David Hoese

Outline

- When and why should I rewrite my Python code?
- When and why is Python slow?
- Extensions with the Python C API
- Ways to connect Python and C
- Cython
- Examples
- Bonus Topics

Why should I rewrite my Python code?

- Python is slow
- Better resource usage goes a long way
  - Shared resources
  - Cheaper cloud machines
- Hot spots, not the whole application
  - small areas of an application that make a big difference
  - Tight loops see biggest improvement
- **Profile!**

When should I rewrite my Python code?

- You probably shouldn't in most cases
- Depend on low-level libraries like numpy/scipy/pandas/etc
- Optimize/improve usage of those libraries (vectorize, reduce temporary arrays, etc)
- Use cheap-ish options like dask, numexpr, numba, or multiprocessing
- Is the development and maintenance time worth it?

Why is Python slow? An interpreted language

Python → **Byte Code** → **Interpreter**

.pyc

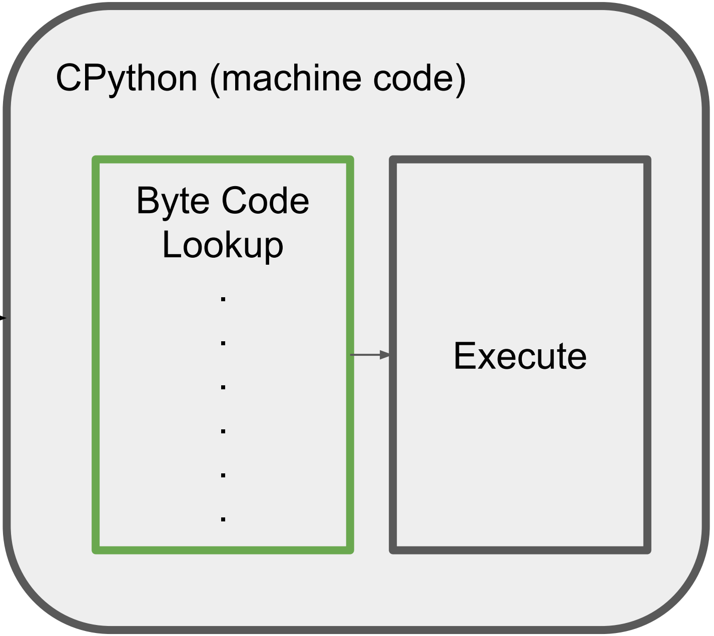
```
In [1]: import dis

In [2]: def my_func(one_list, two_list):
   ...:     res = []
   ...:     for x in one_list:
   ...:         for y in two_list:
   ...:             res.append(x + y)
   ...:     return res
   ...:

In [3]: dis.dis(my_func)
```

```
2         2 BUILD_LIST              0
          4 STORE_FAST              2 (res)

3         6 LOAD_FAST               0 (one_list)
          8 GET_ITER
       >> 10 FOR_ITER               31 (to 74)
         12 STORE_FAST              3 (x)

4        14 LOAD_FAST               1 (two_list)
         16 GET_ITER
       >> 18 FOR_ITER               26 (to 72)
         20 STORE_FAST              4 (y)

5        22 LOAD_FAST               2 (res)
         24 LOAD_METHOD             0 (append)
         46 LOAD_FAST               3 (x)
         48 LOAD_FAST               4 (y)
         50 BINARY_OP               0 (+)
         54 PRECALL                 1
         58 CALL          1
         68 POP_TOP
         70 JUMP_BACKWARD           27 (to 18)

4      >> 72 JUMP_BACKWARD              32 (to 10)

6      >> 74 LOAD_FAST              2 (res)
          76 RETURN_VALUE
```

Why is Python slow? An interpreted language

Python → Byte Code → Interpreter

```
2        2 BUILD_LIST              0
         4 STORE_FAST              2 (res)

3        6 LOAD_FAST               0 (one_list)
         8 GET_ITER
         >>   10 FOR_ITER          31 (to 74)
         12 STORE_FAST             3 (x)

4        14 LOAD_FAST              1 (two_list)
         16 GET_ITER
         >>   18 FOR_ITER          26 (to 72)
         20 STORE_FAST             4 (y)

5        22 LOAD_FAST              2 (res)
         24 LOAD_METHOD            0 (append)
         46 LOAD_FAST              3 (x)
         48 LOAD_FAST              4 (y)
         50 BINARY_OP              0 (+)
         54 PRECALL                1
         58 CALL           1
         68 POP_TOP
         70 JUMP_BACKWARD          27 (to 18)

4        >>   72 JUMP_BACKWARD               32 (to 10)

6        >>   74 LOAD_FAST         2 (res)
                   76 RETURN_VALUE
```



