UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Python: asyncio

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Python and threads

# Multithread limitations

- The concurrency restrictions in the CPython interpreter are driven by its garbage-collection approach, which uses reference counts on objects to determine when they are no longer in use.

- Reference counting works by keeping track of who currently needs access to a particular Python object

- In a multi-threaded program, reference-count operations must be performed in a thread-safe manner, to avoid corrupted counts on objects.

- To avoid such problems, only one thread can be running in the interpreter (i.e. to actually be running Python code) at a time.

# What is GIL ?

- The Global Interpreter Lock (GIL, pronounced “gill”) prevents any Python process from executing more than one Python bytecode instruction at any given time.

- Therefore, even if we have multiple threads on a machine with multiple cores, we can only have one thread running Python code at a time.

- There’s a recent (Oct’21) implementation that seems to solve the problem eliminating in an effective way the GIL… we’ll see if it gets into the official version of Python… it must not break C extensions that rely on GIL, it must not slow down single threaded programs.

# Interpreters and GIL

- There exist several Python interpreters, e.g.
- CPython - the original C-based interpreter, that has GIL
- IronPython, written in C# does NOT have GIL
- Jython, written in Java does NOT have GIL

# GIL and multithreading

- Python supports multithreading but due to GIL it can be used only to support concurrency, not parallelism !

```
import threading

def hello_from_thread():
    thread_name = threading.current_thread()
    print(f'Hello from thread {thread_name}!')

hello_thread = threading.Thread(target=hello_from_thread)
hello_thread.start()

hello_thread2 = threading.Thread(target=hello_from_thread)
hello_thread2.start()

total_threads = threading.active_count()
thread_name = threading.current_thread().getName()

print(f'Python is currently running {total_threads} thread(s)')
print(f'The current thread is {thread_name}')

hello_thread.join()
```

# GIL and I/O

- The global interpreter lock is released when I/O operations happen.

- Threads executing I/O operations can run in parallel, so we can speed-up our program

```
import time
import threading
import requests
def read_example() -> None:
  response = requests.get('https://www.example.com')
  print(response.status_code)

thread_1 = threading.Thread(target=read_example)
thread_2 = threading.Thread(target=read_example)

thread_start = time.time()
thread_1.start()
thread_2.start()
print('All threads running!')
thread_1.join()
thread_2.join()
thread_end = time.time()
print(f'Running with threads took {thread_end - thread_start} seconds.')
```

# GIL and I/O

- In the case of I/O the low-level system calls are outside of Python runtime.
  GIL is released because it is not interacting with Python objects directly.
- GIL is only re-acquired when the data received is translated back into a Python object.
  - Then at the operating system level the I/O operations execute concurrently.
- The GIL does not have much impact on the performance of I/O-bound multi-threaded programs as the lock is shared between threads while they are waiting for I/O.

# I/O bound vs. CPU bound

- In the case of a CPU bound operation, it would complete faster the CPU was more powerful.
  CPU bound operations are typically computations and processing code.

- In the case of an I/O bound operation it would get faster if our I/O devices could handle more data in less time.
  In an I/O bound operation we spend most of our time waiting on a network or other I/O device.

# Multiprocessing

- CPU-bound tasks can be speeded-up using multiprocessing instead of multithreading

- Processes do not use shared memory, but there are features for sharing data and passing messages between them

- Each process will have its own GIL.

