Workshop Python voor Wis- en Natuur/ Sterrenkundigen

28 januari 2019

Universiteit Leiden
The Netherlands

Waarom & hoe

- Waarom?
  - Sinds 2016 is het vak Programmeermethoden gesplitst:
    I & W krijgen de taal C++, N & A krijgen Python.
  - Binnen de opleidingen natuur- en sterrenkunde gaat programmeren een belangrijke rol spelen en dit wordt gedaan aan de hand van Python 3.

- Hoe?
  - 1 uur hoorcollege: introductie, basisvaardigheden, etc.
  - 2 uur werkcollege: oefenen aan de hand van oefenopgaven.
  - Het dictaat van Programmeermethoden NA is te vinden op de website:

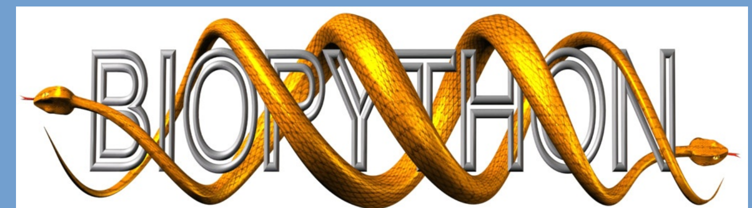
  http://liacs.leidenuniv.nl/~rietveldkfd/courses/prna2018/dictaat.pdf

Wat is Python & Waarom Python?

- "Scripttaal", ontworpen door Guido van Rossum eind jaren '80 / begin jaren '90.
- Eenvoudig & portable (werkt op zowat alle systemen).
- Complexe bewerkingen in maar enkele regels code -- hierdoor een ultiem gereedschap!
- Zeer populair geworden in de laatste tien jaar.

Waarom zo populair?

- Zeer uitgebreide standaard bibliotheek.
- Eenvoudig om uitbreidingen te schrijven.
- Er zijn vele modules ontwikkeld voor het doen van numeriek rekenwerk en maken van plots.
- Hierdoor zeer populair in verschillende wetenschappelijke disciplines.



Compileren vs. interpreteren

- C++ is een *gecompileerde* programmeertaal.

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

- Python is een *geïnterpreteerde* programmeertaal.

```
gedit programma.py
python3 programma.py
```

Compileren vs. interpreteren (2)

- Python wordt ook wel een *scripttaal* genoemd, net als bijvoorbeeld Perl, Ruby en PHP. De programma's noemen we vaak "scripts".
- Geen compilatieslag, dus sneller testen.
- Nadeel: minder fouten worden van te voren ontdekt.
  - (Hier zijn echter wel tools voor: *pylint*, *pyflakes*, maar deze kunnen niet alle fouten van te voren ontdekken. Hoewel voor de C++ compiler hetzelfde geldt).

Compileren vs. interpreteren (3)

- Omdat Python een geïnterpreteerde taal is, heeft Python ook een interactieve modus.

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
[GCC 4.2.1 Compatible Apple LLVM 10.0.0 (clang-1000.11.45.2)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
>>>
```

- Nog mooier is iPython:

```
Python 3.6.7 (default, Oct 21 2018, 08:56:20)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: 3 + 4
Out[1]: 7

In [2]: Out[1] * 10
Out[2]: 70

In [3]: _2 + 4
Out[3]: 74

In [4]:
```

Python verkrijgen

- Python 3 is "open source" en gratis te verkrijgen.
- Linux: steeds vaker standaard geïnstalleerd.
  - Anders altijd beschikbaar via je Linux distributie (apt-get, yum, ...).
- Op Windows en Mac is standaard geen Python 3 geïnstalleerd. Zelf downloaden!
  - Wij raden aan Anaconda Python (https://www.anaconda.com/)
  - Kies voor “Python 3.6 version”!
  - Komt inclusief handige ontwikkelomgeving: Spyder.
  - (Mac diehards kunnen ook installeren via MacPorts of brew)
- Zie ook het dictaat voor details.

Spyder

Spyder (Python 3.6)

File Edit Search Source Run Debug Consoles Projects Tools View Help

C:\Users\kris

Editor - C:\Users\kris\.spyder-py3\temp.py

temp.py

```
1 # -*- coding: utf-8 -*-
2 """
3 Spyder Editor
4
5 This is a temporary script file.
6 """
7
8
```

Help

Source Console Object

Usage

Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in

Variable explorer | File explorer | Help

IPython console

Console 1/A

```
Python 3.6.5 |Anaconda, Inc.| (default, Mar 29 2018, 13:32:41) [MSC
v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.4.0 -- An enhanced Interactive Python.

