PYTHON ACADEMY

APRENDA SOBRE CONCORRÊNCIA E PARALELISMO EM PYTHON

Guia completo de Concorrência vs Paralelismo em Python: diferenças, GIL, threading vs multiprocessing vs async, quando usar cada abordagem (I/O-bound vs CPU-bound), exemplos práticos.

PYTHONACADEMY.COM.BR

Crie ebooks profissionais incríveis em minutos com IA

Chega de formatar texto no Google Docs, Word ou ferramentas que só te fazem perder tempo...

E deixe que nossa IA faça o trabalho pesado!

- Capas gerados por IA
- Infográficos feitos por IA
- Edite em Markdown em Tempo Real
- Adicione Banners Promocionais

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS

> ✅ ***Atualizado para Python 3.13*** *(Dezembro 2025) Conteúdo enriquecido com tabela de decisão rápida (threading vs multiprocessing vs async) e quando usar cada abordagem.*

Salve salve Pythonista 👋

Concorrência e **Paralelismo** são técnicas essenciais para melhorar performance, mas são **conceitos diferentes**! Concorrência é sobre **lidar com múltiplas tarefas** (switching), enquanto Paralelismo é sobre **executar simultaneamente** (múltiplos cores).

Neste guia, você vai aprender:

- ✅ **Concorrência vs Paralelismo** - Diferenças fundamentais
- ✅ **GIL (Global Interpreter Lock)** - Limitações do Python
- ✅ **Threading vs Multiprocessing vs Async** - Quando usar
- ✅ **Tabela de decisão** - I/O-bound vs CPU-bound

Concorrência vs Paralelismo: Afinal, o que é isso?

Tanto concorrência quanto paralelismo são técnicas usadas para fazer com que um programa execute tarefas de maneira mais eficiente.

Embora os termos sejam relacionados, eles representam abordagens distintas.

Concorrência descreve uma situação onde duas ou mais tarefas parecem progredir ao mesmo tempo, dentro de um mesmo período.

Imagine que você tem várias tarefas em andamento, e você consegue começar a trabalhar em uma antes de terminar a outra.

Em termos mais técnicos, dizemos que tarefas concorrentes têm seus períodos de execução sobrepostos.

É importante notar que, concorrência não implica necessariamente em execução simultânea; as tarefas podem ser executadas de forma intercalada, como se estivessem revezando o uso de um recurso.

Para ilustrar o conceito de concorrência, vamos analisar o seguinte exemplo:

```
from time import time, sleep

def fazer_bolo(tipo):
    print(f'Fazendo bolo {tipo}...')
    sleep(1)
    print(f'Bolo {tipo} feito!')

inicio = time()
fazer_bolo('chocolate')
fazer_bolo('cenoura')
print(f'Tempo total: {time() - inicio}')
```

A saída será:

```
Saída
Fazendo bolo chocolate...
Bolo chocolate feito!
Fazendo bolo cenoura...
Bolo cenoura feito!
Tempo total: 2.0023531913757324
```

Neste exemplo, executamos as funções `fazer_bolo('chocolate')` e `fazer_bolo('cenoura')` de forma sequencial. Isso significa que:

- Primeiro, a função `fazer_bolo('chocolate')` é executada completamente: ela imprime "Fazendo bolo chocolate...", espera 1 segundo, e então imprime "Bolo chocolate feito!".
- Somente após a conclusão da primeira função, a função `fazer_bolo('cenoura')` começa a ser executada, seguindo o mesmo processo.

Por que este exemplo demonstra Concorrência (em um sentido mais amplo):

Embora o código seja executado de forma sequencial, ele ilustra a ideia de concorrência no tempo.

As tarefas de "fazer o bolo de chocolate" e "fazer o bolo de cenoura" acontecem dentro de um período de tempo que se inicia quando começamos a fazer o primeiro bolo e termina quando finalizamos o segundo.

Nesse intervalo, ambos os "trabalhos" de fazer bolos estão a decorrer, mesmo que um seja iniciado somente após o outro terminar.

É crucial entender: Este exemplo, por ser sequencial, não demonstra a concorrência no sentido técnico de execução intercalada ou paralela que vemos em sistemas operacionais ou com técnicas como `threading` e asyncio.

Ele serve para introduzir a noção de que múltiplas tarefas podem estar em andamento dentro de um mesmo período global de tempo, mesmo que sejam executadas uma após a outra.

