Clase 049 — async / httpx / aiohttp para data scientists

Parte: 0 — Prerrequisitos · Fuente: docs asyncio + httpx + Beazley Python Concurrency.

Duración estimada: 80 min.

Clase 049 — async / httpx / aiohttp para data scientists

> *Parte: 0 — Prerrequisitos · Fuente: docs asyncio + httpx + Beazley Python Concurrency. Duración estimada: 80 min.*

Objetivo

Aprender asyncio y httpx —el HTTP client moderno con soporte sync + async + HTTP/2— para hacer scraping y consumo de APIs en paralelo sin bloquear. Pasar de "1 request por segundo" con requests a "100+ concurrentes" con httpx.AsyncClient. Comparar con aiohttp (alternativa popular) y concurrent.futures (parallelism con threads).

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Definir async def y await; ejecutar con asyncio.run.
- Usar httpx.AsyncClient para fetches concurrentes con asyncio.gather.
- Limitar concurrencia con asyncio.Semaphore para no DOS-ear a la API.
- Implementar rate limiting + retries con backoff exponencial.
- Decidir entre asyncio, threading, multiprocessing según I/O-bound vs CPU-bound.

Temas

- Event loop, coroutines, await.
- httpx: API unificada sync/async, HTTP/2, timeouts.
- asyncio.gather, asyncio.as_completed.
- Semaphore para limitar concurrencia.
- Backoff exponencial con tenacity o backoff lib.
- aiohttp como alternativa (más antigua, más utilities).
- Cuándo NO async: tareas CPU-bound (usar multiprocessing).

Definiciones y características

- Coroutine: función async def, devuelve un objeto coroutine que el event loop ejecuta.
- await: cede control al event loop hasta que la operación termina.
- Event loop: scheduler que ejecuta coroutines cooperativamente.
- asyncio.gather(*coros): ejecuta varias coroutines concurrentemente, espera todas.
- asyncio.as_completed(coros): itera resultados a medida que llegan.
- asyncio.Semaphore(n): limita a n coroutines concurrentes.
- httpx: HTTP client moderno (Tom Christie, dev de Django REST Framework).
- aiohttp: client + server async. Más viejo, más maduro, sin sync API.

Dataset / recursos

- Una API pública lenta: https://httpbin.org/delay/2 (espera 2 seg).

- Lista de 100 URLs para fetcher.
- Librerías: httpx, aiohttp, opcional tenacity.

Ejercicios

1. Sync baseline: fetcher de 50 URLs con requests.get en loop. Medir tiempo.
2. Async con httpx: async with httpx.AsyncClient() as c: results = await asyncio.gather(*[c.get(url) for url in urls]). Comparar tiempo (debería ser ~20-50× más rápido).
3. Semaphore: limitar a 10 concurrent. Útil para no ser bloqueado por rate limits.
4. Retry exponencial: usar tenacity.retry(stop=stop_after_attempt(5), wait=wait_exponential(min=1, max=30)) sobre un fetcher.
5. httpx vs aiohttp: hacer el mismo benchmark con ambos. Similar performance; httpx tiene mejor DX.

Homework verificable

Scraper concurrente de 200 URLs (una página de Wikipedia o API pública):

1. Implementar con httpx.AsyncClient + asyncio.gather.
2. Limitar a 20 concurrent con Semaphore.
3. Retry con backoff sobre errores 5xx.
4. Reportar tiempo total y % de éxito.

Criterio de aceptación: tiempo total ≤ 1/10 de la versión sync; success rate ≥ 95 %.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
RuntimeError: This event loop is already r	Jupyter ya tiene un loop. Fix: nest_asynci
Olvido await → coroutine never awaited war	Fix: await coro() o asyncio.run(coro()).
Mezclar sync requests adentro de coroutine	Bloquea el event loop. Fix: usar httpx.Asy
Sin limit → API te banea	Demasiada concurrencia. Fix: Semaphore + d
AsyncClient no cerrado	Resource leak. Fix: async with httpx.Async

Preguntas frecuentes

async o threading?

Async para I/O-bound (HTTP, DB) — más eficiente, menos overhead. Threading para I/O-bound legacy code. Multiprocessing para CPU-bound (cálculo numérico, ML training).

httpx o aiohttp?

httpx es default moderno: API unificada sync/async, mejor DX, HTTP/2. aiohttp tiene más utilities (sessions, file uploads) pero solo async.

async con pandas / numpy?

No tiene sentido — son CPU-bound. Async brilla cuando esperás de I/O externo.

Test de async code?

pytest-asyncio con @pytest.mark.asyncio.

async en Django / Flask?

Django 4+ soporta async views. Flask 2+ también. FastAPI es async-native (default moderno).

Referencias

- httpx docs
- asyncio docs
- aiohttp docs
- Beazley, D. Python Concurrency from the Ground Up (PyCon 2015).
- tenacity para retries.

Siguiente parte

Clase 050 — Panorama del ML: tipos, batch vs online, instance vs model-based

Apéndice: notebook (primer bloque)

Self-contained: levantamos un mock server local con aiohttp (sin internet) y comparamos requests sync vs httpx.AsyncClient con asyncio.gather. Vemos Semaphore, retry con backoff y patrón productor-consumidor.

```
import asyncio, time, threading, random

try:
    import httpx
    import aiohttp
    from aiohttp import web
    import requests
except ImportError as e:
    raise ImportError('Instalá: pip install httpx aiohttp requests') from e

# En Jupyter el loop ya corre — habilitamos nesting si está disponible
try:
    import nest_asyncio; nest_asyncio.apply()
except ImportError:
    pass

random.seed(42)
print(f'httpx {httpx.__version__} | aiohttp {aiohttp.__version__} | requests {requests.__version__}')
```

Archivos complementarios

- notebook.ipynb