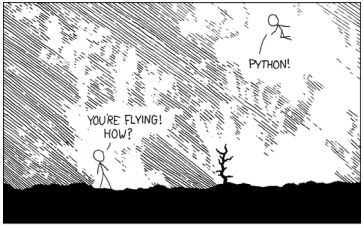
Software-based Speed-up Techniques

Scientific Programming with Python

Christian Elsasser

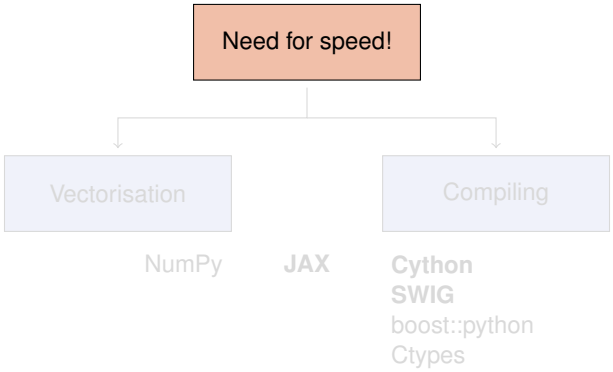
Partially based on a talk by Stéfan van der Walt

Python is nice, but by construction not very fast . . .

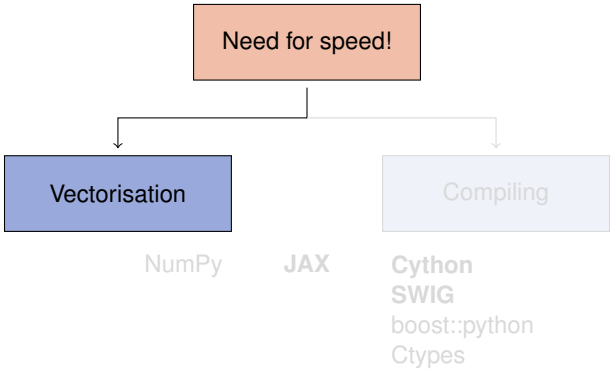


[xkcd]

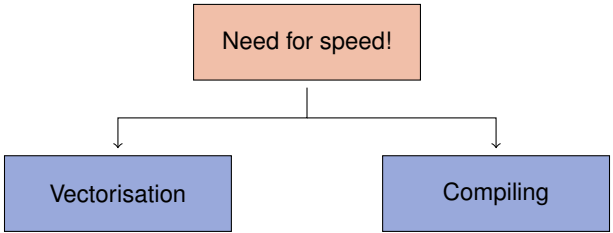
. . . so how can we overcome this issue?



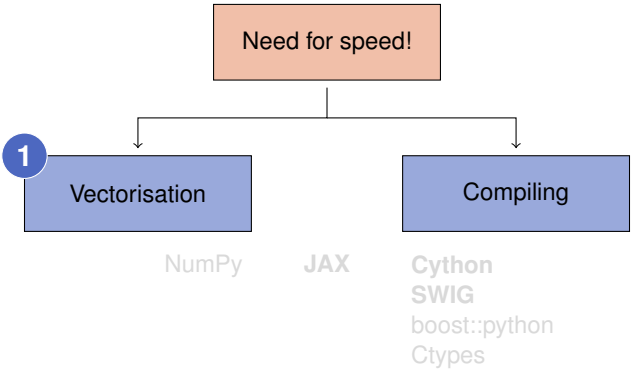
. . . so how can we overcome this issue?



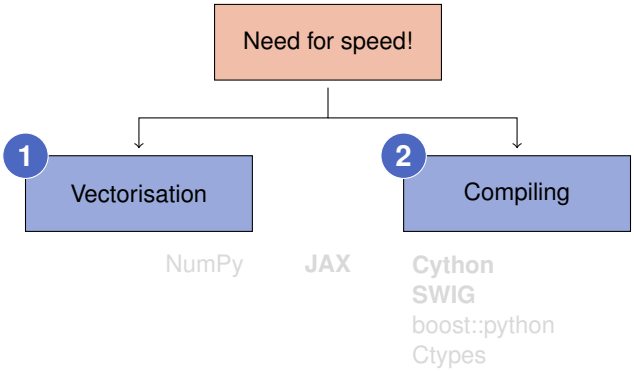
. . . so how can we overcome this issue?



