commando's[2]:

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

Bij het draaien van het programma kunnen er verschillende fouten optreden:

- `SyntaxError`, bij het inlezen van het programma. Als deze error optreedt, dan klopt de syntax van het programma niet. Er staat bijvoorbeeld een haakje verkeerd, "elze" in plaats van "else", etc.
- `IndentationError`, bij het inlezen van het programma. In dit geval is er verkeerd ingesprongen (komt aan bod in Sectie 5.3).
- `NameError`, bij het uitvoeren van het programma. Er worden variabelen gebruikt die niet zijn gedefinieerd.
- `ValueError`, een ongeldige typeconversie wordt uitgevoerd, zie Sectie 3.3.
- `ZeroDivisionError`, er wordt door nul gedeeld.

Soms is het vervelend dat fouten in het programma pas worden ontdekt bij het daadwerkelijke uitvoeren van het programma. Er bestaan hulpprogramma's om van te voren het hele programma een keer in te lezen en te kijken naar fouten. Voorbeelden zijn *pylint* en *pyflakes*. Hoewel deze programma's een hoop fouten van te voren kunnen opsporen, kunnen niet alle fouten van te voren worden gevonden (zoals bijvoorbeeld delen door 0).

De Python-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 "{0} + {1} = {2}".format(a, b, a + b)
3 + 4 = 7
>>>
```

[1] Verkrijgbaar via: `http://liacs.leidenuniv.nl/~kosterswa/pm/osdic.pdf`.

[2] `gedit` mag worden vervangen met uw teksteditor naar keuze.

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.

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 en matplotlib. We kijken ook kort naar iPython, maar iPython is niet nodig om alle programmeeropdrachten te kunnen maken. Als je zelf je Linux machine beheert zijn deze uitbreidingen misschien nog niet geïnstalleerd. Met de package manager (bijv. "apt-get" of "yum") kun 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 `pip`. Download `get-pip.py` vanaf `https://pip.pypa.io/en/stable/installing/`. Daarna moet je `pip` en de iPython packages installeren:

```
python get-pip.py --user  # In de directory waar je get-pip.py hebt gedownload
$HOME/Library/Python/2.7/bin/pip install --user ipython
$HOME/Library/Python/2.7/bin/ipython
```

Plaats eventueel `$HOME/Library/Python/2.7/bin` in je `PATH` (zie `https://coolestguidesontheplanet.com/add-shell-path-osx/`). Wanneer je ook de "web notebooks" wilt installeren, herhaal het `pip install` commando met `jupyter` in plaats van `ipython`.

Er zijn ook andere manieren om iPython op een Mac te installeren, bijvoorbeeld door de Anaconda distributie te installeren (`https://www.continuum.io/download`, kies voor de Python 2.7 versie) of gebruik te maken van MacPorts (`http://www.macports.org/`).

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 Canopy Express (`https://www.enthought.com/canopy-subscriptions/`) en Python(x, y) (`http://python-xy.github.io/`).

2 Basisconcepten

In dit hoofdstuk zullen de basisconcepten van de programmeertaal Python worden uitgelegd aan de hand van eenvoudige voorbeelden. Om te beginnen, bekijken we een eerste voorbeeldprogramma:

```
1 # dit is een simpel programma
2 getal = 42  # een variabele declareren en initialiseren
3 print "Geef een geheel getal ..",
4 getal = int(raw_input())
5 print "Kwadraat is:", getal * getal
```

2.1 Commentaar

Als eerste regel in dit programma zien we een commentaarregel. Dergelijke commentaarregels beginnen met een hekje #. De interpreter slaat deze regels over bij het inlezen van het programma. Commentaarregels hebben dus geen invloed op de uitvoering van het programma. Deze regels zijn alleen opgenomen voor de menselijke lezer: de programmeur zelf of anderen. Commentaar in programmacode wordt bijvoorbeeld gebruikt als geheugensteuntje of om uit te leggen wat voor beslissingen er zijn gemaakt bij het schrijven van de code. Een andere handige toepassing: het tijdelijk "uitzetten" van code zodat deze wordt overgeslagen door de interpreter.

Commentaar mag ook worden toegevoegd aan het einde van een regel, zie regel 2 in het voorbeeldprogramma. Alles dat volgt na het hekje (tot het einde van die regel) zal worden overgeslagen. Als je meerdere regels commentaar wilt, moet elke regel met een hekje beginnen:

```
#
# Dit zijn
# meerdere regels
# commentaar
#
```

Voor het gebruik van commentaar in de programma's zoals die voor het vak Programmeermethoden NA gemaakt moeten worden zijn richtlijnen opgesteld. Deze zijn terug te vinden op de website van het vak.

2.2 Variabelen

Variabelen spelen een hoofdrol in programmeertalen. Een variabele is een klein stukje in het (RAM) geheugen dat gereserveerd wordt voor het opslaan van een bepaald type data. In Python geldt de regel dat voordat een variabele kan worden gebruikt in een expressie (en dus wordt uitgelezen) er altijd eerst een waarde aan moet zijn toegekend. Indien dit niet gebeurt, dan zal Python de variabele niet herkennen bij het maken van een berekening en zal er een `NameError` volgen.

In het volgende voorbeeld worden drie variabelen aangemaakt en geïnitialiseerd, waarin in elk van deze een ander type data wordt opgeslagen:

```
geheel_getal = 30
komma_getal = 543.2146
letters = "hello world"
```

In Sectie 3.2 komen we nog uitgebreid op variabelen, andere types en conversies tussen types terug.

2.3 Condities

Een van de belangrijkste mogelijkheden in een programma is het uitvoeren van verschillende stukken code afhankelijk van een bepaalde conditie of test. Denk hierbij bijvoorbeeld aan: Als `x` groter is dan 0 dan is `x` positief, anders is `x` 0 of negatief. Bovenstaande conditie zou er in Python bijvoorbeeld als volgt uit kunnen zien:

```
if x > 0:
    print "x is positief"
else:
    print "x is 0 of negatief"
```

Als de gegeven conditie bij `if` waar is, zal het eerste ingesprongen codeblok worden uitgevoerd[3]. Als deze conditie niet waar is, wordt het "else-blok" uitgevoerd.

Er kunnen ook meerdere gevallen worden getest, dit gebeurt met "else if" (als niet, dan ...) wat in Python wordt geschreven als `elif`:

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

Merk op dat je `elif` dus mag weglaten als je deze niet nodig hebt. Dit geldt ook voor `else`! De conditie kan ook bestaan uit meerdere tests of "eisen", bijvoorbeeld:

```
if x > 0 and x < 25:
```

Andere testen die kunnen worden gebruikt zijn:

- de gelijkheidstest, "is gelijk aan": ==,
- ongelijkheidstest, "is niet gelijk aan": !=,
- kleiner of gelijk aan: <=,
- groter of gelijk aan: >=.

Bij het maken van meervoudige testen maken we gebruik van Booleaanse expressies. We zagen al de Booleaanse operator `and`, ook is er `or` en `not`. 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,
# wel moet je de regels afsluiten met een backslash: \
if y >= 3 and (x == 4 or x == 5) and \
   z == 12 and (q >= 10 or q <= -10):  ...
```

In tegenstelling tot vele andere programmeertalen, kan een meervoudige conditie zoals `0.0 <= x <= 1.0` in Python gewoon op deze manier worden geformuleerd. Een andere manier om dit te doen zou zijn:

```
if 0.0 <= x and x <= 1.0:
```

Tenslotte is het belangrijk om je ervan bewust te zijn dat bij een conditie met `and` de tweede test niet meer wordt gedaan als de eerste test al uitsluitsel geeft. De volgende code geeft dus geen problemen:

[3] Inspringen is zeer belangrijk en moet consistent gebeuren, zie ook de Sectie 5.3.

```
if x != 0 and 1.0 / x == 0.5:
```

als x inderdaad niet gelijk is aan nul, dan wordt de tweede test gedaan. Dus, als de tweede test wordt uitgevoerd, dan zal x gegarandeerd niet gelijk zijn aan nul, waardoor er geen deling door nul zal optreden. Dit principe heeft "short-circuiting" of "lazy evaluation".

2.4 Gereserveerde woorden

Programmeertalen kennen altijd een set van gereserveerde woorden, in het Engels ook wel "keywords" genoemd. Deze woorden hebben in de programmeertaal een speciale betekenis. Derhalve mogen deze woorden niet worden gebruikt als naam voor een variabele of functie. De gereserveerde woorden in Python zijn:

```
and       as      assert   break  class
continue  def     del      elif   else
except    exec    finally  for    from
global    if      import   in     is
lambda    not     or       pass   print
raise     return  try      while  with
yield
```

2.5 Samenvatting basisconcepten

Ter afronding van dit hoofdstuk bekijken we een compleet voorbeeldprogramma waarin alle tot nu toe besproken concepten voorkomen:

```
1  # Dit is een regel met commentaar ...
2  import math  # voor de "pi" constante
3  print "Geef straal, daarna Enter ..",
4  straal = float(raw_input())
5  if straal > 0:
6      print "Oppervlakte:",
7      print math.pi * straal * straal
8  else:
9      print "Niet zo negatief ..."
10 print "Einde van dit programma."
```

De eerste regel van het programma is een commentaarregel, aangeduid met het hekje, dat commentaar bevat voor de menselijke lezers van het programma. In de tweede regel wordt aangegeven dat er zaken uit de module (bibliotheek) `math` zullen worden gebruikt. In de derde regel wordt een vraag voor de gebruiker op het scherm gezet. Alles dat tussen de dubbele aanhalingstekens staat wordt letterlijk op het scherm gezet (de aanhalingstekens zelf dus niet). De komma aan het einde van het `print`-statement zorgt ervoor dat er na deze uitvoer geen nieuwe regel wordt begonnen. Vervolgens wordt er in de vierde regel gewacht tot de gebruiker een antwoord invult (`raw_input()`). Dit antwoord wordt omgezet naar een kommagetal en opgeslagen in de variabele `straal`. Vervolgens komen we op een keuzemoment bij het `if`-statement. In het geval de waarde opgeslagen in de variabele `straal` groter is dan 0, wordt het ingesprongen `if`-blok uitgevoerd. Let erop dat het gehele `if`-blok met hetzelfde aantal spaties (of tabs) moet worden ingesprongen, anders volgt een `IndentationError`. In het blok wordt de oppervlakte van een cirkel met de gegeven straal uitgerekend en op het scherm gezet. In alle andere gevallen wordt het `else`-blok uitgevoerd: de tekst "Niet zo negatief ..."' wordt op het scherm gezet. Tot slot wordt nog een tekst afgedrukt en zijn we bij het einde van het programma aangekomen.

Denk er tenslotte aan dat een enkele = in Python een toekenning (assignment) is en niet een test op gelijkheid. Gelukkig wordt het volgende programma door de Python interpreter geweigerd:

```
x = 0
if x = 0:
    print "x is nul"
```

De interpreter geeft aan dat er een syntax error is bij het =-teken in het `if`-statement. Om te testen op gelijkheid moet een dubbele = worden gebruikt, dus == en in het voorbeeld `if x == 0:`.

