|| & .py

GitHub

Basic Interface

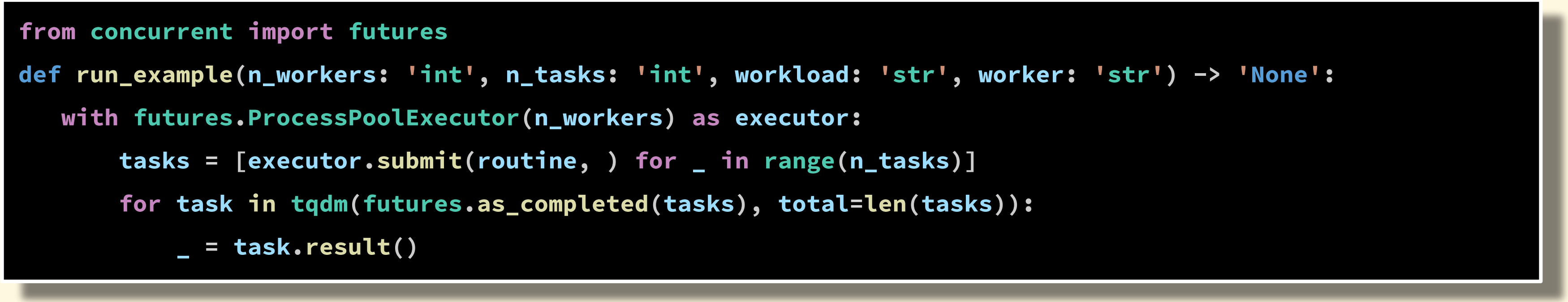
how to create process / thread in python?

- interface of processes
- interface of threads

Processes

If the task is simple, all process management can be left to special packages.

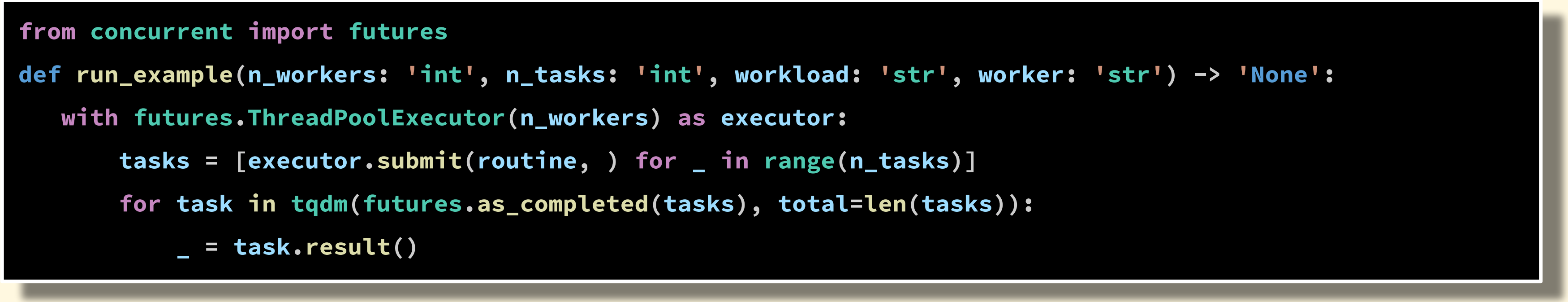
Mechanisms of synchronization between processes will be omitted :(



```python
from concurrent import futures
def run_example(n_workers: 'int', n_tasks: 'int', workload: 'str', worker: 'str') -> 'None':
   with futures.ProcessPoolExecutor(n_workers) as executor:
       tasks = [executor.submit(routine, ) for _ in range(n_tasks)]
       for task in tqdm(futures.as_completed(tasks), total=len(tasks)):
           _ = task.result()
```

Threads

If the task is simple, all process management can be left to special packages.

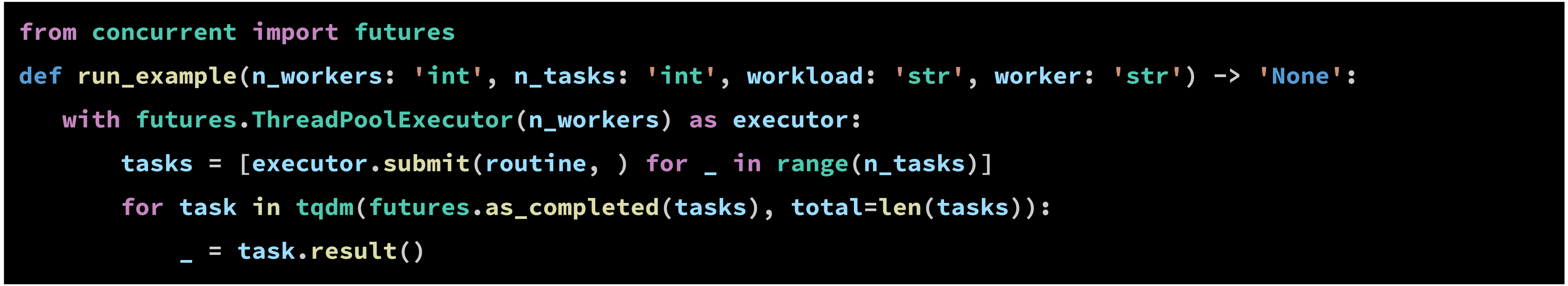


```
from concurrent import futures
def run_example(n_workers: 'int', n_tasks: 'int', workload: 'str', worker: 'str') -> 'None':
   with futures.ThreadPoolExecutor(n_workers) as executor:
       tasks = [executor.submit(routine, ) for _ in range(n_tasks)]
       for task in tqdm(futures.as_completed(tasks), total=len(tasks)):
           _ = task.result()
```

Threads

If the task is simple, all process management can be left to special packages.

