Incorporating Other Languages into Python

Joey Bernard

Land Acknowledgement

Housekeeping

Introduction

- Python has become the default glue language for science

Introduction

- Python has become the default glue language for science
- It is not ideal for all cases

Introduction

- Python has become the default glue language for science
- It is not ideal for all cases
- We will look at how to offload issues to another language

Installation

- We need several tools

Installation

- We need several tools
- Almost everything we will discuss involves C/C++

Installation

- We need several tools
- Almost everything we will discuss involves C/C++
- You will need Python plus a C/C+ compiler

Installation

- We need several tools
- Almost everything we will discuss involves C/C++
- You will need Python plus a C/C+ compiler
- All of this work should be done in a virtual environment (now necessary under Ubuntu)

Software Options

- Most Linux distributions will have everything you need in their package management system

Software Options

- Most Linux distributions will have everything you need in their package management system
- For Windows, you can use WSL to get a Linux environment

Software Options

- Most Linux distributions will have everything you need in their package management system
- For Windows, you can use WSL to get a Linux environment
- You can also use scoop (https://scoop.sh) to install Windows developer tools

Software Options

- Most Linux distributions will have everything you need in their package management system
- For Windows, you can use WSL to get a Linux environment
- You can also use scoop (https://scoop.sh) to install Windows developer tools
- For Apple Macs, you can use homebrew (https://brew.sh) to do the same thing

Pre-existing Examples

- Several of the high performance libraries already do this

Pre-existing Examples

- Several of the high performance libraries already do this
- numpy uses C, C++ and FORTRAN (in order of usage)

Pre-existing Examples

- Several of the high performance libraries already do this
- numpy uses C, C++ and FORTRAN (in order of usage)
- scipy uses C, FORTRAN and C++ (in order of usage)

Why do this

- Python is an object oriented language, without static typing

Why do this

- Python is an object oriented language, without static typing
- This means loops can be horrendous

Why do this

- Python is an object oriented language, without static typing
- This means loops can be horrendous
- Also have the GIL, throttling multi-process work

Virtual Environments

- The first step is creating a virtual environment

```
python -m venv python_project1
```

- This creates a new directory for your project
- You can activate it with

```
cd ./python_project1
. ./bin/activate
```

- When you are done, you can simply run the command

```
deactivate
```

First step - Numba

- In some cases, you just need a slightly faster Python

First step - Numba

- In some cases, you just need a slightly faster Python
- Whenever you try to optimize, remember the quote - ***Early optimization is the root of all evil***

First step - Numba

- In some cases, you just need a slightly faster Python
- Whenever you try to optimize, remember the quote - ***Early optimization is the root of all evil***
- You want to do the bare minimum to get the results that you actually need

First step - Numba

- In some cases, you just need a slightly faster Python
- Whenever you try to optimize, remember the quote - ***Early optimization is the root of all evil***
- You want to do the bare minimum to get the results that you actually need
- Numba allows for compiling portions of your Python code

Numba - installation

- Numba is installed using the command

```
pip install numba
```

- This will install the numba module, along with llvmlite
- This why you should use virtual environments - to keep your projects clean and isolated

Numba - cont'd

- Numba uses decorators to encapsulate your code

Numba - cont'd

- Numba uses decorators to encapsulate your code
- The most common decorator is `@jit`

Numba - cont'd

- Numba uses decorators to encapsulate your code
- The most common decorator is `@jit`
- This decorator has loads of options, including whether to parallelize or whether to target a GPU

Numba - options

- ***nogil*** - whether to release the GIL when entering the compiled code

Numba - options

- ***nogil*** - whether to release the GIL when entering the compiled code
- ***cache*** - whether to save off compiled code into a file cache to avoid the compiling step each time

Numba - options

- ***nogil*** - whether to release the GIL when entering the compiled code
- ***cache*** - whether to save off compiled code into a file cache to avoid the compiling step each time
- ***parallel*** - whether to parallelize compiled code when possible (e.g. loops)

Numba - options

- ***nogil*** - whether to release the GIL when entering the compiled code
- ***cache*** - whether to save off compiled code into a file cache to avoid the compiling step each time
- ***parallel*** - whether to parallelize compiled code when possible (e.g. loops)
- ***fastmath*** - whether to use strict IEEE 754 math (similar to the GCC flag)

