NUMBA

SCHNELLES PYTHON, GANZ EINFACH

Christoph Deil (MPIK Heidelberg)

10. Mai, 2019 bei QX, Frankfurt

Slides at https://christophdeil.com

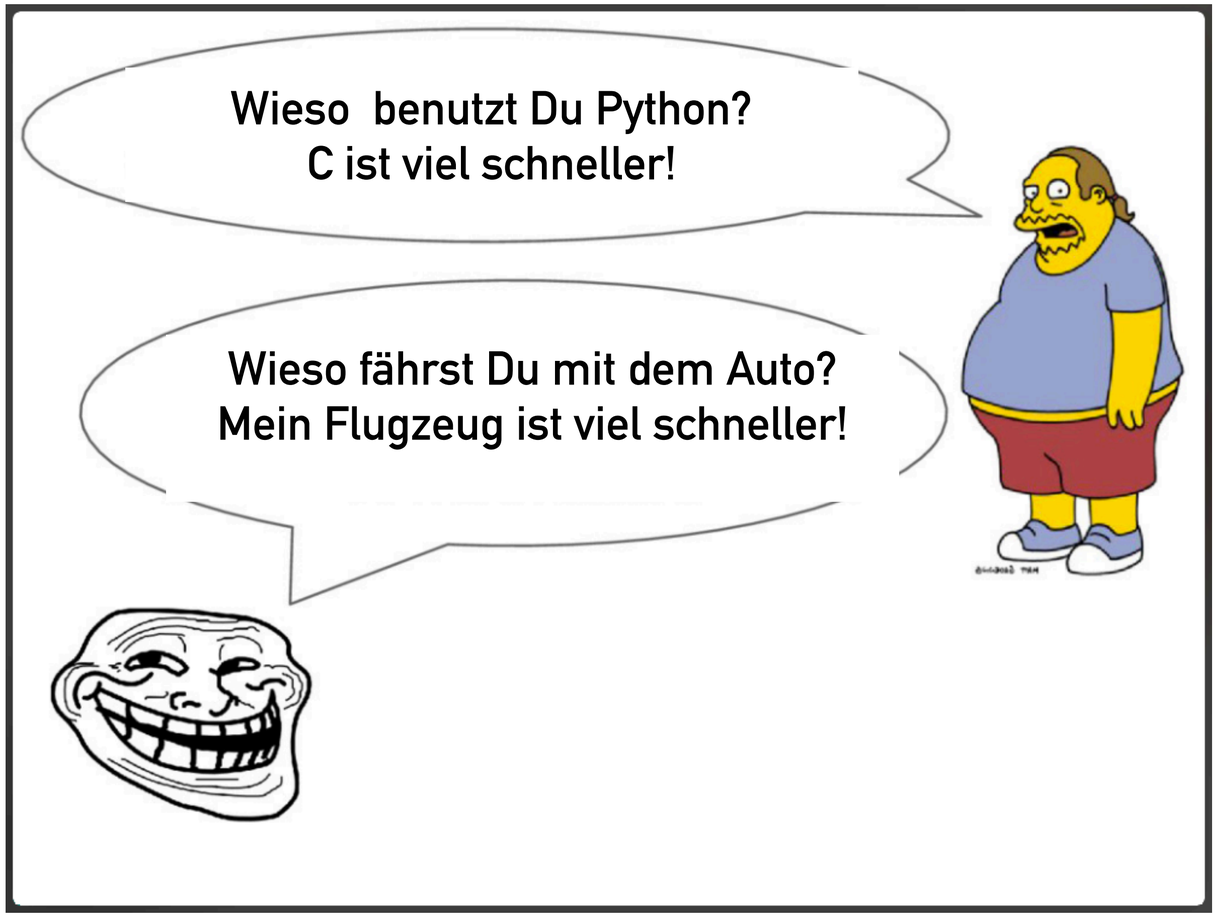
MAX-PLANCK-INSTITUT
FÜR KERNPHYSIK

EINLEITUNG

- Wie funktioniert Python, Numpy und Numba?
  Wann und wie benutzt man Numba?
- Ich benutze Numba erst seit kurzem, bin kein Experte.
- Was benutzt ihr?
  — Python, Numpy, Numba, C, C++, Cython, … ?
- Wer hat Speed-Probleme?
  Wer benutzt multi-core CPU, CPU Cluster, GPU, …?
- Gerne Fragen jederzeit!
  Gerne Kommentare & Erfahrungen am Ende!

PYTHON & NUMPY

SCHNELL CODEN & SCHNELLER CODE



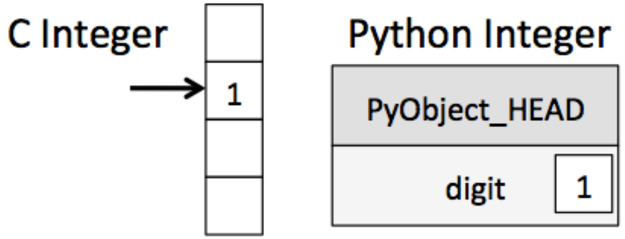
Quelle: Jake Vanderplas 2013 (LINK)

- Entwickler-Zeit ist oft wichtig, CPU-Zeit sekundär
- Für viele Anwendungen ist Python & Numpy (& PyData Pakete) Klasse
- Aber: manche Algorithmen sind schwer in Python & Numpy zu schreiben, oder zu langsam

```
x = 42
```

```
type(x)
```

```
int
```




```
import sys
sys.getsizeof(x)
```

```
28
```

