SuperFastPython

7-Day Course

Python Threading Jump-Start

Develop Concurrent IO-bound Programs And Work With The GIL

Jason Brownlee

Python Threading Jump-Start

Develop Concurrent IO-bound Programs And Work With The GIL

Jason Brownlee

2022

Praise for *SuperFast*Python

"*I'm reading this article now, and it is really well made (simple, concise but comprehensive). Thank you for the effort! Tech industry is going forward thanks also by people like you that diffuse knowledge.*"

– **Gabriele Berselli**, Python Developer.

"*I enjoy your postings and intuitive writeups - keep up the good work*"

– **Martin Gay**, Quantitative Developer at Lacima Group.

"*Great work. I always enjoy reading your knowledge based articles*"

– **Janath Manohararaj**, Director of Engineering.

"*Great as always!!!*"

– **Jadranko Belusic**, Software Developer at Crossvallia.

"*Thank you for sharing your knowledge. Your tutorials are one of the best I've read in years. Unfortunately, most authors, try to prove how clever they are and fail to educate. Yours are very much different. I love the simplicity of the examples on which more complex scenarios can be built on, but, the most important aspect in my opinion, they are easy to understand. Thank you again for all the time and effort spent on creating these tutorials.*"

– **Marius Rusu**, Python Developer.

"*Thanks for putting out excellent content Jason Brownlee, tis much appreciated*"

– **Bilal B.**, Senior Data Engineer.

"*Thank you for sharing. I've learnt a lot from your tutorials, and, I am still doing, thank you so much again. I wish you all the best.*"

– **Sehaba Amine**, Research Intern at LIRIS.

"*Wish I had this tutorial 7 yrs ago when I did my first multithreading software. Awesome Jason*"

– **Leon Marusa**, Big Data Solutions Project Leader at Elektro Celje.

"*This is awesome*"

– **Subhayan Ghosh**, Azure Data Engineer at Mercedes-Benz R&D.

Copyright

Disclaimer

The information contained within this book is strictly for educational purposes. If you wish to apply ideas contained in this book, you are taking full responsibility for your actions.

The author has made every effort to ensure the accuracy of the information within this book was correct at time of publication. The author does not assume and hereby disclaims any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from accident, negligence, or any other cause.

Preface

Python concurrency is deeply misunderstood.

Opinions vary from "*Python does not support concurrency*" to "*Python concurrency is buggy*".

I created the website SuperFastPython.com to directly counter these misunderstandings.

Python threads are limited by the infamous Global Interpreter Lock (GIL). Because of this, most Python developers use processes and the `multiprocessing` module for all their concurrency needs.

Processes are heavyweight units of concurrency, some operating systems like Windows limit the number of child processes that can be created, and sharing data between processes is slow, requiring all data to be serialized and transmitted using inter-process communication.

This is a major problem when performing blocking IO tasks, such as reading and writing from files and sockets. These tasks are much better suited to threads.

The GIL is released when performing blocking-IO, allowing other threads to run. Threads can also share data directly, adding no computational overhead. Finally, we can create as many threads as our system can support, allowing 100s or 1,000s of concurrent database connections or files to be downloaded at once.

Python threads have an important and often overlooked place in any program performing blocking IO.

This guide was carefully designed to help Python developers (*like you*) to get productive with the `threading` module as fast as possible. After completing all seven lessons, you will know how to bring process-based concurrency with the `threading` module API to your own projects, super fast.

Together, we can make Python code run faster and change the community's opinions about Python concurrency.

Thank you for letting me guide you along this path.

Jason Brownlee, Ph.D.
SuperFastPython.com
2022.

Contents

Introduction

The Python `threading` module allows us to execute IO-bound tasks concurrently with Python.

Thread-based concurrency provided by the `threading` module. Although Python threads are limited by the Global Interpreter Lock (GIL), they remain the best approach to executing hundreds or thousands of concurrent IO-bound tasks, such as reading or writing from files or socket connections.

This book provides a jump-start guide to get you productive with the Python `threading` module in 7 days.

It is not a dry, long-winded academic textbook. Instead, it is a crash course for Python developers that provides carefully designed lessons with complete and working code examples that you can copy-paste into your project today and get results.

Before we dive into the lessons, let's take a look at what is coming with a breakdown of this book.

Who Is This Course For?

Before we dive into the course, let's make sure you're in the right place.

This course is designed for Python developers who want to discover how to use and get the most out of the `threading` module to write fast programs.

Specifically, this course is for:

- Developers that can write simple Python programs.
- Developers that need better performance from current or future Python programs.
- Developers that are working with IO-bound tasks.

This course does not require that you are an expert in the Python programming language or concurrency.

Specifically:

- You do not need to be an expert Python developer.
- You do not need to be an expert in concurrency.

Next, let's take a look at what this course will cover.

7-Day Course Overview

This course is designed to bring you up-to-speed with how to use the `threading` module as fast as possible.

As such, it is not exhaustive. There are many topics that are interesting or helpful, that are not on the critical path to getting you productive fast.

This course is divided into 7 lessons, they are:

- **Lesson 01**: Thread-Based Concurrency
- **Lesson 02**: Create and Start New Threads
- **Lesson 03**: Configuring and Interacting with Threads
- **Lesson 04**: Synchronize and Coordinate Threads
- **Lesson 05**: Share Data Between Threads
- **Lesson 06**: Run Tasks with Reusable Workers in Thread Pools
- **Lesson 07**: Close, Stop, and Kill Threads

Next, let's take a closer look at how lessons are structured.

Lesson Structure

Each lesson has two main parts, they are:

1. The body of the lesson.
2. The lesson overview.

The body of the lesson will introduce a topic with code examples, whereas the lesson overview will review what was learned with exercises and links for further information.

Each lesson has a specific learning outcome and is designed to be completed in less than one hour, although most lessons can be completed within about 30 minutes.

Each lesson is also designed to be self-contained so that you can read the lessons out of order if you choose, such as dipping into topics in the future to solve specific programming problems.

The lessons were written with some intentional repetition of key concepts. These gentle reminders are designed to help embed the common usage patterns in your mind so that they become second nature.

We Python developers learn best from real and working code examples.

Next, let's learn more about the code examples provided in the book.

Code Examples

All code examples use Python 3.

Python 2.7 is not supported because it reached end of life in 2020.

I recommend the most recent version of Python 3 available at the time you are reading this, although Python 3.9 or higher is sufficient to run all code examples in this book.

You do not require any specific integrated development environment (IDE). I recommend typing code into a simple text editor like Sublime Text or Atom that run on all modern operating systems. I'm a Sublime user myself, but any text editor will do. If you are familiar with an IDE, then by all means, use it.

Each code example is complete and can be run as a standalone program. I recommend running code examples from the command line (also called the command prompt on Windows or terminal on MacOS) to avoid any possible issues.

To run a Python script from the command line:

1. Save the code file to a directory of your choice with a `.py` extension.
2. Open your command line (also called the command prompt or terminal).
3. Change directory to the location where you saved the Python script.
4. Execute the script using the Python interpreter followed by the name of the script.

For example:

```
python my_script.py
```

I recommend running scripts on the command line because it is easy, it works for everyone, it avoids all kinds of problems that beginners have with notebooks and IDEs, and because scripts run fastest on the command line.

That being said, if you know what you're doing, you can run code examples within your IDE or a notebook if you like. Editors like Sublime Text and Atom will let you run Python scripts directly, and this is fine. I just can't help you debug any issues you might encounter because they're probably caused by your development environment.

All lessons in this book provide code examples. These are typically introduced first via snippets of code that begin with an ellipsis (`...`) to clearly indicate that they are not a complete code example. After the program is introduced via snippets, a complete code example is always listed that includes all of the snippets tied together, with any additional glue code and import statements.

I recommend typing code examples from scratch to help you learn and memorize the API.

Beware of copy-pasting code from the EBook version of this book as you may accidentally lose or add white space, which may break the execution of the script.

A code file is provided for each complete example in the book organized by lesson and example within each lesson. You can execute these scripts directly or use them as a reference.

You can download all code examples from here:

- Download Code Examples
  https://SuperFastPython.com/ptj-code

All code examples were tested on a POSIX machine by myself and my technical editors prior to publication.

APIs can change over time, functions can become deprecated, and idioms can change and be replaced. I keep this book up to date with changes to the Python standard library and you can email me any time to get the latest version. Nevertheless, if you encounter any warnings or problems with the code, please contact me immediately and I will fix it. I pride myself on having complete and working code examples in all of my lessons.

Next, let's learn more about the exercises provided in each lesson.

Practice Exercises

Each lesson has an exercise.

The exercises are carefully designed to test that you understood the learning outcome of the lesson.

I strongly recommend completing the exercise in each lesson to cement your understanding.

NOTE: I recommend sharing your results for each exercise publicly.

This can be done easily using social media sites like Twitter, Facebook, and LinkedIn, on a personal blog, or in a GitHub project. Include the name of this book or SuperFastPython.com to give context to your followers.

I recommend sharing your answers to exercises for three reasons:

- It will improve your understanding of the topic of the lesson.
- It will keep you accountable, ensuring you complete the lesson to a high standard.
- I'd love to see what you come up with!

You can email me the link to each exercise directly via:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

Or share it with me on Twitter via @SuperFastPython.

Next, let's consider how we might approach working through this book.

How to Read

You can work at your own pace.

There's no rush and I recommend that you take your time.

This book is designed to be read linearly from start to finish, guiding you from being a Python developer at the start of the book to being a Python developer that can confidently use the `threading` module in your project by the end of the book.

In order to avoid overload, I recommend completing one or two lessons per day, such as in the evening or during your lunch break. This will allow you to complete the transformation in about one week.

I recommend maintaining a directory with all of the code you type from the lessons in the book. This will allow you to use the directory as your own private code library, allowing you to copy-paste code into your projects in the future.

I recommend trying to adapt and extend the examples in the lessons. Play with them. Break them. This will help you learn more about how the API works and why we follow specific usage patterns.

Next, let's review your new found capabilities after completing this book.

Learning Outcomes

This book will transform you from a Python developer into a Python developer that can confidently bring concurrency to your projects with the `threading` module.

After working through all of the lessons in this book, you will know:

- The difference between thread-based and process-based concurrency and the types of tasks that are well suited to the capabilities of the `threading` module.
- How to execute your own ad hoc functions concurrently using the `Thread` class.
- How to identify the main thread and the life-cycle of new threads.
- How to configure a new thread and get access to all running threads and query their status.
- How to coordinate and synchronize threads using mutex locks, semaphores, condition variables and the full suite of concurrency primitives.
- How to share data between threads using shared global variables and how to return values from new threads using instance variables.
- How to send and receive data between threads using thread-safe queues.
- How to create and configure thread pools to execute ad hoc tasks using reusable worker threads.
- How to handle results and errors and query the status of asynchronous tasks executed in thread pools.
- How to close threads, and how to trigger a graceful shutdown or forcefully kill new threads.

Next, let's discover how we can get help when working through the book.

Getting Help

The lessons were designed to be easy to read and follow.

Nevertheless, sometimes we need a little extra help.

A list of further reading resources is provided at the end of each lesson. These can be helpful if you are interested in learning more about the topic covered, such as fine grained details of the standard library and API functions used.

The conclusions at the end of the book provide a complete list of websites and books that can help if you want to learn more about Python concurrency and the relevant parts of the Python standard library. It also lists places where you can go online and ask questions about Python concurrency.

Finally, if you ever have questions about the lessons or code in this book, you can contact me any time and I will do my best to help. My contact details are provided at the end of the book.

Now that we know what's coming, let's get started.

Next

Next up in the first lesson, we will discover thread-based concurrency in Python.

Lesson 01: Thread-Based Concurrency

In this lesson, we will explore thread-based concurrency vs process-based concurrency and the sweet-spot for using the `threading` module.

After completing this lesson, you will know:

- What a thread is, how it compares to a process, and the difference between concurrency and parallelism.
- About threading in Python and the key limitations of thread-based concurrency.
- About the Global Interpreter Lock and how it impacts Python threads.
- About multiprocessing in Python and the key limitations of process-based concurrency.
- When to use the thread-based concurrency in the `threading` module in your programs.

Let's get started.

Python Concurrency

Before we dive into the details, let's make sure we are on the same page when it comes to *threads*, *processes*, *concurrency*, and *parallelism*.

- A **process** is a computer program.
- A **thread** refers to a *thread of execution* within a computer program.

Therefore, each program is a process that has at least one thread that executes instructions for that process.

- **Concurrency** refers to executing tasks out of order.
- **Parallelism** refers to executing tasks simultaneously.

When we are developing code, we can achieve concurrency with or without parallelism, although concurrency (e.g. task order being irrelevant) is a prerequisite for parallelism.

Our goal is to speed-up a program by executing two or more tasks simultaneously. We are almost always interested in parallelism when we talk about Python concurrency.

Overview of Python Threads and Processes

Python has support for both threads and processes.

That is, it provides classes that allow us to create and interact with threads and processes that are managed by the underlying operating system.

Thread-based concurrency is provided via the `threading` module and the `Thread` class. Threads are fast to start and can share data with each other within a single Python process.

Nevertheless, thread-based concurrency in Python is limited by the Global Interpreter Lock (GIL). Threads are only capable of parallelism when the GIL is released, such as when performing blocking I/O operations or explicitly by third party libraries.

Process-based concurrency is provided via the `multiprocessing` module and the `Process` class.

It was developed in Python v2.6 as an alternative to thread-based concurrency that is not limited by the GIL.

Processes are a heavyweight approach to concurrency as processes are slower to start than threads. Unlike threads, they are subject to the cost of having to serialize (pickle) data in order to transmit it between processes. This can add some computational overhead proportional to the amount of data that is shared between processes.

Nevertheless, the `multiprocessing` module is capable of concurrency and full parallelism in Python.

We now have a birds-eye view of threading and multiprocessing in Python.

Next, let's take a closer look at the `threading` and `multiprocessing` modules.

What is Threading in Python

The `threading` module provides thread-based concurrency in Python.

Technically, it is implemented on top of another lower-level module called `_thread`.

As we saw above, a thread refers to a thread of execution in a computer program.

Each program is a process and has at least one thread that executes instructions for that process.

Threading API

Central to the `threading` module is the `Thread` class that provides a Python handle on a native thread (managed by the underlying operating system).

A function can be run in a new thread by creating an instance of the `Thread` class and specifying the function to run via the `target` argument. The `start()` method can then be called which will execute the target function in a new thread of execution.

The rest of the `threading` module provides tools to work with `Thread` instances.

This includes a number of static module functions that allow the caller to get a count of the number of active threads, get access to a `Thread` instance for the current thread, enumerate all active threads in the process and so on.

There is also a `local` API that provides access to thread-local data storage, a facility that allows threads to have private data not accessible to other threads.

Finally, the `threading` API provides a suite of concurrency primitives for synchronizing and coordinating threads.

This includes mutex locks in the `Lock` and `RLock` classes, semaphores in the `Semaphore` class, thread-safe boolean variables in the `Event` class, a thread with a delayed start in the `Timer` class and finally a barrier pattern in the `Barrier` class.

Threading Limitations

The API of the `Thread` class and the concurrency primitives were inspired by the Java concurrency API, such as the `java.lang.Thread` class and later the `java.util.concurrent` API.

In fact, the API originally had many camel-case function names, like those in Java, that were later changed to be more Python compliant names.

