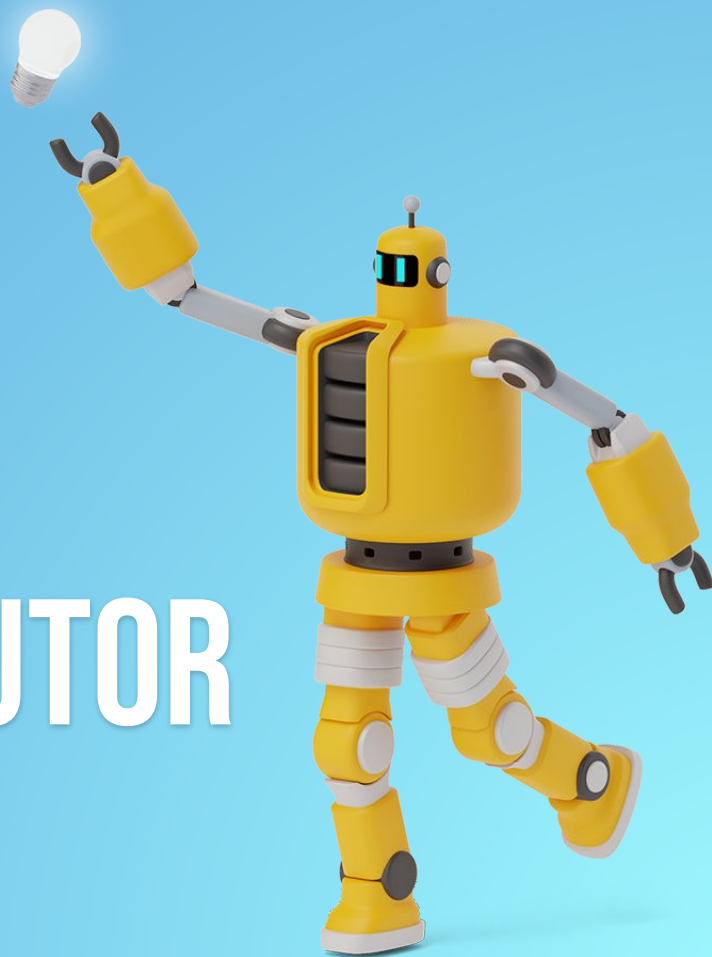


CONCURRENCIA EN PYTHON CON THREADPOOLEXECUTOR



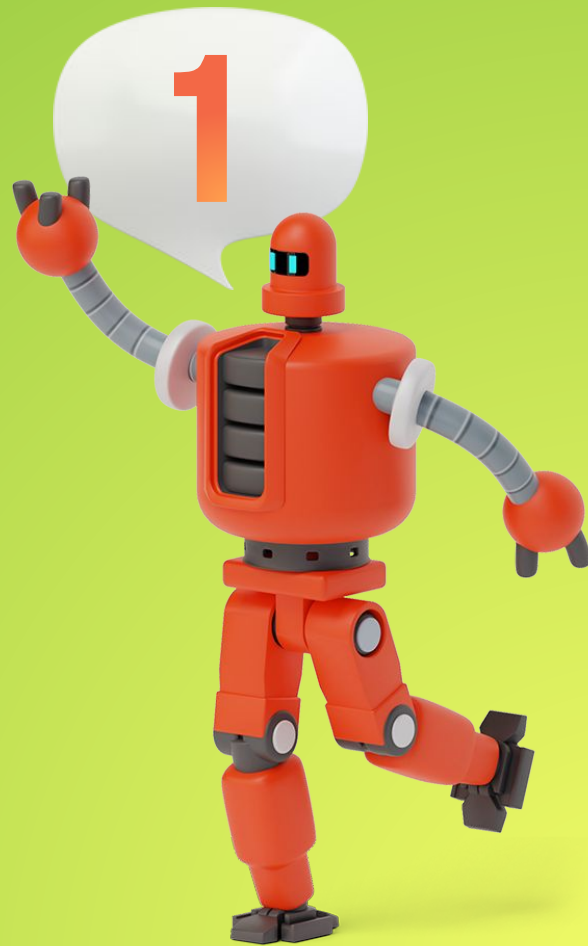


Soy Diana Piñeiro

- Ingeniera Informática con más de 20 años de experiencia.
- Master de Big Data en Universidad Autónoma de Madrid.
- Hace 3 años me incorporé al equipo de Kairos, como Data Engineer en su cliente ING.
- He pasado por muchos lenguajes y roles pero sobre todo he trabajado en Java, .NET y Big Data