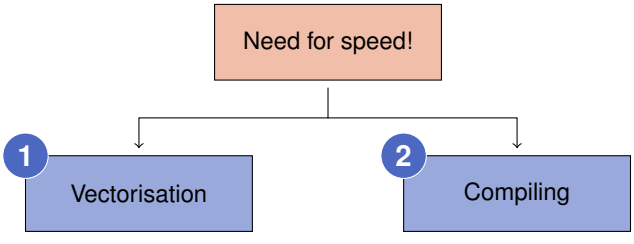
. . . so how can we overcome this issue?



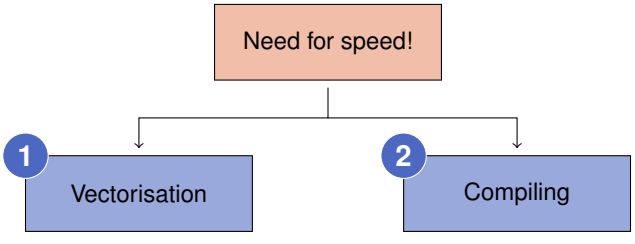
. . . so how can we overcome this issue?



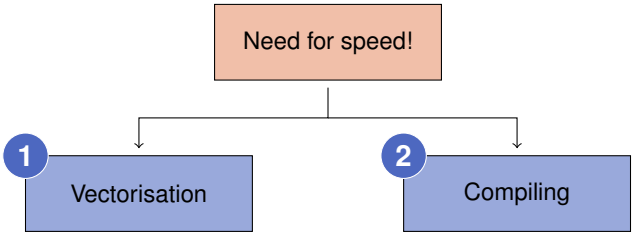
. . . so how can we overcome this issue?



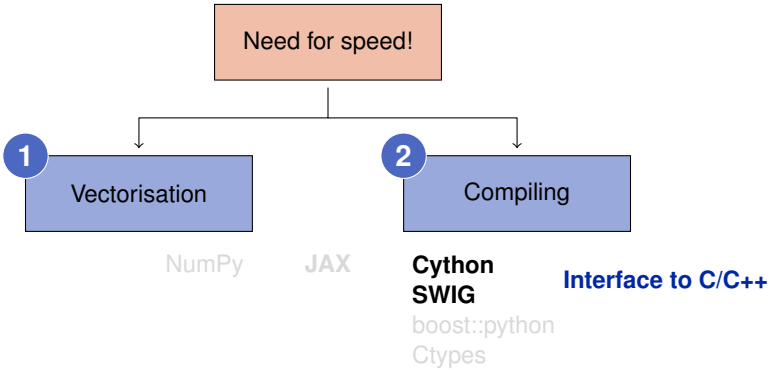
. . . so how can we overcome this issue?



. . . so how can we overcome this issue?



. . . so how can we overcome this issue?



JAX - Accelerator-oriented Array Computation and Program Transformation

by Google

- **Auto-vectorisation:** Translating of code into an optimised version that is better leveraging vectorisation
- **Just-in-time (JIT) compilation:** Compiling of functions “on the fly” to improve their performance
- **Auto-parallelisation:** Automated conversion of code into a code running on multiple CPUs or GPUs
- **Automatic differentiation:** techniques leveraging mathematical rules to algorithmically evaluate derivatives, in particular for nested functions (*i.e.* via the chain rule)

- Same code can be run not just on CPUs, but also on GPUs and TPUs.

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/`)

The sum function . . .

Pure Python

```
>>>>>> a = [1,2,3,4,5,6,7,8]
>>>>>> sum(a)
36
>>>>>> b = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8]
>>>>>> sum(b)
3.6
```

. . . looks in the back like this . . .

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

C++ implementation

```
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

Example 1: Fibonacci series

Fibonacci function - Python

```
def fib(n):

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

Example 1: Fibonacci series

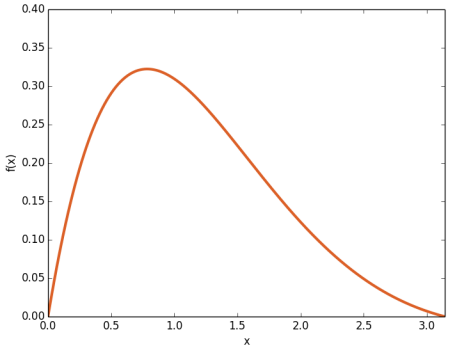
Fibonacci function - Cython

```
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 $\mathcal{O}(100)$

Example 2: Numerical Integration

$$\int_0^{\pi} f(x) = \sin x \cdot e^{-x} \qquad \text{Exact result: } \frac{e^{-\pi}+1}{2} = 0.521607$$



Example 2: Numerical Integration

$$\int_0^{\pi} f(x) = \sin x \cdot e^{-x} \qquad \text{Exact result: } \frac{e^{-\pi}+1}{2} = 0.521607$$

- 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 in the appendix and exercises)
- 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 leveraging NumPy and its vectorisation

Cython can also handle and interact with other features of C/C++

- Cython comes with access to **fundamental C libraries** like (`math`, `stdlib`)
- There is a mapping between Python types and **STL containers** (*e.g.* `std::vector`) (see appendix).
- The same is also true for **exceptions/errors** in Python and C++.

You can find more details in the appendix and in the exercises.

Integration of C Functions in Cython

Starting point: `.c/.h` file for function definition *e.g.* `fast_inv_sqrt`

1. Expose it to **Cython** by declaring the function signature.
2. Integrating it into Cython either via direct usage or by defining a wrapper function.

C function definition in c_func.c

