Todd Leonhardt
todd.leonhardt@gmail.com

Interfacing C/C++ with Python

Using Cython, SWIG, and CFFI

Code on GitHub at: https://github.com/tleonhardt/Python_Interface_Cpp

Overview

- Talk
  - Learn to leverage the strengths of C/C++ and Python
  - While alleviating the weaknesses
  - Efficiently integrate them to maximize productivity

- Lab Exercises
  - Do these on your own
  - Prerequisite info on GitHub
  - Exercises in GitHub repo on *master* branch
    - Solutions on the *solutions* branch

What will be covered

- Wrapping an existing C/C++ shared library
  - And making calls into lib from Python
  - Using these tools:
    - Cython
    - SWIG
    - CFFI
- Optimizing existing Python code
  - By converting a small amount of critical code to C/C++
  - Using these tools:
    - Cython
- Calling Python code from C/C++ (lab)

What won't be covered

- Other ways of interfacing C/C++ with Python
  - Ways which don't involve wrapping a shared library
- Using Python subprocess to call C/C++ app
  - And using stdin/stdout to communicate
- Using inter-process communication (IPC)
  - Such as sockets or 0MQ
- Embedded Python interpreter in C/C++ app
- Weak asynchronous file-based exchange
- Official Python C-API

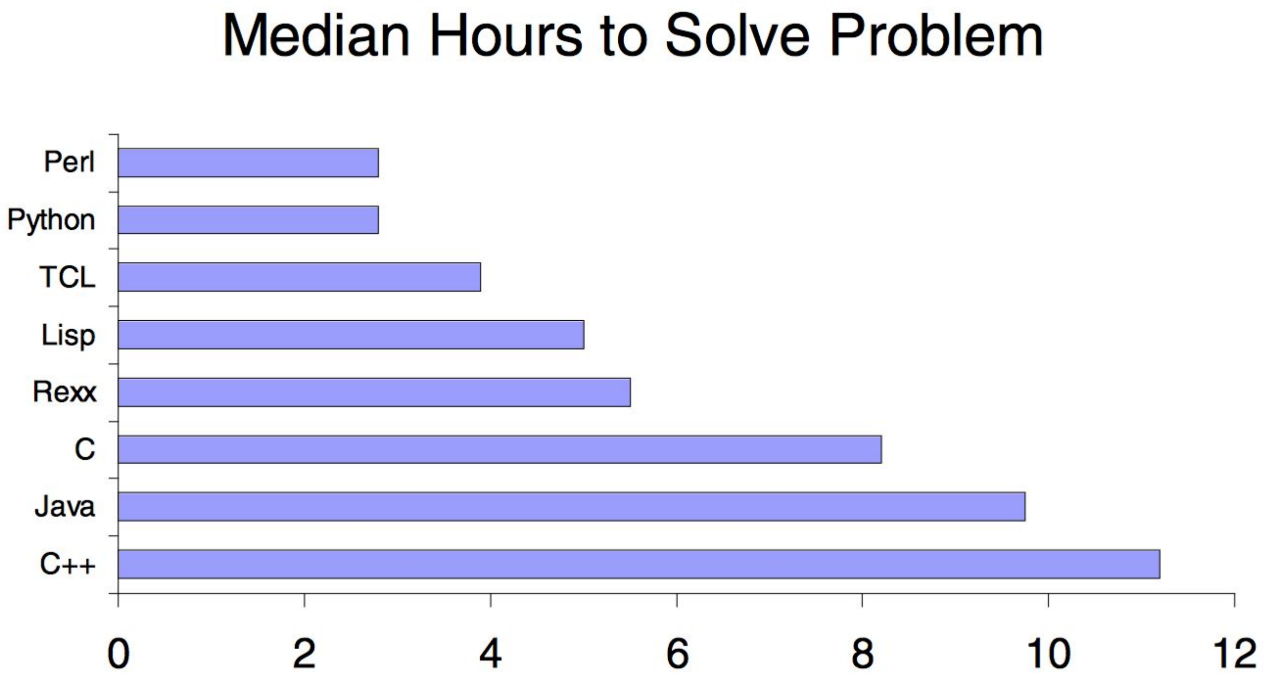
Motivation

- Use C/C++ strengths to offset Python weakness
  - **Fast runtime performance**
  - Availability on all platforms, including mobile
  - Ability for compiler to catch some errors
  - Reliable performance boost using threads
- Use Python strengths to offset C/C++ weakness
  - **Rapid development**
  - Amazing standard and 3rd party libraries
  - Zero compile time and great testing tools
  - Easier to write code that is less vulnerable to exploit

Typical Use Cases

- Leverage existing C/C++ in Python
  - For Python’s rapid development and better libraries
  - For Python’s ease of creating a UI
  - For Python’s testing capabilities
- Use Python within C/C++
  - For access to Python’s better libraries
    - Don’t re-invent the wheel / save time & budget
  - For consistency within a multi-language app (logging)
- Optimize pre-existing Python
  - Make critical parts run faster (after profiling, of course)

Productivity - C/C++ vs Python



