

Getting to Know `asyncio`

يتناول هذا الفصل:

- والفوائد التي يوفرها `asyncio` ما هو
- العمليات (Processes) والConcurrency (Threads) والتوازي (Parallelism)
- التحديات التي يفرضها على التزامن (Global Interpreter Lock)
- كيف يمكن تحقيق التزامن باستخدام `thread sockets` غير المحظورة (Non-blocking sockets)
- أساسيات كيفية عمل التزامن القائم على الحلقة الحديثة (Event-loop-based concurrency)

تطبيقات الويب والعمليات غير المتزامنة

تشمل (I/O) تعتمد العديد من التطبيقات، وخاصة في عالم تطبيقات الويب اليوم، بشكل كبير على عمليات الإدخال/الإخراج: هذه العمليات:

- تنزيل محتويات صفحة ويب من الإنترن特
- التواصل عبر الشبكة مع مجموعة من خدمات الميكروسييرفيس
- تشغيل عدة استعلامات معًا على قاعدة بيانات مثل MySQL أو Postgres

التحديات في عمليات الإدخال/الإخراج

قد تستغرق طلبات الويب أو الاتصالات مع خدمة ميكروسييرفيس مئات الميلي ثانية، أو حتى ثوانٍ إذا كانت الشبكة بطيئة. يمكن أن يكون الاستعلام عن قاعدة بيانات مستهلكًا للوقت، خاصة إذا كانت قاعدة البيانات تحت حمل عالي أو كان الاستعلام معقدًا. قد يحتاج خادم الويب إلى التعامل مع مئات أوآلاف الطلبات في نفس الوقت.

مشكلة الأداء

مرة واحدة يمكن أن يؤدي إلى مشاكل كبيرة في الأداء. إذا قمنا بتشغيل هذه (I/O) إجراء العديد من طلبات الإدخال/الإخراج. الطلبات واحدًا تلو الآخر كما نفعل في تطبيق يعمل بشكل تسلسلي، فسوف نرى تأثيرًا متزايدًا على الأداء.

مثال توضيحي

- إذا كنا نكتب تطبيقًا يحتاج إلى تنزيل 100 صفحة ويب أو تشغيل 100 استعلام، كل منها يستغرق ثانية واحدة للتنفيذ، فسيستغرق تطبيقنا على الأقل 100 ثانية للتشغيل.
- وبأننا التزيلات وانتظرنا في الوقت نفسه، نظرًا، يمكننا إكمال هذه (concurrency) ومع ذلك، إذا استغلينا التزامن "thread":

في بايثون `asyncio` مكتبة

كطريقة إضافية للتعامل مع الأحمال الكبيرة المتزامنة (concurrent) لأول مرة في Python 3.4 تم تقديم مكتبة `asyncio` تم تقديم مكتبة `concurrent.futures` تم تقديم مكتبة `multiprocessing` وتعدد العمليات (multiprocessing).

فوائد `asyncio`:

- يمكن أن يؤدي استخدام الصحيح لهذه المكتبة إلى تحسينات كبيرة في الأداء واستخدام الموارد للتطبيقات التي تعتمد على عمليات الإدخال/الإخراج.
- تتيح لنا بدء العديد من المهام الطويلة التشغيل معًا.

أهداف الفصل

سنستكشف الفروقات **asyncio** في هذا الفصل، سنقدم أساسيات التزامن لفهم كيفية تحقيقه باستخدام بايثون ومكتبة **asyncio** بين:

- العمل المرتبط بوحدة المعالجة المركزية (CPU-bound work) 
- العمل المرتبط بالإدخال/الإخراج (I/O-bound work) 

سنفهم أي نموذج تزامن يناسب احتياجاتنا الخاصة.

المحتوى الذي سنتناوله:

- على (Global Interpreter Lock (GIL)) والتحديات الفريدة التي يفرضها threads التعرف على أساسيات العمليات و التزامن في بايثون.
- مع حلقة الأحداث (non-blocking I/O) فهم كيفية استخدام مفهوم الإدخال/الإخراج غير المحظوظ thread! واحد فقط في بايثون لتحقيق التزامن باستخدام عملية واحدة و (event loop).

إليك النص المنظم مع إضافة الرموز التعبيرية  هذا هو نموذج التزامن الرئيسي لمكتبة **asyncio**:

ما هو asyncio؟

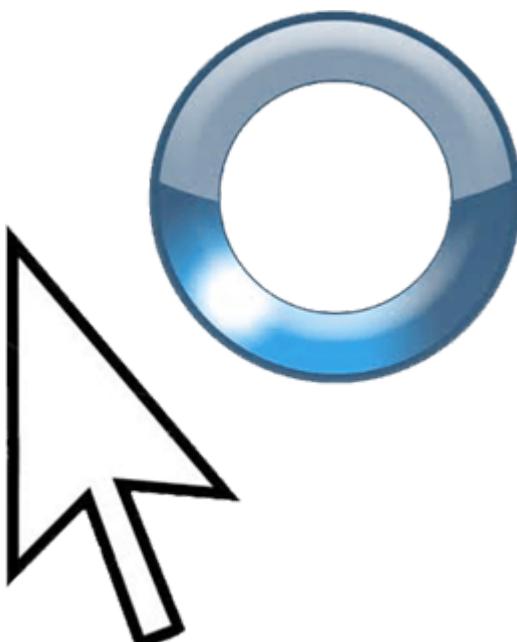
يتم تشغيل الكود بالتسلسل. يتم تنفيذ السطر التالي من الكود، **synchronous application**، في التطبيقات المتزامنة بمجرد أن ينتهي السطر السابق، ولا يحدث سوى شيء واحد في وقت واحد.

التحديات في هذا النموذج 

- هذا النموذج يعمل بشكل جيد للعديد من التطبيقات، إن لم يكن معظمها.
- ولكن ماذا لو كان هناك سطر من الكود بطيء بشكل خاص؟

في هذه الحالة، سيظل باقي الكود بعد هذا السطر البطيء معلقاً في انتظار إكماله. يمكن أن تمنع هذه الأسطر البطيئة المحتملة التطبيق من تشغيل أي كود آخر.

مثال على المشكلة: ربما رأينا هذا من قبل في واجهات المستخدم التي تحتوي على أخطاء، حيث نضغط بسعادة هنا  أو واجهة غير مستجيبة (spinner) وهناك حتى يتجمد التطبيق، مما يتركنا مع دائرة انتظار



هذا مثال على تطبيق يتوقف عن العمل، مما يؤدي إلى تجربة مستخدم سيئة .

إليك النص المنظم مع إضافة الرموز التعبيرية:

التحديات في التطبيقات المتزامنة

بينما يمكن لأي عملية أن تتسبب في توقف التطبيق إذا استغرقت وقتاً طويلاً، فإن العديد من التطبيقات ستتوقف في (I/O) انتظار عمليات الإدخال/الإخراج.

ما هو الإدخال/الإخراج؟

- يشير الإدخال/الإخراج إلى أجهزة إدخال وإخراج الكمبيوتر مثل لوحة المفاتيح، القرص الصلب، والأكثر شيوعاً، بطاقة الشبكة.
- تعتمد على الويب (API) تنتظر هذه العمليات إدخال المستخدم أو استرجاع المحتويات من واجهة برمجة تطبيقات.

سننظر عالقين في انتظار إكمال هذه العمليات قبل أن نتمكن من تشغيل أي شيء آخر. يمكن أن يتسبب ذلك في مشاكل في الأداء والاستجابة، حيث لا يمكننا تشغيل سوى عملية طويلة واحدة في أي وقت معين، وهذه العملية ستوقف تطبيقنا عن القيام بأي شيء آخر.

الحل: التزامن (Concurrency)

بساطة، يعني التزامن السماح بتنفيذ أكثر من مهمة. **Concurrency** أحد الحلول لهذه المشكلة هو إدخال مفهوم التزامن في نفس الوقت. يمكن أن يكون ذلك من خلال تنفيذ عدة عمليات في نفس الوقت أو السماح باتصالات متعددة بخادم الويب في نفس الوقت.

كيفية تحقيق التزامن في بايثون

تشير **asyncio** هناك عدة طرق لتحقيق هذا التزامن في بايثون. واحدة من أحدث الإضافات إلى نظام بايثون هي مكتبة **asyncio** إلى الإدخال/الإخراج غير المتزامن (asynchronous I/O).

* فوائد مكتبة **asyncio**:

- تتيح لنا تشغيل الكود باستخدام نموذج البرمجة التزامن.
- تسمح لنا بالتعامل مع عمليات إدخال/إخراج متعددة في آن واحد، بينما يبقى التطبيق مستجيباً.

ما هي البرمجة غير المتزامنة (I/O)؟

تعني أن مهمة معينة طولية الأمد يمكن تشغيلها في الخلفية بشكل منفصل عن التطبيق الرئيسي. بدلاً من أن يمنع التطبيق من تنفيذ أي كود آخر أثناء انتظار اكتمال تلك المهمة الطويلة، يكون النظام حراً في القيام بأعمال أخرى لا تعتمد على تلك المهمة.

عند اكتمال المهمة الطويلة، سنُخطر بأن المهمة قد انتهت حتى نتمكن من معالجة النتيجة

إليك النص المنظم مع إضافة الرموز التعبيرية وصورة الشكل

التوازي (Parallelism)

إلى أن هناك مهام متعددة تُعالج في نفس الوقت، فإنه لا يعني بالضرورة أنها تعمل معًا (**Concurrency**) بينما يشير التزامن بشكل متوازي.

ما معنى التوازي؟ عندما نقول إن شيئاً ما يعمل بشكل متوازي، نعني بذلك أن هناك مهتمين أو أكثر تحدثان بشكل متزامن، ولكن أيضاً تنفذان في نفس الوقت.