Mechanisms of synchronization will be discussed. But later.



```python
from concurrent import futures
def run_example(n_workers: 'int', n_tasks: 'int', workload: 'str', worker: 'str') -> 'None':
    with futures.ThreadPoolExecutor(n_workers) as executor:
        tasks = [executor.submit(routine, ) for _ in range(n_tasks)]
        for task in tqdm(futures.as_completed(tasks), total=len(tasks)):
            _ = task.result()
```

Process
vs
Thread

which one should be used?

- cpu tasks & processes
- io tasks & processes
- etc.
- …
- comparison

?

- Is there true thread parallelism in .py?
- Is there true processes parallelism in .py?

Threads & True ||

```
100%|██████████| 24/24 [01:08<00:00,  2.87s/it]
[# threads = 1, # tasks = 24, workload = io]
Total time 68.7927 s

100%|██████████| 24/24 [00:45<00:00,  1.89s/it]
[# threads = 2, # tasks = 24, workload = io]
Total time 45.3669 s

100%|██████████| 24/24 [00:43<00:00,  1.79s/it]
[# threads = 4, # tasks = 24, workload = io]
Total time 43.0095 s
```

for io bound tasks threads are **useful**

Threads & True ||

```
100%|██████████| 24/24 [00:00<00:00, 45.88it/s]
[# threads = 1, # tasks = 24, workload = cpu]
Total time 0.6458 s


100%|██████████| 24/24 [00:00<00:00, 114.38it/s]
[# threads = 2, # tasks = 24, workload = cpu]
Total time 0.6140 s


100%|██████████| 24/24 [00:00<00:00, 77612.41it/s]
[# threads = 4, # tasks = 24, workload = cpu]
Total time 0.6088 s
```

for cpu bound tasks threads are **useless**

Threads & True ||

```
100%|██████████| 24/24 [00:00<00:00, 45.88it/s]
[# threads = 1, # tasks = 24, workload = cpu]
Total time 0.6458 s


100%|██████████| 24/24 [00:00<00:00, 114.38it/s]
[# threads = 2, # tasks = 24, workload = cpu]
Total time 0.6140 s


100%|██████████| 24/24 [00:00<00:00, 77612.41it/s]
[# threads = 4, # tasks = 24, workload = cpu]
Total time 0.6088 s
```

for cpu bound tasks threads are useless*

Processes & True ||

```
100%|██████████| 24/24 [01:10<00:00,  2.92s/it]
[# processes = 1, workload = io]
Total time 70.1880 s

100%|██████████| 24/24 [00:49<00:00,  2.05s/it]
[# processes = 2, workload = io]
Total time 49.3300 s

100%|██████████| 24/24 [00:32<00:00,  1.35s/it]
[# processes = 4, workload = io]
Total time 32.4546 s
```

for cpu bound tasks threads are **useful**

Processes & True ||

```
100%|██████████| 24/24 [00:00<00:00, 35.59it/s]
[# processes = 1, workload = cpu]
Total time 0.6933 s

100%|██████████| 24/24 [00:00<00:00, 67.80it/s]
[# processes = 2, workload = cpu]
Total time 0.3709 s

100%|██████████| 24/24 [00:00<00:00, 109.61it/s]
[# processes = 4, workload = cpu]
Total time 0.2443 s
```

for cpu bound tasks processes are **useful**

```
100%|██████████| 24/24 [00:00<00:00, 77612.41it/s]
[# threads = 4, # tasks = 24, workload = cpu]
Total time 0.6088 s
```

```
100%|██████████| 24/24 [00:43<00:00,  1.79s/it]
[# threads = 4, # tasks = 24, workload = io]
Total time 43.0095 s
```

```
100%|██████████| 24/24 [00:00<00:00, 109.61it/s]
[# processes = 4, workload = cpu]
Total time 0.2443 s
```

```
100%|██████████| 24/24 [00:32<00:00,  1.35s/it]
[# processes = 4, workload = io]
Total time 32.4546 s
```

And why do we need threads?

```
100%|██████████| 24/24 [00:00<00:00, 77612.41it/s]
[# threads = 4, # tasks = 24, workload = cpu]
Total time 0.6088 s

100%|██████████| 24/24 [00:43<00:00,  1.79s/it]
[# threads = 4, # tasks = 24, workload = io]
Total time 43.0095 s
```

```
100%|██████████| 24/24 [00:00<00:00, 109.61it/s]
[# processes = 4, workload = cpu]
Total time 0.2443 s

100%|██████████| 24/24 [00:32<00:00,  1.35s/it]
[# processes = 4, workload = io]
Total time 32.4546 s
```

And why do we need threads?

```
Total Python's memory: 144.84375 MB
100%|██████████| 10/10
```

i.e. ~ 14mB/process

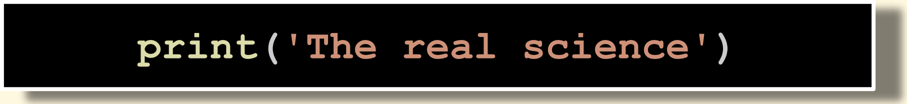
threads and cpu bound tasks

what is the problem?

