

# Einführung in Cython

Jonathan Helgert  
jonathan.helgert@posteo.de

[https://github.com/jhelgert/cython\\_einfuehrung](https://github.com/jhelgert/cython_einfuehrung)

15. Oktober 2021



# Übersicht

## Motivation

Wie Cython verwenden?

Statisches Typsystem

Codeprofiling

Typed Memoryviews

Extratuning

Casestudy

# Einstiegsbeispiel

Wir suchen die Anzahl an Pythagorastripeln, d.h. wie viele  $(a, b, c) \in \mathbb{Z}^3$  mit  $a^2 + b^2 = c^2$  und  $0 < a < b < c \leq N$  gibt es für ein gegebenes  $N$ ?

```
// -- C++ --
int count_triples(const int N){
    int found = 0;
    for(int a = 1; a <= N; ++a){
        for(int b = a+1; b <= N; ++b){
            for(int c = b+1; c <= N; ++c){
                if(a*a + b*b == c*c){
                    found++;
                }
            }
        }
    }
    return found;
}

// N = 10 000, Timing: 12.2s
```

```
# -- Python --
def count_triples(N):
    found = 0
    for a in range(1, N+1):
        for b in range(a+1, N+1):
            for c in range(b+1, N+1):
                if a*a + b*b == c*c:
                    found += 1
    return found

# N = 10 000, Timing: 21045s = 5h 50min
```

**Takeaway 1.** *Pythonschleifen sind im Vergleich zu low-level Sprachen (C, C++, Fortran) sehr langsam.*

# Einstiegsbeispiel

Und mit numpy?

```
# -- Python --
```

```
import numpy as np
```

```
def count_triples(N):
```

```
    a = np.arange(1, N+1, dtype=np.int32)
```

```
    b = np.arange(1, N+1, dtype=np.int32)
```

```
    c = np.arange(1, N+1, dtype=np.int32)
```

```
    ab = a[:, None]**2 + b[None, :]**2
```

```
    ab_updiag = ab[np.triu_indices_from(ab)]
```

```
    return np.sum(np.isin(ab_updiag, c**2))
```

```
# N = 10 000, Timing: 15.1s
```

# Motivation

*“What makes Python fast for development is what makes Python slow for code execution”*

- Jake VanderPlas.

## Python in a nutshell:

- High-Level Sprache: Programm wird von einem Interpreter (verbreitetester: CPython) ausgeführt.
- Dynamisches Typsystem: Typ einer Variablen wird erst bei der Verwendung überprüft.
- *Dynamic Dispatching* für jede Operation und Funktionsaufruf.
- Umfangreiche Fehlerkontrolle.

## Aber numpy, scipy, tensorflow etc. sind nicht langsam?

- Laufzeitkritischer Code wird in C/C++/Fortran geschrieben und eingebunden.
- Oder direkt in Cython geschrieben.

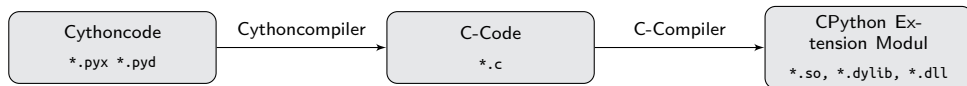
**Einige Projekte, welche Cython verwenden:** Sage, Scipy, Numpy, Pandas, scikit-learn, scikit-image, lxml, mpi4py, petsc4py, cyipopt, ...

# Was ist Cython?

*"Cython gives you the combined power of Python and C."*

- cython.org

- Eine Programmiersprache, die Python um das statische Typsystem von C und C++ erweitert.
- Ein "Compiler", der Cythonquellcode in effizienten C oder C++ Code umwandelt, der wiederum in ein CPython Extension Modul kompiliert wird.
- Beachte: Auch unveränderter Pythoncode kann vom Cythoncompiler kompiliert werden.



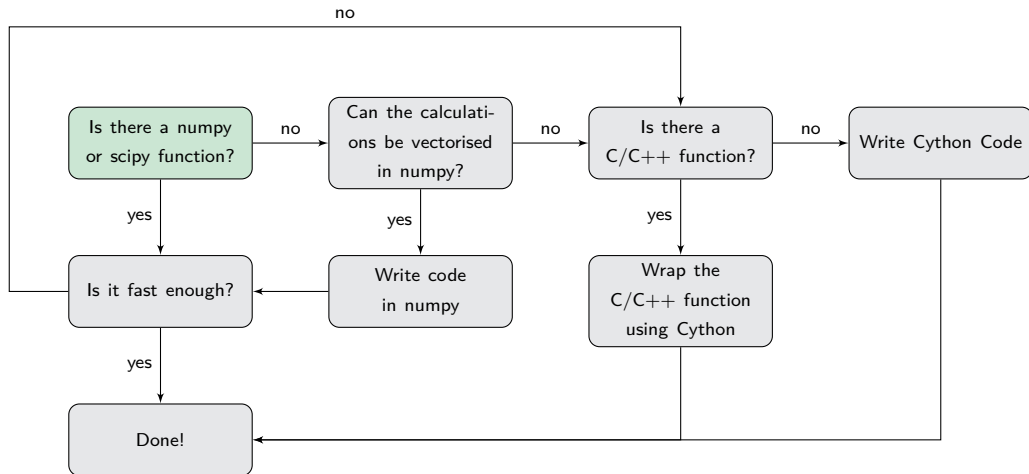
## Anwendungsfälle:

1. Performanceoptimierung von Pythoncode.
2. *Wrapping* von vorhandenem C oder C++ in Python.

# Python, Cython oder C/C++?

*"Premature optimization is the root of all evil."*

- Donald Knuth.



