SuperFastPython
making python developers awesome at concurrency (https://superfastpython.com/)

Concurrent Programming in Python

DECEMBER 24, 2022 by JASON BROWNLEE (/ABOUT) in PYTHON CONCURRENCY (HTTPS://SUPERFASTPYTHON.COM/CATEGORY/CONCURRENCY/)

Last Updated on November 25, 2023

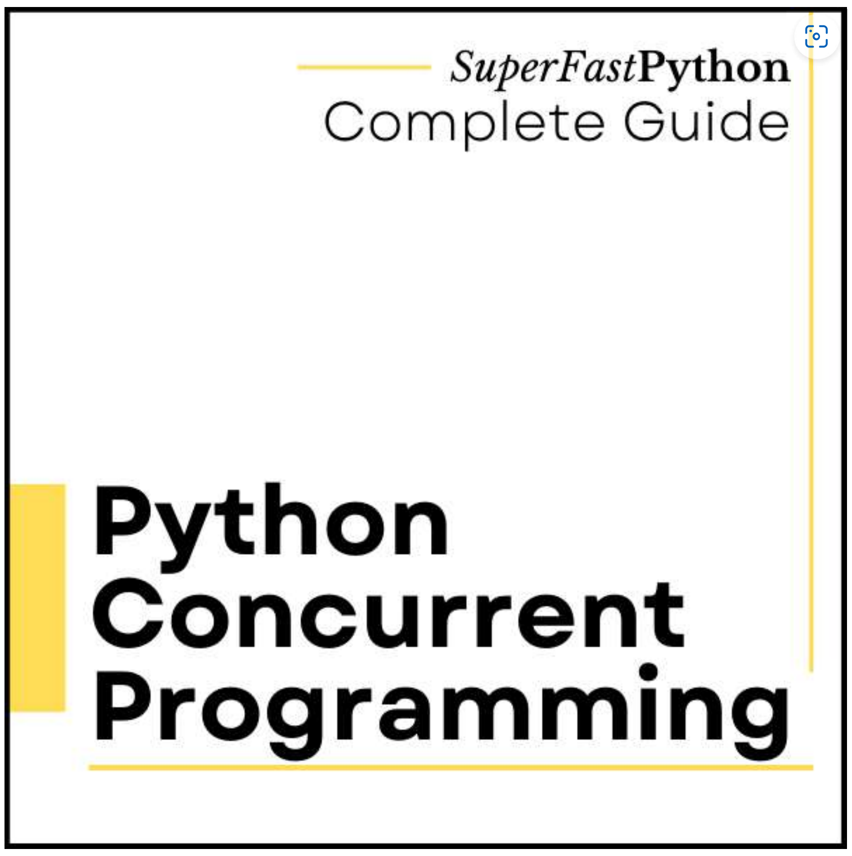
Concurrent programming refers to a type of programming focused on executing independent tasks at the same time.

Unlike traditional programming where instructions or tasks are executed one after the other, concurrent programming allows multiple tasks to make progress at the same time. It facilitates other types of programming, such as parallel programming where tasks are executed simultaneously on separate CPUs.

In this tutorial, you will discover concurrent programming in Python.

- You will discover what concurrent means and how it is similar but different from parallel.
- You will discover the discipline of concurrent programming including the primitives, patterns, and failure modes unique to this style of programming.
- You will discover the modules and classes in the Python standard library that support concurrent programming that you can bring to your own programs.

Let's get started.



CONCURRENT PROGRAMMING IN PYTHON

Table of Contents

What is Concurrent

Before we dive into concurrent programming in Python, let's understand what "concurrent" means.

Concurrent means simultaneous or at the same time, as opposed to asynchronous which means not at the same time.

> *concurrent: operating or occurring at the same time*
>
> — MERRIAM-WEBSTER DICTIONARY (HTTPS://WWW.MERRIAM-WEBSTER.COM/DICTIONARY/ASYNCHRONOUS)

For example, when programming, if we refer to two or more tasks as executing concurrently (https://en.wikipedia.org/wiki/Concurrency_(computer_science)), we mean that they execute at the same time.

