Workshop: School of Data Science and Computation Thinking, Stellenbosch University

Introduction to Parallel Computation with GPUs: Programming with JAX, NUMBA, THRUST, and CUDA

Martin BUCHER
(bucher@sun.ac.za)

LECTURE 7 (11 March 2025)

Course dates (4-5pm SAST):

Tu 18 Feb	Th 20 Feb
Tu 25 Feb	Th 27 Feb
Tu 4 March	Th 6 March
Tu 11 March	Th 13 March

Zoom link:

`https://u-paris.zoom.us/j/82740807191?pwd=8yAGlP3R6VKbsjNCIeRjJU3u0AQtU6.1`

Course webpage:

`http://www.sun.ac.za/english/data-science-and-computational-thinking/parallelcomputation`
`https://github.com/martinabucher/cudaCourse`

Lecture 7 Outline

1. Why Native Python is Slow
2. How Python Data is Represented Internally
3. numpy and Cython
4. Dynamic Linking
5. Some functional programming in Python
6. Python Decorators
7. Example: Timing with Decorators
8. NUMBA JIT Example
9. How NUMBA JIT actually works
10. More Details on NUMBA JIT
11. GIL (Global Interpreter Lock)

How Python Works

- Python code as executed by the interpreter is incredibly slow. Lines of code are executed one by one through an in principle possibly infinite Read–Eval–Print loop. There is lots of overhead to this approach.

  `https://en.wikipedia.org/wiki/Read%E2%80%93eval%E2%80%93print_loop`

- Native C/C++ code is much faster because
  - Entire functions may be analyzed as a unit
  - Intermediate type information need not be carried along.
  - Instructions may be reordered and made to execute in parallel, even on a single CPU. [N.B. A modern Intel CPU can execute up the 32 floating point operations in parallel]
- There are lots of nice things in Python, which is sometimes described as a productivity language (together with Matlab, Python with the numpy library, and Julia) as opposed to an "efficiency language" (C, C++, Rust, ...).
- The C/C++ philosphy is to make no compromises that would make C/C++ codes significantly slower than would be possible with assembly (almost machine) code. Little effort is devoted to making compilation fast.
- A Python script makes it easy to examine intermediate results. They can for example be plotted from the command line, and variables (in the same scope) remain in the namespace (until they fall out of scope). Thus Python is a great language for **prototyping** and **trying out new ideas**.

How Python works (II)

- The REPL description above is partially an oversimplification and for many problems exagerates the slowness of Python. The `numpy` library is a prime example how linear algebra can be performed at C speed within a Python code.
- `numpy` arrays are essentially C arrays, with data stored contiguously in memory (as it would be in an array created with malloc in a C program) but wrapped with some additional information, dimension, stride, datatype,....
- We could represent an array in native Python as a list (in the case of a one-dimensional array), or as a list of lists (in the case of a two-dimensional array).

An example of incredibly slow native Python

We use a list of lists (i.e., nested lists) to represent a square matrix:

```
a=[[1, 2, 3],
   [4, 5, 6],
   [7, 8, 9]]

def listBasedMatMul(a, b):
  def getElement(a,b,i,j,n):
     sum=0.
     for k in range(n):
       sum+= (a[i])[k] *(b[k])[j]
     return sum;
  dimArows=len(a)
  dimAcolumns=len(a[0])
  dimBrows=len(b)
  dimBcolumns=len(b[0])
  n=dimAcolumns
  result=[ [ getElement(a,b,i,j,n) for j in range(dimBcolumns)
           ]   for i in range(dimArows)
         ]
  return(result)

print(listBasedMatMul(a, a))
```

Why is Native Python So Slow?