```c
#include <stdio.h>

double fast_inv_sqrt( double number )
  {
        ...
  }
```

Integration of C Functions in Cython

Starting point: `.c/.h` file for function definition *e.g.* `fast_inv_sqrt`

1. Expose it to **Cython** by declaring the function signature.
2. Integrating it into Cython either via direct usage or by defining a wrapper function.

Wrapping the function

```
cdef extern from "c_func.c":
    double fast_inv_sqrt(double number)

def py_fis(number:double) -> double:
    return fast_inv_sqrt(number)

def norm_vector(values:list) -> list:
    length_squared = sum([x**2 for x in values])
    return [x*fast_inv_sqrt(length_squared) for x in values]
```

Compiling Cython Code outside of a notebook

Support via `setuptools` for building and installing Python modules ⇒ applicable for cython

Cython setup script

```
from setuptools import setup
from Cython.Build import cythonize

setup(ext_modules = cythonize([<name of .pxy files>],
                              language_level=3
))
```

Execute: `python setup.py build_ext --inplace`
Creates a `.c/.cpp` file for each `.pyx` file, then compiles it to an executable (in `build` sub-directory) and compiles a `.so` file (or a `.pxd` if you are using Windows)

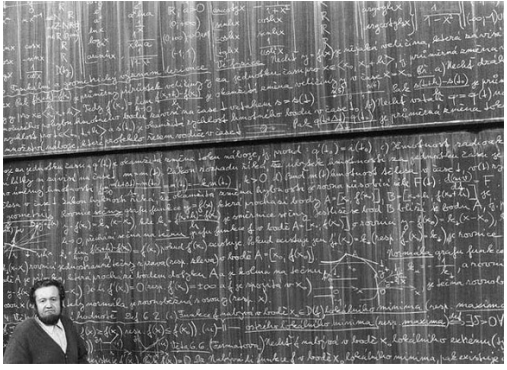
Further options for `cythonize` via help explorable

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

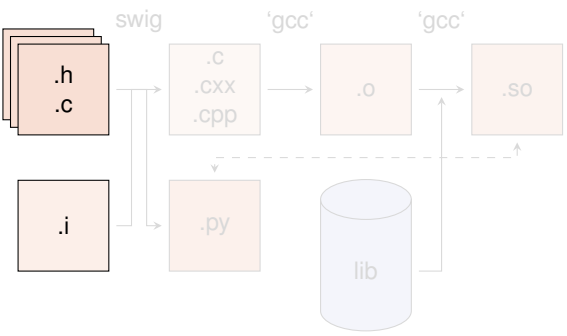


SWIG

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 4.0.2

SWIG – in a Nutshell

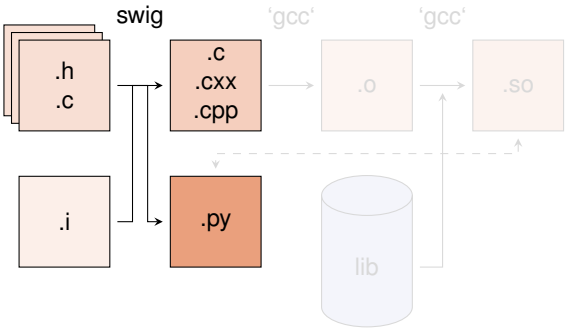


1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Compile shared object (*i.e.* library)

   Step 2 best handed to setuptool (`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

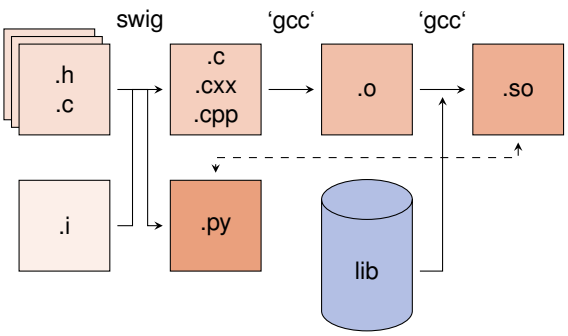


1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Compile shared object (*i.e.* library)

   Step 2 best handed to setuptool (`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

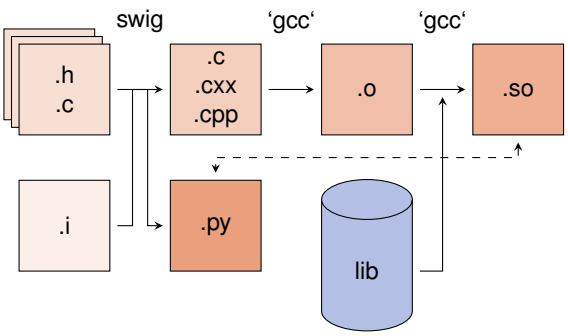


1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Compile shared object (*i.e.* library)

   Step 2 best handed to setuptool (`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

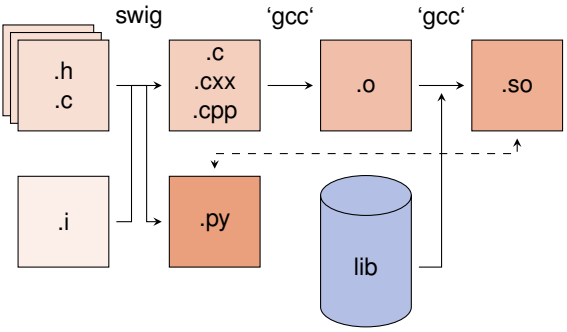


1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Compile shared object (*i.e.* library)

   Step 2 best handed to setuptool (`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



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

1. Create python wrapper and necessary C files
   `swig -c++ -python <name>.i`
2. Compile shared object (*i.e.* library)

   Step 2 best handed to setuptool (`setup.py`)
   `python setup.py build_ext --inplace`

SWIG – The `setup.py` file

setuptools setup script (setup.py)