Quelle: <https://blt.ly/3hByQfr>

# Übersicht

Motivation

Wie Cython verwenden?

Statisches Typsystem

Codeprofiling

Typed Memoryviews

Extratuning

Casestudy



# Wie Code compilieren und ausführen?

## Möglichkeit 1: Jupyter Notebook

- Innerhalb eines Jupyternotebooks `%load_ext cython` um bequem innerhalb eines Notebooks Cython verwenden zu können.
- Nun kann man das *Magiccommand* `%%cython` innerhalb eines Codeblocks verwenden. Die im Codeblock enthaltenen Funktionen sind anschließend automatisch im aktuellen Notebook importiert.
- Tipp: Sehr nützlich ist die *annotate* option, d.h. `%%cython -a`.

## Möglichkeit 2: Eigenes Package (Teil 2)

- Schreibe eine `helloworld.pyx` Datei mit dem Cythoncode.
- Schreibe eine `setup.py` Datei und nutze `setuptools`.
- Das Paket via `python3 setup.py build_ext --inplace` lokal im selben Verzeichnis installieren.
- Das Modul kann nun innerhalb des gleichen Verzeichnisses importiert werden.

# Übersicht

Motivation

Wie Cython verwenden?

Statisches Typsystem

Codeprofiling

Typed Memoryviews

Extratuning

Casestudy

# Datentypen

Python besitzt nur die beiden reellen Zahlentypen `int` und `float`. Cython unterstützt alle C-Typen:

	Größe	Suffix
[unsigned] char	8 Bit (1 Byte)	[U]
[unsigned] short	16 Bit (2 Byte)	[U]
[unsigned] int	32 Bit (4 Byte)	[U]
[unsigned] long	64 Bit (8 Byte)	[U]L
[unsigned] long long	64 Bit (8 Byte)	[U]LL

	Größe
float	32 Bit (4 Byte)
double	64 Bit (8 Byte)
long double	min. 80 Bit (10 Byte)

**Tabelle:** Größe der Gleitkomma-Datentypen.

**Tabelle:** Größe der ganzzahligen Datentypen.

- Wertebereich für ganzzahligen Datentypen  $[0, 2^n - 1]$  für vorzeichenlose (unsigned) bzw.  $[-2^{n-1}, 2^{n-1} - 1]$  für vorzeichenbehaftet, wobei  $n$  die Anzahl der Bits ist.
- In Python sind **ints** unbeschränkt! Python's **float** entspricht **double** in C.
- Der boolsche C-Datentyp **bool** heißt in Cython `bint`.

**Takeaway 2.** *Die C-Typen sind deutlich schneller, haben allerdings auch C-Semantik (Overflow bei ganzzahligen Datentypen und unterschiedliche Größe je nach Plattform)*

# Datentypen

- Reguläre Pythonvariablen haben dynamischen Typ und können genau so in Cython verwendet werden. Der Typ heißt in Cython **object**.
- Mit dem Schlüsselwort **cdef** können wir innerhalb von Cython Variablen auf C-Level deklarieren.

```
# -- Cython --
```

```
a = 2                                # ein Python-int (unbeschränkt)
b = 2**31                            # = 2147483648
cdef int      c = 2**31            # ein C-int
cdef int      d = 2147483648       # = -2147483648, da Overflow :(
cdef long     e = 2147483648L      # = 2147483648
cdef unsigned int h = 2147483648U  # = 2147483648
cdef unsigned long f = 2147483648UL # = 2147483648

x = 0.1                              # Python float (64 Bit)
cdef float    y = 0.1              # C float (32 Bit)
cdef double   z = 0.1              # C double (64 Bit)
cdef long double q = 0.1           # C long double (min. 80 Bits)

cdef bint f = True                  # boolscher Typ
```

# Eigene Datentypen

- Cython bietet die structs aus C, um unterschiedliche Datentypen zu einer Struktur zusammenzufassen, die als neuer Datentyp genutzt werden kann.

```
# -- Cython --
```

```
ctypedef struct my_pair:
```

```
    int a
```

```
    int b
```

```
ctypedef struct my_struct:
```

```
    my_pair p
```

```
    double c
```

```
cdef my_pair  p1 = my_pair(1, 2)
```

```
cdef my_struct s1 = my_struct(p1, 0.1)
```

## cimport vs import

Mit Hilfe von **cimport** können wir C-Funktionen, C-structs, C-variablen und Extension Types einbinden.

- Ein **cimport** bindet das Modul und die darin enthaltenen Funktion über dessen Cythonschnittstelle auf C-Level ein statt auf Pythonlevel (deutlich weniger Overhead).
- Einige Scipy packages bieten Cythonmodule an: `scipy.linalg.cython_blas`, `scipy.linalg.cython_lapack`, `scipy.optimize.cython_optimize`, `scipy.special.cython_special`
- Vorteil: Diese Funktionen haben deutlich weniger Overhead.