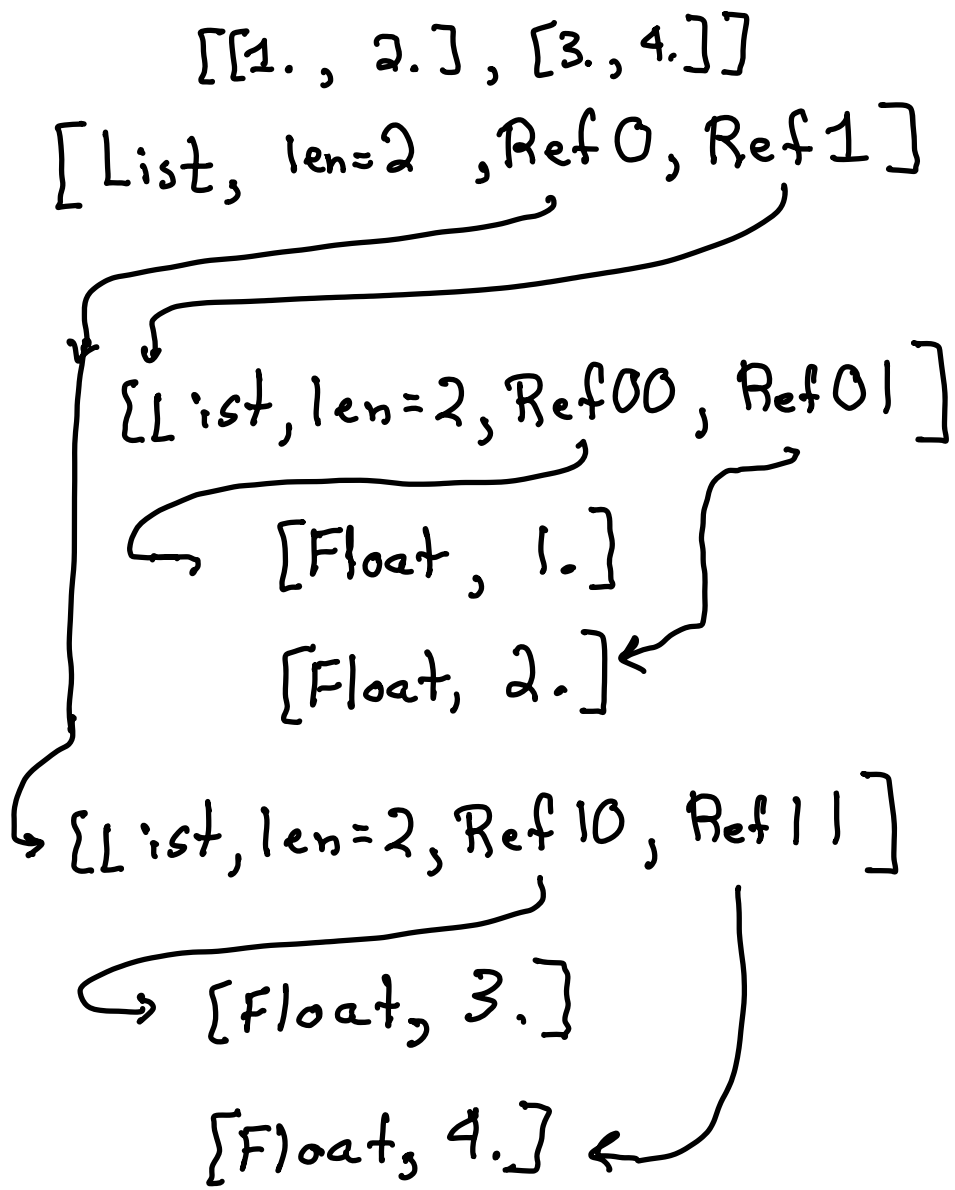
Python's lists are very flexible. Elements of the same list may have heterogeneous types.

```
[a, b]
```

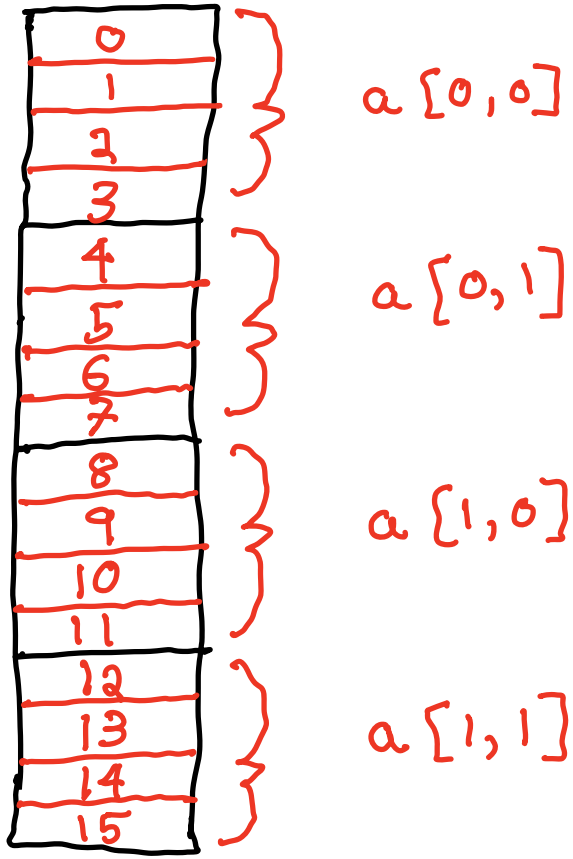
```
List, length= 2, Reference to object a, Reference to object b.
```

