LESSON NAME

Intro

Prerequisites

- FIXME
- …
- …

20 min	filename

Episode template

Questions

- What syntax is used to make a lesson?
- How do you structure a lesson effectively for teaching?
- `questions` are at the top of a lesson and provide a starting point for what you might learn. It is usually a bulleted list.

Objectives

- Show a complete lesson page with all of the most common structures.
- …

This is also a holdover from the carpentries-style. It could usually be left off.

The introduction should be a high level overview of what is on the page and why it is interesting.

The lines below (only in the source) will set the default highlighting language for the entire page.

Section

A section.

Discussion

g.

Skip to content

- A discussion section
- Another discussion topic

Section

```python
print("hello world")
# This uses the default highlighting language
```

```
print("hello world)
```

Exercises: description

Exercise Topic-1: imperative description of exercise

Exercise text here.

Solution

Solution text here

Summary

A Summary of what you learned and why it might be useful. Maybe a hint of what comes next.

See also

- Other relevant links
- Other link

Keypoints

- What the learner should take away
- point 2
- …

This is another holdover from the carpentries style. This perhaps is better done in a “summary” section.

Cython

Cython is a superset of Python that additionally supports calling C functions and declaring C types on variables and class attributes. It is also a versatile, general purpose compiler. Since it is supports a superset of Python syntax, nearly all Python code, including 3rd party Python

Skip to content

valid Cython code. Under Cython, source code gets translated into optimized compiled as Python extension modules.

Developers can either:

- prototype and develop Python code in IPython/Jupyter using the `%%cython` magic command (**easy**), or
- run the `cython` command-line utility to produce a `.c` file from a `.py` or `.pyx` file, which in turn needs to be compiled with a C compiler to an `.so` library, which can then be directly imported in a Python program (**intermediate**), or
- use setuptools or meson with meson-python to automate the aforementioned build process (**advanced**).

Herein, we restrict the discussion to the Jupyter-way of using the `%%cython` magic. A full overview of Cython capabilities refers to the documentation.

> Important
>
> Due to a known issue with `%%cython -a` in `jupyter-lab` we have to use the `jupyter-nbclassic` interface for this episode.

Python: Baseline (step 0)

> Demo: Cython
>
> Consider a problem to integrate a function:
>
> $$I = \int^{b}_{a}(x^2 - x)dx$$
>
> which can be numerically approximated as the following sum:
>
> $$I \approx \delta x \sum_{i=0}^{N-1} (x_i^2 - x_i)$$
>
> where $a \le x_i \lt b$, and all x_i are uniformly spaced apart by $\delta x = (b - a) / N$.
>
> **Objective**: Repeatedly compute the approximate integral for 1000 different combinations of a, b and N.

Python code is provided below:

```python
import numpy as np

def f(x):
    return x ** 2 - x

def integrate_f(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f(a + i * dx)
    return s * dx

def apply_integrate_f(col_a, col_b, col_N):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f(col_a[i], col_b[i], col_N[i])
    return res
```

We generate a dataframe and apply the `apply_integrate_f()` function on its columns, timing the execution:

```python
import pandas as pd

df = pd.DataFrame(
    {
        "a": np.random.randn(1000),
        "b": np.random.randn(1000),
        "N": np.random.randint(low=100, high=1000, size=1000)
    }
)

%timeit apply_integrate_f(df['a'], df['b'], df['N'])
# 101 ms ± 736 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Cython: Benchmarking (step 1)

In order to use Cython, we need to import the Cython extension:

```python
%load_ext cython
```

As a first cythonization step, we add the cython magic command (`%%cython -a`) on top of Jupyter

Skip to content

t by a simply compiling the Python code using Cython without any changes.
n below:

```
%%cython -a

import numpy as np

def f_cython_step1(x):
    return x * (x - 1)

def integrate_f_cython_step1(a, b, N):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_cython_step1(a + i * dx)
    return s * dx

def apply_integrate_f_cython_step1(col_a, col_b, col_N):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step1(col_a[i], col_b[i], col_N[i])
    return res
```

Yellow lines hint at Python interaction.
Click on a line that starts with a " + " to see the C code that Cython generated for it.

```
 01:
+02: import numpy as np
 03:
+04: def f_cython_step1(x):
+05:     return x * (x - 1)
 06:
+07: def integrate_f_cython_step1(a, b, N):
+08:     s = 0
+09:     dx = (b - a) / N
+10:     for i in range(N):
+11:         s += f_cython_step1(a + i * dx)
+12:     return s * dx
 13:
+14: def apply_integrate_f_cython_step1(col_a, col_b, col_N):
+15:     n = len(col_N)
+16:     res = np.empty(n,dtype=np.float64)
+17:     for i in range(n):
+18:         res[i] = integrate_f_cython_step1(col_a[i], col_b[i], col_N[i])
+19:     return res
```

ANNOTATED CYTHON CODE OBTAINED BY RUNNING THE CODE ABOVE. THE YELLOW COLORING IN THE OUTPUT SHOWS US THE AMOUNT OF PURE PYTHON CODE.

Our task is to remove as much yellow as possible by *static typing*, *i.e.* explicitly declaring arguments, parameters, variables and functions.

Skip to content

e Python code just using Cython, and it may give either similar or a slight increase in performance.