A key limitation of `Thread` for thread-based concurrency is that it is subject to the Global Interpreter Lock (GIL). This means that only one thread can run at a time in a Python process, unless the GIL is released, such as during blocking I/O or explicitly in third-party libraries.

This limitation means that although threads can achieve concurrency (executing tasks out of order), it can only achieve parallelism (executing tasks simultaneously) under specific circumstances.

NOTE: This was a lot to take in. Don't worry, we will look at each of these features of the `threading` API in coming lessons. By the end, we will know how to use them all in our program.

Next, let's take a closer look at the GIL and why it matters for thread-based concurrency.

What is the Global Interpreter Lock

The internals of the Python interpreter are not thread-safe.

This means that there can be race conditions between multiple threads within a single Python process, potentially resulting in unexpected behavior and corrupt data.

As such, the Python interpreter makes use of a Global Interpreter Lock, or GIL for short, to make instructions executed by the Python interpreter (called Python bytecodes) thread-safe.

The GIL is a programming pattern in the reference Python interpreter called CPython, although similar locks exist in other interpreted languages, such as Ruby. It is a lock in the

sense that it uses a synchronization primitive called a mutual exclusion or a mutex lock to ensure that only one thread of execution can execute instructions at a time within a Python process.

The effect of the GIL is that whenever a thread within a Python program wants to run, it must acquire the lock before executing. This is not a problem for most Python programs that have a single thread of execution, called the main thread. It can become a problem in multi-threaded Python programs.

The lock is explicitly released and re-acquired periodically by each Python thread, specifically after approximately every 100 bytecode instructions executed within the interpreter. This allows other threads within the Python process to run, if present.

The lock is also released in some specific circumstances, allowing other threads to run.

An important example is when a thread performs a blocking I/O operation, such as reading or writing from an external resource like a file, socket, or device.

The lock is also explicitly released by some third-party Python libraries when performing computationally expensive operations in C-code, such as many array operations in NumPy.

The GIL is a simple and effective solution to thread safety in the Python interpreter, but it has the major downside that full thread-based parallelism is not supported by Python.

An alternative solution might be to explicitly make the interpreter thread-safe by protecting each critical section. This has been tried a number of times and typically results in worse performance of single-threaded Python programs by up to 30%.

Next, let's take a brief look at process-based concurrency so we can contrast it with thread-based concurrency later.

What is Multiprocessing in Python

The `multiprocessing` module provides process-based concurrency in Python.

Jesse Noller and Richard Oudkerk proposed and developed the `multiprocessing` module (originally called `pyprocessing`) in Python specifically to overcome the limitations and side-step the GIL seen in thread-based concurrency.

With this goal in mind the `multiprocessing` module attempted to replicate the `threading` module API, although implemented using processes instead of threads.

A process refers to a computer program.

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

Multiprocessing API

Central to the `multiprocessing` module is the `Process` class that provides a Python handle on a native process, managed by the underlying operating system.

Much of the rest of the `multiprocessing` module provides tools to work with `Process` objects, that mimics the `threading` module.

This includes a number of static module functions that allow the caller to get access to a `Process` objects for the current process, enumerate all active child processes and so on.

The `multiprocessing` API provides a suite of concurrency primitives for synchronizing and coordinating processes, as process-based counterparts to the threading concurrency primitives. This includes `Lock`, `RLock`, `Semaphore`, `Event`, `Condition`, and `Barrier`.

Process-safe queues are provided in `Queue`, `SimpleQueue` and so on that mimic the thread-safe queues provided in the `queue` module.

This is where the similarities end.

Multiprocessing Limitations

A key limitation of the `multiprocessing` module is sharing data between processes.

Unlike threads that have shared memory, processes must share data and variables using Inter-Process Communication (IPC).

This requires that data be serialized in order to be transmitted to another process, and deserialized once received.

Python-native serialization in the `pickle` module is used and performed automatically. Data is shared using pipes or queues.

This requires that all data share between processes can be pickled, and adds a computational overhead compared to the shared memory model used by threads.

Now that we are familiar with both the `threading` and `multiprocessing` modules at a high-level, let's consider when we should be using threads (and when we shouldn't).

When to Use Threads

The `threading` module provides powerful and flexible concurrency, although it is not suited for all situations.

In this section, we'll look at broad types of tasks and why they are or are not appropriate for threads.

Use Threads for IO-Bound Tasks

The sweet-spot for the `threading` module is for blocking 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 or I/O), 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. Performing IO is very slow compared to the speed of the CPU.

Interacting with devices, reading and writing files, and socket connections involve calling instructions in our operating system (the kernel), which will wait for the operation to complete. If this operation is the main focus for our CPU, such as executing in the main thread of our Python program, then our 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.

We can free-up the CPU from IO-bound operations by performing IO-bound operations on another thread of execution. This allows the CPU to start the task and pass it off to the operating system (kernel) to do the waiting and free it up to execute in another application thread.

Let's consider some examples of IO-bound tasks.

Examples of IO-Bound Tasks

Some examples of common 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.

Next, let's consider the types of tasks where threads would not be appropriate.

Don't Use Threads for CPU-Bound Tasks

You should probably not use the `threading` module for CPU-bound tasks.

A CPU-bound task is a type of task that involves performing a computation and does not involve IO, and where data needs to be shared between processes.

CPU-bound tasks typically only involve data in main memory (RAM) or cache 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.

Python threads are limited by the GIL, as discussed. Therefore, although we may have many threads started and running, only a single thread can acquire the GIL and execute a

CPU-bound task at a time.

This means performance of threads on CPU-bound tasks will probably be as good or more likely much worse than not using threads at all. Instead, we should consider using the `multiprocessing` module for CPU-bound tasks.

Next, let's consider some examples of CPU-bound tasks that are probably not appropriate for threads.

Examples of CPU-Bound Tasks

Some examples of CPU-bound tasks include:

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

Next, let's consider some exceptions where we might be able to execute CPU-bound tasks with threads and achieve full parallelism.

Exceptions to the Rule

Sometimes threads are appropriate for CPU-bound tasks.

One exception are those tasks that execute code in a third-party library that explicitly releases the GIL.

For example:

- When calculating cryptographic hashes using the `hashlib` library.
- When calculating matrix operations using the `NumPy` library from the SciPy suite.
- When compressing data or files (e.g. `zlib`) and when working with video and image data (e.g. `OpenCV`).

Therefore, before we discount threads because we are working with CPU-bound tasks, we should carefully review the APIs being used to see if they in fact release the GIL, making threads an appropriate solution.

Another consideration is when we are using an alternate Python interpreter to run our programs.

The GIL is present in the CPython interpreter that we download from python.org, but the GIL does not exist in some other Python interpreters.

Examples include: PyPy, IronPython, and Jython.

If any of these Python interpreters are used to execute the program, then Python threads can be used and achieve full parallelism for CPU-bound tasks.

If the `multiprocessing` module always achieves full concurrency by by-passing the GIL, why even bother with threads?

Why Not Always Use Multiprocessing

If the `threading` module is limited by the GIL and only achieves parallelism in limited situations, why not always use the `multiprocessing` module instead?

Why ever bother with threads?

This is an important question that requires a robust answer.

The reason is that thread-based concurrency has benefits over process-based concurrency.

Processes can be used for IO-bound tasks, although they suffer major limitations that make using `multiprocessing` generally inappropriate.

The limitations of `multiprocessing` for IO-bound tasks are:

1. **Heavyweight**: Processes are heavyweight structures making them slower to start and require more memory.
2. **IPC**: All data sent between processes must be serialized, adding a computational overhead proportional to the data shared.
3. **Limited Number**: The operating system imposes limits on the number of child processes we can create.

Let's take a closer look at each of these concerns.

Processes are Heavyweight

Perhaps the most important difference between the `threading` and `multiprocessing` modules is the type of concurrency that underlies them.

The focus of the `threading` module is a native thread managed by the operating system. The focus of the `multiprocessing` module is a native process managed by the underlying operating system.

A process is a higher-level of abstraction than a thread.

- A process has a main thread.
- A process may have additional threads.
- A process may have child processes.

A thread belongs to a process, whereas one process may have one or thousands of threads.

Threads are a lightweight construct.

- Threads have a small memory footprint.
- Threads are fast to allocate and create.
- Threads are fast to start.

Processes are a heavyweight construct.

- Processes have a larger memory footprint, e.g. a process is an instance of the Python interpreter.
- Processes are slow to allocate and create, e.g. fork or spawn start methods are used.
- Processes are slow to start, e.g. the main thread must be created and started.

This means creating, starting, and managing thousands of concurrent tasks, such as requests in a server is well suited to threads and not process-based concurrency.

Processes are Slow to Share Data

Concurrency typically requires sharing data or program state between tasks.

Threads and processes have important differences in the way they access shared state.

Threads can share memory within a process.

This means that functions executed in new threads can access the same data. These might be global variables or data shared via function arguments. As such, sharing state between threads is straightforward and very fast.

Processes do not have shared memory like threads.

Instead, state must be serialized and transmitted between processes, called inter-process communication. Although it occurs under the covers, it does impose limitations on what data and state can be shared and adds overhead to sharing data.

Often sharing data between processes requires explicit mechanisms, such as the use of a `Pipe` or a `Queue`. The computational overhead imposed in serializing and deserializing data for transmission in negligible for small programs, but becomes overwhelming when large amounts of data must be shared, such as:

- The result set from a database query.
- A resource downloaded from the internet.
- The content of a file loaded from disk.

The Number of Processes is Limited

The operating system may impose limits on the total number of processes supported by the operating system, or the total number of child processes that can be created by a process.

For example, the limit in Windows is 61 child processes.

When performing tasks with IO, we may require hundreds or even thousands of concurrent workers. For example, each worker may manage a single network connection or file handle. This may not be feasible or possible with processes.

The number of threads that can be created is not directly limited, other than the capacity of main memory and the activity required within each thread.

We can easily create hundreds or thousands of threads in our Python programs. This makes threads appropriate for IO-bound tasks and discounts process-based concurrency for anything other than small programs.

Summary of Differences

It may help to summarize and contrast the differences between the `threading` and `multiprocessing` modules.

Property	**`threading`**	**`multiprocessing`**
Type	Uses native threads.	Uses native processes.
Relation	Belongs to a process.	Has threads and children.
Sharing	Shared memory.	Inter-process comms.
Weight	Light, fast to start.	Heavy, slow to start.
Parallelism	Limited (GIL).	Full (no GIL).
Tasks	IO-bound tasks.	CPU-bound tasks.
Number	10s to 1,000s.	10s (or fewer).

Put bluntly, we can achieve parallelism with Python threads when performing IO-bound tasks, and we can use shared memory to move data around.

In practice, most of the concurrency we require will involve IO of some kind. It is a rare task that is purely CPU bound.

This is why we must know how to use thread-based concurrency.

Lesson Review

Takeaways

Well done, you made it to the end of the lesson.

- You know what a thread is, how it compares to a process, and the difference between concurrency and parallelism.
- You know about threading in Python and the key limitations of thread-based concurrency.
- You know about the Global Interpreter Lock and how it impacts Python threads.
- You know about multiprocessing in Python and the key limitations of process-based concurrency.
- You know when to use the thread-based concurrency in the `threading` module in your programs.

Exercise

Your task for this lesson is to take what you have learned about the `threading` module and think about where you might be able to use it to improve the performance of your programs.

List at least three examples of programs you have worked on recently that could benefit from the concurrency provided by the `threading` module.

No need to share sensitive details of the project or technical details on how exactly threading might be used, just a one or two line high-level description is sufficient.

If you have trouble coming up with examples of recent applications that may benefit from using the `threading` module, then think of applications or functionality you could develop for current or future projects that could make good use of threads.

This is a useful exercise, as it will start to train your brain to see when you can and cannot make use of these techniques in practice.

Share your results online on Twitter, LinkedIn, GitHub, or similar.

Send me the link to your results, I'd love to see what you come up with.

You can send me a message directly via:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

Or share it with me on Twitter via @SuperFastPython.

Further Reading

This section provides resources for you to learn more about the topics covered in this lesson.

- Process (computing), Wikipedia.
  https://en.wikipedia.org/wiki/Process_(computing)
- Thread (computing), Wikipedia.
  https://en.wikipedia.org/wiki/Thread_(computing)
- Inter-process communication, Wikipedia.
  https://en.wikipedia.org/wiki/Inter-process_communication
- Global interpreter lock, Wikipedia.
  https://en.wikipedia.org/wiki/Global_interpreter_lock
- `threading` - Thread-based parallelism.
  https://docs.python.org/3/library/threading.html
- `multiprocessing` - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html
- `pickle` - Python object serialization.
  https://docs.python.org/3/library/pickle.html
- Python Global Interpreter Lock, Python Wiki.
  https://wiki.python.org/moin/GlobalInterpreterLock

Next

In the next lesson, we will discover how we can create and start new threads in Python to execute ad hoc code.

Lesson 02: Create and Start New Threads

In this lesson, you will discover how to create and start new threads using the `threading` module.

After completing this lesson, you will know:

- About the main thread, helper threads and new threads.
- About the life-cycle of the `Thread` class in Python.
- How to run a custom function in a new `Thread` instance.
- How to extend the `Thread` class to run custom code in a new thread.
- How to use thread-local storage that is private to each thread.

Let's get started.

Main Thread, Helper Threads, and New Threads

A Python program is a new process with one default thread.

When we run a Python script, it starts a process that is an instance of the Python interpreter. This process has the name *MainProcess*.

The *MainProcess* will then start one thread to execute our Python code. This thread is called the *MainThread*, and is the default thread.

- **Main Thread**: Default thread created by a main process in a Python program, has the name *MainThread*.

When we talk about the *MainThread* or *main thread*, we are referring to the thread in the main process that is executing the instructions in our program, as opposed to other threads that may be running in the program.

There may be other threads in our program, created by the standard library or third-party libraries. These are typically referred to as helper threads and perform background tasks to assist a primary task.

- **Helper Threads**: Background threads created by library functions or classes to assist a primary activity.

We may create new threads in our program to execute tasks in our program directly.

We might refer to these as *new threads* to differentiate them from the main thread and helper threads not created by us

- **New Threads**: New threads created directly in our program.

Finally, we may create a pool of reusable worker threads, called a thread pool.

We do not create the threads in the pool directly, they are created for us by the pool. They are a specific type of helper thread commonly referred to as *thread workers*, *worker threads* or simply *workers*.

- **Worker Threads**: A type of helper thread created and managed by a thread pool.

Next, let's take a closer look at the life-cycle of a thread.

Life-Cycle of a Thread

A thread in Python is represented as an instance of the `Thread` class.

Once a thread is started, the Python interpreter will interface with the underlying operating system and request that a new native thread be created and started. The `Thread` object then provides a Python-based reference to this underlying native thread.

Each thread follows the same life-cycle. Understanding the stages of this life-cycle can help when getting started with concurrent programming in Python.

For example:

- The difference between creating and starting a thread.
- The difference between run and start.
- The difference between blocked and terminated.

A Python thread may progress through three steps of its life-cycle: a new thread, a running thread, and a terminated thread.

While running, the thread may be executing code or may be blocked, waiting on something such as another thread or an external resource. Although, not all threads may block, it is optional based on the specific use case for the new thread.

1. New Thread.
2. Running Thread.
    1. Blocked Thread (optional).
3. Terminated Thread.

A new thread is a thread that has been constructed by creating a `Thread` object. It is in the *new* state, and is not yet running.