Python int: 28 bytes, C int: 8 bytes

```
%timeit x * x
```

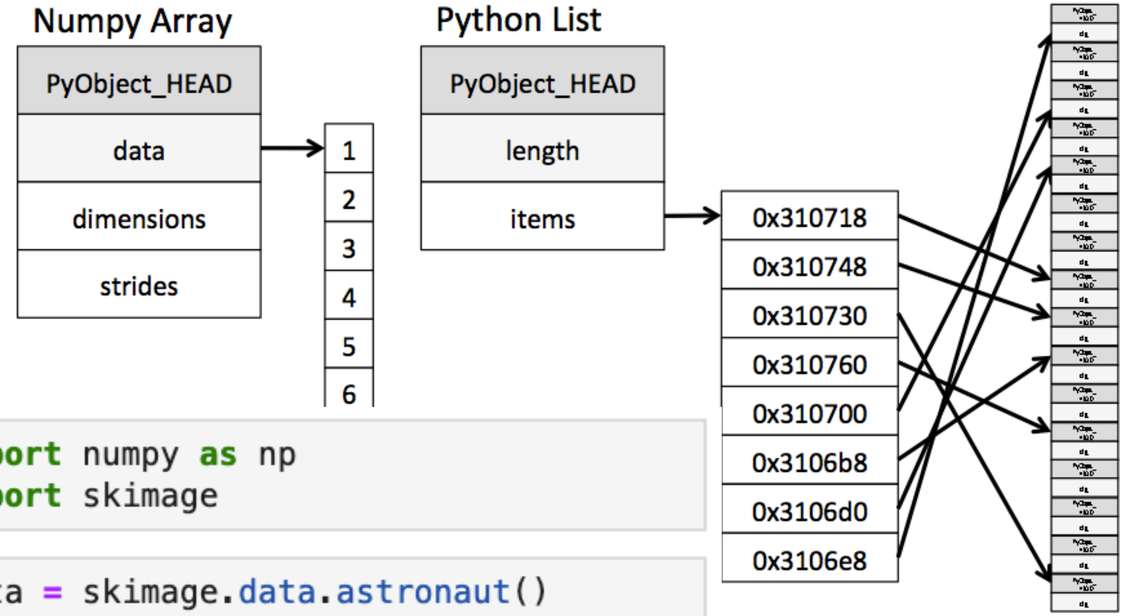
60 ns auf meinem Laptop — langsam!
Funktion rufen, PyObject, Operation, neues PyObject

```
import random

def monte_carlo_pi(nsamples):
    acc = 0
    for i in range(nsamples):
        x = random.random()
        y = random.random()
        if (x ** 2 + y ** 2) < 1.0:
            acc += 1
    return 4.0 * acc / nsamples

%timeit monte_carlo_pi(1_000_000)
```

Beispiel für Mathe-Code, der in Python langsam ist.

PYTHON

- CPython: Bytecode in virtueller Maschine
  “Global Interpreter Lock” - ein Thread
  Sehr einfach, nicht wie JS, Java, PyPy
- Alles in Python ist ein PyObject
  Auch Zahlen (int, float) und list
- Python ist sehr dynamisch. Funktion rufen und Attribut-Zugriff langsam.
- Python ist sehr langsam für und verschwendet Speicher für numerische Daten & Algorithmen!




```
import numpy as np
import skimage
```

```
data = skimage.data.astronaut()
```

```
data.dtype
```

```
dtype('uint8')
```

```
data.ndim
```

```
3
```

```
data.shape
```

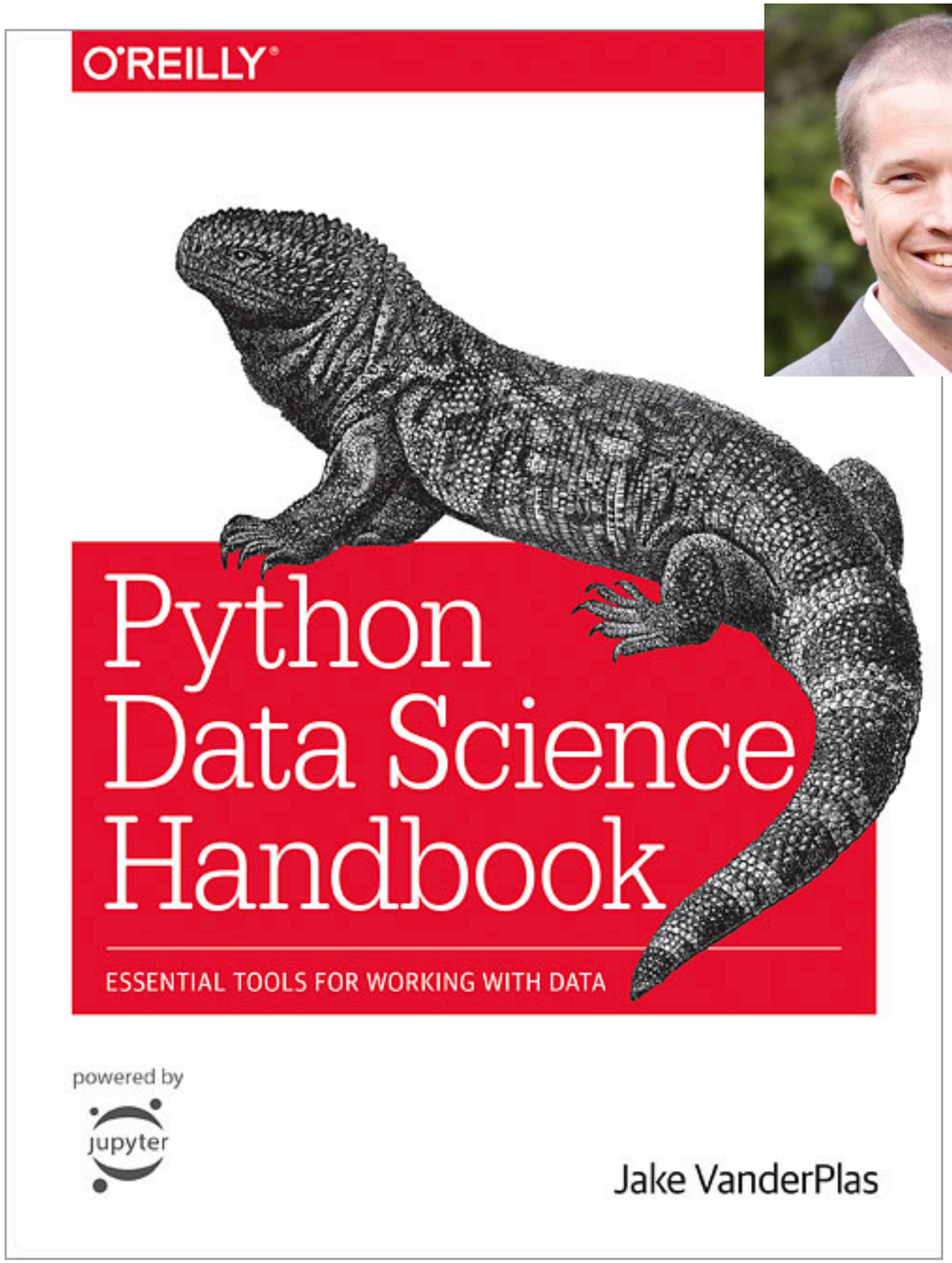
```
(512, 512, 3)
```

```
np.mean(data[100:110, 200:210, 2])
```

```
123.74
```