In [1]:
```

IPython console | History log

Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 1 Column: 1 Memory: 32 %

Een eerste Python programma

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

Een eerste Python programma (1)

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

- Commentaarregels

Een eerste Python programma (2)

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

- “Keywords”

Een eerste Python programma (3)

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

- Inspringen (indentation).
- Dit moet consistent gebeuren!

Een eerste Python programma (4)

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

- Manier om geen nieuwe regel op het beeldscherm "af te drukken".

Mogelijke fouten

- Wat voor fouten kunnen er optreden bij het draaien van een programma?
- Bij het inlezen van het programma:
  - "*SyntaxError*": de syntax van het programma klopt niet, er staat bijv. een haakje verkeerd, "elze" in plaats van "else", etc.
  - "*IndentationError*": er is verkeerd ingesprongen (komt later aan bod).
- Bij het uitvoeren van het programma:
  - "*NameError*": er worden variabelen gebruikt die niet zijn gedefinieerd,
  - "*ValueError*": ongeldige conversie,
  - delen door 0.
  - enz.

Variabelen in Python

- In C++ moesten variabelen vooraf worden gedeclareerd als een bepaald type en kan dit type niet meer veranderen.
- Dit is in Python niet nodig, we maken variabelen met een toekenningsstatement (assignment).
- Toekenning op een al bestaande naam overschrijft de oude waarde.

```
a = 4
b = "testje!"
a = "overschrijven" # oude waarde van a wordt overschreven
d = a + g
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name ’g’ is not defined
```

Variabelen in Python (2)

- Elke variabele in Python heeft een type.

```
>>> a, b, c = 9, 3.14, "strrrr"
>>> type(a)
<type 'int'>
>>> type(b)
<type 'float'>
>>> type(c)
<type 'str'>
>>> a = "strrr2" # oude waarde a wordt overschreven
>>> type(a)
<type 'str'>
```

Getalen

- *int*: Integers, zo groot als maar past in het geheugen van de computer. Zeer grote getallen mogelijk!
  - *(Let op: bij de meeste programmeertalen zijn integers meestal 4 of 8 bytes groot en hebben daardoor een beperkt bereik; -2^{31} tot $2^{31} - 1$ respectievelijk -2^{63} tot $2^{63} - 1$. We krijgen hier later bij NumPy ook mee te maken.)*
- *bool*: True of False.
- *float*: Benaderingen (!) van reële getallen. In Python altijd double precision (8 bytes).
- *complex*: Complexe getallen. Ingebouwd!

Conversie van getallen

- `float( )` is een type conversiefunctie. Accepteert ook strings: `pi = float("3.14")`.
- Andere typeconversies: `int( )`, `complex( )`, `str( )`.
- Belangrijk: niet hetzelfde als een C++ cast, die kan bijvoorbeeld niet van (een ouderwetse) string naar float!
- Operatie op twee verschillende typen resulteert in een impliciete conversie: type coercion.

print functie

- `print( )` zet data op het scherm.
- Functie-aanroep `print`, met als argumenten een reeks van expressies.
- Impliciete conversie naar strings.
- Spaties ingevoegd tussen uitvoeren van verschillende expressies.