- **Concurrent Tasks**: Two or more tasks executed at the same time.

The key to understanding concurrent tasks is that the tasks are independent. They are discrete units of execution that are separate from each other.

> *Concurrency is about multiple tasks that can happen independently from one another.*
>
> — PAGE 6, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3CZ7ZH6), 2022.

As such, they may execute out of sequence, such as at the same time, although they are not required to execute in sequence, e.g. one after the other.

Let's make concurrency concrete. We do things concurrently all the time.

- Breathing and reading.
- Driving and talking.
- Programming and listening to music.

We could drive, then talk, but that would be strange. Instead, we do both things concurrently.

Our body may or may not be able to strictly request a turn of the wheel and speak a word simultaneously, but we can simulate some or all of it by switching between the tasks quickly. We're not concerned about how the tasks are executed at the same time, only that we get the results.

Therefore, although tasks may be independent and capable of concurrent execution, there is no restriction of when they are executed.

> *Concurrency: The capability of having more than one computation in progress at the same time. These computations may be on separate cores or they may be sharing a single core by being swapped in and out by the operating system at intervals.*
>
> — PAGE 266, THE ART OF CONCURRENCY (HTTPS://AMZN.TO/3TKCUWX), 2009.

This means that concurrent tasks may execute in any order, such as in parallel, overlapping, interleaved, and more.

The details of how concurrent tasks are executed are left to the underlying system, such as the software library and/or system hardware.

> *Concurrency enables a computer to do many different things seemingly at the same time. For example, on a computer with one CPU core, the operating system rapidly changes which program is running on the single processor. In doing so, it interleaves execution of the programs, providing the illusion that the programs are running simultaneously.*
>
> — PAGE 225, EFFECTIVE PYTHON (HTTPS://AMZN.TO/3SY9FQ1), 2019.

Run loops using all CPUs, download your FREE book (https://superfastpython.com/plip-incontent) to learn how.

What is Concurrent Programming

We often describe concurrent tasks, however, concurrency is broader.

In fact, concurrent computing or concurrent programming (https://en.wikipedia.org/wiki/Concurrent_computing) is an entire sub-discipline of computer science.

> *Concurrent programming is all about independent computations that the machine can execute in any order.*
>
> — PAGE 6, THE ART OF CONCURRENCY (HTTPS://AMZN.TO/3TKCUWX), 2009.

It refers to the techniques, patterns, algorithms, and data structures required to execute tasks concurrently.

- **Concurrent programming**: The suite of methods used to achieve and support the concurrent execution of tasks.

Concurrent Programming is Orthogonal

Concurrent programming is orthogonal to the programming paradigm, such as procedural programming, functional programming, or object-oriented programming.

For example:

- Execute routines concurrently in procedural programming.
- Execute functions concurrently in functional programming.
- Execute methods concurrently in object-oriented programming.

This means we can overlay concurrent programming over the top of our functions in procedural programming or our classes and objects in object-oriented programming.

Concurrent Programming Primitives

Concurrent programming is a discipline.

As such, there is a suite of primitives for coordinating and synchronizing in code that belongs to the field.

Examples include:

- Mutual exclusion locks, often called mutex locks to protect a critical section.
- Reentrant mutex locks.
- Semaphores to limit access to a critical section or resource.
- Condition variables (or monitor) to wait and notify.
- Barrier to executing and waiting for others.

And so on.

These primitives collectively are typically referred to as concurrent programming primitives or synchronization primitives.

- **Synchronization Primitives**: Algorithms and patterns for coordinating and synchronization in concurrent programs such as locks, barriers, semaphores, condition variables, and more.

Concurrent Programming Patterns

Standardized solutions to common programming problems are often referred to as patterns.

For example, the most common collection of programming patterns are those used in object-oriented programming, referred to simply as "design patterns (https://en.wikipedia.org/wiki/Software_design_pattern)".

Concurrent programming also has patterns, mostly centered around how data is shared between concurrent tasks, such as message passing.