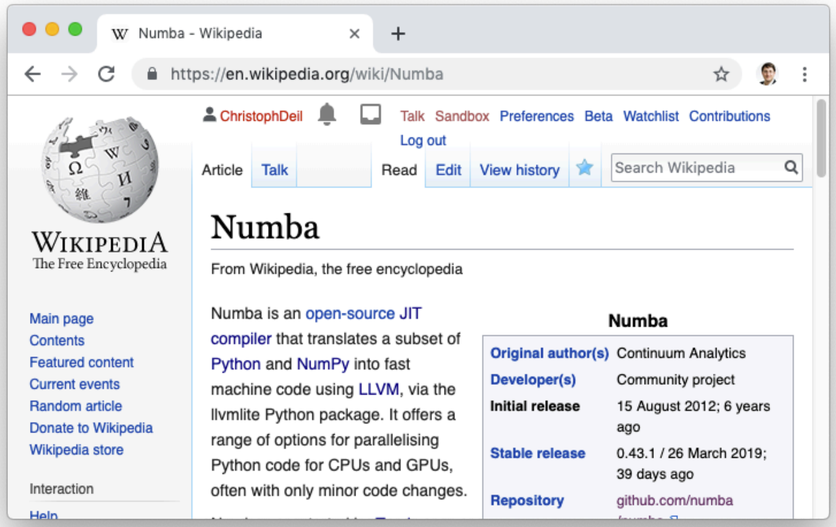
NUMPY

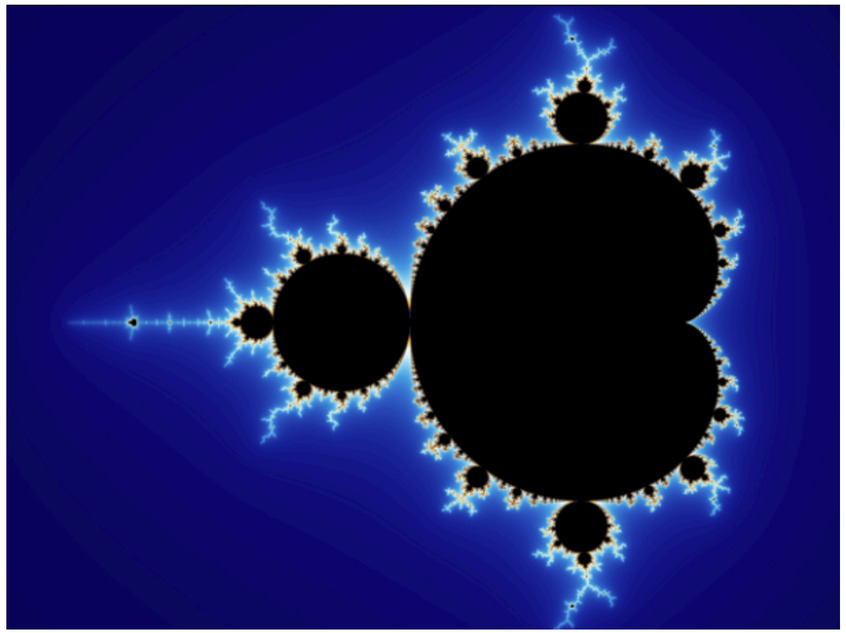
- Numpy hat zwei Sachen:
  - **numpy.ndarray:** fixer dtype, flexibel: ndim, shape, strides
  - **universal functions:** vektorisiert, broadcasting
- Viele Daten passen in Arrays: Bild, Video, Sound, Zeitreihen, …
  (oder werden passend gemacht: word2vec)
- Viele Algorithmen sind nur 1-10 Zeilen Numpy Expressions oder Funktionsaufrufe
- Vieles schon da: pandas, scipy, scikit-image, scikit-learn, dask, pytorch, tensorflow, …



TIP

- Tip zum Lernen von Python & Numpy *(und IPython, Jupyter, matplotlib, scikit-learn)*
- "Python Data Science Handbook" von Jake Vanderplas!
- Wer kennt das?
- Jupyter notebooks = Kapitel (10-30 min)
- Frei verfügbar auf Github: jakevdp/PythonDataScienceHandbook
- Ausprobieren: Binder oder Google Colab
- Beispiel: 2.1: "Understanding data types"





Beispiele:
Objekte
Bäume
Graphen
Text
…

Beispiel:
Mandelbrot
Fraktal

PROBLEME MIT NUMPY

1. **Daten**: nicht alles passt in ein Array
2. **Code**: nicht alle Algorithmen schon verfügbar oder einfach zu schreiben mit Python, Numpy, pandas, Scipy, …
3. **Performance**: Numpy meist ein Thread, moderne CPUs and GPUs multi-Thread. Temporäre Array-Kopien ineffizient

