The difference between **synchronization** and **asynchronization** lies in how tasks or operations are managed in terms of time and execution, particularly when dealing with multiple tasks or processes.

**Synchronization**

**Definition**: Synchronization involves executing tasks one after another in a specific order. Each task must complete before the next one starts. This approach ensures that tasks are performed in a well-defined sequence.

**Blocking**: Synchronized tasks are "blocking." This means that if a task is time-consuming, it will block the execution of subsequent tasks until it finishes.

**Example**: Imagine you are at a bank teller. Each customer is served one by one in the order they arrived. If one customer takes a long time, everyone else has to wait.

**Use Cases**: Synchronization is useful when tasks depend on each other, need to share resources safely, or require strict ordering (e.g., reading from and writing to a shared file).

**Asynchronization**

**Definition**: Asynchronization involves executing tasks in a non-blocking manner, allowing multiple tasks to be in progress simultaneously. Tasks do not have to wait for each other to complete and can start as soon as they are ready.

**Non-Blocking**: Asynchronous tasks are "non-blocking." When an operation encounters a wait (e.g., waiting for data to load from a server), it does not pause the entire program. Instead, it continues with other tasks.

**Example**: This is like being in a restaurant with multiple waiters. While one waiter takes your order, another can serve food to someone else, and another can handle billing. The tasks are done concurrently.

**Use Cases**: Asynchronization is useful for tasks that involve waiting (like network requests or I/O operations), as it allows other tasks to proceed without delay.

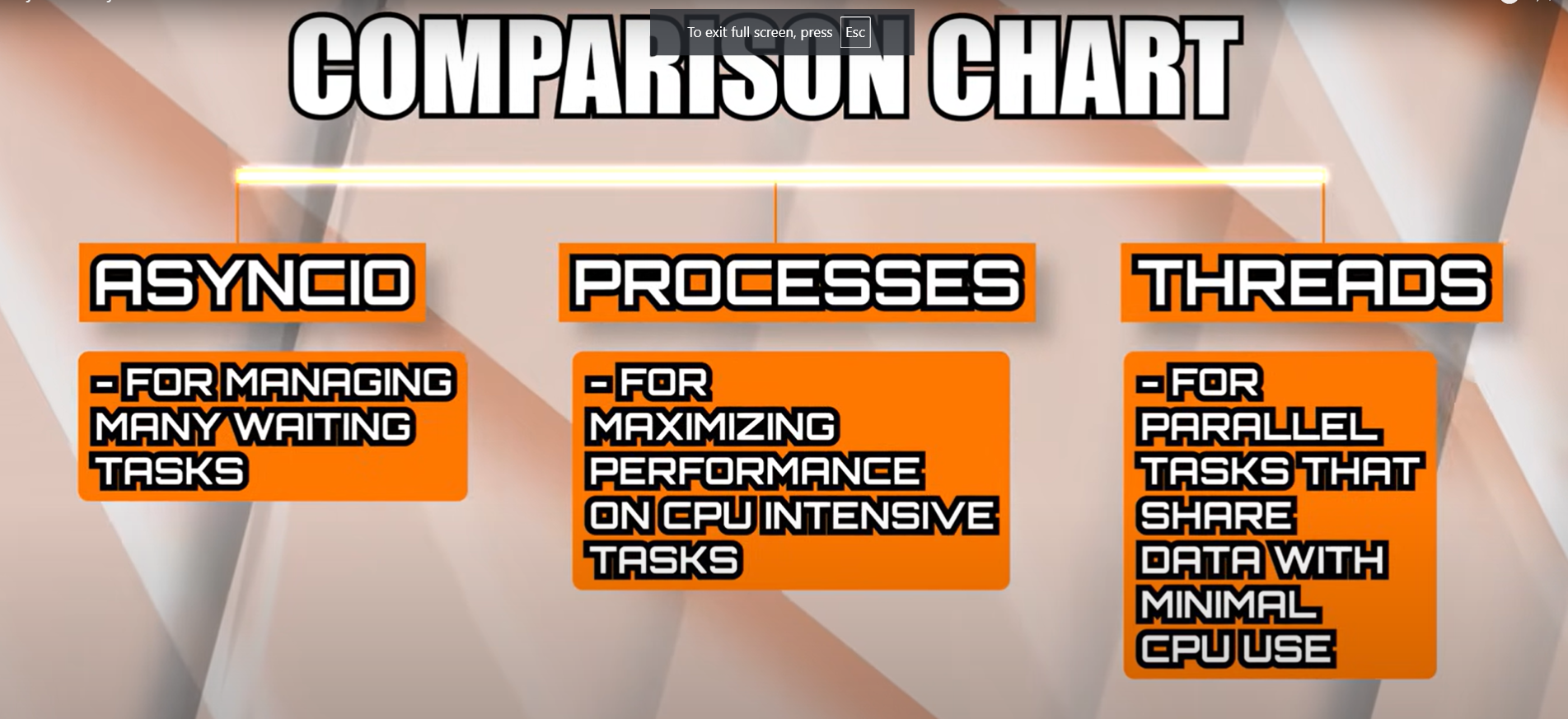
**asynchronous** code described above is also sometimes called **"concurrency"**. It is different from **"parallelism"**.

**asyncio**: the Python package that provides a foundation and API for running and managing coroutines.

**Parallelism** consists of performing multiple operations at the same time. **Multiprocessing** is a means to effect parallelism, and it entails spreading tasks over a computer’s central processing units (CPUs, or cores). Multiprocessing is well-suited for CPU-bound tasks: tightly bound [for loops](https://realpython.com/python-for-loop/) and mathematical computations usually fall into this category.

**Concurrency** is a slightly broader term than parallelism. It suggests that multiple tasks have the ability to run in an overlapping manner. (There’s a saying that concurrency does not imply parallelism.)

**Threading** is a concurrent execution model whereby multiple [threads](https://en.wikipedia.org/wiki/Thread_(computing)) take turns executing tasks. One process can contain multiple threads. Python has a complicated relationship with threading thanks to its [GIL](https://realpython.com/python-gil/),



**Coroutines:-**

At the heart of async IO are coroutines. A coroutine is a specialized version of a Python generator function. Let’s start with a baseline definition and then build off of it as you progress here: a coroutine is a function that can suspend its execution before reaching return, and it can indirectly pass control to another coroutine for some time.

­­

1:- **time.sleep()** is like telling your program to pause completely for a few seconds (or however long you specify). During this time, your program can’t do anything else—it’s just sitting there, waiting. This is called "blocking" because nothing else can happen until the wait is over.

2:- **asyncio.sleep()** is different. It still tells your program to wait, but in a way that allows it to do other things during the wait time. This is "non-blocking" because your program can keep working on other tasks while it’s waiting.

**An Intro to Threading in Python**

**Multithreading**is defined as the ability of a processor to execute multiple threads concurrently. In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed **context switching**. In context switching, the state of a thread is saved and the state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallelly (this is termed **multitasking**.