Why is Python slow? Everything is an object

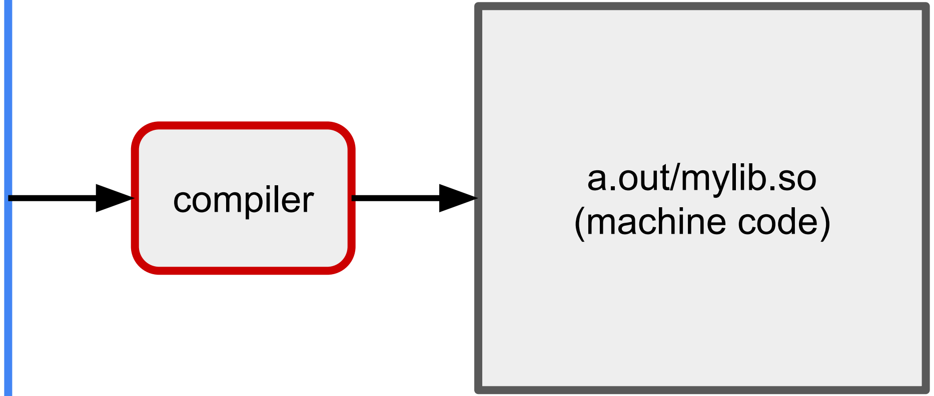
- "wasted" space
  - Implemented as a C struct: PyObject, PyLongObject, etc.
  - Reference counting
- "wasted" time
  - Allocation
  - Initialization
  - Reference counting/garbage collection

Why is Python slow? Global Interpreter Lock (GIL)