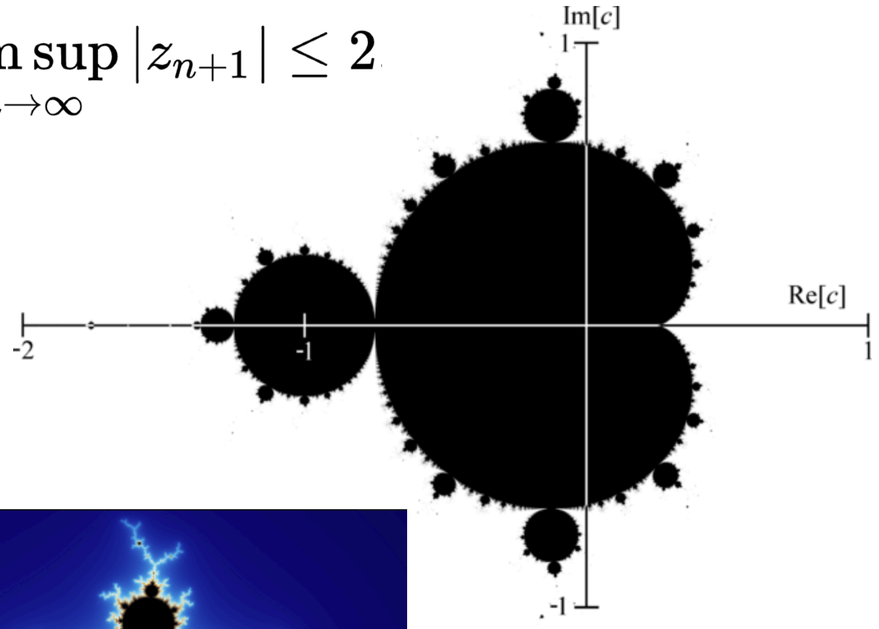
__Numba__ braucht auch Array Daten,
hilft mit Problem Code & Performance!

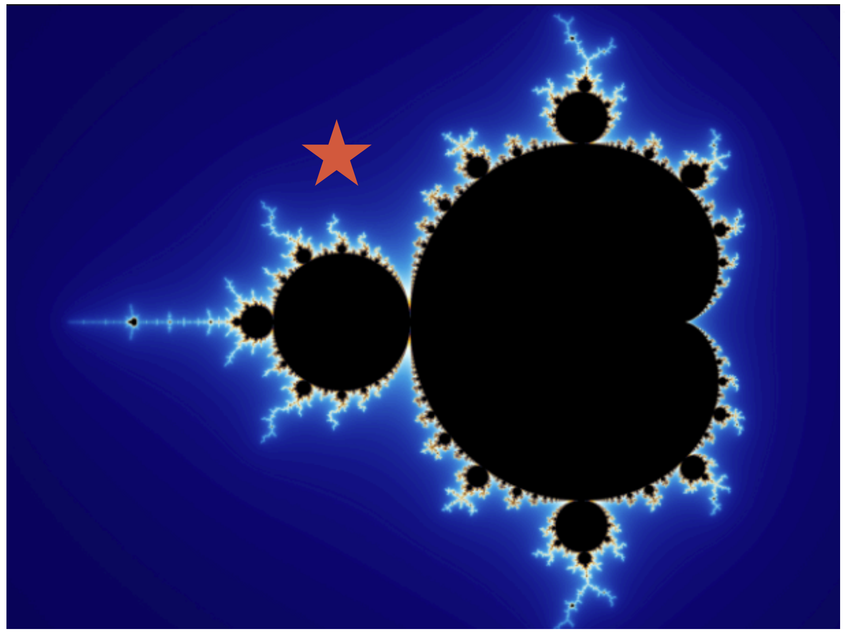
AMD intel NVIDIA®

MANDELBROT FRAKTAL

$$z_0 = 0 \quad z_{n+1} = z_n^2 + c$$

$$c \in M \iff \limsup_{n \to \infty} |z_{n+1}| \leq 2$$





- Mandelbrot Fraktale Bild als Beispiel für sehr rechen-intensiven Algorithmus
- Definition der Mandelbrot-Menge mit der komplexen Zahlen-Ebene
- Konvergenz von z_n definiert Menge Geschwindigkeit definiert Farbe

```
def mandel(c, i_max=100):
    z, i = 0, 0
    while abs(z) < 2 and i < i_max:
        z = z ** 2 + c
        i = i + 1

    return i
```

```
mandel(-1 + 0.5j)
```

5

```python
def mandel(c, i_max=100):
    z, i = 0, 0
    while abs(z) < 2 and i < i_max:
        z = z ** 2 + c
        i = i + 1

    return i
```

```python
def mandelbrot_image_python(width, height):
    image = np.empty((height, width), dtype=np.uint8)

    for h in range(height):
        for w in range(width):
            c = # aus (h, w) und Bild min/max ausrechnen
            image[h, w] = mandel(c)

    return image
```

```python
def mandelbrot_image_numpy(height, width, i_max=100):
    image = np.full((height, width), i_max, dtype=np.uint8)
    # Koordinaten-Arrays aus Bild min/max ausrechnen
    y, x = np.ogrid[y_min:y_max:height*1j, x_min:x_max:width*1j]
    c = x + y * 1j

    z = c
    for i in range(i_max):
        z = z ** 2 + c
        diverged = np.absolute(z) > 2
        diverging_now = diverged & (image == i_max)
        image[diverging_now] = i
        z[diverged] = 2

    return image
```

MANDELBROT CODE

- Bild = 2-dim Numpy array
- Python-Code: schön einfach, aber Python Pixel-Loop ist extrem langsam
- Numpy-Code:
  - Naive Implementierung sehr ineffizient (alle Pixel bis i_max iteriert)
  - Optimierte Implementierung komisch und immer noch langsam
- Optionen in solchen Fällen:
  Cython, C, C++, Numba, …

MANDELBROT MIT CYTHON


```
%%cython --cplus -c-O3
import cython, numpy        # load Python interface to Numpy
cimport numpy               # load C++ interface to Numpy (types end in _t)

@cython.boundscheck(False) # turn off bounds-checking
@cython.wraparound(False)  # turn off negative index wrapping (e.g. -1 for last element)
def run_cython(int height, int width, int maxiterations=20):
    y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
    c = x + y*1j
    fractal = numpy.zeros(c.shape, dtype=numpy.int32) + maxiterations
    cdef numpy.ndarray[numpy.complex128_t, ndim=2, mode="c"] c_array = c
    cdef numpy.ndarray[numpy.int32_t,      ndim=2, mode="c"] fractal_array = fractal
    cdef numpy.complex128_t z
    for h in range(height):
        for w in range(width):
            z = c_array[h, w]
            for i in range(maxiterations):
                z = z**2 + c_array[h, w]
                if abs(z) > 2:
                    fractal_array[h, w] = i
                    break
    return fractal
```