Examples include:

- Thread Pool
- Producer-consumer
- Pipeline
- Channel

These patterns may be referred to as "concurrency patterns (https://en.wikipedia.org/wiki/Concurrency_pattern)".

- **Concurrency patterns**: Standardized solutions to common problems in concurrent programming.

Sometimes it is not clear whether a solution is a pattern or a primitive, often they are both.

Examples include barriers, semaphores, and condition variables.

Concurrent Programming Errors

Concurrency primitives and patterns are required to avoid specific failure modes in concurrent programming.

These failure modes are errors or bugs unique to the discipline of concurrent programming and occur when insufficiently protecting code from concurrent access, e.g. code that is not "safe".

Examples include:

- Race conditions that corrupt data or result in unexpected behavior.
- Deadlocks that halt the program.
- Livelocks that prevent tasks from progression

And so on.

These bugs collectively may be referred to as concurrency failure modes.

- **Concurrency Failure Modes**: Bugs specific to concurrent programs, such as race conditions, deadlocks, and livelocks.

Concurrency is often confused with parallelism, for good reason. The definitions of the terms are very close.

Next, let's disentangle concurrency from parallelism when describing programming tasks.

Concurrency is not Parallelism

Parallel means simultaneous, which sounds a lot like the definition of concurrent above.

> *parallel: an arrangement or state that permits several operations or tasks to be performed simultaneously rather than consecutively*
>
> — MERRIAM-WEBSTER DICTIONARY (HTTPS://WWW.MERRIAM-WEBSTER.COM/DICTIONARY/ASYNCHRONOUS)

In programming, parallel (https://en.wikipedia.org/wiki/Parallel_computing) has a slightly different meaning.

Parallel tasks are executed strictly at the same time.

Concurrent tasks may execute at the same time, or not. Their order is not prescribed.

> *While concurrency implies that multiple tasks are in process simultaneously, it does not imply that they are running together in parallel.*
>
> — PAGE 5, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3CZ7ZH6), 2022.

Parallel tasks are concurrent tasks that specify that they are to be completed at the same time.

> *When we say something is running in parallel, we mean not only are there two or more tasks happening concurrently, but they are also executing at the same time.*
>
> — PAGE 5, PYTHON CONCURRENCY WITH ASYNCIO (HTTPS://AMZN.TO/3CZ7ZH6), 2022.

Therefore, concurrency is a precondition for parallelism.

- Concurrent tasks may parallel tasks.
- Parallel tasks are concurrent tasks.

> *A system is said to be concurrent if it can support two or more actions in progress at the same time. A system is said to be parallel if it can support two or more actions executing simultaneously. The key concept and difference between these definitions is the phrase "in progress."*
>
> — PAGE 1, THE ART OF CONCURRENCY (HTTPS://AMZN.TO/3TKCUWX), 2009.

Let's pick these two terms apart further with an example

We may have two tasks to perform an elaborate calculation with many steps.

They are independent tasks and may be completed concurrently.

For example:

- We may complete task1, then complete task2 (sequential).
- We may do a little of task1, all of task2, then the rest of task1 (overlapping).
- We may do a little of task1, a little of task2, a little of task1, and so on (interleaved).
- We may perform task1 and task 2 at the same time (parallel).

The tasks are examples of completing the tasks concurrently. They are happening at the same time, although the caller is not interested in how they are being performed.

We may choose to execute the tasks in parallel.

This is more perspective. It assumes the tasks are concurrent and now we are specifying the way in which the concurrent tasks are to be completed.

- We perform task1 and task2 at the same time (parallel).

Strictly, it requires specialized system hardware, e.g. one CPU core per task.

> *Parallel: Executing more than one computation at the same time. These computations must be on separate cores to be running in parallel.*
>
> **— PAGE 270, THE ART OF CONCURRENCY (HTTPS://AMZN.TO/3TKCUWX), 2009.**

Anything less than executing the two tasks at the same time would not be parallel, it would be concurrent.

This may happen if we request parallel execution of two tasks on a system that only has a single CPU core, or has multiple CPU cores, both most are busy on other tasks, and so on.