INTRODUCCIÓN



CONCURRENCIA VS PARALELISMO

Concurrencia

pueden ocurrir simultáneamente

Paralelismo

ejecutar al mismo tiempo



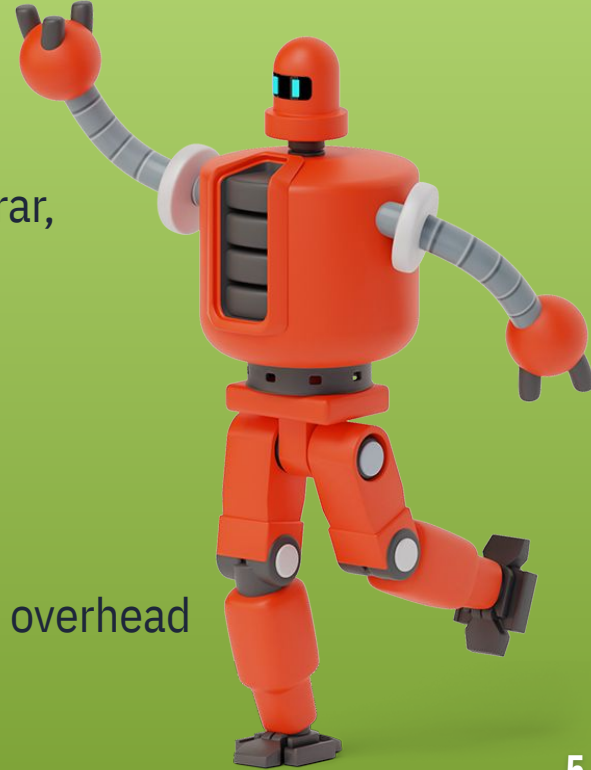
Paralelismo implica concurrencia
pero

Concurrencia no implica paralelismo

Tipo Concurrencia	Num procesos	Apropiado para
Threading	1 proceso	I/O bound tasks
Asincio	1 proceso	non-blocking IO tasks asynchronous programming
Multiprocessing	N procesos	CPU bound tasks

MULTITHREADING PARA APLICACIONES I/O

- Threads
 - Es un flujo separado de ejecución
 - Con *threading.Thread* puedes crear, iniciar, esperar, destruir...
- ThreadPoolExecutor
 - Implementación del patrón ThreadPool
 - Simplifica el uso de Threads
 - Permite reutilizar el mismo thread reduciendo el overhead de destruir y recrear
 - Basado en Executor & Futures



USOS PARA THREAD / THREADPOOLEXECUTOR



Tasks son funciones sin estado ni efectos colaterales.



Operaciones I/O (leer del disco, imprimir, download/upload datos, request server o BD, ...)



Tienes que lanzar la misma función muchas veces



Tienes que aplicar la misma función a una colección en un for



Solo tienes una tarea.