A new thread can transition to a running thread by calling the `start()` method.

A running thread may block in many ways, such as reading or writing from a file or a socket or by waiting on a concurrency primitive such as a semaphore or a lock. After blocking, the thread will run again.

Next, let's look at an example of running a custom function concurrently in a new thread.

How to Run a Function in a New `Thread`

Python functions can be executed in a separate thread using the `Thread` class.

To run a function in another thread:

1. Create an instance of the `Thread` class.
2. Specify the name of the function via the `target` argument.
3. Call the `start()` method.

First, we must create a new `Thread` object and specify the function we wish to execute in a new thread via the `target` argument.

```
...
# create a thread
thread = Thread(target=task)
```

The function executed in another thread may have arguments in which case they can be specified as a tuple and passed to the `args` argument of the `Thread` class constructor or as a dictionary to the `kwargs` argument.

```
...
# create a thread
thread = Thread(target=task, args=(arg1, arg2))
```

We can then start executing the thread by calling the `start()` method.

The `start()` method will return immediately and the operating system will execute the target function in a separate native thread as soon as it is able.

```
...
# run the new thread
thread.start()
```

The caller can block and wait for the new thread to complete. This can be achieved by joining the new thread by calling the `join()` method.

```
...
# wait for the thread to finish
thread.join()
```

And that's all there is to it.

We do not have control over when the thread will execute precisely or which CPU core will execute it. Both of these are low-level responsibilities that are handled by the underlying operating system.

Next, let's look at a worked example of executing a function in a new thread.

In this example, we will define a custom function that blocks for a moment then prints a message. We will then execute this function in a new thread and wait for the thread to terminate by joining it from the main thread.

The complete example is listed below.

```
# SuperFastPython.com
# example of running a function in a new thread
from time import sleep
from threading import Thread

# custom function to be executed in a new thread
def task():
    # block for a moment
    sleep(1)
    # report a message
    print('This is from another thread')

# protect the entry point
if __name__ == '__main__':
    # create a new thread instance
    thread = Thread(target=task)
    # start executing the function in the new thread
    thread.start()
    # wait for the thread to finish
    print('Waiting for the thread...')
    thread.join()
```

Running the example first creates the `Thread` then calls the `start()` method. This does not start the thread immediately, but instead allows the operating system to schedule the function to execute as soon as possible.

At some point the new thread is run by the underlying operating system, which will execute our target function.

The main thread of our main thread then prints a message waiting for the new thread to complete, then calls the `join()` method to explicitly block and wait for the new thread to terminate.

The custom function runs, blocks for a moment, then reports a message.

Once the custom function returns, the new thread is closed. The `join()` method then returns and the main thread exits.

```
Waiting for the thread...
This is from another thread
```

NOTE: Unlike new processes, new threads do not need to explicitly flush the buffer when using the `print()` function from a new process (e.g. setting `flush=True` is not necessarily).

NOTE: Unlike new processes, we do not need to protect the entry point of the program when starting new threads (e.g. adding `if __name__ == '__main__':`). Nevertheless, we will in the complete examples as it is a good practice.

Next, let's explore how we might extend the `Thread` class to run custom code in a new thread.

How to Extend the `Thread` Class

We can also execute functions in a new thread by extending the `Thread` class and overriding the `run()` method.

This can be achieved by first extending the class, just like any other Python class.

For example:

```
# custom thread class
class CustomThread(Thread):
    # ...
```

Then the `run()` method of the `Thread` class must be overridden to contain the code that we wish to execute in another thread.

For example:

```
# override the run method
def run(self):
    # ...
```

And that's it.

Given that it is a custom class, we can define a constructor for the class and use it to pass in data that may be needed in the `run()` method, stored such as instance variables.

We can also define additional methods in the class to split up the work we may need to complete in another thread.

Next, let's look at a worked example of extending the `Thread` class.

We can define a class `CustomThread` that extends the `Thread` class and overrides the `run()` method with custom code that blocks for a moment, then reports a message. We can then create an instance of the custom class and start it like a `Process` which will execute the content of the `run()` method in a new thread.

The complete example is listed below.

```
# SuperFastPython.com
# example of extending the thread class
from time import sleep
```

```
from threading import Thread

# custom thread class
class CustomThread(Thread):
    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # report a message
        print('This is another thread')

# protect the entry point
if __name__ == '__main__':
    # create the thread
    thread = CustomThread()
    # start the thread
    thread.start()
    # wait for the thread to finish
    print('Waiting for the thread to finish')
    thread.join()
```

Running the example first creates an instance of the custom class.

The inherited `start()` method is then called which starts a new thread and executes the content of the `run()` method in the new thread.

The main thread blocks, waiting for the new thread to finish.

The overridden `run()` method blocks for a moment then reports a message.

The main thread unblocks and continues on, exiting the main thread and in turn terminating the main thread.

```
Waiting for the thread to finish
This is another thread
```

Thread-Local Storage

Thread-local data storage is a mechanism in multithreaded programming that allows data to be stored and accessed in a way that is private to each thread.

Typically this involves creating an instance of a thread-local object that is shared between objects on which data is stored and retrieved.

Data stored and retrieved on the thread-local object may have the same variable names across threads, meaning the same code can be used across threads, a common approach when using worker threads.

Importantly, the reads and writes to the thread-local object are unique and private at the thread-level. This means one thread may write a variable with the name `address` and another thread may read or write a variable with an identical name but it will not interact with the variable stored by the first thread.

If executing the same code and using the same thread-local instance, then each thread has its own private version of a named variable and its assigned value in the thread-local storage.

This can be useful when multiple threads need to store local data such as a partial solution or a temporary variable and need to use the same code while executing, such as the same instance of an object.

Threads can store local data via an instance of a thread-local object via the `local()` module function.

For example:

```
...
# create a local instance
local = local()
```

Then data can be stored on the local instance with attributes of any arbitrary name.

For example:

```
...
# store some data
local.custom = 33
```

Importantly, other threads can use the same property names on the same instance but the values will be limited to each thread.

This is like a namespace limited to each thread and is called *thread-local data*. It means that threads cannot access or read the local data of other threads.

Importantly, each thread must be able to access the same `local` instance in order to access the stored data.

We can explore how to use thread-local data with a worked example.

In this example we will create a thread-local storage object in the main thread. The main thread will store a value and report the value that was stored.

A new thread will then be created and the thread-local object will be shared with it. The main thread will wait for the new thread to terminate. The new thread will block for a moment to simulate work, then will store a value on the shared thread-local object using the same attribute name, although with a different value. It will then report the value stored.

Finally, main thread will run again and report the value stored on the thread-local object to confirm it was not changed by the new thread.

The complete example is listed below.

```
# SuperFastPython.com
# example of sharing thread-local storage
from time import sleep
from threading import Thread
from threading import local

# custom function executed in a new thread
def task(shared_local):
    # block for a moment to simulate work
    sleep(1)
    # store a private variable on the thread local
    shared_local.value = 33
    # report the stored value
    print(f'Thread stored: {shared_local.value}')

# protect the entry point
if __name__ == '__main__':
    # create a shared thread-local instance
    local_storage = local()
    # store a private variable on the thread local
    local_storage.value = 100
    # report the stored value
    print(f'Main stored: {local_storage.value}')
    # create a new thread to run the custom function
    thread = Thread(target=task, args=(local_storage,))
    # start the new thread
    thread.start()
    # wait for the thread to terminate
    thread.join()
    # report the stored value
    print(f'Main sees: {local_storage.value}')
```

Running the example first creates a thread-local object.

The main thread then defines a new private attribute on the thread-local object named `value` and assigns it a value of 100.

A new thread is then created and configured to execute our custom `task()` function and is passed the thread-local object as an argument.

The main thread then starts the new thread, then blocks, waiting for the new thread to terminate.

The new thread first blocks for a second, then defines a new private attribute on the thread-local object named `value`, the same as the main thread, and assigns it a different value of 33. It then reports that this `value` attribute exists and was assigned correctly.

The new thread terminates and the main thread continues. It reports the value of the `value` attribute on the shared thread local object.

In this case, we confirm that the `value` was not changed by the new thread, even though both threads operated on the same shared thread-local object.

This highlights how multiple threads may share the same thread-local object and maintain private variables with the same name.

If the new thread reported the `value` attribute on the thread-local object before it assigned it, then an error would be reported. This is because the `value` attribute assigned by the main thread is private to the main thread and is not visible to the new thread.

```
Main stored: 100
Thread stored: 33
Main sees: 100
```

Lesson Review

Takeaways

Well done, you made it to the end of the lesson.

- You know about the main thread, helper threads and new threads.
- You know about the life-cycle of the `Thread` class in Python.
- You know how to run a custom function in a new `Thread` instance.
- You know how to extend the `Thread` class to run custom code in a new thread.
- You know how to use thread-local storage that is private to each thread.

Exercise

Your task for this lesson is to use what you have just learned about running ad hoc code in a new thread.

Devise a small Python program that executes a repetitive task, such as calling the same function multiple times in a loop. Execute this program and record how long it takes to complete.

The specifics of the task do not matter. You can try to complete something practical, or if you run out of ideas, you can calculate a number, or block with the `sleep()` function.

Now update the program to execute each task using a separate thread. Record how long it takes to execute.

Compare the execution time between the serial and concurrent versions of the program. Calculate the difference in seconds (e.g. it is faster by 5 seconds). Calculate the ratio that the second program is faster than the first program (e.g. it is 2.5x faster).

These calculations may help:

- difference = serial time - concurrent time
- ratio = serial time / concurrent time

If it turns out that the concurrent version of the program is not faster, perhaps change or manipulate the task so that the serial version is slower than the faster version.

This exercise will help you develop the calculations and practice needed to benchmark and compare the performance before and after making code concurrent with the `threading` module.

Share your results online on Twitter, LinkedIn, GitHub, or similar.

Send me the link to your results, I'd love to see what you come up with.

You can send me a message directly via:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

Or share it with me on Twitter via @SuperFastPython.

Further Reading

This section provides resources for you to learn more about the topics covered in this lesson.

- Built-in Functions.
  https://docs.python.org/3/library/functions.html
- `__main__` - Top-level code environment.
  https://docs.python.org/3/library/__main__.html
- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html
- `threading` - Thread-based parallelism.
  https://docs.python.org/3/library/threading.html

Next

In the next lesson, we explore how to configure new threads and how to interact with them in our program.

Lesson 03: Configuring and Interacting with Threads

In this lesson, you will discover how to configure new threads and how to interact with threads in Python via the `threading` module API.

After completing this lesson, you will know:

- How to configure the name of a thread and whether it is a daemon.
- How to query the status of a thread such as if it is running or terminated.
- How to get access to the current thread, main thread, and all running threads.
- How to handle unexpected exceptions in new threads.

Let's get started.

How to Configure New Threads

We can configure new threads by providing arguments to the `Thread` class constructor.

There are two properties of a thread that can be configured, they are the name of the thread and whether the thread is a daemon or not.

Let's take a closer look at each.

Configure Thread Name

Threads can be assigned custom names.

Meaningful names can be assigned based on the tasks or functions executed within the application. This can be helpful for debugging and logging.

The name of a thread can be set via the `name` argument in the `Thread` constructor.

Once created, the name of the thread can be retrieved or changed again by the `name` attribute on a `Thread` object.

The example below demonstrates how to set the name of a thread in the `Thread` class constructor and then report it via the `name` attribute.

```
# SuperFastPython.com
# example of setting the thread name in the constructor
from threading import Thread

# protect the entry point
if __name__ == '__main__':
    # create a thread with a custom name
    thread = Thread(name='MyThread')
    # report thread name
    print(thread.name)
```

Running the example creates the new thread with the custom name then reports the name of the thread.

```
MyThread
```

Next, let's explore how to start daemon threads.

Configure Daemon Threads

Threads can be configured to be *daemon* or *daemonic*, that is, they can be configured as background threads.

The main thread can only exit once all non-daemon threads have exited.

This means that daemon threads can run in the background and do not prevent the main thread of a Python program from exiting when the primary parts of the program have completed.

A thread can be configured to be a daemon by setting the `daemon` argument to `True` in the `Thread` constructor.

The daemon status of a `Thread` object can be queried via the `daemon` attribute. If the thread has not yet started, the daemon attribute can be changed again.

The example below shows how to create a new daemon thread.

```
# SuperFastPython.com
# example of creating a daemon thread
from threading import Thread

# protect the entry point
if __name__ == '__main__':
    # create a daemon thread
    thread = Thread(daemon=True)
    # report if the thread is a daemon
    print(thread.daemon)
```

Running the example creates a new thread and configures it to be a daemon thread via the constructor.

```
True
```

Next, let's explore how we might query the status of a thread via its `Thread` object.

How to Query the Status of a Thread

An instance of the `Thread` class provides a handle on a new thread of execution in the Python interpreter.

As such, it provides attributes that we can use to query properties and the status of the underlying thread.

We saw in the last section how we might set, but also query the name and daemon status of a thread.

Two more qualities of a thread we may want to query in our programs include the native thread identifier and whether the thread is still running (or not).

Let's take a closer look at each.

Query Thread Native Identifier

Each thread has a unique identifier assigned by the Python interpreter and a unique native thread identifier (TID), assigned by the operating system.

The latter is perhaps more interesting we may be able to relate it to the operating systems perspective on currently running threads via other diagnostic software.

Python threads are real native threads, meaning that each thread we create is actually created and managed by the underlying operating system. As such, the operating system will assign a unique integer to each thread that is created on the system across all threads on the system.

Knowing the TIDs for threads in our program can help with logging and with the general maintenance of the system. Other programs may query the TID of the thread.

The thread identifier can be accessed via the `native_id` attribute and is assigned after the thread has been started.

The example below creates a`Thread` object and reports the assigned native identifier.

```
# SuperFastPython.com
# example of reporting the thread native identifier
from threading import Thread

# protect the entry point
if __name__ == '__main__':
    # create the thread
```

```
    thread = Thread()
    # report the thread identifier
    print(thread.native_id)
    # start the thread
    thread.start()
    # report the thread identifier
    print(thread.native_id)
```

Running the example first creates the thread and confirms that it does not have a native identifier before it was started.

The thread is then started and the assigned a native identifier is reported.

NOTE: The native identifier will vary each time the program is run because it is allocated by the operating system.

```
None
16302
```

Next, let's explore how we might check if a thread is running.

Query If Thread Is Alive

A `Thread` object can be alive (*running*) or dead (*terminated*).

If a thread is alive, it means that the `run()` method of the `Thread` object is currently executing.

This means that before the `start()` method is called and after the `run()` method has completed, the thread will not be alive.

We can check if a thread is alive via the `is_alive()` method on the `Thread` object.

The example below creates a `Thread` object then checks whether it is alive.

```
# SuperFastPython.com
# example of assessing whether a thread is alive
from threading import Thread

# protect the entry point
if __name__ == '__main__':
    # create the thread
    thread = Thread()
    # report the thread is alive
    print(thread.is_alive())
```

Running the example creates a new `Thread` object then reports that the thread is not alive.

```
False
```

Next, let's explore how we might get a `Thread` object for running thread within our program.

How to Get the Current, Main, and All Threads

The `threading` module provides a number of utility functions for getting `Thread` objects for currently running threads.