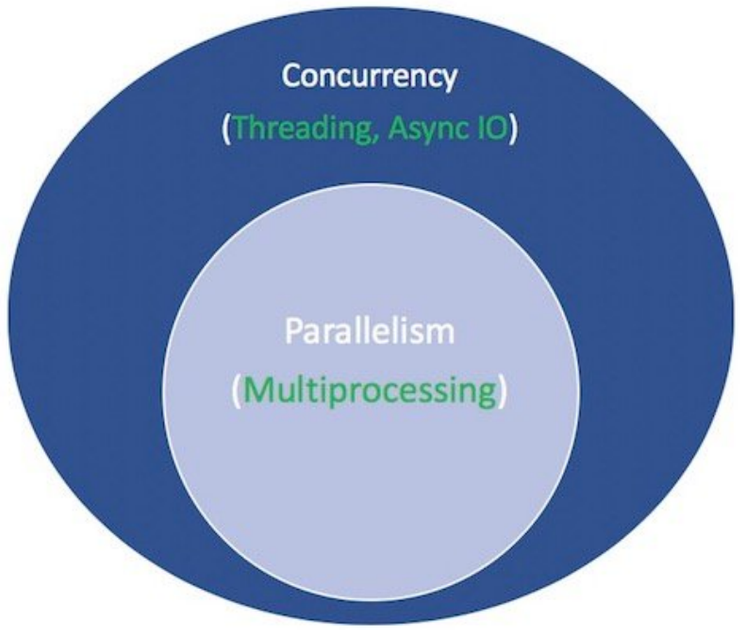
# Libraries

- Libraries like Numpy, PyTorch, Tensorflow are not affected by GIL: it's possible for the C/C++ code of the backend to release GIL if they are not touching Python objects

- The same can be done in Cython code, to write C/C++ extensions, using OpenMP-based parallelism

# Concurrency and parallelism in Python

- Asynchronous processing of I/O operations let us obtain parallelism also out of concurrent solutions like asyncio




```python
import time
import requests
def read_example() -> None:
  response = requests.get('https://www.example.com')
  print(response.status_code)

sync_start = time.time()
read_example()
read_example()
sync_end = time.time()
print(f'Running synchronously took {sync_end - sync_start} seconds.')
```

```
200
200
Running synchronously took 1.6331820487976074 seconds.
```

```python
import time
import threading
import requests
def read_example() -> None:
  response = requests.get('https://www.example.com')
  print(response.status_code)

thread_1 = threading.Thread(target=read_example)
thread_2 = threading.Thread(target=read_example)

thread_start = time.time()
thread_1.start()
thread_2.start()
print('All threads running!')
thread_1.join()
thread_2.join()
thread_end = time.time()
print(f'Running with threads took {thread_end - thread_start} seconds.')
```

```
All threads running!
200
200
Running with threads took 0.674811840057373 seconds.
```

# Profiling

- Python includer 2 profilers cProfile (faster, C-extension) and profile (slower, pure Python) that can be executed as:
  - `python -m cProfile [-o output_file] (-m module | myscript.py)`
- PyCharm provides facilities to run cProfile in the IDE
- It is possible to instrument the code using cProfile `Profile` class

