How to Choose the Right Python Concurrency API

Python standard library offers 3 concurrency APIs.

How do you know which API to use in your project?

In this tutorial, you will discover a helpful step-by-step procedure and helpful questions to guide you to the most appropriate concurrency API.

After reading this guide, you will also know how to choose the right Python concurrency API for current and future projects.

Let's get started.

Table of Contents

Problem of Python's Concurrency APIs

The Python standard library (stdlib) offers three ways to execute tasks concurrently in your programs.

They are the **multiprocessing** module for process-based concurrency, the **threading** module for thread-based concurrency, and the **asyncio** module for coroutine-based concurrency.

The abundance of choices makes it confusing.

It's worse than you think:

- For example, after choosing a module, should you use a pool of workers or should you code up the concurrent task yourself?
- If you choose a pool of workers, you should use the Pools API or the Executors API.

Even experienced Python developers are baffled by the options.

Which Python concurrency API should you use for your project?

You need rules of thumb to guide you to the most appropriate concurrency API.

Before we look at a handy procedure for choosing, let's take a closer look at the problem and structure it carefully.

Which Python Concurrency API Should You Use?

You want to use concurrency in your Python program.

There's just one problem. Which API should you use?

You're not alone. This is perhaps one of the most common questions I receive.
That's why I wrote this guide.

Firstly, there are three main Python concurrency APIs, they are:

- **Coroutine-based**, provided by the "*asyncio*" module.
- **Thread-based**, provided by the "*threading*" module.
- **Process-based**, provided by the "*multiprocessing*" module.

Choosing among these three is relatively straightforward. I'll show you how in the next section.

The problem is, that there are further decisions to make.

You will need to consider whether you should use a pool of reusable workers or not.

For example:

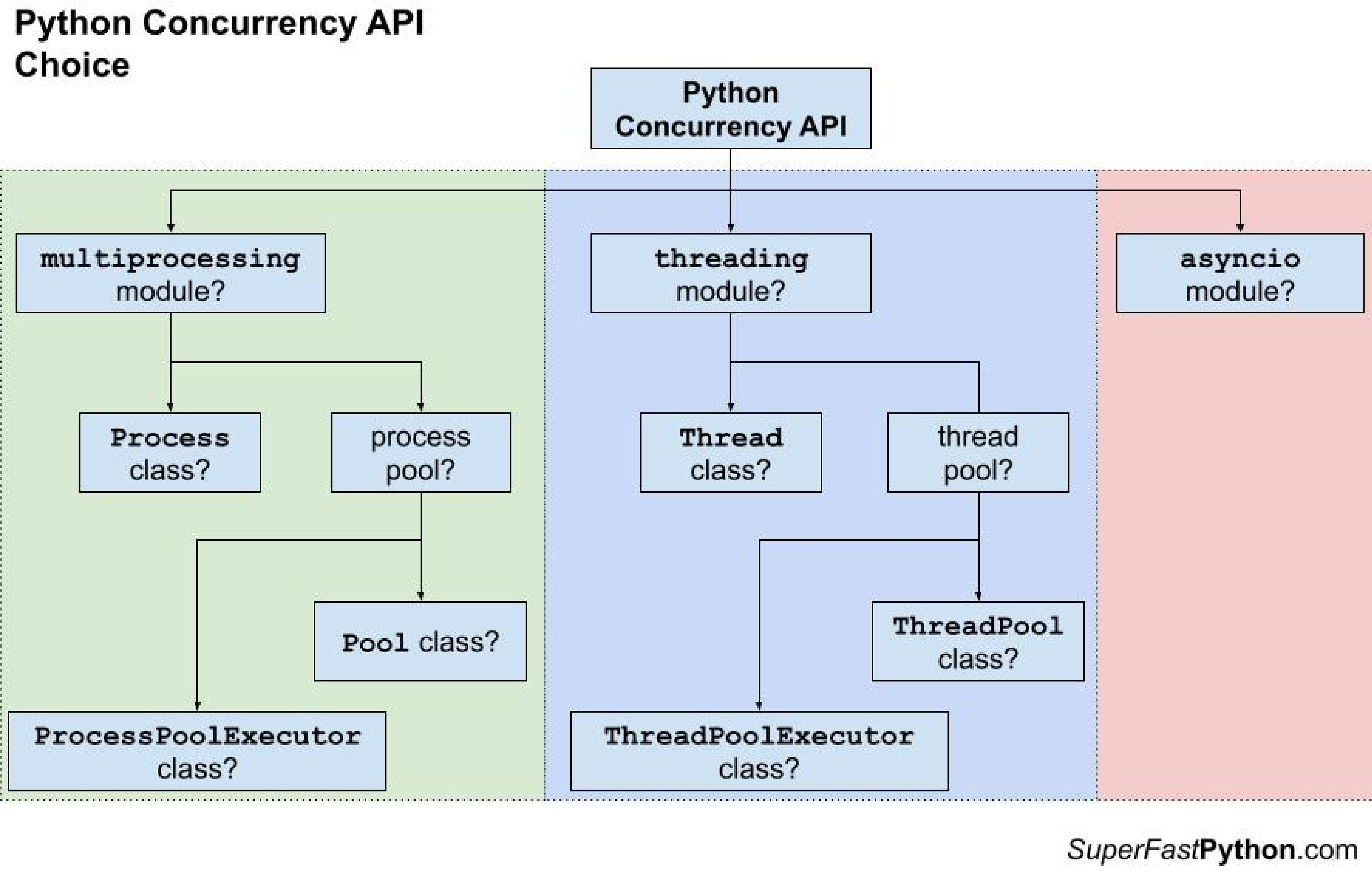
- If you decide you need thread-based concurrency, should you use a thread pool or use the **Thread** class in some way?
- If you decide you need process-based concurrency, should you use a process pool or use the **Process** class in some way?