Data compiled from studies by Prechelt [1] and Garret [2] of a particular string processing problem. Connelly Barnes, public domain 2006

Performance - C/C++ vs Python

Python 3 programs versus C gcc

all other Python 3 programs & measurements

by benchmark task performance

pidigits

source	secs	mem	gz	cpu	cpu load
Python 3	3.41	9,992	382	3.40	1% 2% 100% 1%
C gcc	1.73	1,992	448	1.73	1% 100% 1% 0%

reverse-complement

source	secs	mem	gz	cpu	cpu load
Python 3	2.93	265,636	800	4.28	80% 46% 21% 2%
C gcc	0.42	145,900	812	0.57	0% 26% 20% 100%

regex-redux

source	secs	mem	gz	cpu	cpu load
Python 3	14.87	433,868	486	28.02	32% 45% 84% 29%
C gcc	1.89	155,412	1230	4.28	100% 47% 42% 41%

Python 3 programs versus C++ g++

all other Python 3 programs & measurements

by benchmark task performance

pidigits

source	secs	mem	gz	cpu	cpu load
Python 3	3.41	9,992	382	3.40	1% 2% 100% 1%
C++ g++	1.89	3,740	508	1.89	2% 99% 0% 2%

regex-redux

source	secs	mem	gz	cpu	cpu load
Python 3	14.87	433,868	486	28.02	32% 45% 84% 29%
C++ g++	6.02	218,304	848	8.57	15% 100% 14% 15%

reverse-complement

source	secs	mem	gz	cpu	cpu load
Python 3	2.93	265,636	800	4.28	80% 46% 21% 2%
C++ g++	0.59	217,564	2275	0.84	26% 78% 12% 34%

Fibonacci Number Example Code

Fibonacci Number Example

- Use same example code to demo all tools
  - Apples-to-apples comparison
  - Anyone with any CS background should be familiar
- Fibonacci numbers are numbers in sequence:
  - 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …
- Recurrence relation for F_n
  - $F_n = F_{n-1} + F_{n-2}$
- Seed values for our example:
  - $F_{-1} = 1$, $F_0 = 1$
  - Thus, $F_1 = 2$, $F_2 = 3$, etc.

Fibonacci Number C Code

```c
int compute_fibonacci(int n)
{
    int temp;
    int a = 1;
    int b = 1;
    for (int x=0; x<n; x++)
    {
        temp = a;
        a += b;
        b = temp;
    }
    return a;
}
```

Fibonacci Number Python Code

```python
def compute_fibonacci(n):
    """
    Computes fibonacci sequence
    """
    a = 1
    b = 1
    intermediate = 0
    for x in range(n):
        intermediate = a
        a = a + b
        b = intermediate
    return a
```

C Foreign Function Interface for Python

CFFI

http://cffi.readthedocs.io

What is CFFI?