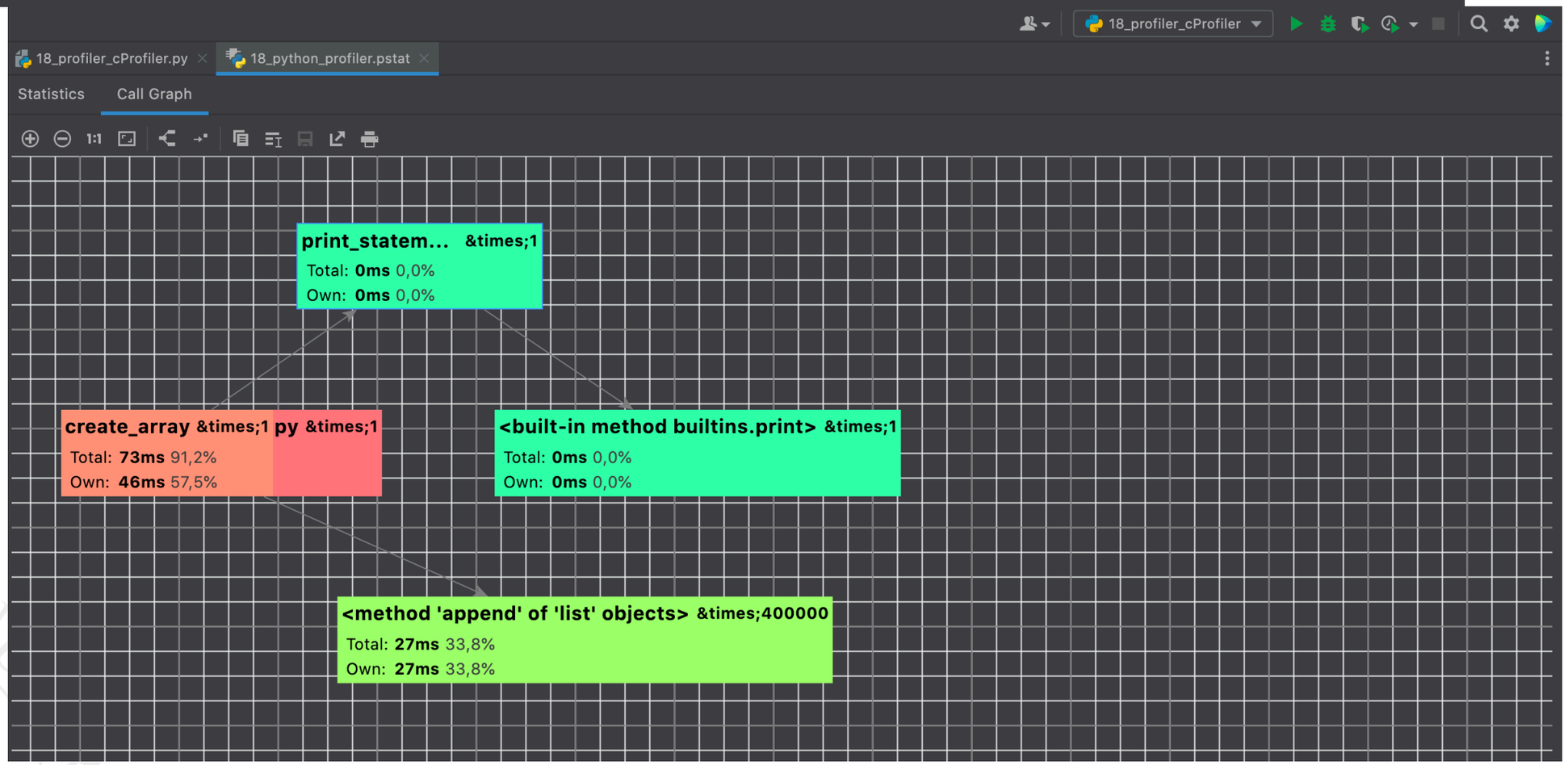
# Profiling example

18_profiler_cProfiler

18_profiler_cProfiler.py | 18_python_profiler.pstat

Statistics | Call Graph

| Name | Call Count | Time (ms) | Own Time (ms) |
|---|---|---|---|
| create_array | 1 | 73 91,2% | 46 57,5% |
| <method 'append' of 'list' objects> | 400000 | 27 33,8% | 27 33,8% |
| main | 1 | 80 100,0% | 6 7,5% |
| <built-in method builtins.print> | 1 | 0 0,0% | 0 0,0% |
| print_statement | 1 | 0 0,0% | 0 0,0% |
| 18_profiler_cProfiler.py | 1 | 80 100,0% | 0 0,0% |