Numba - explicit typing

- One issue with Python is that variables are untyped
- You can assign a type signature as part of the jit decorator
- For example

```
from numba import jit

@jit(int32(int32,int32))
def my_func(val1, val2):
    return val1 + val2
```

- This allows numba to know what the data types are and to compile away the usual checks that Python has to do

Numba - usage

- Compiling your code is as easy as

```
numba my_code.py
```

- You can also output debugging information with options like

```
numba my_code.py --annotate
OR
numba my_code.py --dump_llvm
```

Numba - numpy universal functions

- You can create numpy ufuncs by decorating your Python code

Numba - numpy universal functions

- You can create numpy ufuncs by decorating your Python code
- For scalar input arguments, using the ***@vectorize*** decorator for your function

Numba - numpy universal functions

- You can create numpy ufuncs by decorating your Python code
- For scalar input arguments, using the ***@vectorize*** decorator for your function
- For data structures, you can use the ***@guvectorize*** decorator

Numba - numpy universal functions

- You can create numpy ufuncs by decorating your Python code
- For scalar input arguments, using the ***@vectorize*** decorator for your function
- For data structures, you can use the ***@guvectorize*** decorator
- While you could just use the ***@jit*** decorator with an iteration loop, but this method adds in the numpy features, like reduction, accumulation or broadcasting

Numba - numpy example

```
from numba import vectorize, float64

@vectorize([float64(float64, float64)])
def f(x, y):
    return x + y
```

Numba - AOT compilation

- Usually numba compiles code at run-time

Numba - AOT compilation

- Usually numba compiles code at run-time
- You can pre-compile your code before having to use it

Numba - AOT compilation

- Usually numba compiles code at run-time
- You can pre-compile your code before having to use it
- This allows you to distribute the code to users who may not have numba installed

Numba - AOT example

```python
from numba.pycc import CC
cc = CC('my_module')

@cc.export('multf', 'f8(f8, f8)')
@cc.export('multi', 'i4(i4, i4)')
def mult(a, b):
    return a * b

@cc.export('square', 'f8(f8)')
def square(a):
    return a ** 2

if __name__ == "__main__":
    cc.compile()
```

Numba - AOT compiling

- Running the above script generates a shared library that contains the compiled code

Numba - AOT compiling

- Running the above script generates a shared library that contains the compiled code
- This won't work for numpy ufuncs

Numba - AOT compiling

- Running the above script generates a shared library that contains the compiled code
- This won't work for numpy ufuncs
- Exported functions don't check argument types

Numba - AOT compiling

- Running the above script generates a shared library that contains the compiled code
- This won't work for numpy ufuncs
- Exported functions don't check argument types
- AOT produces generic architecture code, while JIT produces specific code

Numba - jit_module

- In some cases, you may have an entire module worth of code that you want to pass through numba's JIT

Numba - jit_module

- In some cases, you may have an entire module worth of code that you want to pass through numba's JIT
- You can use the ***jit_module()*** function within your module code to apply the changes, rather than having decorate every function individually

Numba - jit_module

- In some cases, you may have an entire module worth of code that you want to pass through numba's JIT
- You can use the ***jit_module()*** function within your module code to apply the changes, rather than having decorate every function individually
- Any functions that you do decorate will use those options, rather than the module level options

Next step - Cython

- Cython allows for adding C/C++ data types, and outputting compiled code

Next step - Cython

- Cython allows for adding C/C++ data types, and outputting compiled code
- You need to annotate your code in order to tell Cython what is expected

Next step - Cython

- Cython allows for adding C/C++ data types, and outputting compiled code
- You need to annotate your code in order to tell Cython what is expected
- You will need to have your own C/C++ compiler - ideally the same as the compiler used for Python

Next step - Cython

- Cython allows for adding C/C++ data types, and outputting compiled code
- You need to annotate your code in order to tell Cython what is expected
- You will need to have your own C/C++ compiler - ideally the same as the compiler used for Python
- This becomes easy to mess up under Windows - consider strongly using WSL

Cython - annotation

- Cython will compile pure Python
- Using static typing will give a decent initial speedup
- You can annotate so that it looks like

```
import cython

i: cython.int
j: cython.double
```