مثال توضيحي: خبز الكعكة

تخيل أننا نملك مساعدة من خباز آخر. في هذه الحالة، يمكننا العمل على الكعكة الأولى بينما نعمل الخباز الثاني على الكعكة الثانية. شخصان يصنعان العجين في نفس الوقت هو مثال على التوازي، لأن لدينا مهتمين مميزتين تعملان بشكل متزامن وتنفذان في نفس الوقت.

الفرق بين التزامن والتوازي

- **Concurrency:** لدينا مهام متعددة تحدث في نفس الوقت، لكننا نقوم بتنفيذ واحدة منها فقط بشكل نشط في لحظة معينة.
- **Parallelism:** لدينا مهام متعددة تحدث ونقوم بتنفيذ أكثر من واحدة منها في نفس الوقت بشكل نشط.

💡 الاستنتاج:

- التزامن يسمح بتعدد المهام، لكن لا يعني أن هذه المهام تنفذ في نفس اللحظة بينما التوازي يعني تنفيذ هذه المهام فعلياً في نفس اللحظة، مما يزيد من كفاءة الأداء عند التعامل مع مهام متعددة.

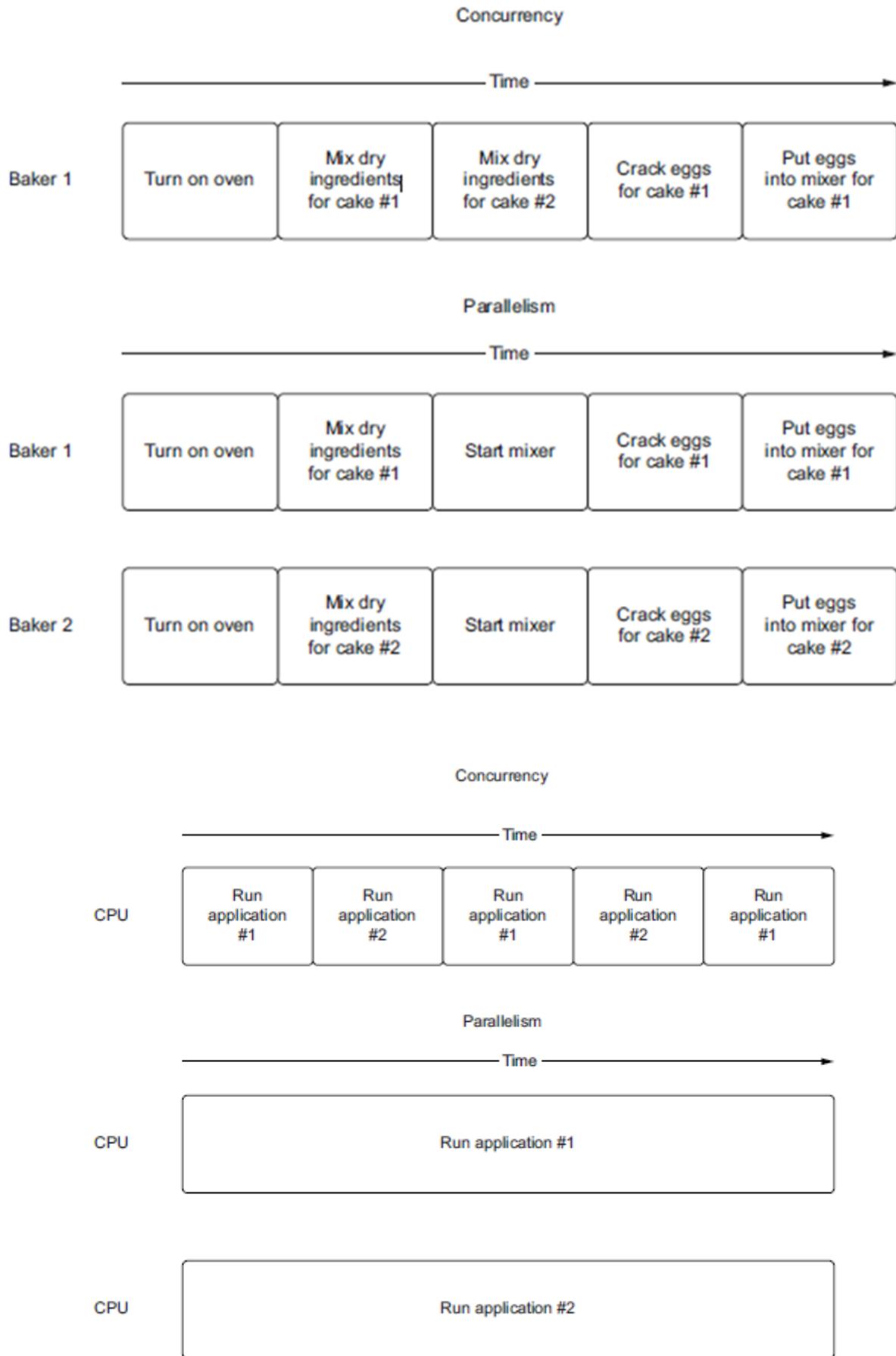


Figure 1.2 With concurrency, we switch between running two applications. With parallelism, we actively run two applications simultaneously.

الفرق بين التزامن (Concurrency) والتوازي (Parallelism)

- يتعلق بوجود مهام متعددة يمكن أن تحدث بشكل مستقل عن بعضها البعض. يمكن تحقيق **Concurrency** (التزامن) تحتوي على نواة واحدة، حيث ستستخدم العملية تعدد المهام القسري (CPU) التزامن حتى على وحدة معالجة مركبة (preemptive multitasking) للتبديل بين المهام.
- يعني أنه يجب علينا تنفيذ مهاترين أو أكثر في نفس الوقت. على جهاز يحتوي على نواة **Parallelism** (التوازي) واحدة، هذا ليس ممكناً. لجعل ذلك ممكناً، تحتاج إلى وحدة معالجة مركبة تحتوي على نوى متعددة يمكنها تشغيل مهاترين معاً.

نقطة مهمة: بينما يشير التوازي إلى إمكانية تنفيذ عدة مهام في نفس الوقت، فإن التزامن لا يعني وجود توازي 🔍.

على جهاز يحتوي على نوى متعددة، فهذا يعني أن **(multithreading)** عندما تتحدث عن تطبيق يستخدم المهام المتعددة التطبيق يمكنه القيام بأكثر من مهمة في نفس الوقت. في هذا النوع من الإعدادات، يمكن تنفيذ مهام متعددة بشكل تزامني، حيث تعمل كل مهمة بشكل مستقل على نواة مختلفة.

ما هو تعدد المهام؟ (What is multitasking?)

تعدد المهام موجود في كل مكان في عالم اليوم. نحن نقوم بتعدد المهام أثناء إعداد الإفطار من خلال تلقي مكالمة أو الرد على رسالة نصية بينما ننتظر غليان الماء لتحضير الشاي. نحن حتى نقوم بتعدد المهام أثناء التنقل إلى العمل، من خلال قراءة كتاب بينما تأخذنا القطار إلى المحطة.

نوعان رئيسيان من تعدد المهام:

1. تعدد المهام القسري (Preemptive Multitasking):

- في هذا النموذج، نترك نظام التشغيل يقرر كيفية التبديل بين المهام التي يتم تنفيذها حالياً من خلال عملية **Time Slicing**.
- عندما يقوم نظام التشغيل بالتبديل بين الأعمال، نطلق عليه اسم **Preempting**.
- يتم إدارة تشغيل المهام بواسطة نظام التشغيل، حيث يقوم بتحديد متى وكيف ينتقل بين المهام المختلفة.

كيف يعمل؟

◦ تقسيم الوقت (Time Slicing):

- يقوم نظام التشغيل بتقسيم الوقت المتاح لكل مهمة إلى فترات زمنية قصيرة تُعرف باسم "شريحة الوقت".
- يتم تخصيص كل شريحة زمنية لمهمة معينة، مما يسمح لنظام التشغيل بتنفيذ العديد من المهام بشكل متسلسل.

◦ الإيقاف القسري (Preempting):

- عندما تنتهي شريحة الوقت لمهمة معينة، يقوم نظام التشغيل بإيقاف هذه المهمة (أو "الإيقاف القسري") والانتقال إلى مهمة أخرى.
- هذا يعني أن المهام يمكن أن تتوقف في أي وقت إذا استهلكت وقتها المحدد، حتى لو كانت لا تزال تعمل.

2. تعدد المهام التعاوني (Cooperative Multitasking):

- هذا النموذج هو أسلوب لإدارة المهام حيث يتم التحكم بتبديل المهام بشكل صريح بواسطة التطبيق نفسه، وليس بواسطة نظام التشغيل.
- في هذا النموذج، تقوم كل مهمة في التطبيق بتحديد نقاط معينة في الكود حيث يمكنها التوقف مؤقتاً والسماح لمهام أخرى بالعمل.

⑤ كيفية عمله:

- تقوم المهام في هذا النظام بالتعاون مع بعضها البعض، حيث تقول كل مهمة بشكل صريح: "سأتوقف عن العمل مؤقتاً؛ يمكنكم الآن تشغيل المهام الأخرى".
- هذا الأسلوب يتطلب من المبرمجين كتابة الكود بحيث تكون هناك نقاط يمكن عندها التبديل بين المهام، مما يضمن أن جميع المهام تحصل على فرصتها للعمل دون تدخل مباشر من نظام التشغيل.
- على سبيل المثال، إذا كانت مهمة معينة تحتاج إلى انتظار مدخلات من المستخدم، يمكنها التوقف عند تلك النقطة والسماح لمهام أخرى بالعمل حتى تتلقى المدخلات المطلوبة و تستأنف عملها.