Beispiele:

```
# -- Cython --  
import numpy as np    # Benutzt numpy über Python  
cimport numpy as np  # Benutzt direkt die C-API für numpy  
from scipy.linalg.cython_blas cimport dgemv    # Matrixvektorprodukt  
from scipy.linalg.cython_lapack cimport dsgevs # LGS-Loeser  
from scipy.optimize.cython_optimize cimport bisect # Nullstelle via Bisektion  
from scipy.special.cython_special cimport gamma # Gammafunktion
```

## cimport vs import

Beispiel: In C gibt es standardisierte ganzzahlige Typen innerhalb von `stdint.h`. Diese stehen in Cython innerhalb von `libc.stdint` zur Verfügung.

```
# -- Cython --
```

```
from libc.stdint cimport *
```

```
cdef int8_t a = 1 # signed char, 8 Bit
```

```
cdef int16_t b = 1 # signed short, 16 Bit
```

```
cdef int32_t c = 1 # signed int, 32 Bit
```

```
cdef int64_t d = 1 # signed long long, 64 Bit
```

```
cdef uint8_t e = 1 # unsigned char, 8 Bit
```

```
cdef uint16_t f = 1 # unsigned short, 16 Bit
```

```
cdef uint32_t g = 1 # unsigned int, 32 Bit
```

```
cdef uint64_t h = 1 # unsigned long long, 64 Bit
```

# Datencontainer

- Cython bietet auch Pythons Datencontainer list, dict, tuple für statisches Typing an.
- Zu Pythons **tuple** gibt es ein effizientes Tuple auf C-Level, welches intern einer simplen C **struct** entspricht.

Beispiel:

```
# -- Cython --
```

```
cdef (int, int)    tup1                # tup1 ist ein C-tuple aus zwei ints.
cdef (int, double) tup2 = (1, 2.0)     # C-tuple aus int und double
cdef list         l1 = ["hi", 2, 3.0]  # eine liste
cdef dict         d1 = {"key" : 2}     # ein dictionary
cdef int[3]       carr = [1, 2, 3]    # ein C-Array (andere Syntax als in C!)
```



# Datencontainer

- Zudem sind auch die gängigsten C++ STL Container innerhalb von `libc++` verfügbar, z.B. `std::list<T>`, `std::map<K,T>`, `std::pair<T1,T2>` und `std::vector<T>`.
- Das Spezifizieren der Templates erfolgt via `Container[typ]`.
- Der Code muss dann mittels C++ statt C Compiler kompiliert werden.

Beispiel:

```
# -- Cython --
# distutils: language=c++
from libc++.vector cimport vector

ctypedef struct my_pair:
    int a
    double b

cdef vector[int]    v1 = [1, 2, 3]                # ein vector aus ints
cdef vector[double] v2 = vector[double](3, 10)    # ruft entsprechenden Konstruktor auf
cdef vector[my_pair] v3                          # ein vector aus my_pairs
cdef double el

for el in v2:                                     # Bekannte Pythonsyntax trotz C++ Vector
    # .. mache etwas mit el ...
```

## Funktionen: **def**, **cdef** und **cpdef**

Es gibt in Cython drei mögliche Schlüsselwörter um eine Funktion zu definieren: **def**, **cdef** und **cpdef**.

- **def**: Normale Pythonfunktion.
- **cdef**: C-Funktion. Parameter und Rückgabewert sind C-Variablen oder Pythonobjekte. Vorteil: Kein Overhead. Nachteil: Nicht so flexibel wie Pythonfunktionen, können nicht außerhalb eines Cythonmoduls aufgerufen werden.
- **cpdef**: Es wird automatisch eine **cdef** und **def** Funktion erstellt. Kann von überall aufgerufen werden. Innerhalb von Cython wird immer die C Funktion aufgerufen.

```
def func1(x):           # Kann innerhalb von Python aufgerufen werden.
```

```
    cdef int y = x
    return y*y
```

```
cdef int func2(int x): # Kann NICHT innerhalb von Python aufgerufen werden.
```

```
    return x*x
```

```
def func3(int x):       # Cython erstellt automatisch func1
```

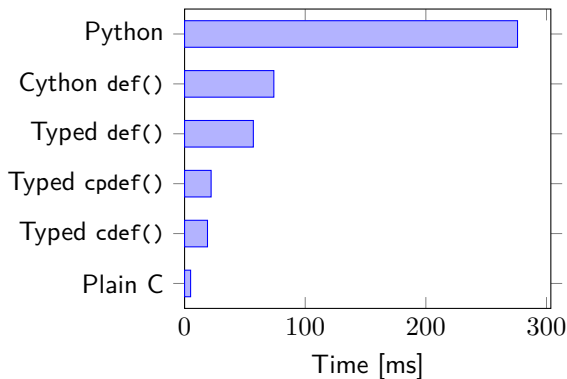
```
    return x*x
```

```
cpdef func4(int x):    # Cython erstellt automatisch func1 und func2
```

```
    return x*x
```

## Wie schnell sind **def**, **cdef**, bzw. **cpdef**?

Trivialer Benchmark: Rekursive Implementierung der Fibonaccifolge. Aufruf `fib(30)`:



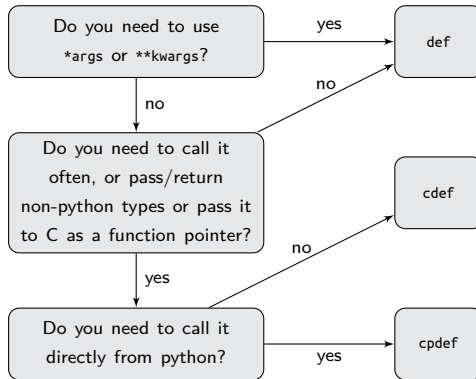
**Takeaway 3.** *Pythonfunktionen haben großen Overhead.*

```
# -- Python / Cython --  
def fib(N):  
    if N < 2:  
        return N  
    return fib(N-2) + fib(N-1)
```

```
# -- Cython --  
def int fib(int N):  
    if N < 2:  
        return N  
    return fib(N-2) + fib(N-1)
```

```
# -- Cython --  
cpdef int fib(int N):  
    if N < 2:  
        return N  
    return fib(N-2) + fib(N-1)
```

## Wann `def`, `cdef` oder `cpdef`?



Quelle: <https://bit.ly/334YioK>

**Takeaway 4.** Sobald eine Funktion aufgerufen wurde, besteht für den Code innerhalb der Funktion kein Performanceunterschied zwischen `def`, `cdef` und `cpdef`.

# C Libraries

- Cython hat die Header einiger C Standardlibraries bereits innerhalb von `libc` implementiert, z.B. `stdio.h`, `stdlib.h`, `math.h`, `complex.h`

Beispiel:

```
# -- Cython --
from libc.stdio cimport printf
from libc.math cimport cos as c_cos

# x sei ein np.array mit shape (N,)
def print_my_c_cos(x):
    cdef int N = x.size
    cdef int i
    for i in range(N):
        printf("Printed with C function: %lf\n", c_cos(x[i]))
        print(f"Printed with Python {c_cos(x[i])}")
```

# C Libraries

- Cython hat die Header einiger C Standardlibraries bereits innerhalb von `libc` implementiert, z.B. `stdio.h`, `stdlib.h`, `math.h`, `complex.h`

Beispiel:

```
# -- Cython --
from libc.stdio cimport printf
from libc.math cimport cos as c_cos

# x sei ein np.array mit shape (N,)
def print_my_c_cos(x):
    cdef int N = x.size
    cdef int i
    for i in range(N):
        printf("Printed with C function: %lf\n", c_cos(x[i]))
        print(f"Printed with Python {c_cos(x[i])}")
```

Wir können also innerhalb von Cython direkt C schreiben und gleichzeitig die Vorteile von Python genießen.

# C++ Standard Template Library

- Neben den Containern sind auch die meisten Libraries der C++ Standard Template Library (STL) verfügbar, z.B. `algorithm`, `iterator` und `numeric`.

Beispiel:

```
# -- Cython --
#distutils: language = c++

from libcpp.vector cimport vector
from libcpp.algorithm cimport count_if          # ab Cython 3.0.x
from libcpp.numeric cimport iota, inner_product # ab Cython 3.0.x

cdef vector[double] v1 = vector[double](10)    # Speicher für 10 Einträge
cdef vector[double] v2 = vector[double](10, 1) # [1.0, 1.0, ..., 1]

cdef bint is_odd(int i): return i % 2

iota(v1.begin(), v1.end(), 1)                  # Füllt v1 mit 1.0, 2.0, 3.0, ..., 10.0
tmp = inner_product(v1.begin(), v1.end(), v2.begin(), 0) # berechnet Skalarprodukt aus v1, v2
tmp = count_if(v1.begin(), v1.end(), is_odd)         # wie viele Einträge sind ungerade?
```

# Fused Types

Angelehnt an C++-Templates, bietet Cython *fused types* für Funktionsparameter.

- Das Erstellen der passenden Funktion (*Dispatching*) findet zur Compilezeit statt, nicht zur Laufzeit.
- Einschränkung: Wird der gleiche Fused Type mehrmals in den Funktionsargumenten verwendet, müssen diese Argumente auch den gleichen Typ haben.
- Cython bietet drei vordefinierte Fused Types: `integral`, `floating` und `numeric`.

```
# -- Cython --
# Deklarationen von fused types
ctypedef fused T1:
    int
    double

ctypedef fused T2:
    T1
    float

from cython cimport floating
```

```
# Fused Types für Funktionsargumente:
cdef f(T1 arg1, T1 arg2, T2 arg3):
    # do something

# Auch für Memoryviews
def g(floating[:, ::1] A, floating[:, ::1] b):
    # ...

f(1, 1.0, 1.0) # kompiliert nicht!
f(1, 1, 1.0)   # f(int, int, double)
f(1.0, 1.0, 1) # f(double, double, int)
```

**Takeaway 5.** Verwende *Fused Types* für generischen Cythoncode.



# Übersicht

Motivation

Wie Cython verwenden?

Statisches Typsystem

Codeprofiling

Typed Memoryviews

Extratuning

Casestudy

# Code Profiling via Line Profiler

Jetzt: Wie können wir genau herausfinden, wie viel Zeit welche Abschnitte unseres Codes in Anspruch nehmen?

- Die Syntax innerhalb eines Notebooks ist

```
%lprun -f name_der_funktion name_der_funktion(*args)
```

- Um Cythoncode *profilen* zu können, müssen wir ihn mit entsprechenden Directives und Flags kompilieren.
- Innerhalb eines Notebooks muss dazu

```
# cython: linetrace=True, binding=True
```

```
# distutils: define_macros=CYTHON_TRACE_NOGIL=1
```

in die obersten Zeilen der entsprechende Zelle mit Cythoncode.

