



PYTHON
ACADEMY

PROGRAMAÇÃO ASSÍNCRONA COM PYTHON

Guia completo de Programação Assíncrona em Python: async/await, asyncio, quando usar vs threads vs sync, casos de uso (APIs, I/O), diferenças entre concorrência e paralelismo.

PYTHONACADEMY.COM.BR

Este ebook foi gerado por




Crie Ebooks técnicos incríveis em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**




Chega de formatar código no Google Docs

Deixe que nossa IA faça o trabalho pesado

 Syntax Highlight

 Adicione Banners Promocionais

 Edite em Markdown em Tempo Real

 Infográficos feitos por IA

TESTE AGORA 

 PRIMEIRO CAPÍTULO 100% GRÁTIS

✓ **Atualizado para Python 3.13** (Dezembro 2025)

Conteúdo enriquecido com comparação `async` vs `sync` vs `threading` e quando usar cada abordagem.

Salve salve Pythonista!

Programação Assíncrona permite que seu código execute múltiplas tarefas de I/O **concorrentemente** sem criar threads. É ideal para operações I/O-bound como **requisições HTTP, banco de dados e arquivos**.

Neste guia completo, você vai aprender: - ✓ O que é programação assíncrona - ✓ **`async/await`** e coroutines - ✓ **`asyncio`** - biblioteca padrão - ✓ **Async vs Sync vs Threading** - quando usar cada um - ✓ Casos de uso práticos (APIs, I/O)

Prepare-se para muitos exemplos práticos e um mini-projeto utilizando programação assíncrona!

Introdução à Programação Assíncrona

Vamos começar pelo princípio: o que é programação assíncrona?

Em duas palavras, programação assíncrona é sobre **não esperar**.

Normalmente, quando o nosso código necessita de algum recurso, seja um arquivo no disco, um dado vindo da rede, ou simplesmente um cálculo longo, ele simplesmente para e espera esse recurso ficar disponível.

Nesse meio tempo, o processador fica lá, sem fazer nada.

Agora imagine se em vez de esperar, ele pudesse executar outra parte do código que não dependesse desse recurso?

É exatamente isso que a programação assíncrona faz.

Em vez de bloquear a execução esperando algum recurso, o código é capaz de “pular” essa parte e continuar a execução em outro lugar.

Quando o recurso finalmente fica disponível, o código pode retomar de onde parou.

Agora vamos ao “Hello World” da programação assíncrona com Python:

```
import asyncio

async def main():
    print('Olá ...')
    await asyncio.sleep(5)
    print('... Mundo')

# Python 3.7+
asyncio.run(main())
```

E como aqui na Python Academy nós gostamos de te entregar tudo mastigado, vamos às explicações:

- `async def main()`: Esta é a definição de uma função assíncrona. A palavra-chave `async` indica que a função é assíncrona e pode usar `await` para aguardar outras operações assíncronas.
- `await asyncio.sleep(5)`: Esta linha faz a função `main` aguardar assincronamente por 5 segundos. Durante este tempo, outras partes do programa podem continuar executando. `asyncio.sleep(5)` é uma função assíncrona que simula uma operação de bloqueio (como uma operação de I/O), mas sem realmente bloquear.

- `asyncio.run(main())` : Esta é a maneira de iniciar e gerenciar a execução da função assíncrona `main()`. `asyncio.run` é uma função que executa a *coroutine* `main()` e não retorna até que a `main()` esteja completa.

Como o Processador e o Sistema Operacional Tratam a Programação Assíncrona

A verdade é que, internamente, o processador e o sistema operacional não estão fazendo nada muito diferente quando se trata de programação assíncrona.

O que muda é a forma como o programa é estruturado.

O programa consegue organizar-se de forma que o processador sempre tem algum código para executar, nunca ficando parado.

A **biblioteca `asyncio`** que vimos no exemplo anterior é um exemplo de um tipo de sistema chamado `event loop`.