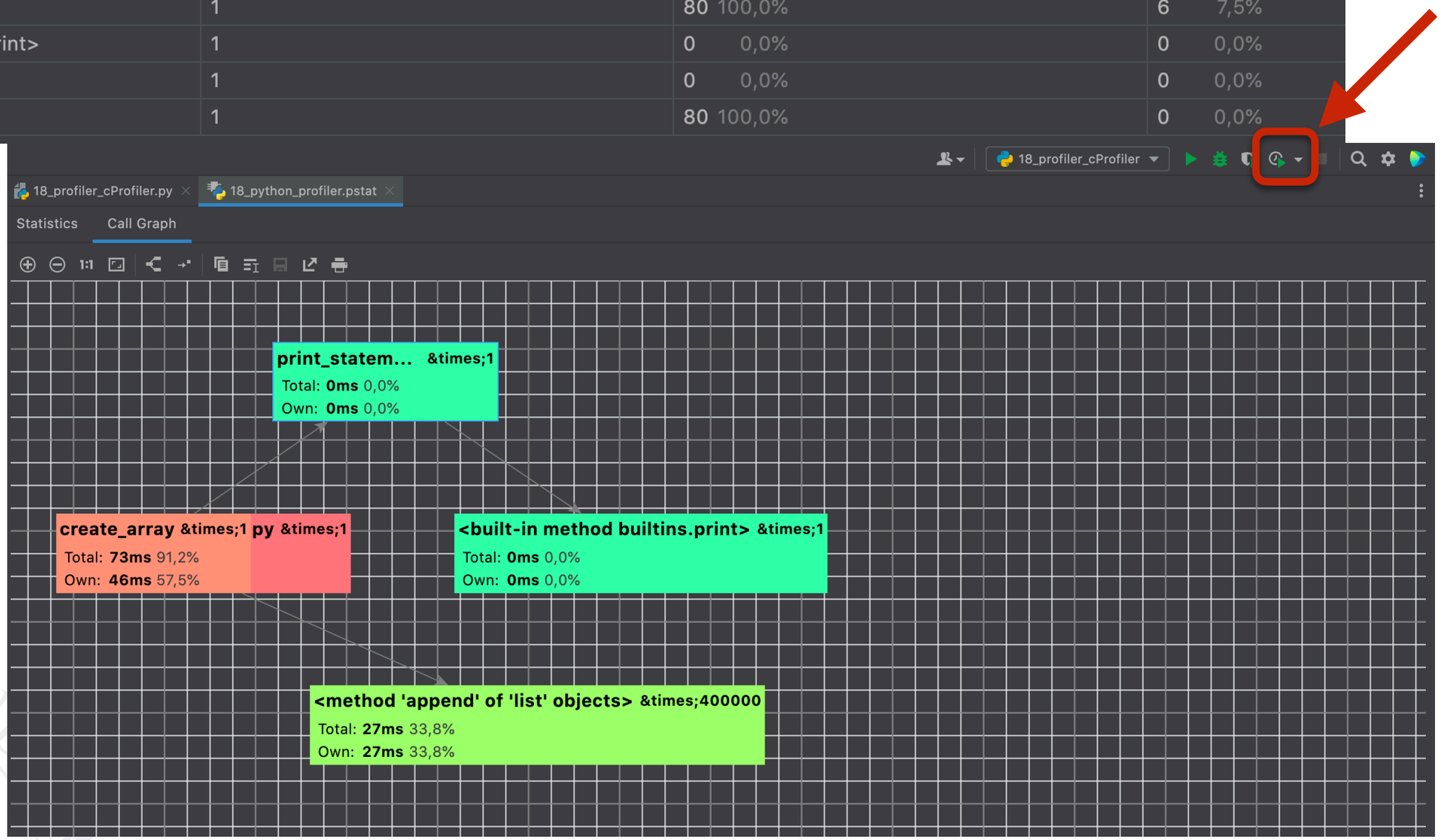
# Profiling example

18_profiler_cProfiler

18_profiler_cProfiler.py | 18_python_profiler.pstat

Statistics | Call Graph

| Name | Call Count | Time (ms) | Own Time (ms) |
|---|---|---|---|
| create_array | 1 | 73 91,2% | 46 57,5% |
| <method 'append' of 'list' objects> | 400000 | 27 33,8% | 27 33,8% |
| main | 1 | 80 100,0% | 6 7,5% |
| <built-in method builtins.print> | 1 | 0 0,0% | 0 0,0% |
| print_statement | 1 | 0 0,0% | 0 0,0% |
| 18_profiler_cProfiler.py | 1 | 80 100,0% | 0 0,0% |



# Line profiler

- Use line_profiler for a more fine grained profiling of single functions

  - Decorate the functions to profile with `@profile`

- Create profiling data with:
  `kernprof -l script_to_profile.py`

- Show profiling with:
  `python -m line_profiler`
  `script_to_profile.py.lprof`

# Line profiler example

- Line number
- Hits: number of times the line is run
- Time: the total amount of time this line executed across all hits
- Per Hit: the average amount of time
- % Time: The percentage of time spent on that line relative to the total amount of recorded time spent in the function.

```
Total time: 0.000637 s
File: 18_profiler_line_profiler.py
Function: search_function at line 22

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
    22                                           @profile
    23                                           def search_function(data):
    24      1001        274.0      0.3     43.0      for i in data:
    25      1000        363.0      0.4     57.0          if i in [100, 200, 300, 400, 500]:
    26                                                    print("success")
```

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# asyncio

# Asyncio

- Asynchronous programming means that a particular long-running task can be run in the background separate from the main application.

- Asyncio was introduced in Python 3.4 to allow asynchronous I/O operations.

- Asyncio exploits the fact that I/O operations release the GIL.

