(PRACTICAL) COMPUTATIONAL PHYSICS

Physics 551
Lecture 12

NOTATION

Extra Reading

Optional Exercise

Recommended

- This lecture slides for this course will attempt to use a uniform notation throughout. A normal paragraph looks like this.

✒ *Italicized paragraphs with pen bullets will indicate definitions, with the defined word or phrase shown in* ***Small-Caps****.*

✏ Pencil bullets will indicate the introduction of **new notation**.

☞ Pointing hand bullets indicate important points that might otherwise be overlooked.

ANNOUNCEMENTS

ANNOUNCEMENTS

- To **clone** this week's **Python demonstration materials** please invoke

```
$git clone https://github.com/hughdickinson/CompPhysL12Python.git
    /home/computationalphysics/Documents/python/lecture12
```

- A new ***Python*** package called **Cython** is required. To install it please invoke

```
$sudo pip install cython
```

☞The required **password** is the same as the Ubuntu login password.

☞You can also find these commands on the Blackboard Learn website.

SCIENTIFIC COMPUTATION IN PYTHON USING SCIPY

`scipy.io`

https://docs.scipy.org/doc/scipy-0.15.1/reference/tutorial/io.html

IO
PACKAGE OVERVIEW

- Provides functions that enable data to be **loaded** from several **proprietary file formats**.
- **Supported formats** include:
  - MATLAB files
  - IDL files
  - Unformatted FORTRAN output files.
  - Matrix Market files.
- **Writing** is also supported for **some** formats.