- execution workflow
- garbage collector
- GIL
- workflow with GIL

Execution workflow

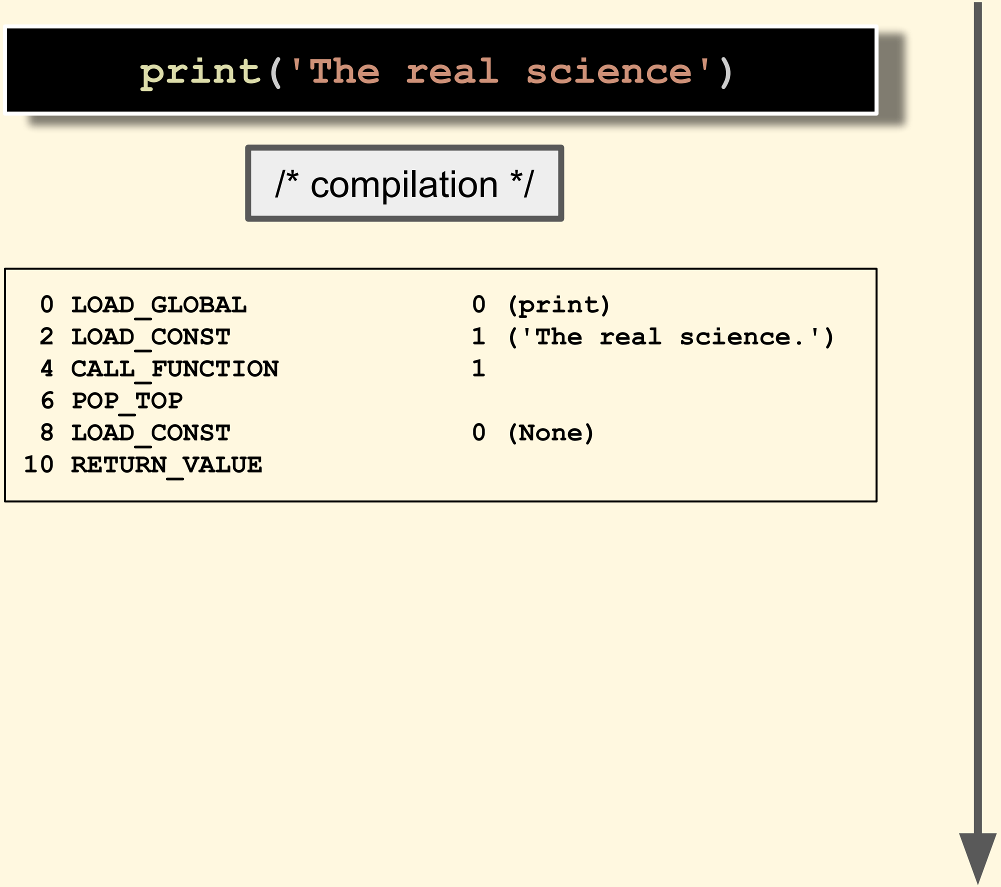
file.py *(source code)*



```
print('The real science')
```