Necesita estado



Necesita coordinación o sincronización con otras tareas

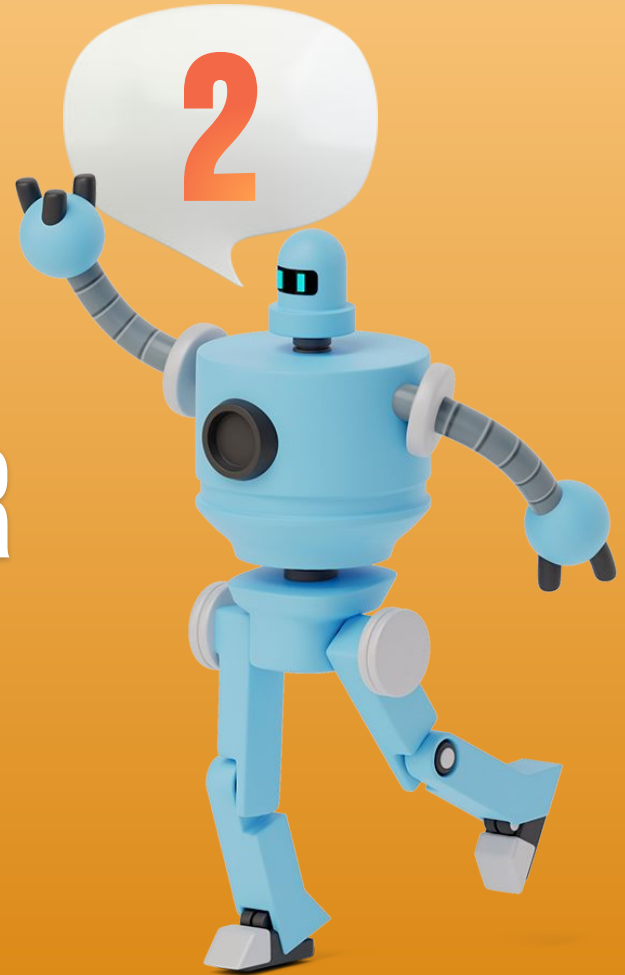


Tiene que esperar a un evento



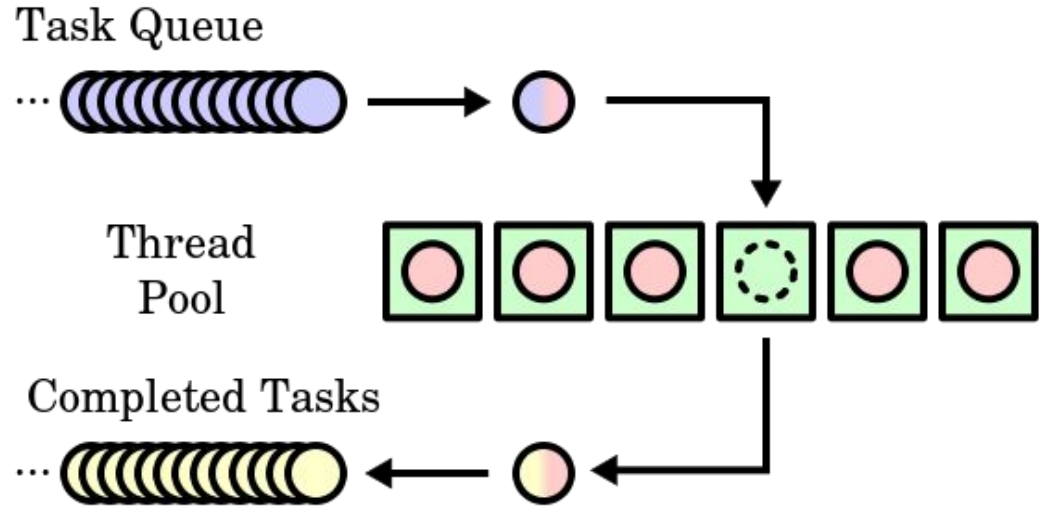
CPU

COMO FUNCIONA THREADPOOLEXECUTOR



ESTRUCTURA DEL THREADPOOLEXECUTOR

- Queue
- WorkItem - Futures
- Workers



FUNCIONAMIENTO DEL THREADPOOLEXECUTOR

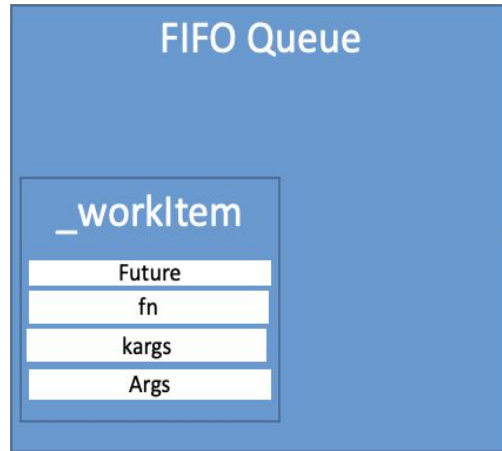
Submit(task)

1. Create workitem
2. Put in queue
3. Add thread if < max_workers

Threads no se crean al crear el pool.