إليك النص مع إضافة بعض الرموز التعبيرية لتحسين العرض:

-Cooperative Multitasking) فوائد تعدد المهام التعاوني

تعدد المهام التعاوني لتحقيق التزامن. عندما تصل تطبيقاتنا إلى نقطة قد تضطر فيها Python في `asyncio` تستخدم مكتبة للانتظار لبعض الوقت للحصول على نتيجة، نقوم بتمييز هذه النقطة في الكود بشكل صريح. هذا يسمح لجزء آخر من الكود بالعمل أثناء انتظارنا للحصول على النتيجة في الخلفية

فائدة هذا الأسلوب: بمجرد الانتهاء من المهمة التي قمنا بتمييزها، يمكننا "الاستيقاظ" واستئناف تنفيذ المهمة. هذا (Parallelism)، يمنحنا شكلاً من أشكال التزامن، حيث يمكن أن تبدأ المهام في نفس الوقت، ولكن بشكل مهم، ليس التوازي لأنهم لا ينفذون الكود في نفس الوقت.

Preemptive Multitasking) فوائد تعدد المهام التعاوني مقارنة بتعدد المهام القسري

1. استهلاك الموارد:

- أو عملية قيد thread تعدد المهام التعاوني أقل استهلاكاً للموارد. عندما يحتاج نظام التشغيل إلى التبديل بين (Context Switch). التسليق، يتطلب ذلك عملية تُسمى التبديل السياقي
- thread تعتبر عمليات التبديل السياقي مكثفة لأن نظام التشغيل يجب أن يحتفظ بمعلومات حول العملية أو الذي يتم تشغيله ليكون قادرًا على إعادة تحميله.

2. الدقة:

- أو المهمة يجب أن تتوقف بناءً على أي خوارزمية جدولة يستخدمها، ولكن قد thread يعرف نظام التشغيل أن لا يكون ذلك هو أفضل وقت للتوقف
- مع تعدد المهام التعاوني، نقوم بتمييز المناطق التي تكون الأفضل لتوقف مهامنا. هذا يمنحنا بعض مكاسب الكفاءة لأننا نتبادل المهام فقط عندما نعرف صراحة أنه الوقت المناسب للقيام بذلك

(Multiprocessing) و تعدد العمليات، threads، (Multithreading) فهم العمليات،



وال (Processes) تحتاج أولاً إلى فهم الأساسيةيات حول كيفية عمل العمليات. لفهم كيفية عمل التزامن في عالم **Multithreading** (Threads) ثم سنستعرض كيفية استخدام تعدد المهام المتعدد (Multiprocessing) للأداء الأعمالي بشكل متزامن.

إذا كان هناك أي شيء آخر تود إضافته أو تغييره، فأخبرني!

تعريفات أساسية:

- **(Process) العملية:**

- العملية هي تطبيق يتم تشغيله وله مساحة ذاكرة لا يمكن للتطبيقات الأخرى الوصول إليها في سطر **python** أو كتابة "hello world" سيكون تشغيل تطبيق بسيط مثل Python مثل على إنشاء عملية الأوامر لبدء بيئة العمل التفاعلية (REPL).

ملاحظة: يمكن تشغيل عمليات متعددة على جهاز واحد. إذا كنا على جهاز يحتوي على وحدة معالجة مركبة (CPU) بها أنوية متعددة، يمكننا تنفيذ عمليات متعددة في نفس الوقت.

- إذا كنا على وحدة معالجة مركبة تحتوي على نواة واحدة فقط، يمكننا أيضًا تشغيل تطبيقات متعددة في وقت واحد من خلال تقنيات **تقسيم الوقت (Time Slicing)**.

- عندما يستخدم نظام التشغيل تقسيم الوقت، فإنه سيتحول بين العمليات التي تعمل تلقائيًا بعد فترة زمنية معينة. تختلف الخوارزميات التي تحدد متى يحدث هذا التبديل حسب نظام التشغيل.

(Threads)

كعمليات أخف وزنًا. بالإضافة إلى ذلك، فإنها تعتبر أصغر وحدة يمكن إدارتها بواسطة نظام التشغيل. **Threads** يمكن اعتبار ذاكرة خاصة بها كما تفعل العمليات؛ بدلاً من ذلك، تشارك ذاكرة العملية التي أنشأتها. ترتبط **Threads** لا تمتلك واحد على الأقل مرتبط بها، وعادة ما يُعرف بـ **main thread**. ستحتوي العملية دائمًا على

(**worker** or **background threads**) العاملة أو الخلية **multithreading** أخرى، والتي تُعرف بشكل أكثر شيوعًا **Threads** تتشبه **Threads** **concurrently** مع **main thread**. يمكن أن تؤدي هذه العمليات، حيث يمكن تشغيلها جنبًا إلى جنب على وحدة معالجة مركبة متعددة النوى، ويمكن أيضًا أن يتبدل نظام التشغيل بينها عبر تقسيم الوقت.

الذي سيكون مسؤولاً عن تشغيل تطبيق **main thread**، يقوم بإنشاء عملية بالإضافة إلى Python عند تشغيل تطبيق Python الخاص بنا.

بسط بسيط في تطبيق Python للعمليات والـ

لفهم **Threads**, **Multithreading**, و **Multiprocessing**:

```
import threading

total_threads = threading.active_count()
thread_name = threading.current_thread().name
print(f'Python is currently running {total_threads} thread(s)')
print(f'The current thread is {thread_name}')
```

النشطة حالياً باسم الـ **threads** في Python عرض عدد الـ **threading** في هذا المثال البسيط يعرض كيفية استخدام المكتبة **thread** الحالي.

وهو معرف، **process ID**، أولاً، نحصل على معرف العملية **main Thread**. نقوم بإنشاء تطبيق بسيط ليوضح لنا أساسيات النشطة **threads** فريد للعملية، ونطبعه لإثبات أن لدينا بالفعل عملية مخصصة قيد التشغيل. بعد ذلك، نحصل على عدد الـ

بينما سيكون معرف **thread** واحداً وهو **main thread**. أنتا نشغل **thread** التي تعمل بالإضافة إلى اسم الـ **process** الذي لا يظهر أبداً. وإن تشغيل القائمة 1.2 سيعطي إخراجاً مشابهاً لما يلي:

```
import threading
import os

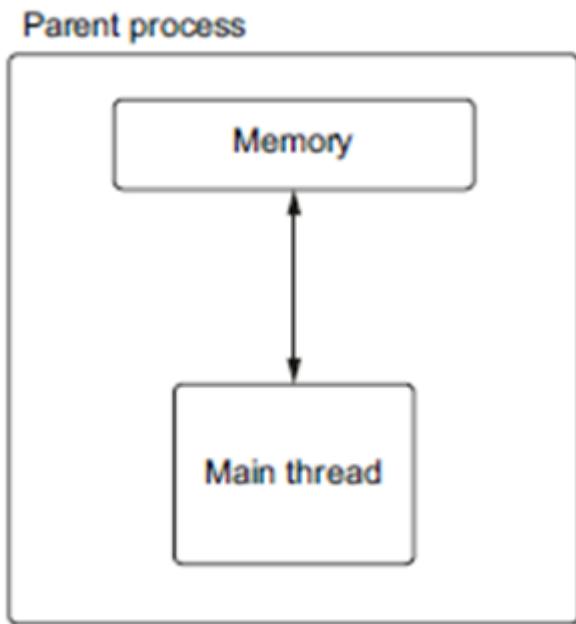
process_id = os.getpid() # Get the current process ID
total_threads = threading.active_count() # Get the number of active threads
thread_name = threading.current_thread().name # Get the name of the current thread

print(f'Process ID: {process_id}')
print(f'Python is currently running {total_threads} thread(s)')
print(f'The current thread is {thread_name}')
```

الإخراج المتوقع:

```
Process ID: 12345
Python is currently running 1 thread(s)
The current thread is MainThread
```

سيكون 1 إذا لم نقم بإنشاء أي **threads** سيختلف في كل مرة يتم فيها تشغيل البرنامج، لكن عدد الـ **Process ID** لا يختلف.



إليك النسخة المعدلة باستخدام المصطلحات الإنجليزية:

واحد يقرأ من الذاكرة مع Main Thread

نقوم بإنشاء تطبيق بسيط يوضح لنا أساسيات الـ **Process** في **الشكل السابق**. نرسم الـ **Process** ونقوم بطبعته لإثبات أن لدينا بالفعل **Process** وهو معرف فريد له. أولاً، نحصل على معرف الـ **Threads** التي تعمل حالياً بالإضافة إلى اسم الـ **Threads** مخصصة قيد التشغيل. بعد ذلك، نحصل على عدد الـ **Threads**.

واحداً — وهو الـ Thread الحالي لإظهار أننا نشغل Main Thread.

مختلفاً في كل مرة يتم فيها تشغيل هذا الكود، فإن تشغيل القائمة 1.2 سيعطي إخراجاً Process بينما سيكون معرف الـ مشابهاً لما يلي:

```
Python process running with process id: 98230
Python currently running 1 thread(s)
The current thread is MainThread
```

القيام Threads الرئيسية. يمكن لهذه الـ Process أخرى تشارك الذاكرة الخاصة بالـ Threads إنشاء Process يمكن أيضاً للـ بأعمال أخرى بشكل متزامن لنا من خلال ما يُعرف بـ **multithreading**.