Cython - different notation

Pure Python

```
def primes(nb_primes: cython.i
    i: cython.int
    p: cython.int[1000]

    if nb_primes > 1000:
    nb_primes = 1000
    # Only if regular Python i
    if not cython.compiled:
    # Make p work almost like
    p = [0] * 1000

    len_p: cython.int = 0  # T
    n: cython.int = 2
```

Older Cython

```
def primes(int nb_primes):
    cdef int n, i, len_p
    cdef int[1000] p

    if nb_primes > 1000:
    nb_primes = 1000




    # The current number of el
    len_p = 0
    n = 2                      ts
    while len_p < nb_primes:
```

Cython - usage

- The easiest way to build Cython code is to use setuptools

```
pip install cython
pip install setuptools
```

- This way, you can use setuptools to build your Cython module
- Files can use endings *.pyx* or *.py*

Cython - hello world

- We can start with the classic *Hello World* in the file ***hello.pyx***

```
def say_hello_to(name):
    print(f"Hello {name}!")
```

Cython - setuptools

- To build it, we'll need a *setup.py* script

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

setup(
    name='Hello World app',
    ext_modules=cythonize("hello.pyx"),
)
```

Cython - building

- To build it, you would use the command

```
python setup.py build_ext --inplace
```

- Then you can use it with

```
from hello import say_hello_to
```

Cython - basics

- You can call C functions from libraries

Cython - basics

- You can call C functions from libraries
- You have the ability to use static types

Cython - basics

- You can call C functions from libraries
- You have the ability to use static types
- Writing Python wrappers allows your Python code to use C libraries

Cython - basics

- You can call C functions from libraries
- You have the ability to use static types
- Writing Python wrappers allows your Python code to use C libraries
- Your Cython code gets compiled down to C

Cython - stdlib functions

- You can import and use C functions from the standard library

```
from cython.cimports.libc.stdlib import atoi

@cython.cfunc
def parse_charptr_to_py_int(s: cython.p_char):
    assert s is not cython.NULL, "byte string value is NULL
    return atoi(s)  # note: atoi() has no error detection!
```

Cython - other libraries

- You can also import from other libraries, e.g. *** from cython.cimports.libc.math import sin***

Cython - other libraries

- You can also import from other libraries, e.g. *** from cython.cimports.libc.math import sin***
- Some libraries (like math) are not automatically linked - you will have to add linking information to your ***setup.py*** file

Cython - external libraries

- Using other external libraries isn't quite so seemless

Cython - external libraries

- Using other external libraries isn't quite so seemless
- You will need to write a ***.pxd*** file to wrap the details from the header file for your external library

Cython - external libraries

- Using other external libraries isn't quite so seemless
- You will need to write a ***.pxd*** file to wrap the details from the header file for your external library
- You then will need to write a Python wrapper class to encapsulate the calls to the C code

Cython - cqueue pxd file

```
cdef extern from "c-algorithms/src/queue.h":
    ctypedef struct Queue:
    pass
    ctypedef void* QueueValue
    Queue* queue_new()
    void queue_free(Queue* queue)
    int queue_push_head(Queue* queue, QueueValue data)
    QueueValue  queue_pop_head(Queue* queue)
    QueueValue queue_peek_head(Queue* queue)
    int queue_push_tail(Queue* queue, QueueValue data)
    QueueValue queue_pop_tail(Queue* queue)
    QueueValue queue_peek_tail(Queue* queue)
    bint queue_is_empty(Queue* queue)
```

Cython - wrapper class for a queue

```
from cython.cimports import cqueue

@cython.cclass
class Queue:
    _c_queue: cython.pointer[cqueue.Queue]