Se crean on-demand hasta el max_workers

Una vez creados siguen vivos esperando a otra task hasta el shutdown del pool



Get workitem()
if canceled return
else run

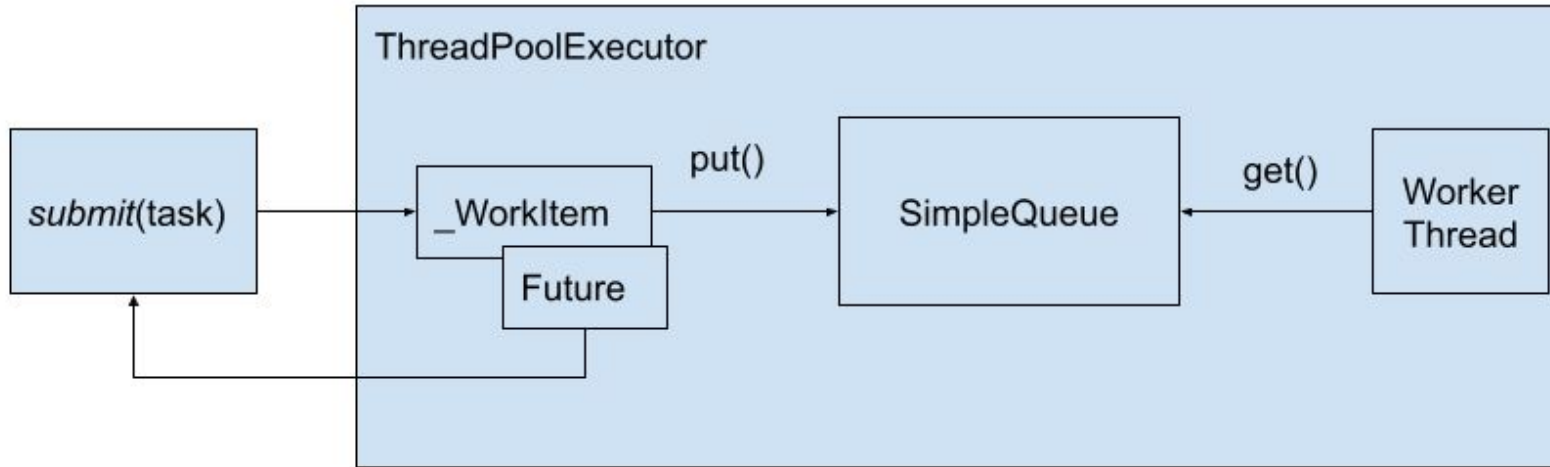
Thread 1

Thread n

Task in queue son cancelables
Task in thread (running) no

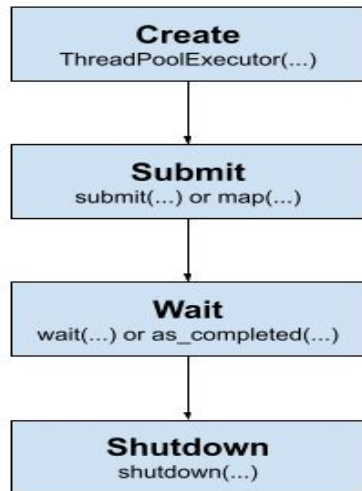
INTERIOR DEL THREADPOOLEXECUTOR

ThreadPoolExecutor Internals



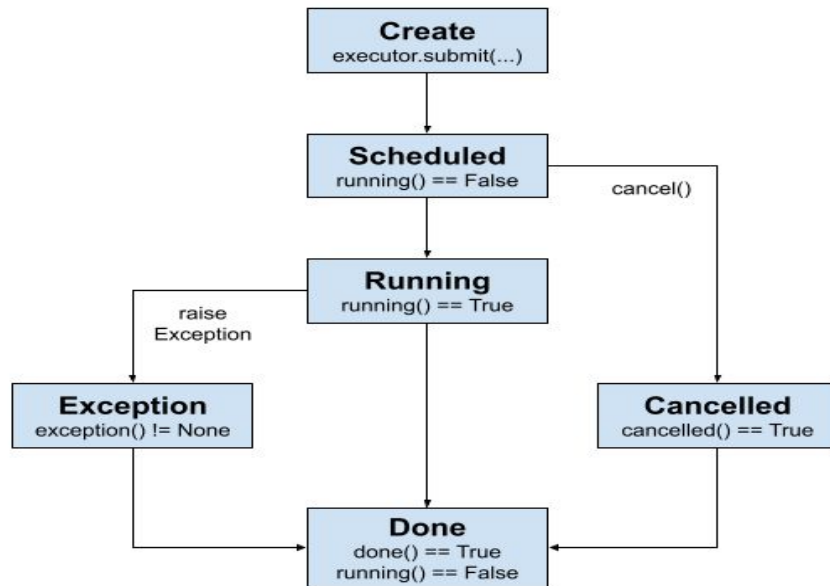
CICLO DE VIDA

Python ThreadPoolExecutor Life-Cycle



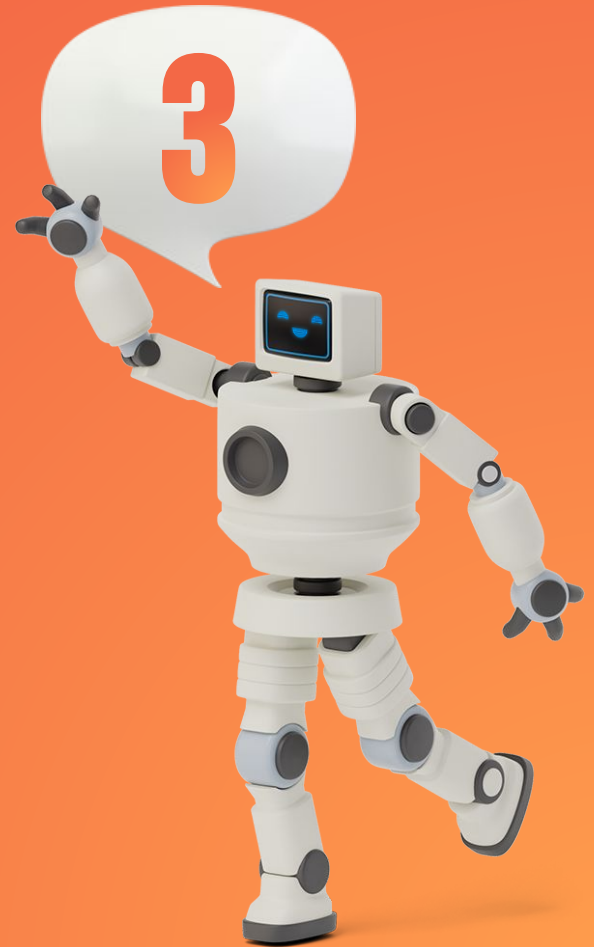
SuperFastPython.com

Python Future Life-Cycle



SuperFastPython.com

MI CASO DE USO



MI PROBLEMA

Creación de variables para un modelo

Se ejecutan varias queries independientes

El tiempo total demasiado largo. Necesitamos reducirlo

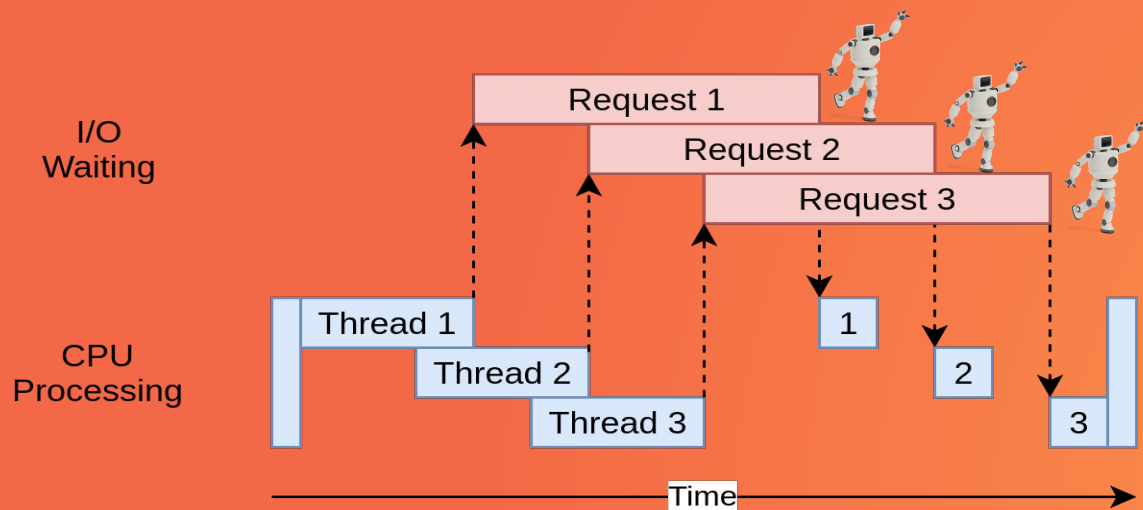
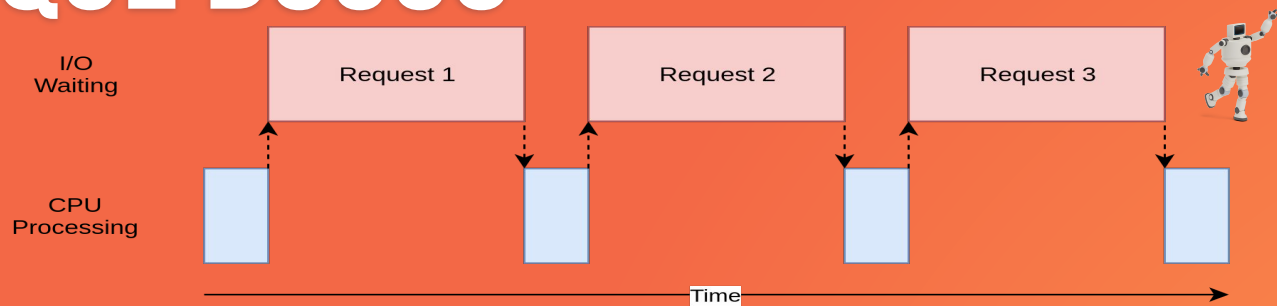