Then, if you decide to use reusable workers for concurrency, you have multiple options to choose from.

For example:

- If you decide you need a thread pool, should you use the **ThreadPool** class or the **ThreadPoolExecutor**?
- If you decide you need a process pool, should you use the **Pool** class or the **ProcessPoolExecutor**?

The below figure summarizes these decision points.



(https://superfastpython.com/wp-content/uploads/2022/07/Python-Concurrency-API-Choice.jpg)

PYTHON CONCURRENCY API CHOICE (CLICK TO ENLARGE)

How do you figure all of this out?

How do you choose a Python concurrency API for your project?

Process to Choose a Python Concurrency API

One approach you can use, which is perhaps the most common, is to choose one API in an ad hoc manner.

Many development decisions are made this way and your program will probably work fine.

But maybe it won't.

I recommend a 3 step process when choosing the right Python concurrency API for your project.

The steps are as follows:

1. **Step 1**: CPU-Bound vs IO-Bound (Multiprocessing vs Threading)
    1. **Step 1.1** Choosing Between AsyncIO and Threading
2. **Step 2**: Many Ad Hoc Tasks vs One Complex Task?
3. **Step 3**: Pools vs Executors?

I have also distilled the decisions into a handy picture, as follows:

(https://superfastpython.com/wp-content/uploads/2022/07/Python-Concurrency-API-Decision-Tree.jpg)

PYTHON CONCURRENCY API DECISION TREE (CLICK TO ENLARGE)

Next, let's take a closer look at each step and add some nuance.

Step 1: CPU-Bound vs IO-Bound Tasks?

The first step in choosing the Python concurrency API to use is to think about the limiting factor of the task or tasks you want to execute.

Are the tasks mostly CPU-bound or IO-bound?

If you get this part right, the other decisions you need to make are less important.

Let’s take a closer look at each in turn.

CPU-Bound Tasks

A CPU-bound task is a type of task that involves performing computation and does not involve IO.

The operations only involve data in main memory and performing computations on or with that data.

As such, the limit on these operations is the speed of the CPU. This is why we call them CPU-bound tasks.

Examples include:

- Calculating points in a fractal.
- Estimating Pi
- Factoring primes.
- Parsing HTML, JSON, etc. documents.
- Processing text.
- Running simulations.

CPUs are very fast and we often have more than one CPU. We would like to perform our tasks and make full use of multiple CPU cores in modern hardware.

Now that we are familiar with CPU-bound tasks, let’s take a closer look at IO-bound tasks.

IO-Bound Tasks

An IO-bound task is a type of task that involves reading from or writing to a device, file, or socket connection.

The operations involve input and output (IO), and the speed of these operations is bound by the device, hard drive, or network connection. This is why these tasks are referred to as IO-bound.

CPUs are really fast. Modern CPUs, like a 4GHz, can execute 4 billion instructions per second, and you likely have more than one CPU in your system.

Doing IO is very slow compared to the speed of CPUs.

Interacting with devices, reading and writing files, and socket connections involve calling instructions in your operating system (the kernel), which will wait for the operation to complete. If this operation is the main focus for your CPU, such as executing in the main thread of your Python program, then your CPU is going to wait many milliseconds, or even many seconds, doing nothing.

That is potentially billions of operations that it is prevented from executing.

Examples of IO-bound tasks include:

- Reading or writing a file from the hard drive.
- Reading or writing to standard output, input, or error (stdin, stdout, stderr).
- Printing a document.
- Downloading or uploading a file.
- Querying a server.
- Querying a database.
- Taking a photo or recording a video.
- And so much more.