```
>>> 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
```

Uitvoerformattering

- Nieuwe stijl, oude stijl staat omschreven in dictaat.
- Met een format field specificeren we hoe een variabele moet worden afgedrukt.
- `{0:8.4f}` minimum veld breedte 8; precisie 4
- `{1:>10s}` minimum veld breedte 10; rechts uitgelijnd

```
print("{0:6d} {1:8.4f} {2:20s}".format(a, b, "test"))
print("{een} {twee}".format(een=a, twee=b))
```

Strings

- In Python is een string een object, net als in C++ de klasse "string" wordt gebruikt.
  - Python heeft geen `char` type.
- Strings werken als een volledig geïntegreerd type:

```
>>> woord = "De."
>>> len(woord)
3
>>> woord == "test"
False
>>> woord == "De."
True
```

Strings - Indexing en slicing

- In een ouderwetse C-string kunnen we met een index een individueel array-element uitlezen.
- In Python kunnen we objecten van het type `str` ook indexeren:
  - `woord[1]`
- Je mag ook een start en eind-index geven, bijvoorbeeld om een substring uit te lezen. We noemen dit slicing:
  - `zin[3:14]`
  - De eind-index telt **niet** mee! Dus we selecteren hier indices 3 t/m 13.

Strings - Indexing en slicing

```
>>> s = "een lange test string"
>>> s[2]
’n’
>>> s[-4]
’r’
>>> s[3:8]
' lang'
>>> s[6:]
’nge test string’

# daarnaast vele andere methoden, bijv.:
>>> s.startswith("een")
True
```

Lijsten

- Een `list`-object is een geordende lijst van variabelen.
- Verschillen met C++ arrays:
  - De variabelen hoeven **niet** van hetzelfde type te zijn.
  - De grootte van de lijst staat niet vast, je kunt eenvoudig elementen toevoegen en verwijderen.
- Er wordt vaak over lijsten gesproken als compound data type of sequence type.

Lijsten (2)

- Lijsten kunnen worden aangemaakt met blokhaken:

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

- Of pas een lege lijst later aan:

```
>>> a = []
>>> a.append("een")
>>> a.append("twee")
>>> a.append("drie")
>>> a.insert(0, "nul")
>>> a
["nul", "een", "twee", "drie"]
>>> a.remove("een")
>>> a.pop()
"drie"
>>> a
["nul", "twee"]
```

Lijsten - Indexing en slicing

- Indexing & slicing zoals we al bij strings zagen:

```
>>> a = [0, 1, 2, 3, 4, 5, 6, 7]
>>> len(a)
8
>>> a[6]
6
>>> a[2:5]
[2, 3, 4]
>>> a[3:]
[3, 4, 5, 6, 7]
>>> a[:6]
[0, 1, 2, 3, 4, 5]
>>> a[4] = 'ha!'
>>> a
[0, 1, 2, 3, 'ha!', 5, 6, 7]
```

Uitgebreid slicing

- Slicing kent ook een stapgrootte:

```
start : eind : stap
```

- Eind-index telt niet mee.
- Elk van de delen mag worden weggelaten.
- Bij lijsten mag je ook toekenningen doen aan de slice.

Uitgebreid slicing (2)

```
>>> a = range(10)
>>> a[2:5]
[2, 3, 4]
>>> a[2:]
[2, 3, 4, 5, 6, 7, 8, 9]
>>> a[:5]
[0, 1, 2, 3, 4]
>>> a[2:8:2]
[2, 4, 6]
>>> a[::2]
[0, 2, 4, 6, 8]
>>> a[::3]
[0, 3, 6, 9]
```

```
>>> a = range(10)
>>> a[0:5] = ['a', 'b', 'c', 'd', 'e']
>>> a
['a', 'b', 'c', 'd', 'e', 5, 6, 7, 8, 9]
```

Controlestructuren

- De belangrijkste controlestructuren in Python zijn:
  - `if` voor het maken van keuzes.
  - `for` voor een vast aantal herhalingen.
  - `while` voor een onbekend aantal herhalingen.

if - then - else

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

Boolean expressies

- De predicaten die je kent uit C++ werken gewoon: `==`, `!=`, `<`, `>=`, enzovoort.
- In plaats van `!`, && en `||` gebruiken we `not`, `and` en `or`.
- Als je zowel `and` als `or` in een expressie gebruikt: gebruik haakjes om verwarring te voorkomen!
- Ook in Python wordt short-circuiting toegepast.
  - `(x != 0 and y / x == 7)`

Boolean expressies

```
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": ...
if y >= 3 and (x == 4 or x == 5) and \
   z == 12 and (q >= 10 or q <= -10): ...
```

for loops

- `for`-loops in Python een stuk eenvoudiger.
- We drukken een iteratie van een lijst uit.
- De iteratievariabele neemt opeenvolgend de verschillende waarden van de lijst aan.