**Takeaway 6.** *Kompilieren mit Profilingflags verlangsamt den Code deutlich. Daher niemals für Endcode verwenden.*

## Zurück zum Einstiegsbeispiel

Wir suchen die Anzahl an Pythagorastripeln, d.h. wie viele  $(a, b, c) \in \mathbb{Z}^3$  mit  $a^2 + b^2 = c^2$  und  $0 < a < b < c \leq N$  gibt es für ein gegebenes  $N$ ?

```
# -- Python --
```

```
def count_triples(N):  
    found = 0  
    for a in range(1, N+1):  
        for b in range(a+1, N+1):  
            for c in range(b+1, N+1):  
                if a*a + b*b == c*c:  
                    found += 1  
    return found
```

```
# -- Cython --
```

```
def count_triples(int N):  
    cdef int a, b, c  
    cdef int found = 0  
    for a in range(1, N+1):  
        for b in range(a+1, N+1):  
            for c in range(b+1, N+1):  
                if a*a + b*b == c*c:  
                    found += 1  
    return found
```

# Übersicht

Motivation

Wie Cython verwenden?

Statisches Typsystem

Codeprofiling

Typed Memoryviews

Extratuning

Casestudy

# Typed Memoryviews

Ein *Typed Memoryview* ermöglicht uns effizienten Zugriff auf Datenbuffer.

- Unterstützt unter anderem `np.arrays`, `cython.arrays` und C-Arrays.
- Indizieren, Kopieren, Transponieren und Broadcasting wird unterstützt.
- Beinhaltet zudem einige Attribute mit Informationen über den Datenbuffer, z.B. `shape`, `size`, `nbytes` oder `base`.

Wir beschränken uns vorerst auf Memoryviews auf `np.arrays`:

```
# -- Cython --
```

```
cdef int[:] a = np.arange(20, dtype=np.int32)           # a.shape: (20,)
cdef float[:, :] A = np.random.rand(10, 9, dtype=np.float32) # A.Shape: (10, 9)
cdef double[:, :, :] C = np.random.rand(10, 3, 5)        # C.Shape: (10, 3, 5)
```

**Takeaway 7.** Verwende *Typed Memoryviews* für effizienten Zugriff auf `np.arrays` und vermeide Schleifen über `np.arrays`, falls möglich.

# Typed Memoryviews

Numpy Typ	C stdint	C Typ
<code>np.int8</code>	<code>int8_t</code>	<code>char</code>
<code>np.int16</code>	<code>int16_t</code>	<code>short</code>
<code>np.int32</code>	<code>int32_t</code>	<code>int</code>
<code>np.int64</code>	<code>int64_t</code>	<code>long [long]</code>
<code>np.uint8</code>	<code>uint8_t</code>	<code>unsigned char</code>
<code>np.uint16</code>	<code>uint16_t</code>	<code>unsigned short</code>
<code>np.uint32</code>	<code>uint32_t</code>	<code>unsigned int</code>
<code>np.uint64</code>	<code>uint64_t</code>	<code>unsigned long [long]</code>
<code>np.float32</code>		<code>float</code>
<code>np.float64</code>		<code>double</code>

**Tabelle:** Übersicht der realen Numpy dtypes.

- Für ganzzahlige Arrays ist der Defaulttyp `np.int64`, für Arrays mit Gleitkommazahlen dagegen `np.float64`.

**Takeaway 8.** Verwende für `np.arrays` den richtigen dtype. Tipp: `.astype()` Methode für Umwandlungen.

# Typed Memoryviews

```
cdef double[:, :] mv1 = np.random.rand(10, 20)
```

```
cdef double[:, :] y1, y2, y3, y4
```

*# Zuweisungen und Kopien:*

```
y1 = mv1           # ACHTUNG: y ist ein View auf mv1 (d.h. teilen sich den Datenbuffer)
```

```
y2[:, :] = mv1     # Kopiert Werte von mv1 in y
```

```
y3[:, :] = mv1.copy() # Kopiert ebenfalls
```

```
y4[:, :] = 1.0     # funktioniert auch für Skalare
```

*# Transponieren*

```
cdef double[:, :] mv2 = mv1.T # Transponierte, Achtung: View (genau wie bei einem np.array)
```

*# Slicing über einen Index/eine Dimension via start:end:step*

```
cdef double[:] mv3 = mv1[:, 0]           # View auf Spalte 0 von mv1
```

```
cdef double[:] mv4 = mv1[:, 0].copy()    # Kopie der Spalte 0 von mv1
```

```
cdef double[:] mv5 = mv1[3, 0:20:2]      # View auf jedes 2. Element aus Zeile 3
```

*# Broadcasting*

```
cdef double[:] mv6 = np.ones((10, ))
```

```
cdef double[:, :] mv7 = mv6[None, :].copy() # shape (1, 10)
```

# Typed Memoryviews

- **Achtung:** Ein Memoryview ist kein `np.array`. Man muss es also ausdrücklich in ein `np.array` umwandeln bei der Rückgabe an Python.
- Die Funktion `np.asarray` erkennt dann `shape` und `Typ` des Memoryviews.