💡 Ya que podemos establecer varias conexiones a la BD, en lugar de ejecutarlas en serie se podrían ejecutar a la vez y así reducir el tiempo total.

Son task I/O, nuestro programa no consume recursos, está a la espera de recibir los datos de la BD.

Por tanto usaremos `Threads / ThreadPoolExecutor`



QUÉ BUSCO



EN LA PRÁCTICA

ejemplos de código de <https://superfastpython.com/>



CONTEXT MANAGER

- al salir se llama automáticamente a shutdown
- útil si tu código tiene que esperar a la ejecución de las task
- no tanto si quieres seguir mientras las tasks se ejecutan en background o reusar el pool en otros momentos.

```
with ThreadPoolExecutor(max_workers=10) as pool:
```

```
    # submit tasks and get results
```

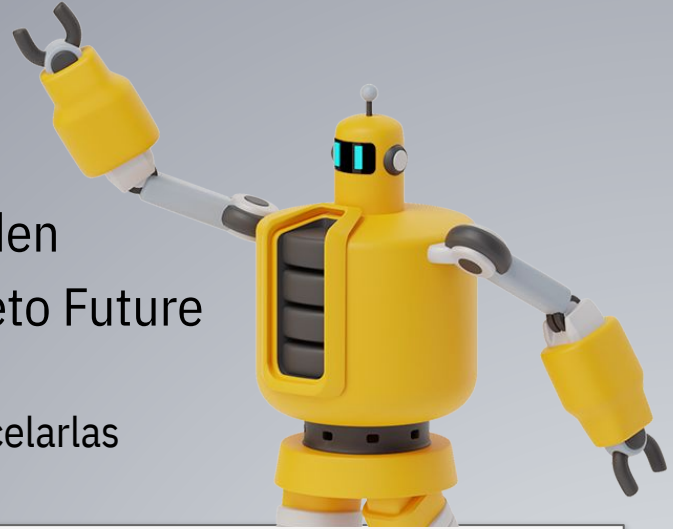
```
    # ...
```

```
# the pool is shutdown at this point
```