- Similar to built-in *ctypes*, but more direct
- Let’s you interact with C code from Python
  - Based on C-like declarations
    - That you can copy-and paste from headers
- No wrapper required
  - Makes it very quick and easy to use
- Fully compatible with PyPy JIT

CFFI Pros and Cons

- Pros
  - Self-contained - all code is inline within Python
    - No need to create any external files
  - Quickest way to call C functions from Python
    - Assuming you already have a dynamic C library
- Cons
  - No C++ support
  - Performance much worse than SWIG or Cython
  - ABI of the C API is generally not stable
    - May break between versions of Python
      - Need specific version of CFFI to “match” version of Python

How does CFFI Work?

- Call C functions from C libraries <u>directly</u>
- Make calls at binary level using C ABI
- You specify function prototypes inline
  - It will crash if you do this incorrectly
- CFFI safely converts datatypes for you
  - Python ← → C
  - C ← → Python
  - This can be somewhat expensive performance-wise

Using CFFI

- 4 step process:

1. Instantiate the main top-level CFFI class
    a. `ffi = cffi.FFI()`
2. Declare function prototype
    a. `ffi.cdef(‘int compute_fibonacci(int n);’)`
3. Load the dynamic library
    a. `libfib = ffi.dlopen(‘./libfibonacci.so’)`
4. Call a function in the library
    a. `fib_20 = libfib.compute_fibonacci(20)`

CFFI Fibonacci Code

```
import cffi
import fib_python

if __name__ == '__main__':
    # The main top-level CFFI class that you instantiate once
    ffi = cffi.FFI()

    # Parses the given C source.  This registers all declared functions.
    ffi.cdef('int compute_fibonacci(int n);')

    # Load and return a dynamic library.  The standard C library can be loaded by passing None.
    libfib = ffi.dlopen('./libfibonacci.so')

    n = 20
    fib_py = fib_python.compute_fibonacci(n)
    fib_cffi = libfib.compute_fibonacci(n)
    if fib_py != fib_cffi:
        raise (ValueError(fib_cffi))
```

Simplified Wrapper and Interface Generator

SWIG

http://www.swig.org

What is SWIG?

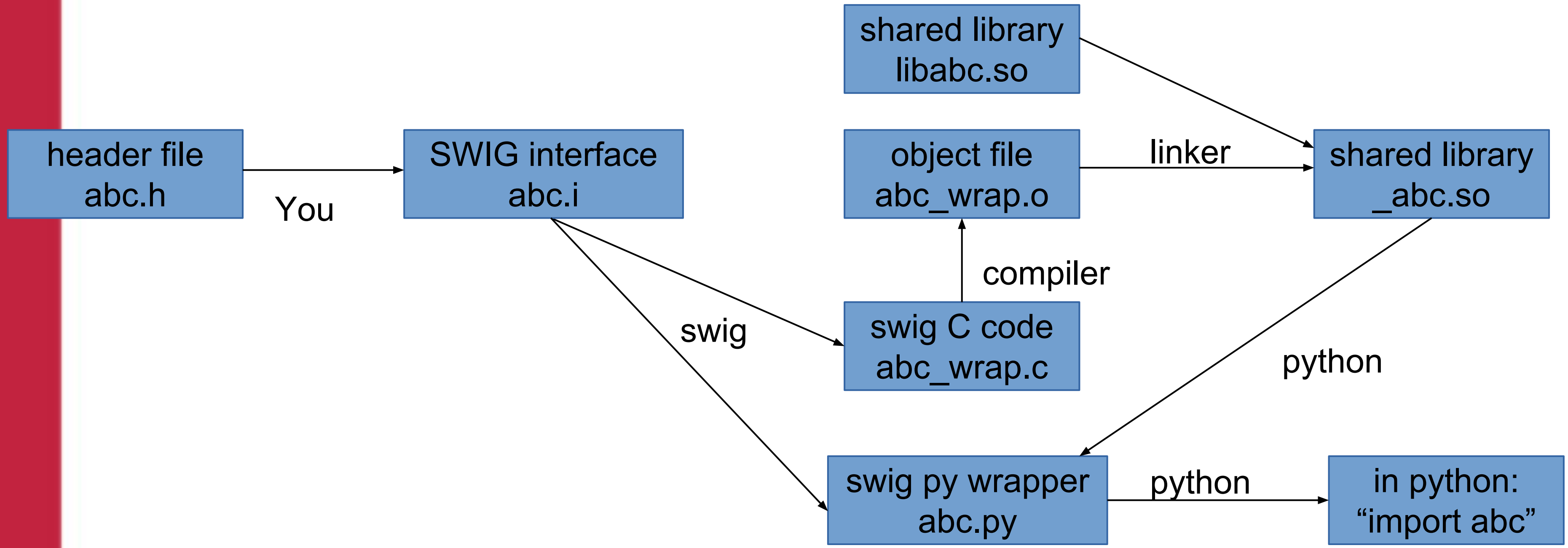
- Tool auto-generates wrappers for C/C++ code
- Can target about 20 programming languages
  - Python, Java, C#, PHP, Javascript, Perl, Ruby, R, Go ...
- You create *.i interface files
  - Tell it what to wrap and give it a few hints
  - You only need to wrap public interface
  - And only things you want to actually use
- SWIG does the rest
  - Generates a Python Extension Module
  - You can import this directly in Python

