Need for Speed – Python meets C/C++

Scientific Programming with Python

Christian Elsasser

Python is nice, but by construction slow . . .

[xkcd]

. . . why not therefore interfacing it with C/C++
(or something similar, *e.g.* if you don't feel too young to use Fortran)

Easy to use/Flexibility

Python

Cython → Performance improvement
SWIG → Interfaces
boost::python
ctypes

. . .

C/C++

Fortran

Speed/Complexity

. . . why not therefore interfacing it with C/C++
(or something similar, *e.g.* if you don't feel too young to use Fortran)

Easy to use/Flexibility

Cython → Performance improvement

SWIG → Interfaces

boost::python

ctypes

Python

. . .

C/C++

Fortran

Speed/Complexity
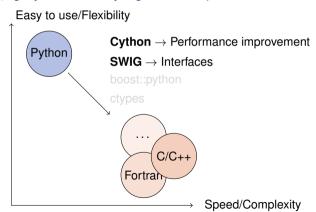
. . . why not therefore interfacing it with C/C++
(or something similar, *e.g.* if you don't feel too young to use Fortran)

Easy to use/Flexibility
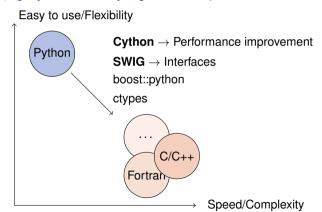
Cython → Performance improvement

SWIG → Interfaces

boost::python

ctypes

Python

. . .

C/C++

Fortran

Speed/Complexity

Table of Contents

- Introduction
  - Why should I care?
  - When should I consider something else?
- cython – a hybrid programming language/compiler
  - Speed-up examples
  - Standard Template Library
  - Classes
  - Exceptions
- SWIG (and other wrappers)

C++ on one Slide

- ► C++ is an (if not the) object-oriented programming language (like Python)
- ► including inheritance (like Python does in a slightly different way)
- ► ...operator overloading (like Python)
- ► It has a rich variety of libraries (like Python)
- ► It can raise exceptions (like Python)
- ► It requires declaration of variables (not like Python)
- ► **It is (usually) a compiled language! (not like Python)**

⇒ C++ and Python share a lot of similarities!

C is just the non-object-oriented version of C++ (minus some other missing features, *e.g.* exceptions)

A Few Words of Warning

Bad code stays bad code! – Better clean it up than trying to overpaint it!

Do not expect miracles! – You have to master two languages!

C keeps Python running . . .

- ▶ CPython is the standard implementation of the Python interpreter written in C.
- ▶ The Python C API (application programming interface) allows to build C libraries that can be imported into Python (https://docs.python.org/3/c-api/) . . .
- ▶ . . . and looks like this:

Pure Python

```
>>> a = [1,2,3,4,5,6,7,8]
>>> sum(a)
36
```

. . . but takes a lot of the fun out of Python

Python C can understand

```
sum_list(PyObject *list) {
   int i, n;
   long total = 0;
   PyObject *item;
   n = PyList_Size(list);
   if (n < 0)
     return -1; /* Not a list */
   for (i = 0; i < n; i++) {
     item = PyList_GetItem(list, i); /* Can't fail */
     if (!PyInt_Check(item)) continue; /* Skip non-integers */
     total += PyInt_AsLong(item);
   }
   return total;
}
```

C/C++ in Python: Not a New Thing

NumPy's C API

```
ndarray typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
} PyArrayObject ;
```

$\Rightarrow$ Several Python "standard" libraries are using C/C++ to speed things up

Cython – An easy way to get C-enhanced compiled Python code

(http://cython.org)

- ▶ Hybrid programming language combining Python and an interface for using C/C++ routines.
- ▶ ...or a static compiler for Python allowing to write C/C++ extensions for Python and heavily optimising this code.
- ▶ It is a successor of the Pyrex language.

⇒ Every valid Python statement is also valid when using cython.

⇒ Code needs to be compiled → Time!

- ▶ Translates you "C-enhanced" Python code into C/C++ code using the C API

Cython (v0.25.2) understands Python 3, and also most of the features of C++11

Requirements: Cython package and a C compiler

- ► cython
  The latest version can be downloaded from http://cython.org.

- ► C/C++ compiler, *e.g.* gcc/g++/clang (or for Windows: mingw)

  Mille viae ducunt hominem per saecula ad compilorem!