This includes functions for getting the current thread, the main thread, and a list of currently active threads.

Let's take a look at each in turn.

Get the Current Thread

We can get a `Thread` object for the thread running the current code.

This can be achieved via the `current_thread()` module function that returns a `Thread` object.

The example below demonstrates how we can use this function to access the `Thread` object for the current thread, which will be the main thread.

```
# SuperFastPython.com
# example of getting access to the current thread
from threading import current_thread

# protect the entry point
if __name__ == '__main__':
    # get the current thread
    thread = current_thread()
    # report details
    print(thread)
```

Running the example gets the `Thread` object for the currently running thread.

The details are then reported, showing that we have accessed the main thread, the name of the main thread as *MainThread* as expected, its status of *started* and the Python thread identifier.

NOTE: The thread identifier will vary each time the program is run because it is allocated by the Python interpreter.

```
<_MainThread(MainThread, started 4506099200)>
```

Get the Main Thread

We can get a `Thread` object for the main thread.

This is helpful if we need to access the main thread from another thread.

A `Thread` object for the main thread can be acquired via the `main_thread()` module function.

The example below demonstrates how we can use this function to access the `Thread` object for the main thread.

```
# SuperFastPython.com
# example of getting access to the main thread
from threading import main_thread

# protect the entry point
if __name__ == '__main__':
    # get the main thread
    thread = main_thread()
    # report details
    print(thread)
```

Running the example first gets a `Thread` object that represents the main thread then reports its details.

NOTE: The thread identifier will vary each time the program is run because it is allocated by the Python interpreter.

```
<_MainThread(MainThread, started 4423302656)>
```

Get All Active Threads

We can get an iterable of all active threads in the current process.

This can be achieved via the `enumerate()` module function that returns an iterable of `Thread` objects, one for each running thread.

We can develop a small example that first starts a number of new threads, has each new thread block for a moment, then have the main thread get and report the list of running threads.

We have to fully qualify the `threading.enumerate()` module function rather than import it directly because it conflicts with the built-in `enumerate()` function.

The complete example is listed below.

```
# SuperFastPython.com
# example of getting a list of active threads
from time import sleep
from threading import Thread
import threading

# custom function to be executed in a new thread
def task():
    # block for a moment
```

```
    sleep(1)

# protect the entry point
if __name__ == '__main__':
    # create a number of new threads
    threads = [Thread(target=task) for _ in range(5)]
    # start the new threads
    for thread in threads:
        thread.start()
    # get a list of all running threads
    running_threads = threading.enumerate()
    # report a count of active threads
    print(f'Active Threads: {len(running_threads)}')
    # report each in turn
    for thread in running_threads:
        print(thread)
```

Running the example first creates five new threads to run our custom `task()` function, then starts them.

Each thread blocks for a moment, giving the main thread time to perform the query.

A list of all active threads is then retrieved in the main thread. The count is reported, which is shown as six, as we expect (five new threads plus the main thread), then the details of each thread are then reported.

This highlights how we can access all active threads within a Python process.

It also shows that new threads are assigned names automatically based on the order they were created, e.g. *Thread-1*, *Thread-2*, and so on.

NOTE: The thread identifiers will vary each time the program is run because they are allocated by the Python interpreter.

```
Active Threads: 6
<_MainThread(MainThread, started 4542012928)>
<Thread(Thread-1, started 123145554694144)>
<Thread(Thread-2, started 123145571483648)>
<Thread(Thread-3, started 123145588273152)>
<Thread(Thread-4, started 123145605062656)>
<Thread(Thread-5, started 123145621852160)>
```

How to Handle Unexpected Exceptions in New Threads

It is possible for new threads to raise an unexpected exception while executing.

This has a catastrophic effect on the new thread, unwinding the stack of function calls and

ultimately stopping the thread from running.

Ideally, we would like to know when an unexpected exception occurs in new threads so that we might take appropriate action to clean-up any resources and perhaps log the fault.

The `threading` module API provides this capability by registering an exception handler function called an `excepthook`.

How to Register a Thread Exception Handler

We can specify how to handle unexpected errors and exceptions that occur within new threads via the exception hook, referred to as `excepthook`.

By default, there is no exception hook, in which case the `sys.excepthook` function is called that reports the exception and a stack trace.

We can specify a custom exception hook function that will be called whenever a `Thread` fails with an unhandled `Error` or `Exception`.

This can be achieved via the `threading.excepthook` function.

First, we must define a function that takes a single argument that will be an instance of the `ExceptHookArgs` class, containing details of the exception and thread.

Of note in the `ExceptHookArgs` class is the `exc_value` attribute that will have any message reported in the exception that was raised. The `threading` module API documentation has more information on the other attributes.

```
# custom exception hook
def custom_hook(args):
    # ...
```

We can then specify the exception hook function to be called whenever an unhandled exception bubbles up to the top level of a thread.

```
...
# set the exception hook
threading.excepthook = custom_hook
```

And that's all there is to it.

Note, using an `excepthook` requires Python version 3.8 or higher.

Example of Handling Thread Exceptions

We can explore how to use the exception hook to handle an unexpected exception.

In this example we will register a custom function to handle an exception raised in new threads. The handler will report the message of any exception raised. We will then execute a custom function in a new thread. The function will block for a moment, then raise an

exception (not so) unexpectedly with a custom message. The handler will then report this message and the main thread will remain unaffected by the raised exception.

The complete example is listed below.

```
# SuperFastPython.com
# example of handling a thread's unexpected exception
from time import sleep
from threading import Thread
import threading

# custom exception hook
def custom_hook(args):
    # report the failure
    print(f'Thread failed: {args.exc_value}')

# target function that raises an exception
def task():
    # report a message
    print('Working...')
    # block for a moment
    sleep(1)
    # rise an "unexpected" exception
    raise Exception('Something bad happened')

# protect the entry point
if __name__ == '__main__':
    # register the exception hook function
    threading.excepthook = custom_hook
    # create a thread
    thread = Thread(target=task)
    # run the thread
    thread.start()
    # wait for the thread to finish
    thread.join()
    # report that the main thread is not dead
    print('Continuing on...')
```

Running the example first registers the custom function to handle exceptions raised in new threads.

The main thread then creates a new thread configured to execute our custom `task()` function.

The thread is started, then the main thread blocks and waits for the new thread to terminate.

The new thread starts running. It blocks for a moment then fails with an exception.

The exception bubbles up to the top level of the thread at which point the custom exception

hook function is called. Our custom message is then reported.

The main thread then continues on and reports a message.

This example provides a helpful template if we want to perform special actions when a thread fails unexpectedly, such as log a message to a file.

```
Working...
Thread failed: Something bad happened
Continuing on...
```

Lesson Review

Takeaways

Well done, you made it to the end of the lesson.

- You know how to configure the name of a thread and whether it is a daemon.
- You know how to query the status of a thread such as if it is running or terminated.
- You know how to get access to the current thread, main thread, and all running threads.
- You know how to handle unexpected exceptions in new threads.

Exercise

Your task for this lesson is to use what you have learned about interacting with running threads in your program.

Develop a small program that creates and starts one or more threads performing some arbitrary task.

Then from the main thread, wait for some time or a trigger and query the status one or more of the threads you have created.

Extend the example so that the main thread waits in a loop for a running thread to terminate, checking the status each iteration. This is called a busy loop.

Share your results online on Twitter, LinkedIn, GitHub, or similar.

Send me the link to your results, I'd love to see what you come up with.

You can send me a message directly via:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

Or share it with me on Twitter via @SuperFastPython.

Further Reading

This section provides resources for you to learn more about the topics covered in this lesson.

- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html
- `threading` - Thread-based parallelism.
  https://docs.python.org/3/library/threading.html

Next

In the next lesson, we will discover how we might synchronize and coordinate threads using concurrency primitives like locks and semaphores.

Lesson 04: Synchronize and Coordinate Threads

In this lesson, we will explore how to use concurrency primitives to synchronize and coordinate threads.

After completing this lesson, you will know:

- How to protect critical sections from race conditions with mutex locks.
- How to limit access to a protected resource with a semaphore.
- How to signal between threads using an event.
- How to coordinate action with wait and notify using a condition variable.
- How to coordinate multiple threads using a barrier.

Let's get started.

How to Protect Critical Sections with a Mutex Lock

A mutual exclusion lock or mutex lock is a concurrency primitive intended to prevent a race condition.

A race condition is a concurrency failure case when two threads run the same code and access or update the same resource (e.g. data variables, stream, etc.) leaving the resource in an unknown and inconsistent state.

Race conditions often result in unexpected behavior of a program and/or corrupt data.

These sensitive parts of code that can be executed by multiple threads concurrently and may result in race conditions are called critical sections. A critical section may refer to a single block of code, but it also refers to multiple accesses to the same data variable or resource from multiple functions.

Python provides a mutual exclusion lock for use with threads via the `Lock` class.

An instance of the `Lock` class can be created and then acquired by threads before accessing a critical section, and released after exiting the critical section.

For example:

```
...
# create a lock
lock = Lock()
# acquire the lock
lock.acquire()
# ...
# release the lock
lock.release()
```

Only one thread can have the lock at any time. If a thread does not release an acquired lock, it cannot be acquired again.

The thread attempting to acquire the lock will block until the lock is acquired, such as if another threads currently holds the lock then releases it.

We can also use the lock via the context manager interface, allowing the critical section to be a block within the context manager and for the lock to be released automatically once the block is exited, normally or otherwise.

For example:

```
...
# create a lock
lock = Lock()
# acquire the lock
with lock:
    # ...
```

This is the preferred usage as it makes it clear where the protected code begins and ends, and ensures that the lock is always released, even if there is an exception or error within the critical section.

We can develop an example to demonstrate how to use the mutex lock.

In this example we will define a target task function that takes a lock as an argument and uses the lock to protect a critical section, which in this case will print a message and block for a moment.

The complete example is listed below.

```
# SuperFastPython.com
# example of protecting a critical section with a mutex
from time import sleep
from random import random
from threading import Thread
from threading import Lock

# custom function to be executed in a new thread
def task(shared_lock, ident, value):
    # acquire the lock
```

```
    with shared_lock:
        # report a message
        print(f'>{ident} got lock, sleeping {value}')
        # block for a fraction of a second
        sleep(value)

# protect the entry point
if __name__ == '__main__':
    # create the shared mutex lock
    lock = Lock()
    # create a number of threads with different args
    threads = [Thread(target=task,
        args=(lock, i, random())) for i in range(10)]
    # start the threads
    for thread in threads:
        thread.start()
    # wait for all threads to finish
    for thread in threads:
        thread.join()
```

Running the example starts ten threads that are all configured to execute our custom function.

The mew threads are then started and the main thread blocks until all new threads finish.

Each new thread attempts to acquire the lock within the `task()` function.

Only one thread can acquire the lock at a time and once they do, they report a message including their id and how long they will sleep. The thread then blocks for a fraction of a second before releasing the lock.

NOTE: Results will vary each time the program is run given the use of random numbers.

```
>0 got lock, sleeping 0.006231113512758513
>1 got lock, sleeping 0.827755165754169
>2 got lock, sleeping 0.922983492451851
>3 got lock, sleeping 0.13485349385955503
>4 got lock, sleeping 0.4675411006553878
>5 got lock, sleeping 0.7642881058679402
>6 got lock, sleeping 0.30274382407738154
>7 got lock, sleeping 0.7857943658088345
>8 got lock, sleeping 0.23981771304174082
>9 got lock, sleeping 0.8035570947894208
```

The `threading` module also provides a reentrant lock via the `RLock` class. This mutex allows a thread to acquire the lock subsequently multiple times (e.g. is reentrant) unlike the `Lock` class.

Reentrant locks are helpful in cases where the same lock may be used in multiple places in

the code and a function that makes use of the lock may be called from a critical section where the lock is already held.

Next, let's explore how to use a semaphore to protect a shared resource.

How to Limit Access to a Resource with a Semaphore

A semaphore is a concurrency primitive that allows a limit on the number of threads that can acquire a lock protecting a critical section or resource.

It is an extension of a mutual exclusion (mutex) lock that adds a count for the number of threads that can acquire the lock before additional threads will block. Once full, new threads can only acquire access on the semaphore once an existing thread holding the semaphore releases access.

When a semaphore is created, the upper limit on the counter is set. If it is set to be 1, then the semaphore will operate like a mutex lock.

Python provides a semaphore for threads via the `Semaphore` class.

The `Semaphore` object must be configured when it is created to set the limit on the internal counter. This limit will match the number of concurrent threads that can hold the semaphore.

For example, we might want to set it to 100:

```
...
# create a semaphore with a limit of 100
semaphore = Semaphore(100)
```

In this implementation, each time the `Semaphore` is acquired, the internal counter is decremented. Each time the `Semaphore` is released, the internal counter is incremented.

The `Semaphore` cannot be acquired if it has no available positions (e.g. the count is zero) in which case, threads attempting to acquire it must block until a position becomes available.

The semaphore can be acquired by calling the `acquire()` method, for example:

```
...
# acquire the semaphore
semaphore.acquire()
```

By default, it is a blocking call, which means that the calling thread will block until access becomes available on the semaphore.

Once acquired, the semaphore can be released again by calling the `release()` method.

```
...
# release the semaphore
semaphore.release()
```

The `Semaphore` class supports usage via the context manager, which will automatically acquire and release the semaphore for us. As such it is the preferred way to use semaphores

in our programs.

For example:

```
...
# acquire the semaphore
with semaphore:
    # ...
```

We can explore how to use a `Semaphore` with a worked example.

In this example, we will start a suite of threads but limit the number that can perform an action simultaneously. A semaphore will be used to limit the number of concurrent tasks that may execute which will be less than the total number of threads, allowing some threads to block, wait for access, then be notified and acquire access.

The complete example is listed below.

```
# SuperFastPython.com
# example of a semaphore to limit access to resource
from time import sleep
from random import random
from threading import Thread
from threading import Semaphore

# custom function to be executed in a new thread
def task(shared_semaphore, ident):
    # attempt to acquire the semaphore
    with shared_semaphore:
        # generate a random value between 0 and 1
        val = random()
        # block for a fraction of a second
        sleep(val)
        # report result
        print(f'Thread {ident} got {val}')

# protect the entry point
if __name__ == '__main__':
    # create the shared semaphore
    semaphore = Semaphore(2)
    # create threads
    threads = [Thread(target=task,
        args=(semaphore, i)) for i in range(10)]
    # start new threads
    for thread in threads:
        thread.start()
    # wait for new threads to finish
    for thread in threads:
```

```
        thread.join()
```

Running the example first creates the shared `Semaphore` object then starts ten new threads.

All ten threads attempt to acquire the semaphore, but only two threads are granted access at a time. The threads on the semaphore do their work and release the semaphore when they are done, at random intervals.

Each release of the semaphore (via the context manager) allows another thread to acquire access and perform its simulated calculation, all the while allowing only two of the threads to be running within the critical section at any one time, even though all ten threads are executing their run methods.

NOTE: Results will vary each time the program is run given the use of random numbers.

```
Thread 0 got 0.5216719174591998
Thread 1 got 0.8041024694461771
Thread 2 got 0.5849073010981766
Thread 4 got 0.3716280159276608
Thread 3 got 0.7747051189484008
Thread 6 got 0.35838650237191405
Thread 7 got 0.017818017527911234
Thread 5 got 0.7913564693663327
Thread 8 got 0.9457018638391528
Thread 9 got 0.9381748992644386
```

Next, we will explore how we might signal between threads using an event.

