



# Need for Speed – Python meets C/C++

Scientific Programming with Python

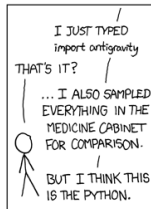
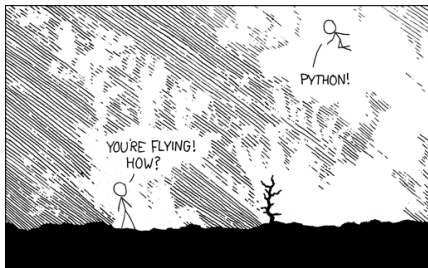
Christian Elsasser

Based partially on a talk by Stéfan van der Walt



This work is licensed under the *Creative Commons Attribution-ShareAlike 3.0 License*.

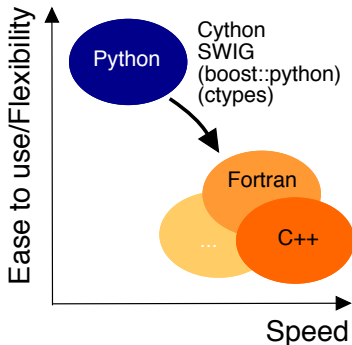
## Python vs. C/C++



[xkcd]



## Python vs. C/C++



- ▶ Python is nice, but by construction slow ...
- ▶ ... therefore interfacing it with C/C++ (or something similar, e.g. if you don't feel too young to use Fortran)



## Table of Contents

- ▶ Introduction
  - ▶ A few words of warning
  - ▶ C/C++ and its connection to Python
- ▶ cython – a hybrid programming language/compiler
  - ▶ Speed-up examples
  - ▶ Standard Template Library
  - ▶ Exceptions
  - ▶ Classes
- ▶ SWIG – an automatic wrapper



## A Few Words of Warning (I)

**Bad code stays bad code!**



## A Few Words of Warning (II)

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





## C++ on one Slide

[www.cplusplus.com](http://www.cplusplus.com) and [www.learncpp.com](http://www.learncpp.com)

- ▶ 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)**



## Python's C API

### ... or who the hell thinks this is useful?

- ▶ The Python C API (application programming interface) allows to build C libraries that can be imported into Python (<https://docs.python.org/2/c-api/>) ...
- ▶ ... and looks like this:

### Pure Python

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





## Python's C API

### 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 ;
```

⇒ Several Python “standard” libraries are using C/C++ to speed things up



## First Solution: Cython

(<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.

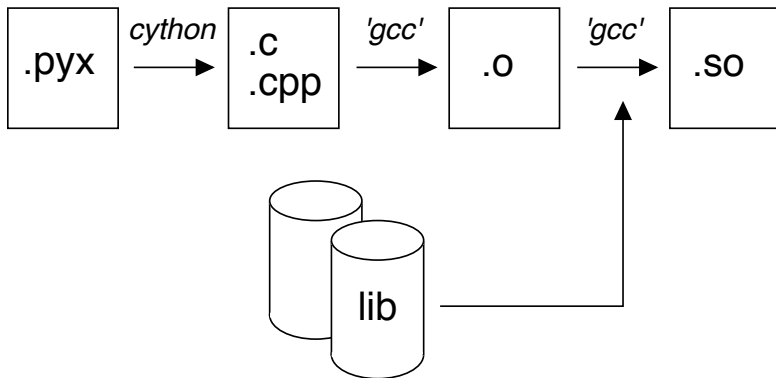
⇒ Every valid Python statement is also valid when using cython.

⇒ Code needs to be compiled → Time!

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

**Cython (v0.21.1) understands Python 3, and also most of the features of C++11**

## Workflow



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



## Requirements

- ▶ 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 compilem!*

**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 (Pure Python)

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



## Benchmark One

### 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 (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`)  $\Rightarrow$  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)

To get to importable Python extensions from Cython Code three steps are required:

1. Compile Cython code to C/C++ code

```
cython <name>.pyx  
→ creates <name>.c
```

2. Create files

```
gcc -O2 -fPIC -I<path_to_python_include> -c <name>.c -o  
<name>.o  
→ creates <name>.o without linking
```

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

```
gcc -bundle -undefined dynamic_lookup  
-L<path_to_python_library> <name>.o -o <name>.so  
→ creates <name>.so (this is a Mac-specific compile command!)
```

Module with <name> ready to be imported (*i.e.* `import <name>`)



## Compiling Cython Code (The easy way)

Support via the distutils (distribution utilities) package in building and installing Python modules ⇒ 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 → additional HTML file

- ▶ bad performance → yellow marking
- ▶ allows to investigate code and to learn about performance tuning

Generated by Cython 0.21.1

Raw output: [fib.c](#)