```
for karakter in ['a', 'b', 'c', 'd', 'e']:
    print(karakter, end='')
# drukt af: a b c d e
```

```
for i in [1, 2, 3, 4, 5]:
    print(i)
```

`range( )` functie

- Voor grote aantallen herhalingen wil je niet met de hand zo'n lijst schrijven.
- Met `range( )` kunnen getallenreeksen worden gemaakt.
- `range(start, stop, step)`
- De gegeven stop-waarde doet **niet** mee!
- *(range() genereert geen volledige lijst in het geheugen, maar een object dat kan worden geitereerd. Dus ook efficient voor grote reeksen).*

range( ) functie

```
>>> list(range(6))
[0, 1, 2, 3, 4, 5]
>>> list(range(3,6))
[3, 4, 5]
>>> list(range(0, 50, 5))
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
>>> list(range(20, 50, 5))
[20, 25, 30, 35, 40, 45]


>>> for i in range(10):
...     print(i, end='')
...
0 1 2 3 4 5 6 7 8 9
```

Inspringregels

- In C++ kun je slordig zijn met de layout van je code, Python is daar echter veel strikter in.
- Correct inspringen is een must, verkeerd inspringen wordt bestraft met een `IndentationError`.
- Wanneer inspringen?
  - Om blokken van statements te vorm.
  - `if`-statements, loops en definiëren van functies.
  - In C++ plaats je bij bijna al deze gevallen accolades!
- Binnen eenzelfde blok **moet** er op elke regel op dezelfde manier worden ingesprongen.
- De eerste regel die anders wordt ingesprongen maakt geen deel meer uit van dat blok.

Inspringregels (2)

- In het volgende voorbeeld van een geneste loop vinden we 3 niveau's terug:

```
for i in range(1, 6):
    print("{0}:".format(i), end='')
    for j in range(1, i+1):
        print(i * j, end='')
    print()
```

pass statement

- In C++ konden we een accolade openen en direct weer sluiten, zonder statements er tussen. Een leeg blok!
- Dat wordt met inspringen een beetje lastig ...
- Oplossing: `pass`-statement.

```cpp
x = 10;

if (x > 0) {

}

cout << “test”;
```

```python
x = 10
if x > 0:
    # niets --- fout!!!!!
print("test")

if x > 0:
    pass
print("test")
```

Dangling else in Python

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

- C++: "Zorg ervoor dat de layout klopt - de compiler kijkt er niet naar."
- Maar in Python wordt er door de interpreter **juist wel** naar de layout gekeken!
- Er kan geen verwarring zijn: de layout (het inspringniveau) is leidend.

Functies

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

q = telop(12354, 23451234)


def paar(a, b, c):
    # Gebruik tuple als returnwaarde
    return (a, a + b, a + b + c)

x, y, z = paar(1, 2, 3)
```

`t = paar(1, 2, 3)` **mag ook!**

Opletten!

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

q = telop(12354, "hallo!!")
```

- Bij de regel `c = a + b` probeert Python nu een `int` en `str` bij elkaar op te tellen.
- En dat gaat niet; er wordt een `TypeError` gegeven.

Default & keyword arguments

- Het is mogelijk om een "standaardwaarde" op te geven voor elke **formele** parameter.
- In dat geval is het niet meer verplicht om het **actuele** argument op te nemen.

```
def teken_cirkel(x, y, straal, kleur="blauw"):
    # Code om een cirkel te tekenen.

teken_cirkel(0, 0, 10)
teken_cirkel(0, 0, 10, "rood")
teken_cirkel(0, 0)  # Mag *NIET*!
```

- Ook veel gebruikt: keyword arguments waarbij de formele parameter bij naam wordt genoemd.