1.3: إنشاء تطبيق متعدد الـ Python Threads

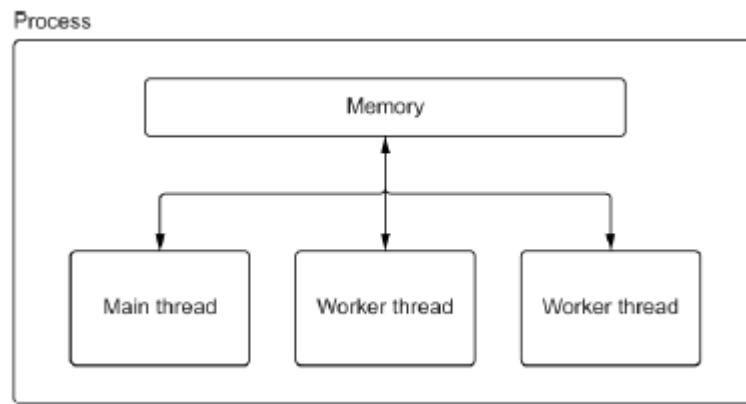
```
import threading

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

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

total_threads = threading.active_count()
thread_name = threading.current_thread().name
print(f'Python is currently running {total_threads} thread(s)')
print(f'The current thread is {thread_name}')

hello_thread.join()
```



الخاصية ProcessThreads في الشكل لتشغيل تلك الدالة. بعد ذلك، نستدعي Thread الحالي، ثم نقوم بإنشاء Thread بالكود. نقوم بإنشاء دالة لطباعة اسم الـ والتي ستؤدي إلى إيقاف البرنامج حتى ينتهي الـ join لبدء تشغيله. أخيراً، نستدعي دالة Thread الخاصة بالـ start دالة Thread الذي بدأناه.

إذا قمنا بتشغيل الكود السابق، سنرى إخراجاً مشابهاً لما يلي:

```
Hello from thread <Thread(Thread-1, started 123145541312512)>!
Python is currently running 2 thread(s)
The current thread is MainThread
```

لاحظ أنه عند تشغيل هذا الكود، قد ترى رسالتي "Hello from thread" و "Python is currently running 2 thread(s)" ؛ سنتكشف قليلاً عن ذلك في القسم التالي وفي **race condition** تطبعان في نفس السطر. هذه مشكلة تُعرف بـ **race condition**، لأننا مقيدون بـ **القفل العالمي للمترجم (Global Interpreter Lock)** (GIL) المدخلات والمخرجات. سيتم مناقشته في القسم التالي.

وسيلة شائعة لتحقيق التزامن في العديد من لغات البرمجة. ومع ذلك، هناك بعض **Threads** تُعد التطبيقات متعددة الـ **Threads** مفيدة فقط للأعمال المعتمدة على **Threads**. في Python، التحديات في استخدام التزامن مع الـ **I/O-bound work**، الذي يعيّن مقيدين بـ **القفل العالمي للمترجم (Global Interpreter Lock)** لأننا مقيدون بـ **القفل العالمي للمترجم (Global Interpreter Lock)** لأننا مقيدين بـ **القفل العالمي للمترجم (Global Interpreter Lock)**.

ليس الطريقة الوحيدة التي يمكننا من خلالها تحقيق التزامن؛ يمكننا أيضًا إنشاء عمليات متعددة للقيام **Threads** **تعدد الـ Threads** في **تعدد العمليات**، تقوم الـ **Parent Process** **multiprocessing** (بالأعمال بالتوالي). تُعرف هذه الطريقة بـ **تعدد العمليات** **Child Processes**. بعد ذلك، يمكنها توزيع العمل على الـ **Child Processes** بإنشاء واحدة أو أكثر من الـ **Child Processes**.

للتعامل مع هذا، تُعد واجهة البرمجة الخاصة بها مشابهة جدًا لتلك الموجودة في Python **multiprocessing** وحدة **target function**. جديدة باستخدام دالة مستهدفة **Process** نقوم أولاً بإنشاء **threading** وحدة **start** دالة **join** للانتظار حتى تنتهي الـ **Process** الخاصة بالـ **Process**.

يظهر في الكود إنشاء عمليات متعددة

```
import multiprocessing
import os

def hello_from_process():
    print(f'Hello from child process {os.getpid()}!')

if __name__ == '__main__':
    hello_process = multiprocessing.Process(target=hello_from_process)
    hello_process.start()

    print(f'Hello from parent process {os.getpid()}')
    hello_process.join()
```

الخاصة بها كما هو موضح في الكود السابق. نقوم بإنشاء طريقة لطباعة اسم **thread** في الشكل السابق، نعرض عملية **thread** للخيط لبدء تشغيله. أخيراً، نستدعى طريقة **start** الدالة. بعد ذلك، نستدعى طريقة **join** الحالي، ثم ننشئ **thread** في **__main__** حتى يكتمل تشغيل الخيط الذي بدأناه **join** ستنسب **join** نستدعى دالة **join**.

إذا قمنا بتشغيل الكود السابق، سنرى مخرجات مشابهة لما يلي:

```
Hello from thread <Thread(Thread-1, started 123145541312512)>!
Python is currently running 2 thread(s)
The current thread is MainThread
```

تظهر على "hello from thread" و "python is currently running 2 thread(s)" لاحظ أنه عند تشغيل هذا، قد ترى رسائل ؛ وسنستكشف القليل عن ذلك في القسم التالي وفي الفصول 6 و 7 (race condition) نفس السطر. هذه حالة سباق.

تعتبر التطبيقات المتعددة وسيلة شائعة لتحقيق التزامن في العديد من لغات البرمجة. ومع ذلك، هناك بعض التحديات في مفيدة فقط للعمل الذي يعتمد على الإدخال / threads في بايثون. تعتبر البرمجة المتعددة threads استخدام التزامن مع الذي سيتم مناقشه في القسم 1.5، لأننا مقيدون بـ GIL (Global Interpreter Lock) أو O-bound (I/O-bound) الإخراج.

الطريقة الوحيدة لتحقيق التزامن؛ يمكننا أيضًا إنشاء عمليات متعددة للقيام بالعمل threads لا تعد البرمجة المتعددة في البرمجة المتعددة العمليات، تقوم العملية (multiprocessing). يقوم ذلك باسم "البرمجة المتعددة العمليات". الأهم بإنشاء واحدة أو أكثر من العمليات الفرعية التي تديرها. ثم يمكنها توزيع العمل على العمليات الفرعية.

مشابهة لوحدة API للتعامل مع هذا. واجهة برمجة التطبيقات `multiprocessing` تقدم بايثون لنا وحدة `threading`. لانتظار `join` لتنفيذها وأخيرًا طريقة `start` ثم نستدعي طريقة (target function). نقوم أولاً بإنشاء عملية مع دالة الهدف الخاصة بها كما هو موضح في الكود السابق. نقوم بإنشاء عملية threads اكتمال تشغيلها. في الكود التالي، نعرض عملية الخاص بها، ونقوم أيضًا بطباعة معرف العملية الأم لإثبات أننا نقوم (process ID) فرعية واحدة تطبع معرف العملية بتشغيل عمليات مختلفة.

عادةً ما تكون البرمجة المتعددة العمليات أفضل عندما يكون لدينا عمل مكثف على وحدة المعالجة المركزية (CPU-intensive work).

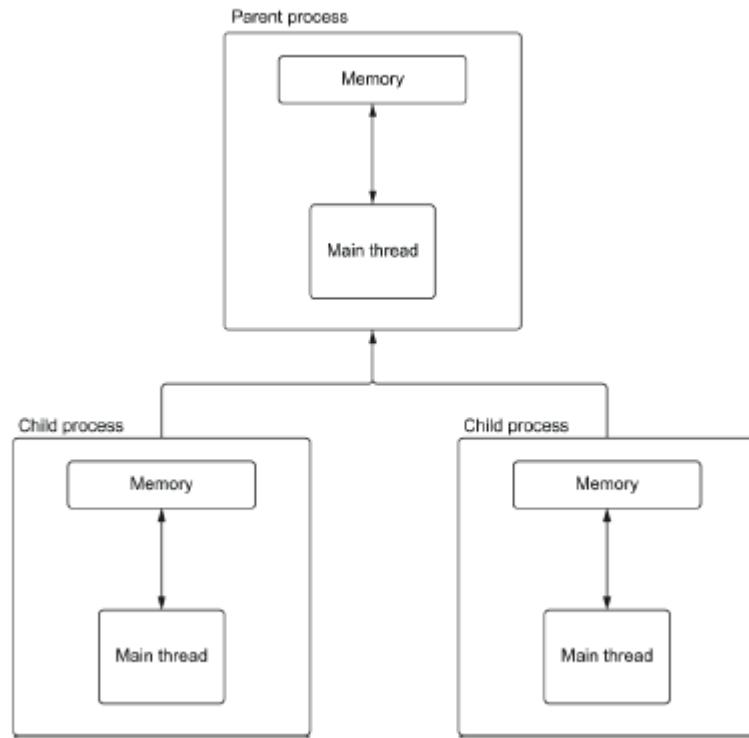
والبرمجة المتعددة العمليات وكأنها حلول سحرية لتمكن التزامن في بايثون. ومع ذلك، threads قد تبدو البرمجة المتعددة - فإن قوة هذه النماذج في التزامن تعيقها تفاصيل التنفيذ في بايثون GIL.

إنشاء عمليات متعددة

```
import multiprocessing
import os

def hello_from_process():
    print(f'Hello from child process {os.getpid()}!')

if __name__ == '__main__':
    hello_process = multiprocessing.Process(target=hello_from_process)
    hello_process.start()
    print(f'Hello from parent process {os.getpid()}')
    hello_process.join()
```



إليك النص المترجم والمتنسق ليكون

مناسباً للاستخدام في تنسيق Markdown:

فهم القفل العام للمترجم (Global Interpreter Lock - GIL)

عملية بايثون واحدة GIL هو موضوع مثير للجدل في مجتمع بايثون. بإيجاز، يمنع الـ **GIL** القفل العام للمترجم، المختصر بـ على جهاز مزود **threads** من تنفيذ أكثر من تعليمات بايت في وقت واحد. وهذا يعني أنه حتى لو كان لدينا عدة واحدة فقط في وقت معين. في عالم يحتوي على وحدات **thread** بأئوية متعددة، يمكن لعملية بايثون واحدة أن تعمل بها معالجة مركزية متعددة النوى، يمكن أن يشكل ذلك تحدياً كبيراً لمطوري بايثون الذين يسعون للاستفادة من **multiprocessing** لتحسين أداء تطبيقاتهم.