DEMONSTRATION

SciPy Examples

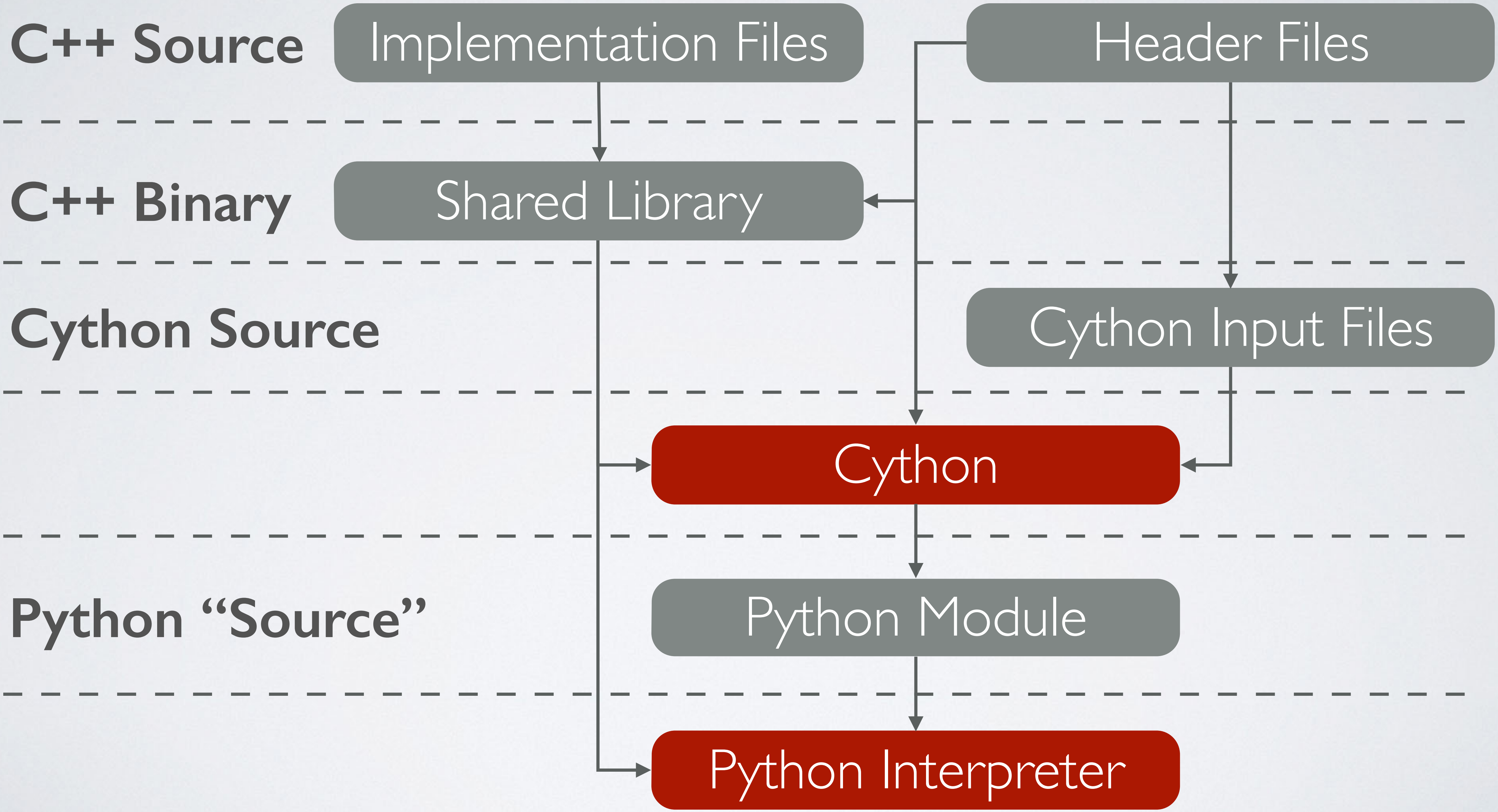
4. Handling Special File Formats

***If necessary*, clone the Lecture 11 Python demonstration material from GitHub:**

```
$ git clone https://github.com/hughdickinson/CompPhysL11Python.git /home/computationalphysics/Documents/python/lecture11
```

