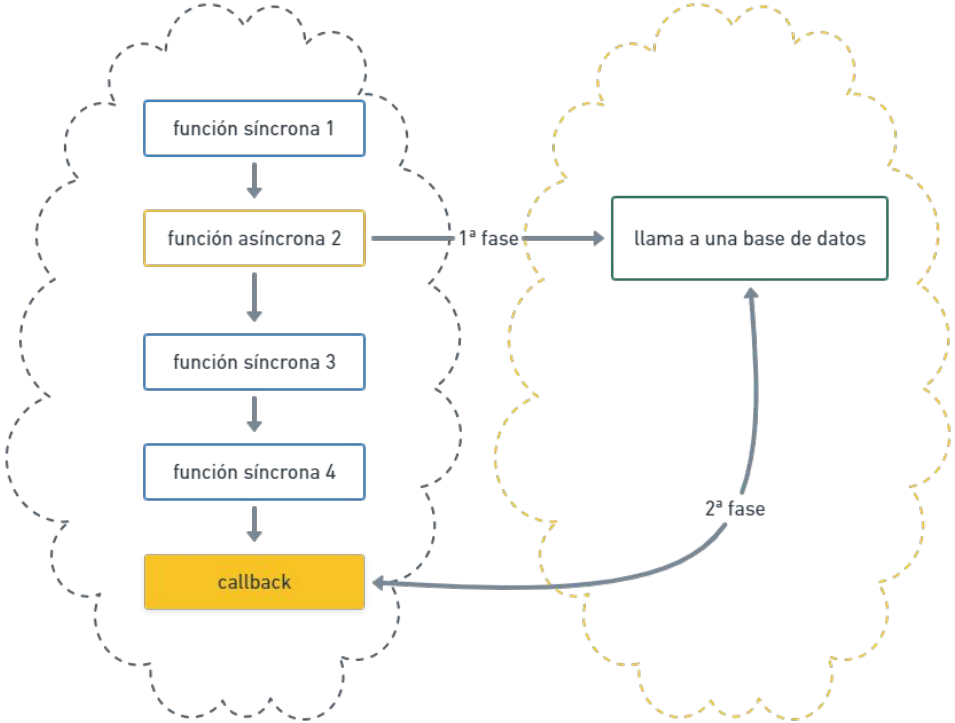
Asynchronous

The objective of this class is to introduce students to the basic concepts of asynchronous programming in Python, highlighting the differences and advantages over traditional synchronous programming. Practical examples will be provided to demonstrate how to implement asynchronous code and the concepts of asyncio, a Python library for writing concurrent code using the **async/await** syntax, will be explained.

Created by Adrian Carmona

What is Asynchronous Programming?

Asynchronous programming is a programming model that allows time-consuming operations (such as I/O) to run in the "background", allowing the main program to continue its execution without being blocked. This is achieved through the use of coroutines, which are functions that can pause their execution and resume at a later point.

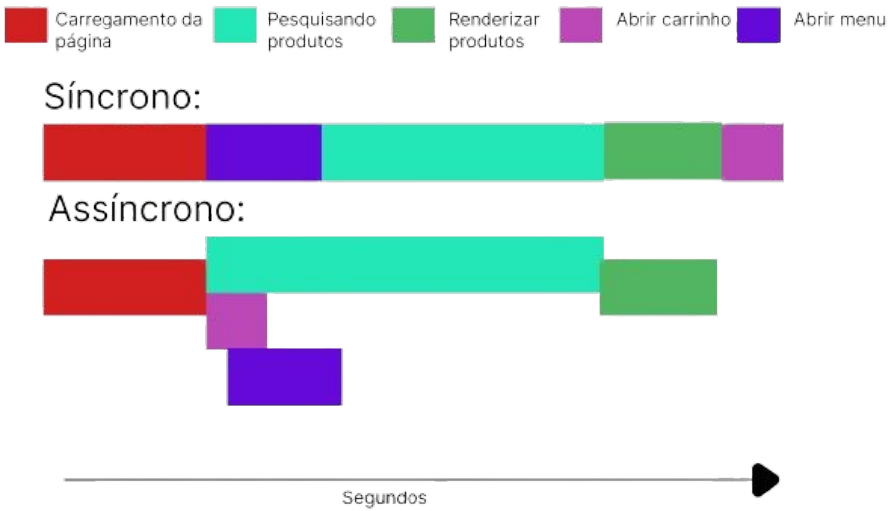


Advantages:

- Increased resource utilization: no more main thread deadlocks! Asynchronous scheduling enables concurrent I/O operations, maximizing efficiency and performance.