SWIG Pros and Cons

- Pros
  - Easy and quick to use if you already have C/C++ code
  - Targets numerous languages
  - Good C++ support
  - True cross-language polymorphism available
- Cons
  - Doesn’t support nested classes/structs
  - Working with arrays can be painful
    - C++ STL strings and containers are easier
  - It’s semi-automatic, so you don’t have full control
  - Performance < Cython (but > CFFI)

How does SWIG Work?

- Partially automates process of wrapping C/C++
- You generate *.i interface file, then run SWIG



SWIG Interface File - Fibonacci

```
/* fibonacci.i */
 %module fibonacci
 %{
 /* Includes the header in the wrapper code */
 #include "fibonacci.h"
 %}

/* Parse the header file to generate wrappers */
 %include "fibonacci.h"
```

Building SWIG Wrapper - setup.py

```python
# coding=utf-8
from distutils.core import setup, Extension


name = "fibonacci"  # name of the module
version = "1.0"  # the module's version number


setup(name=name, version=version,
      # distutils detects .i files and compiles them automatically
      ext_modules=[Extension(name='_{}'.format(name),  # SWIG requires _ as a prefix for module name
                             sources=["fibonacci.i", "fibonacci.c"],
                             include_dirs=[],
                             extra_compile_args=["-std=c11"],
                             swig_opts=[])
                   ])
```

SWIG Fibonacci Code

```python
""" Python wrapper to time the SWIG wrapper for computing the nth fibonacci number
in a non-recursive fashion and compare it to the pure Python implementation.
"""
import fibonacci
import fib_python

if __name__ == '__main__':
    n = 20

    fib_py = fib_python.compute_fibonacci(n)
    fib_swig = fibonacci.compute_fibonacci(n)
    if fib_py != fib_swig:
        raise (ValueError(fib_swig))
```

C-Extensions for Python

Cython

http://cython.org

What is Cython?