```
from setuptools import setup, Extension
extension_mod = Extension("_<name>" , # Use _ to distinguish to final module 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 SWIG (default: `_name`
- Language option only needed for C++
- `python setup.py build_ext --inplace`

Summary

- There are several options to improve the speed of your code:
  - **Vectorisation:**
    - NumPy
    - JAX
  - **Compiling:**
    - JIT compiling
    - Cython
    - Write your code in a compiled language and wrap it for Python
  - Some further tools and considerations discussed by Roman in the afternoon
- Wrapping is particularly interesting for existing code allowing to integrate existing functionality in different languages.

Summary

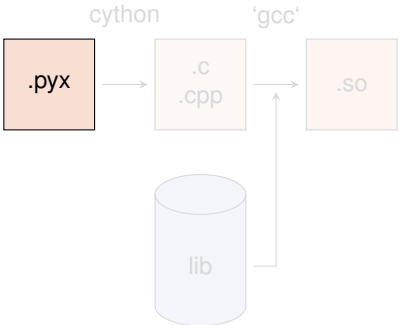
- There are several options to improve the speed of your code:
  - **Vectorisation:**
    - NumPy
    - JAX
  - **Compiling:**
    - JIT compiling
    - Cython
    - Write your code in a compiled language and wrap it for Python
  - Some further tools and considerations discussed by Roman in the afternoon
- Wrapping is particularly interesting for existing code allowing to integrate existing functionality in different languages.

University of Zurich[UZH]

Faculty of Science

Appendix

Compiling Cython Code (The hard way)

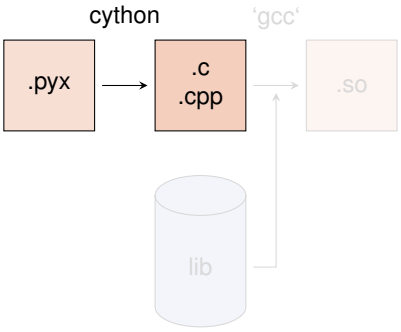


1. Compile Cython code to C/C++ code
   `cython3 -3 <name>.pyx`
2. Compile shared object file (*i.e.* library)
   `gcc [options] -fPIC -O2 -Wall -I<path_to_python_include> -L<path_to_python_library> <name>.c -o <name>.so`
   - If using C++ code, cython needs the option `-+` and `gcc` → `g++`
   - `options`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