    def __cinit__(self):
    self._c_queue = cqueue.queue_new()
```

Cython - strings

- Strings prove to be a bit of a mess

Cython - strings

- Strings prove to be a bit of a mess
- Cython supports 4 types: *bytes*, *str*, *unicode* and *basestring*

Cython - strings

- Strings prove to be a bit of a mess
- Cython supports 4 types: *bytes*, *str*, *unicode* and *basestring*
- Involves a decoding/encoding step when going back and forth between Python and C

Cython - memory management

- Memory management on the Python side is a non-issue

Cython - memory management

- Memory management on the Python side is a non-issue
- Objects are auto-created, and then cleaned up by the garbage collector

Cython - memory management

- Memory management on the Python side is a non-issue
- Objects are auto-created, and then cleaned up by the garbage collector
- Most simple objects move into C by being assigned to the stack

Cython - memory management

- Memory management on the Python side is a non-issue
- Objects are auto-created, and then cleaned up by the garbage collector
- Most simple objects move into C by being assigned to the stack
- Sometimes, you need to manually assign heap space for larger or more complex objects

Cython - using numpy

- You are able to use numpy data types, especially arrays

Cython - using numpy

- You are able to use numpy data types, especially arrays
- This allows faster access and indexing

Cython - using numpy

- You are able to use numpy data types, especially arrays
- This allows faster access and indexing
- ***ndarray*** allows near direct C-like access to data within numpy arrays

Cython - parallelization

- You can write code that uses OpenMP threaded parallelization

Cython - parallelization

- You can write code that uses OpenMP threaded parallelization
- This side-steps the GIL, so you get true concurrent parallel code

Cython - parallelization

- You can write code that uses OpenMP threaded parallelization
- This side-steps the GIL, so you get true concurrent parallel code
- This means that you can't directly use Python objects, you need to move completely into C

Cython - parallelization

- You can write code that uses OpenMP threaded parallelization
- This side-steps the GIL, so you get true concurrent parallel code
- This means that you can't directly use Python objects, you need to move completely into C
- Your C compiler needs to support OpenMP (most do)

Cython - C++ options

- There is also the ability to use C++

Cython - C++ options

- There is also the ability to use C++
- The ***cython.cimports.libcpp*** sub-module provides for lots of C++ imports, like vectors

Cython - C++ options

- There is also the ability to use C++
- The ***cython.cimports.libcpp*** sub-module provides for lots of C++ imports, like vectors
- This requires a native part of the module, specific to your infrastructure

Cython - pure Python

- You may not have the ability to use a C compiler, but still want some performance help

Cython - pure Python

- You may not have the ability to use a C compiler, but still want some performance help
- Cython allows you to statically type your code, along with other cythonic functionality

Cython - pure Python

- You may not have the ability to use a C compiler, but still want some performance help
- Cython allows you to statically type your code, along with other cythonic functionality
- You can use an augmenting ***.pxd*** file to cythonize your ***.py*** file

Cython - pure Python

- You may not have the ability to use a C compiler, but still want some performance help
- Cython allows you to statically type your code, along with other cythonic functionality
- You can use an augmenting ***.pxd*** file to cythonize your ***.py*** file
- You can explicitly mark code as needing or not needing the GIL - this helps the interpreter run parallel threads

Boost-y binding 1 - pybind11

- There is a ***Boost.Python*** library - unfortunately you have to use ***Boost***

Boost-y binding 1 - pybind11

- There is a ***Boost.Python*** library - unfortunately you have to use ***Boost***
- ***pybind11*** provides a much smaller and focused library to pull C++ into Python

Boost-y binding 1 - pybind11

- There is a ***Boost.Python*** library - unfortunately you have to use ***Boost***
- ***pybind11*** provides a much smaller and focused library to pull C++ into Python
- Allows for C++ types, function calls, data structures, classes, etc

pybind11 - installation

- You can install ***pybind11*** through pip:

```
pip install pybind11
```

- You also need a C++ compiler, along with the development package for Python
- You also need a build system (cmake, meson, setuptools)

pybind11 - boilerplate

- You will likely need the following two lines at the top of any of your C++ source code files

```
#include <pybind11/pybind11.h>

namespace py = pybind11;
```

- Now you can add binding code to your C++ source files

pybind11 - example file

```
#include <pybind11/pybind11.h>

int add(int i, int j) {
    return i + j;
}

PYBIND11_MODULE(example, m) {
    m.doc() = "pybind11 example plugin"; // optional module

    m.def("add", &add, "A function that adds two numbers");
}
```

pybind11 - building

- Since ***pybind11*** is based off of Boost, then it is also a header-only package
- This means that you don't need to link to any extra library
- Building is done through compilation

```
$ c++ -O3 -Wall -shared -std=c++11 -fPIC $(python3 -m pybi
```

- You can now import the compiled module in Python the usual way

pybind11 - keyword arguments

- In the exaple, the arguments are positional
- You need to add some code to allow for keyword arguments

```
m.def("add", &add, "A function which adds two numbers",
      py::arg("i"), py::arg("j"));
```

pybind11 - exporting variables