```
def teken_lijn(p1,p2,kleur="zwart",dikte=1.0,pijl=None,stippel=False):
    # hier wordt de lijn getekend
    pass

teken_lijn( (10, 10), (100, 10), stippel=True, dikte=2.0)
```

Bestand inlezen

- In C++ gebruikten we `ifstream` en `ofstream`. In Python hebben we het `file` object.
- C++-achtige manier:

```
f = open("test.txt", "r")
line = f.readline()
while line != "":
    print(line, end='')
    line = f.readline()
f.close()
```

Bestand inlezen (2)

- Dit is meer Python-achtig (en dus simpeler :)

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

Praktisch voorbeeld

```
f = open("getallen.txt", "r")
for line in f:
    line = line.rstrip("\n")
    a, b, c = line.split(" ")
    a, b, c = int(a), int(b), int(c)
    som = a + b + c
    print("Som:", som)
f.close()
```

Schrijven naar bestanden

- Om te schrijven naar een `file` object kun je gebruik maken van de `write` methode of van `print`.
- Bij `write` **moet** de parameter een string zijn:

```
f.write("hello world")
f.write(42)                  # NEE!
f.write(str(42))             # OK
```

```
f = open("uitvoer.txt", "w")
print("Met print is het eenvoudiger", file=f)
print("Geheel getal: {0} Floating point {1}." \
      .format(51, 3.1412345), file=f)
f.close()
```

NumPy introductie

- NumPy: Numerical Python.
- Wordt in heel veel takken van de wetenschap gebruikt voor numeriek rekenwerk.
- Belangrijkste onderdeel: multidimensionale array datastructuur.
  - Redelijk snel: want eigenlijk geïmplementeerd in C!
- NumPy is een “package” en moeten we eerst importeren:

```
import numpy as np
```

De NumPy array

- Multidimensionale array zoals je ook in C++ hebt leren kennen.
- Aantal belangrijke verschillen ten opzichte van Python lijsten:
  - Aantal elementen staat na aanmaken vast.
  - Alle elementen zijn van hetzelfde type.
  - Gebruik van operatoren op NumPy arrays is wat je zou verwachten in tegenstelling tot Python lijsten (zie ook later).

NumPy arrays maken

- Bij het maken van een array moet het aantal elementen worden opgegeven.
- Verschillende manieren:
  - Creeren aan de hand van een Python list.
  - `np.zeros`: initialisatie met nullen.
  - `np.ones`: initialisatie met nullen.
  - `np.tile`: initialisatie met gespecificeerde waarde.

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

NumPy arrays maken (2)

- `np.arange(start, stop, stap)`: maak een getallenreeks. Mag ook floating-point gebruiken!

- `np.linspace(begin, eind, N)`: N getallen uit gesloten interval, gelijke afstand tussen de elementen.

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

Lijst vs. NumPy array

- Laten we eens gaan rekenen met een lijst.

```
>>> 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'
```

Rekenen met NumPy arrays

- Rekenen met Python lijsten geeft ons niet de resultaten die we zouden verwachten.
- Daarom: als je gaat rekenen, gebruik NumPy arrays!
- Operatoren werken elementgewijs.

```
>>> 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])
```

Rekenen met NumPy arrays (2)

- Toepassen formule op een getallenreeks:

```
>>> 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])
```

- Een reductieoperator berekent 1 resultaat voor een gehele array.
  - Voorbeelden: `np.sum()`, `np.mean()`, `np.std()`, `np.amin()`, `np.amax()`
- Ook de wiskundige functies ontbreken niet.
  - Voorbeelden: `np.log()`, `np.exp()`, `np.sin()`, `np.cos()`, `np.tan()`, `np.sqrt()`, `np.floor()`, `np.ceil()`
  - Parameter mag zowel een scalair als NumPy array zijn.
- Merk hier tenslotte nog op: geen loops!

Slicing & indexing

- Indexing en slicing zoals je bent gewend.
- Toekenning aan een slice:
  - Toekenning scalair: elk element in de slice krijgt deze waarde.
  - Toekenning array: arrays moeten evenveel elementen bevatten!