   School-Laptops:
   `gcc -shared -fPIC -O2 -Wall -I/usr/include/python3.9/ <name>.c -o <name>.so`

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

Compiling Cython Code (The hard way)

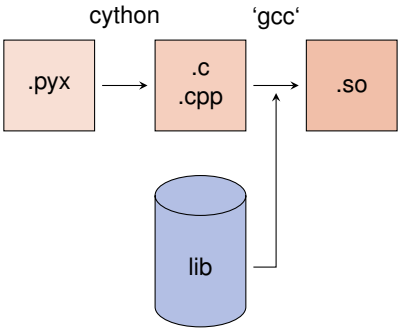


Shared object (`<name>.so`) can be imported into Python with `import <name>`

1. Compile Cython code to C/C++ code
   `cython3 -3 <name>.pyx`
2. Compile shared object file (*i.e.* library)
   ```
   gcc [options] -fPIC -O2 -Wall
   -I<path_to_python_include>
   -L<path_to_python_library>
   <name>.c -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`

   School-Laptops:
   ```
   gcc -shared -fPIC -O2 -Wall
   -I/usr/include/python3.9/
   <name>.c -o <name>.so
   ```

Compiling Cython Code (The hard way)

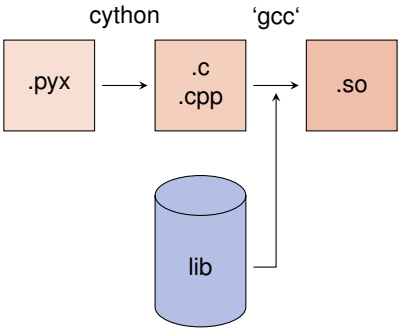


Shared object (`<name>.so`) can be imported into Python with `import <name>`

1. Compile Cython code to C/C++ code
   `cython3 -3 <name>.pyx`
2. Compile shared object file (*i.e.* library)
   `gcc [options] -fPIC -O2 -Wall -I<path_to_python_include> -L<path_to_python_library> <name>.c -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`

   School-Laptops:
   `gcc -shared -fPIC -O2 -Wall -I/usr/include/python3.9/ <name>.c -o <name>.so`

Compiling Cython Code (The hard way)

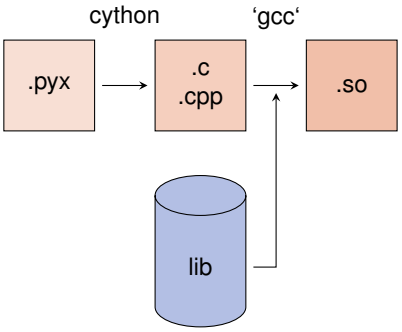


Shared object (`<name>.so`) can be imported into Python with `import <name>`

1. Compile Cython code to C/C++ code
   ```
   cython3 -3 <name>.pyx
   ```
2. Compile shared object file (*i.e.* library)
   ```
   gcc [options] -fPIC -O2 -Wall
   -I<path_to_python_include>
   -L<path_to_python_library>
   <name>.c -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`

   School-Laptops:
   ```
   gcc -shared -fPIC -O2 -Wall
   -I/usr/include/python3.9/
   <name>.c -o <name>.so
   ```

Compiling Cython Code (The hard way)



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

1. Compile Cython code to C/C++ code
   ```
   cython3 -3 <name>.pyx
   ```
2. Compile shared object file (*i.e.* library)
   ```
   gcc [options] -fPIC -O2 -Wall
   -I<path_to_python_include>
   -L<path_to_python_library>
   <name>.c -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`

   School-Laptops:
   ```
   gcc -shared -fPIC -O2 -Wall
   -I/usr/include/python3.9/
   <name>.c -o <name>.so
   ```

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
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 containers (*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 $\Rightarrow$ 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 in Cython - untreated

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

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

Errors in Python

Python Error in Cython - untreated

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

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

Python Error in Cython - treated

```
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

Handling a C Error

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

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`-and-`except` clause!

$\Rightarrow$ But capturing in cython and translating 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

Defining classes in Cython

```
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)
```

Defining Classes in Cython

Let's go back to the integration examples

Adding classes in Python

```
class Poly(Integrand):
        def evaluate(self,double x):
                return x*x-3*x
integrate(Poly(),0.0,2.0,1000)
```

$\Rightarrow$ 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

Exposing C++ classes in 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)
```

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

Wrapping the class for Python

```
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()
        ...
```

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!