  Linux: usually already installed
                   (Ubuntu/Debian: sudo apt-get install build-essential)

  MacOS X: XCode command line tools

  Windows: Download of MinGW from http:// mingw.org and install it

Benchmark One: Fibonacci series

Fibonacci (Pure Python)

```python
def fib(n):
  a,b = 1,1
  for i in range(n):
    a,b = a+b,a
  return a
```

Benchmark One: Fibonacci series

Fibonacci (Cython)

```
def fib(int n):
  cdef int i,a,b
  a,b = 1,1
  for i in range(n):
    a,b = a+b,a
  return a
```

▶ Type declaration (cdef) ⇒ Python/Cython knows what to expect

Benchmark One: Fibonacci series
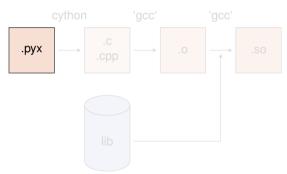
Fibonacci (Cython)

```
def fib(int n):
  cdef int i,a,b
  a,b = 1,1
  for i in range(n):
    a,b = a+b,a
  return a
```

- ▶ Type declaration (cdef) ⇒ Python/Cython knows what to expect

- ▶ A few (simple) modifications can easily change the CPU time by a factor of $\mathcal{O}(100)$
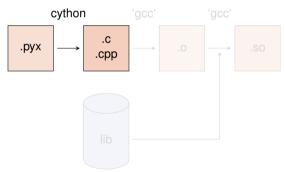
Compiling Cython Code (The hard way)

1. Compile Cython code to C/C++ code
   cython/cython3 <name>.pyx

2. Create object files
   gcc -O2 -fPIC
   -I<path_to_python_include> -c
   <name>.c -o <name>.o

3. Compile shared object (*i.e.* library)
   gcc [options]
   -L<path_to_python_library>
   <name>.o -o <name>.so

   ▶ If using C++ code, cython needs the
     option -+ and gcc → g++
   ▶ options are for MacOS X -bundle
     -undefined dynamic_lookup and
     for Debian -shared

Shared object (<name>.so) can be imported into
Python with import name

Compiling Cython Code (The hard way)

cython   'gcc'   'gcc'

.pyx → .c .cpp → .o → .so

lib

Shared object (`<name>.so`) can be imported into Python with `import name`

1. Compile Cython code to C/C++ code
   `cython/cython3 <name>.pyx`

2. Create object files
   `gcc -O2 -fPIC`
   `-I<path_to_python_include> -c`
   `<name>.c -o <name>.o`

3. Compile shared object (*i.e.* library)
   `gcc [options]`
   `-L<path_to_python_library>`
   `<name>.o -o <name>.so`
   ▸ If using C++ code, cython needs the option `-+` and gcc → g++
   ▸ `options` are for MacOS X `-bundle -undefined dynamic_lookup` and for Debian `-shared`
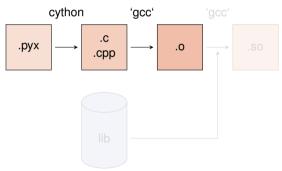
Compiling Cython Code (The hard way)

cython   'gcc'   'gcc'

.pyx → .c .cpp → .o → .so

lib

Shared object (<name>.so) can be imported into Python with `import name`

1. Compile Cython code to C/C++ code
   `cython/cython3 <name>.pyx`
2. Create object files
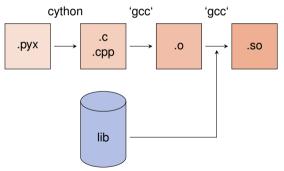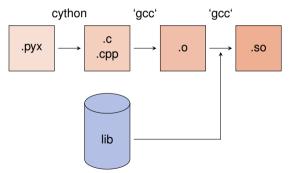   `gcc -O2 -fPIC -I<path_to_python_include> -c <name>.c -o <name>.o`
3. Compile shared object (*i.e.* library)
   `gcc [options] -L<path_to_python_library> <name>.o -o <name>.so`
   ▶ If using C++ code, cython needs the option `-+` and gcc → g++
   ▶ `options` are for MacOS X `-bundle -undefined dynamic_lookup` and for Debian `-shared`

Compiling Cython Code (The hard way)

cython   'gcc'   'gcc'

.pyx → .c / .cpp → .o → .so

lib

Shared object (`<name>.so`) can be imported into Python with `import name`

1. Compile Cython code to C/C++ code
   `cython/cython3 <name>.pyx`
2. Create object files
   `gcc -O2 -fPIC`
   `-I<path_to_python_include> -c`
   `<name>.c -o <name>.o`
3. Compile shared object (*i.e.* library)
   `gcc [options]`
   `-L<path_to_python_library>`
   `<name>.o -o <name>.so`
   - If using C++ code, cython needs the option `-+` and gcc → g++
   - `options` are for MacOS X `-bundle -undefined dynamic_lookup` and for Debian `-shared`