تشغيل تعليمات بايت متعددة في وقت واحد، لأن كل عملية بايثون لديها قفل **multiprocessing** ملاحظة: يمكن لـ GIL خاص بها.

لماذا يوجد الـ GIL؟

تتم إدارة الذاكرة بشكل أساسى بواسطة عملية تُعرف بـ **CPython**. الجواب يكمن في كيفية إدارة الذاكرة في يعمل العد المرجعي عن طريق تتبع من يحتاج حالياً للوصول إلى كائن بايثون معين، (Reference Counting). العد المرجعي مثل عدد صحيح، أو قاموس، أو قائمة.

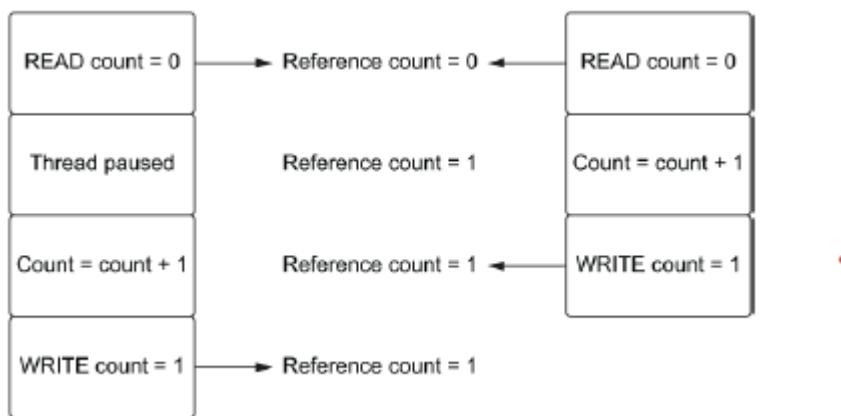
- العد المرجعي هو عدد صحيح يتبع عدد الأماكن التي تشير إلى ذلك الكائن المحدد.
- عندما لا يحتاج شخص ما إلى ذلك الكائن، يتم تقليل العد المرجعي، وعندما يحتاج شخص آخر إليه، يتم زيادته.
- عندما يصل العد المرجعي إلى الصفر، لا أحد يشير إلى الكائن، ويمكن حذفه من الذاكرة.

ما هو CPython؟

هو التطبيق المرجعي للغة بايثون. بمعنى التنفيذ القياسي للغة ويستخدم كمرجع **CPython** المصممة للعمل على **Jython** للسلوك الصحيح للغة. هناك تطبيقات أخرى للغة بايثون مثل **Java Virtual Machine**، **.NET**، **IronPython**، **PyPy**، **PyPy3**، **PyPyC**، **PyPyJ**، **PyPyS**، **PyPyT**، **PyPyX**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**، **PyPyF**، **PyPyG**، **PyPyH**، **PyPyI**، **PyPyJ**، **PyPyK**، **PyPyL**، **PyPyM**، **PyPyN**، **PyPyO**، **PyPyP**، **PyPyQ**، **PyPyR**، **PyPyS**، **PyPyT**، **PyPyU**، **PyPyV**، **PyPyW**، **PyPyX**، **PyPyY**، **PyPyZ**، **PyPyA**، **PyPyB**، **PyPyC**، **PyPyD**، **PyPyE**

ليس آمناً ضد **CPython** عندما نقول إن **threads** ليس آمناً ضد **CPython** من أن التنفيذ في **threads** تتبع المشكلة مع التعديل متغير مشترك، فقد ينتهي هذا المتغير في حالة غير **threads** يعني أنه إذا قامت اثنان أو أكثر من **threads** إلى المتغير، المعروف عموماً بـ **حالة السباق threads** متوقعة. هذه الحالة غير المتوقعة تعتمد على ترتيب وصول **Race Condition**.

إلى الإشارة إلى كائن بايثون في نفس الوقت. كما هو موضح **threads** يمكن أن تنشأ حالات السباق عندما تحتاج اثنان من واحدة في العد المرجعي في وقت واحد، قد نواجه حالة حيث تتسبب **threads** في الشكل التالي إذا زادت اثنان من الأخرى تستخدم الكائن. النتيجة المحتملة لهذا ستكون انهيار التطبيق **thread** أن يكون العد المرجعي صفرًا بينما لا تزال عندما نحاول قراءة الذاكرة المحدوفة المحتملة.



زيادة العد **threads** حيث تحاول اثنان من الـ **race condition** الشكل السادس حالة المرجعي في نفس الوقت. بدلاً من الحصول على عدد متوقع يساوي اثنين، نحصل على واحد.

دعونا نبحث في مهمة تتطلب موارد معالجة مركزية عالية وهي حساب **multithreaded** على برمجة **GIL** لتوضيح تأثيره سنستخدم تنفيذًا بطيئًا نسبيًا للخوارزمية لتبسيط توضيح عملية تستغرق وقتًا طويلاً. الحل **Fibonacci** في تسلسل **nth** العدد أو تقنيات رياضية لتحسين الأداء **memoization** المناسب سيستخدم تقنية.

قائمة 1.5: توليد وتوقيت تسلسل Fibonacci

```

import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

    print(f'fib({number}) is {fib(number)}')

def fibs_no_threading():
    print_fib(40)
    print_fib(41)

```

```

start = time.time()
fibs_no_threading()
end = time.time()
print(f'Completed in {end - start:.4f} seconds.')

```

لإكمالها. إذا كنا في موقف $O(2^N)$ وهو خوارزمية بطيئة نسبياً، تتطلب وقتاً أُنْثِيّاً (recursion) يستخدم هذا التنفيذ التكراري فمن السهل استدعاؤهما بشكل متزامن وقياس النتيجة، كما فعلنا في القائمة Fibonacci، نحتاج فيه إلى طباعة عددين من السابقة.

اعتماداً على سرعة وحدة المعالجة المركزية التي تشغله، سنرى توقيتات مختلفة، لكن تشغيل الكود في قائمة 1.5 سيعطي مخرجات مشابهة لما يلي:

```

fib(40) is 63245986
fib(41) is 102334155
Completed in 65.1516 seconds.

```

مستقلة عن بعضها البعض. وهذا يعني أنه يمكن `print_fibs` هذه عملية حساب طويلة إلى حد ما، لكن استدعاءاتنا إلى يمكن لوحدتنا المعالجة، نظرياً، تشغيلها بشكل متزامن على نوى متعددة، وبالتالي تسرير تطبيقنا `threads` وضعها في عدة.

1.6: تنفيذ تسلسل Fibonacci باستخدام multithreading

```

import threading
import time

def print_fib(number: int) -> None:
    def fib(n: int) -> int:
        if n == 1:
            return 0
        elif n == 2:
            return 1
        else:
            return fib(n - 1) + fib(n - 2)

    def fibs_with_threads():
        fortieth_thread = threading.Thread(target=fib, args=(40,))
        forty_first_thread = threading.Thread(target=fib, args=(41,))

        fortieth_thread.start()
        forty_first_thread.start()

        fortieth_thread.join()
        forty_first_thread.join()

    start_threads = time.time()
    fibs_with_threads()
    end_threads = time.time()
    print(f'Threads took {end_threads - start_threads:.4f} seconds.')

```

ونبدأهما بشكل ،fib(40) والأخر لحساب threads، في القائمة السابقة، نقوم بإنشاء اثنين من مما سيجعل برنامجنا الرئيسي ينتظر حتى، join() ثم نقوم باستدعاء thread. لكل thread متزامن عن طريق استدعاء في نفس الوقت وقمنا بتشغيلهما بشكل متزامن، قد تعتقد fib(40) و fib(41) بالنظر إلى أننا بدأنا حساب threads. تنتهي أننا يمكن أن نرى ترتيباً معقولاً؛ ومع ذلك، سنرى مخرجات مشابهة لما يلي حتى على جهاز مزود بأనوية متعددة:

```
fib(40) is 63245986
fib(41) is 102334155
Threads took 66.1059 seconds.
```

تقريراً نفس كمية الوقت. في الواقع، كانت أبطأ قليلاً! هذا يرجع تقريراً بالكامل إلى أن استغرقت النسخة باستخدام threads تعمل بشكل متزامن، إلا أنه يُسمح لواحدة فقط منها threads بينما من الصحيح أن threads وتكليف إنشاء وإدارة GIL الأخرى في حالة انتظار حتى تكتمل الأولى، مما thread بتشغيل كود بايثون في وقت معين بسبب القفل. هذا يترك المتعددة threads تماماً قيمة.

هل يتم تحرير GIL؟

نظراً لأن Python باستخدام threads بناءً على المثال السابق، قد تتساءل عما إذا كان يمكن تحقيق التزامن في لا يتم الاحتفاظ به إلى الأبد، مما يتتيح لنا GIL في نفس الوقت. ومع ذلك، فإن Python يمنع تشغيل سطرين من كود لصالحنا threads استخدام.

لتنفيذ الأعمال المتزامنة فيما يتعلق threads هذا يسمح لنا باستخدام GIL يتم تحرير I/O هناك بعض الاستثناءات (CPU-bound) المرهق للمعالج Python ولكن ليس عند التعامل مع كود I/O بعمليات (عند القيام بعمليات معالجة ثقيلة في ظروف معينة، وستتناولها في فصل لاحق GIL الملحوظة التي تحرر).

لصفحة ويب status code لشرح ذلك، دعنا نستخدم مثلاً بسيطاً لقراءة.

```
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:.4f} seconds.')
```

مرتين. بناءً على سرعة الاتصال بالإنترنت status code في الكود أعلاه، نقوم بجلب محتويات example.com وطباعة status code في موقعك، قد تحصل على نتائج مشابهة لما يلي:

```
200
200
Running synchronously took 0.2306 seconds.
```

يمكّنا كتابة نسخة متعددة لـ **threads** الآن بعد أن حصلنا على أساس لما يbedo عليه الإصدار المتزامن للمقارنة.