Execution workflow

file.py *(source code)*



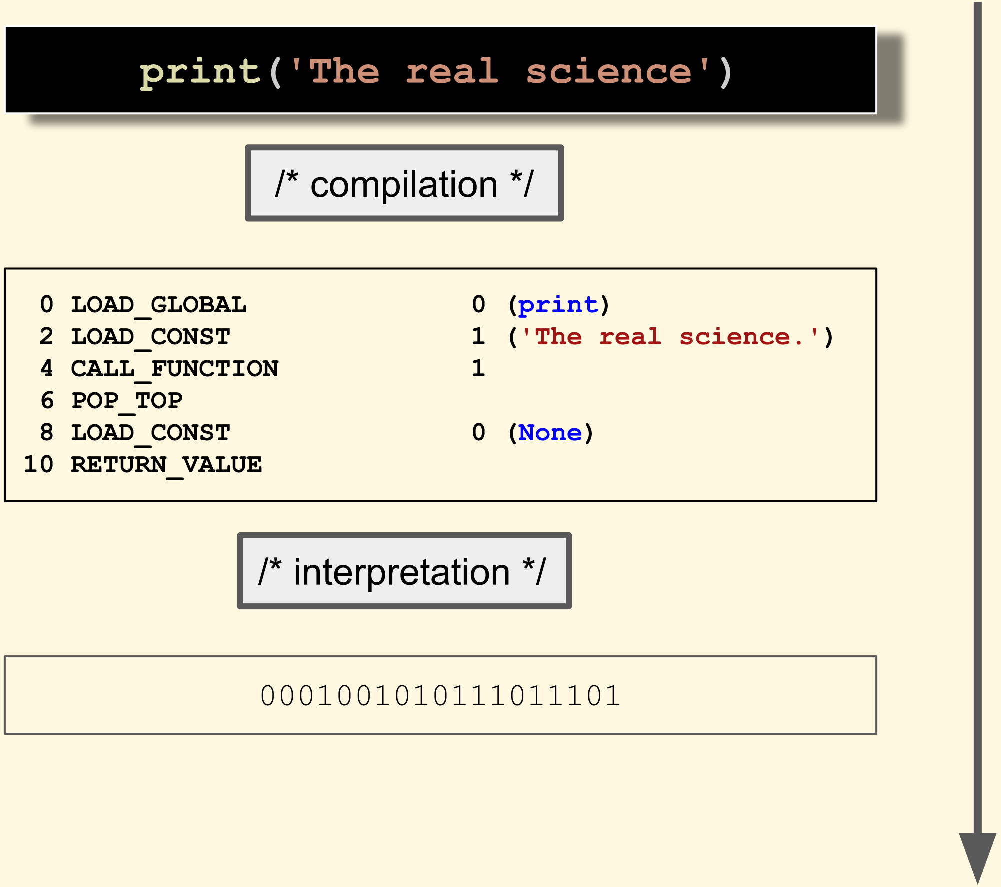
```
print('The real science')
```

/* compilation */

file.pyc *(byte code)*

```
 0 LOAD_GLOBAL              0 (print)
 2 LOAD_CONST               1 ('The real science.')
 4 CALL_FUNCTION            1
 6 POP_TOP
 8 LOAD_CONST               0 (None)
10 RETURN_VALUE
```

Execution workflow



file.py *(source code)*

```
print('The real science')
```

/* compilation */

file.pyc *(byte code)*

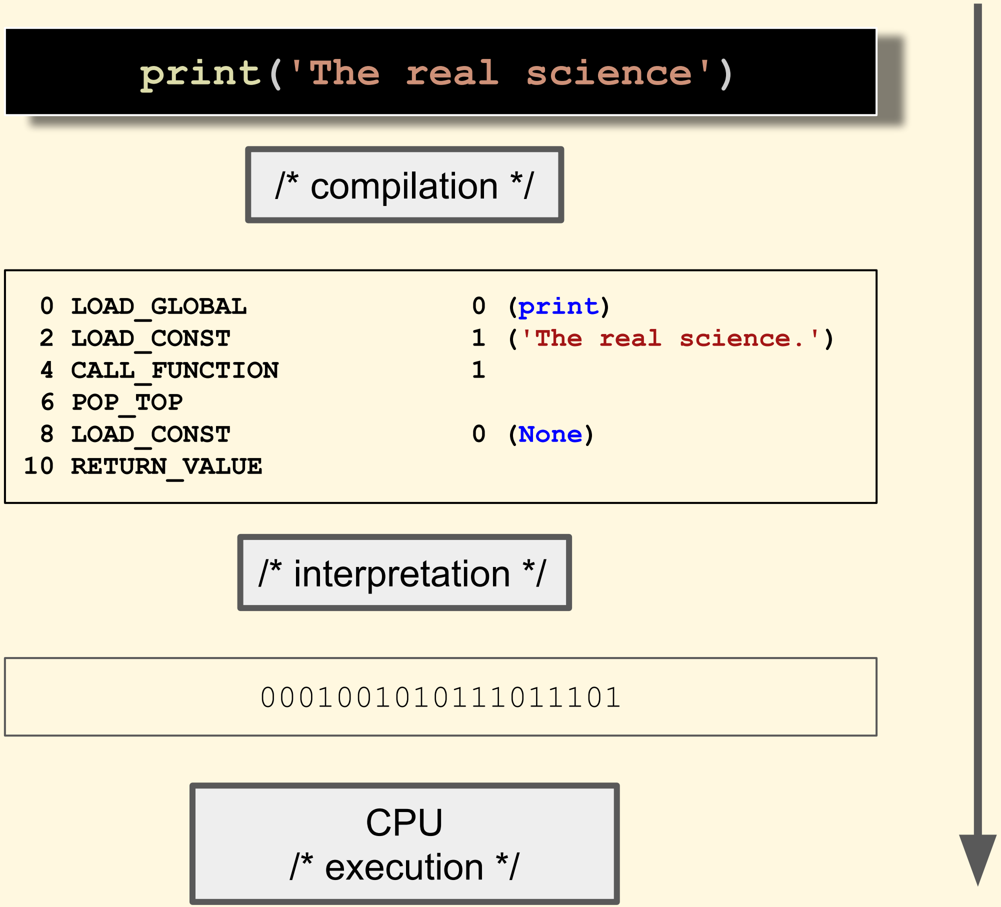
```
 0 LOAD_GLOBAL                  0 (print)
 2 LOAD_CONST                   1 ('The real science.')
 4 CALL_FUNCTION                1
 6 POP_TOP
 8 LOAD_CONST                   0 (None)
10 RETURN_VALUE
```

/* interpretation */

/ ? */ (binary code)*

```
0001001010111011101
```

Execution workflow



file.py *(source code)*

```
print('The real science')
```

/* compilation */

file.pyc *(byte code)*

```
 0 LOAD_GLOBAL                 0 (print)
 2 LOAD_CONST                  1 ('The real science.')
 4 CALL_FUNCTION               1
 6 POP_TOP
 8 LOAD_CONST                  0 (None)
10 RETURN_VALUE
```

/* interpretation */

/ ? */ (binary code)*

```
0001001010111011101
```

CPU
/* execution */

Garbage collection

Garbage is bad

Garbage collection

Garbage is bad

It’s hard to manage it by ourselves

Garbage collection

Garbage is bad

It’s hard to manage it by ourselves

Maybe we can get the machine to do it?

Garbage collection

Garbage is bad

It’s hard to manage it by ourselves

Maybe we can get the machine to do it?

Yeah. We **can**. We'll just keep a *reference count* on it.

Garbage collection

Garbage is bad

It’s hard to manage it by ourselves

Maybe we can get the machine to do it?

Yeah. We **can**. We'll just keep a *reference count* on it.

