Cython tutorial

Release 2011

Pauli Virtanen

September 13, 2011

CONTENTS

authors Pauli Virtanen

... some ideas shamelessly stolen from last year's tutorial by Stefan van der Walt...

CHAPTER

ONE

THE QUEST FOR SPEED: CYTHON

Pauli Virtanen

Institute of Theoretical Physics and Astrophysics, University of Würzburg

St. Andrews, 13 Sep 2011

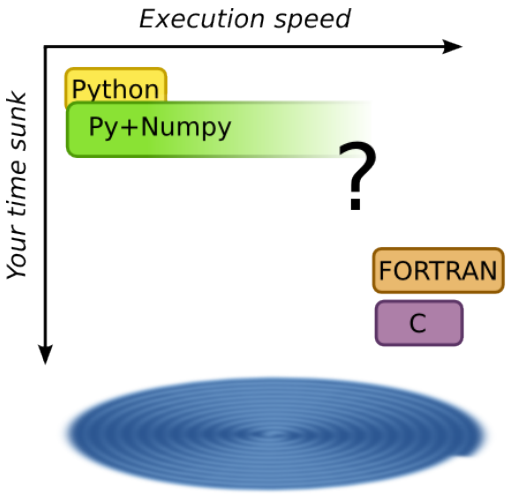
CHAPTER

TWO

KNOW THE GROUNDS

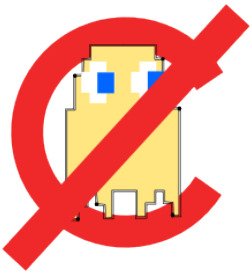
2.1 A question

- Too slow. What to do?



2.2 Who do you call?

- Back to writing C (and dealing with Python C API)?



2.3 Cython

- Language: **superset** of Python (mostly)
- Compiles to (CPython-specific) C code
- Has features to **overcome** several Python overheads
- Makes **interfacing** with existing C code easy.

  ... and avoids the pain of the Python **C API**!
- Ancestry: Cython is based on Pyrex (2002)

[1] http://cython.org/

2.4 Cython is used by...

- *Numpy* (a little bit) for **performance**
- *Scipy* (slightly more) for **performance** & **wrapping C**
- *Sage*, symbolic math software, for **performance** & **wrapping C**
- *mpi4py*, for **wrapping C**
- *petsc4py*, for **wrapping C**
- *lxml*, XML processing, for **wrapping C**
- & others...

CHAPTER

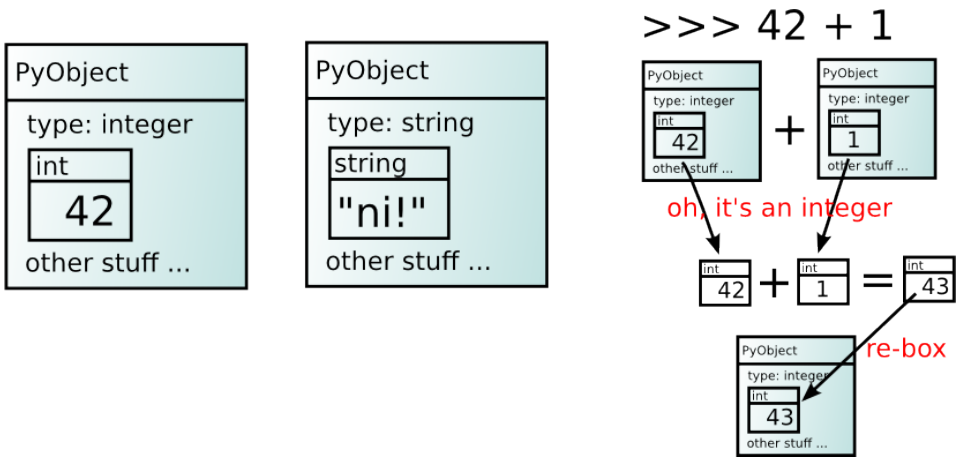
THREE

WHAT'S THERE TO OPTIMIZE?