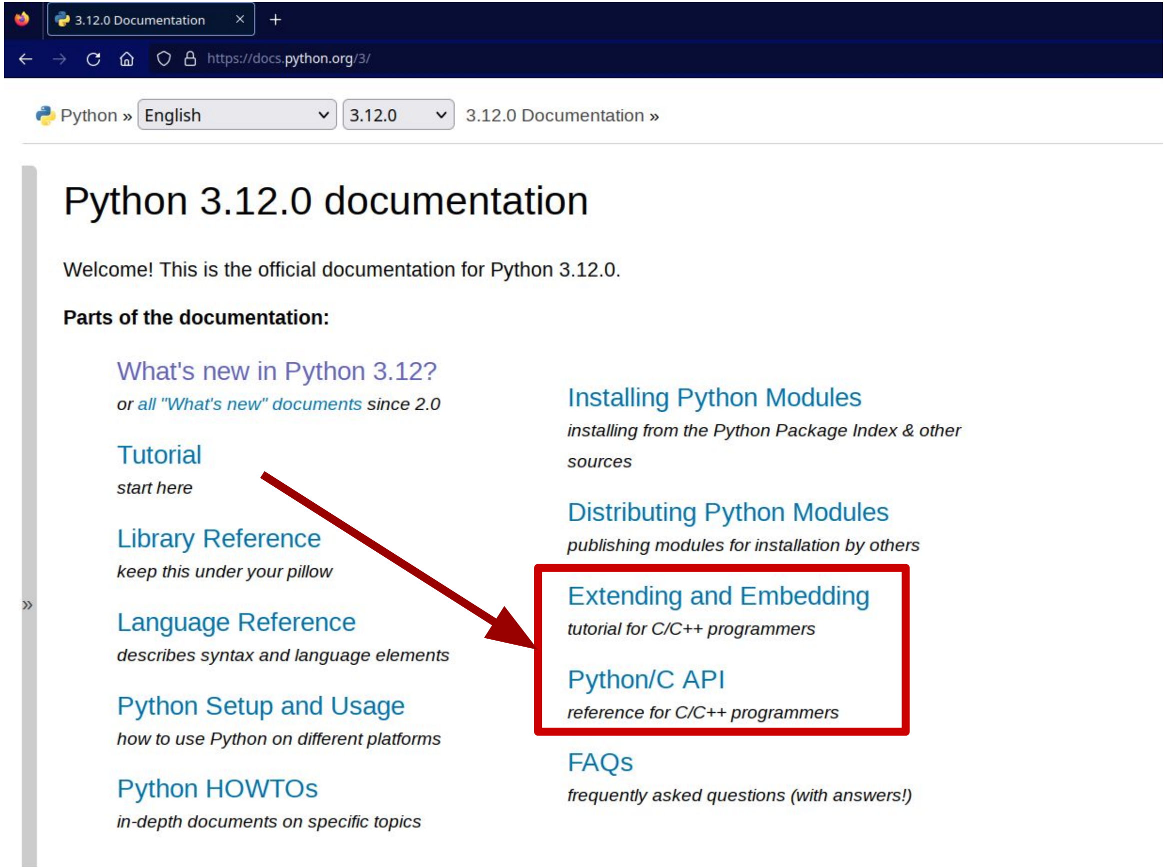
- More on this later…

C is faster…let's use C - Compiled

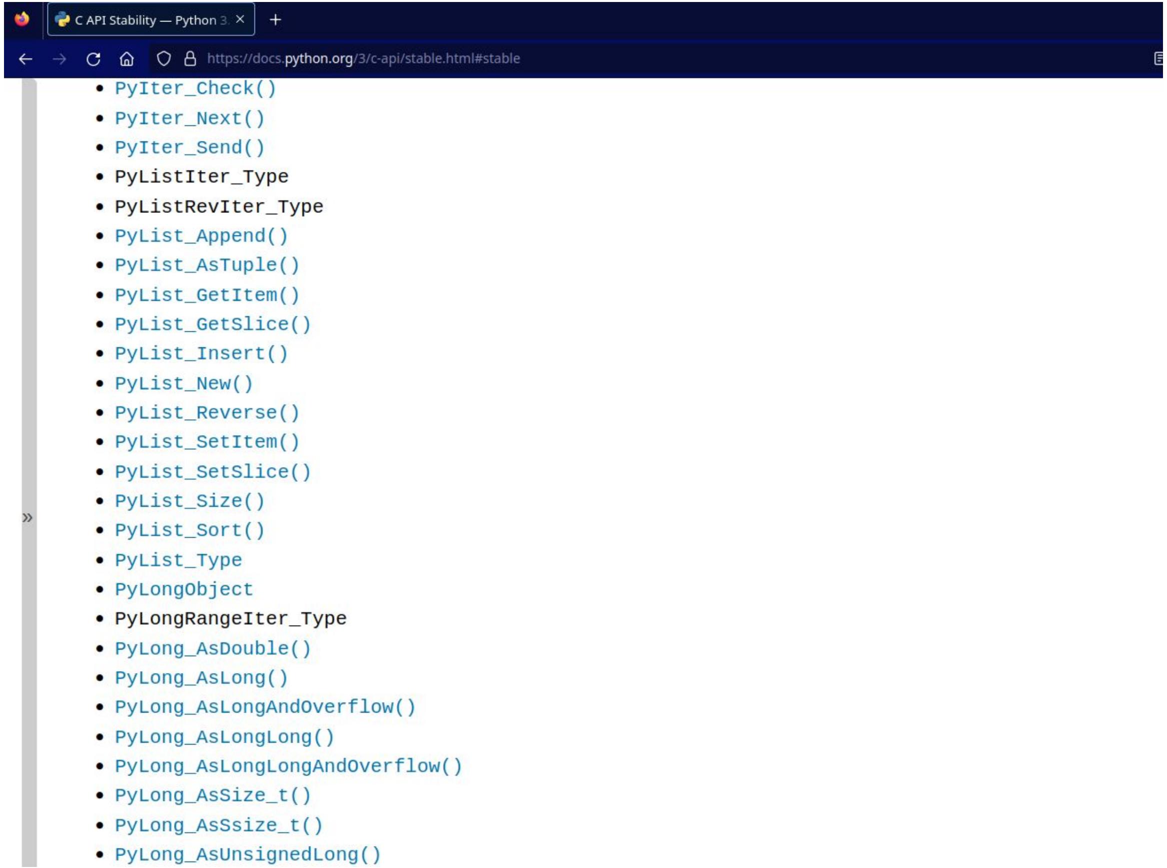
```c
#include <stdio.h>

int main()
{

    // prints hello world
    printf("Hello World");

    return 0;
}
```



