Einführung in Cython

Teil 2

Jonathan Helgert
jhelgert@mail.uni-mannheim.de

https://github.com/jhelgert/cython_einfuehrung

19. November 2020

Contents

Contents

Pointer in a nutshell

Eine Zeigervariable (engl. *Pointer*) speichert die Adresse eines Datenobjekts und gibt an, wie die unter der Adresse abgelegten Bytes zu interpretieren sind.

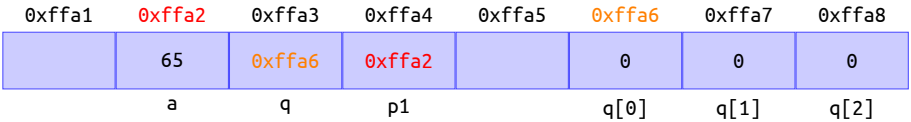


Abbildung: Vereinfachte Darstellung des Speichers. Ein Block entspricht einem Byte.

```c
// -- C --
char   a;      // a ist eine normale (uninitialisierte) char Variable.
char* p1 = &a  // p1 ist ein Pointer, der auf a zeigt. (& gibt uns die Adresse einer Variablen)
p1[0]    = 65  // Ändert den Wert von a auf 90 via Pointer p1

char* q  = calloc(3, sizeof(char)); // Speicher für 3 chars anlegen u. mit 0en füllen.
//  q[0] = 10  // Gib dem 1. Element des Speicherbereichs den Wert 10
//  q[1] = 20  // Gib dem 2. Element des Speicherbereichs den Wert 20
//  q[2] = 30  // Gib dem 3. Element des Speicherbereichs den Wert 30
```

Unterschiede zwischen C und Cython

Cython bietet die bekannten structs, unions und enums aus C:

```c
// -- C --
typedef struct{
    int x;
    double y;
} tup2d;

typedef union{
    double d4[4];
    __m256d d;
} simdd;

enum States{
    BadState = -1,
    GooDState = 0,
    PerfectState = 1
};
```

```cython
# -- Cython --
ctypedef struct tup2d:
    int x
    double y

ctypedef union simdd:
    double d4[4];
    __m256d d;

cdef enum States:
    BadState = -1
    GoodState = 0
    PerfectState = 1
```

Unterschiede zwischen C und Cython

Es gibt in Cython keinen...

- unären Dereferenzierungsoperator. Falls `p` ein Pointer ist, schreiben wir in Cython `p[0]` statt `*p`.
- Pfeiloperator. Falls `p` ein Pointer auf eine Struktur ist, schreiben wir `p[0].x` statt `p->x`.
- Typcasts erfolgen in Cython via `<>`, also `<new_typ>` statt `(new_typ)`.
- In Cython ist ein Typcast bei calloc/malloc notwendig, in C nicht.

```c
// -- C --
#include <stdlib.h>

typedef struct{
    int x;
    double y;
} mytuple;

mytuple t = {.x = 2, .y = 3.0};
mytuple* ptr = &t;
double q = (double) ptr->y;

char* z = calloc(3, sizeof(char));
```

```cython
# -- Cython --
from libc.stdlib cimport calloc

ctypedef struct mytuple:
    int x
    double y

cdef mytuple t = mytuple(x=1, y=3.0)
cdef mytuple* ptr = &t
cdef double q = <double> ptr[0].y

cdef char* z  = <char*> calloc(3, sizeof(char))
```

C-Libraries

Cython hat die Header einiger C Standardlibraries bereits innerhalb von `libc` implementiert:

```
# -- Cython --
cimport libc.stdio   # <stdio.h>
cimport libc.stdlib  # <stdlib.h>
cimport libc.stddef  # <stddef.h>
cimport libc.stdint  # <stdint.h>
cimport libc.math    # <math.h>
cimport libc.complex # <complex.h>
cimport libc.string  # <string.h>
cimport libc.time    # <time.h>
cimport libc.float   # <float.h>
cimport libc.limits  # <limits.h>
```

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

# Beispiel:
def print_my_c_cos(double[::1] x):
    cdef int i, N = x.size
    for i in range(N):
        printf("%lf\n", c_cos(x[i]))
```

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

C++ STL

Werbeblock: Auch der Großteil der C++ STL wird unterstützt.