لقراءة threads نسخة متعددة الـ status code:

```
import time
import threading
import requests

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

# Creating two threads
thread_1 = threading.Thread(target=read_example)
thread_2 = threading.Thread(target=read_example)

# Timing the thread execution
thread_start = time.time()
thread_1.start()
thread_2.start()
print('All threads running!')

# Waiting for the threads to complete
thread_1.join()
thread_2.join()

thread_end = time.time()
print(f'Running with threads took {thread_end - thread_start:.4f} seconds.')
```

عندما ننفذ هذا الكود، سنرى نتائج مشابهة لما يلي، بناءً على سرعة الاتصال بالإنترنت ومواصفات الجهاز ⚡:

```
All threads running!
200
200
Running with threads took 0.0977 seconds.
```

حيث تم تشغيل الطلبيين في نفس الوقت تقريرياً، **threads** هذا أسرع بمرتين تقريباً من النسخة الأصلية التي لم تستخدم ⚡.

ولكن ليس لعمليات I/O لعمليات GIL كيف يتم تحرير ⚡؟

التي تتم في الخلفية تكون خارج تشغيل (system calls) لأن الاستدعاءات النظامية I/O في حالة **GIL** يتم تحرير ⚡ فقط عند ترجمة **GIL** يتم إعادة اكتساب **Python**. لأن النظام لا يتعامل مباشرة مع كائنات **GIL** يسمح هذا بتحرير البيانات المستلمة مرة أخرى إلى كائنات **Python**.

على مستوى النظام بشكل متزامن، مما يوفر التوازي I/O في هذه الحالة، تعمل عمليات ⚡.

asyncio و GIL

واحد فقط. عند استخدامنا **thread** مما يمنحك التزامن حتى مع وجود **I/O** تحرر **GIL** يستغل حقيقة أن عمليات **asyncio** خفيف **thread** على أنها تعمل مثل **coroutine** يمكن التفكير في **coroutines**. يقوم بإنشاء كائنات تدعى **asyncio** لـ الوزن.

متزامنة، يمكن أن **I/O** تعمل في نفس الوقت، كل منها يتعامل مع عملية **threads** تماماً كما يمكن أن يكون لدينا عدة **threads** يمكننا **I/O** المتعلقة بـ **coroutines** تعمل بجانب بعضها البعض. بينما ننتظر انتهاء **coroutines** يكون لدينا العديد من آخر، مما يمنحك التزامن **Python** الاستمرار في تنفيذ كود.

ولا زلنا خاضعين لها. إذا كان لدينا مهمة تتطلب معالجة مكثفة للمعالج، لا تتجاوز **asyncio** من المهم ملاحظة أن **(CPU-bound)**؛ وإلا، **asyncio** فإننا نحتاج لاستخدام عمليات متعددة لتنفيذها بالتزامن (يمكن القيام بذلك مع **sockets**، سبب مشاكل في الأداء في تطبيقنا).

غير المحظورة؟ **sockets** كيف يعمل ذلك مع **asyncio**؟

غير المحظورة، حيث توفر لنا القدرة على إدارة عدة اتصالات **sockets** مع **asyncio** لنبدأ في استكشاف تفاصيل كيفية عمل **I/O** في نفس الوقت، مما يساعدهم في تحسين أداء التطبيقات التي تعتمد على

غير المحظورة **asyncio** مع **sockets** مثال على استخدام

```
import asyncio
import socket

async def fetch_data(host, port):
    reader, writer = await asyncio.open_connection(host, port)
    # Format the host as a string in the bytes literal
    writer.write(f'GET / HTTP/1.0\r\nHost: {host}\r\n\r\n'.encode())
    await writer.drain() # Ensure data is sent

    data = await reader.read(100)
    print(f'Received: {data.decode()}')

    writer.close()
    await writer.wait_closed()

async def main():
    hosts = [('www.example.com', 80), ('www.google.com', 80)]
    tasks = [fetch_data(host, port) for host, port in hosts]
    await asyncio.gather(*tasks)

asyncio.run(main())
```

شرح الكود:

بالطبع! لنغوص في تفاصيل الكود خطوة بخطوة. سأشرح كل جزء من العملية التي تتضمن فتح اتصال غير محظوظ، وكتابة البيانات، وقراءة البيانات، وإغلاق الاتصال.

فتح الاتصال 1.

نقوم بفتح اتصال غير محظوظ مع الخادم. هذا يعني أننا لن نقوم بوقف `asyncio.open_connection` عند استخدام دالة تنفيذ البرنامج أثناء انتظار الاتصال.

```
reader, writer = await asyncio.open_connection(host, port)
```

- `host` هو عنوان الخادم الذي نريد الاتصال به (مثل: `www.example.com`).
- `port` هو رقم المنفذ الذي يستمع إليه الخادم (عادةً 80 لـ HTTP و 443 لـ HTTPS).
- `reader` و `writer` هذان الكائنان يستخدمان لقراءة البيانات من الاتصال وكتابة البيانات إليه، على التوالي.

كتابة البيانات 2.

بعد فتح الاتصال، نحتاج إلى إرسال طلب إلى الخادم. هذا يتم عادةً باستخدام بروتوكول HTTP.

```
writer.write(request.encode())
```

- قبل إرساله `encode()` يجب أن يتم ترميزه إلى بايتات باستخدام HTTP. هو النص الذي يمثل طلب.
- تستخدم هذه الدالة لكتابة البيانات إلى الاتصال. من المهم ملاحظة أننا قد نحتاج إلى استخدام `writer.write`.
- بعد كتابة البيانات قد أرسلت بالفعل `await writer.drain` للتأكد من أن البيانات قد أرسلت بالفعل.

قراءة البيانات 3.

بعد إرسال الطلب، نحتاج إلى الانتظار واستلام البيانات من الخادم. يمكن القيام بذلك باستخدام دالة `reader.read`.

```
response = await reader.read()
```

- هذه الدالة تنتظر حتى يتم استلام البيانات. وعندما تتلقى البيانات، يتم تخزينها في المتغير `response`.
- لقراءة (`reader.readline()`) قد نحتاج إلى تحديد عدد البايتات التي نريد قرائتها، أو يمكن استخدام دوال مثل `reader.read` للبيانات حتى الوصول إلى نهاية السطر.

إغلاق الاتصال 4.

بعد الانتهاء من قراءة البيانات، من المهم إغلاق الاتصال لتحرير الموارد.

```
writer.close()
await writer.wait_closed()
```

- `writer.close()` تغلق هذا الاتصال، مما يعني أننا لن نستطيع إرسال المزيد من البيانات.
- `await writer.wait_closed()` هذه الدالة تضمن أن الاتصال قد تم إغلاقه بالكامل قبل أن ننتقل إلى أي عمليات أخرى. هذا مفيد لضمان عدم وجود تسرب للموارد.

مثال كامل

إليك مثال كامل لكيفية تنفيذ ذلك:

```
import asyncio

async def fetch_data(host, port):
    reader, writer = await asyncio.open_connection(host, port)

    # كتابة طلب HTTP
    request = 'GET / HTTP/1.1\r\nHost: {}\r\n\r\n'.format(host)
    writer.write(request.encode())
    await writer.drain()

    # قراءة الاستجابة
    response = await reader.read()

    # طباعة الاستجابة
    print(response.decode())

    # إغلاق الاتصال
    writer.close()
    await writer.wait_closed()

# تشغيل الدالة
asyncio.run(fetch_data('www.example.com', 80))
```

الخلاصة

التعامل مع اتصالات الشبكة بشكل غير محظوظ، مما يعزز أداء التطبيقات التي تعتمد على إدخال `asyncio` تتيح لنا مكتبة يمكننا، مثل تطبيقات الويب أو البرامج التي تتفاعل مع واجهات برمجة التطبيقات. ⑤ باستخدام على استغلال `asyncio` إجراء اتصالات متعددة بالتزامن، مما يحقق كفاءة أفضل أثناء انتظار استجابة الخادم. هذا يعكس قدرة `GIL` مما يجعلها أداة قوية لتطبيقات الشبكة، I/O لتحسين أداء عمليات.

الواحد Thread كيف تعمل التزامن في الـ

المتعددة كآلية لتحقيق التزامن في عمليات الإدخال والإخراج Threads في القسم السابق، قدمنا الـ

متعددة لتحقيق هذا النوع من التزامن. يمكننا القيام بكل ذلك ضمن حدود عملية واحدة Threads ومع ذلك، لا يحتاج إلى واحد Thread.

نقوم بذلك من خلال استغلال حقيقة أن عمليات الإدخال والإخراج يمكن أن تُنفذ بشكل متزامن على مستوى النظام، مما يعني أنه يمكن للنظام التعامل مع أكثر من عملية إدخال وإخراج في الوقت نفسه.

غير Sockets وبشكل خاص كيفية عمل الـ Sockets، لفهم ذلك بشكل أفضل، سنحتاج إلى الغوص في كيفية عمل الـ الحاصرة.

ما هي الـ Sockets؟

هي مفهوم أساسى لإرسال واستقبال البيانات عبر الشبكة. وهي الأساس لكيفية نقل البيانات إلى ومن **الـ Socket** الخادم.

تدعم عمليتين رئيسيتين **Socket** عمليات الـ 

- إرسال البيانات
- استقبال البيانات

والتي سيتم إرسالها بعد ذلك إلى عنوان بعيد، عادة ما يكون نوعاً من الخوادم. بمجرد إرسال، نكتب البيانات إلى الـ **Socket** الخاص بنا. بعد أن يتم إرسال هذه البيانات مرة أخرى **Socket** هذه البيانات، ننتظر من الخادم أن يكتب رده مرة أخرى إلى الـ **Socket**. يمكننا قراءة النتيجة، إلى الـ **Socket**.