```
%timeit apply_integrate_f_cython_step1(df['a'], df['b'], df['N'])
# 102 ms ± 2.06 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Cython: Adding data type annotation to input variables (step 2)

Now we can start adding data type annotation to the input variables as highlightbed in the code example/cython below:

Pure Python Cython

```
%%cython -a

import cython
import numpy as np


def f_cython_step2(x: cython.double):
    return x ** 2 - x

def integrate_f_cython_step2(a: cython.double, b: cython.double, N: cython.long):
    s = 0
    dx = (b - a) / N
    for i in range(N):
        s += f_cython_step2(a + i * dx)
    return s * dx

def apply_integrate_f_cython_step2(
    col_a: cython.double[:],
    col_b: cython.double[:],
    col_N: cython.long[:],
):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step2(col_a[i], col_b[i], col_N[i])
    return res
```

Skip to content

```
# this will not work
#%timeit apply_integrate_f_cython_step2(df['a'], df['b'], df['N'])

# this command works (see the description below)
%timeit apply_integrate_f_cython_step2(df['a'].to_numpy(), df['b'].to_numpy(), df['N']
# 34.3 ms ± 537 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Warning

You can not pass a Series directly since Cython definition is specific to an array. Instead we should use `Series.to_numpy()` to get the underlying NumPy array which works nicely with Cython.

Note

Cython uses the normal C syntax for types and provides all standard ones, including pointers. Here is a list of some primitive C data types (refer to Cython's documentation on Types):

Cython type identifier	Pure Python dtype
`char`	`cython.char`
`int`	`cython.int`
`unsigned int`	`cython.uint`
`long`	`cython.long`
`float`	`cython.float`
`double`	`cython.double`
`double complex`	`cython.doublecomplex`
`size_t`	`cython.size_t`

Using these data types, we can also annotate arrays (see Typed Memoryviews):

- 1D `np.float64` array would be equivalent to `cython.double[:]`,
- 2D `np.float64` array would be equivalent to `cython.double[:, :]` and so on…

Important

to quote the Cython documentation,

> **Typing is not a necessity**
>
> Providing static typing to parameters and variables is convenience to speed up your code, but it is not a …ize where and when needed. In fact, typing can slow down your code in the case where the …allow optimizations but where Cython still needs to check that the type of some object …clared type.

Cython: Adding data type annotation to functions (step 3)

Next step, we further add type annotation to functions. There are three ways of declaring functions:

- `def` - Python style:
  - Called by Python or Cython code, and both input/output are Python objects.
  - Declaring argument types and local types (thus return values) can allow Cython to generate optimized code which speeds up the execution.
  - Once types are declared, a `TypeError` will be raised if the function is passed with the wrong types.
- `@cython.cfunc` or `cdef` - C style:
  - cdef functions are called from Cython and C, but not from Python code.
  - Cython treats functions as pure C functions, which can take any type of arguments, including non-Python types, *e.g.*, pointers.
  - This usually gives the *best performance*.
  - However, one should really take care of the functions declared by `cdef` as these functions are actually writing in C.
- `@cython.ccall` or `cpdef` - C/Python mixed style:
  - cpdef function combines both `cdef` and `def`.
  - Cython will generate a `cdef` function for C types and a `def` function for Python types.
  - In terms of performance, `cpdef` functions may be *as fast as* those using `cdef` and might be as slow as `def` declared functions.

Pure Python | Cython

Skip to content

```
%%cython -a

import cython
import numpy as np


@cython.cfunc
def f_cython_step3(x: cython.double):
    return x ** 2 - x

@cython.cfunc
def integrate_f_cython_step3(a: float, b: float, N: int):
    s = 0
    dx = (b - a) / N

    for i in range(N):
        s += f_cython_step3(a + i * dx)
    return s * dx

@cython.ccall
def apply_integrate_f_cython_step3(
    col_a: cython.double[:],
    col_b: cython.double[:],
    col_N: cython.long[:]
):
    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step3(col_a[i], col_b[i], col_N[i])
    return res
```

```
%timeit apply_integrate_f_cython_step3(df['a'].to_numpy(), df['b'].to_numpy(), df['N']
# 29.2 ms ± 152 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Cython: Adding data type annotation to local variables and return (step 4)

Last step, we can add type annotation to local variables within functions and the return value.

Cython

```
%%cython -a

import cython
import numpy as np

@cython.cfunc
def f_cython_step4(x: cython.double) -> cython.double:
    return x ** 2 - x

@cython.cfunc
def integrate_f_cython_step4(
    a: cython.double,
    b: cython.double,
    N: cython.long
) -> cython.double:
    s: cython.double
    dx: cython.double
    i: cython.long

    s = 0
    dx = (b - a) / N

    for i in range(N):
        s += f_cython_step4(a + i * dx)
    return s * dx

@cython.ccall
def apply_integrate_f_cython_step4(
    col_a: cython.double[:],
    col_b: cython.double[:],
    col_N: cython.long[:]
) -> cython.double[:]:
    n: cython.int
    i: cython.int
    res: cython.double[:]

    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step4(col_a[i], col_b[i], col_N[i])
    return res
```



```
_integrate_f_cython_step4(df['a'].to_numpy(), df['b'].to_numpy(), df['N']
# 471 µs ± 7.38 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Now it is ~200 times faster than the baseline Python implementation, and all we have done is to add type declarations on the Python code!

Yellow lines hint at Python interaction.
Click on a line that starts with a "`+`" to see the C code that Cython generated for it.