3.1 Starting point

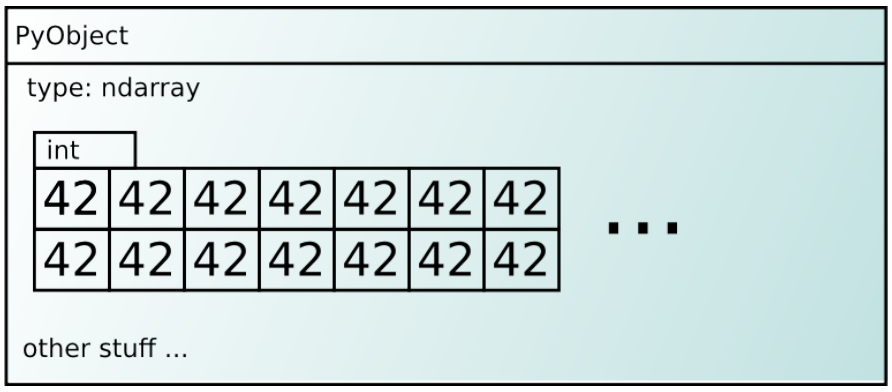
- Python: an *interpreted*, *dynamic* language
- Overheads:
  - Interpreting itself
  - Stuff is in boxes
  - Function calls cost more
  - Global interpreter lock

3.2 Boxing



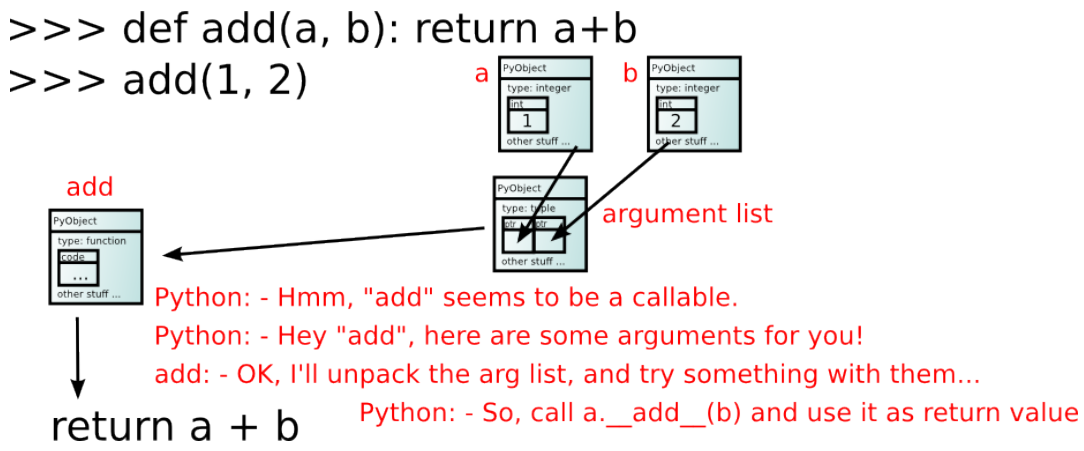
Everything is an object, **everything** is in a box: boxing–unboxing overhead

3.3 Numpy performance



Numpy has large boxes: negligible overhead for large arrays

3.4 Function calls



Function calls involve (some) boxing and checking: some overhead.

3.5 Global Interpreter Lock

- Python can have multiple threads
- It can interpret **Python code** in a **single thread** at a time.

  However...

  - I/O works fine in parallel
  - Non-Python code OK (⇐ insert Cython here)
  - Much of Numpy is non-Python code

CHAPTER

FOUR

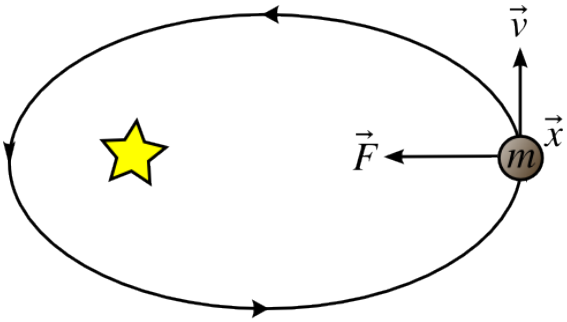
REGAINING SPEED WITH CYTHON

4.1 The Plan

- Take a piece of pure-Python code
- Overcome overheads (where needed) with Cython:

Interpretation:	**compiled to C**
Stuff in boxes:	**explicit types**
Function calls:	**even more explicit types**
GIL:	**releasing GIL**

4.2 Example problem: Planet in orbit



$$\frac{d\vec{x}}{dt} = \vec{v}$$
$$\frac{d\vec{v}}{dt} = \frac{\vec{F}(\vec{x})}{m}$$

```
for j in range(n_steps):
    Fx = ...
    ...
    x = x + dt*vx
    y = y + dt*vy
    ...
    vx = vx + dt*Fx/m
    ...
```

- Solving an ordinary differential equation
- **No way to vectorize!** (Numpy does not help here)

Cython to the rescue?

4.3 Measure first

- **Measure** before you cut
  - Is/Would pure-Python be too slow?

  - Is ~ 10-100x speedup enough? (Note: usual max, discounting Numpy...)
  - Minimize work by locating hotspots (profile, guess)

    Scientific code: usually few hot spots

- Demo (using `line_profiler`):

```
Add @profile to functions (& comment out plot commands)
$ kernprof.py -l run_gravity.py
$ python -m line_profiler run_gravity.py.lprof
```

4.4 My first Cython program

gravity.py

```
from math import sqrt

class Planet(object):
    def __init__(self):
        # some initial position and velocity
        self.x = 1.0
        self.y = 0.0
        self.z = 0.0
        self.vx = 0.0
        self.vy = 0.5
        self.vz = 0.0

        # some mass
        self.m = 1.0

def single_step(planet, dt):
    """Make a single time step"""

    # Compute force: gravity towards origin
    distance = sqrt(planet.x**2 + planet.y**2 + planet.z**2)
    Fx = -planet.x / distance**3
    Fy = -planet.y / distance**3
    Fz = -planet.z / distance**3

    # Time step position, according to velocity
    planet.x += dt * planet.vx
    planet.y += dt * planet.vy
    planet.z += dt * planet.vz

    # Time step velocity, according to force and mass
    planet.vx += dt * Fx / planet.m
    planet.vy += dt * Fy / planet.m
    planet.vz += dt * Fz / planet.m

def step_time(planet, time_span, n_steps):
    """Make a number of time steps forward """

    dt = time_span / n_steps

    for j in range(n_steps):
        single_step(planet, dt)
```

gravity_cy.pyx

```
from math import sqrt

class Planet(object):
    def __init__(self):
        # some initial position and velocity
        self.x = 1.0
        self.y = 0.0
        self.z = 0.0
        self.vx = 0.0
        self.vy = 0.5
        self.vz = 0.0

        # some mass
        self.m = 1.0

def single_step(planet, dt):
    """Make a single time step"""

    # Compute force: gravity towards origin
    distance = sqrt(planet.x**2 + planet.y**2 + planet.z**2)
    Fx = -planet.x / distance**3
    Fy = -planet.y / distance**3
    Fz = -planet.z / distance**3

    # Time step position, according to velocity
    planet.x += dt * planet.vx
    planet.y += dt * planet.vy
    planet.z += dt * planet.vz

    # Time step velocity, according to force and mass
    planet.vx += dt * Fx / planet.m
    planet.vy += dt * Fy / planet.m
    planet.vz += dt * Fz / planet.m

def step_time(planet, time_span, n_steps):
    """Make a number of time steps forward """

    dt = time_span / n_steps

    for j in range(n_steps):
        single_step(planet, dt)
```

- Cython's aim – a superset of Python
- Gets rid of interpreter overhead!

  This is usually only a small gain. (Boxing, etc. are still there...)