A função `asyncio.run()` que usamos para executar nosso código assíncrono na verdade está iniciando um event loop, e o `await asyncio.sleep(5)` está suspendendo a execução da função e retornando o controle à este event loop.

Programando Assincronamente em Python

Trabalhar com código assíncrono em Python é bastante direto, graças à introdução das palavras-chave `async` e `await` a partir do Python 3.5.

Basicamente, uma função é declarada como assíncrona usando a palavra-chave `async def` no lugar de `def` :

```
async def minha_funcao():  
    print("Olá, Mundo Assíncrono!")
```

Agora, essa função é uma **coroutine**, e pode ser pausada e retomada.

A pausa é feita através da palavra-chave `await` , seguida de uma expressão que retorna uma coroutine:

```
async def minha_funcao():  
    print("Iniciando...")  
    await asyncio.sleep(1)  
    print("...terminou!")
```

Nesse exemplo, a função `asyncio.sleep(1)` é uma coroutine que simula um delay de 1 segundo.

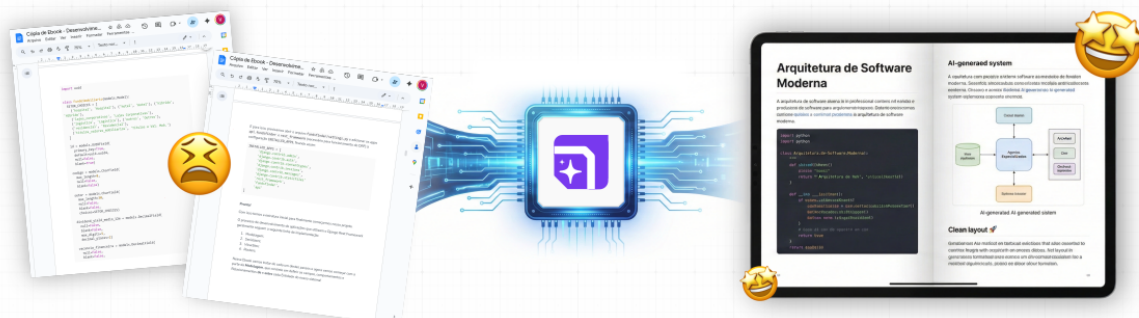
Mas em vez de bloquear a execução, ela apenas indica ao event loop que a execução atual deve ser suspensa, e que deve ser retomada depois do tempo especificado.



*Estou desenvolvendo o **DevBook**, uma plataforma que usa IA para gerar ebooks técnicos profissionais. Depois de ler, dá uma passada no site!*

Crie Ebooks técnicos incríveis em minutos com IA

Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs

Deixe que nossa IA faça o trabalho pesado

Syntax Highlight

Adicione Banners Promocionais

Edite em Markdown em Tempo Real

Infográficos feitos por IA

TESTE AGORA! PRIMEIRO CAPÍTULO 100% GRÁTIS

Programação Assíncrona com aiohttp

Python já vem com várias bibliotecas prontas para auxiliar na programação assíncrona, sendo a mais conhecida delas a **asyncio**, que já mencionamos antes.

Além dela, podemos fazer a instalação da `aiohttp` para fazer requisições HTTP assíncronas com `pip install aiohttp`.

Vamos ver um exemplo de como utilizá-la para fazer requisições assíncronas ao site Oficial do Python:

```
import aiohttp
import asyncio

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

asyncio.run(main())
```

Nesse exemplo, a função `fetch()` está fazendo uma requisição HTTP usando a biblioteca `aiohttp`.

Ela é uma função assíncrona, e pode ser pausada e retomada.

Note que estamos usando a palavra `await` tanto para esperar a resposta HTTP como para ler o texto da resposta.

Ao final da execução, você verá o código HTML da página na saída do programa!

Exemplos de Programação Assíncrona em Python

Vamos mostrar agora alguns exemplos de utilização da programação assíncrona em Python.