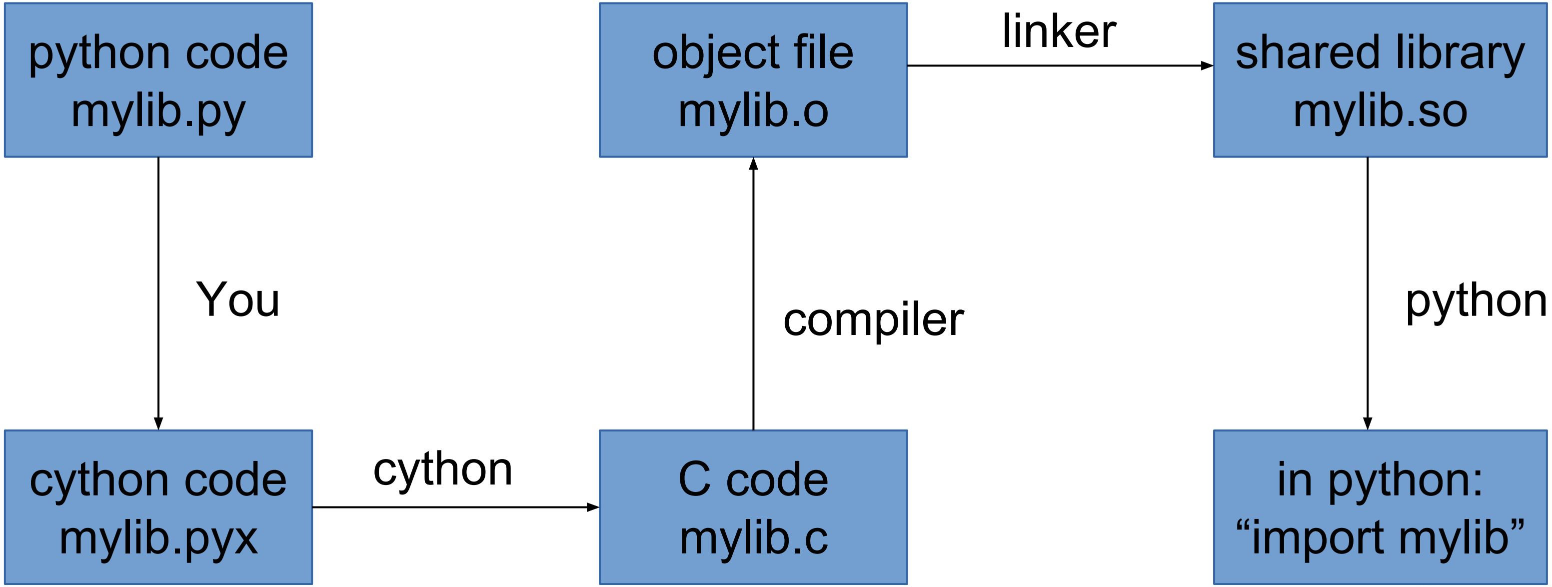
- An optimising static compiler
  - For both Python and the extended Cython language
- A creole programming language
  - Extends Python with optional static type declarations
- Can port performance-critical code to C/C++
  - In a mostly automated fashion
  - Just define static types for arguments and variables
- Can wrap C/C++ in a high performance way
  - But process is not automated like SWIG
- Can embedded Python interpreter in C/C++

Cython Pros and Cons

- Pros
  - Easy and quick way to optimize existing Python code
    - Quickly build Python prototype, optimize as needed
  - Performance is by far the best of tools presented
    - Cython code often runs faster than hand-written C
  - Good C++ support
  - Good debugging and profiling support
- Cons
  - Large learning curve
  - Laborious for wrapping large libraries
    - Not as automated as SWIG

Optimizing Python with Cython

- Allows you to write code which looks like Python
  - With optional static type declarations added
  - Which automatically compiles to a native binary library



Python → Cython Conversion

1. Start with Python module you want to speed up (mylib.py)
2. Rename this file to have a *.pyx file extension (mylib.pyx)
3. Edit code to add optional static C type declarations
4. Invoke cython to create a C language file (mylib.c)
5. Compiler – C file is compiled into an object file (mylib.o)
6. linker – object file is linked into a shared library (mylib.so)
7. Python – at this point the shared library can be imported from a Python script ("import mylib")

Steps 4, 5, and 6 are typically automated and done together.