How to Signal Between Threads Using an Event

An event is a thread-safe boolean flag that can be used to signal between two or more threads.

Python provides an event object for threads via the `Event` class.

An `Event` class wraps a boolean variable that can either be *set* (`True`) or *not set* (`False`). Threads sharing the `Event` object can check if the event is set, set the event, clear the event (make it not set), or wait for the event to be set.

The `Event` provides an easy way to share a boolean variable between threads that can act as a trigger for an action.

First, an `Event` object must be created and the event will be in the *not set* state.

```
...
# create an instance of an event
event = Event()
```

Once created we can check if the event has been set via the `is_set()` method which will return `True` if the event is set, or `False` otherwise.

For example:

```
...
# check if the event is set
if event.is_set():
    # do something...
```

The `Event` can be set via the `set()` method. Any threads waiting on the event to be set will be notified.

For example:

```
...
# set the event
event.set()
```

Finally, threads can wait for the event to set via the `wait()` method. Calling this method will block until the event is marked as set (e.g. another thread calling the `set()` method). If the event is already set, the `wait()` method will return immediately.

```
...
# wait for the event to be set
event.wait()
```

We can explore how to use a `Event` object.

In this example we will create a suite of threads that each will perform some work and report a message. All threads will use an event to wait to be set before starting their work. The main thread will set the event and trigger the new threads to start work.

```
# SuperFastPython.com
# example of using an event object with threads
from time import sleep
from random import random
from threading import Thread
from threading import Event

# custom function to be executed in a new thread
def task(shared_event, number):
    # wait for the event to be set
    print(f'Thread {number} waiting...')
    shared_event.wait()
    # begin work, generate a random number
    value = random()
    # block for a fraction of a second
    sleep(value)
    # report a message
    print(f'Thread {number} got {value}')

# protect the entry point
```

```
if __name__ == '__main__':
    # create a shared event object
    event = Event()
    # create a suite of threads
    threads = [Thread(target=task,
        args=(event, i)) for i in range(5)]
    # start all threads
    for thread in threads:
        thread.start()
    # block thread a moment
    print('Main thread blocking...')
    sleep(2)
    # trigger all threads
    event.set()
    # wait for all threads to terminate
    for thread in threads:
        thread.join()
```

Running the example first creates and starts five new threads.

Each new thread waits on the event before it starts its work, reporting a message that it is waiting.

The main thread blocks for a moment, allowing all new threads to begin and start waiting on the event.

The main thread then sets the event. This triggers all five new threads that perform their simulated work and report a message.

NOTE: Results will vary each time the program is run given the use of random numbers.

```
Thread 0 waiting...
Thread 1 waiting...
Thread 2 waiting...
Thread 3 waiting...
Thread 4 waiting...
Main thread blocking...
Thread 0 got 0.16159900626143686
Thread 3 got 0.24313379561234727
Thread 1 got 0.40336241377107984
Thread 2 got 0.41388494943215526
Thread 4 got 0.7352919749640477
```

Next, let's explore how we can wait and notify between threads using a condition variable.

How to Coordinate Using a Condition Variable

A condition variable (also called a monitor) allows multiple threads to wait and be notified about some result.

A condition can be acquired by a thread after which it can wait to be notified by another thread that something has changed. While waiting, the thread is blocked and releases the lock on the condition for other threads to acquire.

Another thread can then acquire the condition, make a change in the program, and notify one, all, or a subset of threads waiting on the condition that something has changed.

The waiting thread can then wake-up, re-acquire the condition, perform checks on any changed state and perform required actions.

Python provides a condition variable via the `Condition` class.

```
...
# create a new condition variable
condition = Condition()
```

In order for a thread to make use of the `Condition`, it must acquire it and release it, like a mutex lock.

This can be achieved manually with the `acquire()` and `release()` methods.

For example, we can acquire the `Condition` and then wait on the condition to be notified and finally release the condition as follows:

```
...
# acquire the condition
condition.acquire()
# wait to be notified
condition.wait()
# release the condition
condition.release()
```

An alternative to calling the `acquire()` and `release()` methods directly is to use the context manager, which will perform the acquire/release automatically for us, for example:

```
...
# acquire the condition
with condition:
    # wait to be notified
    condition.wait()
```

The `wait()` method will wait forever until notified by default. We can also pass a `timeout` argument which will allow the thread to stop blocking after a time limit in seconds.

We also must acquire the condition in a thread if we wish to notify waiting threads. This too can be achieved directly with the acquire/release methods calls or via the context manager.

We can notify a single waiting threads via the `notify()` method.

For example:

```
...
# acquire the condition
with condition:
    # notify a waiting threads
    condition.notify()
```

The notified threads will stop-blocking as soon as it can reacquire the condition. This will be attempted automatically as part of its call to `wait()`, we do not need to do anything extra.

We can notify all threads waiting on the condition via the `notify_all()` method.

```
...
# acquire the condition
with condition:
    # notify all threads waiting on the condition
    condition.notify_all()
```

Now that we know how to use the `Condition` class, let's look at a worked example.

In this example, we will create a new thread to simulate performing some work that the main thread is dependent upon. Once prepared, the new thread will notify the waiting main thread, then the main thread will continue on.

The complete example is listed below.

```
# SuperFastPython.com
# example of wait/notify with a condition for threads
from time import sleep
from threading import Thread
from threading import Condition

# custom function to be executed in a new thread
def task(shared_condition):
    # block for a moment
    sleep(1)
    # notify a waiting thread that the work is done
    print('Thread sending notification...')
    with shared_condition:
        shared_condition.notify()

# protect the entry point
if __name__ == '__main__':
    # create a condition
    condition = Condition()
    # acquire the condition
    print('Main thread waiting for data...')
    with condition:
```

```
        # create a new thread to execute the task
        thread = Thread(target=task, args=(condition,))
        # start the new new thread
        thread.start()
        # wait to be notified by the new thread
        condition.wait()
    # we know the data is ready
    print('Main thread all done')
```

Running the example first creates the condition variable.

The condition variable is acquired, then a new new thread is created and started.

The new thread blocks for a moment to simulate work, then notifies the waiting main thread.

Meanwhile the main thread waits to be notified by the new thread, then once notified it continues on.

```
Main thread waiting for data...
Thread sending notification...
Main thread all done
```

Next, let's explore how we can coordinate threads using a barrier.

How to Coordinate Threads with a Barrier

A barrier is a synchronization primitive.

It allows multiple threads to wait on the same barrier object (e.g. at the same point in code) until a predefined fixed number of threads arrive (e.g. the barrier is full), after which all threads are then notified and released to continue their execution.

Internally, a barrier maintains a count of the number of threads waiting on the barrier and a configured maximum number of parties (threads) that are expected. Once the expected number of parties reaches the pre-defined maximum, all waiting threads are notified.

This provides a useful mechanism to coordinate actions between multiple threads.

Python provides a barrier via the `Barrier` class.

A `Barrier` object must first be created and configured via the constructor specifying the number of parties (threads) that must arrive before the barrier will be lifted.

For example:

```
...
# create a barrier
barrier = Barrier(10)
```

We can also perform an action once all threads reach the `Barrier` which can be specified via the `action` argument in the constructor.

This action must be a callable such as a function or a lambda that does not take any arguments and will be executed by one thread once all threads reach the barrier but before the threads are released.

```
...
# configure a barrier with an action
barrier = Barrier(10, action=my_function)
```

Once configured, the `Barrier` object can be shared between threads and used.

A threads can reach and wait on the barrier via the `wait()` method, for example:

```
...
# wait on the barrier for all other threads to arrive
barrier.wait()
```

This is a blocking call and will return once all other threads (the pre-configured number of parties) have reached the `Barrier`.

Now that we know how to use the barrier in Python, let's look at a worked example.

In this example we will create a suite of threads, each required to perform some blocking calculation. We will use a `Barrier` to coordinate all threads after they have finished their work and perform some action in the main thread.

The complete example is listed below.

```
# SuperFastPython.com
# example of using a barrier with threads
from time import sleep
from random import random
from threading import Thread
from threading import Barrier

# custom function to be executed in a new thread
def task(shared_barrier, ident):
    # generate a unique value between 0 and 10
    value = random() * 10
    # block for a moment
    sleep(value)
    # report result
    print(f'Thread {ident} got: {value}')
    # wait for all other threads to complete
    shared_barrier.wait()

# protect the entry point
if __name__ == '__main__':
    # create a barrier for (5 threads + 1 main thread)
    barrier = Barrier(5 + 1)
    # create the new threads
```

```
    threads = [Thread(target=task,
        args=(barrier, i)) for i in range(5)]
    # start the new threads
    for thread in threads:
        # start thread
        thread.start()
    # wait for all new threads to finish
    print('Main thread waiting on all results...')
    barrier.wait()
    # report once all thread are done
    print('All threads have their result')
```

Running the example first creates the shared `Barrier` then creates and starts the new threads.

Each new thread performs its calculation and then waits on the barrier for all other threads to finish.

Finally, the threads finish and are all released, including the main thread, reporting a final message.

NOTE: Results will vary each time the program is run given the use of random numbers.

```
Main thread waiting on all results...
Thread 3 got: 3.4959139560584775
Thread 0 got: 4.361852403169983
Thread 2 got: 5.248809802581375
Thread 1 got: 8.513618947370842
Thread 4 got: 9.517751779792144
All threads have their result
```

Lesson Review

Takeaways

Well done, you made it to the end of the lesson.

- You know how to protect critical sections from race conditions with mutex locks.
- You know how to limit access to a protected resource with a semaphore.
- You know how to signal between threads using an event.
- You know how to coordinate action with wait and notify using a condition variable.
- You know how to coordinate multiple threads using a barrier.

Exercise

Your task for this lesson is to use what you have learned about concurrency primitives.

Develop a program where multiple threads add and subtract from a single shared integer value.

You could have one or more threads that add one to a balance many times each in a loop, and one or more threads do the same by subtracting one from the same shared global variable.

For example:

```
# add to the balance
def add():
    global balance
    for i in range(1000000):
        balance += 1

# subtract from the balance
def subtract():
    global balance
    for i in range(1000000):
        balance -= 1
```

Confirm that the program results in a race condition by running the example multiple times and getting different results.

Update the example to be thread-safe and no longer suffer the race condition. Try a mutex lock. Also try a semaphore.

It is important that you experience a race condition in Python. Many developers falsely believe that race conditions are not possible in Python or are not something the need to worry about. Once you see one for yourself and know how to fix it, you will be able to bring this confidence with you into your future projects.

Share your results online on Twitter, LinkedIn, GitHub, or similar.

Send me the link to your results, I'd love to see what you come up with.

You can send me a message directly via:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

Or share it with me on Twitter via @SuperFastPython.

Further Reading

This section provides resources for you to learn more about the topics covered in this lesson.

- Race condition, Wikipedia.
  https://en.wikipedia.org/wiki/Race_condition
- Mutual exclusion, Wikipedia.
  https://en.wikipedia.org/wiki/Mutual_exclusion

- Semaphore (programming), Wikipedia.
  https://en.wikipedia.org/wiki/Semaphore_(programming)
- Monitor (synchronization), Wikipedia.
  https://en.wikipedia.org/wiki/Monitor_(synchronization)
- Barrier (computer science), Wikipedia.
  https://en.wikipedia.org/wiki/Barrier_(computer_science)
- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html
- `random` - Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- `threading` - Thread-based parallelism.
  https://docs.python.org/3/library/threading.html

Next

In the next lesson, we will discover how to share data between threads and return values from new threads.

Lesson 05: Share Data Between Threads

In this lesson, we will explore how we can share data between threads.

After completing this lesson, you will know:

- How to share data between threads using global variables.
- How to return values from threads using instance variables.
- How to share data between threads using thread-safe queues.

Let's get started.

Share Data Using Global Variables

A simple approach to share data between threads is to use global variables.

A benefit of threads is the memory model. Threads are able to share memory within a process, such as variables.

For example, all threads in a process will be able to access global variables.

This can be dangerous, but it may be a preferred approach to data sharing if we already have a function executing in a new thread and need to get data out of the thread.

Review of Global Variables

A global variable is a variable defined outside of a function.

For example:

```
...
# define a global variable
data = 66
```

If the global variable is defined prior to the definition of functions and classes, then those functions and classes can access it directly, but not modify it.

For example:

```
# custom function
def custom():
    # access the global variable
    print(data)
```

If the global variable is defined after functions and classes, then the scope of the variable can be specified using the `global` expression, allowing the global variable to be accessed at the correct scope and modified if needed.

For example:

```
# custom function
def custom():
    # scope the global variable
    global data
    # modify the global variable
    data = 33
```

Next, let's consider a possible issue in having a global variable accessed by multiple threads.

Access Global Variables From Multiple Threads

Multiple threads can access a global variable directly.

The danger of sharing data using global variables is that there could be a race condition between the new thread storing data and one or more other threads reading that data. Race conditions are still possible, even in the presence of the GIL as we saw in *Lesson 04: Synchronize and Coordinate Threads*.

The solution could be to protect the data with a concurrency primitive.

We can protect the global variable from race conditions by using a mutual exclusion lock via the `Lock` class.

First, we can create a `Lock` at the same scope as the global variable.

For example:

```
...
# define a global variable
data = 66
# define a lock to protect the global variable
lock = Lock()
```

Each time the global variable is read or modified, it must be done so via the `Lock`.

Specifically, we must acquire the `Lock` before we attempt to interact with the global variable. This will ensure that only one thread is interacting with the global variable at a time.

For example:

```
# custom function
def custom():
    # acquire the lock for the global variable
    with lock:
        # modify the global variable
        data = 33
```

Next, let's look at an example of sharing data between threads using a global variable.

Example of Sharing Data via Global Variables

We can explore how we might use global variables to share data between threads.

In this example, we will define a global variable in the main thread. We will then start a new thread to execute a custom function that accesses the global variable and assigns it a new value. Once the new thread is completed, the main thread will report the value that was assigned to the global variable by the new thread.

This approach to sharing data might be preferred if we already have a function that we are running in a new thread and we don't want to modify it by having it receive a shared object as an argument.

The complete example is listed below.

```
# SuperFastPython.com
# example of threads sharing data with global variables
from time import sleep
from threading import Thread

# custom function to be executed in a new thread
def task():
    # block for a moment
    sleep(1)
    # correctly scope the global variable
    global data
    # store data in the global variable
    data = 'Hello from a new thread'

# protect the entry point
if __name__ == '__main__':
    # define the global variable
    data = None
    # create a new thread
    thread = Thread(target=task)
    # start the thread
    thread.start()
    # wait for the thread to finish
```

```
    thread.join()
    # report the global variable
    print(data)
```

Running the example first define the global variable and assigns it a value of `None`.

The main thread then creates a new thread and starts it, executing our custom target function.

The new thread then stores data against the global variable, being sure to explicitly specify the scope to avoid confusion in reading the code and possible bugs in the future.

The main thread blocks until the new thread terminates then reports the simulated return value stored in the global variable.

This highlights how we might share a global variable between threads. In this case, access and modification to the global variable is thread-safe and a mutex lock is not required.

```
Hello from a new thread
```

Next, let's explore how we might share data using instance variables on `Thread` objects.

Return Values From Threads via Instance Variables

When using new threads, we may need to return a value from the thread to another thread, such as the main thread.

