SuperFastPython

7-Day Course

Python Multiprocessing Jump-Start

Develop Parallel Programs, Side-Step the GIL, and Use All CPU Cores

Jason Brownlee

Python Multiprocessing Jump-Start

Develop Parallel Programs, Side-Step the GIL, and Use All CPU Cores

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

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 only provide limited parallelism given the presence of a Global Interpreter Lock (GIL). This is widely known and leads to one of the most commonly asked questions in Python concurrency: "*How can we work around the GIL and achieve full parallelism in Python?*"

The answer is the `multiprocessing` module.

This guide was carefully designed to help Python developers (*like you*) to get productive with the `multiprocessing` module as fast as possible.

After completing all seven lessons, you will know how to bring process-based concurrency with the `multiprocessing` module 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 `multiprocessing` module allows you to execute code in parallel with Python.

Process-based concurrency provided by the `multiprocessing` module was developed to side-step the Global Interpreter Lock that limits Python threads. It allows Python developers to bring full parallelism to their programs and run code on all CPU cores.

This book provides a jump-start guide to get you productive with the Python `multiprocessing` 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 `multiprocessing` 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 CPU-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 `multiprocessing` 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**: Process-Based Concurrency
- **Lesson 02**: Create and Start Child Processes
- **Lesson 03**: Configuring and Interacting with Processes
- **Lesson 04**: Synchronize and Coordinate Processes
- **Lesson 05**: Share Data Between Processes
- **Lesson 06**: Run Tasks with Reusable Workers in Pools
- **Lesson 07**: Share Centralized Objects with Managers

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/pmj-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 `multiprocessing` 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 `multiprocessing` 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 `multiprocessing` module.
- How to execute your own ad hoc functions in parallel using the `Process` class.
- How to identify the main process, about parent and child processes, and the life-cycle of processes in Python.
- How to configure a new child process and access `Process` instance for running processes, kill them, and query their status such as their exit code and whether they are still running.
- How to coordinate and synchronize Python processes using mutex locks, semaphores, condition variables and the full suite of concurrency primitives.
- How to inherit and use global variables from parent processes when using the fork start method.
- How to share data between processes using shared `ctypes`, how to send and receive data using pipes and how to create producer and consumer processes using queues.
- How to create and configure multiprocessing pools to execute ad hoc tasks using reusable child worker processes.
- How to process results, handle errors, and query the status of asynchronous tasks executed in multiprocessing pools.
- How to create centralized Python objects that can be accessed and used in a process-safe manner using proxy objects.

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 process-based concurrency in Python.

Lesson 01: Process-Based Concurrency

In this lesson, you will discover process-based concurrency vs thread-based concurrency and the sweet-spot for using the `multiprocessing` module.

After completing this lesson, you will know:

- What a process is and how it compares to a thread, and the difference between concurrency and parallelism.
- What is threading in Python and what is the key limitation of thread-based concurrency.
- What is multiprocessing in Python and how it overcomes the limitation of threading.
- The key similarities and differences between thread-based and process-based concurrency.
- When to use the process-based concurrency in the `multiprocessing` 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 *processes*, *threads*, *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.

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 created and 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 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.

Now that we are on the same page when it comes to threads, processes, concurrency and parallelism, let's dive into more detail on threads and processes and when we should be using multiprocessing for process-based concurrency in Python.

Next, let's take a closer look at the `threading` module as the `multiprocessing` module that we will use later.

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`.

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 `Thread.start()` function 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 `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 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.

Next, let's take a closer look at the `multiprocessing` module that addresses the limitations of the `threading` module.

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 your 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 you type into your 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` instances, that mimics the `threading` module.

This includes a number of static module functions that allow the caller to get access to a `Process` instance 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 IPC

The `multiprocessing` module provides more capabilities, focused on Inter-Process Communication (IPC), the manner in which data is transmitted between processes.

This is more involved than sharing data between threads, which happens more simply within one process.

In addition to process-safe versions of queues, `Connection` are provided that permit connection between processes both on the same system and across systems. This provides a foundation for primitives such as the `Pipe` for one-way and two-way communication between processes.