C is faster…let's use C - Python C API



C is faster…let's use C - Python C API



C is faster…let's use C - Python C API

https://github.com/pytroll/aggdraw/blob/main/aggdraw.cxx

main

aggdraw / aggdraw.cxx

Code Blame 2726 lines (2359 loc) · 84.6 KB

```
static PyObject*
draw_new(PyObject* self_, PyObject* args)
{
    char buffer[10];
    int ok;

    PyObject* image;
    char* mode;
    int xsize, ysize;
    PyObject* background = NULL;

    if (PyArg_ParseTuple(args, "O:Draw", &image)) {

        /* get mode (use a local buffer to avoid GC issues) */
        PyObject* mode_obj = PyObject_GetAttrString(image, "mode");
        if (!mode_obj)
            return NULL;
        if (PyBytes_Check(mode_obj)) {
            strncpy(buffer, PyBytes_AS_STRING(mode_obj), sizeof buffer);
            buffer[sizeof(buffer)-1] = '\0'; /* to be on the safe side */
            mode = buffer;
        } else if (PyUnicode_Check(mode_obj)) {
            PyObject* ascii_mode = PyUnicode_AsASCIIString(mode_obj);
            if (ascii_mode == NULL) {
                mode = NULL;
            } else {
```

Others ways to combine C and Python

- ctypes
- cffi
- others…
- Cython!

Cython...Do you mean Python? No!

- Programming language and Tool
- Transpiles Python-like Cython to C (or C++)
  - C is then compiled to an importable shared library
- Write your own code or interface with external C/C++ libraries
- Mix python and C/C++ operations in the same code
- Cython language (.pyx) files or decorations/annotations in Python
- Low-level control
  - Global interpreter lock
  - Treat exceptions as integers

Example 1 - _fast_stuff.pyx - source

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

    if nb_primes > 1000:
        nb_primes = 1000

    len_p = 0  # The current number of elements in p.

    n = 2
    while len_p < nb_primes:
        for i in p[:len_p]:
            # Is n not prime?
            if n % i == 0:
                break
        else:
            # If no break occurred in the loop, we have a prime.
            p[len_p] = n
            len_p += 1
        n += 1

    result_as_list = [prime for prime in p[:len_p]]
    return result_as_list
```

Example 1 - _fast_stuff.pyx - cythonize

```
$ cythonize -3 _fast_stuff.pyx
Compiling /tmp/_fast_stuff.pyx because it changed.
[1/1] Cythonizing /tmp/_fast_stuff.pyx

$ head _fast_stuff.c
/* Generated by Cython 3.0.6 */

/* BEGIN: Cython Metadata
{
        "distutils": {
        "name": "_fast_stuff",
```

Example 1 - _fast_stuff.c - source 1

```
static PyObject *__pyx_pf_11_fast_stuff_primes(CYTHON_UNUSED PyObject *__pyx_self, int __pyx_v_nb_primes) {
  int __pyx_v_n;
  int __pyx_v_i;
  int __pyx_v_len_p;
  int __pyx_v_p[0x3E8];
  PyObject *__pyx_v_result_as_list = NULL;
  int __pyx_7genexpr__pyx_v_prime;
  PyObject *__pyx_r = NULL;
  __Pyx_RefNannyDeclarations
  int __pyx_t_1;
  int *__pyx_t_2;
  int *__pyx_t_3;
  int *__pyx_t_4;
  PyObject *__pyx_t_5 = NULL;
  PyObject *__pyx_t_6 = NULL;
  int __pyx_lineno = 0;
  const char *__pyx_filename = NULL;
  int __pyx_clineno = 0;
  __Pyx_RefNannySetupContext("primes", 1);
```

Example 1 - _fast_stuff.c - source 2

```
/* "_fast_stuff.pyx":12
*
*     n = 2
*     while len_p < nb_primes:              # <<<<<<<<<<<<<<
*         for i in p[:len_p]:
*             # Is n not prime?
*/
 while (1) {
  __pyx_t_1 = (__pyx_v_len_p < __pyx_v_nb_primes);
  if (!__pyx_t_1) break;