Now that we are familiar with both CPU-bound and IO-bound tasks, let’s consider the types of Python concurrency APIs we should use.

Choose Python Concurrency API

Recall that the multiprocessing module provides process-based concurrency and the threading module provides thread-based concurrency within a process.

Generally, you should use process-based concurrency if you have a CPU-bound task and thread-based concurrency if you have an IO-bound task.

- **CPU-Bound Tasks**: Use the “**multiprocessing**” module for process-based concurrency.
- **IO-Bound Tasks**: Use the “**threading**” module for thread-based concurrency.

The multiprocessing module is suitable for tasks that focus on computing or calculating something with relatively little data shared between tasks. Multiprocessing is not suitable for tasks that send or receive a lot of data with other processes because of the computational overhead added and because all data shared between processes must be serialized.

You would execute one task per logical CPU core or one per physical CPU core and maximize the capabilities of the underlying hardware.

The threading module is suitable for tasks that focus on reading or writing from an IO device with relatively little calculation. Threading is not suitable for tasks that perform a lot of CPU computation as the Global Interpreter Lock (GIL) prevents more than one Python thread from executing at a time. The GIL is generally only released when performing blocking operations, like IO, or specifically in some third-party C libraries, such as NumPy.

You can execute tens, hundreds, or thousands of thread-based tasks to maximize the capabilities of the underlying hardware as IO spends most of the time waiting.

The following figure summarizes the decision point in choosing between the threading module for thread-based concurrency or the multiprocessing module for process-based concurrency.

(https://superfastpython.com/wp-content/uploads/2022/07/Python-Concurrency-API-threading-vs-multiprocessing.jpg)

PYTHON CONCURRENCY API THREADING VS MULTIPROCESSING (CLICK TO ENLARGE)

You can learn more about choosing between using a **Thread** and using a **Process** in the tutorials:

- Thread vs Process in Python (https://superfastpython.com/thread-vs-process/)
- Threading vs Multiprocessing in Python (https://superfastpython.com/threading-vs-multiprocessing-in-python/)
- Why Not Always Use Processes in Python (https://superfastpython.com/why-not-always-use-processes-in-python/)

Also these guides on threading and multiprocessing will help:

- Python Multiprocessing Complete Guide (https://superfastpython.com/multiprocessing-in-python/)

- Python Threading Complete Guide (https://superfastpython.com/threading-in-python/)

Next, let’s consider if AsyncIO is appropriate.

Step 1.1 Choosing Between Threading and AsyncIO

If your tasks are mostly IO-bound, you have another decision point.

You must choose between using the “**threading**” module and using the “**asyncio**” module.

Recall that the threading module provides thread-based concurrency and the asyncio module provides coroutine-based concurrency within a thread.

Generally, you should use coroutine-based concurrency if you have many socket connections (or prefer asynchronous programming), and thread-based concurrency otherwise.

- **Many Socket-Connections**: Use the “**asyncio**” module for coroutine-based concurrency.
- **Otherwise**: Use the “**threading**” module for thread-based concurrency.

The asyncio module focuses on concurrent non-blocking IO for socket connections. For example, if your IO tasks are file-based, then asyncio would not be an appropriate choice, at least for this reason alone.

The rationale is that coroutines are more lightweight than threads, therefore a single thread may host many more coroutines than a process may manage threads. For example, asyncio may allow thousands, tens of thousands, or more coroutines for socket-based IO as opposed to hundreds to low thousands of threads in the threading API.

Another consideration is that you may want or need to use an asynchronous programming paradigm in developing your program, e.g. async/wait. Therefore this requirement would override any requirements imposed by the tasks.

Similarly, you may have an aversion to the asynchronous programming paradigm and therefore this preference would override any requirements imposed by the tasks.

For more on the differences between asyncio and thread-based concurrency, see the tutorial:

- ThreadPoolExecutor vs. AsyncIO in Python (https://superfastpython.com/threadpoolexecutor-vs-asyncio/)

Step 2: Many Ad Hoc Tasks vs One Complex Task?

The second step is to consider if you need to execute independent ad hoc tasks or a large complex task.

What we are thinking about at this decision point is whether you need to issue one or many ad hoc tasks that may benefit from a pool of reusable workers. Otherwise, whether you need a single task where a pool of reusable workers would be overkill.

Another way to think about it is whether you have one or a few different but complex tasks like monitors, schedulers, or similar that might live for a long time, such as the duration of the program. These would not be ad hoc tasks, and may not benefit from a reusable pool of workers.