Exemplo 1: Função Assíncrona simples

```
import asyncio

async def contar_ate_dez():
    for i in range(10):
        print(i)
        await asyncio.sleep(1)
    print("Contei até dez!")

asyncio.run(contar_ate_dez())
```

A função `contar_ate_dez()` é uma função assíncrona que imprime os números de 0 a 9, com um intervalo de 1 segundo entre eles.

Ela utiliza o comando `await asyncio.sleep(1)` para esperar assincronamente por 1 segundo.

Colocando-o em execução, você verá a seguinte saída:

```
0
1
2
3
4
5
6
7
8
9
Contei até dez!
```

Exemplo 2: Executando múltiplas coroutines

Vamos ver nesse exemplo como executar múltiplas funções assíncronas:

```
import asyncio

async def contar_ate_dez(nome):
    for i in range(10):
        print(f'{nome}: {i}')

        # Simula uma operação demorada...
        await asyncio.sleep(1)

async def main():
    await asyncio.gather(
        contar_ate_dez('Tarefa 1'),
        contar_ate_dez('Tarefa 2')
    )

asyncio.run(main())
```

Esse exemplo mostra como executar várias *coroutines* em paralelo.

A função `asyncio.gather()` é usada para combinar várias *coroutines* em uma única tarefa.

Elas serão executadas assincronamente, e a função `await asyncio.gather()` só vai completar quando todas elas tiverem completado.

Mini Projeto com Programação Assíncrona

Para terminar, vamos fazer um mini-projeto utilizando programação assíncrona.

Vamos fazer um *Crawler* simples que vai baixar várias páginas web em paralelo.

Ele utilizará o `aiohttp`, portanto não esqueça de fazer sua instalação com `pip install aiohttp`:

```

import asyncio
import aiohttp

lista_urls = [
    'http://python.org',
    'http://google.com',
    'http://facebook.com',
    # Adicione quantas URLs quiser aqui...
]

async def fetch_page(session, url):
    async with session.get(url) as response:
        return await response.text()

async def fetch_all_pages():
    async with aiohttp.ClientSession() as session:
        tasks = []
        for url in lista_urls:
            tasks.append(fetch_page(session, url))
        pages = await asyncio.gather(*tasks)
        return pages

async def main():
    pages = await fetch_all_pages()
    for i, page in enumerate(pages):
        print(f'PÁGINA {i}: \n {page[:150]}...\n\n{"*" * 30}')

asyncio.run(main())

```

Nesse código, `fetch_page()` é uma coroutine que baixa uma página web usando a biblioteca `aiohttp`.

A função `fetch_all_pages()` usa a função `asyncio.gather()` para baixar todas as páginas em paralelo, e retorna uma lista com o conteúdo de todas elas.

Ao final da sua execução, será mostrado os 150 primeiros caracteres de cada página:

PÁGINA 0:

```
<!doctype html>
<!--[if lt IE 7]>    <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">    <![endif]-->
<!--[if IE 7]>        <html class="no-js ie7 lt-ie8 lt-...
```

PÁGINA 1:

```
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
lang="pt-BR"><head><meta content="text/html; charset=UTF-8" http-
equiv="Content...
```

PÁGINA 2:





```
<!DOCTYPE html>
<html lang="pt" id="facebook" class="no_js">
<head><meta charset="utf-8" /><meta name="referrer" content="default"
id="meta_referrer" ...
```

Em posse desses dados, você pode fazer qualquer tipo de processamento!

Você pode acessar outras páginas, como páginas de notícias, e-commerces para verificação de preços entre outras várias possibilidades!





Async vs Sync vs Threading: Quando Usar?

