Incorporating Other Languages into Python

Joey Bernard

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
- 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)

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
- 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
- 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 - cont'd

- 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
- 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
- ***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
```

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 - 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
    while len_p < nb_primes:
    # Is n prime?
    for i in p[:len_p]:
        if n % i == 0:
```

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:
    # Is n prime?
    for i in p[:len_p]:
        if n % i == 0:
        break
```

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

say_hello_to('Joey')
```

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 - 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
- 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
- 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
- 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++
- 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 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***
- ***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)

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, meso



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



## CFFI - installation
- Installation can be done through pip:
```bash
pip install cffi
```

- Includes a library (libffi) that can be messy to setup correctly
```

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

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++
- Works for languages like Python, Tcl, Perl and Guile

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
- 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++
- This requires more tooling to develop code
- pyO3 can be used to call Rust in Python, or Python in Rust

pyO3 - installation

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

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

- To build code, use

```
maturin develop
```