```
1: # Calculation of n-th fibonacci number
+2: def f(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;
```

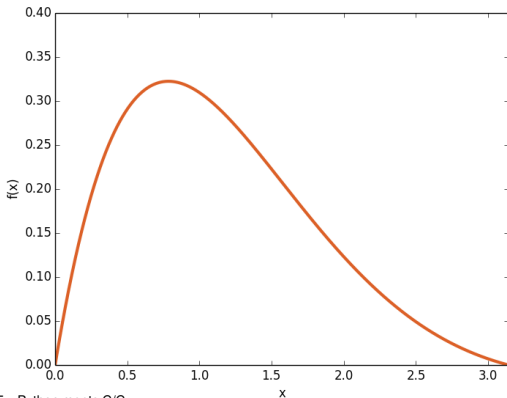
- ▶ Not every yellow part can be improved!



## Benchmark Two

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

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





## Benchmark Two

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

⇒ 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

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

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

### Python layer (expensive)

```
integrate(a,b,N)
```

```
f(x)
```

### C layer (cheap)

```
'_pyx_integrate'(a,b,N)
```

```
for (i=0; i<N; i++)
```

```
'_pyx_f'(x)
```

```
sum updated
```



## Benchmark Two

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

⇒ 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

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

⇒ 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

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

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

### Integrate

```
from libc.math cimport 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

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

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

### Integrate

```
from libc.math cimport 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 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

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

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

- ▶ Return values of function can be specified via the key word `cdef`
- ▶ `cpdef` ⇒ 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



## STL Containers

Nice and handy!





## STL Containers

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

... and cython knows how to treat them!

Python → C++ → Python

<code>iterable</code>	→	<code>std::vector</code>	→	<code>list</code>
<code>iterable</code>	→	<code>std::list</code>	→	<code>list</code>
<code>iterable</code>	→	<code>std::set</code>	→	<code>set</code>
<code>iterable (len 2)</code>	→	<code>std::pair</code>	→	<code>tuple (len 2)</code>
<code>dict</code>	→	<code>std::map</code>	→	<code>dict</code>
<code>bytes</code>	→	<code>std::string</code>	→	<code>bytes</code>



## STL Containers

### A few remarks!

- ▶ iterators (e.g. `it`) can be used  $\Rightarrow$  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!  $\Rightarrow$  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 containers (like `std::unordered_map`) are partially implemented

## Short Break! (5 min)



© Uli Stein & Koch <sup>Gesd</sup> Cartoon Edition GmbH, 30916 Isernhagen

You are here!





## Exceptions/Errors

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

- ▶ Raising of a Python error in cython code  $\Rightarrow$  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  $\Rightarrow$  C++ exception terminates – if not caught – program



## Errors in Python

### Python Error

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

⇒ Just prints a warning



## Errors in Python

### Python Error

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

⇒ Just prints a warning

### 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
```

⇒ allows to indicate error codes from C ⇒ raises `SystemError`

## Exceptions in C++



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++

### C++ → Python

<code>bad_alloc</code>	→	<code>MemoryError</code>
<code>bad_cast</code>	→	<code>TypeError</code>
<code>domain_error</code>	→	<code>ValueError</code>
<code>invalid_argument</code>	→	<code>ValueError</code>
<code>ios_base::failure</code>	→	<code>IOError</code>
<code>out_of_range</code>	→	<code>IndexError</code>
<code>overflow_error</code>	→	<code>OverflowError</code>
<code>range_error</code>	→	<code>ArithmeticError</code>
<code>underflow_error</code>	→	<code>ArithmeticError</code>
(all others)	→	<code>RuntimeError</code>



## Exceptions in C++

### ... and let Python understand it!

- ▶ `cdef <function raising exception>() except +`  
⇒ translates a C++ exception into a Python error according to the previous scheme
- ▶ `cdef <function raising exception>() except +<Python Error>`  
*e.g.* `MemoryError` ⇒ translates every thrown C++ exception into a `MemoryError`
- ▶ `cdef int <function raising Python error>()`  
`cdef <function raising exception>() except +<function raising Python error>` ⇒ runs `<function raising Python error>` if the C++ function throws any exception. If `<function raising Python error>` does not raise an error, a `RuntimeError` will be raised.



## 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

Assuming a C++ class Rectangle

### Rectangle.h

```
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

```
#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: ",r.getLength()
        print "My first x-coordinate is: ",r.x0
    finally:
        del r
```



## Integration of C++ Classes

... and using it!

Or for creating a Python (wrapper) class!

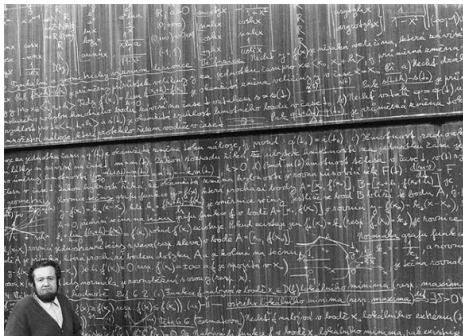
```
rect.pyx
```