But ... What about the race condition?

Garbage collection

Garbage is bad

It’s hard to manage it by ourselves

Maybe we can get the machine to do it?

Yeah. We **can**. We'll just keep a *reference count* on it.

But ... What about the race condition?

We'll prevent multiple threads from touching this counter.
How?
GIL - Global Interpreter Lock, i.e. mutex on the interpreter

Garbage collection

Garbage is bad

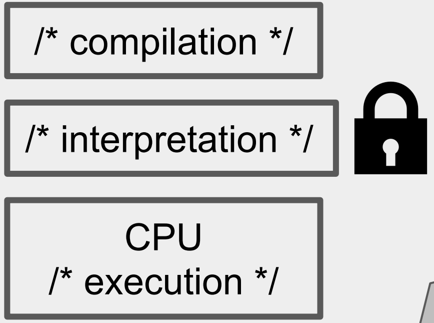
It’s hard to manage it by ourselves

Maybe we can get the machine to do it?

Yeah. We **can**. We'll just keep a *reference count* on it.

But ... What about the race condition?

We'll prevent multiple threads from touching this counter.
How?
GIL - Global Interpreter Lock, i.e. mutex on the interpreter:

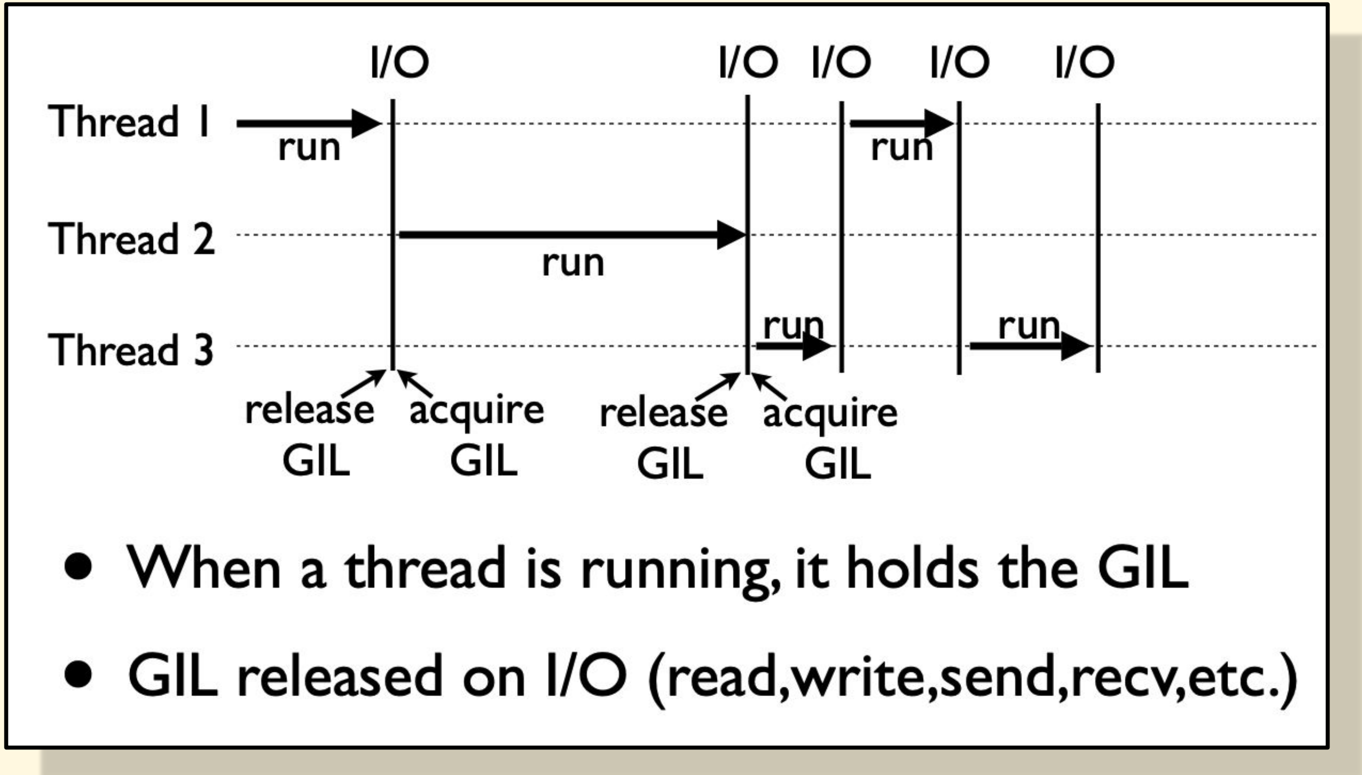


/* compilation */

/* interpretation */

CPU
/* execution */

Workflow with GIL



- When a thread is running, it holds the GIL
- GIL released on I/O (read,write,send,recv,etc.)

That’s why cpu bound tasks are not accelerated by adding threads.

https://www.fatalerrors.org/a/python-concurrent-programming.html

Okay;
Threads are friends with io bound tasks;
Processes are friends with everything;
Thanks for your attention.