3 Variabelen en getallen

In dit hoofdstuk gaan we uitgebreid bekijken hoe er kan worden gewerkt met variabelen en hoe verschillende elementaire bewerkingen kunnen worden uitgevoerd met getallen. Er bestaan verschillende manieren om getallen in een computer op te slaan: we spreken van verschillende "types". Aan het converteren, afdrukken en werken met deze verschillende types zitten een aantal haken en ogen. Deze zullen hier worden behandeld. We starten met een wat uitgebreidere omschrijving van het toekenningsstatement.

3.1 Toekenningsstatement

Zoals we al eerder zagen, worden in Python variabelen 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 al 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
```

3.2 Werken met verschillende types

Elke variabele in Python heeft een type. Het type van de variabele wordt bepaald aan de hand van het soort data dat aan een naam wordt toegekend in een toekenningsstatement. We bespreken nu eerst de belangrijkste types.

Gehele getallen. Gehele getallen zijn in de meeste gevallen van het type `int` (integer). Meestal zijn dit 4 of 8 bytes. Hierdoor is het bereik -2^{31} tot $2^{31}-1$, ofwel van ongeveer -2 miljard tot 2 miljard, of -2^{63} tot $2^{63}-1$ ofwel van ongeveer -9 triljoen tot 9 triljoen.

Python heeft ook een type voor hele grote gehele getallen. Deze noemen we long integers en hebben als 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.

We kunnen reële getallen omzetten naar een geheel getal met behulp van een typeconversie, de functie `int`:

```
>>> geheel = int(3.14)
>>> geheel
3
```

Reële getallen. Reële getallen worden **bij benadering** opgeslagen als zogenaamde floating-point getallen (type `float`). In Python zijn floating-point getallen altijd double precision en dat maakt ze 8 bytes groot. Wanneer een floating-point getal wordt omgezet naar een integer, zoals we hierboven zagen, dan wordt het getal altijd naar beneden afgerond (de cijfers achter de komma worden weggegooid). Om af te ronden op een natuurlijke manier kun je gebruik maken van de functie `round`:

```
>>> getal = 4.75
>>> int(getal)
4
>>> int(round(getal))
5
```

Voorts is het belangrijk om je te realiseren dat een floating-point getal een benadering is van een reëel getal. Hierdoor geeft een test `if x == y:` met `x` en `y` een floating-point getal niet altijd het resultaat dat je verwacht. Zelfs een test als

```
if abs(x - y) < epsilon:
```

met `epsilon` bijvoorbeeld 0.00001 voldoet niet. Dus moet er iets als volgt worden gebruikt:

```
if abs(x - y) < max(abs(x), abs(y)) * epsilon:
```

Vanaf Sectie 16 zullen we gaan werken met NumPy. Hierin zit een dergelijke test al ingebouwd: `np.isclose()`. In Sectie 4 bekijken we hoe we het afdrukken van floating-point getallen kunnen instellen.

Complexe getallen. Python heeft ingebouwde ondersteuning voor complexe getallen! Het type is `complex`. Voorbeeld:

```
>>> z = 6+9j    # "j" is de imaginaire eenheid, in de wiskunde i geheten
>>> 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)
```

Boolean. Python heeft een ingebouwd type `bool` voor het representeren van Boolean waarden. Deze kunnen de waarde `True` of `False` aannemen.

Strings. Karakters moeten worden opgeslagen met behulp van "strings". Een string is een rijtje van karakters (een "string of characters" in het Engels). Er is een speciaal type voor strings: `str`. Van strings kan de lengte worden bepaald en de gelijkheidstest werkt zoals je zou verwachten:

```
>>> woord = "De."
>>> len(woord)
3
>>> woord == "test"   # Vergelijken van strings kan gewoon met de == operator
False
>>> woord == "De."
True
```

Gegeven een string, dan is het mogelijk om een gedeelte van de string, of een individueel karakter, uit te lezen. Dit kan worden gedaan door tussen blokhaken de positie (index of subscript) van het uit te lezen karakter op te geven. Het uitlezen van een substring kan worden gedaan door zowel een start als eindindex op te geven. Dit noemen we "slicing" en hier komen we nog uitgebreid op terug in Sectie 10.4. Ook negatieve indexen zijn 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 Sectie 3.5. Het volgende voorbeeld demonstreert de verschillende manieren van indexing:

```
>>> 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'
```

3.3 Typeconversies

Voor elke variabele houdt Python het type bij. We kunnen het type van de variabele opvragen met de functie `type`:

```
>>> 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. Stel nu we hebben het volgende voorbeeld:

```
>>> geheel_getal = 120
>>> komma_getal = 35.34
>>> print geheel_getal, komma_getal
120 35.34
>>> geheel_getal = komma_getal
>>> geheel_getal
35.34
>>> type(geheel_getal)
<type 'float'>
```

We zien hier dat wanneer we de waarde van het kommagetal toekennen aan de variabele `geheel_getal`, het type van deze variabele mee verandert naar `float`. In sommige gevallen willen we echter een bestaande floating-point waarde opslaan als geheel getal. Om dit te bereiken, moeten we Python dit expliciet vertellen met behulp van een typeconversie:

```
>>> geheel_getal = int(komma_getal)
>>> print geheel_getal
35
```

Hier maken we gebruik van de functie `int()` om van een floating-point een geheel getal te maken[4]. Ook voor de andere numerieke types zijn er ingebouwde conversiefuncties: `int()`, `long()`, `complex()`. Het is zelfs mogelijk om strings die een getal bevatten om te zetten naar een numeriek type: `float("3.14")` resulteert in een float type met de waarde 3.14. Niet elke string wordt zomaar geaccepteerd. Zo werkt `int("3.14")` bijvoorbeeld niet. Python weigert om een string die geen integerwaarde bevat om te zetten naar een integer. We ontvangen in zo'n geval een `ValueError`. 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 ...
```

3.4 Rekenen

Het rekenen met getallen werkt met de gebruikelijke rekenkundige operatoren (de operator * wordt gebruikt voor vermenigvuldiging). Een aantal voorbeelden:

```
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
                 # hier staat eigenlijk hetzelfde als b = b + 1
a -= 1           # dit is hetzelfde als a = a - 1
getal += a       # getal = getal + a
a = 19 / 7       # Integer deling: a wordt 2
b = 19 % 7       # Rest bij integerdeling (modulo): b wordt 5
q = (6+9j) + (4+2j)  # Optelling complexe getallen: resultaat is (10+11j)
q = (6+9j) * 2       # Resultaat: (12+18j)
```

[4]Merk nogmaals op dat er niet netjes wordt afgerond, maar alles achter de komma gewoon wordt weggegooid, zoals we al zagen in Sectie 3.2.

Een aantal zaken werken echter anders dan je zou verwachten. In het voorbeeld met de integerdeling zagen we al dat dit resulteert in een geheel getal, waar je wellicht een reëel getal zou verwachten. 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
```

Wat gebeurt er nu als er een operatie wordt gespecificeerd op twee verschillende types, 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`).

3.5 String manipulaties

Op strings kunnen verschillende operaties worden uitgevoerd. 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). Deze worden in het volgende voorbeeld gedemonstreerd:

```
>>> 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 de 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 complexere expressies bouwen door de operatoren te combineren:

```
>>> a = "aaa"
>>> b = "bbb"
```

```
>>> a * 3
'aaaaaaaaa'
>>> (a + b) * 3
'aaabbbaaabbbaaabbb'
>>> a * 3 + b * 3
'aaaaaaaaabbbbbbbbb'
```

3.6 ASCII waarden

Computergeheugen bestaat louter uit enen en nullen. Om karakters op te kunnen slaan in het geheugen, moeten deze dus worden omgezet in getallen die vervolgens in binaire vorm worden opgeslagen. Deze omzetting wordt gedaan aan de hand van de *ASCII-tabel*. In deze tabel is voor elk karakter een getal gedefinieerd. In Python bestaan er functies `ord()` en `chr()` om een karakter om te zetten naar een ASCII-waarde en vice versa:

```
>>> ord('s')
115
>>> chr(115)
's'
>>> ord('S')
83
>>> chr(83)
'S'
```

Let erop dat de functie `ord()` alleen maar strings bestaande uit 1 karakter accepteert.

4 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
```

Met behulp van uitvoerformattering ("output formatting") is het mogelijk om tot in detail in te stellen hoe de resultaten op het scherm moeten verschijnen. Voor reële getallen is het bijvoorbeeld mogelijk om aan te geven hoe breed de uitvoer moet zijn en hoeveel cijfers er na de komma moeten volgen. We bekijken hier de meest moderne methode voor het instellen van uitvoerformattering in Python. De oudere manier wordt besproken in een apart kader en zal diegenen die ervaring hebben met C/C++ bekend voorkomen.

Uitvoerformattering is een combinatie van een format string en een rijtje van argumenten voor de format string. In de format string geef je met een specifieke notatie aan waar je wilt dat

een variabele wordt afgedrukt en op welke manier deze moet worden afgedrukt. Deze aanduiding is een conversiespecificatie en wordt in de Python documentatie ook wel een *format field* genoemd. De format fields zijn te herkennen in de format string aan de accolades. 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
```

In de format string komen twee format fields voor. Het eerste getal tussen de accolades geeft het hoeveelste element aan uit het rijtje van argumenten dat moet worden afdrukt. In de eerste format string wordt de variabele `a` afgedrukt voor het eerste format field en variabele `b` voor het tweede. In de tweede format string worden variabelen `a` en `c` afgedrukt. In plaats van getallen, mag je hier ook gebruik maken van namen:

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

Ook mogen de getallen tussen de accolades worden weggelaten. In dit geval worden opeenvolgend de elementen uit het rijtje van argumenten genomen:

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

Verdere controle over de manier waarop een variabele wordt afgedrukt kan worden uitgeoefend door een dubbele punt toe te voegen met daarachter een verdere "uitvoerspecificatie". Zo kan na de dubbele punt de gewenste minimum breedte van het veld en ook een type worden aangeven:

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

Het eerste format field zal de variabele `a` afdrukken in een veld van minimaal 6 spaties breed. De letter `d` in het format field geeft aan dat de waarde moet worden geconverteerd naar een geheel getal voordat deze wordt afgedrukt. Het tweede format field drukt wederom `a` af in een veld van minimaal 6 spaties breed, maar door het conversiekarakter `f` wordt de waarde eerst omgezet naar een waarde van het type `float`, voordat deze op het scherm wordt gezet. De precisie, het aantal cijfers achter de komma, is standaard 6. Hierdoor past het getal niet binnen de minimum breedte van 6 spaties en wordt er meer ruimte gebruikt. Het derde format field drukt `c` af in een veld van minimaal 8 spaties breed en gebruikt 4 cijfers achter de komma. We zien dat dit wel past in een veld van 8 spaties breed, waardoor er spaties worden ingevoegd. Tenslotte drukt het vierde format field de string `"testje"` af in een veld van 20 spaties breed.

We kunnen ook aangeven of velden links of rechts moeten worden uitgelijnd, of moeten worden gecentreerd:

```
>>> print "{0:>20s}".format("testje")
              testje
>>> print "{0:^20}".format("testje")
       testje
>>> print "|{0:<8d}|{0:^8d}|{0:>8d}|".format(123)
|123     |  123   |     123|
```

Tevens zijn er ook conversiekarakters voor wetenschappelijke notatie ("scientific notation") en percentages. Merk op dat bij percentages er automatisch met 100 wordt vermenigvuldigd:

```
>>> print "{0:6e} {0:.2%}".format(0.00123453455)
1.234535e-03 0.12%
```

Ter afsluiting, het resultaat van een formattering kan gewoon worden opgeslagen in een string:

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

Zelfs alleen de format string kan als string worden opgeslagen, om verschillende keren te worden hergebruikt. Binnen deze formatteringstaal is er nog veel meer mogelijk, voor alle details zie de documentatie: `https://docs.python.org/2/library/string.html#formatspec`.

C-stijl uitvoerformattering **(+)**

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 rijtje 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 oude stijl 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'
```

5 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. Het `if`-statement hebben we al behandeld in Sectie 2.3. De `for`- en `while`-statements worden gebruikt voor het vormen van "loops". Een loop is in feite het meerdere malen herhalen van een aantal statements. In het algemeen wordt er een blok van statements aangegeven dat moet worden herhaald en een specificatie hoe vaak dit blok moet worden herhaald.