- Asyncio does not circumvent the GIL, it is still subject to it. If we have a CPU bound task, we still need to use multiple processes to execute it concurrently (which can be done with asyncio itself).

# Event loop

- An event loop is at the heart of every asyncio application.

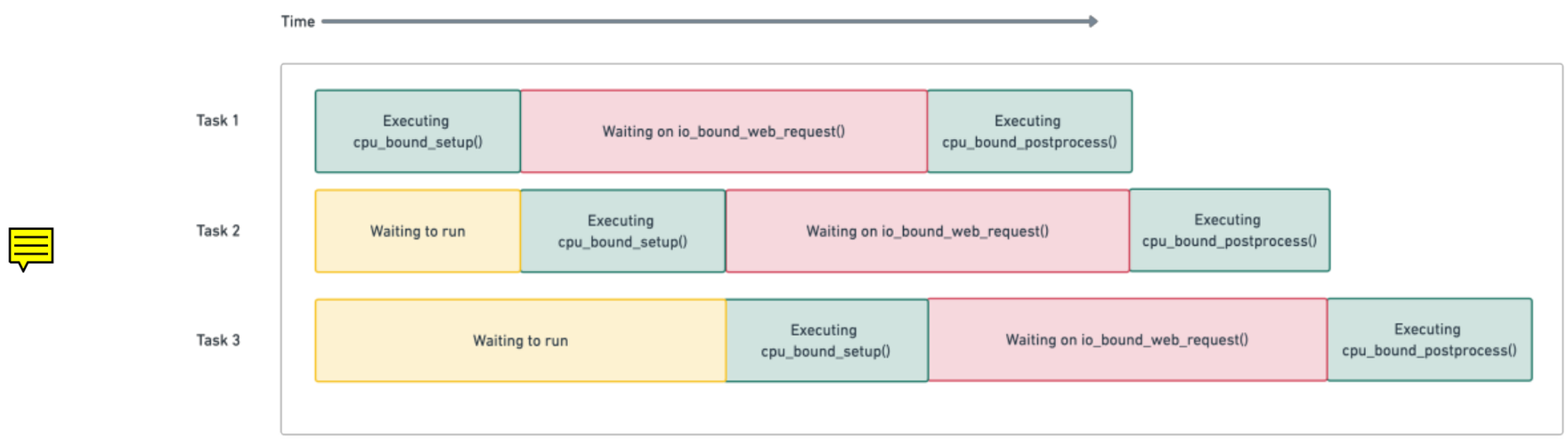
- In Asyncio the event loop keeps a queue of tasks. Tasks pause their execution when hitting a I/O bound operation and will let the event loop run other tasks that are not waiting for I/O operations to complete.

- Each iteration of the event loop checks for tasks that need to be run and will run them one at a time until a task hits an I/O operation.

- When I/O operations are completed the waiting tasks are awakened.

# Parallel I/O

- Considering different tasks that process alternated CPU-bound and I/O-bound operations we may get an execution like:




- At a given time there's only one CPU-bound operation executing, because of GIL, but there may be parallel I/O-bound operations executing

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Asyncio basics

# Coroutines

- A *coroutine* is a function that can suspend execution to be resumed later.
  - Also other languages have introduced them, eg. C++20

- The `async` keyword will let us define a coroutine and the `await` keyword will let us pause our coroutine when we have a long-running operation.

- When that long-running operation is complete, we can “wake up” our paused coroutine and finish executing any other code in that coroutine.

# Coroutines

- A *coroutine* is a function that can suspend execution to be resumed later.
  - Also other languages have introduced them, eg. C++20

This is cooperative multitasking

- The `async` keyword will let us define a coroutine and the `await` keyword will let us pause our coroutine when we have a long-running operation.
- When that long-running operation is complete, we can “wake up” our paused coroutine and finish executing any other code in that coroutine.

# Coroutines

- coroutines don’t get executed when we call them directly, instead we create a coroutine object that can be run later.

- To run a coroutine, we need to explicitly put it on an event loop.

- Since Python 3.7 we can use asyncio library functions to create and mage event loops, like `asyncio.run`

# Coroutines

- The `await` keyword is usually followed by a call to a coroutine, or an awaitable object (also tasks and futures), and executes it.