Programação Síncrona (Normal)

Use quando: -  Tarefas **CPU-bound** (cálculos intensivos) -  Código **simples** e direto -  **Sem I/O** ou I/O mínimo -  Projeto **pequeno**

```
# Síncrono: uma tarefa por vez
def processar_dados(data):
    resultado = calcular(data)  # Bloqueia aqui
    return resultado
```




Programação Assíncrona (async/await)


Use quando: -  Tarefas **I/O-bound** (rede, disco, DB) -  Múltiplas **requisições HTTP** simultaneamente -  **Single-threaded** (sem problemas de concorrência) -  Quer **escalabilidade** (milhares de conexões)

```
# Assíncrono: não bloqueia durante I/O
import asyncio

async def fetch_data(url):
    # Enquanto espera resposta, executa outras tarefas
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return await response.text()
```

Threading/Multiprocessing

Use Threading quando: -  Tarefas **I/O-bound** sem suporte assíncrono -  Bibliotecas **síncronas** (sem async) -  Paralelismo **simples** (poucas threads)

Use Multiprocessing quando: -  Tarefas **CPU-bound** (cálculos intensivos) -  Precisa **verdadeiro paralelismo** (contornar GIL) -  Processamento de **dados massivos**

Comparação Rápida

| Abordagem | I/O-bound | CPU-bound | Escalabilidade | Complexidade |
|-----------------|-------------|--------------|----------------|--------------|
| Síncrono | ❌ Lento | ✅ OK | ❌ Baixa | ✅ Simples |
| Async | ✅ Rápido | ❌ Ruim | ✅ Alta | ⚠ Média |
| Threading | ✅ Bom | ❌ Ruim (GIL) | ⚠ Média | ⚠ Média |
| Multiprocessing | ❌ Over-head | ✅ Excelente | ⚠ Média | ❌ Complexo |

Casos de Uso Práticos

Async: Web scraping, APIs REST, WebSockets, chat servers

Threading: Download de arquivos, GUI responsivas

Multiprocessing: Processamento de imagens, ML training, análise de dados

Síncrono: Scripts simples, cálculos, ETL básico

💡 **Regra de Ouro:** Se é *I/O-bound* (rede, disco), use **async**. Se é **CPU-bound** (cálculos), use **multiprocessing**.

Conclusão

Neste guia completo sobre **Programação Assíncrona**, você aprendeu:

- ✓ **Conceito** - Concorrência sem threads
- ✓ **async/await** - Palavras-chave para coroutines
- ✓ **asyncio** - Biblioteca padrão para async
- ✓ **Async vs Sync vs Threading** - Quando usar cada um
- ✓ **Casos de uso** - APIs, I/O, web scraping

Principais lições: - Async é ideal para **tarefas I/O-bound** (rede, disco) - **NÃO** use async para **CPU-bound** (use multiprocessing) - Async oferece **alta escalabilidade** (milhares de conexões) - `asyncio` é a biblioteca padrão (sem instalação extra) - Evite **mixing sync e async** no mesmo código

Próximos passos: - Pratique com `aiohttp` para requisições HTTP - Explore frameworks async (FastAPI, aiohttp.web) - Estude `asyncio.gather()` para executar múltiplas tarefas - Aprenda sobre async context managers e iterators

Com a programação assíncrona, seu código nunca mais precisará ficar parado esperando I/O!

Espero que esse tutorial tenha sido útil, e até a próxima, Pythonista!

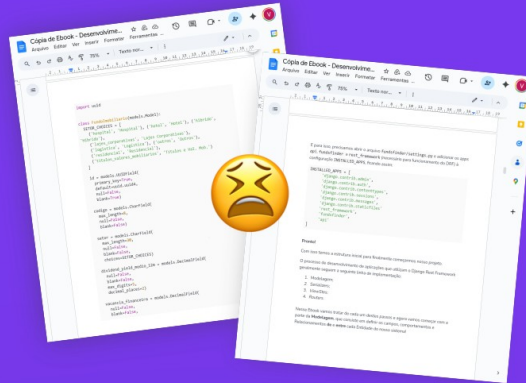
Não se esqueça de conferir!



DevBook

Crie Ebooks técnicos em minutos com IA


Conheça a 1ª IA Especializada na criação de Ebooks **com código!**



Chega de formatar código no Google Docs



 Syntax Highlight

 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