```
>>> a = np.ones(8)
>>> a[2:6] = 4
>>> a
array([ 1.,  1.,  4.,  4.,  4.,  4.,  1.,  1.])
>>> a[:4] = range(10, 14)
>>> a
array([ 10.,  11.,  12.,  13.,   4.,   4.,   1.,   1.])
```

Multidimensionale arrays

- NumPy arrays kunnen een arbitrair aantal dimensies aan.
- Dimensies worden ook wel "assen" genoemd.
- Elke as heeft ene bepaalde lengte.
- Elke NumPy array heeft een "vorm" waarin de lengte van elke as is vastgelegd.
  - `(3, )`: 1 dimensie lengte 3.
  - `(3, 4)`: 2 dimensies: 3 rijen, 4 kolommen.
  - `(10, 3, 4)`: 3 dimensies: 10 vlakken, 3 rijen, 4 kolommen.

Multidimensionale arrays (2)

- Voor het alloceren van een multidimensionale array werken de gebruikelijke functies. In plaats van een aantal elementen vul je een shape tuple in.

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

- `np.eye(n)` maakt een *n* x *n* identiteitsmatrix.

```
>>> I = np.eye(3)     # Een 3x3 identiteitsmatrix
>>> print(I)
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Arrays kopiëren

- Pas op: een toekenning is geen kopieeractie!!

```
>>> A = np.eye(3)
>>> B = A           # Kopieert niet, maar legt een extra
                    # referentie aan.
>>> B[0,2] = 9      # Indexeren zien we zo.
>>> 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.
```

Indexeren in meerdere dimensies

- Om een element aan te duiden in een multidimensionale array: geef per as (dimensie) een index op, gescheiden door komma's.
  - `B[0,2]`
  - `B[1,2,3,4,5]`
- In plaats van een index mag je ook een slice opgeven!
  - De lege slice : selecteert de gehele as.

Slicing

```
>>> A[:,:]
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
>>> A[2,1]       # Selecteer een enkel element
11
>>> A[2,:]       # Selecteer de derde rij.
array([10, 11, 12, 13, 14])
>>> A[2]         # Slices aan het einde mag je weglaten
array([10, 11, 12, 13, 14])
>>> A[:,3]       # Selecteer de vierde kolom
array([ 3,  8, 13, 18])
```

Slicing (2)

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

`A[:,::2]`

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19

`A[::3,::2]`

Het veranderen van de vorm

- Met `.reshape()` kun je de vorm van een array aanpassen. De parameter is een vorm-tuple met de gewenste vorm.
- Let op: het aantal elementen blijft hetzelfde.

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

Illustratie

Zeef van Eratosthenes

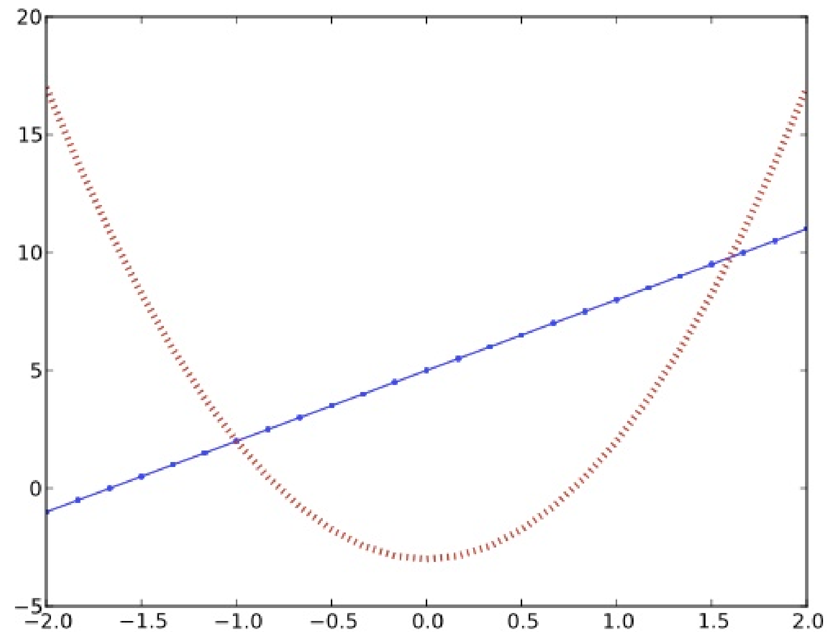
```
N = 1000
wortel = np.sqrt(N)
# Initialiseer op True, tot tegendeel bewezen is ...
zeef = np.ones(N, dtype=np.bool8)
zeef[0] = False
zeef[1] = False
for getal in range(2, int(np.ceil(wortel))):
    if zeef[getal]:
        # Streep veelvouden door
        veelvoud = 2 * getal
        while veelvoud < N:
            zeef[veelvoud] = False
            veelvoud += getal

for getal in range(2, N):
    if zeef[getal] == True:
        print(getal, end='')
# Of: print np.where(zeef == True)
```