- **Shorter-lived and/or many ad hoc**: Use a thread or process pool.
- **Longer-lived and/or complex tasks**: Use the **Thread** or **Process** class.

In the case where you have chosen thread-based concurrency, the choice is between a thread pool or using the Thread class.

In the case where you have chosen process-based concurrency, the choice is between a process pool or using the Process class.

Some additional considerations include:

- **Heterogeneous vs. Homogeneous Tasks**: A pool is perhaps more appropriate for a diverse set of tasks (heterogeneous) whereas a Process/Thread class is appropriate for one type of task (homogeneous).
- **Reuse vs. Single Use**: A pool is appropriate for reusing the basis of concurrency, e.g. reusing a thread or a process for many tasks, whereas a Process/Thread class is appropriate for a single use task, perhaps a long-lived one.

- **Multiple Tasks vs. Single Task**: A pool naturally supports many tasks, perhaps issued in many ways, whereas a Process/Thread class only supports one type of task, once configured or overridden.

To make this more concrete, let’s consider some examples:

- A for-loop that calls a function many times with different arguments in each iteration may be appropriate for a thread pool as workers can be reused for each task automatically as needed.
- A background task that monitors a resource may be appropriate for a Thread/Process class as it is a long-running single task and may have a lot of complex and specialized functionality perhaps spread across many function calls.
- A script that downloads many files would be appropriate for a pool of workers as each task is short in duration and there may be many more files than there are workers, allowing reuse of workers and queuing of tasks to complete.
- A one-off task that maintains an internal state and interacts with the main program may be appropriate for Thread/Process class as the class can be overridden to use instance variables for state and methods for modular functionality.

The figure below may help choose between using a pool of workers vs a **Thread** or **Process** class.

(https://superfastpython.com/wp-content/uploads/2022/07/Python-Concurrency-API-Worker-Pool-vs-Class.png)

PYTHON CONCURRENCY API WORKER POOL VS CLASS (CLICK TO ENLARGE)

For more on pools of workers vs one complex task, see:

- ThreadPoolExecutor vs. Thread in Python (https://superfastpython.com/threadpoolexecutor-vs-threads/)
- Multiprocessing Pool vs Process in Python (https://superfastpython.com/multiprocessing-pool-vs-process/)

Next, let's consider the type of pool that might be considered.

Step 3: Pools vs Executors?

The third step is to consider the type of worker pool to use.

There are two main types, they are:

- **Pool**: **multiprocessing.pool.Pool** and the port of the class to support threads in **multiprocessing.pool.ThreadPool**.
- **Executors**: The **concurrent.futures.Executor** class and two sub-classes **ThreadPoolExecutor** and **ProcessPoolExecutor**.

Both provide pools of workers. The similarities are many and the differences are few and subtle.

For example, the similarities are:

- Both have thread- and Process-based versions.
- Both can execute ad hoc tasks.
- Both support synchronous and asynchronous task execution.
- Both provide support for checking the status and waiting for asynchronous tasks.
- Both support callback functions for asynchronous tasks.

Choosing one over the other will not make a big impact on your program.

The main difference is in the API offered by each, specifically minor differences in focus or in how tasks are handled.

For example:

- Executors provide the ability to cancel issued tasks, whereas the Pool does not.
- Executors provide the ability to work with collections of heterogeneous tasks, whereas the Pool does not.
- Executors do not provide the ability to forcefully terminate all tasks, whereas the Pool does.
- Executors do not provide multiple parallel versions of the **map()** function, whereas the Pool does.
- Executors provide the ability to access an exception raised in a task, whereas the Pool does not.

The difference between these pools that I think matters is that the Pool is focused on concurrent for-loops with many different versions of the map() function, e.g. apply a function to each argument in an iterable.

The Executors have this capability but the focus is more on issuing ad hoc tasks asynchronously and managing the collection of tasks.

The figure below helps to summarize the differences between pools and executors

(https://superfastpython.com/wp-content/uploads/2022/07/Python-Concurrency-API-Pools-vs-Executors.png)

PYTHON CONCURRENCY API POOLS VS EXECUTORS (CLICK TO ENLARGE)

For more on Pools vs Executors see:

- Multiprocessing Pool vs ProcessPoolExecutor in Python (https://superfastpython.com/multiprocessing-pool-vs-processpoolexecutor/)
- Multiprocessing Pool in Python: The Complete Guide (https://superfastpython.com/multiprocessing-pool-python/)

Takeaways

You now know how to choose between the different Python concurrency APIs

Did this guide help?

Let me know (https://superfastpython.com/contact/).

Do you have any questions?

Message me any time (https://superfastpython.com/contact/) and I will do my best to help.