In principe kunnen alle `for`-loops ook als `while`-loops worden geschreven en vice versa. Het is gebruikelijk om een `for`-loop te gebruiken voor situaties waarbij het aantal herhalingen vast ligt ("drie maal bellen", "zes eieren breken en toevoegen"). De `while`-loop wordt gebruikt als het aantal herhalingen van te voren niet bekend is ("net zolang zeuren totdat", "kloppen totdat het beslag glad is") of lastig te bepalen. We zullen nu beide soorten loops bespreken.

5.1 For-statement

In Python wordt een `for`-loop uitgedrukt als een iteratie over een rij van elementen. 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 rij aannemen. Daarnaast volgt het gereserveerd woord `in` gevolgd door de te itereren rij. Afsluitend een dubbele punt met daaronder een *correct ingesprongen* blok met statements die tot de loop behoren en dus moeten worden herhaald. In het algemeen:

```
for variabele in rij:
    statementblok
```

Laten we een eerste programma bekijken dat de eerste `10` getallen met hun kwadraat afdrukt. Let op, computers beginnen met tellen bij 0:

```
for i in range(10):
    print i, "--", i * i
```

In dit voorbeeld is `i` de iteratievariabele. De eerste keer dat de loop wordt uitgevoerd, heeft `i` de waarde 0, daarna 1, enzovoort. De laatste keer heeft `i` de waarde 9. `range()` is een speciale functie die wordt gebruikt om getallenreeksen te maken. Het meeste simpele gebruik van `range()` geeft ons een lijst van getallen vanaf 0. Je kunt ook een ander startpunt opgegeven, of zelfs een stapwaarde: `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. Tevens kan `range()` buiten een `for`-loop worden gebruikt. Een aantal voorbeelden:

```
>>> 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 dit 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 >>>. In het volgende voorbeeld gebruiken we de uitgebreide versie van `range()` om de getallen 3 tot en met 17 te bezoeken met een stapgrootte van 2:

```
for i in range(3, 18, 2):
    print i, "-",
```

Naast getallenreeksen, kunnen we ook rijen met andere soorten data bezoeken. Deze rijen worden in het algemeen opgeschreven als lijsten, waarop we nog in zullen gaan in Sectie 10[5]. Zie voorbeeld de volgende voorbeelden:

```
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
```

We kunnen ook "dubbele" `for`-loops schrijven, we spreken dan van het nesten van loops. Dit gebeurt in het volgende programma, dat de eerste `i` veelvouden van getal `i` op het scherm zet:

```
for i in range(1, 6):
    print "{0}:".format(i),
    for j in range(1, i+1):
        print i * j,
    print                    # print zonder argument geeft een newline
```

5.2 While-statement

Het `while`-statement wordt in het algemeen gebruikt in het geval het aantal herhalingen van te voren niet bekend is. Anders dan het `for`-statement wordt er geen lijst afgelopen – we kunnen een dergelijke lijst dan ook niet van te voren bepalen. Het gevolg hiervan is dat je in vele gevallen zelf een iteratorvariabele moet bijhouden en ophogen. Bijvoorbeeld in het volgende programma, dat overigens equivalent is aan de `for`-loop besproken aan het begin van dit hoofdstuk:

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

Het volgende voorbeeld, waarbij het aantal herhalingen van te voren niet bekend is, is typisch geschikt voor een `while`-loop:

```
x = 1234
while x != 1:
    if x % 2 == 0:
        x = x / 2
    else:
        x = 3 * x + 1
```

Tot op heden is het nog niet bekend of deze `while`-loop voor ieder positief geheel begingetal `x` stopt. En indien het stopt, is het nog maar de vraag wat het aantal doorgangen door de test is geweest. Het probleem staat onder meer bekend als het Syracuse-probleem, het Collatz-probleem of het $3x + 1$-vermoeden. Ook in het volgende programma:

[5]Merk daarnaast op dat het resultaat van de functie `range()` in principe weer een lijst is met de gevraagde getallen.

```
x = 1
while x < 1000:
    x = 2 * x
```

is het eenvoudig te zien dat het ooit zal stoppen en dat na afloop variabele `x` de waarde `1024` heeft, maar het aantal doorgangen is in het algemeen (met `n` in plaats van `1000`), niet voor elke situatie vooraf vast te stellen. Een `while`-loop geniet hier dus de voorkeur.

Tenslotte, in tegenstelling tot verschillende andere programmeertalen kent Python geen `do-while` statement.

5.3 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 "IndentationError", 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.

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.

Tenslotte nog een opmerking over lege blokken. Als we een blok willen hebben zonder statement, maar we moeten wel inspringen, hoe krijgen we dit dan voor elkaar? Python heeft hier een speciaal keyword voor: `pass`. `pass` is een statement dat niets doet, een "empty statement", en 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"
```

6 Functies

Zodra programma's beginnen te groeien, begint het vaak voor te komen dat je een bepaald stuk code op verschillende plaatsen wilt gebruiken. Voorbeelden zijn het berekenen van een faculteit en het netjes op het scherm zetten van een matrix. Natuurlijk gaan we deze code niet meerdere malen in ons programma opnemen. We maken er liever een klein subprogramma van en roepen dan dit kleine subprogramma vanaf verschillende plekken in het programma aan. Deze kleine subprogramma's worden *functies* en *procedures* genoemd.

Een functie voert een aantal berekeningen uit en levert een resultaat op. Bijvoorbeeld het uitrekenen van een kwadraat, faculteit of de oppervlakte van een cirkel. Een procedure voert ook een aantal berekeningen of handelingen uit, maar heeft verder geen resultaat. Voorbeeld: het op het scherm zetten van een matrix.

Voordat een functie in een Python-programma kan worden gebruikt, moet deze worden gedeclareerd. De declaratie bestaat uit het opgeven van de naam van de functie en eventuele ar-

gumenten (een functie zonder argumenten is ook toegestaan). Globaal ziet de definitie van een functie er 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. Bijvoorbeeld:

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

de functie wordt gebruikt als volgt:

```
zeg_hallo()
```

Het "gebruiken" van een functie noemen we het aanroepen van een functie of een functieaanroep (Engels: "function call"). Omdat er hier geen (expliciet) resultaat wordt opgeleverd, is er eigenlijk sprake van een "procedure". Een resultaat opleveren en teruggeven gaat met het keyword `return`. Achter `return` volgt een naam van een variabele of een expressie. De waarde hiervan wordt de returnwaarde van de functie. Een voorbeeld:

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

# Aan te roepen als volgt:
resultaat1 = telop(10, 41)
resultaat2 = telop(23, 64)
```

Wat gebeurt er nu precies wanneer we een functie aanroepen? Er wordt onthouden waar we zijn met het uitvoeren van het programma. Vervolgens wordt er gesprongen naar het begin van de functie. Zodra we in die functie een regel met het `return` keyword tegenkomen, springen we terug naar waar we waren gebleven – dat hadden we immers onthouden. Functie-aanroepen kunnen ook worden genest: vanuit een aangeroepen functie kun je weer een andere functie aanroepen. De plekken waarnaar we moeten terugspringen worden op een stapel gezet. De locatie waar we het "eerst" naar moeten terugspringen (dus de locatie van de meest recente functieaanroeper) staat altijd bovenaan de stapel.

Merk op dat het ook is toegestaan om vanuit een functie een aanroep te doen naar dezelfde functie. Bijvoorbeeld, we roepen vanuit onze functie `telop` nogmaals de functie `telop` aan. Er wordt dan onthouden waar we waren in `telop` en we roepen nogmaals `telop` aan. Zodra we een `return` tegenkomen, springen we terug naar waar we waren gebleven — dat was ook in `telop`! NB: Het is dus *niet* zo dat je door dezelfde functie aan te roepen de uitvoering van de functie opnieuw laat beginnen (en vergeet waar je mee bezig was). De techniek van een functie zichzelf aan te laten roepen wordt gebruikt om "recursie" te implementeren, wij zullen dit echter in deze cursus niet behandelen.

In Python hebben functies 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.

De argumenten van een functie worden ook wel *parameters* genoemd. In de functie `telop` zijn `a` en `b` *formele parameters*. Deze formele parameters krijgen als startwaarde dezelfde waarde als de corresponderende argument in de functieaanroep. De argumenten in de functieaanroep, zoals `10`

en 41 in het eerste voorbeeld, worden *actuele parameters* genoemd. In `telop` zijn `a`, `b` en `c` lokale variabelen. Hun scope is beperkt tot de functie `telop`. Dus zodra deze functie is afgelopen, zijn de namen van de lokale variabelen niet meer beschikbaar.

We bekijken nu functiedefinities waarbij er meerdere returnwaarden worden teruggegeven:

```
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
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
```

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. Tuples zijn dus heel handig voor het teruggeven van meerdere resultaten!

Bij het gebruik van functies is het belangrijk dat de argumenten geen type hebben en we geen compiler hebben die nakijkt of de argumenten in functieaanroepen van het juiste type zijn. Bijvoorbeeld `telop(10, "hallo")` gaat fout!

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.

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

In de Informatica heeft men het bij het doorgeven van functieargumenten vaak over call-by-value en call-by-reference. Hoe zit dat eigenlijk 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. `a` en `b` zijn 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
```

Wanneer we een lijst (zie Hoofdstuk 10) 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 nadoen 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/.