Don't We Always Want Parallelism?

No.

There are many cases where we want concurrency and do not or can not achieve strict parallelism.

For example, we may want to read two files on one hard drive concurrently.

The hard drive is limited in reading data for one file at a time. But there is a large gap between requesting a hard drive read from the operating system and getting the bytes.

The gap could be a few to tens of milliseconds. Recall there are 1,000 milliseconds in one second.

Further, the file may be large and may require many read operations to get the bytes from the disk into the main memory.

During this time, thousands or millions of CPU instructions could be executed.

Rather than having the CPU wait around for the request to be fulfilled, we can release the CPU in our program and have it execute some other tasks, like requesting the next file to read or process the bytes read from the last file requested.

We can perform many of these I/O-type tasks concurrently and achieve a dramatic speed-up compared to performing the tasks sequentially.

This speed-up can be achieved without strict parallelism, but instead by using some form of interleaving of the tasks, using multitasking.

You can learn more about why we don't always want parallelism in the tutorial:

- Why Not Always Use Processes in Python (https://superfastpython.com/why-not-always-use-processes-in-python/)

Next, let's consider concurrent programming support in Python.

Overwhelmed by the python concurrency APIs?
Find relief, download my FREE Python Concurrency Mind Maps (https://marvelous-writer-6152.ck.page/8f23adb076)

Concurrent Programming in Python

Concurrent programming in Python refers to executing two or more tasks at the same time.

More broadly, concurrent programming in Python provides a suite of synchronization primitives such as mutex locks and semaphores and safe data structures like queues.

As such, Python provides three main types of concurrent programming, each centered around a different unit of concurrency and each in a separate Python module.

They are:

- Python multiprocessing for process-based concurrency.
- Python threading for thread-based concurrency.
- Python asyncio for coroutine-based concurrency.

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

Process-Based Concurrency

A process (https://en.wikipedia.org/wiki/Process_(computing)) refers to a computer program.

Every Python program is a process and has one thread called the main thread used to execute your program instructions. Each process is in fact one instance of the Python interpreter that executes Python instructions (Python byte-code), which is a slightly lower level than the code you type into your Python program.

We can create new processes to run concurrently alongside our Python program.

Python provides real system-level processes via the **multiprocessing.Process** class (https://docs.python.org/3/library/multiprocessing.html).

The underlying operating system controls how new processes are created. On some systems that may require spawning a new process and on others it may require that the process is forked. The operating-specific method used for creating new processes in Python is not something we need to worry about as it is managed by your installed Python interpreter.

A task can be run in a new process by creating an instance of the **Process** class and specifying the function to run in the new process via the "**target**" argument.

```
...
# define a task to run in a new process
process = Process(target=task)
```

Once the process is created, it must be started by calling the **start()** function.

```
...
# start the task in a new process
process.start()
```

We can then wait around for the task to complete by joining the process with the **join()** function, for example:

```
...
# wait for the task to complete
process.join()
```

Whenever we create new processes, we must protect the entry point of the program.

```
# entry point for the program
if __name__ == '__main__':
    # do things...
```

Tying this together, the complete example of creating a Process to run an ad hoc task function is listed below.

```
# SuperFastPython.com
# example of running a function in a new process
from time import sleep
from multiprocessing import Process

# a simple task that blocks for a moment and prints a message
def task():
    # block for a moment
    sleep(1)
    # display a message
    print('This is coming from another process', flush=True)

# entry point for the program
if __name__ == '__main__':
    # define a task to run in a new process
    process = Process(target=task)
    # start the task in a new process
    process.start()
    # display a message
    print('Waiting for the new process to finish...')
    # wait for the task to complete
    process.join()
```

Running the example creates the process object to run the **task()** function.

The process is started and the **task()** function is executed in the child process. The task sleeps for a moment, meanwhile, in the main thread in the parent process, a message is printed that we are waiting around and the main process joins the new child process.