- The `await` expression will also pause the coroutine that it is contained in until the coroutine we awaited finishes and returns a result.

- When the coroutine we awaited finishes, we'll have access to the result it returned and our containing coroutine will 'wake up' to handle the result.

# Tasks: creation

- Tasks are wrappers around a coroutine that schedule it on the event loop as soon as possible. Scheduling and execution are non-blocking, i.e. once a task is created, it is possible to execute other code while the task is running.

- The `await` keyword is blocking, i.e. it pauses the entire coroutine until the result of the await expression is returned.

- When these tasks wrap a long-running operation, any waiting they do will happen concurrently.

- Tasks are created using the `asyncio.create_task` function.

# Tasks: results and completion

- It is possible to check if a task is completed with the done() method and get its result with `result()` method.

- `await` doesn't allow to cancel or timeout a coroutine that can't complete, but tasks can be cancelled and their cancellation can be checked with `cancelled()` method.

# Tasks: cancellation and timeout

- Each task object has a method named `cancel` to stop a task.
  Cancelling a task will cause that task to raise a `CancelledError` when we await it.

- To set a timeout to cancel a task use `asyncio.wait_for`. This function takes in a coroutine or task object and a timeout specified in seconds. It then returns a coroutine that we can await. When the timeout is reached a asyncio.`TimeoutError` exception is raised.

# Tasks: cancellation

- To avoid cancellation of an awaitable wrap it with the `asyncio.shield`, function that disallow cancellation of the passed coroutine.

- Cancellation or timeout result in an exception but the awaitable is not stopped so we can still get the result (using another `await` or looping over `done()`)

# Futures

- A `Future` is a Python object that contains a single value that you expect to get at some point in the future, but may not yet have.
  - Used also in Java and C++11 parallel programming.
  - It is an *awaitable* like coroutines and tasks.
- A future is completed when it can wrap a value that is provided by some process (use `set_result()`).
  - Use `done()` method to check if the future is completed and `result()` to get the value.

# Event loop

- Creating explicitly an event loop, instead of relying solely on `asyncio.run` allows to create custom logic to manage the tasks and perform low-level operation such as scheduling tasks (e.g. using `call_soon()` to schedule a task to run earlier)

- We can create an event loop by using the `asyncio.new_event_loop` method.

- Use the method named `run_until_complete` which takes a coroutine and runs it until it finishes.

- Close the loop to free up any resources it was using. Do this in `finally` block so that any exceptions thrown doesn't stop closing the loop.

- Get the default loop with `get_event_loop()`

# Asyncio pitfalls - CPU

- asyncio has a single-threaded concurrency model. This means we are still subject to the limitations of a single thread and the global interpreter lock (GIL).

- If we need perform CPU bound work and still want to use `async` / `await` syntax we can do so, but we need to use multiprocessing and tell asyncio to run our tasks in a process pool.

# Asyncio pitfalls - blocking

- When running a blocking API call inside a coroutine we're blocking the event loop thread itself, meaning we stop any other coroutines or tasks from executing.
    - Examples of blocking API calls include libraries such as `requests`, or `time.sleep`.
        - Generally, any function that performs I/O that is not a coroutine or performs time-consuming CPU operations can be considered blocking.
- Use non-blocking libraries (e.g. `aiohttp` instead of `requests`) or tell asyncio to use multithreading with a thread pool executor.

# Dubugging asyncio

- Asyncio provides a debugging modality that logs slow coroutines and missing coroutine await using exceptions

- It can be activated in the API or using a CLI parameter or an environment variable

- `asyncio.run(coroutine(), debug=True)`

- `python3 -X dev program.py`

- `PYTHONASYINCIODEBUG=1 python3 program.py`

- Using `set_debug()` on the loop object

# Timing coroutine

- It is possible to use a decorator to time a coroutine
  - a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.

- ```
  def timing_async():
      def wrapper(func: Callable) -> Callable:
          @functools.wraps(func)
          async def wrapped(*args, **kwargs) -> Any:
              print(f'starting {func} with args {args} {kwargs}')
              start = time.time()
              try:
                  return await func(*args, **kwargs)
              finally:
                  end = time.time()
                  total = end - start
                  print(f'finished {func} in {total:.4f} second(s)')
          return wrapped
      return wrapper
  ```