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 kunt altijd netjes commentaar boven aan de functie zetten. Ook kun 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 kun 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 kun je in de interpreter hulp krijgen over je functie
help(teken_lijn)
```

7 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. Een functie kan alleen worden aangeroepen **nadat** deze is gedefinieerd. De definitie van een functie moet dus altijd boven het eerste gebruik van deze functie staan.

Het is in Python niet verplicht om alle code binnen een functie te plaatsen. Vaak is het wel netjes om een "hoofd"-functie te maken waar het programma begint. Deze functie wordt dan aangeroepen vanuit de globale code. Het is conventie om een dergelijke hoofdfunctie de naam "main" te geven. Een nette manier om een Python programma te structureren 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.

8 Globale variabelen

Bij onze discussie over functies zagen we dat de formele parameters van een functie lokale variabelen in die functie zijn. Een lokale variabele bestaat alleen binnen die functie en daarbuiten niet. Er bestaan ook *globale variabelen* die buiten een functie worden gedefinieerd en daarom "overal" in het programma (en dus in alle functies) bestaan. Wanneer Python de naam van een variabele ziet, kijkt Python eerst of er een lokale variabele met deze naam bestaat (voorkomt in de lokale scope), zo niet, dan wordt er gekeken of er een globale variabele bestaat met die naam. Een voorbeeld is als volgt:

```
globaal = 2345

def drukaf():
    print globaal


print globaal  # 2345
drukaf()       # 2345
globaal = 12
drukaf()       # 12
```

We moeten goed opletten wanneer we gaan werken met globale en lokale variabelen die dezelfde naam hebben. Python hanteert hier de volgende regel: wanneer binnen een functie alleen maar uit een variabele wordt gelezen die niet als lokale variabele bestaat, wordt er gekeken of er een globale variabele is met die naam. Wordt er ergens binnen een functie geschreven naar een variabele, dan wordt er *altijd* een lokale variabele gemaakt.

Een voorbeeld van het eerste geval zagen we al: in `drukaf` wordt er alleen gelezen uit `globaal`. `globaal` bestaat niet als lokale variabele en daarom wordt er naar een globale variabele gezocht. Laten we nu een voorbeeld voor het tweede geval bekijken:

```
x = 195

def test(q):
    # Let op: omdat er wordt geschreven naar 'x' in deze functie,
    # wordt altijd een *lokale* variabele x gemaakt.
    x = q
    print x

print x     # 195
test(2314)  # 2314
print x     # 195
```

Omdat er wordt geschreven naar `x` binnen de functie `test`, wordt er altijd een lokale variabele `x` gemaakt die voorrang krijgt boven de globale variabele. Voor de regel maakt het niet uit *waar* in de functie er naar de variabele wordt geschreven, als er maar een keer naar de variabele wordt geschreven. Hierdoor is de volgende code niet geldig:

```
x = 195

def test(q):
    print x
    x = q
```

Er wordt ergens in `test` geschreven naar `x` waardoor dit een lokale variabele zal zijn. Vervolgens wordt de lokale variabele `x` al gelezen voordat er een waarde aan is toegekend.

Wat nu als we binnen een functie juist wel aan een globale variabele een waarde willen toekennen. Dus dat in ons voorbeeld de `x` binnen `test` niet automatisch een nieuwe lokale variabele wordt? Hiervoor heeft Python het keyword `global` dat moet verschijnen als eerste regel in een functie:

```
x = 195

def test(q):
    global x
    x = q
    print x

print x      # 195
test(2314)   # 2314
print x      # 2314
```

Hiermee geven we aan het begin van de functie aan welke variabelen als globale variabelen moeten worden gezien.

In de meeste gevallen heb je, met uitzondering van het definiëren van constanten bovenaan het programma, geen globale variabelen nodig. Merk op dat omdat aan constanten nooit een nieuwe waarde wordt toegekend vanuit functies, je hiermee nooit in de problemen komt en ook geen `global` keyword nodig hebt.

9 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.

Het werken met bestanden in Python gaat aan de hand van 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 is een uitgebreide manier om het inlezen van een bestand op te schrijven. De `while`-loop wordt gebruikt om te kijken of we bij het einde van het bestand zijn aangekomen (EOF: end-of-file), in welk geval we willen stoppen met het herhalen van de loop. 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. Of, als laatste, nog mooier zodat we nooit `close` kunnen vergeten:

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

In dit geval wordt `close` eigenlijk automatisch aangeroepen wanneer we het `with`-blok afsluiten door te stoppen met inspringen.

Als je een bestand karakter voor karakter wilt uitlezen, dan kan dat ook. De aanroep `f.read(1)` leest precies 1 byte (1 karakter) uit een bestand en geeft het resultaat terug als string (bestaande uit 1 karakter). Het volgende programma kopieert een bestand "invoer.txt" karakter-voor-karakter naar een bestand "uitvoer.txt":

```
invoer = open("invoer.txt", "r")
uitvoer = open("uitvoer.txt", "w")

kar = invoer.read(1)
while kar:
    uitvoer.write(kar)
    kar = invoer.read(1)

invoer.close()
uitvoer.close()
```

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 waarin iedere regel drie gehele getallen bevat, 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, kun 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
```

10 Lijsten

Tot nu toe hebben we alleen maar gewerkt met variabelen die een enkele waarde konden vasthouden. Het is ook mogelijk om rijen van variabelen te maken. Hiervoor gebruiken we lijsten, type `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 in de lijst hoeven niet hetzelfde te zijn. Zoals we zullen zien staat de grootte van de lijst niet vast, je kunt eenvoudig elementen toevoegen en verwijderen aan de lijst.

10.1 Lijsten maken en indexeren

Lijsten kunnen worden aangemaakt met blokhaken, zoals is te zien in het volgende voorbeeld:

```
a = [11, 12, 13, 14, 15]
b = [1.0, 2.5, 3.4]
c = [1, "test", 4.5, False]     # Verschillende typen variabelen
```

Na het maken van een lijst, kunnen de verschillende vakjes als volgt benaderd worden:

```
print a[2]   # Geeft 13
print a[0]   # Geeft 11
```

Het getal `2` tussen de blokhaken wordt de *index* of *subscript* genoemd. Deze index geeft aan welk element van de lijst je wilt benaderen. We beginnen met tellen bij `0`. Wanneer je een index gebruikt die niet bestaat, resulteert dit in een "IndexError". Je kunt ook het resultaat van een toekenning laten wegschrijven in een vakje, de vorige waarde in dat vakje wordt dan overschreven. Met de functie `len` kan de lengte (grootte) van de lijst worden opgevraagd. Het volgende voorbeeld vat dit samen:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7]
>>> len(a)
8
>>> a[6]
6
>>> a[4] = 'ha!'    # Element van de lijst overschrijven
>>> a
[0, 1, 2, 3, 'ha!', 5, 6, 7]
```

Lijsten worden veelvuldig gebruikt in combinatie met een loop. Stel we hebben een lijst bestaande uit integers en we willen elk element vermenigvuldigen met 5. Om dit voor elkaar te krijgen, schrijven we een loop die elk element van de lijst bezoekt, waarbij `i` steeds een lijst-index voorstelt:

```
a = range(10, 110, 10)
for i in range(len(a)):
    a[i] = a[i] * 5
```

Samenvattend zijn er vijf manieren om lijsten te initialiseren:

1. Direct initialiseren met verschillende elementen: `a = [11, 12, 13, 33, 44, 55, 66]`.
2. Op basis van een andere lijst: `a2 = list(a)`.
3. Middels een functieaanroep: `b = range(10, 110, 10)`.
4. n-keer hetzelfde element:

```
c = [0] * 10
d = [0 for i in range(10)]
e = [i ** 2 for i in range(10)]
```

In het laatste geval wordt `e` geïnitialiseerd als lijst met de waarden $0^2, 1^2, \ldots, 9^2$.

5. Lege lijst (lengte 0):

```
f = []
g = list()
```

10.2 Werken met lege lijsten

In plaats van lijsten direct met waarden te initialiseren, is het 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").

Om waarden toe te voegen aan de lijst kun je gebruik maken van de methoden *append* en *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 kunt 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(10, 15)         # [10, 11, 12, 13, 14]
>>> b.pop(2)                  # Geef en verwijder element op index 2
12
>>> del b[1]                  # Verwijder element op index 1
>>> del b[2]                  # LET OP: dit was b[3]!
>>> b
[10, 13]
```

10.3 Lijsten kopiëren

Het is belangrijk om te realiseren dat de toekenningsoperator geen kopie maakt wanneer deze wordt toegepast op een lijst [6]. Dit kan worden gezien in het volgende voorbeeld:

```
>>> a = [1, 2, 3, 4]
>>> b = a
>>> a.remove(2)
>>> a
[1, 3, 4]
>>> b
[1, 3, 4]
```

We zien in dit voorbeeld dat `a` en `b` naar dezelfde lijst wijzen. Wanneer `a` wordt veranderd, zien we dit terug in `b`. Soms wil je juist wel een kopie maken en deze onafhankelijk aanpassen. Hoe bereiken we dat? Hiervoor maken we een nieuwe lijst die we initialiseren aan de hand van de bestaande lijst:

```
>>> a = [1, 2, 3, 4]
>>> b = list(a)     # Maakt een nieuwe lijst: een kopie.
>>> a.remove(2)
>>> a
[1, 3, 4]
>>> b
[1, 2, 3, 4]
```

10.4 Uitgebreid indexeren: slicing

Bij strings zagen we dat we op dezelfde manier individuele karakters van een string kunnen uitlezen (maar niet veranderen). Substrings kunnen worden uitgelezen door een start- en eindindex op te geven. Deze methode heet "slicing" en werkt ook voor lijsten. De formele definitie van de specificatie van een "slice" is als volgt:

$$start : eind : stap$$

hier is $start$ 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 in het geval van lijsten 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. Voor strings wordt dit niet ondersteund, als je een string wilt aanpassen, dien je altijd een nieuwe string te maken. Een en ander wordt in het volgende voorbeeld gedemonstreerd:

```
>>> a = range(10, 110, 10)        # Lijst 10 t/m 100, stapgrootte 10
>>> a[2:5]
[30, 40, 50]
>>> a[2:]
[30, 40, 50, 60, 70, 80, 90, 100]
>>> a[:5]
```

[6] Dit heeft te maken met de wijze waarop functies worden geïmplementeerd. Wanneer een functie wordt aangeroepen, worden er voor de formele parameters lokale variabelen gemaakt waaraan de waarden van de actuele parameters worden toegekend. Stel de actuele parameter zou een grote lijst zou en de toekenningsoperator zou standaard kopiëren, dan zou de grote lijst bij elke functieaanroep worden gekopieerd, ook al is dit niet nodig!

```
[10, 20, 30, 40, 50]
>>> a[2:8:2]
[30, 50, 70]
>>> a[::2]              # Gehele lijst, stapgrootte 2
[10, 30, 50, 70, 90]
>>> a[::3]
[10, 40, 70, 100]
>>> 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
[]
```

10.5 Elementen zoeken en tellen

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, kun je gebruik maken van de methode `index`[7]. 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
```

10.6 Het nesten van lijsten

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-dimensionale arrays of matrices. De geneste lijsten mogen namelijk elke gewenste grootte aannemen. We komen later nog terug op het gebruik van echte arrays en matrices.

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

```
>>> 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__'
```

10.7 Eenvoudige functies met lijsten

Het is geen enkel probleem om een lijst als functieargument te gebruiken. Het volgende voorbeeld laat zien hoe we zelf een functie zouden kunnen schrijven om alle elementen van een lijst te sommeren en hoe deze functie moet worden aangeroepen:

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

# Aanroepen:
reeks = [4, 45, 5, 23, 4]
som = sommeer(reeks)
```

Merk hier op dat de lokale variabele `lijst` naar dezelfde lijst wijst als `reeks`, er wordt *geen* kopie van de lijst gemaakt. Zoals we hier lijsten als functieargument gebruiken, werkt dat zo ook voor dictionaries (zie later) en tuples.