4.5 Compiling the Cython program

setup.py

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [
        Extension("gravity_cy",
                  ["gravity_cy.pyx"],
        ),
])
```

```
$ python setup.py build
```

or

```
$ python setup.py build_ext -i   # <- so it's importable from same dir
```

4.6 Type declarations

- Syntax

```
def some_function(some_type some_parameter):

    cdef another_type variable

    ...
```

- Restrict types of variables and arguments
- Gets rid of boxes! (Drawback: no duck-typing)

gravity.py

```
def step_time(planet,
              time_span,
              n_steps):
    dt = time_span / n_steps

    for j in range(n_steps):
        single_step(planet, dt)
```

gravity_cy.pyx

```
def step_time(planet,
              double time_span,
              int n_steps):
    cdef double dt
```

```
    cdef int j

    dt = time_span / n_steps

    for j in range(n_steps):
        single_step(planet, dt)
```

4.7 Cython annotated output

- Where is the overhead? Did I miss something?
- Demo:

```
cython -a gravity_cy.pyx
```

4.8 Type declarations for classes

- Restrict types of member variables (and methods)
- Again: less boxes! (In Cython code...)

gravity.py

```
class Planet(object):
    def __init__(self):
        # some initial position and velocity
        self.x = 1.0
        self.y = 0.0
        self.z = 0.0
        self.vx = 0.0
        self.vy = 0.5
        self.vz = 0.0

        # some mass
        self.m = 1.0
```

gravity_cy.pyx

```
cdef class Planet(object):
    cdef public double x, y, z, vx, vy, vz, m

    def __init__(self):
        # some initial position and velocity
        self.x = 1.0
        self.y = 0.0
        self.z = 0.0
        self.vx = 0.0
        self.vy = 0.5
        self.vz = 0.0
```

```
        # some mass
        self.m = 1.0
```

- `public`: make the member variable accessible from Python

4.9 Function declarations

- Remove Python-specific function call overhead (when calling from Cython)
- Syntax

```
def some_function(int a, int b):
    """Callable from Cython & Python"""
    ...

cdef some_function(int a, int b):
    """Callable from Cython only (but optimized)"""
    ...

cpdef some_function(int a, int b):
    """Callable from Cython (optimized) & Python (not optimized)"""
    ...
```

gravity.py

```
def single_step(planet, dt):
    """Make a single time step"""
    ...
```

gravity_cy.pyx

```
cdef void single_step(Planet planet,
                      double dt):
    ...
```

4.10 Interfacing with C

- We can use the C standard library to do some of the calculations

gravity.py

```
from math import sqrt
```

gravity_cy.pyx

```
cdef extern from "math.h":
    double sqrt(double x)
```

4.11 Giving up some of Python

- Divide by zero raises an exception ⇒ Needs more checks (slower) & the GIL
- Instruct Cython to use the C semantics for division:

gravity.py

```
def single_step(planet, dt):
    """Make a single time step"""
    cdef float x, y
    ...
    x = 0
    y = 1/x
```

gravity_cy.pyx

```
cimport cython

@cython.cdivision(True)
cdef void single_step(Planet planet,
                      double dt):
    cdef float x, y
    ...
    x = 0
    y = 1/x     # No exception!
```

4.12 Releasing the GIL

- Overcome issues with multithreading: release the GIL in Cython
- Without GIL, you cannot do **anything** that messes with Python boxes (!!)

  Raising exceptions, calling Python functions, touching non-cdef class attributes, ...

gravity_cy.pyx

```
cimport cython

cdef extern from "math.h":
    double sqrt(double x) nogil

...

@cython.cdivision(True)
cdef void single_step(Planet planet,
                      double dt) nogil:
    ...

def step_time(Planet planet, double time_span, int n_steps):
    """
    Make a number of time steps forward
```

```
    """
    cdef double dt
    cdef int j

    dt = time_span / n_steps

    with nogil:
        for j in range(n_steps):
            single_step(planet, dt)
```

- Demo: does it help?

4.13 Summary

So yes, we have

- Eliminated boxing via `cdef some_type` & `cdef class` & et al.
- Eliminated function call overhead via `cdef some_function`
- Called C functions with `cdef extern from`
- Eliminated GIL with `nogil`, to run multiple threads
- Demo.

4.14 Oh snap!

- ... all this and we gained a lousy factor of `10x` ??