matplotlib

- Matplotlib is een plotting "package" waarmee hoge kwaliteit plots kunnen worden gemaakt.
- Zeer veel mogelijkheden.
- Wordt gebruikt in combinatie met NumPy.

Een eerste plot

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

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

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

plt.show()
```



Kleuren en markers

- `plt.plot( )` accepteert een groot aantal argumenten.
- `color="red"`
- `marker="o"` - punten markeren met cirkels.
- `linewidth=2.5` - dikke lijn.
- `linestyle="dotted"` - stippellijn.
- `label="Mijn lijn"` - komt in de legenda terecht.

Titel & labels

- Zonder titel en aslabels is de plot natuurlijk niet af.
- `plt.title("titel")`: titel van de plot.
- `plt.xlabel("label")`, `plt.ylabel("label")`: aslabels.
- We mogen TeX gebruiken in matplotlib strings

Grid en assen

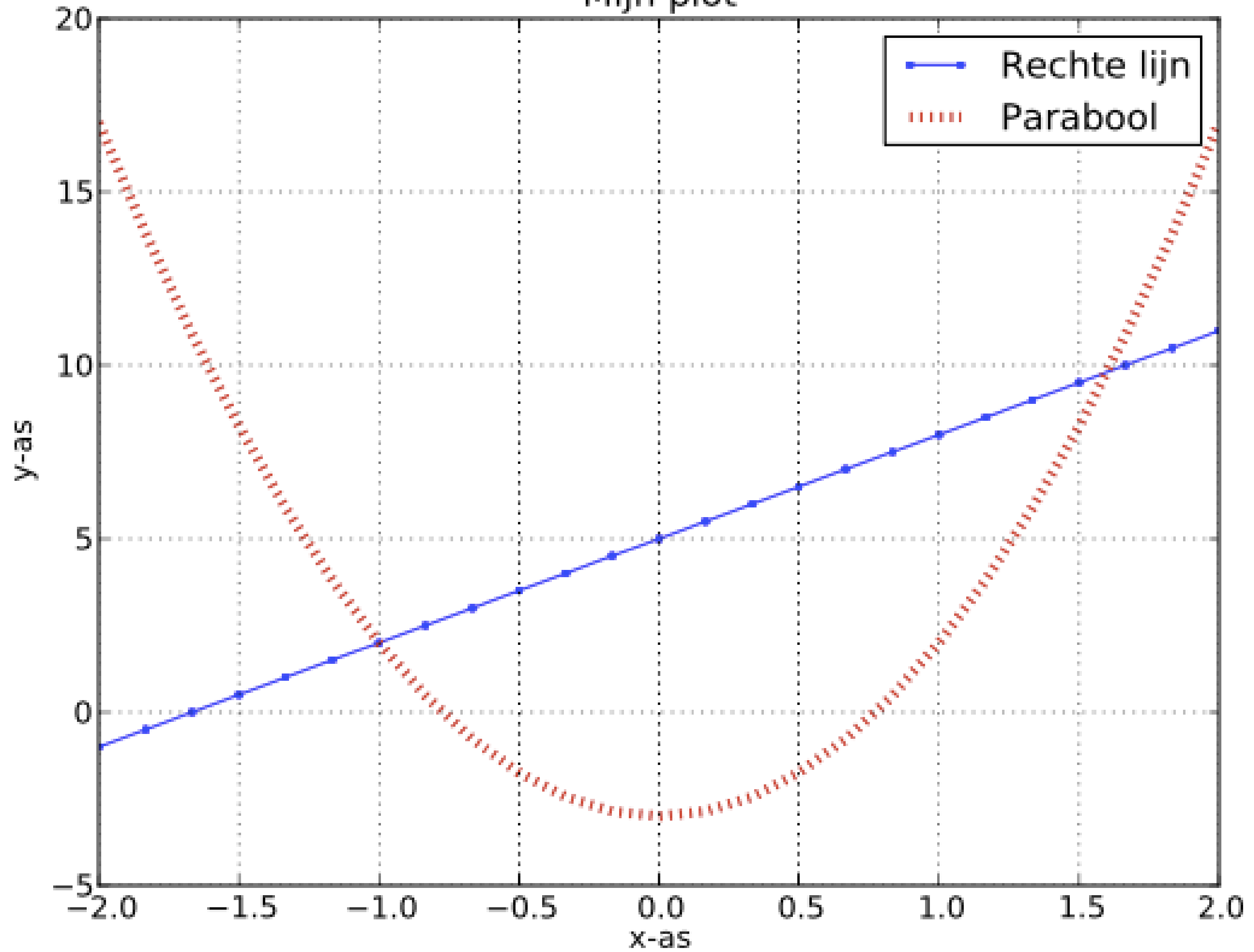
- Met `plt.grid(True)` kun je een achtergrond grid aanzetten.
- De intervallen van de assen kunnen op verschillende manieren worden ingesteld:
  - `plt.ylim(-2, 10)` en analoog voor `plt.xlim()`.
  - Of: `plt.axis(xmin=0, xmax=20., ymin=-10, ymax=100.)`.
- `plt.xscale("log")`: geef de x-as een logaritmische schaal.

Legenda

- De opgegeven labels kunnen eenvoudig in een legenda worden afgebeeld.
- `plt.legend(loc="upper right")`.
- Je mag ook opgeven iets als `center`, `lower left`, etc.

Voorbeeld