```
PYBIND11_MODULE(example, m) {
    m.attr("the_answer") = 42;
    py::object world = py::cast("World");
    m.attr("what") = world;
}
```

```
>>> import example
>>> example.the_answer
42
>>> example.what
'World'
```

Boost-y binding 2 - Nanobind

- ***nanobind*** is another Boost-y module, by the same person who wrote ***pybind11***

Boost-y binding 2 - Nanobind

- ***nanobind*** is another Boost-y module, by the same person who wrote ***pybind11***
- ***nanobind*** is even smaller, providing a subset of C++ functionality for your Python code

nanobind - installation

- Like everything else today, you can install using pip:

```
pip install nanobind
```

- You will also need a C++ compiler
- ***nanobind*** support various build systems (cmake, meson, bazel)

nanobind - basics

- A basic module looks like

```
#include <nanobind/nanobind.h>

int add(int a, int b) { return a + b; }

NB_MODULE(my_ext, m) {
    m.def("add", &add);
}
```

- Building is through a ***CMakeLists.txt***

nanobind - cmake general

```
cmake_minimum_required(VERSION 3.15...3.27)
project(my_project) # Replace 'my_project' with the name of

if (CMAKE_VERSION VERSION_LESS 3.18)
  set(DEV_MODULE Development)
else()
  set(DEV_MODULE Development.Module)
endif()

find_package(Python 3.8 COMPONENTS Interpreter ${DEV_MODULE
```

nanobind - cmake specifics

```
# Detect the installed nanobind package and import it into
execute_process(
  COMMAND "${Python_EXECUTABLE}" -m nanobind --cmake_dir
  OUTPUT_STRIP_TRAILING_WHITESPACE OUTPUT_VARIABLE nanobind
find_package(nanobind CONFIG REQUIRED)

nanobind_add_module(my_ext my_ext.cpp)
```

nanobind - example

```
#include <nanobind/nanobind.h>

namespace nb = nanobind;
using namespace nb::literals;

int add(int a, int b = 1) { return a + b; }

NB_MODULE(my_ext, m) {
    m.def("add", &add, "a"_a, "b"_a = 1,
      "This function adds two numbers and increments if onl
}
```

CFFI

- CFFI (C Foreign Function Interface) for Python is a more raw library

CFFI

- CFFI (C Foreign Function Interface) for Python is a more raw library
- Unlike systems like Cython, CFFI doesn't add extra syntax

CFFI

- CFFI (C Foreign Function Interface) for Python is a more raw library
- Unlike systems like Cython, CFFI doesn't add extra syntax
- You just need to know C and Python

CFFI - installation

- Installation can be done through pip:

```
pip install cffi
```

- Includes a library (libffi) that can be messy to setup correctly on some platforms

CFFI - basics

- You start with an ***FFI()*** object

CFFI - basics

- You start with an ***FFI()*** object
- You use the *cdef()* method to provide C declarations

CFFI - basics

- You start with an ***FFI()*** object
- You use the *cdef()* method to provide C declarations
- You use the *set_source()* method to define the Python extension module, along with the associated C code

CFFI - basics

- You start with an ***FFI()*** object
- You use the *cdef()* method to provide C declarations
- You use the *set_source()* method to define the Python extension module, along with the associated C code
- You use the *compile()* method to generate the compiled library

CFFI - basics

- You start with an ***FFI()*** object
- You use the *cdef()* method to provide C declarations
- You use the *set_source()* method to define the Python extension module, along with the associated C code
- You use the *compile()* method to generate the compiled library
- You can then import this library like any other Python module

CFFI - example

```
from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef("""
    float pi_approx(int n);
""")

ffibuilder.set_source("_pi_cffi",
"""
     #include "pi.h"   // the C header of the library
""",
     libraries=['piapprox'])   # library name, for the link
```

```
if __name__ == "__main__":
    ffibuilder.compile(verbose=True)
```

CFFI - setup.py