Em contextos de programação concorrente mais avançados em Python, você verá a concorrência implementada de maneiras que permitem uma progressão mais simultânea das tarefas, especialmente em operações de entrada/saída (I/O), utilizando técnicas com a biblioteca `threading` ou `asyncio`.

No entanto, este exemplo mais simples ajuda a construir uma base para entender a diferença fundamental entre concorrência e paralelismo, que será explorada a seguir.

Já o **paralelismo**, por outro lado, é quando duas ou mais tarefas são executadas ao mesmo tempo.

```
from multiprocessing import Process
from time import time, sleep

def fazer_bolo(tipo):
    print(f'Fazendo bolo {tipo}...')
    sleep(1)
    print(f'Bolo {tipo} feito!')

inicio = time()
p1 = Process(target=fazer_bolo, args=('chocolate',))
p2 = Process(target=fazer_bolo, args=('cenoura',))

p1.start()
p2.start()

p1.join()
p2.join()
print(f'Tempo total: {time() - inicio}')
```

E a saída:

```
Saída
Fazendo bolo cenoura...
Fazendo bolo chocolate...
Bolo chocolate feito!
Bolo cenoura feito!
Tempo total: 1.0561878681182861
```

Como pode ver, utilizamos duas bibliotecas diferentes para mostrar exemplos de concorrência (usa método sequencial) e paralelismo (usa `multiprocessing.Process` para atingir a execução simultânea).

Os benefícios de entender e aplicar esses dois conceitos passam por desempenho, escalabilidade e construção de programas mais responsivos e eficientes.

Contudo, existem desafios no seu uso, principalmente relacionados ao manuseio de recursos compartilhados e questões de segurança.

Threading, Multiprocessing e Asyncio

Threading permite a execução simultânea de várias threads dentro de um único processo, sendo ideal para operações I/O-bound (operações de entrada e saída, como leituras/gravações em disco, acesso a banco de dados ou acesso à uma API externa são exemplos), mas limitado pelo Global Interpreter Lock (famoso **GIL**, explicação aqui embaixo).

Multiprocessing cria múltiplos processos independentes com seu próprio espaço de memória, perfeito para operações CPU-bound (operações com grande uso de CPU, como cálculos matemáticos, multiplicação de matrizes e etc), pois cada processo pode ser executado em um núcleo diferente do processador, evitando problemas de concorrência.

Asyncio oferece concorrência usando a sintaxe `async` / `await` em uma única thread, ideal para operações I/O-bound que envolvem muita espera, mas não adequado para tarefas que exigem paralelismo verdadeiro em operações CPU-bound.

> ***Obs.**: O **GIL (Global Interpreter Lock)** é um mecanismo do Python que garante que apenas uma thread execute o código Python por vez, mesmo que o programa tenha múltiplas threads. Imagine o GIL como uma chave que apenas uma thread pode segurar por vez para acessar o interpretador Python. Isso evita que duas threads modifiquem dados ao mesmo tempo e causem problemas. No entanto, isso também significa que, mesmo em um computador com múltiplos núcleos, apenas uma thread pode executar código Python por vez, o que pode limitar a performance em programas que fazem uso intenso de threads.*

Threading, Multiprocessing e Asyncio em Python

Agora veremos esses conceitos aplicados na programação Python!

Threading

Threading é uma forma de paralelismo que permite que várias operações ocorram simultaneamente dentro de um único processo.

Em Python, o módulo `threading` fornece uma API de alto nível para criar e gerenciar threads.

Exemplo de uso do `threading`:

```python
import threading
import time

def worker():
    print("Thread iniciada")
    time.sleep(2)
    print("Thread terminada")

# Criar threads
threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()

# Aguardar todas as threads terminarem
for t in threads:
    t.join()

print("Todas as threads foram concluídas")
```

No código acima:

- Criamos a unidade de trabalho, que no nosso caso será a função `worker`
- Em seguida, criamos as threads com `threading.Thread(target=worker)` e as iniciamos com `t.start()`
- Por fim, aguardamos a finalização de todas com `t.join()`

Multiprocessing

Multiprocessing envolve a criação de novos processos no Sistema Operacional, cada um com seu próprio espaço de memória.

Isso evita problemas de concorrência associados à programação multithreaded, especialmente em sistemas com CPU múltiplos.

Essa é uma forma de contornar os limites impostos pelo GIL, já que ele apenas consegue restringir o Interpretador Python em um único processo, e vários serão criados.