Internally lists in Python are more like vectors in the C++ Standard Library, although from the name and methods defined on them one might believe that they are like the forward links described in any CS data structures book.

Simplified Python Data Representation



C/C++ Representation



Compiler must keep track of types and structure in memory.

Lists in CPython

```
typedef struct {
    PyObject_HEAD
    Py_ssize_t ob_size;

    /* Vector of pointers to list elements.  list[0] is ob_item[0], etc. */
    PyObject **ob_item;

    /* ob_item contains space for 'allocated' elements.  The number
     * currently in use is ob_size.
     * Invariants:
     *     0 <= ob_size <= allocated
     *     len(list) == ob_size
     *     ob_item == NULL implies ob_size == allocated == 0
     */
    Py_ssize_t allocated;
} PyListObject;
```

https://stackoverflow.com/questions/3917574/how-is-pythons-list-implemented

https://blog.codingconfessions.com/p/cpython-object-system-internals-understanding

For the Curious: Python Interpreter Source Code

The CPython reference implementation is available on github for inspection:

`https://github.com/python/cpython/blob/d57f8a9f76e75384ec997686c2a826b1dc3c69c4/Include/pytypedefs.h`

`https://github.com/python/cpython/blob/72e5b25efb580fb1f0fdfade516be90d90822164/Include/object.h`

How dynamic loading works: A simple example

```
//POSIX LINUX DYNAMIC LOADING
//https://pubs.opengroup.org/onlinepubs/7908799/xsh/dlfcn.h.html
#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h>
#include <gnu/lib-names.h>

int main(int argc, char **argv){
   void *handle;
   double (*sine)(double);
   char *error;
   handle = dlopen(LIBM_SO, RTLD_LAZY);
    if (!handle) {
        fprintf(stderr, "%s\n", dlerror());
        exit(EXIT_FAILURE);}
   dlerror();
   *(void **) (&sine) = dlsym(handle, "sin");
   if ((error = dlerror()) != NULL)  {
        fprintf(stderr, "%s\n", error);
        exit(EXIT_FAILURE); }

   printf("%f\n", (*sine)(4.0));
    dlclose(handle);
    exit(EXIT_SUCCESS);
}
```

Pointers From Hell Explained

```
double (*sine)(double);

*(void **) (&sine) = dlsym(handle, "sin");

The type of sine is a pointer to a function.

&sine is a pointer to a pointer of a function

(**void) is a cast to a pointer to a void pointer.

((**void) (&sine)) is a pointer to a void pointer.

*( (**void) (&sin) ) = dlsym(handle, "sin");

puts the void pointer returned by dlsym(handle, "sin");

Libraries can be loaded while the program is running.

https://linuxhint.com/linux-dlopen-system-call-c/
```

A Simple CTypes Example: Calling C Functions From Python

```
$cat sum.c
int our_function(int num_numbers, int *numbers) {
    int i;
    int sum = 0;
    for (i = 0; i < num_numbers; i++) {
        sum += numbers[i];
    }
    return sum;
}
```

Compile into a dynamic library as follows:

```
$cc -fPIC -shared -o libsum.so sum.c
```

A Simple Example with CTypes: Calling C Functions From Python (II)

On the Python side, we do as follows:

```
import ctypes

_sum = ctypes.CDLL('libsum.so')
_sum.our_function.argtypes = (ctypes.c_int, ctypes.POINTER(ctypes.c_int))

def our_function(numbers):
    global _sum
    num_numbers = len(numbers)
    array_type = ctypes.c_int * num_numbers
    result = _sum.our_function(ctypes.c_int(num_numbers), array_type(*numbers))
    return int(result)
```

Reference for this example:

https://pgi-jcns.fz-juelich.de/portal/pages/using-c-from-python.html

and more complete documentation

https://docs.python.org/3/library/ctypes.html

Converting a Python array to a ctypes Array

As we discussed Python arrays (lists, tuples, etc.) have a different structure from C arrays, so a conversion is necessary:

```
pyarr = [1, 2, 3, 4]
seq = ctypes.c_int * len(pyarr)
arr = seq(*pyarr)
```

ctypes are defined for a large number of primitive Python data types for which adapters can be written based on the above pattern.