The API provides a `Manager` API that creates a server process for managing centralized versions of Python objects. It then provides proxy objects that may be shared with other processes that allow them to interact with the centralized object in a seamless manner, while inter-process communication is occurring under the covers.

More simply, the shared `ctypes` API allows sharing of data primitives between processes with the `Value` and `Array` classes.

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

Now that we are familiar with the `threading` and `multiprocessing` modules, let's compare them.

Multiprocessing vs Threading

Now that we are familiar with the `multiprocessing` and `threading` modules from a high-level, let's review their similarities and differences.

Similarities

The Multiprocessing and Threading modules are very similar, let's review some of the most important similarities.

They are:

1. Both Modules are Used For Concurrency.
2. Both Have The Same API (mostly).
3. Both Support The Same Concurrency Primitives.

We've just reviewed these similarities and we don't need to go over them again in great detail.

Next, let's review the key differences between the two modules.

Differences

The `threading` and `multiprocessing` modules are also quite different, let's review some of the most important differences.

They are:

1. Native Threads vs. Native Processes.
2. Shared Memory vs. Inter-Process Communication.
3. Limited vs Full Parallelism (GIL).

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

Native Threads vs. Native Processes

Perhaps the most important difference between the 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.

Whereas a thread belongs to a process.

This is the central difference between the two modules and supports all other differences.

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

Threads are a lightweight construct.

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

Processes are a heavyweight construct.

- They have a larger memory footprint, e.g. a process is an instance of the Python interpreter.
- They are slow to allocate and create, e.g. fork or spawn start methods are used.
- They 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.

Shared Memory vs. Inter-Process Communication

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.

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.

Typically sharing data between processes requires explicit mechanisms, such as the use of a `Pipe` or a `Queue`.

As such, sharing state between threads is easy and lightweight, and sharing state between processes is harder and heavyweight.

Limited vs Full Parallelism (GIL)

Thread-based concurrency supports limited parallelism, whereas process-based concurrency supports full parallelism.

Multiple threads are subject to the global interpreter lock (GIL), whereas multiple child processes are not subject to the GIL.

The GIL is a programming pattern in the reference Python interpreter, e.g. CPython, the version of Python you download from Python.org.

It is a lock in the sense that it uses a mutual exclusion lock to ensure that only one thread of execution can execute instructions at a time within a Python process.

This means that although we may have multiple threads in our program, only one thread can execute at a time.

The GIL is used within each Python process, but not across processes. This means that multiple child processes can execute at the same time and are not subject to the GIL.

This has implications for the types of tasks best suited to each class.

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).

When to Use Multiprocessing

The `multiprocessing` module provides powerful and flexible concurrency, although is not suited for all situations where you need to run a background task.

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

Use the Multiprocessing for CPU-Bound Tasks

You should probably use the `multiprocessing` module for CPU-bound tasks.

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

The operations only involve data in main memory (RAM) or cache (CPU 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.

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.

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

Using processes via the `multiprocessing` module in Python is probably the best path toward achieving this end.

Don't Use Multiprocessing for IO-Bound Tasks (probably)

You can use the `multiprocessing` module for IO-bound tasks, although the `threading` module is likely a better fit.

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

The operations involve input and output (IO), and the speed of these operations is bound by the device, hard drive, or network connection. This is why these tasks are referred to as IO-bound. Doing 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 your operating system (the kernel), which will wait for the operation to complete. If this operation is the main focus for your CPU, such as executing in the main thread of your Python program, then your CPU is going to wait many milliseconds, or even many seconds, doing nothing.

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

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.

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.

Limitations of Multiprocessing for IO-Bound Tasks

Processes can be used for IO-bound tasks, although there are major limitations.

- Processes are heavyweight structures; each has at least a main thread.
- All data sent between processes must be serialized.
- The operating system may impose limits on the number of processes you can create.

When performing IO-operations, we very likely will need to move data between worker processes back to the main process. This may be costly if there is a lot of data as the data must be pickled at one end and unpickled at the other end. Although this data serialization is performed automatically under the covers, it adds a computational expense to the task.

Additionally, 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 (e.g. each managing a network connection), and this may not be feasible or possible with processes.