```
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()
```


Mijn plot
Rechte lijn
Parabool
y-as
x-as
20
15
10
5
0
−5
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0

Opslaan naar een bestand

- Om een plot op te slaan als PDF bestand: vervang `plt.show()` met `plt.savefig("hallo.pdf")`.

Meer Python

- Leer werken met het ingebouwde help-systeem:
  - `help(str)`, `help(np)`, `help(list.append)`
  - In iPython kun je een vraagteken achter een variabele of functie zetten om er informatie over te krijgen: `np.sum?`

- Daarnaast is er uitgebreide online documentatie:
  - Python tutorial: https://docs.python.org/3/tutorial/
  - Standard library reference: https://docs.python.org/3/library/index.html
  - NumPy: https://docs.scipy.org/doc/numpy-1.15.4/reference/
  - Matplotlib: https://matplotlib.org/contents.html

Standaard libraries

- Python heeft een zeer uitgebreide standard library:
  - CSV import/export, regular expressions, werken met data/kalenders, werken met zipfiles, SQL database toegang, internet protocollen (e-mail, HTTP), enz.
- Naast NumPy komt SciPy wellicht ook van pas.
  - Natuurkundige constanten, MATLAB I/O, IDL I/O, sparse matrices, lineaire algebra, differentiaal vergelijkingen.
- Daarnaast op internet vele andere modules beschikbaar.
  - Zoeken via PyPI: https://pypi.python.org/pypi
  - Installeren via Linux package manager, of pip.

Tot slot

- Veel meer informatica in het dictaat:
  - http://liacs.leidenuniv.nl/~rietveldkfd/courses/prna2018/dictaat.pdf

- Zometeen werkcollege in de computerzaal.