- Use `@timing_async()` to decorate a coroutine
  ```

# Exceptions

- When an exception is thrown inside a task, the task is considered done with its result as an exception. This means that no exception is thrown up the call stack and no cleanup could be performed.

- When we `await` a task that failed, the exception will get thrown where we perform the `await` and the traceback will reflect that. If we don't await a task at some point in our application, we run the risk of never seeing an exception that a task raised.

# Shutting down

- adding custom shutdown logic to our application allows any in-progress tasks a few seconds to finish, sending any messages they might want to send.

- This can be implemented using low-level event loop calls, e.g. adding signal handling (to manage Unix-based signals such as SIGINT/SIGKILL) with `loop.add_signal_handler()`

# Tasks: concurrent execution

- We can use list comprehension instead of manually creating and awaiting tasks
  - We need a list comprehension to create tasks and another one to wait for them:

```
tasks = [asyncio.create_task(foo(param)) for
                    param in params]
[await task for task in tasks]
```

  - Do not use only one or we get only sequential execution due to the initial `await`:

```
async def main() -> None:
  delay_times = [3, 3, 3]
  [await asyncio.create_task(foo(param))
          for param in params]
```

# Tasks: concurrent execution

- Asyncio provides a specific API call: `asyncio.gather` for concurrent execution of tasks

- This function takes in a sequence of awaitables and lets us run them concurrently all in one line of code. If any of the awaitables we pass in is a coroutine, gather will automatically wrap it in a task to ensure that it runs concurrently.
    - This means that we don't have to manually wrap everything with `asyncio.create_task`

- `asyncio.gather` returns an awaitable itself, thus it can be used in an await expression: it will pause until all the passed awaitables are complete, returning their results once everything finishes.
    - The order of results is the same of the tasks, independently from their order of completion.

# Gather and exceptions

- `asyncio.gather` has a `return_exceptions` optional parameter, which allows us to specify how we want to deal with exceptions from our awaitables.

- With default `False` value `asyncio.gather` won't cancel any other tasks that are running if there is a failure, and if there is more than one exception we'll only see the first one that occurred when we await the gather.

- When set to `True` the results will report the exceptions and it will be possible to handle all them:

```
results = await asyncio.gather(*tasks, return_exceptions=True)

exceptions = [res for res in results if
                    isinstance(res, Exception)]
successful_results = [res for res in results if not
                    isinstance(res, Exception)]
```

# Processing results

- Gather returns results when all tasks are completed, but it is also possible to process them as they are made available.

- To handle this case, asyncio exposes an API function named `as_completed`. This method takes a list of awaitables and returns an iterator of *futures*.

- We can then iterate over these *futures*, awaiting each one. This means that we'll be able to process results as soon as they are available, although the order is not predictable.

# Results and timeout

- as_completed() supports also timeouts. If it takes longer than the timeout each awaitable in the iterator will throw a `TimeoutException` when we await it.
  - However the timed-out tasks will still be running in the background. It is hard to figure out which tasks are still running if we want to cancel them…
- Moreover, `as_completed()` doesn't let us easily associate the results completed with the task the returned them

# Results: finer control

- Both `gather` and `as_completed` do not provide and easy way to cancel running tasks case of an exception.
  - In case of `as_completed` it is challenging also to keep track of exactly which task had completed as the iteration order is nondeterministic.
- `asyncio.wait` provides a more granular control providing optional `timeout` and return timing options (`return_when`)
  - Passing coroutines to wait is deprecated, pass only tasks
  - Timeout does not result in task cancellation or exception launch as in `wait_for` and `as_completed()`

# Asynchronous generators

- Python generators are an implementation of the iterator design pattern
- This pattern allows us to define sequences of data lazily and iterate through them one element at a time. This is useful for potentially large sequences of data where we don’t need to store everything in memory all at once.
- A simple synchronous generator is a normal Python method which contains a yield statement instead of a return statement.
- Use `async for` to implement an asynchronous version of a generator
  - There’s **no parallelization**, it simply allows sequential iteration over an async source.
  - use `async for` to iterate over lines coming from a TCP stream, messages from a websocket, or database records from an async DB driver.

# Books

- Python Concurrency with asyncio, D. B. Kirk and M. Fowler, Manning - Chapt. 1-4