Finally, the child process finishes sleeping, prints a message, and closes. The main thread in the parent process then carries on and also closes as there are no more instructions to execute.

```
Waiting for the new process to finish...
This is coming from another process
```

The multiprocessing module provides a suite of synchronization primitives for developing concurrent programs with processes, including:

- Mutual exclusion locks in the **multiprocessing.Lock** class (https://superfastpython.com/multiprocessing-mutex-lock-in-python/).
- Reentrant mutex locks in the **multiprocessing.RLock** class (https://superfastpython.com/multiprocessing-rlock-in-python/).
- Condition variables in the **multiprocessing.Conditio**n class (https://superfastpython.com/multiprocessing-condition-variable-in-python/).
- Semaphores in the **multiprocessing.Semaphore** class (https://superfastpython.com/multiprocessing-semaphore-in-python/).
- Process-safe events in the **multiprocessing.Event** class (https://superfastpython.com/multiprocessing-event-object-in-python/).
- Barriers in the **multiprocessing.Barrier** class (https://superfastpython.com/multiprocessing-barrier-in-python/).

And more.

Processes do not have shared memory, instead, data is transmitted between processes using inter-process communication.

This adds a computational cost to sharing data between processes and also means that some objects cannot be shared directly or easily.

The multiprocessing module provides many conveniences for sharing data between processes.

The first is provides process-safe pipe and queue data structures for sharing data between processes, such as:

- Pipe in the **multiprocessing.Pipe** class (https://superfastpython.com/multiprocessing-pipe-in-python/).
- FIFO queue in the **multiprocessing.Queue** class (https://superfastpython.com/multiprocessing-queue-in-python/).
- Simplified FIFO queue in the **multiprocessing.SimpleQueue** class (https://superfastpython.com/multiprocessing-simplequeue-in-python/).
- Joinable FIFO queue in the **multiprocessing.JoinableQueue** class (https://superfastpython.com/multiprocessing-joinablequeue-on-python/).

The multiprocessing module also provides managers that create a server process to host Python objects and proxy objects for the hosted objects that can be shared among processes easily.

To learn more about managers, see the tutorial:

- What is a Multiprocessing Manager (https://superfastpython.com/multiprocessing-manager/)

It is common to need to execute many tasks concurrently using processes.

Rather than creating a new process for every task, worker processes can be reused to execute ad hoc tasks. This is a programming pattern called a process pool.

Two process pools are provided in the Python standard library, including:

- Classical process pools in the **multiprocessing.pool.Pool** class (https://superfastpython.com/multiprocessing-pool-python/).
- Modern executor process pools in the **concurrent.futures.ProcessPoolExecutor** class (https://superfastpython.com/processpoolexecutor-in-python/).

You can learn more about process-based concurrency in the tutorial:

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

Thread-Based Concurrency

A thread (https://en.wikipedia.org/wiki/Thread_(computing)) refers to a thread of execution by a computer program.

Every Python program is a process with one thread called the main thread used to execute your program instructions. We can create new threads to run concurrently within our Python program.

Python provides real or native (system-level) threads via the **threading.Thread** class.

A task can be run in a new thread by creating an instance of the **Thread** class and specifying the function to run in the new thread via the target argument.

```
...
# create and configure a new thread to run a function
thread = Thread(target=task)
```

Once the thread is created, it must be started by calling the **start()** function.

```
...
# start the task in a new thread
thread.start()
```

We can then wait around for the task to complete by joining the thread by calling the **join()** function, for example:

```
...
# wait for the task to complete
thread.join()
```

We can demonstrate this with a complete example of a task that sleeps for a moment and prints a message.

The complete example of executing a target task function in a separate thread is listed below.

```
# SuperFastPython.com
# example of executing a target task function in a separate thread
from time import sleep
from threading import Thread

# a simple task that blocks for a moment and prints a message
def task():
    # block for a moment
    sleep(1)
    # display a message
    print('This is coming from another thread')

# create and configure a new thread to run a function
thread = Thread(target=task)
# start the task in a new thread
thread.start()
# display a message
print('Waiting for the new thread to finish...')
# wait for the task to complete
thread.join()
```

Running the example creates the thread object to run the **task()** function.