Cython

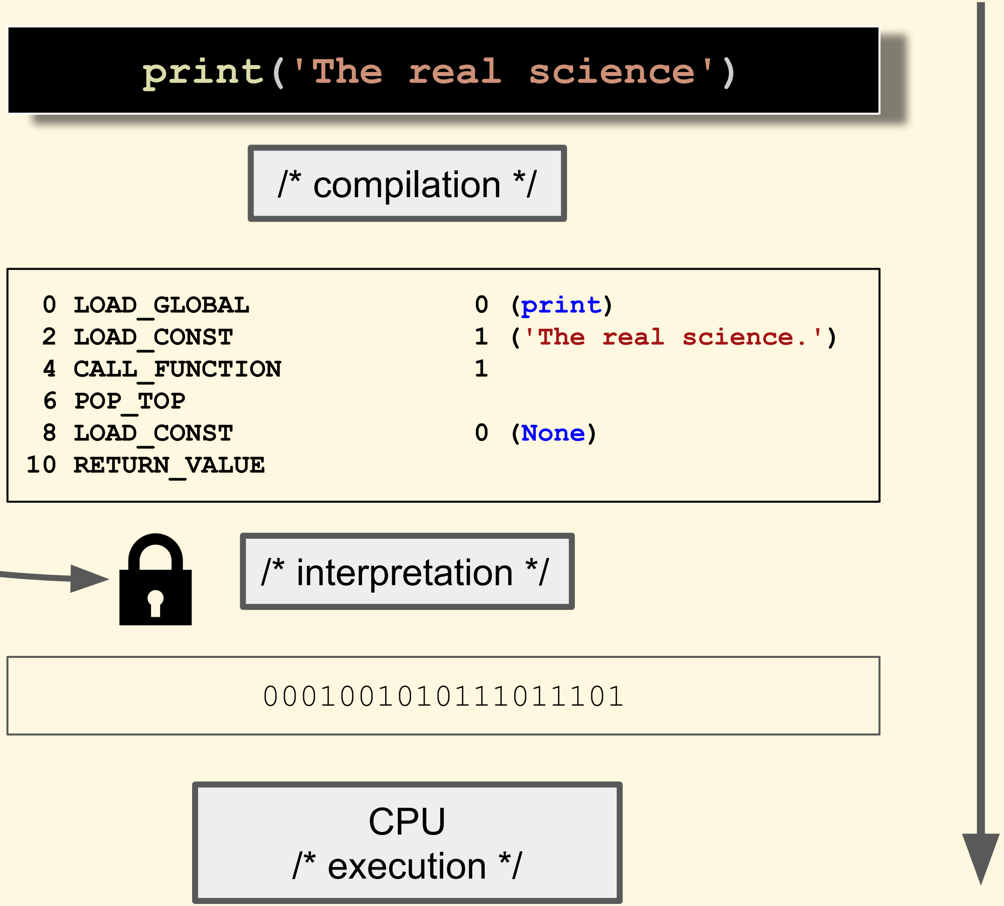
… or freeing ourselves from the shackles

- avoiding GIL
- threads & true parallelism
- fun

Avoiding GIL

We can avoid blocking the interpreter if we promise to keep track of memory ourselves. Then we get true || even with threads.

```
with nogil:
    ...
```



```
print('The real science')
```

/* compilation */

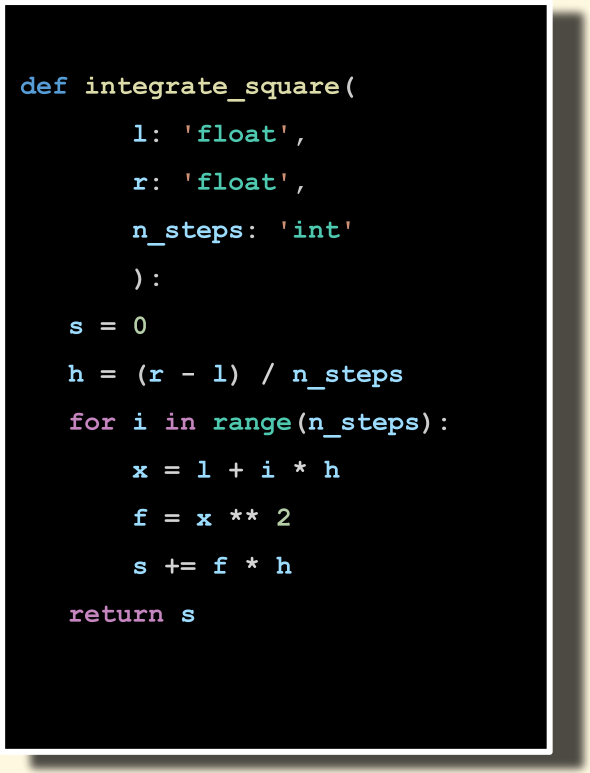
```
 0 LOAD_GLOBAL              0 (print)
 2 LOAD_CONST               1 ('The real science.')
 4 CALL_FUNCTION            1
 6 POP_TOP
 8 LOAD_CONST               0 (None)
10 RETURN_VALUE
```