Ways of Compiling Cython

- Easiest: Jupyter Notebook
  - Interactive browser-based interface
  - Which allows you to evaluate cells of source code
- Easy: `cythonize` command-line script
  - Provided as part of Cython
  - Wrapper around all compilations steps (4, 5, 6)
  - `cythonize -b -a mylib.pyx`
    - Generates mylib.c and compiles that to mylib.so
- Medium: setup.py (**recommended**)
  - Use Python packaging tools to compile Cython
  - Best for distributing code to others

Fibonacci Python Code - *.py

```
def compute_fibonacci(n):
    """
    Computes fibonacci sequence
    """
    a = 1
    b = 1
    intermediate = 0
    for x in range(n):
        intermediate = a
        a = a + b
        b = intermediate
    return a
```

Fibonacci Cython Code - *.pyx

```
""" Cython implementation for computing the nth fibonacci number in a
non-recursive fashion.
"""


cpdef int compute_fibonacci_cython(int n):
    """ Compute the nth fibonacci number in a non-recursive fashion.
    """
    cdef int a, b, intermediate, x
    a, b = 1, 1
    for x in range(n):
        intermediate = a
        a += b
        b = intermediate
    return a
```

Building Cython - setup.py

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

Cython.Compiler.Options.annotate = True

setup(
    name="fib",
    ext_modules=cythonize('fib.pyx', compiler_directives={'embedsignature': True}),
)
```

Using Cython Fibonacci Code

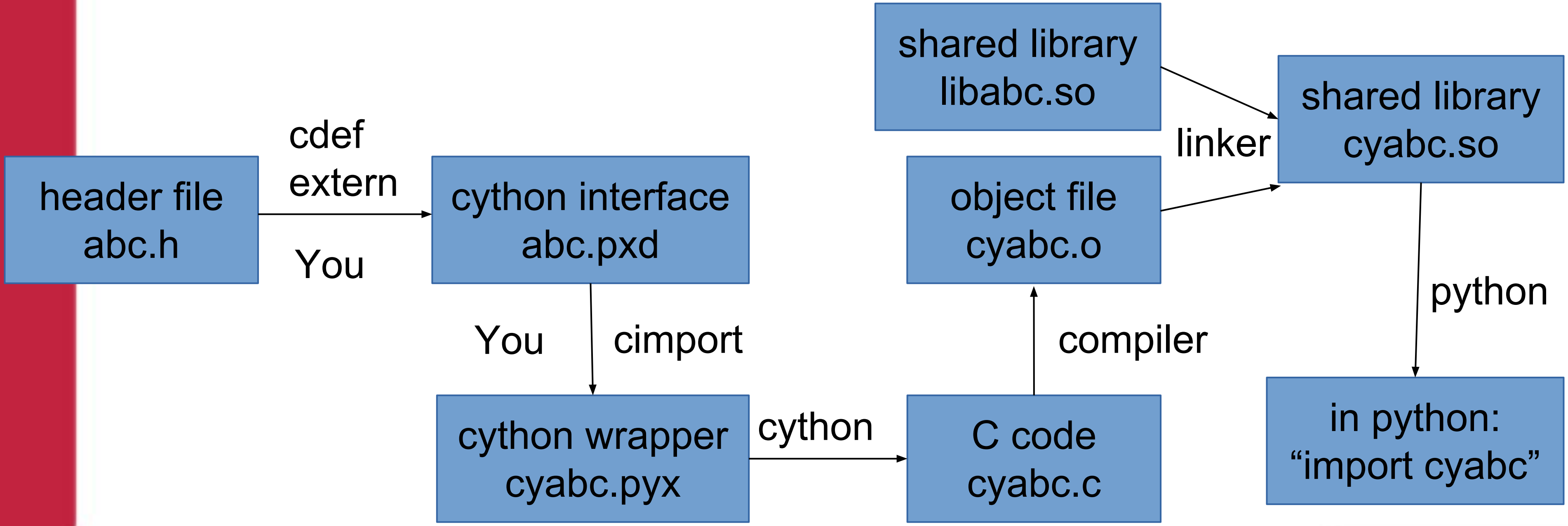
```
""" Python wrapper to time the Cython implementation for computing the nth fibonacci number
in a non-recursive fashion.
"""
from fib_python import compute_fibonacci
from fib import compute_fibonacci_cython