Exemplo de uso do `multiprocessing`:

```
import multiprocessing
import time

def worker():
    print("Processo iniciado")
    time.sleep(2)
    print("Processo terminado")

# Criar processos
processes = []
for i in range(5):
    p = multiprocessing.Process(target=worker)
    processes.append(p)
    p.start()

# Aguardar todos os processos terminarem
for p in processes:
    p.join()

print("Todos os processos foram concluídos")
```

O código acima é similar ao código que fizemos no exemplo de uso de `threading`.

A diferença está no container que irá executar nosso código: no primeiro, serão múltiplas Threads. No segundo, múltiplos processos serão executados.

Asyncio

Asyncio é uma biblioteca para escrita de código concorrente usando a sintaxe `async` / `await` introduzida no Python 3.5.

É especialmente útil para operações de I/O que podem ser aguardadas, como chamadas de rede.

Exemplo de uso do `asyncio`:

```
import asyncio

async def worker():
    print("Tarefa iniciada")
    await asyncio.sleep(2)
    print("Tarefa terminada")

async def main():
    # Criar tarefas
    tasks = [worker() for _ in range(5)]

    # Aguardar todas as tarefas terminarem
    await asyncio.gather(*tasks)

# Executar o loop de eventos
asyncio.run(main())

print("Todas as tarefas foram concluídas")
```

Nesse código:

- `async def worker()`: Define uma função assíncrona chamada `worker`.
- `tasks = [worker() for _ in range(5)]`: Cria uma lista de 5 tarefas worker.
- `await asyncio.gather(*tasks)`: Aguarda todas as tarefas na lista `tasks` serem concluídas. O `asyncio.gather` executa todas as tarefas de forma concorrente.
- `asyncio.run(main())`: Executa a função `main` dentro do loop de eventos `asyncio`. Isto inicializa a execução assíncrona.

> 💡 *Estou desenvolvendo o **Ebookr.ai**, uma plataforma que transforma suas ideias em ebooks profissionais usando IA — com geração de capa, infográficos e exportação em PDF. Te convido a conhecer!*

Comparação entre Threading, Multiprocessing e Asyncio

Threading:

- Usa múltiplas threads dentro do mesmo processo.
- Melhor para operações I/O-bound devido ao Global Interpreter Lock (GIL).

- Menor overhead de criação e destruição de threads comparado a processos.

Multiprocessing:

- Cria processos separados com seu próprio espaço de memória.
- Ideal para operações CPU-bound porque cada processo pode ser executado em um core diferente.
- Evita problemas de concorrência, como race conditions, presentes no threading.

Asyncio:

- Usa um único thread, mas permite executar múltiplas tarefas de forma assíncrona.
- Excelente para operações I/O-bound onde muitas operações de espera são necessárias.
- Não é adequado para operações CPU-bound que necessitam de paralelismo verdadeiro.

Cada uma dessas abordagens tem seu próprio caso de uso ideal, e a escolha entre elas depende da natureza da tarefa a ser executada.

Desempenho e Escalabilidade

Em Python, o principal fator limitante para concorrência e paralelismo é o Global Interpreter Lock (GIL).

Como explicamos anteriormente, o GIL é um mecanismo que sincroniza a execução de threads para evitar conflitos de estado.

Se você começar a experimentar problemas de desempenho com threads e lock contention, pode ser a hora de começar a ver alternativas, tais como processos ou greenlets.

Tabela de Decisão Rápida

Qual Técnica Usar?

Cenário	Abordagem	Por quê
Requisições HTTP múltiplas	`asyncio`	I/O-bound, alta escalabilidade
Processar arquivos grandes	`multiprocessing`	CPU-bound, paralelismo real
Download de arquivos	`threading`	I/O-bound, simples
GUI responsiva	`threading`	Não bloquear interface
Processamento de imagens	`multiprocessing`	CPU-bound, contornar GIL
Web scraping	`asyncio` ou `threading`	I/O-bound, depende da lib
Machine Learning training	`multiprocessing`	CPU-bound, usar todos cores
Chat server	`asyncio`	I/O-bound, milhares de conexões
Cálculos matemáticos	`multiprocessing`	CPU-bound, paralelismo
Ler/escrever DB	`asyncio`	I/O-bound, concorrência

Regras Simples:

✅ **I/O-bound** (rede, disco, DB) → Use **async** (melhor) ou **threading** ✅ **CPU-bound** (cálculos) → Use **multiprocessing** ✅ **Mixed workload** → Use **ProcessPoolExecutor** + **ThreadPoolExecutor**