Cython:

- Start mit Python oder Numpy Code

- Hier und da C Typen hinschreiben

- Cython und C Compiler aufrufen

- Python C extension Modul importieren

MANDELBROT MIT C++ AND PYBIND11

- *Rewrite in C++*
- *Python-Interface mit pybind11*
- *C++ Compiler aufrufen*
- *Python C extension Modul*

```
%%writefile run_pybind11.cpp
#include <complex>
#include <pybind11/pybind11.h>
#include <pybind11/numpy.h>
namespace py = pybind11;
void run(int height, int width, int maxiterations,
         py::array_t<std::complex<double>> np_c, py::array_t<int> np_fractal) {
    std::complex<double>* c = static_cast<std::complex<double>*>(np_c.request().ptr);
    int* fractal = static_cast<int*>(np_fractal.request().ptr);
    for (int h = 0;  h < height;  h++) {
        for (int w = 0;  w < width;  w++) {
            std::complex<double> ci = c[h + height*w];
            std::complex<double> z = ci;
            for (int i = 0;  i < maxiterations;  i++) {
                z = z*z + ci;
                if (std::abs(z) > 2) {
                    fractal[h + height*w] = i;
                    break;
}}}}}
PYBIND11_MODULE(run_pybind11, m) {
    m.def("run", &run, "the inner loop");
}
```

```
%%bash

# Compile it as a Python extension module.

c++ -Wall -shared -std=c++11 -fPIC -O3                  \
    `python -m pybind11 --includes` run_pybind11.cpp    \
    -o run_pybind11`python3-config --extension-suffix`
```

```
import run_pybind11

starttime = time.time()
height, width, maxiterations = 3200, 4800, 20
y, x = numpy.ogrid[-1:0:height*1j, -1.5:0:width*1j]
c = x + y*1j
fractal = numpy.full(c.shape, maxiterations, dtype=numpy.int32)
run_pybind11.run(height, width, maxiterations, c, fractal)
```

```python
import numba

@numba.jit
def mandelbrot(width, height):
    ...

@numba.jit
def mandelbrot_image_python(height, width):
    ...

image = mandelbrot_image_python(1200, 1600)
plt.imshow(image, cmap='gray_r')
```

Method	Setup	time (ns/px)	speedup	Cores
Python	automatic	5588.5	1×	1
Numba	automatic	111.4	50×	1
Numba-parallel	easy	33.89	165×	all (12)
Numpy	medium	359.6	15×	1
CuPy	medium	72.8	77×	GPU
Dask	medium	214.8	26×	all (12)
Numba-CUDA	difficult	6.95	800×	GPU
pybind11	challenging	165.4	34×	1
pybind11-fastmath	challenging	63.1	90×	1
Cython	maddening	1501.8	3.7×	1

MANDELBROT MIT NUMBA

- Workflow:
  - Start mit Python & Numpy Code
  - **@numba.jit** hinzufügen
  - Funktion aufrufen wievorher
- Keine neue Sprache (C, C++, Cython)
  Keine extra Dateien
  Kein C Compiler gebraucht
- Numba ist ein “Just in time” (JIT)
  Kompiler: Python → Maschinen-Code
- Einfachste und schnellste Lösung für z.B. Mandelbrot (Benchmarks: LINK, LINK)