if __name__ == '__main__':
    n = 20

    fib_py = compute_fibonacci(n)
    fib_cy = compute_fibonacci_cython(n)
    if fib_py != fib_cy:
        raise(ValueError(fib_cy))
```

Wrap Existing C/C++ with Cython

- More complicated than optimizing Python with Cython
- Also more complicated than wrapping with SWIG
- But a very high performance way to wrap C/C++ in Python



C Header and Cython Interface

fibonacci.h

```c
#pragma once
extern int compute_fibonacci(int n);
```

cfib.pxd

```cython
"""
Cython declaration file specifying what function(s) we are using from which external C header.
"""
cdef extern from "fibonacci.h":
    int compute_fibonacci(int n)
```

Cython Wrapper - cyfib.pyx

```
# distutils: libraries = "fibonacci"
# distutils: library_dirs = "."
cimport cfib

cpdef int compute_fibonacci_wrapper(int n):
    return cfib.compute_fibonacci(n)
```

Using Cython Wrapper

```python
""" Python wrapper to time the Cython implementation for computing the nth fibonacci number
in a non-recursive fashion.
"""
from fib_python import compute_fibonacci
from cyfib import compute_fibonacci_wrapper

if __name__ == '__main__':
    n = 20

    fib_py = compute_fibonacci(n)
    fib_cy = compute_fibonacci_wrapper(n)
    if fib_py != fib_cy:
        raise(ValueError(fib_cy))
```

Building Cython Wrapper

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

setup(
    name="cyfib",
    ext_modules=cythonize('cyfib.pyx', compiler_directives={'embedsignature': True}),
)
```

Final Thoughts

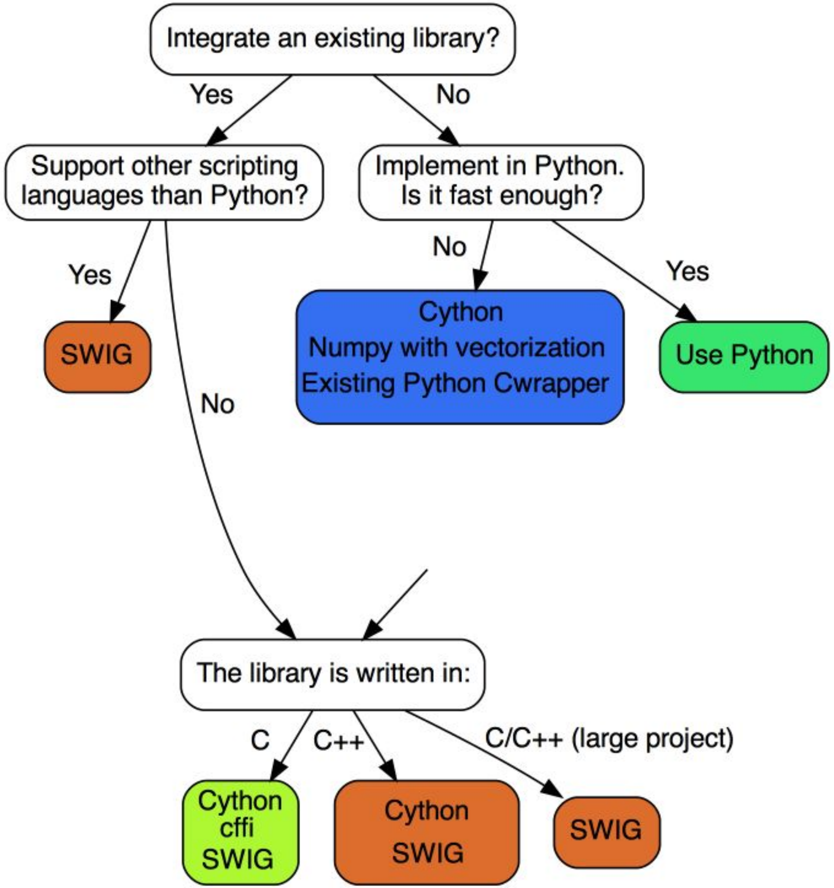
Fibonacci Performance

One metric for evaluating these tools is performance. Here is a table which shows speedup factor relative to the pure Python implementation of the Fibonacci code. So a speedup of 2 means that code ran twice as fast as the pure Python version.