More complex Python data types are trickier and are discussed in the documentation.

NUMPY: A Pythonization of Arrays

- As we saw, Python can call C functions either through shared libraries loaded by the python interpreter at startup, or other shared libraries loaded later, using dynamically linked libraries. In this sense, there is nothing that C can do that Python cannot once appropriate libraries have been created and plumbing between Python data types and C data types have worked out.
- This is simple for simple data types. For more complicated data types, this is much trickier. Python classes are quite different from C/C++ data types.
- In much numerical code, data structures are rather trivial: the time-consuming operations are array and linear algebra operations. numpy provides a standardized interface to to array operations.
- Python arrays (whose internal structure we shall examine below) are much more restrictive than native python containers (eg lists) which may contain heterogeneous types. This is because Python arrays are pointer (or reference) arrays. The array elements are Python objects are stored directly in the ndarray object with a minimum of indirection.

`https://web.mit.edu/dvp/Public/numpybook.pdf`

NUMPY Internal Structure

- In the structure definition below, `char*` is legacy C for a void pointer, or `void*`.
- The pointer `data` points to the data of the array, stored coninguously in memory without any excess baggage (e.g., Python objects). This data is generally accessed only indirectly, through numpy functions and methods, which can use C code, and also hardware specific BLAS calls to achieve maximum speed.

```
typedef struct PyArrayObject {
    PyObject_HEAD
    char *data;
    int nd;
    npy_intp *dimensions;
    npy_intp *strides;
    PyObject *base;
    PyArray_Descr *descr;
    int flags;
    PyObject *weakreflist;
    /* version dependent private members */
} PyArrayObject;
```

A Short Digression: Functional Programming in Python

Can a program return a function?

```
def parent(num):
    def first_child():
        return "Hi, I'm Elias"
    def second_child():
        return "Call me Ester"
    if num == 1:
        return first_child
    else:
        return second_child
```

The above program maps integers into a void function with a side effect, namely printing a message.

Python (unlike C and C++) allows functions to be defined within a function. In many cases, this is merely a technique for organizing the namespaces, to limit the scope of function names, prevent naming conflicts, and add clarity by indicating which functions are helper functions and which are not.

A Short Digression: Functional Programming in Python (cont.)

```
def parent(num):
    def first_child():
        return "Hi, I'm Elias"
    def second_child():
        return "Call me Ester"
    if num == 1:
        return first_child
    else:
        return second_child
```

Function thus returned may be assigned

```
In [4]: myFunc0=parent(0)

In [5]:  myFunc1=parent(1)
```

and called as if they were normal functions

```
In [6]:  myFunc0()
Out[6]: 'Call me Ester'

In [7]:  myFunc1()
Out[7]: "Hi, I'm Elias"
```

Wrapping Functions and Python "Decorators" (I)

We show how a function mapping void functions to void functions can be used to wrap functions. For simplicity, we avoid the issue of input and output arguments.

```
def myDecorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def praisePython():
    print("Python is the greatest computer language of all times.")
```

The above gives

```
In [2]: (myDecorator(praisePython))()
Something is happening before the function is called.
Python is the greatest computer language of all times.
Something is happening after the function is called.
```

Wrapping Functions and Python "Decorators" (II)

Here is another nicer looking, more compact way of doing the same:

```
$ ipython
Python 3.10.15 (main, Oct  3 2024, 02:24:49) [Clang 14.0.6 ]

In [1]: def myDecorator(func):
   ...:     def wrapper():
   ...:         print("Something is happening before the function is called.")
   ...:         func()
   ...:         print("Something is happening after the function is called.")
   ...:     return wrapper

In [2]: @myDecorator
   ...: def praisePython():
   ...:     print("Python is the greatest computer language of all time.")

In [3]: praisePython()
Something is happening before the function is called.
Python is the greatest computer language of all time.
Something is happening after the function is called.
```

The syntax `myDecorator` provides a shorter way of function wrapping.

