Cython: Interfacing Python with C

Jørgen Høgberget, FV309

Department of Fabulous Physics
University of Oslo, N-0316 Oslo, Norway

Computational Physics

Interfacing? Cython?

- Interfacing: Creating a bridge of communication between different programming languages.
- Why? Python objects (strings, integers, floats) are not general for all languages.
- The 'real world' language analogue.
- We need something to interpret the python objects, translate them, and explain the translation to C.
- This is Cython's job!
- The 'real world' translator analogue.

Why use interfacing? Python vs. C++

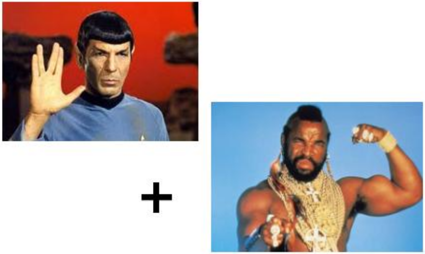
- C is **fast** whereas Python is **slow**.
- C can be an abstract mess to newcomers, whereas Python is beautiful and intuitive (art).
- C code usually takes longer to implement (pointers, declarations, **segmentation faults**, compilation), whereas Python is straight forward with excellent error handling.
- Getting (large) C codes to be structured and provide a sufficient error feedback usually requires a high(er) understanding of the language.
- Expanding/altering a (large) Python code is very simple in comparison to C/C++.

Summary, basic idea

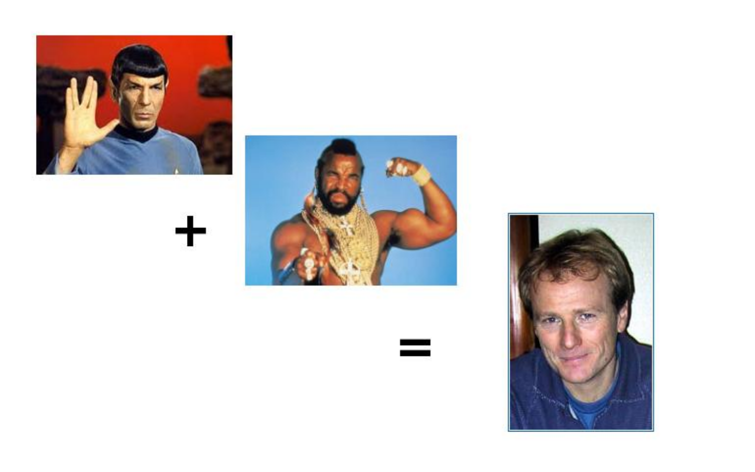
You take the best of two worlds and combine them to fit **your needs**.

Illustrative example

Illustrative example



Illustrative example



Cython? Why bother?

- Reduces program runtime by 50-100 times if done correctly, depending on the complexity of the code.
- Requires very little additional work: **90% of the job is writing the python code!**
- All *Cythonic* statements are seperate lines: No need to alter the original code structures.
- Compilation is automatic, all you need to specify is a name.
- Cython fully supports the familiar Numpy arrays.
- Sections left without any Cython 'seasoning' are run with Python.

A small Python example: 42.5s runtime

```
from numpy import *

def calc():
    n = 10000000
    s = 0
    c = zeros(n)
    for i in range(n):
        c[i] = 1;
        c[i] +=1;
        c[i] -=1;
        c[i] *=2;
        c[i] /=2;
        c[i] /= c[i]
        s += c[i]

    return s/n

print calc()
```

The same thing in standard c++: 0.24s runtime

```
double calc(){
  int i, n;
  double s;
  n = 10000000;
  double* c = new double[n];
  for (i = 0; i<n; i++){
    c[i] = 1;
    c[i] += 1;
    c[i] -= 1;
    c[i] *= 2;
    c[i] /= 2;
    c[i] /= c[i];
    s += c[i];
  }

  return s/n;
}
```

A small Cython example: 0.24s runtime

```
import numpy as np
cimport numpy as np
cimport cython

ctypedef np.float_t DTYPE_t

@cython.boundscheck(False)
cdef double calc():
    cdef int i, n
    cdef double s
    cdef np.ndarray[DTYPE_t] c

    ... (100% equivalent code) ...
```

Cython strategy

Strategy:

- Copy your Python code and rename it with **.pyx**
- Copy-paste the import statements.
- Assuming your Python code is bug-free, turn off array boundschecks.
- Declare variables used in the **slow regions of the code.**
- Remember: Profile your code. No need to cythonize everything.

Compiling the linked library

- Copy-paste the `setup.py` file:

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

numpy = "/local/lib/python2.5/site-packages/numpy/core/include/"

setup(
    cmdclass = {'build_ext': build_ext},
    ext_modules = [Extension("my_lib", ["superfast.pyx"],
                             include_dirs = [numpy])]
)
```

- Create the linked library by running the setup.py file. Then you may simply import it:

```
...$ python setup.py build_ext --inplace
...$ python -c "import my_lib"
```

Example: Project1 FYS3150

C++ -O3, Python and Cython with $n = 1000$ scaled with 10^{-4}:

- C++ with O3 optimization: $1.88s$.
- Python: $110s$.
- Cython with one of the original functions cythonized: $2.4s$.

Exp, sqrt, log etc. in Cython

```
from libc.math cimport exp as c_exp

#RHS function and the corresponding exact solution
    of Poisson's eq.
cdef double f(double x):
    return 100*c_exp(-10*x)
```

Profiling in Python

```
>>> import profile
>>> profile.run('profile_me.main()',sort=1)
total time used: 30.6238 s
         1000031 function calls in 19.126 CPU seconds

   Ordered by: internal time

   ncalls  tottime  ...  percall  filename:lineno(function)
        1   10.265  ...   19.069  profile_me.py:8(speed_me_up)
  1000000    8.731  ...    0.000  profile_me.py:5(<lambda>)
        3    0.073  ...    0.024  :0(range)

     ...

        1    0.000         0.000  prof..(set_boundaries)
```

Do not bother optimizing `set_boundaries()`..!