```
__pyx_v_distance = sqrt(((pow(__pyx_v_planet->x, 2.0)
                        + pow(__pyx_v_planet->y, 2.0))
                        + pow(__pyx_v_planet->z, 2.0)));
```

- That pesky `pow` (it's **slow**)!
- Tell the C compiler to work harder (and that we don't care anymore):

```
$ OPT="-O3 -ffast-math" python setup.py build_ext -i
```

- `80x`, looks better

CHAPTER

FIVE

NUMPY ARRAYS IN CYTHON

5.1 Basics

- Cython supports Numpy arrays!
- Usage:

```
import numpy as np
cimport numpy as np

...

cdef np.ndarray[np.double_t, ndim=2] some_array

some_array = np.zeros((50, 50), dtype=np.double)
```

5.2 Data types

- Note: data type needs to be declared:

```
cdef np.ndarray[np.double_t, ndim=2] some_array
```

- Mapping:

Numpy	Cython
np.int8	np.int8_t
np.int16	np.int16_t
...	...
np.single	np.float_t (same as float32)
np.double	np.double_t (same as float64)
np.complex	np.complex_t
...	...

5.3 What is faster

- **Only indexing** (at the moment):

```
some_array[i, j]
```

- Number of dimensions and data type needed in advance:

```
cdef np.ndarray[np.double_t, ndim=2] some_array
```

- Broadcasting, slicing, fancy indexing: the usual Numpy speed

  Check your `cython -a`

- Demo.

5.4 Accessing the raw data

- The library you want to use only deals with `double *`?
- The raw data buffer can be obtained:

```
np.ndarray[np.double_t, ndim=1] some_array
cdef double *data

some_array = np.zeros((20,), dtype=np.double)
data = <double*>some_array.data

data[0]  # the first element -- or maybe not (depends on strides!)
```

- Remember to do:

```
if not some_array.flags.c_contiguous:
    raise ValueError("Only C-contiguous arrays are supported!")
```

5.5 Turning off more Python

- Sure your code works? Turn off bounds checking!

```
cimport cython
@cython.boundscheck(False)
def some_function(...):
    ...
```

- A real reason for doing this: using `nogil`

  (Without GIL, out-of-bounds exceptions cannot be raised.)

CHAPTER

SIX

USEFUL STUFF TO KNOW

6.1 Profiling Cython code

Requires a recent Cython version (here we have 0.15)

- Add this to the top:

```
# cython: profile=True
```

- Extract (function-level only) profile:

```
import pstats, cProfile
import run_gravity

cProfile.runctx("run_gravity.main()", globals(), locals(), "profile.prof")
s = pstats.Stats("Profile.prof")
s.strip_dirs().sort_stats("time").print_stats()
```

- Or:

```
kernprof.py run_gravity
python -m pstats run_gravity.py.prof
% strip
% sort time
% stats 20
```

- Or:

```
In [10]: %prun some_code.py
```

- Demo.

6.2 Exceptions in cdef functions

```
cdef int foo():
    raise ValueError("error!")
```

If you know Python C API, you see a problem:

- Python C API functions return a NULL instead of a PyObject, on exception
- But `foo` returns an `int` – how can it indicate an exception?

```
cdef int foo() except -1:
    if something:
        return 0  # but *never* -1
    raise ValueError("error!")

cdef int foo() except? -1:
    if something:
        return -1  # or anything
    raise ValueError("error!")

cdef int foo() except *:
    raise ValueError("error!")
```

6.3 More on compilation

Include files et cetera

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext
import numpy

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [
        Extension("gravity_cy",
                  ["gravity_cy.pyx"],
                  include_dirs=[numpy.get_includes()],
                  libraries=["m"],
        ),
])
```

Magic!

```
import pyximport
pyximport.install()

import gravity_cy
```

6.4 Python-compatible syntax

```
import cython

@cython.locals(x=cython.double, y=cython.double)
def func(x):
    y = x + 5
    return y
```

6.5 And more

- See http://docs.cython.org/
- Exercises

CHAPTER

SEVEN

EXERCISES

7.1 Exercise 1: Cythonization

Study the provided `fractal.py` for computing the Newton fractal. Determine (by timing, profiling, or just guessing) which part is the bottleneck and decide which part is worthwhile to convert to Cython. Do the conversion and add necessary type declarations etc.