The thread is started and the **task()** function is executed in another thread. The task sleeps for a moment, meanwhile, in the main thread, a message is printed that we are waiting around and the main thread joins the new thread.

Finally, the new thread finishes sleeping, prints a message and closes. The main thread then carries on and also closes as there are no more instructions to execute.

```
Waiting for the new thread to finish...
This is coming from another thread
```

The threading module provides a suite of synchronization primitives for developing concurrent programs with threads, including:

- Mutual exclusion locks in the **threading.Lock** class (https://superfastpython.com/thread-mutex-lock/).
- Reentrant mutex locks in the **threading.RLock** class (https://superfastpython.com/thread-reentrant-lock/).
- Condition variables in the **threading.Condition** class (https://superfastpython.com/thread-condition/).
- Semaphores in the **threading.Semaphore** class (https://superfastpython.com/thread-semaphore/).
- Thread-safe events in the **threading.Event** class (https://superfastpython.com/thread-event-object-in-python/).
- Barriers in the **threading.Barrier** class (https://superfastpython.com/thread-barrier-in-python/).

And more.

It also provides thread-safe queue data structures in the "**queue**" module, such as:

- FIFO queue in the **queue.Queue** class (https://superfastpython.com/thread-queue/).
- Simplified FIFO queue in the **queue.SimpleQueue** class (https://superfastpython.com/thread-safe-simplequeue-in-python/).
- LIFO queue in the **queue.LifoQueue** class (https://superfastpython.com/thread-lifoqueue/).
- Priority queue in the **queue.PriorityQueue** class (https://superfastpython.com/thread-priority-queue/).

The reference Python interpreter CPython prevents more than one thread from executing bytecode at the same time.

This is achieved using a mutex called the Global Interpreter Lock or GIL, as we learned in the previous section.

There are times when the lock is released by the interpreter and we can achieve parallel execution of our concurrent code in Python.

Examples of when the lock is released include:

- When a thread is performing blocking IO.
- When a thread is executing C code and explicitly releases the lock.

As such, threads are appropriate for I/O-bound tasks, and not CPU-bound tasks.

Examples of blocking IO operations 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.
- Reading or writing bytes on a socket connection with a server.
- Downloading or uploading a file.
- Querying a server.
- Querying a database.
- Taking a photo or recording a video.
- And so much more.

It is common to need to execute many tasks concurrently using threads.

Rather than creating a new thread for every task, worker threads can be reused to execute ad hoc tasks. This is a programming pattern called a thread pool.

Two thread pools are provided in the Python standard library, including:

- Classical thread pools in the **multiprocessing.pool.ThreadPool** class (https://superfastpython.com/threadpool-class-in-python/).
- Modern executor thread pools in the **concurrent.futures.ThreadPoolExecutor** class (https://superfastpython.com/threadpoolexecutor-in-python/).

You can learn more about thread-based concurrency in the guide:

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

Coroutine-Based Concurrency

A coroutine (https://en.wikipedia.org/wiki/Coroutine) is a unit of concurrency that is more lightweight than a thread.

A single thread may execute many coroutines in an event loop.

Unlike threads and processes where the operating system controls when a thread or process is suspended and when it is resumed and executed, coroutines themselves control when they are suspended and resumed.

This is called cooperating multitasking.

Python provides coroutines as part of the language. It also utilities for managing coroutines via the "**asyncio**" module with a specific focus on non-blocking I/O with sockets and subprocesses.

Coroutines are defined and used via the async/await syntax in Python.

Coroutines are defined using the "**async**" expression, and running coroutines can execute and wait on coroutines using the "**await**" expression.

A coroutine can be executed by starting the asyncio event loop and passing the coroutine for execution.

This can be achieved by calling the asyncio.run() function and pass it the coroutine instance.