```
cimport libcpp.algorithm     # <algorithm>
cimport libcpp.atomic        # <atomic>
cimport libcpp.cast          # <cast>
cimport libcpp.complex       # <complex>
cimport libcpp.deque         # <deque>
cimport libcpp.forward_list  # <forward_list>
cimport libcpp.functional    # <functional>
cimport libcpp.iterator      # <iterator>
cimport libcpp.limits        # <limits>
cimport libcpp.list          # <list>
cimport libcpp.map           # <map>
cimport libcpp.memory        # <memory>
cimport libcpp.numeric       # <numeric>
cimport libcpp.pair          # <pair>
cimport libcpp.queue         # <queue>
cimport libcpp.set           # <set>
cimport libcpp.stack         # <stack>
cimport libcpp.string        # <string>
cimport libcpp.typeindex     # <typeindex>
cimport libcpp.typeinfo      # <typeinfo>
cimport libcpp.unordered_map # <unordered_map>
cimport libcpp.unordered_set # <unordered_set>
cimport libcpp.utility       # <utility>
cimport libcpp.vector        # <vector>
```

```
# -- Cython --
#distutils: language = c++
from libcpp.vector cimport vector
cdef vector[int] v = range(1, 10, 2) # Spezifizieren des Templates via [typ]
print(v) # [1, 3, 5, 7, 9]
```

Contents

Wrapping C-Code

- Einer der Hauptanwendungszwecke von Cython ist das Einbinden von C Libraries. Wie geht das?
- Alle Funktionen, structs und unions, die wir innerhalb von Cython nutzen möchten, müssen mit einem `cdef extern from` Block deklariert werden.
- Funktionen, die innerhalb der C Library aufgerufen werden, aber nicht innerhalb von Python selbst benötigt werden, müssen nicht deklariert werden!
- Nach der Deklaration sind die Funktion wie normale `cdef` Funktionen aus Teil1 innerhalb der Datei verwendbar.

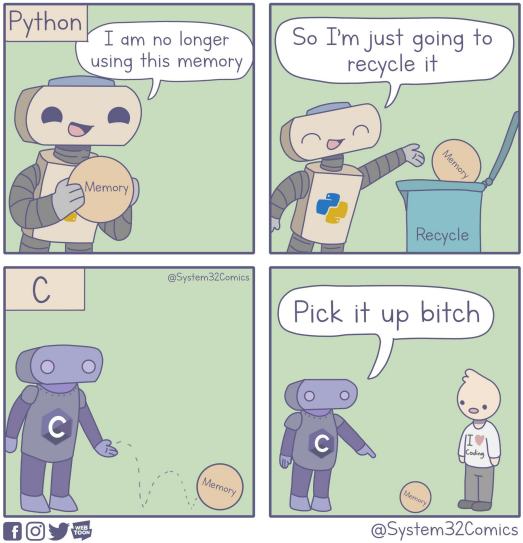
Beispiel:

```c
// -- C -- (our_c_file.h)
typedef struct{
    double* data;
    int size;
} c_vec;

int c_sum(int a, int b);
double c_dot_product(double* a, double* b);
double c_dot_product2(c_vec* a, c_vec* b);
void some_helper_fct_for_c_dot(c_vec* a);
```

```cython
# -- Cython -- (wrapper.pyx)
cdef extern from "our_c_file.h" nogil:
    # Deklarieren der struct
    ctypedef struct c_vec:
        double* data
        int size
    # Deklarieren der Funktionen
    int c_sum(int a, int b)
    double c_dot_product(double* a, double* b)
    double c_dot_product2(c_vec* a, c_vec* b)
```

Contents

Speichermanagement in C und Python



Takeaway 1. *Falls möglich, den C-Speicher für Arrays mit Hilfe von* `np.arrays` *in Python allokieren und via Memoryview an die C-Funktionen übergeben.*

Typed Memoryviews und Pointer

Wir wissen aus Teil 1, dass Memoryviews auch C-Arrays und Pointer unterstützen.

- Ein Pointer auf die C-Standardtypen kann durch einen *Typcast* in ein Memoryview umgewandelt werden.
- Beim *Casten* muss allerdings der `shape` angegeben werden.
- **Achtung:** Der Speicher muss von Hand freigegeben 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[::1] mv1 = <double[:N]> ptr1  # Typcast des Pointers in ein Memoryview
cdef double* ptr2 = &mv1[0]

cdef int[:, ::1] mv1 = <int[:M, :N]> calloc(M*N, sizeof(int)) # Typcast des Pointers in ein Memoryview
cdef int* ptr4 = &mv2[0, 0]

# Aufräumen des Speichers
free(&mv1[0])    # Identisch zu free(ptr1)
free(&mv2[0, 0]) # Identisch zu free(ptr4)
```