> 💡 ***Dica:*** *Começe sempre com código* ***síncrono*** *(normal). Só adicione concorrência/paralelismo quando tiver* ***gargalo de performance medido****!*

Conclusão

Neste guia completo sobre **Concorrência e Paralelismo**, você aprendeu:

✅ **Diferença fundamental** - Concorrência vs Paralelismo ✅ **GIL** - Limitação do Python para threads ✅ **Threading vs Multiprocessing vs Async** - Quando usar ✅ **Tabela de decisão** - I/O-bound vs CPU-bound ✅ **Boas práticas** - Começar simples, medir performance

Principais lições:

- **Concorrência** = lidar com várias tarefas (switching)
- **Paralelismo** = executar simultaneamente (múltiplos cores)
- **I/O-bound** → async ou threading
- **CPU-bound** → multiprocessing (contornar GIL)
- Sempre **medir antes de otimizar**

Próximos passos:

- Pratique com `concurrent.futures` (mais simples que threads diretas)

- Explore `asyncio` para I/O intensivo
- Estude `ProcessPoolExecutor` para CPU-bound
- Aprenda sobre `Queue` para comunicação entre processos

Concorrência e Paralelismo são técnicas poderosas que, quando usadas corretamente, podem aumentar significativamente a eficiência do seu código Python. Comece com problemas pequenos e aumente gradualmente a complexidade.

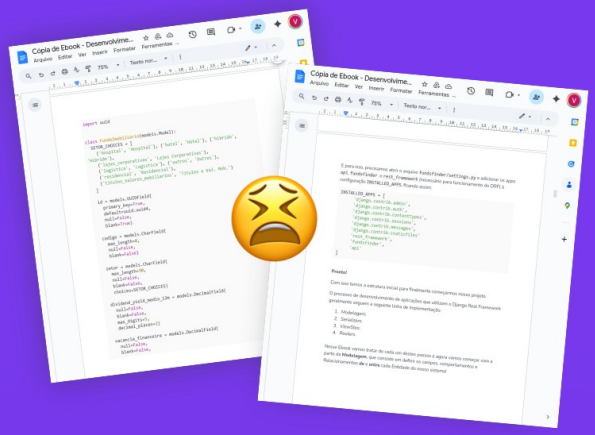
E lembre-se, a prática leva à perfeição!

Bora codar! 👊

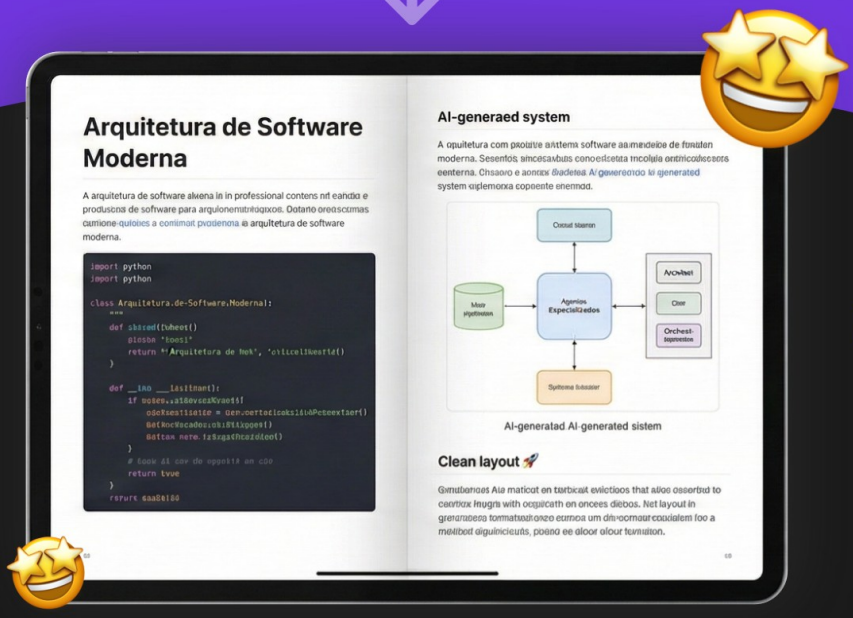
Não se esqueça de conferir!

Ebookr

Crie Ebooks profissionais em minutos com IA



Chega de formatar código no Google Docs ou Word



Capas gerados por IA

Infográficos feitos por IA

Adicione Banners Promocionais

Deixe que nossa IA faça o trabalho pesado

Edite em Markdown em Tempo Real

TESTE AGORA

PRIMEIRO CAPÍTULO 100% GRÁTIS