  /* "_fast_stuff.pyx":13
*     n = 2
*     while len_p < nb_primes:
*         for i in p[:len_p]:              # <<<<<<<<<<<<<<
*             # Is n not prime?
*             if n % i == 0:
*/
  __pyx_t_3 = (__pyx_v_p + __pyx_v_len_p);
  for (__pyx_t_4 = __pyx_v_p; __pyx_t_4 < __pyx_t_3; __pyx_t_4++) {
   __pyx_t_2 = __pyx_t_4;
   __pyx_v_i = (__pyx_t_2[0]);
```

Example 1 - _fast_stuff.pyx - cythonize annotated

```
$ cythonize -3 -a -f _fast_stuff.pyx
[1/1] Cythonizing /tmp/_fast_stuff.pyx
$ firefox _fast_stuff.html
```

Generated by Cython 3.0.6

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

Raw output: _fast_stuff.c

```
 01:
+02: def primes(int nb_primes):
 03:     cdef int n, i, len_p
 04:     cdef int[1000] p
 05:
+06:     if nb_primes > 1000:
+07:         nb_primes = 1000
 08:
+09:     len_p = 0  # The current number of elements in p.
 10:
+11:     n = 2
+12:     while len_p < nb_primes:
+13:         for i in p[:len_p]:
 14:             # Is n not prime?
+15:             if n % i == 0:
+16:                 break
 17:         else:
 18:             # If no break occurred in the loop, we have a prime.
+19:             p[len_p] = n
+20:             len_p += 1
+21:         n += 1
 22:
+23:     result_as_list = [prime for prime in p[:len_p]]
+24:     return result_as_list
```

Example 1 - _fast_stuff.pyx - cythonize annotated

```
    while len_p < nb_primes:
        for i in p[:len_p]:
            # Is n not prime?
            if n % i == 0:
      if (unlikely(__pyx_v_i == 0)) {
        PyErr_SetString(PyExc_ZeroDivisionError, "integer division or modulo by zero");
        __PYX_ERR(0, 15, __pyx_L1_error)
      }
      __pyx_t_1 = (__Pyx_mod_int(__pyx_v_n, __pyx_v_i) == 0);
      if (__pyx_t_1) {
/* … */
      }
    }
    /*else*/ {
                break
        else:
```

Example 1 - _fast_stuff.pyx - directives

Compiler directives

Compiler directives are instructions which affect the behavior of Cython code. Here is the list of currently supported directives:

`binding` (True / False)

Controls whether free functions behave more like Python's CFunctions (e.g. **`len()`**) or, when set to True, more like Python's functions. When enabled, functions will bind to an instance when looked up as a class attribute (hence the name) and will emulate the attributes of Python functions, including introspections like argument names and annotations.

Default is True.

Changed in version 3.0.0: Default changed from False to True

`boundscheck` (True / False)

If set to False, Cython is free to assume that indexing operations ([]-operator) in the code will not cause any IndexErrors to be raised. Lists, tuples, and strings are affected only if the index can be determined to be non-negative (or if `wraparound` is False). Conditions which would normally trigger an IndexError may instead cause segfaults or data corruption if this is set to False. Default is True.

`wraparound` (True / False)

In Python, arrays and sequences can be indexed relative to the end. For example, A[-1] indexes the last value of a list. In C, negative indexing is not supported. If set to False, Cython is allowed to neither check for nor correctly handle negative indices, possibly causing segfaults or data corruption. If bounds checks are enabled (the default, see `boundschecks`

Example 1 - _fast_stuff.pyx - Compiling

setup.py

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

setup(
    ext_modules = cythonize("_fast_stuff.pyx", language_level="3")
)
```

```
$ python setup.py build_ext --inplace
Compiling _fast_stuff.pyx because it changed.
[1/1] Cythonizing _fast_stuff.pyx
running build_ext
building '_fast_stuff' extension
gcc -pthread -B <env>/compiler_compat -fno-strict-overflow -DNDEBUG -O2 -Wall -fPIC -O2 -isystem <env>/include -fPIC -O2
-isystem <env>/include -fPIC -I<env>/include/python3.12 -c _fast_stuff.c -o build/temp.linux-x86_64-cpython-312/_fast_stuff.o
gcc -pthread -B <env>/compiler_compat -shared -Wl,--allow-shlib-undefined -Wl,-rpath,<env>/lib -Wl,-rpath-link,<env>/lib
-L<env>/lib -Wl,--allow-shlib-undefined -Wl,-rpath,/<env>/lib -Wl,-rpath-link,<env>/lib -L<env>/lib
build/temp.linux-x86_64-cpython-312/_fast_stuff.o -o
build/lib.linux-x86_64-cpython-312/_fast_stuff.cpython-312-x86_64-linux-gnu.so
copying build/lib.linux-x86_64-cpython-312/_fast_stuff.cpython-312-x86_64-linux-gnu.so ->
```