We zien dat wanneer we lijsten combineren met `for`-loops we al een aantal simpele algoritmen kunnen implementeren. Hieronder geven we een voorbeeld van het bepalen van het grootste element in een lijst van integers:

```
def grootste(lijst):
    '''Bepalen grootste element in lijst'''
    grootste = 0
    for element in lijst:
        if element > grootste:
            grootste = element
    return grootste

# Roep de functie aan
a = [3, 5, 213, 65, 2, 562, 234]
print "Het grootste element is: ", grootste(a)
```

Alhoewel Python voor veel van dit soort problemen een ingebouwde functie heeft, is het implementeren van dergelijke algoritmen altijd een goede oefening. Voor de bovenstaande functie `sommeer` heeft Python een functie `sum` en voor `grootste` de functie `max`. In Hoofdstuk 17.3 bekijken we eenvoudige algoritmen voor het zoeken in en het sorteren van gegevens.

11 Tuples

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)
```

12 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[8].

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

[8]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.

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 dictionary en key 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 "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
```

13 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[9].

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

Wat nu wanneer we nog geen tuples hebben, maar twee aparte lijsten waar deze data in staat? Maak dan gebruik van de functie `zip`, deze maakt automatisch tuples gegeven twee of meer lijsten met data. Ook handig om coördinaatparen te vormen uit twee aparte lijsten met x- en y-coördinaten!

```
getallen = [1, 2, 3]
letters = ['a', 'b', 'c']
for g, l in zip(getallen, letters):
    print g, ",", l

horz = range(10, 20, 2)
vert = range(13, 23, 2)
for x, y in zip(horz, vert):
    print "({}, {})".format(x, y)
```

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

[9] 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.

```
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.
```

14 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. Het is in Python mogelijk om programma's te schrijven die uit 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 definiëren 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[10], **(2)** de bestandsnaam mag geen streepjes ("-") bevatten. De module `handig.py` zoals we hierboven hebben gebruikt ziet er als volgt uit:

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

```
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.

15 Object Georiënteerd Programmeren

De taal Python biedt ons een aantal ingebouwde standaard-types aan, zoals `int`, `float`, `bool`, enz. Het komt echter vaak voor dat een programmeur deze typen wil samenpakken tot één geheel of totaal nieuwe types met eigen mogelijkheden (functionaliteiten) wil aanmaken. Middels Object Georiënteerd Programmeren (OOP) is het mogelijk om programma's op te bouwen in de vorm van objecten die data opslaan en die kunnen worden gemanipuleerd. Er kunnen objecten worden aangemaakt die data opslaan met behulp van de ingebouwde standaard-types, of juist weer andere object-types, en welke kunnen worden gemanipuleerd door member-functies, of methoden, op een object aan te roepen.

Python biedt naast de ingebouwde standaard-types ook al verschillende andere types objecten aan. Twee voorbeelden zijn:

```
invoer = open("bestand.txt", "r")
...
letter = invoer.read(1)
...
invoer.close()

lijst = list()
lijst.append(143)
lijst.append(542)
```

In dit voorbeeld is `invoer` een variabele van type `file` en `lijst` van type `list`. We zeggen ook wel dat `invoer` en `lijst` *objecten*, of *instanties*, zijn van een bepaald type. Zo is `lijst` een instantie van het type `list`. Op een instantie kunnen allerlei acties worden uitgevoerd middels methoden die voor dat type zijn gedefinieerd. We zien hier de methode `read` voor objecten van het type `file` en `append` voor objecten van het type `list`.

Bij het object georiënteerd programmeren draait alles om het ontwerpen van dergelijke typen (of *klassen*, Engels *classes*) en het maken van instanties hiervan. Programma's worden in feite uitgedrukt als zijnde een reeks van manipulaties van objecten. Voordelen hiervan zijn dat deze manier van programmeren beter aansluit bij de denkwijze van de "echte wereld" en dat de zelf gedefinieerde klassen herbruikbaar zijn en de modulariteit verhogen. Kijk maar eens naar het

onderstaande voorbeeld, waarin objecten worden gebruikt om het bord, spel en een speelstuk te representeren:

```
def schaken():
    nieuwbord = Schaakbord()
    nieuwspel = Schaakspel()
    ditstuk = Schaakstuk()

    nieuwspel.start(nieuwbord)
    ...
    nieuwspel.zetstuk(ditstuk)
    ...

    if nieuwspel.afgelopen():
        print "einde spel"
```

Met behulp van een *class* is het mogelijk een geheel nieuwe klasse te maken. In feite is een *class* een blauwdruk voor een nieuw type. Vervolgens kunnen aan de hand van deze blauwdruk objecten worden geïnstantieerd van dit zelfgemaakte type. Binnen een klasse kan data worden opgeslagen en worden functies gedefinieerd die bepaalde acties op objecten van die klasse uitvoeren. Variabelen die binnen een klasse worden opgeslagen worden *attributes* genoemd. Functies die binnen een klasse-definitie worden gedefinieerd heten *member-functies* of *methoden*. In Python geldt altijd dat de eerste parameter van een methode *self* is, hierin wordt het object doorgegeven waarop de methode is aangeroepen en wat dus het object is dat moet worden gemanipuleerd. Via *self* kunnen dan de bijbehorende attributen worden uitgelezen.

Laten we als voorbeeld een klasse schrijven om breuken te representeren:

```
class Breuk(object):
    def __init__(self, teller, noemer):
        # Initialiseer het nieuwe object "self"
        # Sla teller en noemer in het object op als attributes.
        self.teller = teller
        self.noemer = noemer

    def geefTeller(self):
        return self.teller

    def geefNoemer(self):
        return self.noemer

    def telop(self, breuk):
        self.teller =  self.teller * breuk.geefNoemer() + breuk.geefTeller() *
            self.noemer
        self.noemer *= breuk.geefNoemer()

    def drukaf(self):
        print "{}/{}".format(self.teller, self.noemer)
```

We kunnen nu een object van deze klasse instantiëren en hierop methoden aanroepen:

```
b = Breuk(1, 4)
b.telop(Breuk(1, 2))
b.drukaf()
```

Bij het aanroepen van `drukaf` op object `b`, wordt de member-functie `drukaf` aangeroepen. Voor de parameter `self` wordt het object *voor* de punt (`.`) opgegeven: in dit geval dus `b`. In geval van deze aanroep zal de methode `drukaf` dus de waarden `teller` en `noemer` uit `b` lezen, aangezien de lokale variabele `self` naar hetzelfde object als `b` refereert.

Vaak is het handig om bij het aanmaken van een object een initialisatie-functie uit te voeren. Bij ons voorbeeld van de breuk is dat het initialiseren van de teller en de noemer. Dit gebeurt met de speciale methode `__init__`, in sommige talen ook wel de *constructor* genoemd. Deze functie wordt automatisch aangeroepen bij het maken van een nieuw object. Naast `self` mag de constructor extra (formele) parameters hebben. Bij het instantiëren van het object moeten er voor deze formele parameters corresponderende actuele parameters worden opgegeven. We mogen ook een constructor zonder extra parameters hebben (we krijgen altijd `self` als parameter, welke naar het zojuist gemaakte object wijst) zoals bijvoorbeeld:

```
1  class Trein(object):
2      def __init__(self):
3          self.aantal_passagiers = 0
4          self.vertraging = 0
5
6      def hoeveel_vertraging(self):
7          return self.vertraging
8
9
10 intercity = Trein()
11 print intercity.hoeveel_vertraging()
```

Voor ons breuken-voorbeeld zijn vele uitbreidingen mogelijk: vermenigvuldigen, vereenvoudigen, kopiëren, enz. Met een geavanceerde functionaliteit met de naam *operator overloading* (**(+)**, geen tentamenstof) is het mogelijk dat we objecten van zelfgemaakte types gewoon kunnen manipuleren met standaard operatoren zoals + en direct netjes kunnen printen. Een kleine aanpassing aan onze klasse is vereist (vanaf regel 14):

```
1  class Breuk(object):
2      def __init__(self, teller, noemer):
3          # Initialiseer het nieuwe object "self"
4          # Sla teller en noemer in het object op als attributes.
5          self.teller = teller
6          self.noemer = noemer
7
8      def geefTeller(self):
9          return self.teller
10
11     def geefNoemer(self):
12         return self.noemer
13
14     def telop(self, breuk):
15         nieuwTeller = self.teller * breuk.geefNoemer() + breuk.geefTeller() *
               self.noemer
16         nieuwNoemer = self.noemer * breuk.geefNoemer()
17
18         nieuw = Breuk(nieuwTeller, nieuwNoemer)
19         nieuw.vereenvoudig()
20         return nieuw
21
22     def __add__(self, other):
23         return self.telop(other)
```

```
24
25      def __str__(self):
26          return "{}/{}".format(self.teller, self.noemer)
```

Met `__add__` vertellen we Python hoe de + operator moet worden toegepast op objecten van ons type `Breuk`. `__str__` legt vast hoe objecten van ons type moeten worden afgedrukt. Er zijn vele andere operatoren die op een dergelijke manier voor ons type kunnen worden gedefinieerd. Met deze aanpassing is het mogelijk om de volgende code te schrijven:

```
1 b1 = Breuk(1, 4)
2 b2 = Breuk(1, 2)
3 b3 = b1 + b2
4 print b3
```

en dat ziet er al vele malen intuïtiever uit! Dat is precies de kracht van het object georiënteerd programmeren.

16 NumPy Arrays

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.

16.1 NumPy array datastructuur

Een array is een geordende rij van variabelen, net als een lijst, maar in tegenstelling tot een lijst zijn binnen een array alle variabelen van hetzelfde type. We zetten eerst een aantal belangrijke verschillen ten opzichte van de Python "list" op een rijtje:

- 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. Tevens zijn de operaties op NumPy arrays significant sneller. Dit is belangrijk wanneer je gaat werken met grotere datasets. Geef dus wanneer je gaat rekenen altijd de voorkeur aan NumPy arrays!
- Voor NumPy arrays moet van te voren de grootte worden opgegeven. Aan de hand hiervan wordt het aantal elementen bepaald dat de array zal bevatten en je kunt dit later niet meer uitbreiden. Dus in tegenstelling tot lijsten zijn NumPy arrays niet dynamisch.
- Het is eenvoudig om meer-dimensionale arrays te maken met NumPy. Matrices worden bijvoorbeeld opgeslagen als 2-dimensionale arrays. Het werken met 2-dimensionale arrays is vele malen eenvoudiger dan het werken met geneste lijsten.

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')
```

16.2 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.bool8`, `np.int32`, `np.float64` en `np.complex128`. Voor een compleet overzicht verwijzen we de lezer naar de documentatie[11].