Pythonspeicher an C-Funktion

Für C-Funktionen, welche lediglich Pointer als Argumente erwarten (und nicht zurückgeben), können wir den Speicher bequem via Python verwalten, insofern dieser kompatibel zu einem Memoryview ist:

```
# -- Cython --
cimport numpy as np
import numpy as np

cdef extern from "our_c_file.h" nogil:
    # Diese Funktionen benötigen lediglich einen Speicherbereich
    cdef void my_c_fkt_vec(double* b, int n)
    cdef void my_c_fkt_mat(double* A, int m, int n)

def wrapper1(int N):
    cdef double[::1] b = np.zeros(N)
    my_c_fkt_vec(&b[0], N)                # Übergabe an C-Funktion
    return np.asarray(b)                  # Rückgabe

def wrapper2(int M, int N):
    cdef double[:, ::1] A = np.zeros((M, N))
    my_c_fkt_mat(&A[0, 0], M, N)          # Übergabe an C-Funktion
    return np.asarray(A)                  # Rückgabe
```

C-Speicher in Cython verwalten

Angenommen wir haben eine C-Funktion, die Speicher anlegt und einen Pointer zurückgibt. Können wir diesen in ein `np.array` packen?

```
# -- Cython --
cimport numpy as np
import numpy as np
from libc.stdlib cimport calloc, free

cdef extern from "our_c_file.h" nogil:
    # gibt einen C-Speicherbereich (Pointer) zurück,
    # den wir innerhalb von Python verwalten wollen.
    cdef double* my_c_fkt()

cdef matrix_from_c_mem(int M, int N):
    cdef double[:, ::1] mv = <double[:M, :N]> my_c_fkt()
    return np.asarray(mv)
```

- Das `np.array` "besitzt" den Speicher nicht und wird ihn daher auch nicht automatisch aufräumen.
  $\Longrightarrow$ Memoryleak.
- Einfachste Lösung: Umweg über ein `cython.view.array`.

C-Speicher in Cython verwalten: `cython.view.array`

Speicher, der von einer C-Library angelegt wird und kompatibel zu einem Memoryview ist, können wir bequem mit Hilfe eines `cython.view.arrays` verwalten lassen.

```
# -- Cython --
import numpy as np
cimport numpy as np
from cython.view cimport array as cyarray

cdef extern from "our_c_file.h" nogil:
    # gibt einen C-Speicherbereich (Pointer) zurück,
    # den wir innerhalb von Python verwalten wollen.
    cdef double* my_c_fkt()

cdef matrix_from_c_mem(int M, int N):
    cdef cyarray v = <double[:M, :N]> my_c_fkt()
    v.free_data = True                    # Wichtig !
    return np.asarray(v)
```

Takeaway 2. *Verwende `cython.view.arrays` für die Verwaltung von C-Speicher, der kompatibel zu einem Memoryview ist.*

RAII in Cython

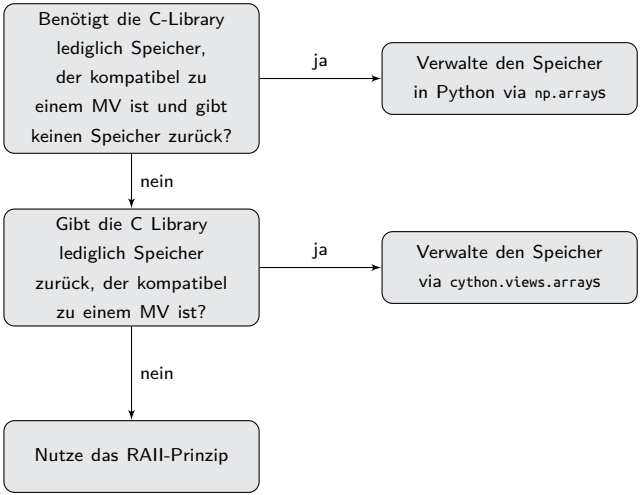
Wie können wir C-Speicher, der nicht kompatibel zu einem Memoryview ist (z.B. eigene structs), elegant von Python verwalten lassen?

```c
// (our_simple_c_lib.h)
typedef struct{
    // ... struct members ...
} stype;

// Funktion, welche Speicher für stype anlegt
// und einen stype-Pointer zurückgibt
stype* clib_init();

// Funktion, welche iwas berechnet
double clib_dosth(stype* T)

// Funktion, welche den von der Struktur
// angelegten Speicher wieder freigibt
void clib_free(stype* T)
```

```python
# (our_wrapper.pyx)
cdef extern from "our_simple_c_lib.h" nogil:
    # hier Funktionen und structs deklarieren

cdef class OurWrapperClass:
    cdef stype* __ptr

    def __cinit__(self):
        self.__ptr = clib_init()

    def __dealloc__(self):
        clib_free(self.__ptr)

    def doSomething(self):
        return clib_dosth(self.__ptr)
```

Takeaway 3. *Nutze das RAII-Prinzip für automatische Speicherverwaltung innerhalb Pythons.*