Compiling Cython Code (The hard way)

cython   'gcc'   'gcc'

.pyx → .c .cpp → .o → .so

lib

Shared object (<name>.so) can be imported into Python with `import name`

1. Compile Cython code to C/C++ code
   `cython/cython3 <name>.pyx`
2. Create object files
   `gcc -O2 -fPIC`
   `-I<path_to_python_include> -c`
   `<name>.c -o <name>.o`
3. Compile shared object (*i.e.* library)
   `gcc [options]`
   `-L<path_to_python_library>`
   `<name>.o -o <name>.so`
   - If using C++ code, cython needs the option `-+` and gcc → g++
   - `options` are for MacOS X `-bundle -undefined dynamic_lookup` and for Debian `-shared`

Compiling Cython Code (The easy way)

Support via the distutils (distribution utilities) package in building and installing Python modules
$\Rightarrow$ applicable for cython

setup.py

```
from distutils.core import setup
from Cython.Build import cythonize

setup(ext_modules = cythonize([<list of .pyx files>],
                              language="c++" # optional
                              )
     )
```

Command `python setup.py build_ext --inplace` creates for each `.pyx` file a `.c/.cpp` file, compiles it to an executable (in the build directory of the corresponding OS/architecture/Python version) and compiles a `.so` file (or a `.pxd` if you are using Windows)

Further options for `cythonize` via help explorable

How Performant is My Code?

`cython -a/--annotate <name>.pxy` $\rightarrow$ additional HTML file

- ▶ bad performance $\rightarrow$ yellow marking
- ▶ allows to investigate code and to learn about performance tuning


```
Generated by Cython 0.26

Yellow lines hint at Python interaction.
Click on a line that starts with a "+" to see the C code that Cython generated for it.

Raw output: fib_py.c

 1: # Calculation of n-th fibonacci number
+2: def fib(n):
+3:     a,b = 1,1
+4:     for i in range(n):
+5:         a,b = a+b,a
+6:     return a
    __Pyx_XDECREF(__pyx_r);
    __Pyx_INCREF(__pyx_v_a);
    __pyx_r = __pyx_v_a;
    goto __pyx_L0;
```

- ▶ Not every yellow part can be improved!

Benchmark Two: Numerical Integration

Integral of $f(x) = \sin x \cdot e^{-x}$ between 0 and π

$\Rightarrow$ Exact result: $(e^{-\pi} + 1)/2 = 0.521607$

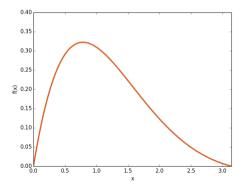## Benchmark Two: Numerical Integration

Integral of $f(x) = \sin x \cdot e^{-x}$ between 0 and π
$\Rightarrow$ Exact result: $(e^{-\pi} + 1)/2 = 0.521607$

Integrate

```
from math import sin,exp

def f(double x):
  return sin(x)*exp(-x)

def integrate(double a,double b,int N):
  cdef double dx,s
  cdef int i
  dx = (b-a)/N
  s = 0.0
  for i in range(N):
    s += f(a+(i+0.5)*dx)
  return s*dx
```

Benchmark Two: Numerical Integration

Integral of $f(x) = \sin x \cdot e^{-x}$ between 0 and π
$\Rightarrow$ Exact result: $(e^{-\pi} + 1)/2 = 0.521607$

Python layer (expensive)	C layer (cheap)
`integrate(a,b,N)`	`'_pyx_integrate'(a,b,N)` `for (i=0; i<N; i++)`
`f(x)`	`'_pyx_f'(x)` `sum updated`

Benchmark Two: Numerical Integration

Integral of $f(x) = \sin x \cdot e^{-x}$ between 0 and π
$\Rightarrow$ Exact result: $(e^{-\pi} + 1)/2 = 0.521607$

Integrate

```
from math import sin,exp

cdef double f(double x):
  return sin(x)*exp(-x)

def integrate(double a,double b,int N):
  cdef double dx,s
  cdef int i
  dx = (b-a)/N
  s = 0.0
  for i in range(N):
    s += f(a+(i+0.5)*dx)
  return s*dx
```

Benchmark Two: Numerical Integration

Integral of $f(x) = \sin x \cdot e^{-x}$ between 0 and π
$\Rightarrow$ Exact result: $(e^{-\pi} + 1)/2 = 0.521607$

Integrate

```
from math import sin,exp

cpdef double f(double x):
  return sin(x)*exp(-x)

def integrate(double a,double b,int N):
  cdef double dx,s
  cdef int i
  dx = (b-a)/N
  s = 0.0
  for i in range(N):
    s += f(a+(i+0.5)*dx)
  return s*dx
```