How large a speedup do you get?

Note: Protips: Cython defines Numpy's complex type as `np.complex_t`. It is not directly compatible with Cython's C-level complex type `cdef double complex`, so to assign one to the other, you need to do `a.real = b.real; a.imag = b.imag`.

Remember also `@cython.cdivision` and others.

Before profiling, comment out plotting commands.

7.2 Exercise 2: Wrapping C

Part 1

In directory `wrapping` is a simple C library computing the `sin` elementwise for a Numpy array. The header `stuff.h` defines a function with the signature:

```
void compute(int n, double *input, double *output)
```

which takes two C arrays of doubles containing `n` doubles.

Write a Cython wrapper:

```
def do_compute(input_array):
    ...
    return output_array
```

for this function.

You can address the `double*` data contained in a Numpy array via the `.data` attribute of the corresponding `cdef`-ed variable. (Remember to check `.flags.c_contiguous`!)

Part 2 (only do at the end if you have time)

In the directory `wrapping2` is an old library written in C that generates anagrams from words. Write a Cython wrapper for it; this requires some string and resource handling.

The only thing you need to know about the C library is that its interface consists of two C functions:

```
char* simple_anagram(char *dictfile, char *word, int index)
void simple_anagram_free(char *data)
```

Usage is as follows:

1. Call `simple_anagram` with the name of a dictionary file, a word you want to generate anagrams for, and the number of the anagram you want to generate (starts with 0).
2. If there is an anagram corresponding to the number, it returns a C string containing the anagram. Otherwise, you get `NULL`.
3. You will need to free the returned C string by calling `simple_anagram_free` on it.

Note: Handling allocatable resources in C needs more care than in Python. In Cython, you can create a Python string copy of a C string by assigning it to a variable declared to be of type `cdef object`.

7.3 Exercise 3: Conway's Game of Life

The Game of Life is a well-known cellular automaton, whose behavior is interesting in many ways. It is defined on a grid of cells, where each cell has 8 neighbors:

```
.........
.........
...ooo...
...oxo...
...ooo...
.........
.........
```

The update rule to get a new state from the old one is:

- Cells with with less than 2 or more than 3 live neighbors die.
- Cell with exactly 3 live neighbors becomes alive.

Write a Cython function `life_update(old_state, new_state)` that takes an *N x N* Numpy array *old_state* of type `int8` containing the old state, and writes the new state to a similar array *new_state*. Just use four nested for-loops, but remember to add type declarations.

Some image file containing well-know interesting shapes are supplied (use `matplotlib.pyplot.imread` to read them into Numpy arrays). Assign them at the center of a big grid, and see what happens!

- `glider.png`: the glider
- `glider_gun.png`: the Gosper glider gun
- `breeder.png`: one sort of a breeder
- ... and others!

These examples come from the very interesting Game of Life simulator Golly.

Animation in Matplotlib

For visualization, a quick-and-dirty way is to show an animation with Matplotlib. Like so:

```
import matplotlib.pyplot as plt
import time
from life import life_update # <-- comes from your Cython module

# put some starting image into state_1
state_1 = ...
state_2 = np.zeros_like(state_1)

# Prepare animation
pixel_size = 2

plt.ion()
fig = plt.figure(dpi=50, figsize=(pixel_size * state_1.shape[1]/50.,
                                  pixel_size * state_2.shape[0]/50.))
plt.axes([0, 0, 1, 1])
img = plt.imshow(state_1, interpolation='nearest')
plt.gray()

print "Press Ctrl-C in the terminal to exit..."

# Animate
try:
    while True:
        life_update(state_1, state_2)
        state_1, state_2 = state_2, state_1   # swap buffers
        img.set_data(state_1)
        plt.draw()
        time.sleep(0.01)
except KeyboardInterrupt:
    pass
```

7.4 Exercise 4: On better algorithms...

Sometimes, the right solution is to (also) use a better algorithm.

Take the `gravity` example, and in the time step function interchange the order of position and velocity updates. This transforms the algorithm (Euler method) to a more appropriate one (the symplectic Euler method). Check what happens to the "exploding orbits" problem when the number of time steps is decreased.