Beispiel:

```
# -- Cython --
```

```
cimport numpy as np
```

```
import numpy as np
```

```
def cy_ones(N):
```

```
    cdef double[:] B = np.empty(N)
```

```
    B[:] = 1.0
```

```
    return np.asarray(B) # <--- Wichtig, shape (N,)
```

```
def cy_one_matrix(N, N):
```

```
    cdef double[:, :] B = np.empty((N, N))
```

```
    B[:, :] = 1.0
```

```
    return np.asarray(B) # <--- Wichtig, shape (N, N)
```



# Typed Memoryviews und C/C++

- Ein Pointer auf die C-Standardtypen kann durch einen *Typcast* in ein Memoryview umgewandelt werden.
- Die Adresse des ersten Elements eines Memoryviews ergibt wieder den ursprünglichen Pointer.

```
# -- Cython --
```

```
from libc.stdlib cimport calloc, free
```

```
cdef double* ptr1 = <double*> calloc(N, sizeof(double))
```

```
cdef double[:] mv1 = <double[:N]> ptr1 # Typcast des Pointers in ein MV mit shape (N,)
```

```
cdef double* ptr2 = &mv1[0]
```

```
cdef int[:, :] mv1 = <int[:M, :N]> calloc(M*N, sizeof(int)) # Typcast in ein MV mit shape (M, N)
```

```
cdef int* ptr4 = &mv2[0, 0]
```

```
# Freigeben des dynamisch angelegten Speichers (Wichtig!)
```

```
free(&mv1[0]) # Identisch zu free(ptr1)
```

```
free(&mv2[0, 0]) # Identisch zu free(ptr4)
```

# Typed Memoryviews und C/C++

Auch ein C++ `std::vector<T>` kann (ohne Kopie) in ein Memoryview umgewandelt werden:

```
# -- Cython (C++ Mode) --
from libcpp.vector cimport vector

cdef int          N = 2
cdef vector[double] v = vector[double](N*N, 1.0) # [1.0, 1.0, 1.0, 1.0]
cdef double[:]    mv1 = <double[:N]>(&v[0])      # MV mit shape (4,)
cdef double[:, :] mv2 = <double[:N, :N]>(&v[0])   # MV mit shape (2, 2)
                mv1[:] = 1.0                      # Verändern der Daten per MV
                mv2[:, :] = 2.0
```

Umgekehrt geht das ganze nur mit einer Kopie der Daten (Limitierung seitens `std::vector`):

```
cdef double[:] a = np.arange(5).astype(np.float64) # [0.0, 1.0, 2.0, 3.0, 4.0]
cdef vector[double] v1
cdef vector[double] v2
v1.assign(&a[0], &a[-1] + 1) # Pointerarithmetik
v2.assign(&a[0], &a[0] + <long>a.size) # Alternative
```

# Memorylayout

Beispiel 1:

Sei  $A \in \mathbb{R}^{1000 \times 1000}$  und  $x \in \mathbb{R}^{1000}$ . Wir berechnen nun  $y_1 = Ax$  und  $y_2 = A^T x$ .

<i># Python: (A und x sind entsprechende np.arrays)</i>	<i># R</i>
<code>At = A.T</code> <i># Transponierte von A</i>	<code>At &lt;- t(A);</code>
<code>y1 = A @ x</code> <i># Timing: 31.5 <math>\mu</math>s</i>	<code>y1 &lt;- A %*% x; # Timing: 73.1 <math>\mu</math>s</code>
<code>y2 = At @ x</code> <i># Timing: 82.2 <math>\mu</math>s</i>	<code>y2 &lt;- At %*% x; # Timing: 42.8 <math>\mu</math>s</code>

Wie lässt sich dieser Effekt erklären?  $\implies$  Unterschiedliche Memorylayouts und Cache-(un)friendly Code.

# Memorylayout

Arrays werden in C/C++ zeilenweise ohne Lücken gespeichert. In anderen Sprachen wie Fortran, Julia oder Matlab dagegen spaltenweise. Anschaulich:

$$A_{\text{row-major}} = \begin{pmatrix} \boxed{1} & \boxed{2} & \boxed{3} \\ \boxed{4} & \boxed{5} & \boxed{6} \\ \boxed{7} & \boxed{8} & \boxed{9} \end{pmatrix}, \quad B_{\text{column-major}} = \begin{pmatrix} \boxed{1} & \boxed{2} & \boxed{3} \\ \boxed{4} & \boxed{5} & \boxed{6} \\ \boxed{7} & \boxed{8} & \boxed{9} \end{pmatrix}$$

Im Speicher sieht das jeweils so aus:

$$\begin{aligned} A_{\text{row-major}} : \dots & \boxed{1} \boxed{2} \boxed{3} \boxed{4} \boxed{5} \boxed{6} \boxed{7} \boxed{8} \boxed{9} \dots \\ B_{\text{column-major}} : \dots & \boxed{1} \boxed{4} \boxed{7} \boxed{2} \boxed{5} \boxed{8} \boxed{3} \boxed{6} \boxed{9} \dots \end{aligned}$$

Setzen wir `::1` bei der Deklaration eines Memoryviews am letzten Index, ist das Memorylayout des Memoryviews row-major. Setzen wir es am ersten Index, ist es column-major:

```
# -- Cython --
```

```
cdef long[:, ::1] A = np.array(np.arange(1, 10).reshape(3,3))           # row-major (default)
cdef long[:, :1] B = np.array(np.arange(1, 10).reshape(3,3), order="F") # column-major
```

**Takeaway 9.** Die Daten in einem `np.array` sind standardmäßig zeilenweise gespeichert.

# Memorylayout

**Achtung:** ein np.array muss nicht zwingend lückenlos als C-contiguous gespeichert sein:

```
# -- Cython --
```

```
cdef long[:,1] a = np.arange(1, 10) # row- & columnmajor
```

```
cdef long[:,1] b = a[0:10:3]          # Slicing: jedes dritte Element von a, KOMPILIERT NICHT!
```

Im Speicher:

a: ... 1 2 3 4 5 6 7 8 9 ...

b: ... 1 2 3 4 5 6 7 8 9 ...