```
from setuptools import setup

setup(
    ...
    setup_requires=["cffi>=1.0.0"],
    cffi_modules=["piapprox_build:ffibuilder"], # "filename
    install_requires=["cffi>=1.0.0"],
)
```

CFFI - modes of use

- ABI - Application Binary Interface

CFFI - modes of use

- ABI - Application Binary Interface
- API - Application Programming Interface

CFFI - modes of use

- ABI - Application Binary Interface
- API - Application Programming Interface
- in-line - everything is setup everytime you import your code

CFFI - modes of use

- ABI - Application Binary Interface
- API - Application Programming Interface
- in-line - everything is setup everytime you import your code
- out-of-line - there is a separate step that compiles your code for import

CFFI - ABI, in-line

```
from cffi import FFI
ffi = FFI()
ffi.cdef("""
    int printf(const char *format, ...);
""")
# loads the entire C namespace
C = ffi.dlopen(None)
# equivalent to C code: char arg[] = "world";
arg = ffi.new("char[]", b"world")
C.printf(b"hi there, %s.\n", arg)
```

CFFI - API, out-of-line - 1

```
from cffi import FFI
ffibuilder = FFI()

ffibuilder.cdef("int foo(int *, int *, int);")

ffibuilder.set_source("_example",
r"""
    static int foo(int *buffer_in,
            int *buffer_out, int x)
    {
    /* some algorithm that is seriously
    faster in C than in Python */
    }
""")
```

CFFI - API, out-of-line - 2

```python
from _example import ffi, lib

buffer_in = ffi.new("int[]", 1000)
# initialize buffer_in here...

# easier to do all buffer allocations
# in Python and pass them to C,
# even for output-only arguments
buffer_out = ffi.new("int[]", 1000)

result = lib.foo(buffer_in, buffer_out, 1000)
```

HPy

- Technically still alpha (version 0.9.0)

HPy

- Technically still alpha (version 0.9.0)
- An attempt at modernizing how to incorporate C into Python

HPy

- Technically still alpha (version 0.9.0)
- An attempt at modernizing how to incorporate C into Python
- It is much more like the C/API

HPy - installation

- Installation is through pip:

```
pip install hpy
```

- You need a C compiler
- You actually write C source code and compile it into a library that can be imported into Python

swig - not just for Python

- ***swig*** (Simplified Wrapper and Interface Generator) builds scripting language interfaces to C and C++

swig - not just for Python

- ***swig*** (Simplified Wrapper and Interface Generator) builds scripting language interfaces to C and C++
- Works for languages like Python, Tcl, Perl and Guile

swig - installation

- swig is not part of the Python community

swig - installation

- swig is not part of the Python community
- You can install it from source

swig - installation

- swig is not part of the Python community
- You can install it from source
- Check your platform package manager to see if it is already there

swig - basics

- You write the C source code in its own file

swig - basics

- You write the C source code in its own file
- You create an interface file to declare what C functions are available

swig - basics

- You write the C source code in its own file
- You create an interface file to declare what C functions are available
- Calling ***swig*** generates the needed wrapper C code

swig - basics

- You write the C source code in its own file
- You create an interface file to declare what C functions are available
- Calling ***swig*** generates the needed wrapper C code
- You then need to compile the C ocde and link it together into a shared library

swig - basics

- You write the C source code in its own file
- You create an interface file to declare what C functions are available
- Calling ***swig*** generates the needed wrapper C code
- You then need to compile the C ocde and link it together into a shared library
- This can then be imported into Python

pyO3 - a Rust option

- Rust is more of a platform than C or C++

pyO3 - a Rust option

- Rust is more of a platform than C or C++
- This requires more tooling to develop code

pyO3 - a Rust option

- Rust is more of a platform than C or C++
- This requires more tooling to develop code
- pyO3 can be used to call Rust in Python, or Python in Rust

pyO3 - a Rust option

- Rust is more of a platform than C or C++
- This requires more tooling to develop code
- pyO3 can be used to call Rust in Python, or Python in Rust
- Packages managed at https://crates.io

pyO3 - a Rust option

- Rust is more of a platform than C or C++
- This requires more tooling to develop code
- pyO3 can be used to call Rust in Python, or Python in Rust
- Packages managed at https://crates.io
- Searching for “science” gives 656 crates, environment is still being built out

pyO3 - installation

- The easiest way is to use ***maturin*** inside a virtual environment to initialize a project

```
pip install maturin
maturin init --bindings pyo3
```

- This creates several project files, the most important of which are ***Cargo.toml*** and ***src/lib.rs***

pyO3 - Cargo.toml

```
[package]
name = "string_sum"
version = "0.1.0"
edition = "2021"