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)
```

16.3 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.

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

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])
```

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]
```

16.4 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` [12]. 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.        ]
```

[12] 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`.

16.5 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 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 (binair talstelsel) en een exponent van -3, dan krijgen we $1.11011_2 \times 2^{-3} = (\frac{1}{1} + \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{32}) \times 2^{-3} = 0.23046875$. Er zijn 53 bits beschikbaar voor de significant (waarvan 1 bit voor de sign) en 11 voor de exponent. We kunnen hiermee ongeveer 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.

16.6 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 (shape). De lengtes van de assen worden vaak weergeven als een tuple. Bijvoorbeeld: `(2, 3, 5)` is een 3-dimensionale array met dimensies van lengte 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))   # 2 vlakken, 4 rijen en 3 kolommen
>>> 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]]
```

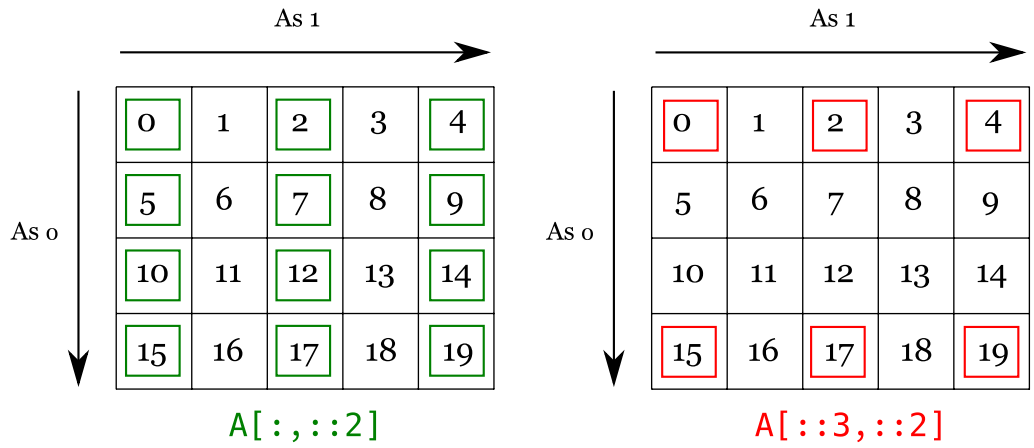
Let ook op dat wanneer je een kopie wilt maken van een array, je dit **expliciet** moet aangeven! Gebruik hiervoor de functie `np.copy`. Let op: in tegenstelling tot lijsten, maakt de expressie `B = A[:]` wanneer toegepast op NumPy arrays **geen** kopie!

```
>>> 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[13]. 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))  # 2 rijen, 5 kolommen
[[10 11 12 13 14]
 [15 16 17 18 19]]
>>> print np.arange(10, 20).reshape((5, 2))  # 5 rijen, 2 kolommen
[[10 11]
 [12 13]
```

[13] Technisch gezien wordt er een extra "view" gemaakt naar dezelfde array.



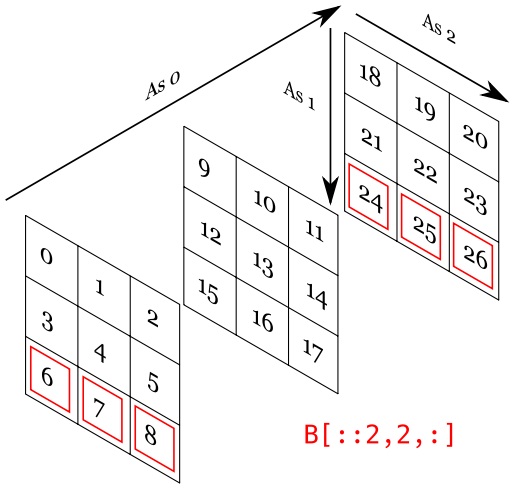
Figuur 1: Representatie van een 2-d array met assen en twee voorbeeld slices.

```
 [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
```

16.7 Indexeren en slicen met multidimensionale arrays

Om indexing en slicing toe te passen op multidimensionale arrays geef je een index of slice op per as (dimensie) van de array, gescheiden door komma's. Voor een element in een 2-dimensionale array geef je dus eerst een rij op, gevolgd door een kolom. 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: rij 2, kolom 1.
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, ...
```



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

```
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],
       [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]]])
```

16.8 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.]])
```

Je kunt er ook voor kiezen om gebruik te maken van de klasse `np.matrix`, een subklasse van `np.array`. In `np.matrix` is de vermenigvuldigingsoperator zo ingesteld dat deze standaard het matrixproduct toepast.

```
>>> A = np.matrix(np.eye(3))
>>> B = np.matrix(np.tile(4, (3, 3)))
>>> A * B
matrix([[ 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

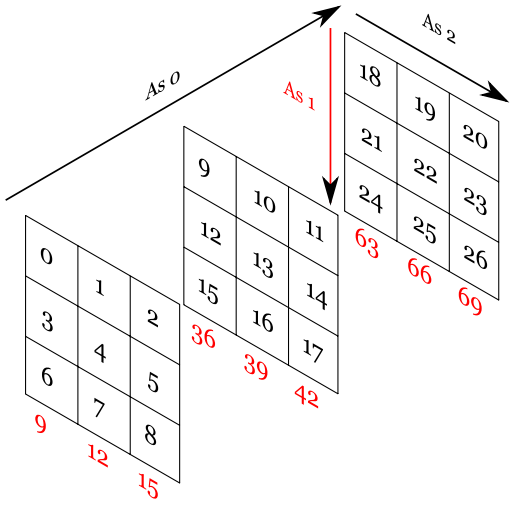
weggelaten wordt er gekeken of er een dimensie kan worden toegevoegd aan de buitenkant (dus vooraan in de vorm-tuple): dus bijvoorbeeld $(3,)$ mag $(1,3)$ worden. Stel we hebben een array met vorm $(5,6)$ en een array met vorm $(6,)$. NumPy maakt van de tweede array een array met vorm $(1,6)$. Nu komen 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.],
       [ 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 reductie-operator 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.

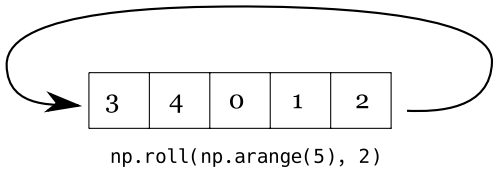


Figuur 3: Grafische weergave van `B.sum(axis=1)`.

```
>>> 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]]])
```



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

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!

16.9 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
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]])
```

16.10 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]]
```

Stel nu dat we in één keer een operatie willen uitvoeren op een rij en kolom van een matrix. Met normaal slicen kunnen we dat niet voor elkaar krijgen, dan moeten we dat altijd in twee stappen doen. We kunnen echter wel een masker opbouwen (hiervoor zijn dan twee stappen nodig) en vervolgens met behulp van dit masker de operatie op de matrix in één keer uitvoeren. Het masker kan ook meerdere keren worden gebruikt, hiermee kunnen de "kosten" voor het maken van het masker worden terugverdiend:

```
>>> A = np.zeros( (5, 5) )
>>> m = np.zeros(A.shape, dtype=np.bool8)
>>> m[2,:] = True
>>> m[:,2] = True
>>> A[m] = 999   # zet alle elementen geselecteerd door het masker op 999
>>> A
array([[   0.,    0.,  999.,    0.,    0.],
       [   0.,    0.,  999.,    0.,    0.],
       [ 999.,  999.,  999.,  999.,  999.],
       [   0.,    0.,  999.,    0.,    0.],
       [   0.,    0.,  999.,    0.,    0.]])
```

16.11 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`[14], 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:

```
>>> 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])
```

16.12 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
```

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

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.]]
```

Het is ook mogelijk om iedere kolom in een aparte array te ontvangen door toe te voegen `unpack=True`:

```
>>> a, b, c = np.loadtxt("test1.txt", unpack=True)
>>> print a
[ 1.  4.  0.  9.]
>>> print b
[ 2.  5.  9.  3.]
```

In het geval de getallen binnen 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]]
```

Een ander probleem doet zich voor wanneer een regel in een bestand uit verschillende typen data bestaat, zoals een string gevolgd door getallen. Alle elementen van een array moeten immers van hetzelfde type zijn. We nemen als voorbeeld `test4.txt`:

```
a 1 2 3
b 4 5 6
c 0 9 1
d 9 3 4
```

Dit probleem is op te lossen door de kolommen met getallen eerst apart te laden en daarna de kolom met strings. We gebruiken hiervoor het argument `usecols` met als waarde een lijst van kolomnummers die moeten worden geladen (we slaan dus kolom `0` over):

```
>>> A = np.loadtxt("test4.txt", usecols=[1, 2, 3])
>>> print A
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 0.  9.  1.]
 [ 9.  3.  4.]]
```

Of door eerst een array van strings in te lezen en daarna de subarray bestaande uit getallen om te zetten naar een float-array:

```
>>> A = np.loadtxt("test4.txt", dtype=str)
>>> B = np.array(A[:,1:], dtype=float)
>>> A = A[:,0]
>>> print A
['a' 'b' 'c' 'd']
>>> print B
[[ 1.  2.  3.]
 [ 4.  5.  6.]
 [ 0.  9.  1.]
 [ 9.  3.  4.]]
```

Mocht de functie `np.loadtxt` geen uitkomst bieden, kijk dan eens naar het uitgebreide broertje `np.genfromtxt`.

17 Algoritmen

Bij het programmeren draait bijna alles om het vervaardigen van stappenplannen of *algoritmen*. We bekijken in dit hoofdstuk een aantal illustratieve en/of veel gebruikte algoritmen. Deze voorbeelden dienen ook als illustratie hoe dergelijke (vaak eenvoudige) problemen kunnen worden omgezet in een geschikt algoritme. We maken gebruik van lijsten of NumPy arrays bij het implementeren van deze algoritmen.

17.1 Rekenalgoritmen

Als eerste behandelen we een aantal algoritmen voor het berekenen van een of ander gegeven. De algoritmen die we zullen bekijken komen voort uit de wiskunde: grootste gemeenschappelijke deler, bepaling priemgetallen, de driehoek van Pascal en matrixvermenigvuldiging.

17.1.1 Grootste gemeenschappelijke deler

Allereerst een functie die de *grootste gemeenschappelijke (gemene) deler*, de *ggd*, van twee gegeven gehele getallen berekent.