/* interpretation */

```
0001001010111011101
```

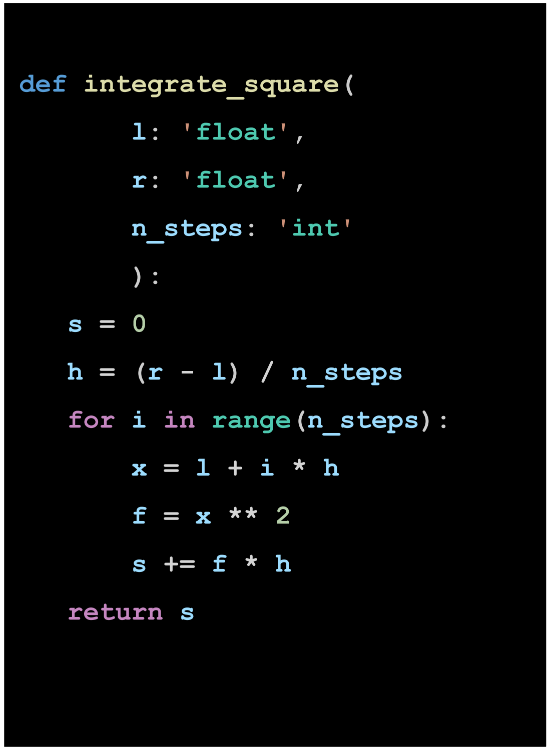
CPU
/* execution */

Ex. `with nogil`



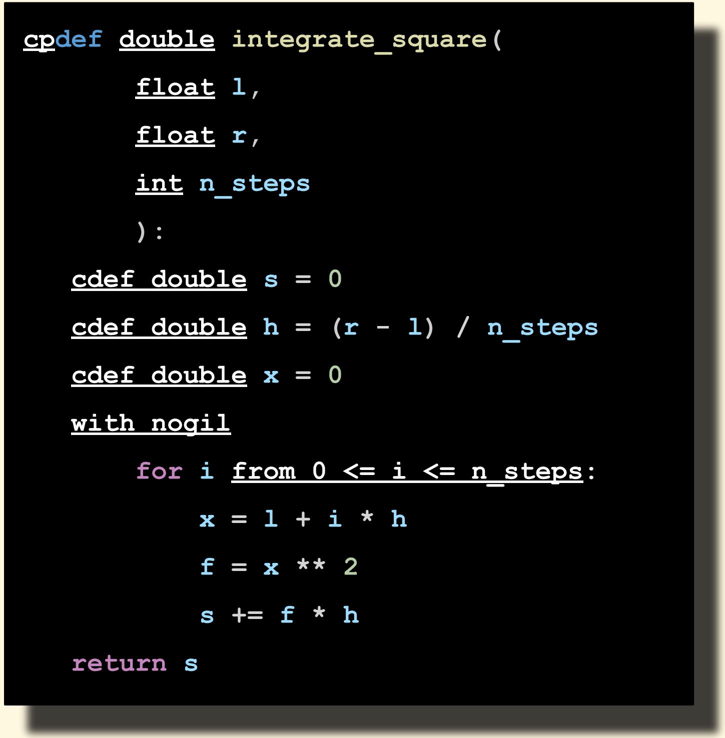
```
def integrate_square(
       l: 'float',
       r: 'float',
       n_steps: 'int'
       ):
   s = 0
   h = (r - l) / n_steps
   for i in range(n_steps):
       x = l + i * h
       f = x ** 2
       s += f * h
   return s
```

Ex. `with nogil`



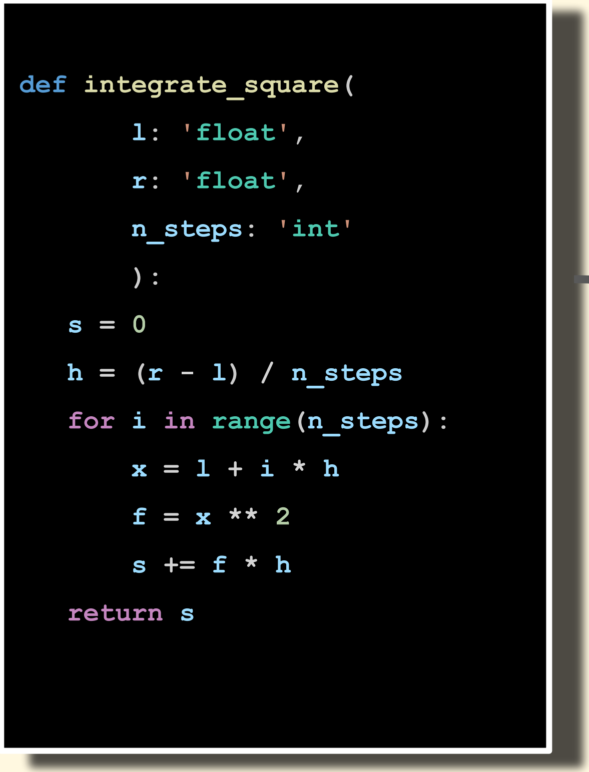
```python
def integrate_square(
        l: 'float',
        r: 'float',
        n_steps: 'int'
        ):
    s = 0
    h = (r - l) / n_steps
    for i in range(n_steps):
        x = l + i * h
        f = x ** 2
        s += f * h
    return s
```