```

import cython
import numpy as np

@cython.cfunc
def f_cython_step4(x: cython.double) -> cython.double:
    return x ** 2 - x

@cython.cfunc
def integrate_f_cython_step4(
    a: cython.double,
    b: cython.double,
    N: cython.long
) -> cython.double:
    s: cython.double
    dx: cython.double
    i: cython.long

    s = 0
    dx = (b - a) / N

    for i in range(N):
        s += f_cython_step4(a + i * dx)
    return s * dx

@cython.ccall
def apply_integrate_f_cython_step4(
    col_a: cython.double[:],
    col_b: cython.double[:],
    col_N: cython.long[:]
) -> cython.double[:]:
    n: cython.int
    i: cython.int
    res: cython.double[:]

    n = len(col_N)
    res = np.empty(n,dtype=np.float64)
    for i in range(n):
        res[i] = integrate_f_cython_step4(col_a[i], col_b[i], col_N[i])
    return res
```

WE INDEED SEE MUCH LESS PYTHON INTERACTION IN THE CODE FROM STEP 1 TO STEP 4.

Other useful features

There are some useful (and possibly advanced) features which are not covered in this episode. Some of these features are called magic attributes. Here are a few:

- `cython.cimports` package for importing and calling C libraries such as libc.math.

Skip to content

Note

Differences between `import` (for Python) and `cimport` (for Cython) statements

- `import` gives access to Python libraries, functions or attributes
- `cimport` gives access to C libraries, functions or attributes

In case of Numpy it is common to use the following, and Cython will internally handle this ambiguity.

Pure Python Cython

```
from cython.cimports.libc.stdlib import malloc, free  # Allocate and free memory
from cython.cimports.libc import math  # For math functions like sin, cos etc.
from cython.cimports import numpy as np # access to NumPy C API
```

- `cython.nogil`, which can act both as a decorator or context-manager, to manage the GIL (Global Interpreter Lock). See Cython and the GIL.
- `@cython.boundscheck(False)` and `@cython.wraparound(False)` decorators to tune indexing of Numpy array. See Cython for NumPy users.
- `@cython.cclass` to declare Extension Types which behave similar to Python classes.

In addition to the above Cython can also,

- augment with .pxd files where the Python code is kept as it is and the `.pxd` file describes the type annotation. In this form `.pxd` is very similar in function to a C/C++ header file or `.pyi` Python type annotation file,
- create parallel code using parallel blocks and `prange` iterator for element-wise parallel operation or reductions based on OpenMP threads (see Writing parallel code with Cython).

Demo

Here is a code which showcases most of the features above, except the `@cython.cclass` feature and the use of `.pxd` files.

Pure Python | Cython | Numpy | Naive Python implementation

```
import cython
from cython.parallel import parallel, prange
from cython.cimports.libc.math import sqrt

@cython.boundscheck(False)
@cython.wraparound(False)
def normalize(x: cython.double[:]):
   """Normalize a 1D array by dividing all its elements using its root-mean-square (RMS) value."""
   i: cython.Py_ssize_t
   total: cython.double = 0
   norm: cython.double
   with cython.nogil, parallel():
      for i in prange(x.shape[0]):
            total += x[i]*x[i]
      norm = sqrt(total)
      for i in prange(x.shape[0]):
            x[i] /= norm
```

Note

If you compare performance of the the Cython code versus the Numpy code, you might observe that it is either on-par, or slightly worse than Numpy. This is because Numpy vectorized operations also makes use of OpenMP parallelism and is heavily optimized. Nevertheless, it is orders of magnitude better than a naive implementation.

Conclusions

Keypoints

- Cython is a versatile, general purpose compiler for Python code
- Cython is a great way to write high-performance code in Python where algorithms are not available in scientific libraries like Numpy and Scipy and require custom implementation

See also

In order to make Cython code reusable often some packaging is necessary. The compilation to binary extension can either happen during the packaging itself, or during installation of a Python package. To learn more about how to

sions, read the following guides:

- *pyOpenSci Python packaging guide*'s page on build tools
- *Python packaging user guide*'s page on packaging binary extensions

Quick Reference

Instructor’s guide

Why we teach this lesson

Intended learning outcomes

Timing

Preparing exercises

e.g. what to do the day before to set up common repositories.

Other practical aspects

Interesting questions you might get

Typical pitfalls

Learning outcomes

FIXME

This material is for …

By the end of this module, learners should:

- …
- …

See also

Credit

FIXME

Don’t forget to check out additional course materials from …

Skip to content

Skip to content