Some Computer Science Jargon

What is "syntactic sugar"?

*For understanding computer language, it is useful to reduce language features to a bare minimum. Often there are alternative ways of expressing the same thing, making for easier to read (or to write) programs, but do not add any new functionality. This is often known as "syntactic sugar." Python decorators are one such example. Another in C is a[n] in the place of *(a+n), which is less human readable.*

What is meant by "functions as first-class citizens?

In early programming languages functions or procedures or subroutines were on a different footing than other variable types. Functions that return functions, write new functions, or allow functions as input arguments were not a supported functionality.

See: https://docs.python.org/3/library/functools.html

A More Useful Example: A Technique for Timing Functions in Python (I)

```
$cat decorators.py
//https://realpython.com/primer-on-python-decorators/#timing-functions
import functools
import time

def timer(func):
    """Print the runtime of the decorated function"""
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()
        value = func(*args, **kwargs)
        end_time = time.perf_counter()
        run_time = end_time - start_time
        print(f"Finished {func.__name__}() in {run_time:.4f} secs")
        return value
    return wrapper_timer
```

The functools allows argument passing.

A More Useful Example: A Technique for Timing Functions in Python (II)

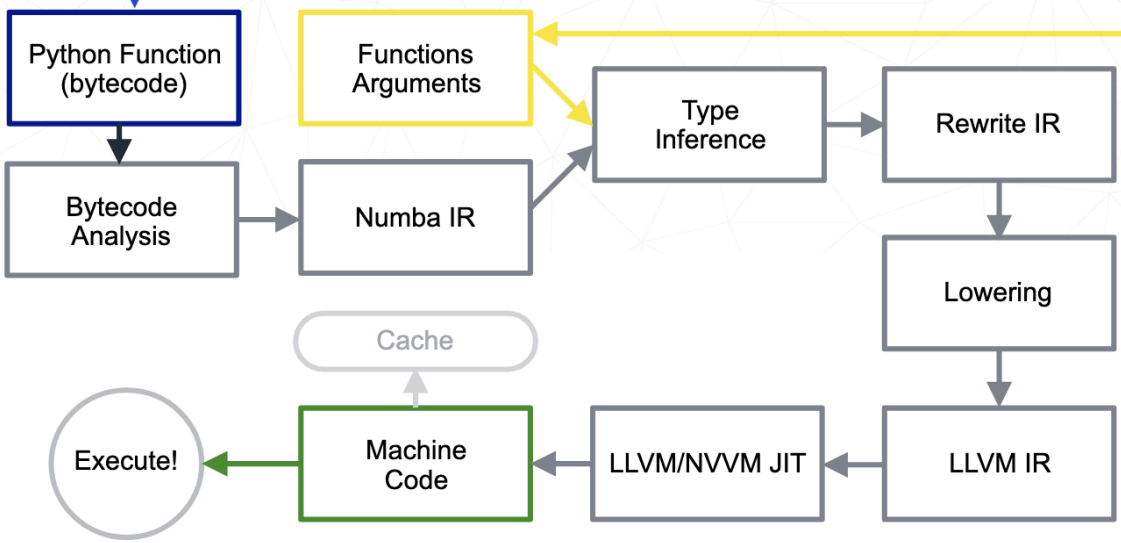
```
>>> from decorators import timer

>>> @timer
... def waste_some_time(num_times):
...     for _ in range(num_times):
...         sum([number**2 for number in range(10_000)])
...

>>> waste_some_time(1)
Finished waste_some_time() in 0.0010 secs

>>> waste_some_time(999)
Finished waste_some_time() in 0.3260 secs
```

A simplest NUMBA JIT example

```
# https://numba.pydata.org/numba-doc/dev/user/5minguide.html
from numba import jit
import numpy as np

x = np.arange(100).reshape(10, 10)

# Set "nopython" mode for best performance, equivalent to @njit
@jit(nopython=True)
def go_fast(a):
    # Function is compiled to machine code when called the first time
    trace = 0.0
    for i in range(a.shape[0]):   # Numba likes loops
        trace += np.tanh(a[i, i]) # Numba likes NumPy functions
    return a + trace              # Numba likes NumPy broadcasting

print(go_fast(x))
```