Example 1 - _fast_stuff.pyx - Importing

```
$ ipython

In [1]: import _fast_stuff

In [2]: _fast_stuff.primes(100)
Out[2]:
[2,
 3,
 5,
```

Not just speed - Memory too

- Can connect to numpy C API
- Can work on array "views"
- Can work on per-pixel calculations without temporary arrays

```
mask = a > 2
result[mask] = np.sqrt(a[mask] ** 2 + b[mask] ** 2)
```

Example 2 - _fast_arrays.pyx

```
cimport cython
from cython cimport floating

from libc.math cimport sqrt, round
cimport numpy as np
import numpy as np

np.import_array()


def do_something(np.ndarray[floating, ndim=1] arr1, np.ndarray[floating, ndim=1] arr2):
    cdef int res
    cdef floating[::1] a1_view = arr1
    cdef floating[::1] a2_view = arr2
    with nogil:
        res = _do_something(a1_view, a2_view)
    return res
```

Example 2 - _fast_arrays.pyx

```
@cython.boundscheck(False)
cdef int _do_something(floating[::1] arr1, floating[::1] arr2) noexcept nogil:
    cdef Py_ssize_t i, j
    cdef int total = 0
    cdef int tmp

    for i in range(arr1.shape[0]):
        for j in range(arr2.shape[0]):
            if arr1[i] > arr2[j]:
                continue
            tmp = int(round(sqrt(arr1[i]**2 + arr2[j]**2)))
            if tmp > 10:
                total += tmp
    return total
```

Example 2 - _fast_arrays.pyx

```
from setuptools import setup
from Cython.Build import build_ext
from Cython.Distutils import Extension

import numpy as np

setup(
    cmdclass={"build_ext": build_ext},
    ext_modules=[
    Extension(
        name="_fast_arrays",
        sources=["_fast_arrays.pyx"],
        cython_directives={"language_level": "3"},
        define_macros=[("NPY_NO_DEPRECATED_API", "NPY_1_7_API_VERSION")],
        include_dirs=[np.get_include()]
    )
    ],
)
```

Example 2 - quick test

```python
import _fast_arrays
import numpy as np

a = np.arange(5.0)
b = np.arange(30.0)
print(_fast_arrays.do_something(a, b))
# Result: 1916
```

Example 2 - slow_arrays.py - just a few elements

```python
import numpy as np

def do_something(a, b):
    a_grid, b_grid = np.meshgrid(a, b)
    tmp = np.sqrt(a_grid ** 2 + b_grid ** 2).round().astype(np.int64)
    tmp[tmp <= 10] = 0
    return tmp.sum()
```

```
In [5]: a = np.arange(5.0)

In [6]: b = np.arange(30.0)

In [9]: %%timeit
   ...: slow_arrays.do_something(a, b)
27.5 µs ± 392 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)

In [10]: %%timeit
    ...: _fast_arrays.do_something(a, b)
1.78 µs ± 28.8 ns per loop (mean ± std. dev. of 7 runs, 1,000,000 loops each)
```

Example 2 - slow_arrays.py - more!

```python
import numpy as np

def do_something(a, b):
    a_grid, b_grid = np.meshgrid(a, b)
    tmp = np.sqrt(a_grid ** 2 + b_grid ** 2).round().astype(np.int64)
    tmp[tmp <= 10] = 0
    return tmp.sum()
```

```
In [11]: a = np.random.random(10000) * 100

In [12]: b = np.random.random(20000) * 100

In [13]: %%timeit
    ...: slow_arrays.do_something(a, b)
2.09 s ± 34 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [14]: %%timeit
    ...: _fast_arrays.do_something(a, b)
797 ms ± 10.3 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Example 2 - slow_arrays.py - MORE!