UNIFYING C++ AND PYTHON

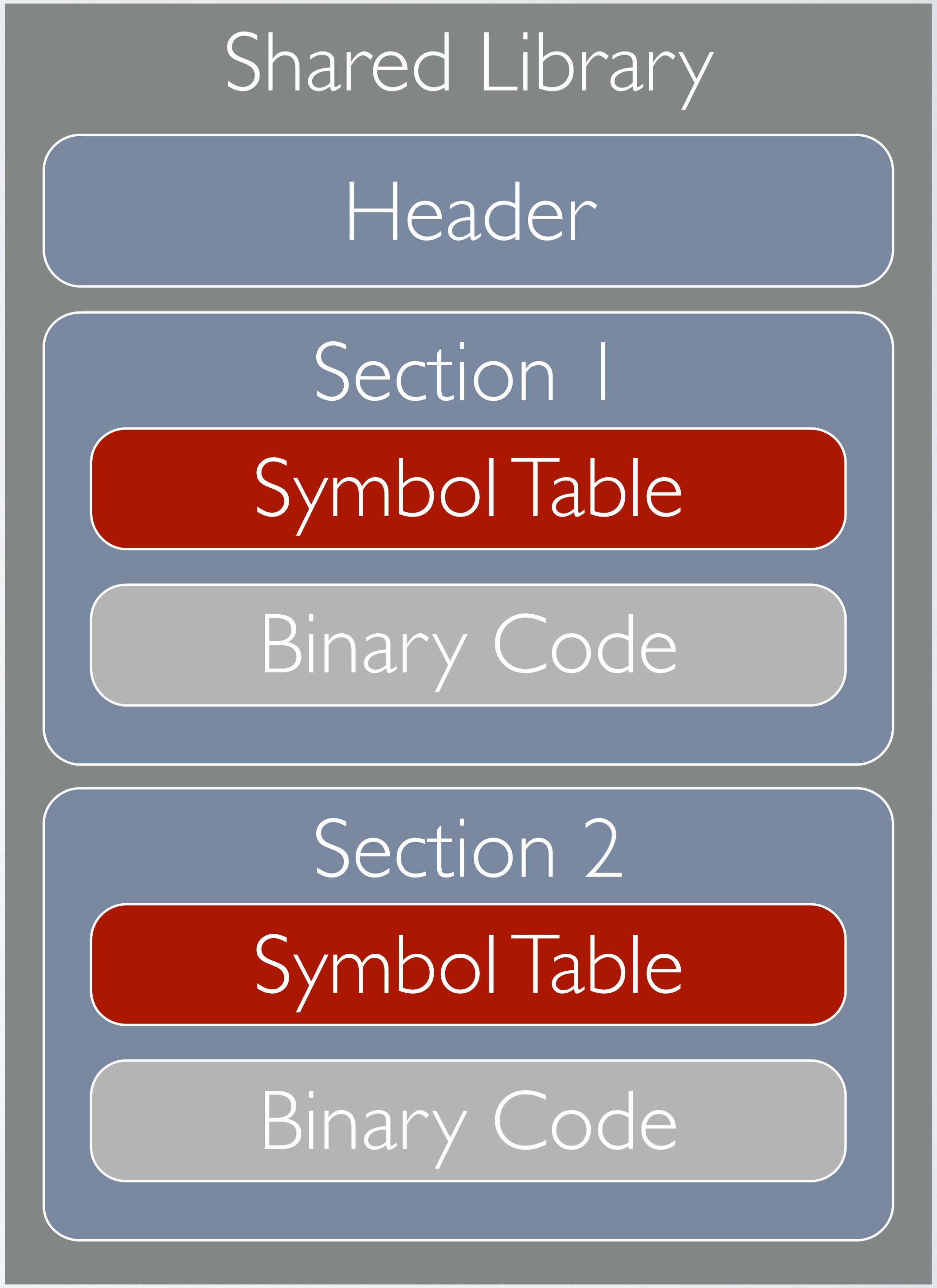
OVERVIEW


C++ Source
Implementation Files
Header Files
C++ Binary
Shared Library
Cython Source
Cython Input Files
Cython
Python "Source"
Python Module
Python Interpreter

C++ SHARED LIBRARY INTERNALS

SHARED LIBRARY INTERNALS

- Shared library files are divided into a ***header*** and one or more ***sections***.
- Each ***section*** contains a ***symbol table*** and a block of executable ***binary code***.
- The ***symbol tables*** list the **definitions** of programatic entities (functions, classes etc.) that the library provides.



SHARED LIBRARY INTERNALS

- The `nm` utility can be used to inspect the symbol tables and thereby determine what functionality a particular shared library provides.
- Consider the example of the main GSL shared library *`libgsl.so`*. Invoking

```
$ nm /usr/lib/libgsl.so
```

  generates a long list of the **symbols** that correspond with the numerous functions that are provided by the GSL.

☞ Symbol names are **character strings** that may **only** comprise alphanumeric characters and underscores.

NAME MANGLING

- The GSL framework is written using the C programming language. Unlike C++, C does **not** allow function overloading.
- As a consequence, the symbol names listed in the `libgsl.so` symbol table **closely resemble** the entity names that appear in the corresponding source code.
- Try invoking

```
$ nm /usr/lib/libgsl.so | grep "gsl_poly"
```

- This should generate a list of familiar-looking symbol names corresponding to the GSL polynomial solving functions.

NAME MANGLING

- Shared libraries that are generated using **C++** source code must contain **symbol names** to represent **overloaded** methods and functions.
- To achieve this, a symbol name is generated that encodes the **identifier** and **parameter types** for each entity.

✑ *This name generation process is a compiler-dependent operation and is called **NAME MANGLING**.*

- For example, `StatsCalculator::computeSum()` is mangled as `__ZN15StatsCalculator10computeSumEv` by `clang++`.

NAME MANGLING

- The `c++filt` utility can be used to invert the name mangling process. Invoking

```
$ c++filt __ZN15StatsCalculatorD2Ev
```

returns "`StatsCalculator::~StatsCalculator()`".

- Piping the output of `nm` to `c++filt` utility will "unmangle" all mangled symbol names in a library's symbol table.

```
$ nm libStatsCalculator.so | c++filt
```

SPECIFYING C LINKAGE

- *It is possible to specify that the names of particular functions in C++ source code should have **C Linkage**. Name mangling is **not** applied to the names of symbols with C linkage.*
- To achieve this, prepend the function declaration or definitions with the `extern "C"` token pair.
- Multiple functions can be excluded from the name mangling process by enclosing their declarations with a code block prepended by the `extern "C"` token pair i.e.

```
extern "C" { /* function declarations */ }
```

CYTHON

CYTHON OVERVIEW

- ✑ ***Cython*** *is a set of tools that streamline the process of generating* ***Python modules*** *using* ***source code*** *and* ***shared libraries*** *that are written in* ***C****.*
- With a little more effort, ***Cython*** can also operate on **source code** and **shared libraries** that are written in **C++**.
- ***Cython*** defines a **Python-like language** that includes several **new keywords**.
- ☞ The `cdef` keyword that is defined by ***Cython*** enables variable **types** to be specified for ***Python*** variables.