Benchmark Two: Numerical Integration

Integral of $f(x) = \sin x \cdot e^{-x}$ between 0 and π
$\Rightarrow$ Exact result: $(e^{-\pi} + 1)/2 = 0.521607$

Integrate

```
from libc.math cimport sin,exp

cpdef double f(double x):
  return sin(x)*exp(-x)

def integrate(double a,double b,int N):
  cdef double dx,s
  cdef int i
  dx = (b-a)/N
  s = 0.0
  for i in range(N):
    s += f(a+(i+0.5)*dx)
  return s*dx
```

Benchmark Two: Numerical Integration

Integral of $f(x) = \sin x \cdot e^{-x}$ between 0 and π
$\Rightarrow$ Exact result: $(e^{-\pi} + 1)/2 = 0.521607$

Integrate

```
from libc.math cimport sin,exp

cpdef double f(double x):
  return sin(x)*exp(-x)

def integrate(double a,double b,int N):
  cdef double dx,s
  cdef Py_ssize_t i
  dx = (b-a)/N
  s = 0.0
  for i in range(N):
    s += f(a+(i+0.5)*dx)
  return s*dx
```

Benchmark Two: Numerical Integration

Integral of $f(x) = \sin x \cdot e^{-x}$ between 0 and π
$\Rightarrow$ Exact result: $(e^{-\pi} + 1)/2 = 0.521607$

- ▶ Return values of function can be specified via the key word `cdef`
- ▶ `cpdef` $\Rightarrow$ function also transparent to Python itself (no performance penalty)

- ▶ C/C++ library can be imported via `from libc/libcpp.<module> cimport <name>` (see later)
- ▶ Using C++ functions can lead to a huge speed-up
- ▶ Try to do as much as you can in the C-layer

- ▶ Already huge speed-up when levaraging numpy and its vectorisation

You are here!

STL Containers

An often used feature of C++ are the Standard Template Library containters (*e.g.* std::vector, std::map, etc.)

Object holders with specific memory access structure, *e.g.*

- ▶ std::vector allows to access any element
- ▶ std::list only allows to access elements via iteration
- ▶ std::map represents an associative container with a key and a mapped values

STL Containers

An often used feature of C++ are the Standard Template Library containters (*e.g.* std::vector, std::map, etc.)

... and Cython knows how to treat them!

Python ⟶ C++ ⟶ Python

```
        iterable   →   std::vector   →   list
        iterable   →    std::list    →   list
        iterable   →    std::set     →   set
iterable (len 2)   →    std::pair    →   tuple (len 2)
            dict   →    std::map     →   dict

           bytes   →   std::string   →   bytes
```

STL Containers

An often used feature of C++ are the Standard Template Library containters (*e.g.* std::vector, std::map, etc.)

A few remarks!

- ▸ iterators (*e.g.* it) can be used ⇒ dereferencing with `dereference(it)` and incrementing/decrementing with `preincrement` (*i.e.* ++it), `postincrement` (*i.e.* it++), `predecrement` (*i.e.* --it) and `postdecrement` (*i.e.* it--) from `cython.operator`
- ▸ Be careful with performance! ⇒ performance lost due to shuffling of data
- ▸ More indepth information can be found directly in the corresponding sections of the cython code https://github.com/cython/cython/tree/master/Cython/Includes/libcpp
- ▸ C++11 containters (like `std::unordered_map`) are partially implemented

Exceptions/Errors

In terms of exception and error handling three different cases need to be considered:

- ► Raising of a **Python error** in cython code ⇒ return values make it impossible to raise properly Python errors (Warning message, but continuing)
- ► Handling of **error codes from pure C functions**
- ► Raising of a **C++ exception** in C++ code used in cython ⇒ C++ exception terminates – if not caught – program

Errors in Python

Python Error

```
cpdef int raiseError():
    raise RuntimeError("A problem")
    return 1
```

$\Rightarrow$ Just prints a warning (and worse gives an ambigious return value)

Errors in Python

Python Error

```
cpdef int raiseError():
    raise RuntimeError("A problem")
    return 1
```

⇒ Just prints a warning (and worse gives an ambigious return value)

Python Error

```
cpdef int raiseError() except *:
    raise RuntimeError("A problem")
    return 1
```

⇒ Propagates the RuntimeError

Errors in C

C does not know exceptions like Python or C++. If errors should be caught, it is usually done via dedicated return values of functions which cannot appear in a regular function call.

Use the except statement to tell cython about this value

C Error

```
cpdef int raiseException() except -1:
    return -1
```

$\Rightarrow$ allows to indicate error codes from C $\Rightarrow$ raises SystemError
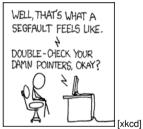
Exceptions in C++

[xkcd]

In cython this is also true for C++ exceptions!