How does Numba work?

```
@jit
def do_math(a, b):
    ...
>>> do_math(x, y)
```



From talk by Stan Seibert, "Numba: A Compiler for Python Functions", (2018)

`https://indico.cern.ch/event/709711/contributions/2915722/attachments/1638199/2614603/2018_04_23_Numba_DIANA_v2.pdf`

How JIT works in NUMBA: Some Simple Examples

Lazy compilation

[Here lazy means that whatever work is to be done is done at the last minute.]
The recommended way to use the jit decorator is to let Numba decide when and how to optimize:
from numba import jit

```
@jit
def f(x, y):
    # A somewhat trivial example
    return x + y
```

In this mode, compilation will be deferred until the first function execution. Numba infers the argument types at call time and generates optimized code based on this information. Numba can also compile separate specializations depending on the input types. For example, calling the f() function above with integer or complex numbers will generate different code paths:

```
f(1, 2)
3


f(1j, 2)
(2+1j)
```

This page is worth reading very carefully.

`https://numba.readthedocs.io/en/stable/user/jit.html`

`https://en.wikipedia.org/wiki/Just-in-time_compilation`

More Simple Examples

Eager compilation (do every you are able to do right away)
You can also tell Numba the function signature (i.e., input and output types) you are expecting. The function f() would now look like:

```
from numba import jit, int32

@jit(int32(int32, int32))
def f(x, y):
    # A somewhat trivial example
    return x + y
```

A machine code version will be compiled and cached for those input types straightaway.

Caching of compiled functions has several known limitations:

- The caching of compiled functions is not performed on a function-by-function basis. The cached function is the main jit function, and all secondary functions (those called by the main function) are incorporated in the cache of the main function.
- Cache invalidation fails to recognize changes in functions defined in a different file. This means that when a main jit function calls functions that were imported from a different module, a change in those other modules will not be detected and the cache will not be updated. This carries the risk that "old" function code might be used in the calculations.
- Global variables are treated as constants. The cache will remember the value of the global variable at compilation time. On cache load, the cached function will not rebind to the new value of the global variable.

  [https://numba.readthedocs.io/en/stable/user/jit.html]

Caching of compiled functions has several known limitations:

Commentary:

- The default is not to cache functions. This is the safe route absent further information.
- In functional programming, there is much talk about so-called pure functions. A pure function has no side-effects. The function depends only on its input arguments. It does not read its environment (eg through global variable). Nor may it write to global variables. If a pure function is given the input it will always produce the same output. This is not generally true in computer programming. matplotlib for example has a very complicated environment which often is invisible, except when one is not happy with the defaults, and can tweak the environmental variables. It is easy to produce efficient code for pure functions. The compiler need look no further than the function body and does not have to guard against hidden dependencies.
- Even if a function needs to be recompiled at each use, in many cases considerable speedup can be obtained by compiling with the JIT compiler, for example if there are long for loops and often tranversed control structures.

When is Numba a Good Idea?

- Numerical algorithms
- Data is in the form of NumPy arrays, or (more broadly) flat data buffers
- Performance bottleneck is a handful of well encapsulated functions
- Example use cases:
  - Compiling user-defined functions to call from another algorithm (like an optimizer)
  - Creating "missing" NumPy/SciPy functions (librosa)
  - Rapidly prototyping GPU algorithms (FBPIC)
  - Constructing specialized Python compilers (HPAT, OAMap)

From talk by Stan Seibert, "Numba: A Compiler for Python Functions", (2018)

`https://indico.cern.ch/event/709711/contributions/2915722/attachments/1638199/2614603/2018_04_23_Numba_DIANA_v2.pdf`

GIL (Global Interpreter Lock): An Obstacle to Python Concurrency

PEP 703: Making the Global Interpreter Lock Optional in CPython

`https://peps.python.org/pep-0703/`

Abstract

CPython's global interpreter lock ("GIL") prevents multiple threads from executing Python code at the same time. The GIL is an obstacle to using multi-core CPUs from Python efficiently. This PEP proposes adding a build configuration (–disable-gil) to CPython to let it run Python code without the global interpreter lock and with the necessary changes needed to make the interpreter thread-safe.

`https://docs.python.org/3/glossary.html#term-global-interpreter-lock`

NEXT TIME:
CONCURRENCY IN PYTHON