Das array b ist ein View auf a und offensichtlich nicht c-contiguous. Also:

```
# -- Cython --
```

```
cdef long[:] b = a[0:10:3]
```

```
# oder:
```

```
cdef long[:,1] b = np.ascontiguousarray(a[0:10:3])
```

# Memorylayout

- Nützliche Funktionen: `np.ascontiguousarray` und `np.asfortranarray`.
- Das `.flags` Attribut liefert Infos über ein `np.array`

*# .flags eines np.array*

<code>C_CONTIGUOUS : True</code>	<i># row-major?</i>
<code>F_CONTIGUOUS : False</code>	<i># column-major?</i>
<code>OWNDATA : True</code>	<i># View ja/nein ?</i>
<code>WRITEABLE : True</code>	<i># Schreibrechte</i>
<code>ALIGNED : True</code>	<i># Memory alignment</i>
<code>WRITEBACKIFCOPY : False</code>	<i># ...</i>
<code>UPDATEIFCOPY : False</code>	<i># ...</i>

# Memorylayout

- Nützliche Funktionen: `np.ascontiguousarray` und `np.asfortranarray`.
- Das `.flags` Attribut liefert Infos über ein `np.array`

*# .flags eines np.array*

```
C_CONTIGUOUS : True      # row-major?
F_CONTIGUOUS : False     # column-major?
OWNDATA : True          # View ja/nein ?
WRITEABLE : True        # Schreibrechte
ALIGNED : True          # Memory alignment
WRITEBACKIFCOPY : False # ...
UPDATEIFCOPY : False    # ...
```

**Takeaway 10.** *Bei der Deklaration das Memorylayout möglichst angeben und Speicher immer in der angelegten Reihenfolge durchgehen (Cache-friendly Code).*

# Übersicht

Motivation

Wie Cython verwenden?

Statisches Typsystem

Codeprofiling

Typed Memoryviews

Extratuning

Casestudy



# Compilerdirectives

	Directive	default-Wert
Prüfen der Indices?	<code>boundscheck</code>	<code>True</code>
Negative Indices erlauben?	<code>wraparound</code>	<code>True</code>
C-Division verwenden?	<code>cdivision</code>	<code>False</code>
Prüft bei Zugriff auf Memoryviews, ob diese initialisiert sind	<code>initializedcheck</code>	<code>True</code>
Gibt einen Fehler zurück bei Integer-Overflow	<code>overflowcheck</code>	<code>False</code>

**Tabelle:** Nützliche Directives für den Cythoncompiler. Mehr unter: <https://bit.ly/3luL7EK>

Innerhalb eines Notebooks können die Directives für eine komplette Zelle via

```
# cython: directive1=value, directive2=value, directive3=value
```

gesetzt werden.

# Compilerflags

	gcc & clang*	MVSC
Aktiviert Codeoptimierungen	-O3	/O2
Verwenden aller verfügbaren CPU-Features der aktuellen Maschine	-march=native	/arch:AVX2 /arch:AVX512
Beschleunigung von Gleitkommaoperationen	-ffast-math	/fp:fast
Feedback bzgl. SIMD-Autovektorisierung	-fopt-info-vec-optimized -Rpass=loop-vectorize*	/Qvec-report:1
Für Parallelisierung via OpenMP	-fopenmp	/openmp

**Tabelle:** Nützliche Compilerflags für die drei gängigsten C-Compiler.

Die Flags werden innerhalb einer Notebookzelle via `%%cython -c=flag1 -c=flag2` gesetzt und dann automatisch an den C-Compiler übergeben.

# Parallelisierung

Cython unterstützt threadbasierte Parallelisierung via OpenMP.

- Cython bietet die `prange` Funktion für parallele Schleifen an, welche automatisch via OpenMP Threads startet.
- Die Anzahl der Threads ist standardmäßig die Anzahl der verfügbaren CPU-Cores.
- Thread-Lokalität und `reductions` für Variablen werden automatisch erkannt.
- Der Code muss mit OpenMP-Compilerflag kompiliert und gelinkt werden.

```
# -- Cython --  
from cython.parallel cimport prange
```

```
# Möglichkeit 1  
cdef func1(int N):  
    cdef int a  
    with nogil:  
        for a in prange(1, N):  
            # do something (parallel)
```

```
# -- Cython --  
from cython.parallel cimport prange
```

```
# Möglichkeit 2  
cdef func2(int N):  
    cdef int a  
    for a in prange(1, N, nogil=True):  
        # do something (parallel)
```