Cython is not able to deal with C++ exceptions in a try'n'except clause!

⇒ But caption in cython and translation to Python exceptions/errors is possible!

Exceptions in C++

. . . and how to tackle them!

- `cdef <C++ function>() except +`
  ⇒ translates a C++ exception into a Python error according to the right-hand scheme

- `cdef <C++ function>() except +<Python Error>` *e.g.* `MemoryError` ⇒ translates every thrown C++ exception into a MemoryError

- `cdef <C++ function>() except +<function raising Python error>` ⇒ runs the indicated function if the C++ function throws any exception. If `<function raising Python error>` does not raise an error, a RuntimeError will be raised.

C++ ⟶ Python		
bad_alloc	→	MemoryError
bad_cast	→	TypeError
domain_error	→	ValueError
invalid_argument	→	ValueError
ios_base::failure	→	IOError
out_of_range	→	IndexError
overflow_error	→	OverflowError
range_error	→	ArithmeticError
underflow_error	→	ArithmeticError
(all others)	→	RuntimeError

Classes

Classes are a common feature of Python and C++

There are two aspects when dealing with cython:

- ► **Defining classes containing C++ code in cython**
- ► C++ classes integrated into Python

Defining Classes in Cython

Let's go back to the integration examples

Integrate with classes

```
cdef class Integrand:
    cpdef double evaluate(self,double x) except *:
        raise NotImplementedError()

cdef class SinExpFunction(Integrand):
    cpdef double evaluate(self,double x):
        return sin(x)*exp(-x)

def integrate(Integrand f,double a,double b,int N):
    ...
    s += f.evaluate(a+(i+0.5)*dx)
    ...
```

Cython does not know @abstractmethod from the module abc!

Defining Classes in Cython

Let's go back to the integration examples

Integrate with classes

```
class Poly(Integrand):
    def evaluate(self,double x):
        return x*x-3*x


integrate(Poly(),0.0,2.0,1000)
```

⇒ Speed lost with respect to definition in cython, but still faster than a pure Python implementation

Integration of C++ Classes in Cython – Possible but cumbersome

Starting point: `.cpp/.h` file for class `Rectangle` defined in a namespace `shapes`

1. Expose it to Cython by delaring the class structure and method signatures
2. Integrating it into Cython either via direct usage or by defining a wrapper class

rect.pyx (File to expose C++ class to cython)

```
# distutils:  language = c++
# distutils:  sources = Rectangle.cpp

cdef extern from "Rectangle.h" namespace "shapes":
    cdef cppclass Rectangle:
        Rectangle(int, int, int, int) except +
        int x0, y0, x1, y1
        int getLength()
        int getHeight()
        int getArea()
        void move(int, int)
```

Automatic Wrappers

. . . since not everybody likes to write lines of error-prone code

- ▶ SWIG
- ▶ boost::python
- ▶ ctypes
- ▶ . . .

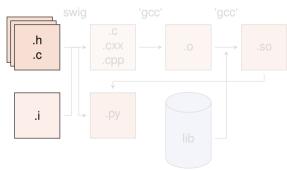Goal: creating compilable C/C++ code based on the Python C API

SWIG

SWIG: Simplified Wrapper and Interface Generator

- ► Generic Wrapper for C/C++ to script-like languages
    - ► R
    - ► Perl
    - ► Ruby
    - ► Tcl
    - ► PHP5
    - ► Java
    - ► ...and **Python**
- ► Pretty old – created in 1995 by Dave Beazley
- ► Current version is 3.0.12

SWIG – in a Nutshell

1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Create object files based on output from the wrapper plus native C/C++ code
3. Compile shared object (*i.e.* library)

   Normally step 2 and 3 can be combined with via Distutils `setup.py`
   `python setup.py build_ext --inplace`

Moduel (`<name>.py`) can be imported into Python with `import name` ⇒ Shared object needs different name
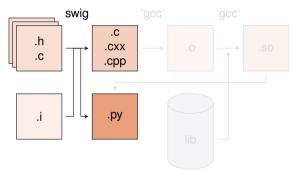
SWIG – in a Nutshell

1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`

2. Create object files based on output from the wrapper plus native C/C++ code

3. Compile shared object (*i.e.* library)

   Normally step 2 and 3 can be combined with via Distutils `setup.py`
   ```
   python setup.py build_ext
   --inplace
   ```

Moduel (`<name>.py`) can be imported into Python with `import name` ⇒ Shared object needs different name