```
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()
    ...
```

## Automatic Wrappers

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

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



Goal: creating “**automatically**” compilable C/C++ code based on the Python C API





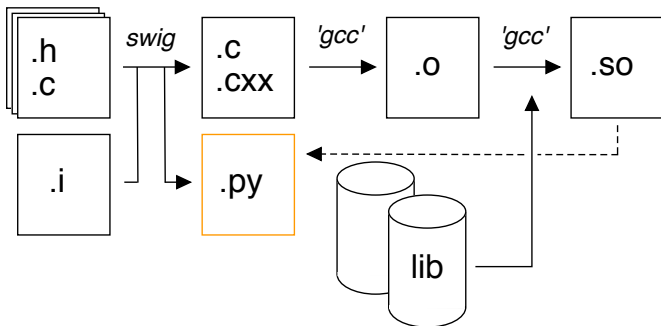
## SWIG

### SWIG: Simplified Wrapper and Interface Generator

- ▶ Generic Wrapper for C/C++ to script-like languages (Perl, Ruby, Tcl, ...)
- ▶ Pretty old – created in 1995 by Dave Beazley
- ▶ Current version is 3.0.2

## SWIG – in a Nutshell

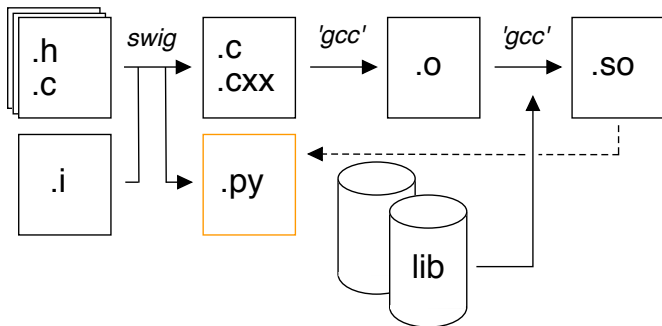
### Work flow



1. swig creates out of one or several .c/.h files via instructions in the .i file a wrapper file (.c or .cxx) and a python module (.py)  
`swig -python -c++/-c <file>.i`

## SWIG – in a Nutshell

### Work flow



2. Wrapper file is compiled to a shared object (`.so`) which will be imported by the python module (low- to high-level-translation)



## SWIG – in a Nutshell

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)



## SWIG – in a Nutshell

### 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 – in a Nutshell

### 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`



## 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*>;  
...
```



## 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





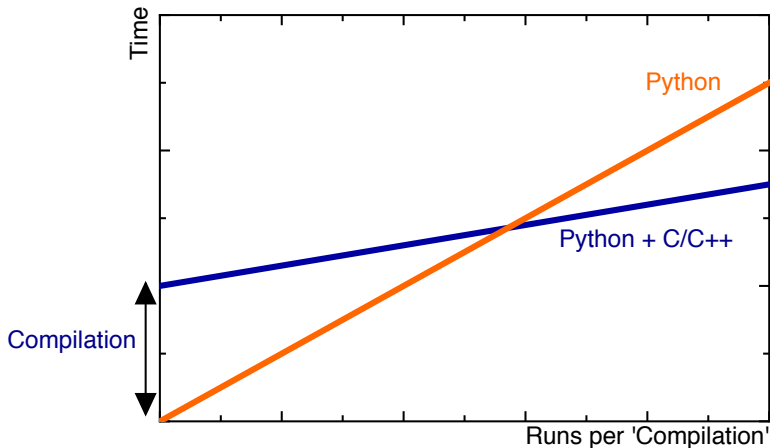
## Conclusion

- ▶ 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





## The End!



## 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 8<sup>th</sup> Python in Science Conference (SciPy 2009)  
⇒ 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>



University of  
Zurich<sup>UZH</sup>



Backup



## Python meets Fortran

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 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, e:  
        print e
```

⇒ OK as `raiseException()` throws a `std::exception` → `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, 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, e:  
        print e
```

⇒ 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, e:  
        print e
```

⇒ 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, e:
        print e
```

⇒ 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, e:
        print e
```

⇒ Not OK as no error is thrown by `raise_py_error()`



## 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

```
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)
```

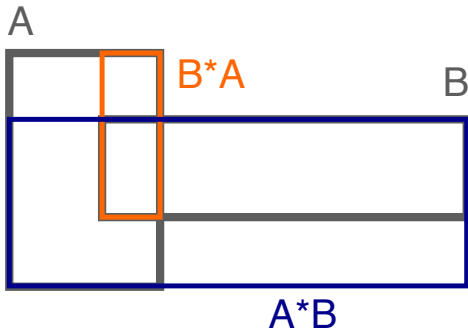




## Special features: Operator Overloading

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

**Example with Rectangles:**



Multiplication of rectangles: Create a new rectangle with the bottom left corner from the first one and the top right corner from the second one



## 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 `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 reinvent the wheel!**



## 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;

    }

}
```