[lib]
# The name of the native library.
name = "string_sum"
# "cdylib" is necessary to produce a shared library for Pyt
crate-type = ["cdylib"]

[dependencies]
pyo3 = { version = "0.25.0", features = ["extension-module'
```

pyO3 - lib.rs

```
use pyo3::prelude::*;

/// Formats the sum of two numbers as string.
#[pyfunction]
fn sum_as_string(a: usize, b: usize) -> PyResult<String> {
    Ok((a + b).to_string())
}

/// A Python module implemented in Rust.
/// The name of this function must match
/// the `lib.name` setting in the `Cargo.toml`,
/// else Python will not be able to
/// import the module.
#[pymodule]
fn string_sum(m: &Bound<'_, PyModule>) -> PyResult<()> {
```

pyO3 - building

- To build code, use

```
maturin develop
```

- This will build the library and install it into the virtual environment that we are currently in

pyO3 - functions

- As we saw in a previous slide, you can decorate a function in Rust so that it can be used in Python

pyO3 - functions

- As we saw in a previous slide, you can decorate a function in Rust so that it can be used in Python
- ***pyO3*** actually creates a C wrapper that acts as an intermediate layer between Rust and Python

pyO3 - functions

- As we saw in a previous slide, you can decorate a function in Rust so that it can be used in Python
- ***pyO3*** actually creates a C wrapper that acts as an intermediate layer between Rust and Python
- Most of the same concerns and functionalities from solutions like Cython also exist here

pyO3 - functions

- As we saw in a previous slide, you can decorate a function in Rust so that it can be used in Python
- ***pyO3*** actually creates a C wrapper that acts as an intermediate layer between Rust and Python
- Most of the same concerns and functionalities from solutions like Cython also exist here
- The same ability to avoid the GIL is provided through ***pyO3***

pyO3 - example

```rust
use pyo3::prelude::*;

#[pyfunction]
fn double(x: usize) -> usize {
    x * 2
}

#[pymodule]
fn my_extension(m: &Bound<'_, PyModule>) -> PyResult<()> {
    m.add_function(wrap_pyfunction!(double, m)?)
}
```

pyO3 - shorthand

```
use pyo3::prelude::*;

#[pymodule]
fn my_extension(m: &Bound<'_, PyModule>) -> PyResult<()> {
    #[pyfn(m)]
    fn double(x: usize) -> usize {
    x * 2
    }

    Ok(())
}
```

pyO3 - parallelism

- Since the Rust code is running outside of Python, it can take advantage of true prallelism

pyO3 - parallelism

- Since the Rust code is running outside of Python, it can take advantage of true prallelism
- There is a call (***Python::allow_threads***) that temporarily releases the GIL and allows other threads within Python to continue running

Conclusion

- If you have started your project in Python, there are lots of ways of incrementally adding other languages for performance

Conclusion

- If you have started your project in Python, there are lots of ways of incrementally adding other languages for performance
- If you have started in your code in C/C++ or some other language, there are lots of options to wrap your code in Python to make it easier to share

Resources

- https://numba.pydata.org

Resources

- https://numba.pydata.org
- https://cython.org

Resources

- https://numba.pydata.org
- https://cython.org
- https://github.com/pybind/pybind11

Resources

- https://numba.pydata.org
- https://cython.org
- https://github.com/pybind/pybind11
- https://github.com/wjakob/nanobind

Resources

- https://numba.pydata.org
- https://cython.org
- https://github.com/pybind/pybind11
- https://github.com/wjakob/nanobind
- https://github.com/python-cffi/cffi

Resources

- https://numba.pydata.org
- https://cython.org
- https://github.com/pybind/pybind11
- https://github.com/wjakob/nanobind
- https://github.com/python-cffi/cffi
- https://hpyproject.org

Resources

- https://numba.pydata.org
- https://cython.org
- https://github.com/pybind/pybind11
- https://github.com/wjakob/nanobind
- https://github.com/python-cffi/cffi
- https://hpyproject.org
- https://www.swig.org

Resources

- https://numba.pydata.org
- https://cython.org
- https://github.com/pybind/pybind11
- https://github.com/wjakob/nanobind
- https://github.com/python-cffi/cffi
- https://hpyproject.org
- https://www.swig.org
- https://github.com/PyO3/pyo3