This may be for many reasons, such as:

- The new thread loaded some data that needs to be returned.
- The new thread calculated something that needs to be returned.
- The new thread needs to share its state or status with another thread.

The problem is, we cannot return values from threads.

Threads Cannot Return Values Directly

The `start()` method on a thread calls the `run()` method of the thread that executes our code in a new thread of execution. The `run()` method in turn may call a target function, if configured.

The `start()` method does not block, instead it returns immediately and does not return a value. The run method also does not return a value.

We can join a new thread from the current thread and block until the new thread terminates, but the `join()` method also does not return a value.

Instead, we must return values from a thread indirectly.

Return Values via Instance Variables

A straightforward approach to returning values from a new thread is to extend the `Thread` class and store return values in instance variables.

This involves defining a new class that extends the `Thread` class and defines a constructor that calls the parent constructor.

The `run()` method can be defined that executes the custom code in a new thread and stores data as instance variables.

For example:

```
# custom thread
class CustomThread(Thread):
    # constructor
    def __init__(self):
        # execute the base constructor
        Thread.__init__(self)

    # function executed in a new thread
    def run(self):
        # ...
```

Next, we can explore how to simulate returning a single value from a thread via an instance variable.

Example of Returning Values via Instance Variables

We can explore how to share data via instance variables on an extended `Thread` class.

This is a helpful example that shows us how we can easily return values from new threads.

In this example we will define a new class that extends the `Thread` class, define the instance variable in the constructor and set it to `None`, then override the `run()` function with custom code and set the instance variable. The main thread will then access the return value.

The complete example is listed below.

```
# SuperFastPython.com
# example of thread return values via instance variables
from time import sleep
from threading import Thread

# custom thread
class CustomThread(Thread):
    # constructor
    def __init__(self):
        # execute the base constructor
        Thread.__init__(self)
```

```
        # set a default value
        self.value = None

    # function executed in a new thread
    def run(self):
        # block for a moment
        sleep(1)
        # store data in an instance variable
        self.value = 'Hello from a new thread'

# protect the entry point
if __name__ == '__main__':
    # create a new thread
    thread = CustomThread()
    # start the thread
    thread.start()
    # wait for the thread to finish
    thread.join()
    # get the value returned from the thread
    data = thread.value
    print(data)
```

Running the example first creates the new thread instance. This executes the constructor of the custom class and initializes an instance variable named `value` to `None`.

The main thread then starts the new thread and blocks, waiting for the new thread to complete.

The new thread executes the overriden `run()` method of the custom class. The thread blocks for a moment, then stores a string value against the instance variable.

The new thread terminates and the main thread continues.

The main thread accesses the instance variable of the thread object and reports its value.

This example highlights how we can share data between threads using instance variables of a custom `Thread` object, in this case to simulate a return value from a thread.

```
Hello from a new thread
```

Next, we can explore how we might share data between threads using thread-safe queues.

Share Data Between Threads with Queues

A queue can be used to share data between threads.

A queue is a thread-safe data structure that can be used to share data between threads without a race condition.

The `queue` module provides a number of types of queue classes, such as:

- `Queue`: A fully-featured first-in-first-out (FIFO) queue.
- `SimpleQueue`: A FIFO queue with less functionality.
- `LifoQueue`: A last-in-first-out (LIFO) queue.
- `PriorityQueue`: A queue where the first items out are those with the highest priority.

Generally, the `Queue` class should be used as it is fully featured and operates intuitively.

Next, let's review how we might use queues generally.

Review of Thread-Safe Queues

A queue is a data structure on which items can be added by a call to `put()` and from which items can be retrieved by a call to `get()`.

Thread-safe means that it can be used by multiple threads to put and get items concurrently without a race condition.

The `Queue` class provides a first-in, first-out FIFO queue, which means that the items are retrieved from the queue in the order they were added. The first items added to the queue will be the first items retrieved. This is opposed to other queue types such as last-in, first-out and priority queues.

The `Queue` can be used by first creating an instance of the class. This will create an unbounded queue by default, that is, a queue with no size limit.

For example:

```
...
# created an unbounded queue
queue = Queue()
```

A maximum capacity can be set on a new `Queue` via the `maxsize` constructor augment.

For example:

```
...
# created a queue with a maximum capacity
queue = Queue(maxsize=100)
```

Items can be added to the queue via a call to the `put()` method, for example:

```
...
# add an item to the queue
queue.put(item)
```

If a size limited queue becomes full, new items cannot be added and calls to `put()` will block until space becomes available.

Items can be retrieved from the queue by calls to `get()`.

For example:

```
...
# get an item from the queue
item = queue.get()
```

By default, the call to `get()` will block until an item is available to retrieve from the queue.

Next, let's consider how we might share a queue between threads.

Share Data Between Threads via a Queue

A queue can be created then shared between multiple threads.

First an instance of the queue must be created as per normal.

```
...
# create a queue
queue = Queue()
```

It can then be shared between threads, such as an argument to a target task function.

For example:

```
# function executed in a new thread
def task(queue):
    # ...
```

Once shared, we may configure one or multiple threads to put data on to the queue.

These are producers threads, in a producer-consumer pattern of worker threads.

For example:

```
...
# loop over data
for i in range(100):
    # add a data item to the queue
    queue.put(i)
```

We may then configure one or multiple other threads to read data from the queue.

These are consumer threads.

For example:

```
...
# loop forever
while True:
    # get an item of data from the queue
    data = queue.get()
    # ...
```

Now that we know about queues and how we might share them with threads, let's explore a worked example.

Example of Sharing Data via a Queue

We can explore an example of sharing data between two threads using a queue.

In this example we will define a custom task function that takes the queue as an argument, generates some data and puts that data on the queue. It is a producer thread.

The main thread will create the queue, share it with the new thread and then block waiting for data to arrive on the queue. The main thread will be the consumer thread.

The complete example of sharing data between threads using a queue is listed below.

```
# SuperFastPython.com
# example of threads sharing data using a queue
from time import sleep
from threading import Thread
from queue import Queue

# custom function to be executed in a new thread
def task(shared_queue):
    # block for a moment
    sleep(1)
    # prepare some data
    item = 'Hello from a new thread'
    # share the data via the queue
    shared_queue.put(item)

# protect the entry point
if __name__ == '__main__':
    # create the shared queue
    queue = Queue()
    # create a new thread
    thread = Thread(target=task, args=(queue,))
    # start the thread
    thread.start()
    # block and wait for data via queue
    data = queue.get()
    # report data from the queue
    print(data)
```

Running the example first creates the shared `Queue` object in the main thread.

A new `Thread` object is then created and configured to execute our custom `task()` function and pass the queue as an argument.

The new thread is then started, then the main thread block, waiting to retrieve one object from the shared queue.

The new thread executes, first blocking for a second, then preparing string data and putting

it on the shared queue.

The data arrives on the queue and is consumed by the main thread, which reports the value.

This highlights how we can easily share data between threads by putting and getting values on a queue.

```
Hello from a new thread
```

Lesson Review

Takeaways

Well done, you made it to the end of the lesson.

- You know how to share data between threads using global variables.
- You know how to return values from threads using instance variables.
- You know how to share data between threads using thread-safe queues.

Exercise

Your task for this lesson is to use what you have learned about sharing data between threads.

Develop a small program where a task is split into subtasks and executed by new threads. The results of the task must be collected in the main thread which will wait for all tasks to complete and all results to be gathered before reporting a final result.

If you are stuck for ideas, generate a random number and block in each task, gather the random numbers in the main thread and sum their values.

Implement the program in a few different ways to explore the different approaches to sharing data between threads. For example, try using a container in a global variable, try storing values as instance variables, and try using a queue.

Sharing data between threads is central to almost all programs that implement concurrency via the `threading` module. This exercise provides the practice that you need to use the techniques confidently in your future projects.

Share your results online on Twitter, LinkedIn, GitHub, or similar.

Send me the link to your results, I'd love to see what you come up with.

You can send me a message directly via:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

Or share it with me on Twitter via @SuperFastPython.

Further Reading

This section provides resources for you to learn more about the topics covered in this lesson.

- Queue (abstract data type), Wikipedia.
  https://en.wikipedia.org/wiki/Queue_(abstract_data_type)
- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html
- `threading` - Thread-based parallelism.
  https://docs.python.org/3/library/threading.html
- `queue` - A synchronized queue class.
  https://docs.python.org/3/library/queue.html

Next

In the next lesson, we will discover how we can execute tasks with reusable worker threads.

Lesson 06: Run Tasks with Workers in Thread Pools

In this lesson, you will discover how you can execute ad hoc tasks with worker threads in the `ThreadPool` class.

After completing this lesson, you will know:

- What are thread pools and when to use them in your programs.
- How to create and configure new thread pool.
- How to execute single and multiple tasks using thread pool.
- How to use callback functions to handle results and errors asynchronously.
- How to interact with tasks issued asynchronously in thread pool.

Let's get started.

What Are Thread Pools

The `Thread` class can execute one-off tasks by creating a new instance of the class and specifying the function to execute via the `target` argument, as we have seen.

This is useful for running one-off ad hoc tasks in a separate threads, although it becomes cumbersome when we have many tasks to run.

Each thread that is created requires the application of resources (e.g. an instance of the Python interpreter and a memory for the thread's stack space). The computational costs for setting up threads can become expensive if we are creating and destroying many threads over and over for ad hoc tasks.

Instead, we would prefer to keep worker threads around for reuse if we expect to run many ad hoc tasks throughout our program.

This can be achieved using a thread pool.

A thread pool is a programming pattern for automatically managing a pool of worker threads.

The pool is responsible for a fixed number of threads.

- It controls when they are created, such as upfront or when they are needed.
- It controls how many tasks each worker can execute before being replaced.

- It also controls what workers should do when they are not being used, such as making them wait without consuming computational resources.

Thread pools can provide a generic interface for executing ad hoc tasks with a variable number of arguments, much like the `target` attribute on the `Thread` class, but does not require that we choose a thread to run the task, start the thread, or wait for the task to complete.

How to Use Thread Pools in Python

Python provides a thread pool via the `ThreadPool` class.

It allows tasks to be submitted as functions to the thread pool to be executed concurrently.

The `ThreadPool` class is in the `multiprocessing` module, rather than the `threading` module because it provides a thread-based wrapper for the `Pool` class that provides a process pool.

Because `ThreadPool` is a wrapper for `Pool`, it does have some aspects that can be confusing initially, such as the number of workers are called `processes`. We will see this in the next section.

To use the `ThreadPool` class, we must first create and configure an instance of the class.

For example:

```
...
# create a thread pool
pool = ThreadPool(...)
```

Once configured, tasks can be submitted to the pool for execution using blocking and asynchronous versions of `apply()` and `map()`.

Once we have finished with the `ThreadPool`, it can be closed and resources used by the pool can be released and prevents any further tasks from being issued to the pool.

For example:

```
...
# close the thread pool
pool.close()
```

The underlying `Pool` class offers the ability to forcefully terminate all workers, regardless of whether they are executing tasks or not. This can be achieved via the `terminate()` method.

Threads cannot be terminated in the same manner as processes. As such, the `terminate()` has the same effect as `close()` on the `ThreadPool` class.

After closing the pool, we may want to then wait for all tasks in the pool to finish.

This can be achieved by calling the `join()` method on the pool.

For example:

```
...
# wait for all issued tasks to complete
pool.join()
```

We can use the `ThreadPool` via the context manager interface, which will confine all usage of the pool to a code block and ensure the pool is closed once we are finished using it.

For example:

```
...
# create and configure the thread pool
with ThreadPool() as pool:
    # issue tasks to the pool
    # ...
# close the pool automatically
```

Next, let's explore how we might configure the `ThreadPool`.

How to Configure the `ThreadPool` Class

The `ThreadPool` is configured via the class constructor.

All arguments to the constructor are optional, therefore it is possible to create a thread pool with all default configuration by providing no arguments.

For example:

```
...
# create a thread pool with a default configuration
pool = ThreadPool()
```

The first argument is `processes` that specifies the number of worker threads (not processes) to create and manage within the pool.

By default this equals the number of logical CPUs in our system.

For example, if we had 4 physical CPU cores with hyperthreading, this would mean we would have 8 logical CPU cores and this would be the default number of workers in the thread pool.

In practice, we may have many more worker threads than we have CPU cores in our system, such as hundreds or thousands of threads.

For example:

```
...
# create a thread pool with a given number of workers
pool = ThreadPool(processes=100)
```

Because `processes` is the first argument, we don't have to specify it by name.

For example:

```
...
# create a thread pool with a given number of workers
pool = ThreadPool(100)
```

We can also configure the pool to initialize each worker thread with a custom initialization function via the `initializer` argument.

Next, let's take a look at how we might execute tasks in the pool.

How to Execute Tasks Synchronously and Asynchronously

We can execute one-off tasks in the `ThreadPool` with the `apply()` method.

It takes the name of the function to execute and any arguments to the function as a tuple to the `args` argument. It will block until the task is complete.

For example:

```
...
# execute a function call by the thread pool
result = pool.apply(task, args=(arg1, arg2))
```

The `apply_async()` method is more useful for issuing one-off tasks to the `ThreadPool`.

Like the `apply()` method, it takes the name of the function to execute and any arguments, but does not block. Instead, it returns immediately with an `AsyncResult` object that provides a handle on the running task.

For example:

```
...
# execute a function call by the pool without blocking
async_result = pool.apply_async(task, args=(arg1, arg2))
```

We can explore how to issue one-off tasks asynchronously with a worked example.

```
# SuperFastPython.com
# example of executing an async one-off task
from multiprocessing.pool import ThreadPool

# custom function to be executed in a worker thread
def task():
    # report a message
    print('This is a worker thread')

# protect the entry point
if __name__ == '__main__':
    # create the thread pool
```

```
    with ThreadPool() as pool:
        # issue a task asynchronously
        async_result = pool.apply_async(task)
        # wait for the task to complete
        async_result.wait()
```

Running the example issues one call to the `task()` function to the `ThreadPool`.

The caller blocks until the task is complete.

```
This is a worker thread
```

The `ThreadPool` provides a concurrent version of the built-in `map()` function for issuing tasks.

The `ThreadPool` `map()` method takes the name of a target function and an iterable. A task is created in the pool to call the target function for each item in the provided iterable. It returns an iterable over the return values from each call to the target function.

Unlike the built-in `map()` function, the `ThreadPool` `map()` method only takes a single iterable of arguments for the target function.

The iterable is first traversed and all tasks are issued at once. An iterable of return values is returned from `map()` once all tasks have completed. This means that the `map()` method blocks until all tasks are done.

For example:

```
...
# iterates return values from the issued tasks
for result in pool.map(task, items):
    print(result)
```

A `chunksize` argument can be specified to split the tasks into groups which may be sent to each worker thread to be executed in batch.

For example:

```
...
# iterates return values from the issued tasks
for result in pool.map(task, items, chunksize=10):
    print(result)
```

An efficient value for the `chunksize` argument can be found via some trial and error.

We can explore how to use `map()` to execute the same function with different arguments concurrently with a worked example, listed below.

```
# SuperFastPython.com
# example executing multiple tasks with different args
from multiprocessing.pool import ThreadPool
```

```
# custom function to be executed in a worker thread
def task(arg):
    # report a message
    print(f'Worker task got {arg}')
    # return a value
    return arg * 2

# protect the entry point
if __name__ == '__main__':
    # create the thread pool
    with ThreadPool() as pool:
        # issue multiple tasks and handle return values
        for result in pool.map(task, range(10)):
            # report result
            print(result)
```

Running the example first issues 10 tasks to the `ThreadPool`.