MAP

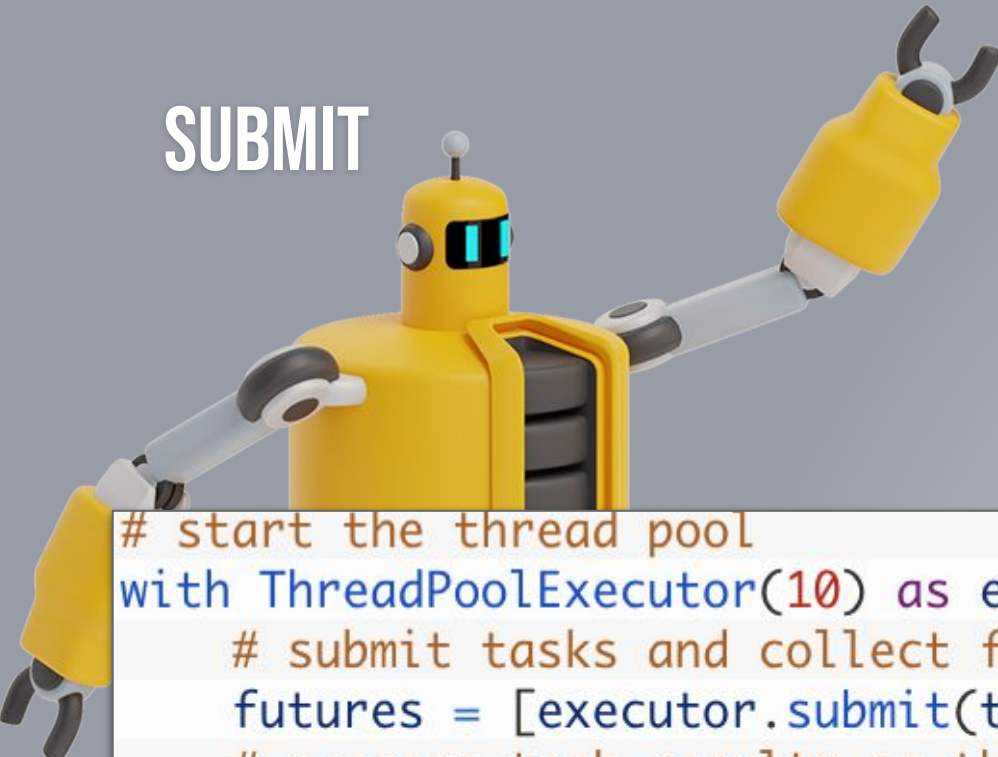
- es más simple por tanto más legible
- los resultados se devuelven en el mismo orden
- solo te da acceso a los resultados, no al objeto Future
 - no te da control sobre el orden de los resultados
 - no puedes checkear el estado de las tasks ni cancelarlas
 - tratamiento limitado de las excepciones



```
# start the thread pool
with ThreadPoolExecutor(10) as executor:
    # execute tasks concurrently and process results in order
    for result in executor.map(task, range(10)):
        # retrieve the result
        print(result)
```

SUBMIT

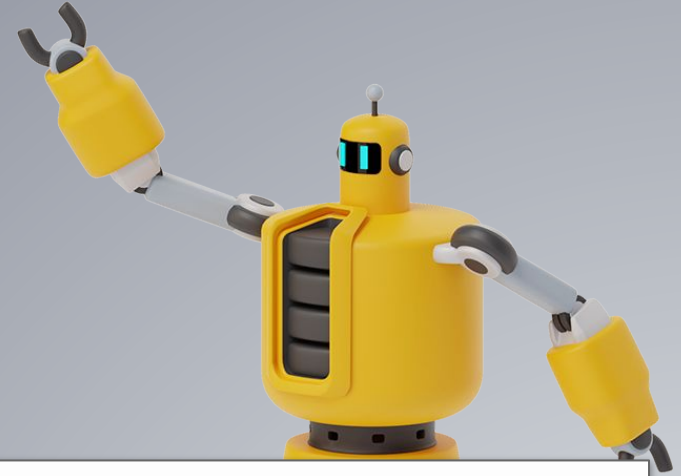
- más flexibilidad
- acceso a los Futures
- más complejidad



```
# start the thread pool
with ThreadPoolExecutor(10) as executor:
    # submit tasks and collect futures
    futures = [executor.submit(task, i) for i in range(10)]
    # process task results as they are available
    for future in as_completed(futures):
        # retrieve the result
        print(future.result())
```

SUMBIT EN ORDEN

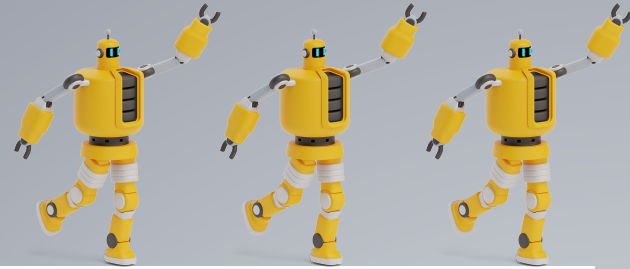
Procesar los resultados en el mismo orden
sin tener que usar map



```
# start the thread pool
with ThreadPoolExecutor(10) as executor:
    # submit tasks and collect futures
    futures = [executor.submit(task, i) for i in range(10)]
    # process task results in the order they were submitted
    for future in futures:
        # retrieve the result
        print(future.result())
```

NO ESPERAR

- No necesitamos todos los resultados



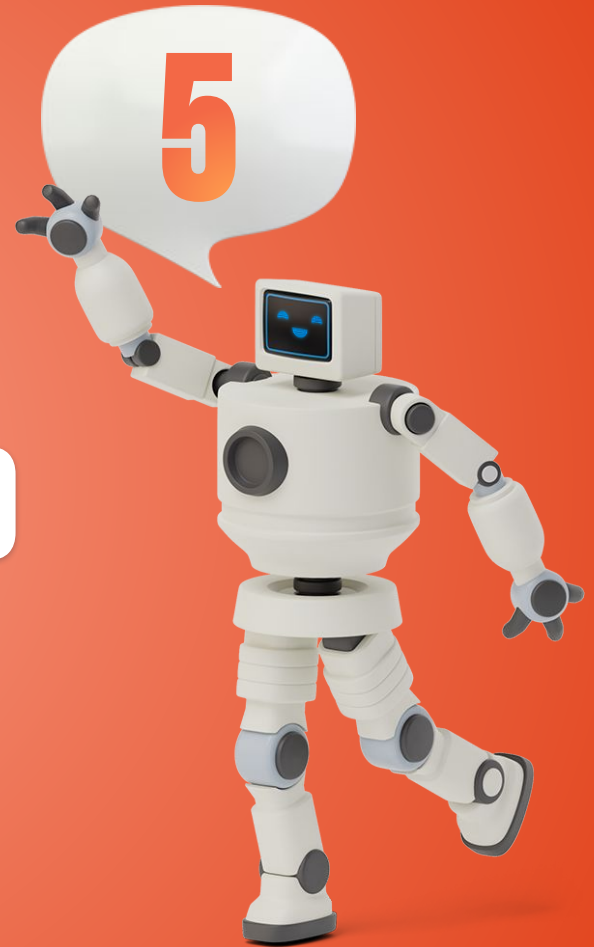
```
# start the thread pool
executor = ThreadPoolExecutor(10)
# submit tasks and collect futures
futures = [executor.submit(task, i) for i in range(10)]
# wait until any task completes
done, not_done = wait(futures, return_when=FIRST_COMPLETED)
# get the result from the first task to complete
print(done.pop().result())
# shutdown without waiting
executor.shutdown(wait=False, cancel_futures=True)
```