Tool	Speedup
Cython (optimized)	27
Cython (wrapper)	25
SWIG	14
pybind11	10
CFFI	7
Python	1

These numbers were measured on a 2013 15" Mac Book Pro using Python 3.6 via Anaconda distro with the latest versions of all tools installed using the conda package manager.

Which tool should I use?



- Like most things in engineering
    - There are tradeoffs …
        - So it depends

Lab Exercises (online)

- Goals
  - Gain familiarity with how to use SWIG and Cython
  - Cover a few scenarios that would be useful in real world
  - Learn enough so you know how/where to learn more
- Tools Needed
  - Python 3.4 or newer (recommend Anaconda distro)
  - C and C++ compiler toolchains
  - SWIG and Cython (Cython comes with Anaconda)
  - Setup instructions available on GitHub at:
    - https://github.com/tleonhardt/Python_Interface_Cpp

Where to learn more

- Cython
  - Main Site: http://cython.org
  - Documentation: http://docs.cython.org
  - 4 hour training video from SciPy 2015 with code
- SWIG
  - Main Site: http://www.swig.org
  - Documentation: http://www.swig.org/Doc3.0
  - 40 minute training video from Univ. Oslo with code
- CFFI
  - Main Site & Docs: http://cffi.readthedocs.io
  - 1 hour training video from PyCon

Additional Content

C, C++, and Python Languages

All are general-purpose imperative cross-platform computer programming languages in wide use

Attribute	**C**	**C++**	**Python**
Paradigm	Structured	Object-oriented	Multi
Type System	Static	Static	Dynamic
Code Execution	Compiled	Compiled	Interpreted
Memory Management	Manual	Manual (mostly)	Automatic
Standard Library	Minimal / Horrible	Medium / OK	Large / Amazing

C/C++ Strengths

- **Very fast code execution speed**
  - By design code maps efficiently to machine instructions
  - Compiled in advance, so that price paid up front
- Compiler can catch many common mistakes
  - Static type system prevents many unintended ops
  - You need to enable warnings and pay attention
- Ultra-portable
  - C compilers available on ANY platform
  - From supercomputers to deeply embedded systems

C/C++ Weaknesses

- Security
  - Easy to write insecure and vulnerable code
  - Lacks automatic checks for memory boundaries, etc.
- **Slow development speed**
  - Need to write more code to achieve same end result
    - Manual memory management and static types
    - Poor standard library
- Slow compile time
  - Trade-off for runtime performance
  - Problem gets worse as application size increases

Python Strengths

- Lets you write code more quickly
  - **Solve problem in Python in ⅓ the time as C/C++**
  - Write ⅓ the lines of code to achieve same result
  - Developers still write same number of lines per hour
  - Save time and $ by using Python, when it makes sense
- Easy
  - Easy to learn, read, use, and maintain
- Security
  - Automatic bounds checking, safe string types, etc.
  - No integer overflow due to automatic type promotion
  - 1st-class Exceptions

Python Weaknesses

- Runtime performance can be slow
  - Because it is an interpreted language
  - **Typically [2 to 150 times slower](#) than compiled C**
- Absence from mobile
  - Python available on Windows, Mac OS X, and Linux
  - But not generally available on Android or iOS
- Dynamic typing is a double-edge sword
  - Python requires more unit testing and/or static analysis
  - Has errors that only show up at runtime (no compiler)
- Global Interpreter Lock (GIL)
  - Limits benefit of CPU-bound multithreaded