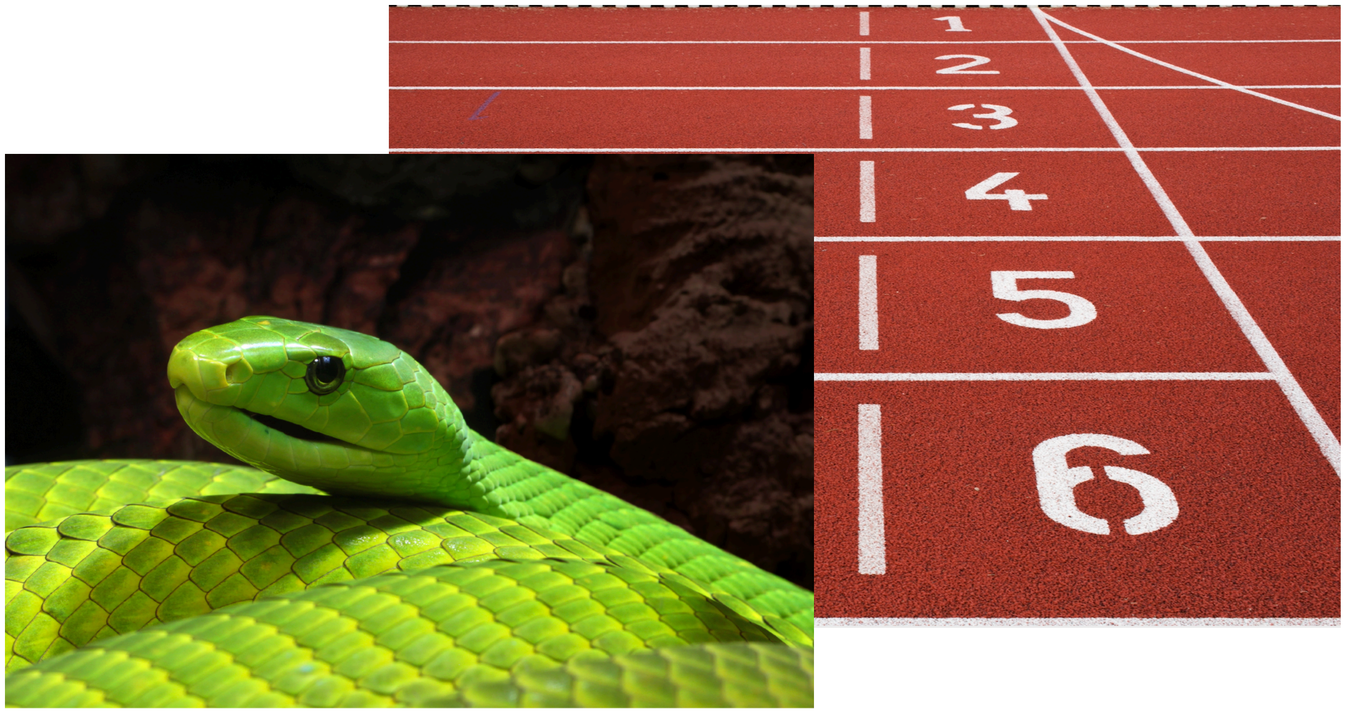
NUMBA

WAS IST NUMBA? — HTTPS://NUMBA.PYDATA.ORG

WIESO “NUMBA”?

Numba logo (https://numba.pydata.org)

“Numba” = “NumPy” + “Mamba”
Numba crunching in Python, schnell wie Mambas.



[1]:
```python
import numpy as np
```

[2]:
```python
data = np.random.uniform(-1, 1, size=1_000_000)
```

[3]:
```python
def data_reduction_python(data):
    total = 0.
    for x in data:
        if x > 0:
            total += np.sqrt(x)
    return total
```

[4]:
```python
def data_reduction_numpy(data):
    return np.sum(np.sqrt(data[data > 0]))
```

[5]:
```python
%timeit data_reduction_python(data)
```

```
851 ms
```

[6]:
```python
%timeit data_reduction_numpy(data)
```

```
7.16 ms
```

← *Numpy/Python speedup: 100x*

[7]:
```python
import numba
f = numba.jit(data_reduction_python)
%timeit f(data)
```

```
3.86 ms
```

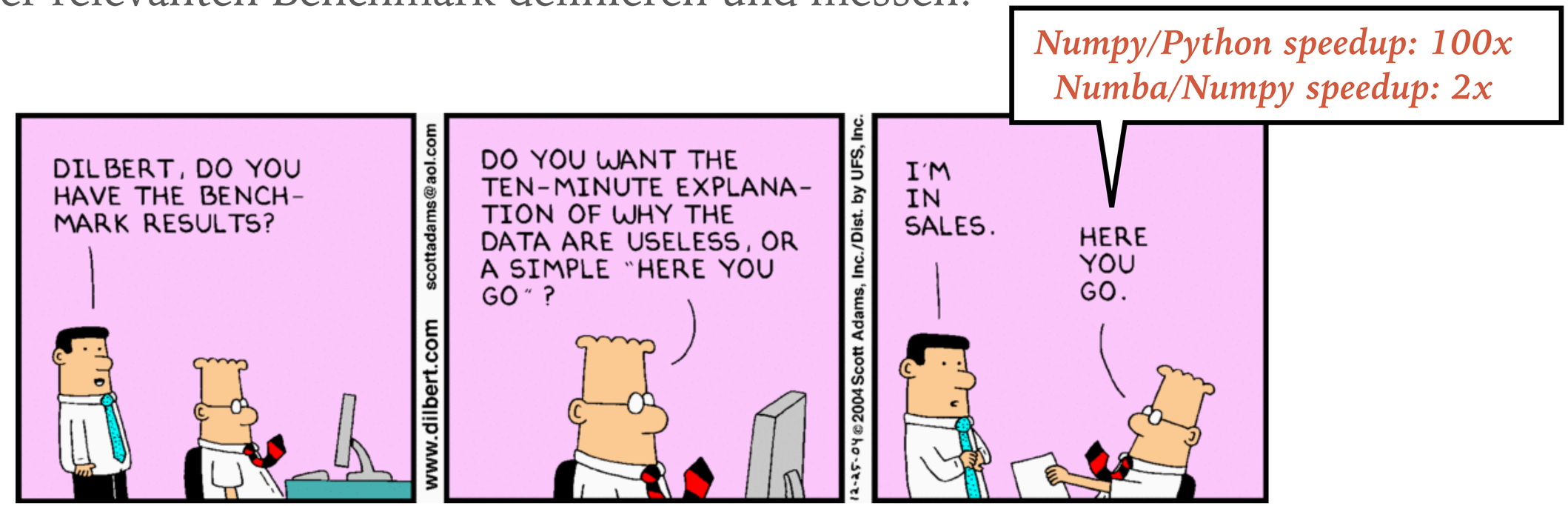
← *Numba/Numpy speedup: 2x*

NUMBA EINFÜHRUNG

- Start: Python & Numpy Code für Daten-Transformation, Reduktion, Analyse
- Falls zu langsam oder zuviel RAM: Timing & Profiling → Problem-Stelle
- Numba ausprobieren:
  - Installation
    $ **conda install numba**
    $ **pip install numba**
  - **import numba** und **@numba.jit** decorator auf Funktionen anwenden
- Klappt oft gut, aber nicht immer.
  - Nicht alles kann kompiliert werden.
  - Nicht immer sofort performant

WIE SCHNELL IST NUMBA?

- Es gibt keine sinnvolle Pauschal-Antwort! Ist für jede Anwendung anders.
- Hängt stark ab von Daten, Analyse, Code, Compiler, Python, Numpy, Hardware, und Numba Compiler Optionen: nopython, parallel, fastmath
- Immer relevanten Benchmark definieren und messen!