```
1 def ggd(x, y):
2     while y != 0:
3         rest = x % y
4         x = y
5         y = rest
6     return x
```

Dit voorbeeld is een implementatie van het *algoritme van Euclides*. We kunnen deze ggd-functie bijvoorbeeld gebruiken om breuken te vereenvoudigen:

```
# Vereenvoudig breuk teller/noemer zoveel mogelijk.
# Aanname: teller >= 0, noemer > 0
def vereenvoudig(teller, noemer):
    deler = ggd(teller, noemer)
    if deler > 1:
        teller = teller / deler
        noemer = noemer / deler

    return teller, noemer
```

De test `if deler > 1:` hoeft er overigens niet bij. Het is wel belangrijk dat er van een hulpvariabele `deler` gebruik wordt gemaakt: wordt er geprobeerd twee maal door de grootste gemene deler van teller en noemer te delen (dus tweemaal "dezelfde" functieaanroep, wat op zich ook al niet zo snugger is), dan zal bij de tweede deling de helaas gewijzigde waarde van `teller` gebruikt worden. De noemer van de breuk zal dan niet veranderen, wat – als de ggd niet 1 is – toch de bedoeling is!

17.1.2 Priemgetallen

Een *priemgetal* is een getal dat geen delers heeft behalve 1 en zichzelf. Er zijn vele manieren om te bepalen of een gegeven geheel getal groter dan 1 een priemgetal is. Een voor de hand liggend algoritme (maar lang niet het snelste) is het volgende:

```
import math  # voor sqrt

def priem(getal):
    '''Levert True precies als getal een priemgetal is'''
    deler = 2
    wortel = math.sqrt(getal)
    geendelers = True
    while (deler <= wortel) and geendelers:
        if getal % deler == 0:
            # deler gevonden: getal niet priem.
            geendelers = False
        deler += 1

    return geendelers
```

De extra hulpvariabele *wortel* voorkomt het steeds opnieuw uitrekenen van de wortel uit het oorspronkelijke getal. Het is duidelijk dat je niet meer voorbij deze wortel hoeft te kijken: als daar een deler van het oorspronkelijke getal zou zitten, zou er ergens voor die wortel ook een moeten zijn.

Een vergelijkbaar algoritme is de *zeef van Erathosthenes*. In dit algoritme wordt in een array bijgehouden welke getallen, kleiner dan een zekere bovengrens, nu wel of niet priem zijn.

```
import numpy as np

def erathosthenes(N):
    wortel = np.sqrt(N)
    # Initialiseer op True, tot tegendeel bewezen is ...
    # Als zeef[i] True is, is "i" een priemgetal.
    zeef = np.ones(N, dtype=np.bool8)
    zeef[0] = False
```

```
    zeef[1] = False
    for getal in range(2, int(wortel)):
        if zeef[getal]:
            # Streep veelvouden door
            veelvoud = 2 * getal
            while veelvoud < N:
                zeef[veelvoud] = False
                veelvoud += getal
    return zeef

def bepaalpriemgetallen(N):
    priem = erathosthenes(N)

    # Druk de priemgetallen af
    for getal in range(2, N):
        if priem[getal] == True:
            print getal,
    # Of: print np.where(priem == True)
```

17.1.3 Driehoek van Pascal

In de welbekende driehoek van Pascal is elk getal in de driehoek de som van de twee getallen erboven. Deze getallen heten ook wel binomiaalcoëfficiënten. Om de driehoek van Pascal te berekenen gebruiken we een matrix en lopen we op een handige manier door deze matrix heen. De elementen voor rij voor rij ingevuld. We maken gebruik van de regel:

$$\binom{i}{j} = \binom{i-1}{j-1} + \binom{i-1}{j}$$

en komen dan uit op het volgende programma:

```
import numpy as np

def pascaldriehoek(n):
    pascal = np.zeros( (n, n), dtype=np.int32 )
    # Nulde kolom bevat enen
    pascal[:,0] = 1
    pascal[0,1] = 0
    print pascal[0,0],
    for i in range(1, n):
        print "\n", pascal[i,0],
        for j in range(1, i+1):
            pascal[i,j] = pascal[i-1,j-1] + pascal[i-1,j]
            print pascal[i,j],
        if i != n - 1:
            pascal[i][i+1] = 0
```

Het is overigens ook mogelijk om met een 1-dimensionaal array te werken. In de `i`-de iteratie bevat dat array dan de getallen uit de `i`-de rij van de driehoek van Pascal. Nu moet elk getal de som van de twee getallen erboven worden, wat in een 1-dimensionaal array betekent de som van het getal links van jezelf en jezelf. Loop je nu van links naar rechts door het array, dan wordt te vroeg de waarde van de array-elementen gewijzigd, namelijk terwijl je hun oude waarde nog nodig hebt. Door van rechts naar links te lopen, en op te merken dat de driehoek toch symmetrisch is, ontstaat een eenvoudig programma.

```
def pascaldriehoekbeter(n):
    rij = np.zeros(n, dtype=np.int32)
    # 0-de kolom altijd 1
    rij[0] = 1
    print rij[0]
    for i in range(1, n):
        for j in reversed(range(1, i+1)):
            rij[j] = rij[j-1] + rij[j]
            print rij[j],
        print rij[0]
```

17.2 Matrixvermenigvuldiging

Een veel toegepaste operatie op matrices is vermenigvuldiging. We zagen al dat matrixvermenigvuldiging is geïmplementeerd in NumPy. Deze operatie is in NumPy behoorlijk snel, dus er is geen reden om deze te vervangen. Maar gewoon uit interesse: hoe zouden we zelf zo'n matrixvermenigvuldiging moeten implementeren? Voor het gemak gaan we uit van matrices A en B die we vermenigvuldigen en het resultaat opslaan in C. De definitie van deze operatie is:

$$C_{ij} = \sum_{k=0}^{n-1} A_{ik} B_{kj}, \quad 0 \leq i, j < n$$

De getallen uit de i-de rij van A en de j-de kolom van B worden paarsgewijs vermenigvuldigd en de resultaten opgeteld ten einde het matrixelement C_{ij} op te leveren.

```
def matmul(A, B):
    """Bereken het matrixproduct van vierkante matrices A en B"""

    # Stel zeker dat kolomdimensie A overeenkomt met rijdimensie B.
    assert A.shape[1] == B.shape[0]

    M, K, N = A.shape[0], A.shape[1], B.shape[1]

    C = np.zeros( (M, N) )
    for i in range(0, M):
        for j in range(0, N):
            for k in range(0, K):
                C[i,j] += A[i,k] * B[k,j]
    return C
```

Zo te zien kost matrixvermenigvuldigen van twee n bij n matrices overigens n^3 vermenigvuldigingen van array-elementen; men zegt wel: een $O(n^3)$-algoritme. Zeker voor grote matrices zal de berekening met de bovenstaande Python-code een tijd duren, gebruik daarom altijd de NumPy-operator!

17.3 Zoeken en Sorteren

Van oudsher hebben algoritmes voor sorteren en zoeken in de belangstelling gestaan. Bijvoorbeeld voor het zoeken van een telefoonnummer of het sorteren van een leden-database. Gezien de vaak grote hoeveelheden gegevens is het belangrijk dat dit enigszins efficiënt gaat. Er zijn dan ook veel verschillende algoritmen bedacht. We zullen een klein aantal algoritmen kort behandelen en we maken gebruik van of een lijst of een 1-dimensionale NumPy array. Om te beginnen bekijken we een functie die het kleinste getal uit een lijst kan opsporen:

```
def minimum(lijst):
    """Geef het kleinste getal uit de gegeven lijst"""
    klein = lijst[0]
    for el in lijst:
        if el < klein:  # kleinere gevonden
            klein = el
    return klein
```

We kunnen deze functie als volgt aanroepen:

```
a = [47, 54, 52, 35, 84, 69, 99, 77, 48, 6,
     46, 75, 29, 67, 63, 13, 30, 41, 86, 97]
print minimum(a)
```

17.3.1 Lineair zoeken

We willen in een ongesorteerde lijst zoeken naar een getal. Een eenvoudige methode is om simpelweg vooraan (of achteraan) te beginnen en element voor element vergelijken tot we het gezochte element hebben gevonden of tot alle elementen geweest zijn en het getal niet voor blijkt te komen.

```
def lineairzoeken(lijst, getal):
    """Zoek getal in lijst volgens methode van lineair zoeken.
    Returnwaarde: index waar getal is gevonden, anders -1"""
    index = 0
    gevonden = False
    while not gevonden and (index < len(lijst)):
        if getal == lijst[index]:
            gevonden = True
        else:
            index += 1
    if gevonden:
        return index
    else:
        return -1
```

Als je pech hebt, bijvoorbeeld als het gezochte getal niet voorkomt, kost je dat voor een lijst met n elementen n vergelijkingen (kortom $O(n)$ (lineaire orde)). Deze methode heet *lineair zoeken*. Er bestaan ook methodes die sneller kunnen zoeken als bekend is dat de gegeven lijst al is gesorteerd, bijvoorbeeld *binair zoeken*.

17.3.2 Een eenvoudige sorteermethode

Nu willen we een lijst bestaande uit gehele getallen op grootte oplopend sorteren. Een eenvoudige methode is de volgende. In de beginsituatie is het gesorteerde stuk van de lijst leeg en het ongesorteerde stuk is de hele rij. We zoeken nu eerst het kleinste element in het ongesorteerde stuk en verwisselen dat met het voorste element van dat stuk. Het gesorteerde stuk wordt nu één element groter en het ongesorteerde stuk één element kleiner. We herhalen dit totdat het ongesorteerde stuk leeg is en de gehele lijst is gesorteerd. Deze methode wordt ook wel *selection sort* genoemd.

```
def simpelsort(lijst):
    for voorste in range(len(lijst)):
        # Zoek kleine element in ongesorteerde stuk [i:]
        plaatskleinste = voorste
        kleinste = lijst[voorste]

        for k in range(voorste + 1, len(lijst)):
            if lijst[k] < kleinste:
                kleinste = lijst[k]
                plaatskleinste = k

        if plaatskleinste > voorste:
            # Wissel om
            lijst[plaatskleinste], lijst[voorste] = \
                lijst[voorste], lijst[plaatskleinste]
```

17.3.3 Bubblesort

Een variant op selection sort is het algoritme *bubblesort*. De code hiervoor is erg compact, maar helaas is het geen hele efficiënte sorteermethode. Het sorteren van een rijtje met n getallen kost altijd $\frac{1}{2}n(n-1)$ vergelijkingen (men zegt vaak $O(n^2)$, kwadratische orde). Er zijn andere methodes die dat sneller kunnen zoals Shellsort en quicksort.