/* C-fication */



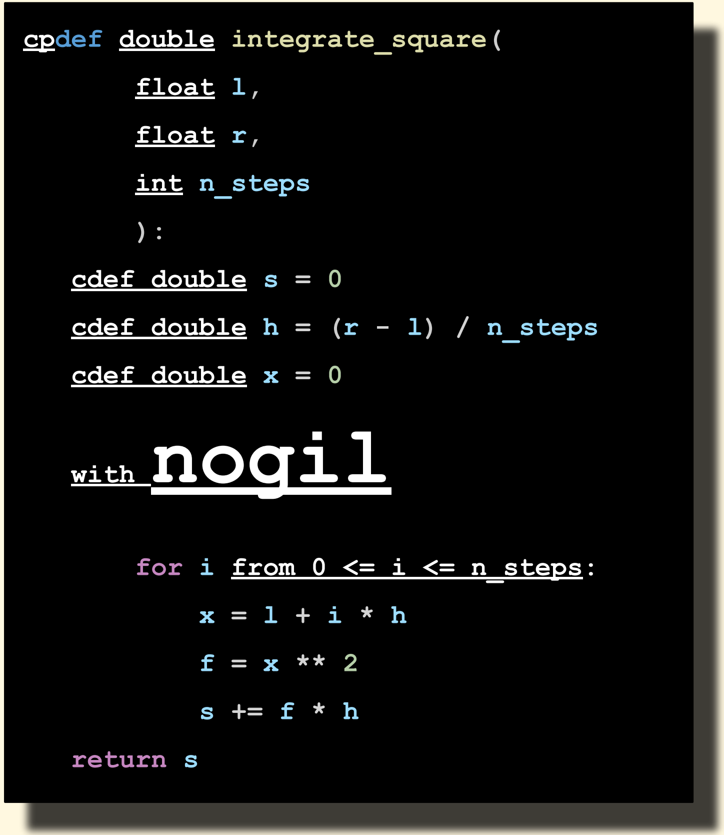
```cython
cpdef double integrate_square(
        float l,
        float r,
        int n_steps
        ):
    cdef double s = 0
    cdef double h = (r - l) / n_steps
    cdef double x = 0
    with nogil
        for i from 0 <= i <= n_steps:
            x = l + i * h
            f = x ** 2
            s += f * h
    return s
```

Ex. `with nogil`



```
def integrate_square(
        l: 'float',
        r: 'float',
        n_steps: 'int'
        ):
    s = 0
    h = (r - l) / n_steps
    for i in range(n_steps):
        x = l + i * h
        f = x ** 2
        s += f * h
    return s
```

/* C-fication */

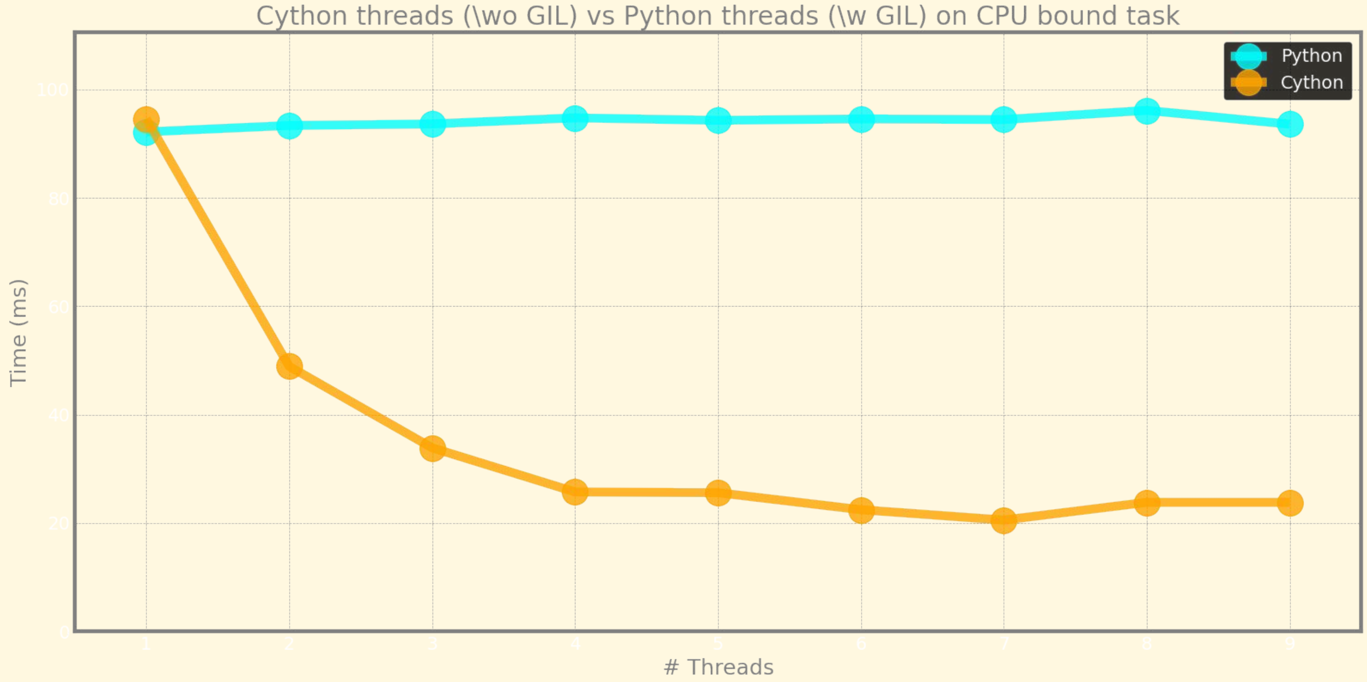


```
cpdef double integrate_square(
        float l,
        float r,
        int n_steps
        ):
    cdef double s = 0
    cdef double h = (r - l) / n_steps
    cdef double x = 0

    with nogil

        for i from 0 <= i <= n_steps:
            x = l + i * h
            f = x ** 2
            s += f * h
    return s
```

Ex. `with nogil`



It really works!

Let's have some fun:

- *break python even with GIL,*
- *steal a little money,*
- *compile the python code,*
- *avoid the fucking GIL and*
- *break semaphore*

GitHub