CYTHON REQUIREMENTS FOR USE WITH C++

- The basic ***Cython*** utility is designed to work with libraries that are implemented using the **C** language.
- ***Cython*** expects that the names of symbols in the shared libraries that it exposes in ***Python*** are **not** mangled.
- To use ***Cython*** in conjunction with **C++** source code it is necessary to define an **interface** of functions with **C linkage**.

☞ A C or C++ **header file** containing **declarations** of the functions that define this interface is a **required input** for ***Cython***.

CYTHON REQUIREMENTS FOR USE WITH C++

- The functions defined by the C linkage interface must be **implemented** and used to build a **shared library**.

☞ The function implementations **may include C++ code**.

☞ ***Cython*** requires **three** additional input files when it operates.

- A ***definition*** file with the suffix ***.pxd*** and written in the ***Cython* language**.
- An ***implementation*** file with the suffix ***.pyx*** and written in the ***Cython* language**.
- A build configuration file with the suffix ***.py*** written in ***Python***.

CYTHON PXD FILES

(C → CYTHON)

- The definition (*.pxd*) file contains declarations of the interface functions written in the ***Cython* language**.
- The definitions should be enclosed within a ***Python*-like code block** with the opening clause

```
cdef extern from "HeaderFile.h":
```

  which specifies that the interface functions declared in *HeaderFile.h* are defined in a separate shared library
- Typically, ***Cython*** function declarations have **identical syntax** to their counterparts in the C++ header file.

CYTHON PYX FILES

(CYTHON → PYTHON)

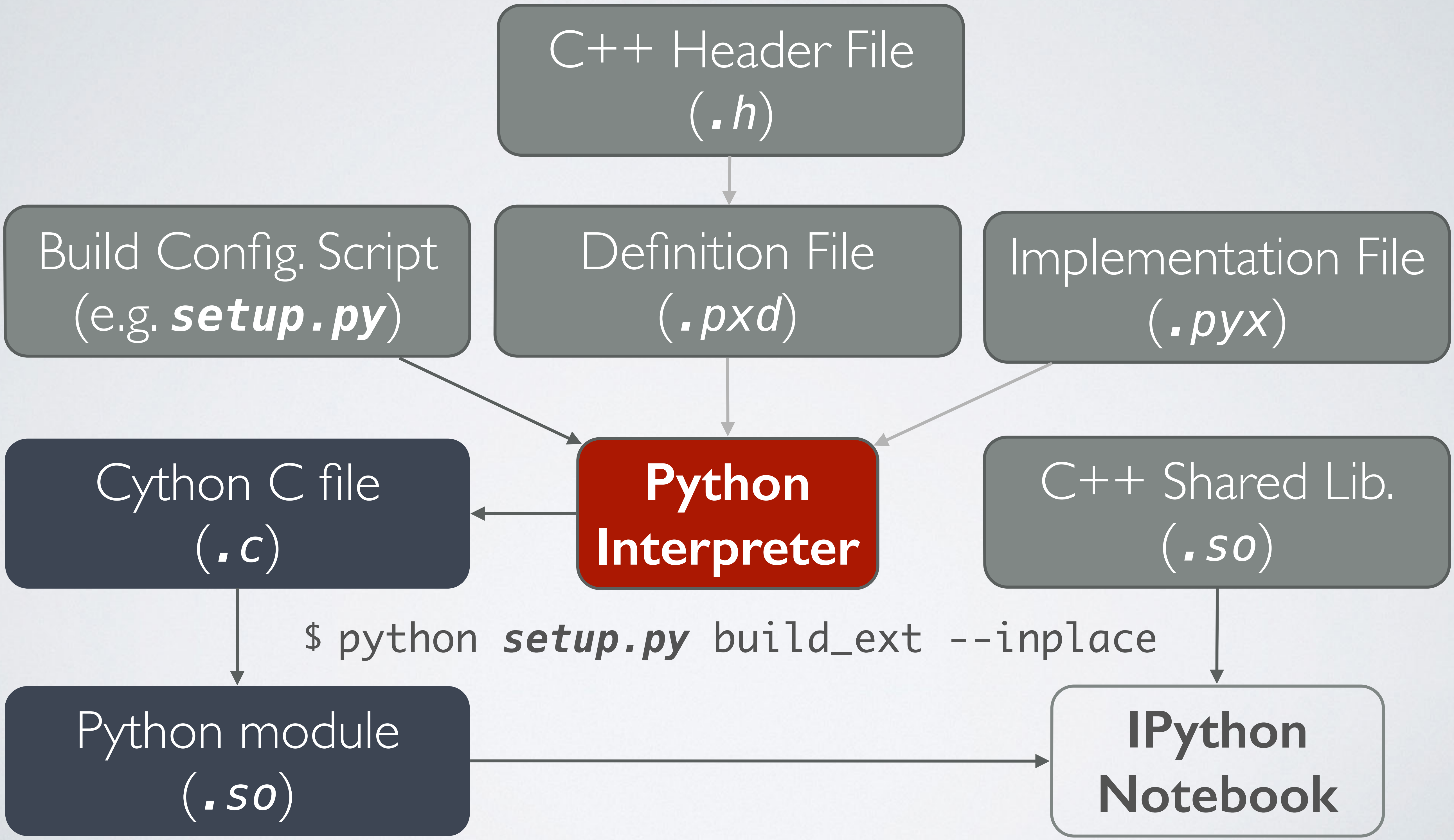
- The implementation (`.pyx`) file contains implementations of functions, and classes written in the ***Cython* language**.
- ***Cython*** entities that are defined using the `cdef` keyword will **not be included** in the ***Python*** module that is ultimately generated by ***Cython***.
- ***Cython*** `.pyx` files may also use the ***Cython***-specific `cpdef` keyword or the standard ***Python*** `def` keyword to declare methods and functions that **will** be added to the module.