SWIG – in a Nutshell

1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Create object files based on output from the wrapper plus native C/C++ code
3. Compile shared object (*i.e.* library)
   Normally step 2 and 3 can be combined with via Distutils `setup.py`
   `python setup.py build_ext --inplace`

Moduel (`<name>.py`) can be imported into Python with `import name` ⇒ Shared object needs different name

SWIG – in a Nutshell

1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Create object files based on output from the wrapper plus native C/C++ code
3. Compile shared object (*i.e.* library)

   Normally step 2 and 3 can be combined with via Distutils `setup.py`
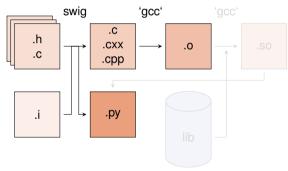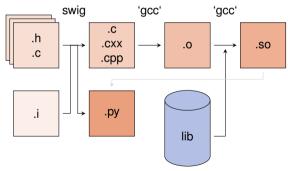   `python setup.py build_ext --inplace`

Moduel (`<name>.py`) can be imported into Python with `import name` ⇒ Shared object needs different name

SWIG – in a Nutshell

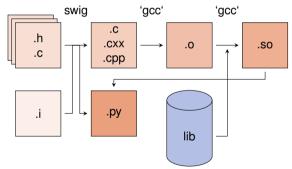
1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Create object files based on output from the wrapper plus native C/C++ code
3. Compile shared object (*i.e.* library)

   Normally step 2 and 3 can be combined with via Distutils `setup.py`
   `python setup.py build_ext`
   `--inplace`

Moduel (`<name>.py`) can be imported into Python with `import name` ⇒ Shared object needs different name

SWIG – The interface file

Main configuration with interface (`.i`) files

- ▶ tells which (header) file(s) contains the C/C++ code to wrap
- ▶ defines some special data types (*e.g.* `std::vector<...>`)
- ▶ handles some additional configuration (*e.g.* exception/error translation)

Interface file

```
%module geom // name of the module
...
// things swig should know about
%include "Shape.h"
%include "Rectangle.h"
...
// things that should be put into the
header of the wrapper file (.c/.cxx)
%{
#include "Shape.h"
#include "Rectangle.h"
%}
```

SWIG – The Distutils file

Distutils (`setup.py`)

```
from distutils.core import setup, Extension

extension_mod = Extension("_<name>",
                          ["<name_wrap>.cxx",
                           "<source1>.cpp",
                           "<source2>.cpp","..."],
                          language="c++")

setup(name = "_<name>", ext_modules=[extension_mod])
```

- To be build extension needs a different name than the module set up by switch ⇒ Avoid name conflicts
- Language option only for C++
- `python setup.py build_ext --inplace`

A Few Remarks about SWIG

- SWIG $\approx$ performance loss with respect to cython
- If SWIG works: ☺
- If it does not: ☹
- . . . and therefore you can lose a lot of time with special problems
- It is not always optimal to expose the whole class to Python
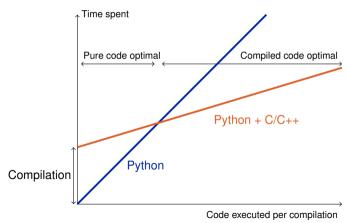
Conclusion

- ▸ Interfacing Python with C/C++ is – or better – can be a way to create powerful code
- ▸ cython and SWIG are two nice tools to do so
- ▸ ...but always make the interfacing maintainable/useful/etc. *i.e.* not a British train door
- ▸ And it's all about finding the sweet spot!

The Sweet Spot!

The End!

[xkcd]

References

1. Stéfan van der Walt, *Speeding up scientific Python code using Cython*, Advanced Scientific Programming in Python, 2013 (Zurich) & 2014 (Split)
2. Stefan Behnel et al., *Cython tutorial*, Proceedings of the 8th Python in Science Conference (SciPy 2009) $\Rightarrow$ based on older cython version, but the main reference of cython
3. Dave Beazley, *Swig Master Class*, PyCon'2008
4. http://docs.cython.org/src/tutorial/
5. http://www.swig.org

Backup

Fortran meets Python

The `f2py` compiler (http://docs.scipy.org/doc/numpy-dev/f2py/) offers – in a similar way as cython – the possibility to generate extension modules for Python from Fortran code.

`f2py -c -m <module name> <fortran file>.f/.f90 -I<path to python header file>` builds from the code in `<fortran file>.f/.f90` a importable module (*i.e.* shared object) `<module name>.so`

Fortran modules and subroutines are exposed to Python on time of the import of the built module.