The tasks complete as workers become available and report a message.

Once all tasks are completed, the `map()` method returns an iterable of return values, which is then traversed in the main thread.

```
Worker task got 0
Worker task got 1
Worker task got 2
Worker task got 3
Worker task got 4
Worker task got 5
Worker task got 6
Worker task got 7
Worker task got 8
Worker task got 9
0
2
4
6
8
10
12
14
16
18
```

If the target function takes multiple arguments, the `starmap()` method can be used. It takes an argument that is an iterable of iterables, where each item provides the arguments for one

call to the target function.

For example:

```
...
# prepare an iterable of iterables for each task
items = [(1,2), (3,4), (5,6)]
# iterates return values from the issued tasks
for result in pool.starmap(task, items):
    print(result)
```

Both the `map()` and `starmap()` methods have asynchronous versions `map_async()` and `starmap_async()` that do not block and instead return immediately with an `AsyncResult` object.

A problem with the `map()` method is that it traverses the provided iterable and issues all tasks to the `ThreadPool` immediately.

This can be a problem if the iterable contains many hundreds or thousands of items.

As an alternative, the `ThreadPool` provides the `imap()` method which is a lazy version of `map()` for applying a target function to each item in an iterable one-at-a-time as works become available. Additionally, return values are yielded from the returned iterable in order as they are completed, rather than after all tasks are completed.

For example:

```
...
# iterates results as tasks are completed in order
for result in pool.imap(task, items):
    # ...
```

This can be helpful if the program is required to be responsive and report results as tasks are completed, in the order they were issued.

The `imap_unordered()` method is the same as `imap()`, except that return values are yielded in the order that tasks are completed, rather than the order they were issued, making it even more responsive.

Next, let's look at how we might handle results and errors in asynchronous tasks with callback functions.

How to Use Callbacks to Handle Results and Errors

The `ThreadPool` class supports custom callback functions.

A callback is a function that is first registered and then called automatically by the `ThreadPool` on some event.

Callbacks are only supported in the `ThreadPool` when issuing tasks asynchronously with any of the following functions:

- `apply_async()`: For issuing a single task asynchronously.
- `map_async()`: For issuing multiple tasks with a single argument asynchronously.
- `starmap_async()`: For issuing multiple tasks with multiple arguments asynchronously.

Callback functions are called in two situations:

- With the results of a task when the task finishes successfully.
- When an exception or error is raised in a task and is not handled.

A result callback can be specified via the `callback` argument.

The argument specifies the name of a custom function to call with the result of asynchronous task or tasks.

For example, if `apply_async()` is configured with a callback, then the callback function will be called with the return value of the task function that was executed.

```
# result callback function
def result_callback(result):
    print(result)

...
# issue a single task
result = apply_async(..., callback=result_callback)
```

Alternatively, if `map_async()` or `starmap_async()` are configured with a result callback, then the callback function will be called with an iterable of return values from all tasks issued to the thread pool.

```
# result callback function
def result_callback(result):
    # iterate all results
    for value in result:
        print(value)

...
# issue a single task
result = map_async(..., callback=result_callback)
```

An error callback can be specified via the `error_callback` argument.

The argument specifies the name of a custom function to call with the error raised in an asynchronous task.

NOTE: The first task to raise an error will be called, not all tasks that raise an error.

For example, if `apply_async()` is configured with an error callback, then the callback function will be called with the error raised in the task.

```
# error callback function
def custom_callback(error):
```

```
    print(error)

...
# issue a single task
result = apply_async(..., error_callback=custom_callback)
```

We can explore how to use a result callback with the thread pool when issuing tasks via the `apply_async()` method.

In this example we will define a task that generates a random number, reports the number, blocks for a moment, then returns the value that was generated. A callback function will be defined that receives the return value from the task function and reports the value.

The complete example is listed below.

```
# SuperFastPython.com
# example of a callback function for a one-off task
from random import random
from time import sleep
from multiprocessing.pool import ThreadPool

# result callback function
def result_callback(return_value):
    # report a message
    print(f'Callback got: {return_value}')

# custom function to be executed in a worker thread
def task(ident):
    # generate a value
    value = random()
    # report a message
    print(f'Task {ident} with {value}')
    # block for a moment
    sleep(value)
    # return the generated value
    return value

# protect the entry point
if __name__ == '__main__':
    # create and configure the thread pool
    with ThreadPool() as pool:
        # issue tasks to the thread pool
        result = pool.apply_async(task, args=(0,),
            callback=result_callback)
        # close the thread pool
        pool.close()
```

```
        # wait for all tasks to complete
        pool.join()
```

Running the example first starts the `ThreadPool` with the default configuration.

Then the task is issued to the `ThreadPool`. The main thread then closes the pool and then waits for the issued task to complete.

The task function executes, generating a random number, reporting a message, blocking and returning a value.

The result callback function is then called with the generated value, which is then reported.

The task ends and the main thread wakes up and continues on, closing the program.

NOTE: Results will vary each time the program is run given the use of random numbers.

```
Task 0 with 0.9971361794383538
Callback got: 0.9971361794383538
```

Next, let's look at how we might check the status and get results from an asynchronous task executed by the multiprocessing pool.

How to Interact with Asynchronous Tasks

The `AsyncResult` represents a result from a task issued asynchronously to the `ThreadPool`.

It provides a mechanism to check the status, wait for, and get the result for a task executed asynchronously in the pool.

An instance of the `AsyncResult` class is returned for each task submitted by the `apply_async()`, `map_async()`, and `starmap_async()` methods.

For example, a call to `map_async()` for a function `task()` with an iterable of ten items, will return a list of ten instances of the `AsyncResult` class.

For example:

```
...
# submit tasks to the pool in a non-blocking manner
async_result = pool.map_async(task, items)
```

For a single task represented via an `AsyncResult` object, we can check if the task is completed via the `ready()` method which returns `True` if the task is completed (successfully or with an error) or `False` otherwise.

For example:

```
...
# check if a task is done
if async_result.ready():
    # ...
```

A task may be completed successfully or may raise an Error or Exception. We can check if a task completed successfully via the `successful()` method. If the task is still running, it raises a `ValueError`.

For example:

```
...
# check if a task was completed successfully
if async_result.successful():
    # ...
```

We can wait for a task to complete via the `wait()` method.

If called with no argument, the method call will block until the task finishes. A `timeout` can be provided so that the method will after a fixed number of seconds if the task has not completed.

For example:

```
...
# wait 10 seconds for the task to complete
async_result.wait(timeout=10)
```

Finally, we can get the result from the task via the `get()` method.

If the task is finished, then `get()` will return immediately. If the task is still running, a call to `get()` will not return until the task finishes and returns the result.

For example:

```
...
# get the result of a task
result = async_result.get()
```

If an exception was raised while the task was being executed, it is re-raised by the `get()` method in the parent thread.

Lesson Review

Takeaways

Well done, you made it to the end of the lesson.

- You know what thread pools are and when to use them in your programs.
- You know how to create and configure new thread pool.
- You know how to execute single and multiple tasks using thread pool.
- You know how to use callback functions to handle results and errors asynchronously.
- You know how to interact with tasks issued asynchronously in thread pool.

Exercise

Your task for this lesson is to use your knowledge of thread pools for executing ad hoc tasks.

Develop an example that involves calling a task function multiple times with different arguments. The task should perform some action and takes a variable amount of time to complete.

If you're stuck for ideas, the task may generate a random number between 0 and 1 then block for a fraction of a second and return the generated number.

Execute the tasks using the `ThreadPool` via the `map()` method.

Update the example to use the `imap()` method and make the main thread responsive, reporting results as tasks are completed.

Finally, update the example one more time to execute tasks using the `imap_unordered()` method and have the main thread report results in the order that tasks are completed.

Thread pools are a helpful way of executing ad hoc tasks concurrently in your Python programs. With a few lines of code you can easily transform a for-loop or list comprehension to execute concurrently. This exercise will improve your confidence in making these types of changes in future projects.

Share your results online on Twitter, LinkedIn, GitHub, or similar.

Send me the link to your results, I'd love to see what you come up with.

You can send me a message directly via:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

Or share it with me on Twitter via @SuperFastPython.

Further Reading

This section provides resources for you to learn more about the topics covered in this lesson.

- Thread pool, Wikipedia.
  https://en.wikipedia.org/wiki/Thread_pool
- Built-in Functions.
  https://docs.python.org/3/library/functions.html
- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html
- `random` - Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- `threading` - Thread-based parallelism.
  https://docs.python.org/3/library/threading.html
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html

Next

In the next and final lesson, we will explore the more advanced topic of how to stop and kill new threads.

Lesson 07: Close, Stop, and Kill Threads

In this lesson, we will explore how to wait for, gracefully stop, and forcefully kill threads.

After completing this lesson, you will know:

- Alternatives to stopping a thread such as waiting or making it a daemon.
- How to use a thread to close itself.
- How to trigger a thread to stop gracefully, allowing clean-up.
- How to forcefully kill a thread on demand.

Let's get started.

Alternatives to Stopping a Thread

Stopping a thread can be a drastic step.

Before we look at how we can intervene and cause a thread to stop itself, be stopped or be killed, let's consider some alternatives.

Two common alternatives include:

1. Waiting for the thread to finish.
2. Making the thread a daemon thread.

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

Wait for Thread to Finish

Rather than stopping a thread, we can wait for it to finish.

A thread can be joined in Python by calling the `join()` method.

For example:

```
...
# join a thread
thread.join()
```

We discovered how to join a thread in *Lesson 02: Create and Start New Threads* when learning about how to create new threads for the first time.

Joining a thread has the effect of blocking the current thread until the target thread that has been joined has terminated.

The target thread that is being joined may terminate for a number of reasons, such as:

- Finishes executing it's target function.
- Finishes executing it's `run()` method if it extends the `Thread` class.
- Raised an error or exception.

Once the target thread has finished, the `join()` method will return and the current thread can continue to execute.

The `join()` method requires that we have a `Thread` object for the thread we wish to join.

This means that if we created the thread, we may need to keep a reference to the `Thread` object. Alternatively, we can use the `enumerate()` module function to enumerate through all active threads and locate the thread we wish to join by name or thread identifier.

The `join()` method also takes a `timeout` argument that specifies how long the current thread is willing to wait for the target thread to terminate, in seconds.

Once the `timeout` has expired and the target thread has not terminated, the `join()` thread will return.

```
...
# join the thread with a timeout
thread.join(timeout=10)
```

When using a timeout, it will not be clear whether the `join()` method returned because the target thread terminated or because of the timeout.

Therefore, we can call the `is_alive()` function to confirm the target thread is no longer running.

For example:

```
...
# join the thread with a timeout
thread.join(timeout=10)
# check if the target thread is still running
if thread.is_alive():
    # timeout expired, thread is still running
else:
    # thread has terminated
```

Next, let's look at another alternative to stopping a thread, which is to make it a daemon.

Make a Thread a Daemon

A thread may be configured to be a daemon thread.

Daemon threads is the name given to background threads. By default, threads are non-daemon threads.

A Python program will only exit when all non-daemon threads have finished. For example, the main thread is a non-daemon thread. This means that daemon threads can run in the background and do not have to finish or be explicitly excited for the program to end.

By making a thread a daemon thread, we don't have to explicitly stop the thread or wait for it to finish. We can just exit the program at any time.

The downside is that it does not allow us to terminate a task executing in a new thread without terminating the whole application.

We can determine if a thread is a daemon thread via the `daemon` attribute.

```
...
# report the daemon attribute
print(thread.daemon)
```

We learned about daemon threads in *Lesson 03: Configuring and Interacting with Threads* when configuring threads.

A thread can be configured to be a daemon by setting the `daemon` argument to `True` in the `Thread` constructor.

For example:

```
...
# create a daemon thread
thread = Thread(daemon=True)
```

We can also configure a thread to be a daemon thread after it has been constructed via the `daemon` property.

For example:

```
...
# configure the thread to be a daemon
thread.daemon = True
```

Next, let's explore how we might use a thread to close itself.

How a Thread Can Close Itself

A thread will stop when it exits, and it has complete control over when this happens.

Generally, a new thread will close when the `run()` function of the `Thread` class returns.

This can happen in one of two ways:

1. The `run()` function returns normally.
2. The `run()` function raises an uncaught error or exception.

We can return or raise an uncaught exception to close a thread, and this can be implemented in a few ways, such as:

- Call `return` from a target task function.
- Call `sys.exit()`.
- Raise an `Error` or `Exception`.

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

Close Thread By Returning

The `run()` function of the `Thread` class will execute our target function in a new thread of execution.

Consider the case where we create a new thread and configure it to execute a custom `task()` function via the `target` argument. The thread is then started by calling the `start()` function.

For example:

```
# task function
def task():
    # ...

...
# create a new thread to run a custom function
thread = threading.Thread(target=task)
# start the thread
thread.start()
```

In this case, the `start()` function executes the `run()` function of the `Thread` class in a new thread and returns immediately.

The `run()` function of the `Thread` class will call our `task()` function. Our task function will return eventually, then the run function will return and the thread will terminate.

This is the normal usage of a thread.

We can choose to close the thread any time from within our task function.

This can be achieved by returning from the task function.

For example:

```
# task function
def task():
    # execute task
    ...
    # determine if the thread should be closed
```

```
    if condition:
        return
```

This will terminate the thread.

The trigger to close the thread may come from another thread, such as a shared global variable or an `Event` object.

Next, let's explore how we might close a thread from a task by calling `sys.exit()`.

Close Thread By Calling `sys.exit()`

Another approach is to call the `sys.exit()` function at any point within our `task()` function or in the functions it calls.

The `sys.exit()` function takes an argument to specify whether the thread is exiting normally (0, the default), or with error (1, or anything other than 0).

The `sys.exit()` function will raise a `SystemExit` exception which will not be caught and will terminate the new thread.

For example:

```
# task function
def task():
    # execute task
    ...
    # determine if the thread should be closed
    if condition:
        exit()
```

This approach is helpful if we are deep down the call-graph of custom functions and the return statement is not convenient.

We can demonstrate this with a worked example, listed below.

```
# SuperFastPython.com
# example thread stopping itself via calling sys.exit()
from time import sleep
from threading import Thread
import sys

# custom function to be executed in a new thread
def task():
    # wait a moment
    sleep(1)
    # report a message
    print('Thread exiting now')
    # exit the thread
```

```
    sys.exit(0)
    # this is never reached
    print('Can you see me?')

# protect the entry point
if __name__ == '__main__':
    # create a new thread
    thread = Thread(target=task)
    # start the thread
    thread.start()
    # wait a moment
    sleep(2)
    # report a message
    print('Main all done')
```

Running the example first configures a new thread to execute the custom task function.

The new thread is then started and the main thread blocks for a few seconds.

The new thread blocks for a moment, reports a message then calls the `sys.exit()` function to terminate the thread. The message after the call to the `sys.exit()` function is not reported because the line of code is not reached.

The main thread wakes up and continues on, reporting a message then terminating the program.