CYTHON BUILD PROCESS
(PROVIDED AS A PYTHON SCRIPT)

- The `Cython.Build.cythonize()` function is the core of the ***Cython*** build system, which generates appropriately initialized instances of the `distutils.extension.Extension` class.
- Generation of the ***Python*** module is delegated to the `distutils.core.setup(…)` function, which uses an **initialized** `Extension` instance for **configuration**.

☞ The `setup()` function generates a file written in **C** containing boilerplate C→***Python*** "*bridging*" code and uses that code to build a **shared library** that acts as a ***Python*** module.

CYTHON OVERVIEW



DEMONSTRATION

Merging C++ and Python using Python

Clone the Lecture 12 Python demonstration material from GitHub:

```
$git clone https://github.com/hughdickinson/CompPhysL12Python.git /home/computationalphysics/Documents/python/lecture12
```

LECTURE 12 SUMMARY

- After reviewing the material from this lecture (including the demonstration material) **and completing the reading exercises** you should know:
    1. That the `scipy.io` package enables files that are stored using various **proprietary formats** to be **read** from and **written** to in ***Python***.
    2. How to use the `scipy.io` package to **read** and **write MATLAB** files in ***Python***.

LECTURE 12 SUMMARY

3. How to inspect the **symbol tables** that shared libraries incorporate using the `nm` utility.
4. Why **C++** compilers must "**mangle**" the **names of symbols** that appear in shared library symbol tables.
5. How to invert the mangling process using the `c++filt` utility.
6. How to use ***Cython*** to generate ***Python*** modules using **C++ header files** and **shared libraries**.

RECOMMENDED READING

`scipy.io`

https://docs.scipy.org/doc/scipy-0.15.1/reference/tutorial/io.html

The Cython Online Tutorial

(docs.cython.org/src/userguide/index.html)

In particular, consult the sections entitled:

- Basic Tutorial
- Calling C functions
- Using C libraries

The Cython Users Guide

(docs.cython.org/src/userguide/index.html)

You may be interested in the section entitled:

- Using C++ in Cython

LECTURE 12 HOMEWORK

Review the **Python** demonstration material from Lectures **11 and 12**!

Continue to refine your **final project proposal** and begin working on your **final project** once it is approved.

- Review the ***Recommended Reading*** items listed on the previous slide.