Nevertheless, the `multiprocessing` module may be appropriate for IO-bound tasks if the requirement on the number of concurrent tasks is modest (e.g. less than 100) and the data sharing requirements between processes is also modest (e.g. processes don't share much or any data).

Lesson Review

Takeaways

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

- You know what a process is and how it compares to a thread, and the difference between concurrency and parallelism.
- You know what is threading in Python and what is the key limitation of thread-based concurrency.
- You know what is multiprocessing in Python and how it overcomes the limitation of threading.
- You know the key similarities and differences between thread-based and process-based concurrency.

- You know when to use process-based concurrency in the `multiprocessing` module in your programs.

Exercise

Your task for this lesson is to take what you have learned about the `multiprocessing` 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 full parallelism provided by the `multiprocessing` module.

No need to share sensitive details of the project or technical details on how exactly multiprocessing 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 `multiprocessing` module, then think of applications or functionality you could develop for current or future projects that could make good use of multiprocessing.

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
- `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

- PEP 371 - Addition of the multiprocessing package to the standard library.
  https://peps.python.org/pep-0371/

Next

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

Lesson 02: Create and Start Child Processes

In this lesson, you will discover how to create and start new child processes with the `multiprocessing` module.

After completing this lesson, you will know:

- About the main process and the difference between parent and child processes.
- The life-cycle of Python processes including each step and their transitions.
- How to protect the entry point of the program and add freeze support.
- How to run a function in a new child process.
- How to extend the `Process` class to run custom code in a child process.

Let's get started.

Main Process in Python

A Python program starts 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 Process**: Default process created to execute a Python program, has the name *MainProcess*.
- **Main Thread**: Default thread created by a main process in a Python program, has the name *MainThread*.

When we refer to the *MainProcess* or *main process*, we mean the process started to run our program (and probably the thread executing the code of our main program), as opposed to other processes we may explicitly or implicitly create in our program.

What makes the main process unique is that it is the first process created when we run our program and the main thread of the process executes the entry point of our program.

As such the main process does not have a parent process.

Next, let's take a closer look at parent and child processes.

Parent and Child Processes

When we create new processes in the main process, they are referred to as child processes.

The creator of a child process is referred to as a parent process. A child process again may also be a parent process if it in turn creates one or more child processes.

- **Parent Process**: Has one or more child processes. May have a parent process, e.g. may also be a child.
- **Child Process**: Has a parent process. May have its own child processes, e.g. may also be a parent.

A parent process can access and interact with child processes and a child process can access and interact with its parent process.

A child process may inherit global variables from the parent process.

Next, let's look at the life-cycle of a process in Python.

Life-Cycle of a Process

A process in Python is represented as an instance of the `Process` class.

Once a process is started, the Python runtime will interface with the underlying operating system and request that a new native process be created. The `Process` instance then provides a Python-based reference to this underlying native process.

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

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

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

1. New Child Process.
2. Running Process. 1a. Blocked Process (optional).
3. Terminated Process.

A new process is a process that has been constructed and configured by creating an instance of the `Process` class.

A new process can transition to a running process by calling the `start()` method. This also creates and starts the main thread for the process that actually executes code in the process.

A running process may block in many ways if its main thread is blocked, 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. Alternatively, a process may explicitly block and wait for another process to terminate by calling its `join()` method.

After blocking, the process will run again.

Finally, a process may terminate once it has finished executing its code or by raising an error or exception. It may also terminate if a signal is raised in the process, such as `SIGKILL`. These signals may be sent by other programs, by the operating system, or from other Python processes.

A process cannot exit normally until:

- All non-daemon threads have terminated, including the main thread.
- All non-daemon child processes have terminated, including the main process.

Next, let's look at how to protect the entry point when using the `multiprocessing` module.

Protect the Entry Point When Using Multiprocessing

A Python program that uses the `multiprocessing` module should protect the entry point of the program.

This can be achieved by using an if-statement to check that the entry point is the top-level environment.

For example:

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

This will help to avoid a `RuntimeError` when creating child processes using the `'spawn'` start method, the default on Windows and MacOS. It is not required in other platforms like Linux, but is highly recommended to ensure the Python program is portable.