WAS NUMBA KOMPILIEREN KANN

```python
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

@jit(nopython=True) # Set "nopython" mode for best performance, equivalent to @njit
def go_fast(a): # Function is compiled to machine code when called the first time
    trace = 0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

WAS NUMBA NICHT KOMPILIEREN KANN

```
from numba import jit
import pandas as pd

x = {'a': [1, 2, 3], 'b': [20, 30, 40]}

@jit
def use_pandas(a): # Function will not benefit from Numba jit
    df = pd.DataFrame.from_dict(a) # Numba doesn't know about pd.DataFrame
    df += 1                        # Numba doesn't understand what this is
    return df.cov()                # or this!

print(use_pandas(x))
```

NUMBA JIT MODES

```python
import numba
```

```python
@numba.jit
def f1(x):
    return 2 * x
```

```python
f1(3)
```

```
6
```

Numba kompiliert f1 zu Maschinen-Code "no Python"

```python
@numba.jit
def f2(n):
    return n * ["spam"]
```

```python
f2(3)
```

```
['spam', 'spam', 'spam']
```

Numba kompiliert f2 zu Maschinen-Code "with Python" (Ruft CPython API)

```python
@numba.jit(nopython=True)
def f2(n):
    return n * ["spam"]
```

```python
f2(3)
```

JIT Compile und Fehler erst hier, beim Funktions-Aufruf

TypingError: Failed in nopython mode pipeline

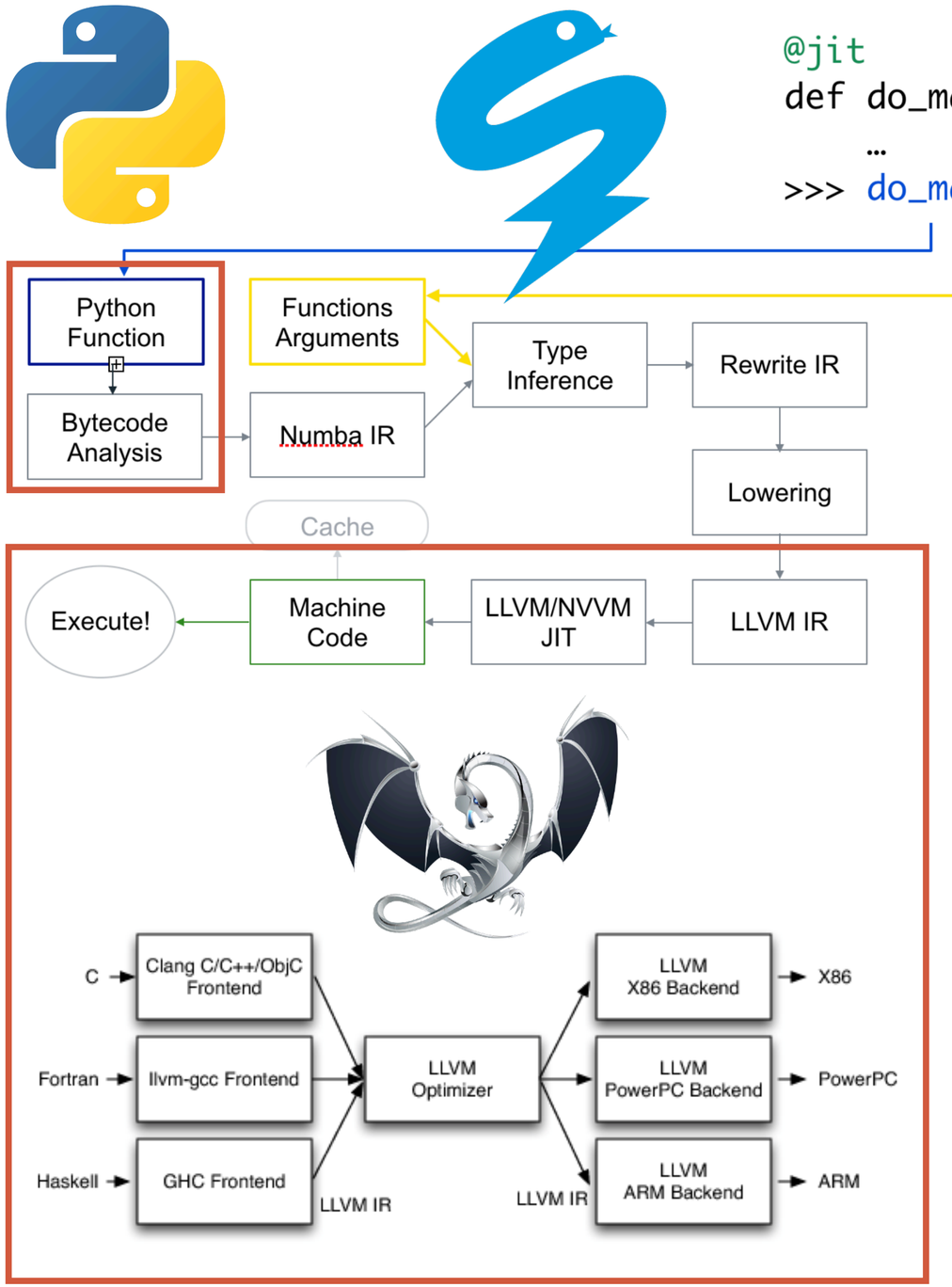
- "Object mode"
  - Der default bei **@numba.jit**
  - Versucht Python → Maschinen Code
  - Sonst Python → CPython C API Code
- "No Python mode"
  - Versucht Python → Maschinen Code
  - Sonst TypingError (erst bei Funktions-Aufruf)

```
>>> def cond():
...     x = 3
...     if x < 5:
...         return 'yes'
...     else:
...         return 'no'
...

>>> dis.dis(cond)
  2           0 LOAD_CONST               1 (3)
              3 STORE_FAST               0 (x)

  3           6 LOAD_FAST                0 (x)
              9 LOAD_CONST               2 (5)
             12 COMPARE_OP               0 (<)
             15 POP_JUMP_IF_FALSE       22

  4          18 LOAD_CONST               3 ('yes')
             21 RETURN_VALUE

  6     >>   22 LOAD_CONST               4 ('no')
             25 RETURN_VALUE
             26 LOAD_CONST               0 (None)
             29 RETURN_VALUE
```

WAS PYTHON MACHT

- Start: Python-Code (Text)
- Python-Compiler:
  - Parser → Abstract Syntax Tree (AST)
  - AST → Bytecode
- Bytecode an Funktion angehängt und Zugänglich (Code = Daten) für Numba