This highlights that a call to `sys.exit()` terminates the thread at the point it is called and does not have an effect on other threads, like the main thread.

```
Thread exiting now
Main all done
```

Next, let's explore how we might close a thread from a task by raising an exception

Close Thread By Exception

Another approach is to raise an `Error` or `Exception` in the target function or any called function.

If the `Error` or `Exception` is uncaught it will unravel the call graph of the thread, then terminate the thread.

For example:

```
# task function
def task():
    # execute task
    ...
    # determine if the thread should be closed
```

```
    if condition:
        raise Exception('Close')
```

The downside of this approach is that the default handler for uncaught exceptions will report the exception to the terminal. This can be changed by specifying a handler via `threading.excepthook`.

Exception handling in new threads was introduced in *Lesson 03: Configuring and Interacting with Threads*.

We can explore how a thread can close itself by raiding an error or exception.

In this example, we will define a custom task that loops forever and generates a random number each iteration and reports it. If the number is above a specific threshold, a message is reported and an exception is raised.

The main thread registers a custom exception handler for new threads to report the exception message only. It starts the new thread and waits for it to terminate.

The complete example is listed below.

```
# SuperFastPython.com
# example of a thread closing itself via an exception
from random import random
from time import sleep
from threading import Thread
import threading

# handle exceptions in new thread
def handler(args):
    print(f'Got: {args.exc_value}')

# task function
def task():
    # loop forever
    while True:
        # generate a random value between 0 and 1
        value = random()
        print(f'.{value}')
        # block
        sleep(value)
        # check if we should close the thread
        if value > 0.9:
            print('Closing thread')
            raise Exception('Stop now!')

# protect the entry point
```

```
if __name__ == '__main__':
    # register a handler for exception in a new thread
    threading.excepthook = handler
    # create and configure the new thread
    thread = Thread(target=task)
    # start the new thread
    thread.start()
    # wait for the thread to terminate
    thread.join()
    # main continues on
    print('Main continuing on...')
```

Running the example first registers a custom handler for exceptions raised in a new thread.

The main thread then starts the new thread to execute our custom task function, then the main thread blocks until the new thread has terminated.

The new thread loops forever, generating and reporting a random value each iteration. If the value is above the threshold, the thread raises an exception.

The exception is not caught and bubbles up to the top-level of the thread, which is the `run()` function. The exception is then reported on the console using the custom exception hook handler.

The main thread then continues on with the application, and reports a message, confirming that the main thread remained unaffected by the raised exception.

Below is a truncated sample output of the program.

NOTE: Results will vary each time the program is run given the use of random numbers.

```
.0.6353448787481277
.0.008743635639916203
.0.6785237214497055
.0.8452186203733817
.0.07177669260268127
.0.5688053356787506
.0.5941374895230044
.0.3623198862193585
.0.6992386131291078
.0.9594335278024986
Closing thread
Got: Stop now!
Main continuing on...
```

Now that we have seen how a thread may close itself, next, let's explore how we might trigger a thread to close from another thread.

How to Gracefully Stop a Thread

We may run a task in a new thread, then later decide to stop the task.

This may be for many reasons, such as:

- The result from the task is no longer required.
- The application is shutting down.
- The outcome from the task has gone astray.

Python does not provide the ability in the threading API to stop a thread.

Instead, we can add this functionality to our code directly.

A thread can be stopped using a shared boolean variable such as a `Event`.

How to Use an `Event` to Stop a Thread

An `Event` is a thread-safe boolean variable flag that can be either set or not set. It can be shared between threads and checked and set without fear of a race condition.

Event objects were introduced in *Lesson 04: Synchronize and Coordinate Threads*.

A new event can be created and then shared between threads, for example:

```
...
# create a shared event
event = Event()
```

The `Event` is created in the *not set* or `False` state.

We may have a task in a custom function that is run in a new thread. The task may iterate, such as in a while-loop or a for-loop.

For example:

```
# custom task function
def task():
    # perform task in iterations
    while True:
        # ...
```

We can update our task function to check the status of an event each iteration.

If the event is set true, we can exit the task loop or return from the `task()` function, allowing the new thread to terminate.

The status of the `Event` can be checked via the `is_set()` function.

For example:

```
# custom task function
def task():
    # perform task in iterations
```

```
    while True:
        # ...
        # check for stop
        if event.is_set():
            break
```

The main thread, or another thread, can then set the event in order to stop the new thread from running.

The event can be set or made `True` via the `set()` function.

For example:

```
...
# set the event
event.set()
# wait for the new thread to stop
thread.join()
```

Now that we know how to stop a Python thread, let's look at a worked example.

Example of Using an `Event` to Stop a Thread

We can develop an example that runs a function in a new thread.

The new task function will execute in a loop and once finished the new thread will terminate and the main thread will terminate.

Firstly, we can define the task function.

The function will take an `Event` object as an argument. It will execute a loop five times. Each iteration it will block for a second, then report a message in an effort to simulate work and show progress.

The task loop to check the status of the event each iteration, and if set to break the task loop.

Once the task is finished, the function will report a final message.

The complete example is listed below.

```
# SuperFastPython.com
# example of stopping a new thread gracefully
from time import sleep
from threading import Thread
from threading import Event

# custom function to be executed in a new thread
def task(shared_event):
    # execute a task in a loop
```

```
    for _ in range(5):
        # block for a moment
        sleep(1)
        # check for stop
        if shared_event.is_set():
            break
        # report a message
        print('Worker thread running...')
    print('Worker closing down')

# protect the entry point
if __name__ == '__main__':
    # create the shared event
    event = Event()
    # create and configure a new thread
    thread = Thread(target=task, args=(event,))
    # start the new thread
    thread.start()
    # block for a while
    sleep(3)
    # stop the worker thread
    print('Main stopping thread')
    event.set()
    # wait for the new thread to finish
    thread.join()
```

Running the example first creates the shared event, which is not set.

The main thread then creates and starts the new thread to execute the custom task function and passes the shared event an argument.

The main thread then blocks for a few seconds.

The new thread executes its task loop, blocking and reporting a message each iteration. It checks the event each iteration, which remains not set and does not trigger the thread to stop.

The main thread wakes up and then sets the event. It then joins the new thread, waiting for it to terminate.

The new thread checks the event which is discovered to be set (e.g, True). The thread breaks the task loop, reports a final message then terminates the new thread.

The main thread then terminates, closing the Python process.

```
Worker thread running...
Worker thread running...
Main stopping thread
```

```
Worker closing down
```

This highlights how we can stop a new thread gracefully, although it does require that the new thread check the status of a shared event frequently.

Next, let's explore how we might forcefully stop a new thread by killing it.

How to Forcefully Kill a Thread

Python does not provide an API to forcefully kill a new thread.

Forcefully terminating or killing a thread is a drastic action. You should strongly consider one of the alternatives proposed above.

Nevertheless, a thread can be terminated or killed by forcibly terminating or killing its parent process.

Recall that each thread belongs to a process. A process is an instance of the Python interpreter, and a thread is a thread of execution that executes code within a process. Each process starts with one default thread called the main thread.

Killing a thread via its parent process may mean that we will want to first create a new `Process` object in which to house any new threads that we may wish to forcefully terminate. This is to avoid terminating the main process.

How to Kill a Thread's Process

A process can be killed by calling the `terminate()` or `kill()` methods on the `Process` object.

Each process in Python has a corresponding instance of the `Process` class.

We can get the current `Process` object by calling the `current_process()` module function.

For example:

```
...
# get the current process
process = current_process()
```

Alternatively, we may hang onto the `Process` object if we create a new process in which to execute the thread we may wish to kill.

```
...
# create a new process
process = Process(...)
```

We can then call the `terminate()` method on the `Process` object to kill it.

This will send the process the `SIGTERM` signal which means *Signal Terminate*.

For example:

```
...
# kill the process
process.terminate()
```

Once the signal is received, the process will terminate, in turn terminating its child threads. It will not terminate any child process.

This signal can be caught and handled within the process, if we so desire. As such, it may be helpful we have clean-up tasks to perform prior to killing.

Alternatively, we may call the `kill()` function on the `Process` object to kill it.

This will send the `SIGKILL` signal to the process, which means *Signal Kill*.

Once this signal is received, the process will terminate like `SIGTERM`, killing all child threads and not killing any child processes.

For example:

```
...
# kill the process
process.kill()
```

Unlike `SIGTERM`, the `SIGKILL` cannot be handled within the process. It is a more forceful way of terminating the process.

Next, let's explore an example of killing a thread via its parent process.

Example of Killing a Thread's Process

We can explore how to kill a thread via its parent process by calling the `terminate()` method.

In this example, we will first create a new child process. The child process will then execute our task in the main thread of the process. The task will loop forever, each iteration it will block for one second then report a message. It provides no way to gracefully terminate. Once the thread is started, we will wait for some time, then forcefully kill the thread via its parent process, wait a moment longer and then terminate the initial process.

The complete example is listed below.

```
# SuperFastPython.com
# example of killing a thread via its parent process
from time import sleep
from multiprocessing import Process

# custom function to be executed in a new thread
def task():
    # run for ever
    while True:
        # block for a moment
```

```
        sleep(1)
        # report a message
        print('Task is running', flush=True)

# protect the entry point
if __name__ == '__main__':
    # create a new process with a new thread
    process = Process(target=task)
    # start the new process and new thread
    process.start()
    # wait a while
    sleep(5)
    # kill the new thread via the new process
    print('Killing the thread via its process')
    process.terminate()
    # wait a while
    sleep(2)
    # all done
    print('All done, stopping')
```

Running the example first created a new process and then started a new main thread to execute our `task()` function.

Our task function would run forever unless the parent process is forcefully killed.

The first process blocks for a moment to allow the new task to run for a while. It then wakes up and kills the task by terminating the parent thread.

This sends the `SIGTERM` signal terminated to the new process, which is not handled by the process and terminates the process and its child main thread immediately.

The main thread of the first process then runs a little longer, blocks for a moment, reports a final message and it also terminates.

This shows that we can kill a new thread by killing its parent process and it has no effect on other threads running in separate processes.

```
Task is running
Task is running
Task is running
Task is running
Killing the thread via its process
All done, stopping
```

Lesson Review

Takeaways

Well done, you made it to the end of the lesson.

- You know about alternatives to stopping a thread such as waiting or making a it daemon.
- You know how to use a thread to close itself.
- You know how to trigger a thread to stop gracefully, allowing clean-up.
- You know how to forcefully kill a thread on demand.

Exercise

Your task for this lesson is to use what you have learned about how to stop and kill a thread.

Develop an example that involves calling a task function that performs some typical activity, such as a loop that blocks each iteration.

If you're stuck for ideas, the task may generate a random number between 0 and 1 then block for a fraction of a second.

The first version should wait for the task to complete.

Explore different ways to stop the task, such as from within the task with a condition or by raising an exception. Update the example to try triggering the task to stop via an `Event`. Finally try killing the thread via its parent process.

It is important to be familiar and confident with the different ways to stop a new thread as you will almost certainly require this functionality in your own programs.

Share your results online on Twitter, LinkedIn, GitHub, or similar.

Send me the link to your results, I'd love to see what you come up with.

You can send me a message directly via:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

Or share it with me on Twitter via @SuperFastPython.

Further Reading

This section provides resources for you to learn more about the topics covered in this lesson.

- `sys` - System-specific parameters and functions.
  https://docs.python.org/3/library/sys.html
- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html

- `random` - Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- `threading` - Thread-based parallelism.
  https://docs.python.org/3/library/threading.html
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html

Next

This was the last lesson, next we will take a look back at how far we have come.

Conclusions

Look Back At How Far You've Come

Congratulations, you made it to the end of this 7-day course.

Let's take a look back and review what you now know:

- You discovered the difference between thread-based and process-based concurrency and the types of tasks that are well suited to the capabilities of the `threading` module.
- You discovered how to execute your own ad hoc functions concurrently using the `Thread` class.
- You discovered how to identify the main thread and the life-cycle of new threads.
- You discovered how to configure a new thread and get access to all running threads and query their status.
- You discovered how to coordinate and synchronize threads using mutex locks, semaphores, condition variables and the full suite of concurrency primitives.
- You discovered how to share data between threads using shared global variables and how to return values from new threads using instance variables.
- You discovered how to send and receive data between threads using thread-safe queues.
- You discovered how to create and configure thread pools to execute ad hoc tasks using reusable worker threads.
- You discovered how to handle results and errors and query the status of asynchronous tasks executed in thread pools.
- You discovered how to close threads, and how to trigger a graceful shutdown or forcefully kill new threads.

You now know how to use the `threading` module and bring thread-based concurrency to your project.

Thank you for letting me help you on your journey into Python concurrency.

Jason Brownlee, Ph.D.
SuperFastPython.com
2022.

Resources For Diving Deeper

This section lists some useful additional resources for further reading.

APIs

- Concurrent Execution API - Python Standard Library.
  https://docs.python.org/3/library/concurrency.html
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html
- `threading` API - Thread-based parallelism.
  https://docs.python.org/3/library/threading.html
- `concurrent.futures` API - Launching parallel tasks.
  https://docs.python.org/3/library/concurrent.futures.html
- `asyncio` API - Asynchronous I/O.
  https://docs.python.org/3/library/asyncio.html

Books

- High Performance Python, Ian Ozsvald, et al., 2020.
  https://amzn.to/3wRD5MX
- Using AsyncIO in Python, Caleb Hattingh, 2020.
  https://amzn.to/3lNp2ml
- Python Concurrency with asyncio, Matt Fowler, 2022.
  https://amzn.to/3LZvxNn
- Effective Python, Brett Slatkin, 2019.
  https://amzn.to/3GpopJ1
- Python Cookbook, David Beazley, et al., 2013.
  https://amzn.to/3MSFzBv
- Python in a Nutshell, Alex Martelli, et al., 2017.
  https://amzn.to/3m7SLGD

Getting More Help

Do you have any questions?

Below provides some great places online where you can ask questions about Python programming and Python concurrency:

- Stack Overview.
  https://stackoverflow.com/
- Python Subreddit.
  https://www.reddit.com/r/python
- LinkedIn Python Developers Community.
  https://www.linkedin.com/groups/25827

- Quora Python (programming language).
  https://www.quora.com/topic/Python-programming-language-1

Contact the Author

You are not alone.

If you ever have any questions about the lessons in this book, please contact me directly:

- Super Fast Python - Contact Page
  https://SuperFastPython.com/contact/

I will do my best to help.

About the Author

Jason Brownlee, Ph.D. helps Python developers bring modern concurrency methods to their projects with hands-on tutorials. Learn more at SuperFastPython.com.

Jason is a software engineer and research scientist with a background in artificial intelligence and high-performance computing. He has authored more than 20 technical books on machine learning and has built, operated, and exited online businesses.

Figure 1: Photo of Jason Brownlee

Python Concurrency Jump-Start Series

Save days of debugging with step-by-step jump-start guides.

Python Threading Jump-Start.
https://SuperFastPython.com/ptj

Python ThreadPool Jump-Start.
https://SuperFastPython.com/ptpj

Python ThreadPoolExecutor Jump-Start.
https://SuperFastPython.com/ptpej

Python Multiprocessing Jump-Start.
https://SuperFastPython.com/pmj

Python Multiprocessing Pool Jump-Start.
https://SuperFastPython.com/pmpj

Python ProcessPoolExecutor Jump-Start.
https://SuperFastPython.com/pppej

Python Asyncio Jump-Start.
https://SuperFastPython.com/paj