The compilation can also be split into a first step generating a signature file, which is in a second step compiled into the extension module

Exceptions in C++

Examples
Two C++ functions `void raiseException()` and `void raiseBadAlloc()` defined in `except_cy.h`

Exception Example 1

```
cdef extern from 'except_cy.h'
    cdef void raiseException() except +
def tryIt():
    try:
        raiseException()
    except RuntimeError as e:
        print(e)
```

$\Rightarrow$ OK as `raiseException()` throws a `std::exception` $\rightarrow$ `RuntimeError`

Exceptions in C++

Examples
Two C++ functions `void raiseException()` and `void raiseBadAlloc()` defined in
`except_cy.h`

Exception Example 2

```
cdef extern from 'except_cy.h'
    cdef void raiseException() except +MemoryError
def tryIt():
    try:
        raiseException()
    except RuntimeError as e:
        print(e)
```

⇒ Not OK as `raiseException()` throws a `std::exception` which is explicitly transformed into a
`MemoryError`

Exceptions in C++

Examples
Two C++ functions `void raiseException()` and `void raiseBadAlloc()` defined in
`except_cy.h`

Exception Example 3

```
cdef extern from 'except_cy.h'
    cdef void raiseBadAlloc() except +
def tryIt():
    try:
        raiseBadAlloc()
    except RuntimeError as e:
        print(e)
```

$\Rightarrow$ Not OK as `raiseBadAlloc()` throws a `std::bad_alloc` which is transformed into a
`MemoryError`

Exceptions in C++

Examples
Two C++ functions `void raiseException()` and `void raiseBadAlloc()` defined in
`except_cy.h`

Exception Example 4

```
cdef extern from 'except_cy.h'
    cdef void raiseBadAlloc() except +
def tryIt():
    try:
        raiseBadAlloc()
    except MemoryError as e:
        print(e)
```

$\Rightarrow$ OK as `raiseBadAlloc()` throws a `std::bad_alloc` which is transformed into a `MemoryError`

Exceptions in C++

Examples
Two C++ functions `void raiseException()` and `void raiseBadAlloc()` defined in
`except_cy.h`

Exception Example 5

```
cdef void raise_py_error() except *:
    raise MemoryError("Problem")

cdef extern from 'except_cy.h':
    cdef void raiseBadAlloc() except +raise_py_error

def tryIt():
    try:
        raiseBadAlloc()
    except MemoryError as e:
        print(e)
```

$\Rightarrow$ OK as `raise_py_error()` throws an error

Exceptions in C++

Examples
Two C++ functions `void raiseException()` and `void raiseBadAlloc()` defined in
`except_cy.h`

Exception Example 6

```
cdef void raise_py_error() except *:
    pass

cdef extern from 'except_cy.h':
    cdef void raiseBadAlloc() except +raise_py_error

def tryIt():
    try:
        raiseBadAlloc()
    except MemoryError as e:
        print(e)
```

$\Rightarrow$ Not OK as no error is thrown by `raise_py_error()`

Integration of C++ Classes

Assuming a C++ class `Rectangle`

Rectangle.h

```cpp
namespace shapes {
    class Rectangle {
    public:
        int x0, y0, x1, y1;
        Rectangle(int x0, int y0, int x1, int y1);
        ~Rectangle(); // destructor
        int getLength();
        int getHeight();
        int getArea();
        void move(int dx, int dy);
    };
}
```

Integration of C++ Classes

Assuming a C++ class `Rectangle`

Rectangle.cpp

```cpp
#include "Rectangle.h"
#include <iostream>

using namespace shapes;

Rectangle::Rectangle(int X0, int Y0, int X1, int Y1) {
    x0 = X0;
    y0 = Y0;
    x1 = X1;
    y1 = Y1;
    std::cout << "Here I am" << std::endl;}

Rectangle::~Rectangle() {
    std::cout << "Byebye" << std::endl;}

...
```

Integration of C++ Classes

Now exposing it to cython

rect.pyx

```
# distutils:  language = c++
# distutils:  sources = Rectangle.cpp
cdef extern from "Rectangle.h" namespace "shapes":
    cdef cppclass Rectangle:
        Rectangle(int, int, int, int) except +
        int x0, y0, x1, y1
        int getLength()
        int getHeight()
        int getArea()
        void move(int, int)
```

Integration of C++ Classes

. . . and using it!

Either in further cython code!

rect.pyx

```
def tryIt():
    cdef Rectangle* r
    try:
        r = new Rectangle(1,2,3,4)
        print("My length is:  %f"%r.getLength())
        print("My first x-coordinate is:  %f"%r.x0)
    finally:
        del r
```

Integration of C++ Classes

. . . and using it!