The example below defines a custom coroutine that takes a message and prints it. The coroutine is then created and passed to the **asyncio.run()** function for execution.

```
# SuperFastPython.com
# example of executing a coroutine using the event loop
import asyncio

# custom coroutine
async def custom_coro(message):
    # report the message
    print(message)

# create and execute coroutine
asyncio.run(custom_coro('Hi from a coroutine'))
```

Running the example creates the coroutine and passes it a message to report.

The coroutine is passed to the **asyncio.run()** function that starts the asyncio event loop and executes the coroutine.

The message is then reported from the coroutine and the asyncio event loop is shut down.

This highlights how we can run a coroutine directly using the asyncio event loop.

```
Hi from a coroutine
```

Coroutines themselves can await coroutines using the await expression.

Coroutines may also schedule tasks for later execution via the **asyncio.create_task()** function.

The asyncio module provides a suite of synchronization primitives for developing concurrent programs with coroutines, including:

- Mutual exclusion locks in the **asyncio.Lock** class (/asyncio-lock).
- Condition variables in the **asyncio.Condition** class (/asyncio-condition-variable).
- Semaphores in the **asyncio.Semaphore** class (/asyncio-semaphore).
- Events in the **asyncio.Event** class (/asyncio-event).

It also provides coroutine-safe queue data structures, such as:

- FIFO queue in the **asyncio.Queue** class (/asyncio-queue).
- LIFO queue in the **asyncio.LifoQueue** class (/asyncio-lifoqueue).
- Priority queue in the **asyncio.PriorityQueue** class (/asyncio-priorityqueue).

Takeaways

You now know about concurrent programming in Python.

Do you have any questions?
Ask your questions in the comments below and I will do my best to answer.

Photo by Yash Sonawale (https://unsplash.com/@curlyhairkid?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText) on Unsplash (https://unsplash.com/s/photos/yellow-car?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText)

About Jason Brownlee

Hi, my name is Jason Brownlee, Ph.D. and I'm the guy behind this website. I am obsessed with Python Concurrency.

I help python developers learn concurrency, super fast.
Learn more (/about).

Parallel Loops in Python

Discover how to run your loops in parallel, download your free book (https://marvelous-writer-6152.ck.page/99ee689b9b) now:

(https://marvelous-writer-6152.ck.page/99ee689b9b)

Your free book "**Parallel Loops in Python**" includes complete and working code templates that you can modify and use right now in your own projects.

Download Your FREE Book (https://marvelous-writer-6152.ck.page/99ee689b9b)

Comments

Ahmed *says*
JANUARY 5, 2023 AT 9:31 PM (HTTPS://SUPERFASTPYTHON.COM/CONCURRENT-PROGRAMMING/#COMMENT-863)

My question is: How can I stop a process, or a thread after starting it

REPLY

> **JASON BROWNLEE (HTTPS://SUPERFASTPYTHON.COM)** *says*
> **JANUARY 6, 2023 AT 6:11 AM (HTTPS://SUPERFASTPYTHON.COM/CONCURRENT-PROGRAMMING/#COMMENT-869)**
>
> This tutorial shows you how to stop a thread:
> https://superfastpython.com/stop-a-thread-in-python/ (https://superfastpython.com/stop-a-thread-in-python/)
>
> This tutorial shows you how to stop a process:
> https://superfastpython.com/safely-stop-a-process-in-python/ (https://superfastpython.com/safely-stop-a-process-in-python/)
>
> **REPLY**

COPYRIGHT © 2024 SUPER FAST PYTHON
LINKEDIN (HTTPS://WWW.LINKEDIN.COM/COMPANY/SUPER-FAST-PYTHON/) | TWITTER (HTTPS://TWITTER.COM/SUPERFASTPYTHON) | FACEBOOK (HTTPS://WWW.FACEBOOK.COM/SUPERFASTPYTHON) | RSS (HTTPS://SUPERFASTPYTHON.COM/FEED/)

SUPER FAST PYTHON PTY. LTD.
PO BOX 206, VERMONT, VICTORIA, 3133, AUSTRALIA
ABN: 65 654 877 433, ACN: 654 877 433
EMAIL: JASON@SUPERFASTPYTHON.COM