 **فهم الـ Sockets:**

كصناديق بريد. يمكنك وضع رسالة في صندوق بريدك، ثم يلتقطها حامل الرسائل ويسلمها **Sockets** يمكنك التفكير في الـ **Socket** إلى صندوق بريد المستلم. يفتح المستلم صندوق بريدك ويقرأ رسالتك. اعتماداً على المحتوى، قد يرسل لك المستلم رسالة رد.

 في هذا التشبيه 

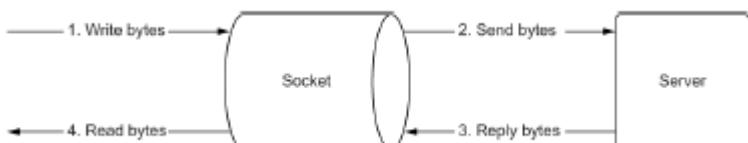
- الرسالة تمثل البيانات أو البيانات التي نريد إرسالها.
- وضع الرسالة في صندوق البريد هو كتابة البيانات إلى الـ **Socket**.
- فتح صندوق البريد لقراءة الرسالة هو قراءة البيانات من الـ **Socket**.

 **آلية النقل** 

يمكن اعتبار حامل الرسائل كآلية النقل عبر الإنترنت، التي تقوم بتوجيه البيانات إلى العنوان الصحيح.

 **مثال عملي**:

ثم **example.com** تتصفح بخادم **Socket** كما رأينا سابقاً، نقوم بفتح **example.com** في حالة الحصول على المحتويات من صفحة **HTML**، وننتظر من الخادم أن يرد بالنتيجة: في هذه الحالة **Socket** نكتب طلباً للحصول على المحتويات إلى تلك الـ **Socket**. يمكننا تصور تدفق البيانات من وإلى الخادم كما هو موضح في الشكل التالي.



Here's the translation with adjustments for clarity, avoiding the term "حصره":

وقراءة البيانات من الـ **Socket** كتابة البيانات إلى الـ **Socket**

بشكل افتراضي، ببساطة، هذا يعني أنه عندما ننتظر من خادم أن يرد ببيانات، نقوم **blocking** تكون **Socket** على **blocking** بإيقاف تطبيقنا حتى نحصل على بيانات للقراءة. وبالتالي، يتوقف تطبيقنا عن تشغيل أي مهام أخرى حتى نحصل على بيانات من الخادم، أو يحدث خطأ، أو يحدث انتهاء للوقت.

 **(non-blocking)**

في وضع **Sockets** على مستوى نظام التشغيل، لاحتاج إلى القيام بهذا الـ **blocking**.

كيف يعمل ذلك؟

1. كتابة البيانات ↳:

يمكننا ببساطة "إطلاق" عملية الكتابة دون الحاجة للانتظار حتى يتم إرسال Socket، عندما نكتب باليات إلى البيانات.

2. استمرار التطبيق 🚧:

بعد كتابة البيانات، يستمر تطبيقنا في العمل على مهام أخرى بدلاً من التوقف. هذا يعني أنه يمكننا تنفيذ عمليات إضافية أو التعامل مع مدخلات جديدة من المستخدم.

3. الإخطار بالبيانات 💡:

لاحقاً، عندما يتم استلام بيانات جديدة من الخادم، يخبرنا نظام التشغيل بذلك. هذا الإشعار يعني أنه يمكننا الآن معالجة البيانات الجديدة دون الحاجة إلى الانتظار.

4. زيادة التفاعلية 💬:

بهذه الطريقة، أصبح أكثر تفاعلية. بدلاً من الانتظار بشكل سلبي حتى نحصل على البيانات، نكون قادرين على الاستمرار في العمل والقيام بأشياء أخرى حتى يتم استلام البيانات.

☒ الفوائد:

- زيادة الكفاءة: نستفيد من الوقت بدلاً من الانتظار.

تحسين الأداء: يمكن أن يؤدي ذلك إلى تطبيقات أكثر استجابة، حيث لا تتوقف عن العمل لمجرد انتظار البيانات.

⚠️ أنظمة الإشعارات 📲:

في الخلفية، يتم تنفيذ هذا بواسطة بعض أنظمة إشعار الأحداث المختلفة، اعتماداً على نظام التشغيل الذي نستخدمه. لمجرد بما يكفي بحيث يتنتقل بين أنظمة الإشعار المختلفة، اعتماداً على النظام الذي يدعمه. الأنظمة المستخدمة للإشعار حسب نظام التشغيل هي:

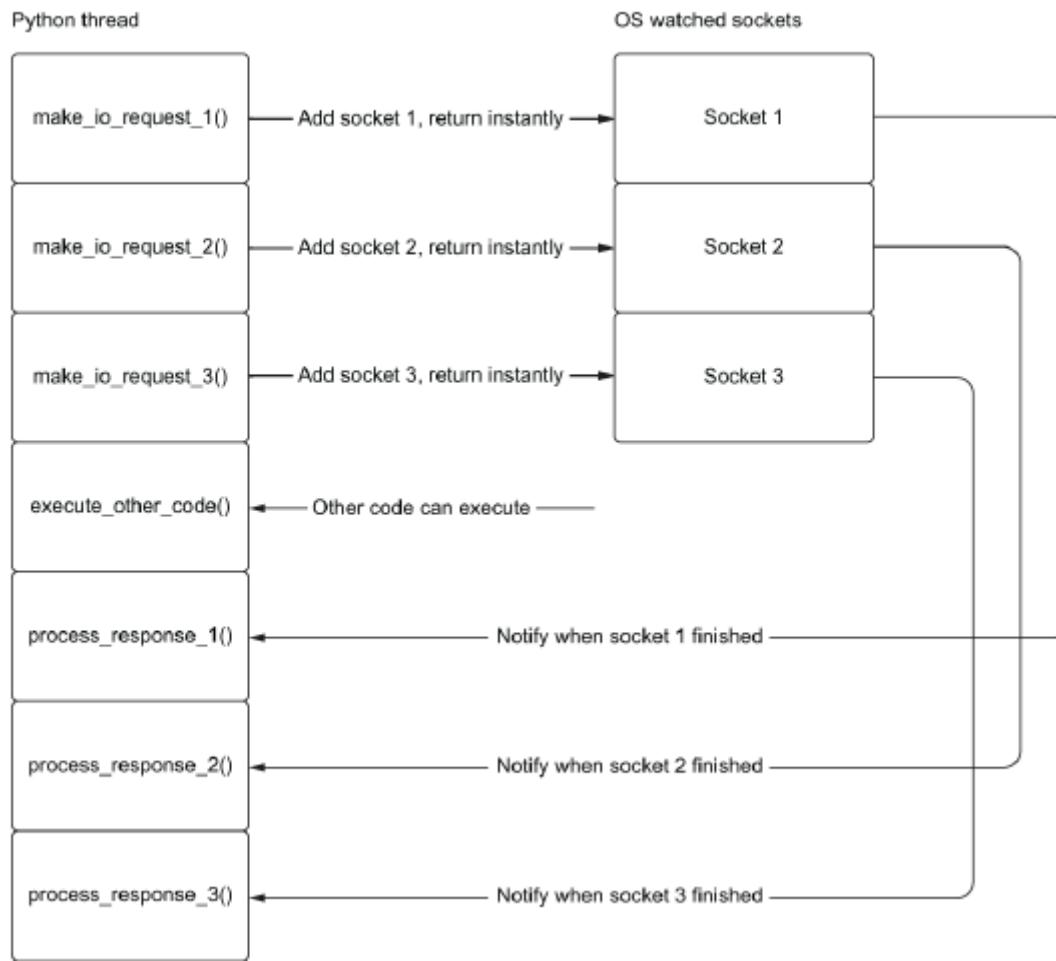
- kqueue** — FreeBSD و Mac OS
- epoll** — Linux
- IOCP (I/O completion port)** — Windows

لدينا وتخطرنا عندما تكون جاهزة ل التعامل معها. يعتبر هذا النظام الإشعاعي unblocking Sockets تحقيق التزامن **asyncio** هو الأساس الذي يمكن من خلاله لـ.

⌚ نموذج التزامن في asyncio:

واحد ينفذ بائيثون في أي وقت. عندما نقوم بعملية إدخال وإخراج، نقوم Thread لدينا فقط **asyncio** في نموذج التزامن لـ بائيثون Thread بتسلیمها إلى نظام الإشعار الخاص بنظام التشغيل الخاص بنا ليقوم بتعقبها لنا. بمجرد أن نقوم بذلك، تكون ليقوم نظام التشغيل unblocking Sockets لدينا حرّة في الاستمرار في تشغيل كود بائيثون آخر أو إضافة المزيد من المهمة التي كانت تنتظر النتيجة ثم تتبع تشغيل أي كود بائيثون "wake up," بتعقبها لنا. عندما تنتهي عملية الإدخال والإخراج آخر جاء بعد تلك العملية. يمكننا تصوّر هذا التدفق في الشكل المقابل مع عدد من العمليات المنفصلة التي تعتمد كل

منها على Socket.



❷ كيف يعمل الطلب

يعيد النتيجة على الفور ويطلب من نظام التشغيل مراقبة المأخذ (non-blocking I/O) إن إجراء طلب إدخال وإخراج (sockets) للحصول على البيانات (data).

❸ تنفيذ الشفرة الكود الآخر

أن تعمل على الفور بدلاً من الانتظار حتى تكتمل طلبات الإدخال والإخراج (execute_other_code()) هذا يسمح بـ

❹ التنبية عند الاتكمال

في وقت لاحق، يمكننا أن نتلقى تنبئها عند اكتمال الإدخال والإخراج ومعالجة الاستجابة.