**Takeaway 11.** Für parallelen Cythoncode muss der GIL deaktiviert werden.

# Parallelisierung

Wir können via `schedule` und `chunksize` steuern, wie die Arbeit unter den Threads aufgeteilt wird:

schedule	default chunksize	Overhead	Anwendungskriterium
static	$\frac{N}{P}$	gering	Das Problem lässt sich in ungefähr gleich große Chunks mit gleicher Laufzeit aufteilen.
dynamic	$\frac{N}{P}$	mittel	Falls Laufzeit und Größe der einzelnen Chunks unterschiedlich und im Voraus nicht bekannt ist

**Tabelle:**  $N$  Anzahl der Gesamtiterationen,  $P$  Anzahl der Threads.

Eine anschauliche Erklärung unter: <http://ppc.cs.aalto.fi/ch3/schedule/>.

```
# -- Cython --  
for i in prange(N, nogil=True, schedule='static', chunksize=1):  
    # ...
```

**Takeaway 12.** Verwende für `prange` geeignetes scheduling und chunksize, um die Arbeit optimal unter den Threads aufzuteilen.

# Parallelisierung

## Scheduling

Seien  $a$ ,  $z$  und  $w$  rechenintensive Funktionen, wobei die Rechenzeit für  $w$  mit dem Argument steigt. Wir gehen von einer CPU mit 4 Kernen aus. Wie parallelisieren wir am besten die folgende Funktion?

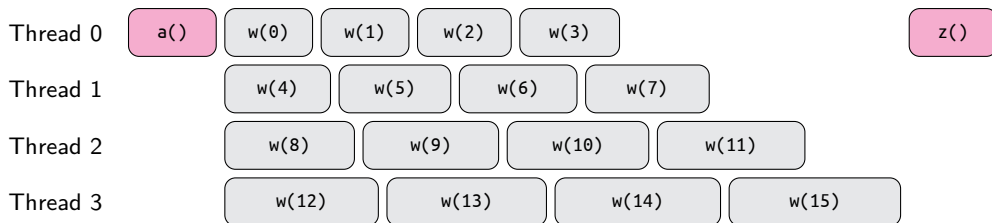
```
def myfunc():  
    a()  
    for i in range(16):  
        w(i)  
    z()
```

# Parallelisierung

## Scheduling

*# -- Cython -- (Möglichkeit 1)*

```
def myfunc():  
    a()  
    for i in prange(16, nogil=True):  
        w(i)  
    z()
```



**Abbildung:** default-Scheduling entspricht `schedule='static'` mit `chunksize=4`.

Quelle: <http://ppc.cs.aalto.fi/ch3/schedule/>

# Parallelisierung

## Scheduling

# -- Cython -- (Möglichkeit 2)

```
def myfunc():  
    a()  
    for i in prange(16, nogil=True, schedule='static', chunksize=1):  
        w(i)  
    z()
```

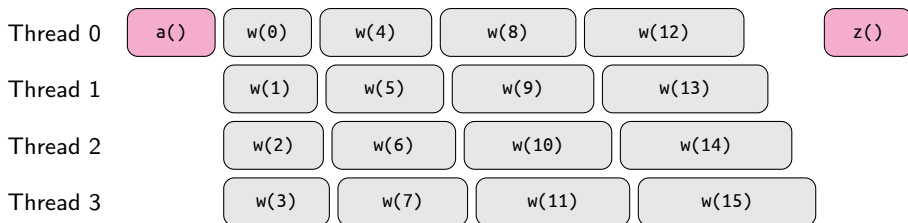


Abbildung: `schedule='static'` mit `chunksize=1`.

Quelle: <http://ppc.cs.aalto.fi/ch3/schedule/>

# Übersicht

Motivation

Wie Cython verwenden?

Statisches Typsystem

Codeprofiling

Typed Memoryviews

Extratuning

Casestudy



# Casestudy: Solving world's hardest sudoku

## The Telegraph

HOME » NEWS » SCIENCE » SCIENCE NEWS

### World's hardest sudoku: can you crack it?

Readers who spend hours grappling in vain with the Telegraph's daily sudoku puzzles should look away now.



The Everest of numerical games was devised by Arto Inkala, a Finnish mathematician, and is specifically designed to be unsolvable to all but the sharpest minds.

```
# -- Python --
```

```
def solve(puzzle):
    row, col = find_empty_cell(puzzle)
    if row == -1 and col == -1:
        return True

    for num in range(1,10):
        if valid(puzzle, num, (row, col)):
            puzzle[row, col] = num
            if solve(puzzle):
                return True
            else:
                puzzle[row, col] = 0
    return False
```

Quelle: <https://bit.ly/32rBeAo>

**Aufgabe:** Beschleunige den gegebenen Pythoncode mit Hilfe von Cython.

# Zusammenfassung

Cython lohnt sich unter anderem

- zur Beschleunigung von verschachtelten Pythonschleifen,
- zur effizienten Implementierung neuer Algorithmen,
- in Fällen, in denen oft Funktionen aufgerufen werden müssen und deren Auswertung relativ billig ist (Numerische Integration, Lösen von ODEs, Nichtlineare Optimierung, ...)
- in Fällen, in denen die verfügbaren numpy-Methoden nicht effizient genug sind.
- zur Einbindung von vorhandenem C/C++ Code in Python.

Cython bietet noch einiges mehr. Unter anderem

- Mit der C++ STL und ihren Algorithmen ...
- C und C++ Strings
- Cythonklassen als alternative zu Pythonklassen (Extension Types).