```
>>> cond.__code__.co_code  # the bytecode as raw bytes
b'd\x01\x00}\x00\x00|\x00\x00d\x02\x00k\x00\x00r\x16\x00d\x03\x00Sd\x04\x00Sd\x00
   \x00S'
>>> list(cond.__code__.co_code)  # the bytecode as numbers
[100, 1, 0, 125, 0, 0, 124, 0, 0, 100, 2, 0, 107, 0, 0, 114, 22, 0, 100, 3, 0, 83,
 100, 4, 0, 83, 100, 0, 0, 83]
```



WAS NUMBA MACHT

- Python: Code → Bytecode
- Numba: Bytecode → LLVM IR
- LLVM: IR → Maschinencode
  (Numba nutzt llvmlite Python Paket)

LLVM "intermediate representation" (IR) sieht so aus:

```
define i32 @add1(i32 %a, i32 %b) {
entry:
  %tmp1 = add i32 %a, %b
  ret i32 %tmp1
}

define i32 @add2(i32 %a, i32 %b) {
entry:
  %tmp1 = icmp eq i32 %a, 0
  br i1 %tmp1, label %done, label %recurse
```

Source: https://youtu.be/LLpIMRowndg

`@njit` kwargs	SVML	Execution time
`None`	No	5.95s
`None`	Yes	2.26s
`fastmath=True`	No	5.97s
`fastmath=True`	Yes	1.8s
`parallel=True`	No	1.36s
`parallel=True`	Yes	0.624s
`parallel=True, fastmath=True`	No	1.32s
`parallel=True, fastmath=True`	Yes	0.576s

PARALLEL & FASTMATH

- Numba vektorisiert: SSE, AVX, AVX-512
- Numba kann parallele CPU Threads auf drei Arten: tbb, openmp, workqueue
- Intel Threading Building Blocks (TBB)
  $ conda install -c tbb
- Intel Short Vector Math Library (SVML)
  $ conda install -c numba icc_rt
- Siehe Numba Docs: "Performance Tips" und "Threading Layers"
- Beispiel links auf Intel 4-Core CPU zeigt 10x Unterschiede in Laufzeit

```
$ numba -s
__Hardware Information__
Machine                                   : x86_64
CPU Name                                  : haswell
CPU count                                 : 8
CPU Features                              :
aes avx avx2 bmi bmi2 cmov cx16 f16c fma fsgsbase invpcid lzcnt mmx movbe pclmul
popcnt rdrnd sahf sse sse2 sse3 sse4.1 sse4.2 ssse3 xsave xsaveopt

__OS Information__
Platform                                  : Darwin-18.5.0-x86_64-i386-64bit

__Python Information__
Python Compiler                           : Clang 4.0.1 (tags/RELEASE_401/final)
Python Implementation                     : CPython
Python Version                            : 3.7.3

__LLVM information__
LLVM version                              : 7.0.0

__CUDA Information__
CUDA driver library cannot be found or no CUDA enabled devices are present.

__ROC Information__
ROC available                             : False

__SVML Information__
SVML operational                          : True

__Threading Layer Information__
TBB Threading layer available             : True
OpenMP Threading layer available          : True
Workqueue Threading layer available       : True
```

NUMBA -S

- Command line tool:
  “**numba -s**” oder “**numba —sysinfo**”
- Von IPython oder Jupyter: “**!numba -s**”
- CPU & GPU Informationen
- Compiler-Informationen
- Library-Informationen: SVML oder TBB
- Numba Compile Debug Hilfen:

```
$ numba myscript.py --annotate
$ numba myscript.py --annotate-html myscript.html
$ numba myscript.py --dump-llvm
$ numba myscript.py --dump-optimized
$ numba myscript.py --dump-assembly
```

NUMBA KANN NOCH MEHR . . .

- **@numba.vectorize** und **@numba.gvectorize** — Numpy UFuncs machen
- **@numba.jitclass** — Python-Klassen (nur ganz einfache Fälle)
- **@numba.stencil** — Kernel-Faltungen
- **@numba.cfunc** — C callbacks
- **numba.pycc.CC** — C extension
- Ahead of time (AOT) Kompilieren & Disk Caching
- NVIDIA CUDA GPU: **conda install cudatoolkit** und **@numba.cuda.jit**
- AMD ROC GPU: **conda install -c numba roctools** und **@numba.roc.jit**

INTEL HPAT — HIGH PERFORMANCE ANALYTICS TOOLKIT

- Baut auf Numba auf, fügt CPU Cluster support via MPI hinzu
  und auch ein bisschen pandas & HDF5 support
- *“High Performance Analytics Toolkit (HPAT) scales analytics/ML codes in Python to bare-metal cluster/cloud performance automatically. It compiles a subset of Python (Pandas/Numpy) to efficient parallel binaries with MPI, requiring only minimal code changes. HPAT is orders of magnitude faster than alternatives like Apache Spark.” — https://github.com/IntelLabs/hpat*

```
@hpat.jit
def logistic_regression(iterations):
    f = h5py.File("lr.hdf5", "r")
    X = f['points'][:]
    Y = f['responses'][:]
    D = X.shape[1]
    w = np.random.ranf(D)
    t1 = time.time()
    for i in range(iterations):
        z = ((1.0 / (1.0 + np.exp(-Y * np.dot(X, w))) - 1.0) * Y)
        w -= np.dot(z, X)
    return w
```

ZUSAMMENFASSUNG

- Numba ist ein Python & Numpy JIT Compiler, via LLVM zu Maschinen-Code
- Kompiliert einzelne ausgewählte Funktionen, nicht ganze Programme wie PyPy
- Einfach zu bedienen: @numba.jit mit Optionen “nopython”, “parallel”, “fastmath”
- Support für Multi-Core CPU (x86, ARM, Power), und auch GPU
- Support für Linux, macOS, Windows
- In Anaconda Python enthalten
- Aktiv entwickelt seit 2012, releases alle 2 Monate (aktuell: 0.43.1)
- Version 1.0 “bald” — aber schon gut geeignet für Data Science Projekte

INTERESSE AN NUMBA? — HTTPS://NUMBA.PYDATA.ORG/