❺ تتبع المهام

ولكن كيف تتبع المهام التي تنتظر الإدخال والإخراج مقارنة بتلك التي يمكن أن تعمل بشكل طبيعي لأنها عبارة عن شفرة (event loop).
بايثون عادي؟ الجواب يمكن في بنية تدعى حلقة الأحداث.

❻ كيف تعمل حلقة الأحداث

تعتبر حلقات الأحداث نمط تصميم شائع في العديد من الأنظمة وقد **asyncio** حلقة الأحداث هي قلب كل تطبيق يستخدم في متصل لإجراء طلب ويب غير متزامن، فقد أنشأت مهمة على **JavaScript** وجدت لفترة طويلة. إذا كنت قد استخدمنا حلقة أحداث.

تطبيقات واجهة المستخدم

ما يُسمى بحلقات الرسائل خلف الكواليس آلية أساسية للتعامل مع الأحداث مثل إدخال **Windows GUI** تستخدم تطبيقات لوحة المفاتيح، مع السماح أيضًا بظهور واجهة المستخدم.

الحلقة الأساسية

الحلقة الأساسية بسيطة جدًا. نقوم بإنشاء قائمة انتظار تحتوي على قائمة من الأحداث أو الرسائل. ثم نكرر إلى الأبد، مع معالجة الرسائل واحدة تلو الأخرى كما تأتي إلى قائمة الانتظار. في بايثون، قد تبدو حلقة الأحداث الأساسية كما يلي:

```
from collections import deque

messages = deque()
while True:
    if messages:
        message = messages.pop()
        process_message(message)
```

حلقة الأحداث في asyncio

يمكن استخدام مصطلحات أكثر دقة لشرح المفهوم. إليك نص معدل

تحتفظ حلقة الأحداث بقائمة من المهام بدلاً من الرسائل. المهام هي وحدات تنفيذية تتعلق بـ **asyncio**. عن التنفيذ عندما تصل إلى عملية مرتبطة بالإدخال والإخراج، مما يسمح لحلقة الأحداث بتشغيل **coroutine** يمكن أن تتوقف مهام أخرى لا تنتظر إكمال عمليات الإدخال والإخراج.

إنشاء حلقة الأحداث

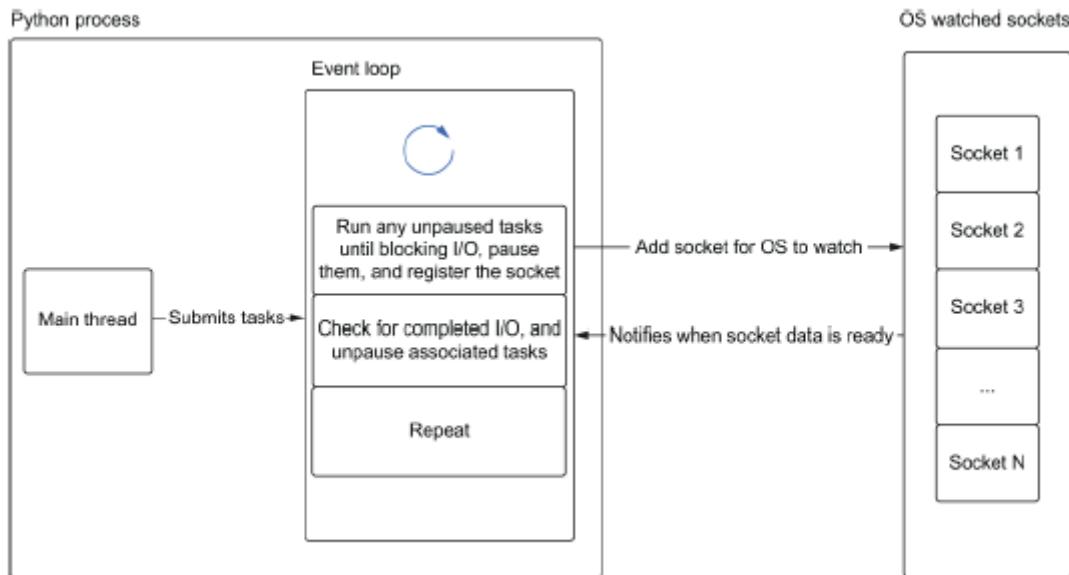
عند إنشاء حلقة الأحداث، نقوم بإنشاء قائمة انتظار فارغة من المهام. يمكننا بعد ذلك إضافة المهام إلى قائمة الانتظار ليتم تشغيلها. كل تكرار في حلقة الأحداث يتحقق من المهام التي تحتاج إلى التنفيذ وسيقوم بتشغيلها واحدة تلو الأخرى حتى تصل المهمة إلى عملية إدخال وإخراج. في ذلك الوقت، ستتوقف المهمة، وسنطلب من نظام التشغيل مراقبة أي مأخذ لإكمال الإدخال والإخراج. ثم نبحث عن المهمة التالية التي سيتم تشغيلها.

مراقبة الإدخال والإخراج

في كل تكرار من حلقة الأحداث، ستحقق مما إذا كانت أي من عمليات الإدخال والإخراج قد اكتملت؛ إذا اكتملت، سنقوم بـ "إيقاظ" أي مهام كانت متوقفة ونعطيها الفرصة لإنها التنفيذ.

التصور

يمكننا تصور ذلك كما هو موضح في الشكل المقابل: الخيط الرئيسي يرسل المهام إلى حلقة الأحداث، التي يمكنها بعد ذلك تشغيلها.



الشكل السابق: مثال عن كيفية إرسال المهام إلى حلقة الأحداث

دعنا نتخيل أن لدينا ثلاثة مهام تقوم كل منها بعمل طلب ويب غير متزامن. تتكون هذه المهام، **asyncio** لتوضيح كيفية عمل من خطوات كالتالي:

1. جزء من الشيفرة يقوم بعمليات معالجة ثقيلة (**CPU-bound**): إعدادات مرتبطة بالمعالجة المركزية.
2. طلب ويب غير متزامن: يقوم بإجراء طلب بيانات من الشبكة.
3. جزء آخر من الشيفرة يعالج البيانات المستلمة (**CPU-bound**) معالجة ما بعد الطلب.

كود التخليمة

يمكننا كتابة الشيفرة التخليمة بالطريقة التالية:

```
def make_request():
    cpu_bound_setup() # CPU-bound
    io_bound_web_request() # Non-blocking web request
    cpu_bound_postprocess() # CPU-bound

task_one = make_request()
task_two = make_request()
task_three = make_request()
```

سير العمل

1. بدء المهام:

- جميع المهام الثلاثة تبدأ في العمل على الإعدادات المرتبطة بالمعالجة المركزية
- بينما تُترك code فإن المهمة الأولى فقط هي التي تبدأ في تنفيذ thread نظرًا لأننا نعمل في وضع أحدادي
- المهام الأخرى في انتظار التشغيل

2. التوقف في انتظار الإدخال والإخراج:

- بعد الانتهاء من إعدادات المعالجة، تصل المهمة 1 إلى عملية إدخال وإخراج

"تتوقف المهمة 1 وتخبر النظام: "أنا في انتظار إدخال وإخراج؛ يمكن لأي مهام أخرى أن تعمل الآن" °"

الانتظار المتزامن

- عندما توقف المهمة 1، يمكن للمهمة 2 أن تبدأ في التنفيذ.
- تبدأ المهمة 2 في العمل على إعداداتها، ثم تتوقف في انتظار الإدخال والإخراج.
- في هذه المرحلة، كل من المهمة 1 والمهمة 2 تنتظران بشكل متزامن لإكمال طلباتهما.

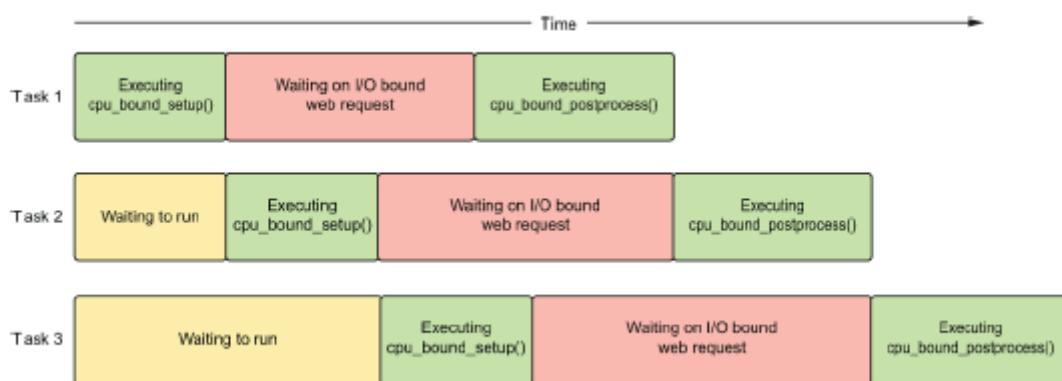
بدء مهمة جديدة

- بما أن المهام 1 و2 في وضع التوقف، يمكننا الآن تشغيل المهمة 3.
- تستمر المهمة 3 حتى تصل إلى مرحلة إدخال وإخراج، ثم تتوقف أيضًا.

استئناف التنفيذ

- بمجرد أن تنتهي المهمة 1 من انتظار إدخالها وإخراجها، يُنهى نظام التشغيل أن العملية قد اكتملت.
- يمكننا الآن استئناف تنفيذ المهمة 1 بينما تسمرة المهام 2 و3 في الانتظار.

الشكل المقابل



- يمكننا رؤية تدفق تنفيذ code
- يظهر أن هناك دائمًا عمل واحد فقط مرتبط بالمعالجة المركزية قيد التشغيل، بينما يمكن أن يكون هناك عمليتان مرتبطتان بالإدخال والإخراج تعملان في نفس الوقت.
- لوفورات كبيرة في الوقت **asyncio** هذه القدرة على الانتظار المتزامن هي السبب وراء تحقيق