EXCEPTION HANDLING

- En el Thread Initialization (function initializer)
 - Rompe el Pool
- Durante Task Execution:
 - Para la task pero no el pool
 - Queda en el Future y se relanza al acceder al resultado
 - Opciones
 - Tratar la excepción en el código de la task y devolver un resultado (None, string vacio, codigo de error...) para que el receptor sepa que ha fallado.
 - Que el receptor del resultado trate la excepción. O con un try al acceder al resultado o consultando la exception del future antes de acceder
 - Durante Task Completion Callbacks:
 - definir una o varias funcion a la que se llame cuando termine la task (sucess / fail / cancel) (future.add_done_callback())
 - una exception en un callback no afecta a otros callbacks ni a las tasks



SOLUCIONANDO MI CASO





Creo un función por cada conjunto de variables que llamará a una query concreta.

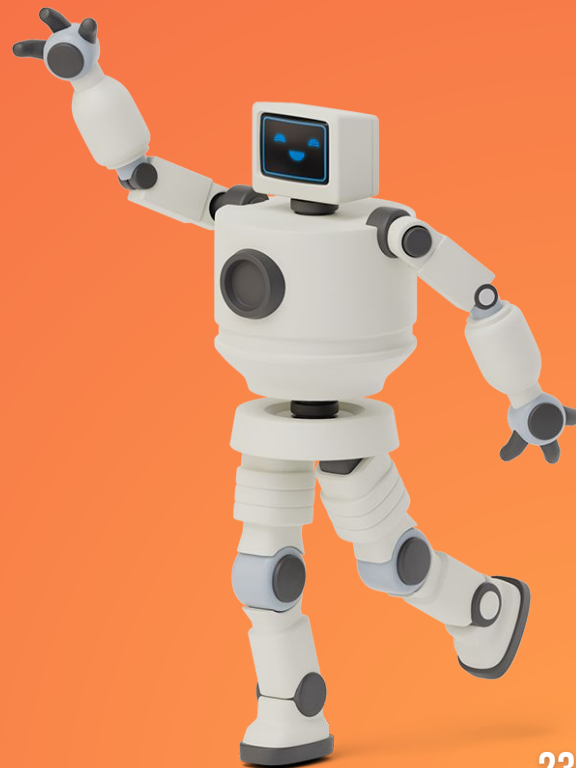
Las ejecuto concurrentemente



Si una de las queries falla no tengo todas las variables y no puedo ejecutar el modelo