```
import numpy as np

def do_something(a, b):
    a_grid, b_grid = np.meshgrid(a, b)
    tmp = np.sqrt(a_grid ** 2 + b_grid ** 2).round().astype(np.int64)
    tmp[tmp <= 10] = 0
    return tmp.sum()
```

```
In [16]: a = np.random.random(10000) * 100

In [17]: b = np.random.random(100000) * 100

In [18]: %%timeit
    ...: slow_arrays.do_something(a, b)
24 s ± 4.34 s per loop (mean ± std. dev. of 7 runs, 1 loop each)

In [19]: %%timeit
    ...: _fast_arrays.do_something(a, b)
4.05 s ± 47.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

~25GB memory

Cython - Pros and Cons

- Pros
  - Fast compared to pure Python (ex. loops)
  - More memory efficient for some work loads
  - GIL control
  - OpenMP compatibility for C-level multi-threading
  - Build-time dependency (versus runtime)
  - No runtime overhead
- Cons
  - Not as fast as numpy if calculation is numpy-friendly
    - vectorized and limited temporary arrays
    - Numpy has SIMD (Single Instruction, Multiple Data) optimizations
  - Much more complicated build process than pure python
    - **sdist (.tar.gz)** - typically .py only + metadata [+ .pyx]
    - **binary wheels (.whl)** - .py + .so + metadata

Bonus Topics and Questions

- GIL and Dask
  - python-geotiepoints GIL changes
  - Pyresample gradient_search GIL bug fix
- Extensions in other languages
  - pyo3 - rust + python

Questions?

Example 3 - _fast_arrays.rs (src/lib.rs) - 1

```rust
use numpy::{PyArray1, IntoPyArray};
use numpy::ndarray::{ArrayView1};
use numpy::{PyReadonlyArray1};
use pyo3::prelude::{pyfunction, pymodule, Py, PyModule, PyResult, Python};
use pyo3::wrap_pyfunction;

#[pyfunction]
fn do_something(py: Python, arr1: PyReadonlyArray1<f64>, arr2:
PyReadonlyArray1<f64>) -> PyResult<u64> {
     let arr1 = arr1.as_array();
     let arr2 = arr2.as_array();
     Ok(_do_something(arr1, arr2))
}

/// A Python module implemented in Rust.
#[pymodule]
fn rust_arrays(_py: Python, m: &PyModule) -> PyResult<()> {
     m.add_function(wrap_pyfunction!(do_something, m)?)?;
     Ok(())
}
```

Example 3 - _fast_arrays.rs (src/lib.rs) - 2

```
fn _do_something(arr1: ArrayView1<f64>, arr2: ArrayView1<f64>) -> u64 {
    let mut total: u64 = 0;
    for &val1 in arr1.iter() {
        for &val2 in arr2.iter() {
            if val1 > val2 {
                continue;
            }
            let tmp = ((val1.powi(2) + val2.powi(2)).sqrt().round() as u64);
            if tmp > 10 {
                total = total.wrapping_add(tmp);
            }
        }
    }
    total
}
```

Example 3 - Cargo.toml (from "maturin init")

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

# See more keys and their definitions at
https://doc.rust-lang.org/cargo/reference/manifest.html
[lib]
name = "rust_arrays"
crate-type = ["cdylib"]

[dependencies]
pyo3 = "0.20.0"
numpy = "0.20"
ndarray = "0.15.6"
```

Example 3 - pyproject.toml (from "maturin init")

```toml
[build-system]
requires = ["maturin>=1.4,<2.0"]
build-backend = "maturin"

[project]
name = "rust_arrays"
requires-python = ">=3.8"
classifiers = [
    "Programming Language :: Rust",
    "Programming Language :: Python :: Implementation :: CPython",
    "Programming Language :: Python :: Implementation :: PyPy",
]
dynamic = ["version"]

[tool.maturin]
features = ["pyo3/extension-module"]
```

Example 3 - Compile

```
maturin develop --profile release
```

Example 3 - Import

```
In [1]: import rust_arrays

In [2]: import numpy as np

In [3]: a = np.random.random(10000) * 100

In [4]: b = np.random.random(20000) * 100

In [5]: %%timeit
   ...: rust_arrays.do_something(a, b)
1.09 s ± 38.5 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```