Speichermanagement

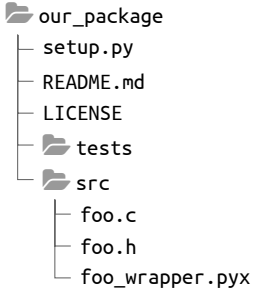


Contents

Struktur eines Cythonpackages

Wir gehen davon aus, dass der Enduser bereits Cython und einen C Compiler installiert hat:

- Verwenden das `setuptools` package innerhalb der `setup.py`.
- Die `.pyx` Datei enthält unseren Cythoncode, in welchem wir den C code einbinden.
- Installation anschließend via `python3 setup.py` innerhalb von `our_package`.
- **Achtung**: Compile- und Linkflags müssen je nach Compiler angepasst werden.
- Bessere Lösung: Verwende *Continuous Integration* z.B. via GitHub Actions, CI Travis oder Appveyor.



Struktur eines Cythonpackages

```python
# -- Python -- (setup.py)
from setuptools import setup, Extension
from Cython.Build import cythonize
from numpy import get_include # nur notwendig, falls wir 'cimport numpy' verwenden
import sys

if 'linux' in sys.platform or 'darwin' in sys.platform:
    compile_flags = ["-std=c11", "-O3", "-march=native"]
else:
    compile_flags = ["/std:c11", "/O2", "/arch:AVX2"]

ext = Extension(name = "FooWrapper", # Name unseres python packages
            sources = ["src/foo.c", "src/foo_wrapper.pyx"],
            include_dirs = [get_include()], # Nur falls wir 'cimport numpy' verwenden
            extra_compile_args = compile_flags)

setup(ext_modules=cythonize(ext))
```

Takeaway 4. *Stelle sicher, dass die `setup.py` deines Cythonpackages mit verschiedenen Compilern funktioniert.*

Contents

Casestudy: Thomas-Algorithmus

Für ein Gleichungssystem $Tx = d$ mit einer quadratischen Tridiagonalmatrix T, d.h.

$$\begin{pmatrix} t_{1,1} & t_{1,2} & 0 & \dots & 0 \\ t_{2,1} & t_{2,2} & t_{2,3} & \ddots & \vdots \\ 0 & t_{3,2} & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & t_{n-1,n} \\ 0 & \dots & 0 & t_{n,n-1} & t_{n,n} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ \vdots \\ \vdots \\ d_n \end{pmatrix}$$

bietet sich der Thomas-Algorithmus an. Nach etwas googeln finden wir dazu schon geschriebenen C-Code, der lediglich aus einer Funktion besteht:

```
void thomas(double* sol, double* a, double* b, double* c, double* rhs, int n)
```

wobei

- b die Diagonalelemente von T sind.
- a die untere und b die obere Nebendiagonale von T ist.

Aufgabe: Binde diese C-Funktion mit Hilfe von Cython in eine Pythonfunktion `cythomas(T, d)` ein. Die Funktion `np.ascontiguousarray` und die `.diagonal(offset=k)` Methode eines `np.array` sind hier nützlich.

Casestudy: Ein noch schnellerer Sudoku Solver

Wie angekündigt, kann man den Sudokusolver aus Teil 1 noch weiter beschleunigen.

- Dazu codieren wir das Sudoku mit vier Hilfsarrays (jeweils mit 9 Einträgen) wie folgt:

$$\texttt{binrows[k]} = (0 \dots 0 \overset{i}{1} 0 \dots 0)_2, \quad \text{falls die Zahl } i \text{ in Zeile } k \text{ gesetzt ist.}$$
$$\texttt{bincols[k]} = (0 \dots 0 1 0 \dots 0)_2, \quad \text{falls die Zahl } i \text{ in Spalte } k \text{ gesetzt ist.}$$
$$\texttt{binboxs[k]} = (0 \dots 0 1 0 \dots 0)_2, \quad \text{falls die Zahl } i \text{ in Box } k \text{ gesetzt ist.}$$
$$\texttt{binindx[k]} = (0 \dots 0 1 0 \dots 0)_2, \quad \text{falls in Zeile } k \text{ und Spalte } i \text{ eine Zahl gesetzt ist.}$$

- Die Funktionen `find_empty_cell()` und `valid()` sind durch Bitoperationen realisiert worden. $\Longrightarrow$ schneller!

Aufgabe: Binde die C-Library mit geringstmöglichem Aufwand und korrekter Speicherwaltung mit Cython ein. Von Python aus soll lediglich eine Funktion `solve(S)` bereitgestellt werden, wobei das Sudoku S als `np.array` übergeben wird.