Or for creating a Python (wrapper) class!

rect.pyx

```
cdef class PyRectangle:
    cdef Rectangle *thisptr
    def __cinit__(self, int x0, int y0, int x1, int y1):
        self.thisptr = new Rectangle(x0, y0, x1, y1)
    def __dealloc__(self):
        del self.thisptr
    def getLength(self):
        return self.thisptr.getLength()
    def getHeight(self):
        return self.thisptr.getHeight()
    ...
```

Special features: STL Stuff with SWIG

- ▶ Dedicated interface files need to be integrated when running SWIG
- ▶ . . . and templates for **each** containers + **each** content need to be defined

Interface file

```
...
%include "std_vector.i"
%include "std_string.i"
...
%template(dVector) std::vector<double>;
%template(rectVector) std::vector<Rectangle*>;
...
```

Special features: Exceptions with SWIG

Interface file

```
...
%include "exception.i"
...
%exceptionclass ShapeError;
%exception *::whine {

  try {

    $action

  } catch(ShapeError & e) {

    ShapeError *ecopy = new ShapeError(e);
    PyObject *err = SWIG_NewPointerObj(ecopy, SWIGTYPE_p_ShapeError, 1);
    PyErr_SetObject(SWIG_Python_ExceptionType(SWIGTYPE_p_ShapeError), err);
    SWIG_fail;

  }
}
```

Special features: Overloading

Cython deals the usual way with overloaded methods in C++:

`rect.pyx` works

```
cdef extern from "Rectangle.h" namespace "shapes":
        ...
        void move(int, int)
        void move(int)
```

but it cannot happen in a Python wrapper class:

`rect.pyx` does not work

```
cdef class PyRectangle:
    ...
    def move(self,dx,dy):
        return self.thisptr.move(dx,dy)
    def move(self,d):
        return self.thisptr.move(d)
```

Special features: Inheritance

As in Python C++ classes can inherit from parent classes including overriding of methods

C++ classes

```cpp
class Shape {
public:
    ...
    void virtual printInfo(); // Prints "Shape"
};

class Rectangle :  public Shape {
public:
    ...
    void printInfo(); // Prints "Rectangle"
};
```

Special features: Inheritance

Cython can also deal with this feature, but there are two points to keep in mind:
1. If parent class is also exposed to cython, no redefinition of overridden methods is required
(and also allow → mis-interpreted as overloading)

C++ classes

```
cdef extern from "Rectangle.h" namespace "shapes":
    cdef cppclass Shape:
        Shape() except +
        void printInfo()

    cdef cppclass Rectangle(Shape):
        Rectangle(int, int, int, int) except +
        ...
        void printInfo() # causes problems
        ...
```

Special features: Inheritance

2. The inheritance can only be transported into wrapper classes if child classes have the same set of methods as the mother class

C++ classes

```
cdef class PyObject:
    cdef Object* thisptr
    def __cinit__(self):
        self.thisptr = new Object()
    def __dealloc__(self):
        del self.thisptr
    def printInfo(self):
        self.thisptr.printInfo()

cdef class PyRectangle(PyObject):
    def __cinit__(self,int x0,int y0,int x1,int y1):
        self.thisptr = new Rectangle(x0,y0,x1,y1)
```
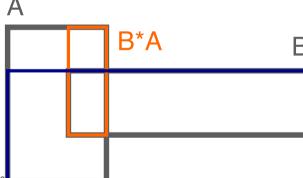
Special features: Operator Overloading

C++ as well as Python offers the potential to define operators for objects.

Example with Rectangles:

Special features: Operator Overloading

C++ code

```
Rectangle operator*(Rectangle& rhs){
    return Rectangle(x0,y0,rhs.x1,rhs.y1);
};
```

rect.pyx

```
# to expose it to cython
Rectangle operator*(Rectangle)


# in the wrapper class
def __mul__(PyRectangle lhs,PyRectangle rhs):
    res = PyRectangle(0,0,0,0)
    res.thisptr[0] = lhs.thisptr[0]*rhs.thisptr[0] # ptr deref
    return res
```

Arrays

Arrays in cython are usually treated via typed memoryviews (*e.g.* `double[:,:]` means a two-dimensional array of doubles, *i.e.* compatible with *e.g.* `np.ones((3,4))`)

Further you can specify which is the fastest changing index by `:1`, *e.g.*

- `double[::1,:,:]` is a F-contiguous three-dimensional array
- `double[:,:,::1]` is a C-contiguous three-dimensional array
- `double[:,::1,:]` is neither F- nor C-contiguous

For example a variable `double[:,::1]` a has as NumPy arrays variables like `shape` and `size` and the elements can be accessed by `a[i,j]`

But be aware: NumPy is already heavily optimised, so do not to reinvent the wheel!