```
def bubblesort(lijst):
    for i in range(1, len(lijst)):
        for j in range(0, len(lijst) - i):
            if lijst[j] > lijst[j + 1]:
                lijst[j], lijst[j + 1] = lijst[j + 1], lijst[j]
```

De gegeven lijst wordt op grootte gesorteerd op de volgende manier. In de eerste ronde worden `lijst[0]` en `lijst[1]` vergeleken en indien nodig (namelijk als `lijst[0]` groter is dan `lijst[1]`) wordt hun inhoud verwisseld. Daarna `lijst[1]` en `lijst[2]`, ..., `lijst[n-2]` en `lijst[n-1]` (met `n = len(lijst)`). Het is duidelijk dat het grootste element nu achteraan staat. In de tweede ronde worden `lijst[0]` en `lijst[1]` vergeleken, ..., `lijst[n-3]` en `lijst[n-2]`. Er vindt dus één vergelijking minder plaats. Zo gaat dit verder en in de laatste (`n-1`) ronde hoeven alleen nog maar `lijst[0]` en `lijst[1]` vergeleken te worden. De grote getallen "bubbelen" als het ware naar achteren en dat is hoe deze methode aan haar naam is gekomen.

17.3.4 Invoegsorteer

Stel nu dat je al een gesorteerde rij hebt en je wilt een element toevoegen. Je kunt dan gebruik maken van de methode *invoegsorteer*, oftewel *insertion sort*. Invoegsorteer zoekt voor een gegeven element de plaats op waar dit element terecht moet komen. Voor een lijst of array betekent dit dat alle elementen vanaf de plek waar het nieuwe element moet worden ingevoegd, één plaats naar achter moeten schuiven.

Een ongesorteerde rij kan men met invoegsorteer sorteren door element voor element aan een oorspronkelijke rij (die "per definitie" gesorteerd is) toe te voegen zoals beschreven. De complexiteit is vergelijkbaar met die van bubblesort. Ter inspiratie geven we een implementatie die werkt met een NumPy array in plaats van een lijst.

```
def invoegsorteer(A):
    N = A.shape[0]
    # We zetten steeds A[i] goed in reeds gesorteerde beginstuk
    for i in range(1, N):
        temp = A[i]
        j = i - 1
        while j >= 0 and A[j] > temp:
            A[j+1] = A[j]
            j -= 1
        A[j+1] = temp
```

18 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"[15]. 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.

18.1 Een eerste plot

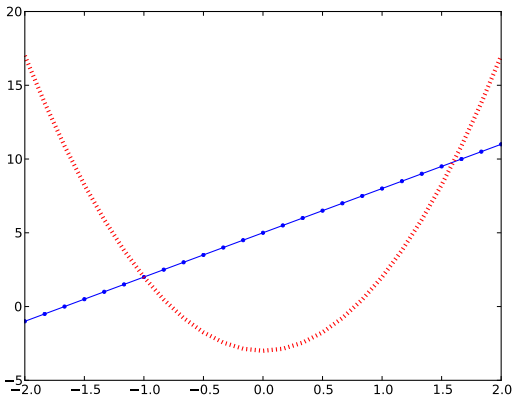
Laten we beginnen met het maken van een plot van een simpele lineaire functie. Eerst importeren we de benodigde packages. Vervolgens berekenen we onze coördinaatparen. Tenslotte plotten we deze coördinaten en zetten we de plot op het scherm (we zullen later zien hoe we de plot naar een bestand kunnen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

18.2 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,

[15] `http://matplotlib.org/gallery.html`



Figuur 5: Plot van twee lijnen met verschillende eigenschappen.

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[16]. 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[17].
- `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.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  x = np.linspace(-2, 2, 25)
5  y1 = 3 * x + 5
6  y2 = 5 * x ** 2 - 3
7
8  plt.plot(x, y1, color="blue", lw=1.0, linestyle="solid", marker=".")
9  plt.plot(x, y2, color="red", lw=4.0, linestyle="dotted")
10
11 plt.show()
```

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

18.3 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.
- `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.

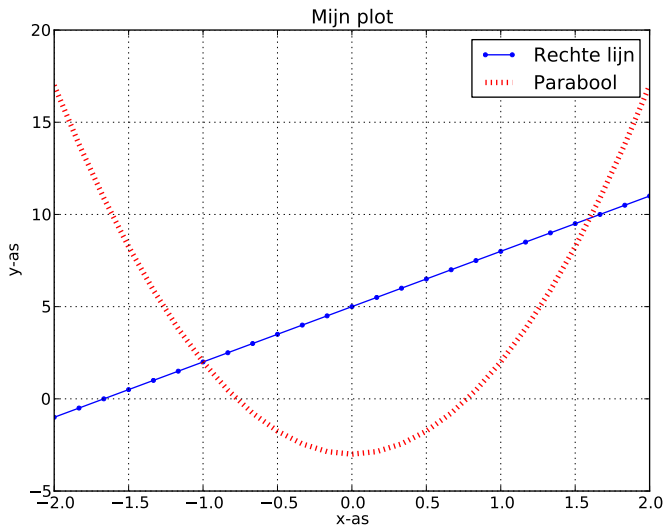
```
import numpy as np
import matplotlib.pyplot as plt

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()
```

18.4 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"`.



Figuur 6: Nette plot met titel en as-labels

18.5 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.

18.6 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 door de corresponderen extensie in de bestandsnaam te gebruiken. Vaak wordt er gebruik gemaakt van PDF bestanden en plaatsen we de extensie `.pdf` achter de bestandsnaam, bijvoorbeeld `plt.savefig("mijnplot.pdf")` slaat de huidige plot op als `mijnplot.pdf` in PDF-formaat. Een andere veel gebruikte extensie is `.png` om PNG-bestanden te verkrijgen.

18.7 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.

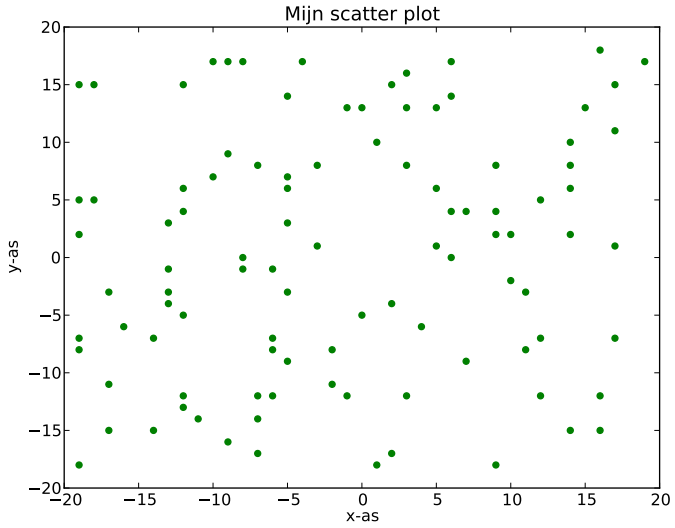
18.8 Scatter plots

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")
```

18.9 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 7: Een voorbeeld van een scatter plot.

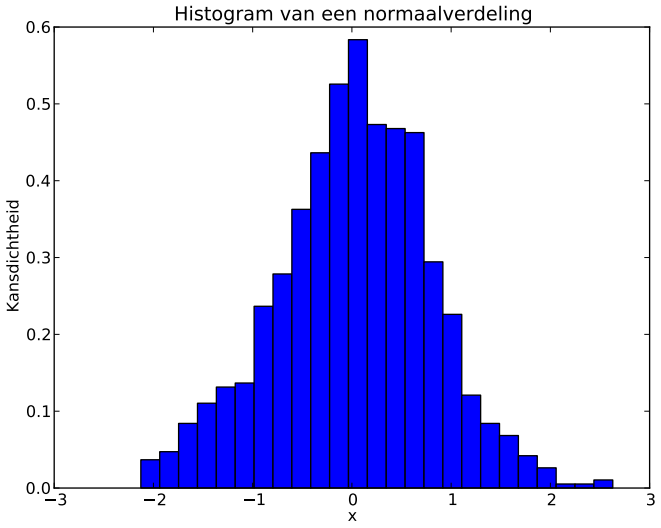
```
import numpy as np
import matplotlib.pyplot as plt

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")
```

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.

19 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 door `Out` te indexeren of de notatie met _ te gebruiken:



Figuur 8: Een histogram van de normaalverdeling.

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

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

In [3]: _1 + 6
Out[3]: 16
```

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. `pycat` doet hetzelfde als `cat` en past ook Python syntax highlighting toe. 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.

Een bestaand Python-programma draaien binnen de iPython shell is mogelijk met het commando `%run`, bijvoorbeeld `%run mijnprogramma.py`.

Wanneer je snel wilt weten wat een functie doet, zet er een vraagteken achter, bijvoorbeeld `list.append?`. iPython geeft dan de documentatie van de functie weer. Dit werkt ook voor NumPy functies.

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 [18] en probeer het vooral eens uit!

[18] `http://ipython.org/notebook.html`

20 Meer leren over Python (+)

Dit dictaat is een beknopte introductie tot Python en het gebruik van NumPy en Matplotlib. Binnen de taal Python zijn er veel meer mogelijkheden. Zoals we al zagen is Python volledig object georiënteerd en kun je zelf klassen schrijven en operators overloaden. Andere functionaliteiten van Python zijn bijvoorbeeld generators implementeren, werken met list comprehensions, lambda functies gebruiken, exceptions gooien, iterator klassen schrijven, function decorators 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, of alle priemgetallen!).

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

```
[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)]
```

Bij het werken met lijsten en generators wordt er ook vaak gebruik gemaakt van de functies `map`, `reduce` en `filter`. De functie `map` maakt een nieuwe lijst, door een gegeven functie toe te passen op de elementen van een gegeven lijst. Stel we willen alle elementen opgeleverd door onze generator-functie omzetten naar het type `str`:

```
strings = map(str, graaf_tel())
```

Met `reduce` passen we een operator toe op alle elementen in de lijst. Het resultaat is een enkele waarde (net als de reductie-operatoren in NumPy die we hebben besproken). Stel we schrijven onze eigen `sum` met behulp van `reduce` om een lijst random getallen te sommeren:

```
import random, operator

R = random.sample(range(1000), 20)  # 20 random getallen
som = reduce(operator.add, R)       # hetzelfde als sum(R)
```

Op eenzelfde manier kunnen we al die getallen met elkaar vermenigvuldigen:

```
product = reduce(operator.mul, R)
```

Tenslotte dan `filter`. Deze functie past een filter-functie toe op elk lijst-element. Indien de filter-functie de waarde `True` geeft voor een lijst-element wordt deze opgenomen in de nieuwe lijst die de returnwaarde van `filter` zal zijn. En anders wordt dit element weggelaten. We specificeren de filter in dit voorbeeld als zogenaamde lambda-functie:

```
# Bepaal alle getallen kleiner dan 100
klein = filter(lambda x: x < 100, R)
```

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"[19] 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[20]. 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[21].

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 zult 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.

Walter A. Kosters. Collegedictaat Programmeermethoden.
`http://liacs.leidenuniv.nl/~kosterswa/pm/colldic.pdf` Geraadpleegd op: 5 september 2016.

[19] `https://docs.python.org/2/library/index.html`
[20] `https://pypi.python.org/pypi`
[21] `https://docs.python.org/2/installing/`

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.

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.