Me interesa parar lo antes posible

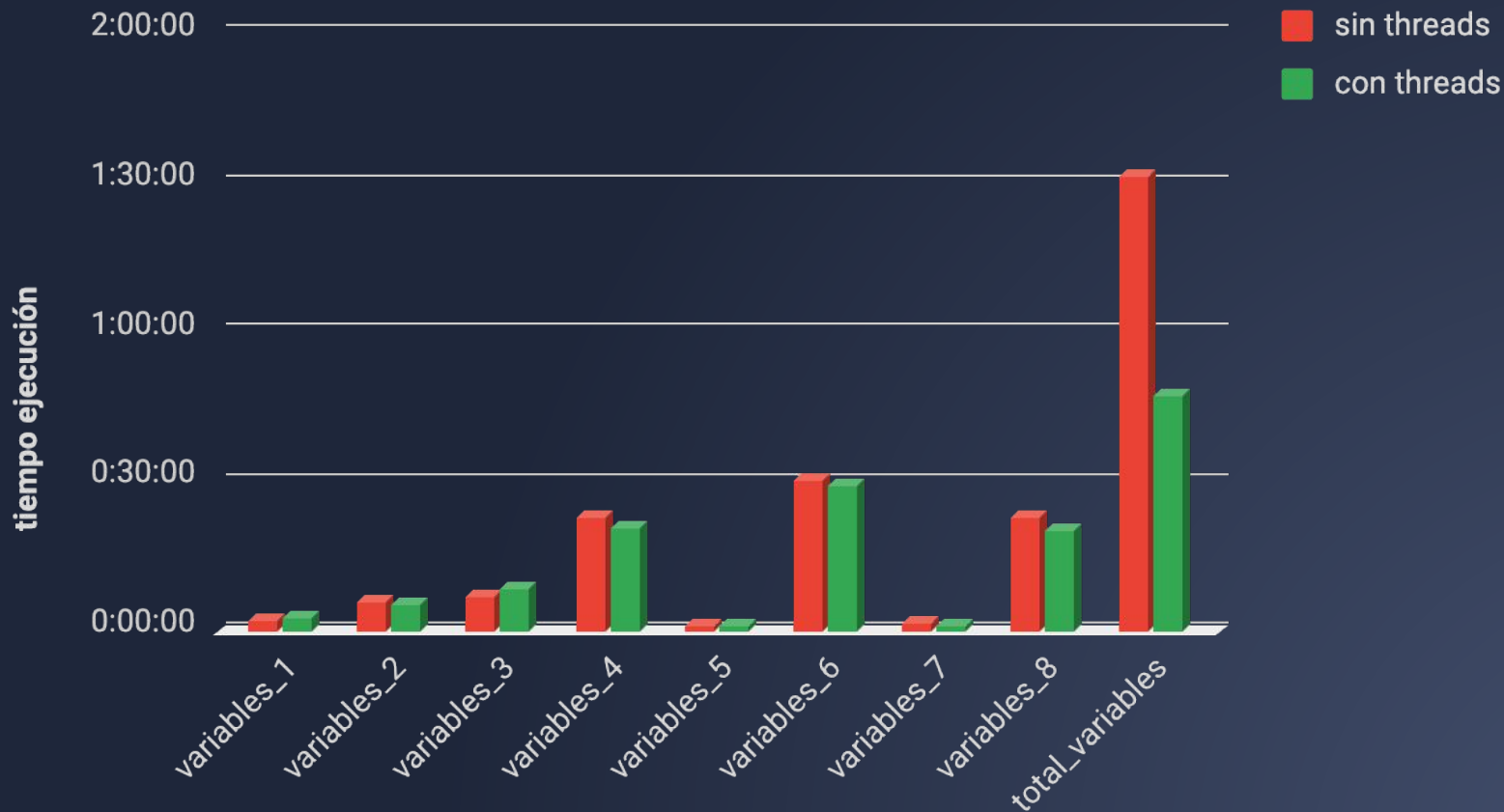
- acceder a los resultados tan pronto estén disponibles
- si salta excepción cancelar el resto de tasks



```
variables_functions = [variables_1, variables_2, variables_3, variables_4, variables_5,
                      variables_6, variables_7, variables_8]
df_variables_final = None
with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
    variables_futures = [executor.submit(the_function, df_population, other_params)
                        for vf in variables_functions]
    for ft in concurrent.futures.as_completed(variables_futures):
        try:
            rdo = ft.result()
            if df_variables_final is None:
                df_variables_final = rdo
            else:
                df_variables_final = df_variables_final.merge(rdo, how="left", on=my_keys)
        except Exception as e:
            canceling = [future.cancel() for future in variables_futures]
            raise e

return df_variables_final
```


Comparación ejecución con y sin threads

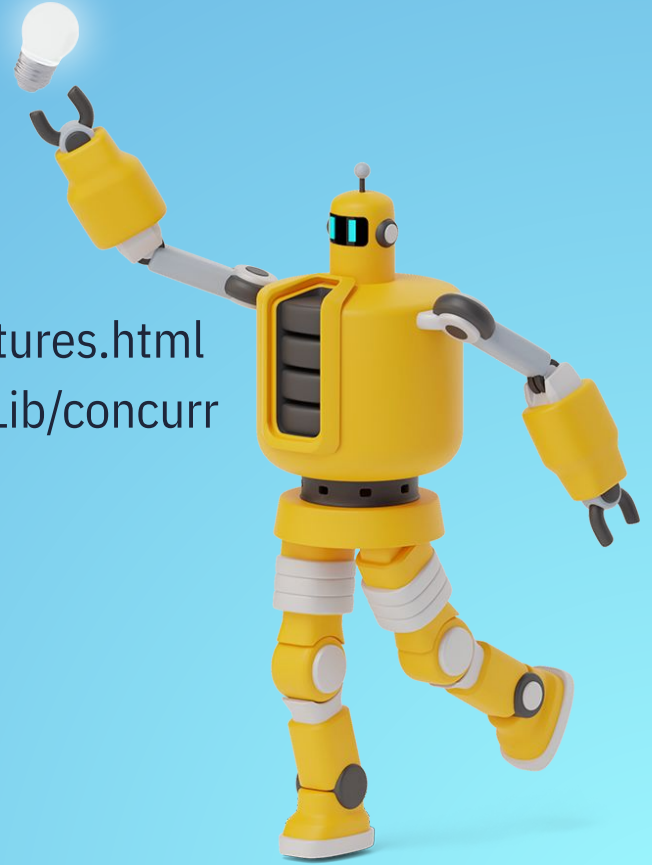


REFERENCIAS

- <https://superfastpython.com/>
- <https://realpython.com/>
- <https://docs.python.org/3/library/concurrent.futures.html>
- <https://github.com/python/cpython/blob/3.10/Lib/concurrent/futures/thread.py>

Free resources:

- Presentation template by [SlidesCarnival](#)
- Robots by [Alex Monge](#)
- Photographs by [Unsplash](#)



ANY QUESTIONS?