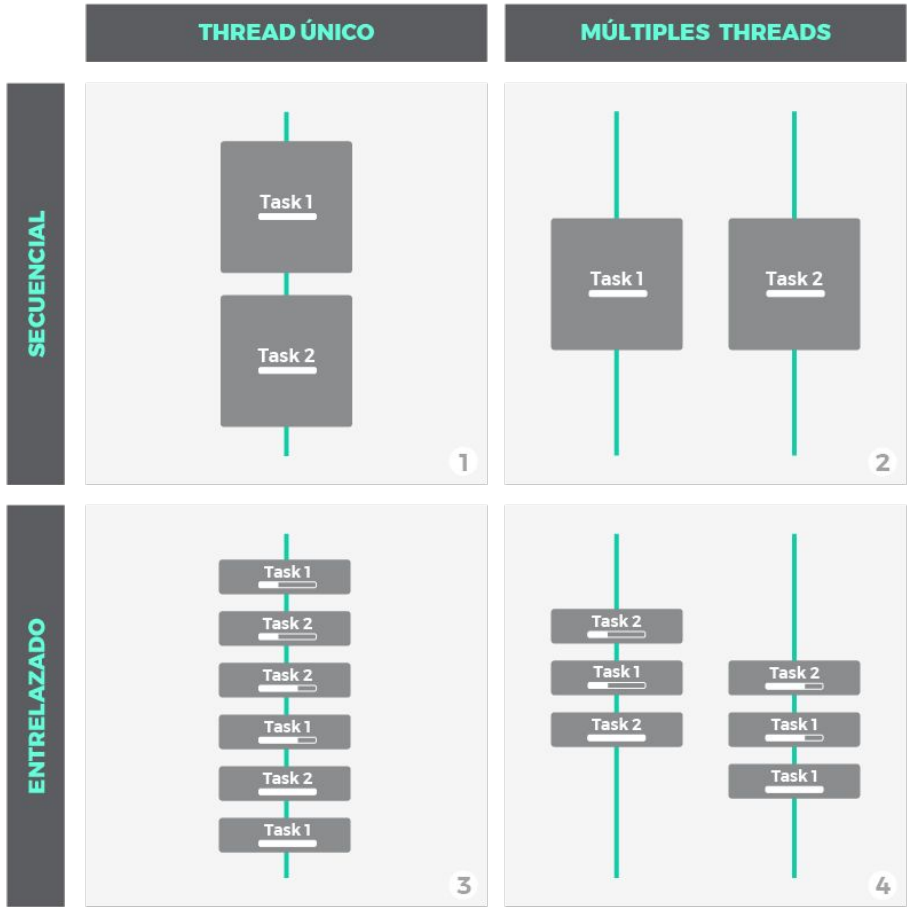
- Smooth user experience: Say goodbye to unresponsive applications. Asynchronous scheduling maintains UI and UX responsiveness even during long-running operations.

- Effortless scalability: Handling large volumes of network connections or requests becomes child's play with asynchronous scheduling.



Key Differences from Synchronous Programming:

- Execution Flow: Synchronous code executes line by line, while asynchronous code can pause and resume, allowing for concurrent operations.
- I/O Handling: Asynchronous programming shines in I/O-intensive scenarios, minimizing wait times and maximizing throughput.



Created by Adrian Carmona

Coding Examples

```python
import requests
import time

def descargar_archivo(url):
    respuesta = requests.get(url)
    nombre_archivo = url.split('/')[-1]
    with open(nombre_archivo, 'wb') as f:
        f.write(respuesta.content)
    print(f'Descargado {nombre_archivo}')

def main():
    urls = [
        'http://example.com/archivo1',
        'http://example.com/archivo2',
        'http://example.com/archivo3',
    ]

    inicio = time.time()
    for url in urls:
        descargar_archivo(url)
    duracion = time.time() - inicio
    print(f'Tiempo total de descarga: {duracion}
segundos')
if __name__ == '__main__':
    main()
```

```python
import asyncio
import aiohttp
import time

async def descargar_archivo(session, url):
    async with session.get(url) as respuesta:
        nombre_archivo = url.split('/')[-1]
        with open(nombre_archivo, 'wb') as f:
            while True:
                chunk = await respuesta.content.read(1024)
                if not chunk:
                    break
                f.write(chunk)
        print(f'Descargado {nombre_archivo}')

async def main():
    urls = [
        'http://example.com/archivo1',
        'http://example.com/archivo2',
        'http://example.com/archivo3',
    ]

    async with aiohttp.ClientSession() as session:
        tareas = [descargar_archivo(session, url) for url in
urls]   inicio = time.time()
        await asyncio.gather(*tareas)
        duracion = time.time() - inicio
        print(f'Tiempo total de descarga: {duracion} segundos')

if __name__ == '__main__':
    asyncio.run(main())
```

In Fact

These examples demonstrate how the asynchronous programming model can significantly improve efficiency by performing multiple I/O operations simultaneously, compared to the traditional synchronous approach. In the next class, we will further explore coroutines, task scheduling, and how to handle exceptions in an asynchronous environment.