We will learn more about process start methods in a later lesson.

Additionally, it is a good practice to add freeze support as the first line of a Python program that uses the `multiprocessing` module.

Freezing a Python program is a process that transforms the Python code into C code for packaging and distribution.

When a program is frozen in order to be distributed, some features of Python are not included or disabled by default.

This is for performance and/or security reasons.

One feature that is disabled when freezing a Python program is multiprocessing.

That is, we cannot create new Python processes via `Process` instances when freezing our program for distribution.

Creating a process in a frozen application will result in a `RuntimeError`.

We can add support for multiprocessing in our program when freezing code via the `freeze_support()` function.

For example:

```
...
# enable support for multiprocessing
freeze_support()
```

This will have no effect on programs that are not frozen.

Protecting the entry point and adding freeze support together are referred to as the *main module* idiom when using `multiprocessing`.

Using this idiom is a best practice when using the `multiprocessing` module.

In fact, the `RuntimeError` we will get if we do not use this idiom will suggest using the idiom.

The full message in the error is as follows:

```
An attempt has been made to start a new process before
the current process has finished its bootstrapping
phase.

This probably means that you are not using fork to start
your child processes and you have forgotten to use the
proper idiom in the main module:

    if __name__ == '__main__':
        freeze_support()
        ...

The "freeze_support()" line can be omitted if the
program is not going to be frozen to produce an
executable.
```

Next, let's look at an example of running a custom function in parallel using a new child process.

How to Run a Function in a Child Process

Python functions can be executed in a separate process using the `Process` class.

To run a function in another process:

1. Create an instance of the `Process` class.

2. Specify the name of the function via the `target` argument.
3. Call the `start()` method.

First, we must create a new instance of the `Process` class and specify the function we wish to execute in a new process via the `target` argument.

```
...
# create a process
process = Process(target=task)
```

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

```
...
# create a process
process = Process(target=task, args=(arg1, arg2))
```

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

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

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

A new instance of the Python interpreter will be created and a new thread within the new process will be created to execute our target function.

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

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

And that's all there is to it.

We do not have control over when the process 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 process.

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 child process and wait for the process to terminate by joining it from the main process.

The complete example is listed below.

```
# SuperFastPython.com
# example of running a function in a new child process
```

```python
from time import sleep
from multiprocessing import Process

# custom function to be executed in a child process
def task():
    # block for a moment
    sleep(1)
    # report a message
    print('This is from another process', flush=True)

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

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

At some point a new instance of the Python interpreter is created that has a new thread which will execute our target function.

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

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

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

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

NOTE: We must explicitly flush the buffer by setting `flush=True` when using the `print()` function from child processes, otherwise the buffer will not flush until the child process terminates.

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

How to Extend the Process Class

We can also execute functions in a child process by extending the `Process` class and overriding the `run()` method.

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

For example:

```
# custom process class
class CustomProcess(Process):
    # ...
```

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

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 process.

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

We can define a class `CustomProcess` that extends the `Process` 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 child process.

The complete example is listed below.

```
# SuperFastPython.com
# example of extending the process class
from time import sleep
from multiprocessing import Process

# custom process class
class CustomProcess(Process):
    # override the run function
    def run(self):
        # block for a moment
        sleep(1)
        # report a message
        print('This is another process', flush=True)

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

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

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

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

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

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

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

Lesson Review

Takeaways

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

- You know about the main process, the main thread, as well as parent and child processes.
- You know about the life-cycle of the `Process` class in Python.
- You know how to protect the entry point of the program and add freeze support.
- You know how to run a custom function in a new child process.
- You know how to extend the `Process` class to run custom code in a child process.

Exercise

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

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 process. Record how long it takes to execute.

Compare the execution time between the serial and parallel 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 - parallel time
- ratio = serial time / parallel time

If it turns out that the parallel 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 parallel with the `multiprocessing` 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
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.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

Next

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

Lesson 03: Configuring and Interacting with Processes

In this lesson, you will discover how to configure new child processes and how to interact with processes in Python via the `multiprocessing` API.

After completing this lesson, you will know:

- How to configure the name of a process and whether it is a daemon.
- How to query the status of a process such as if it is running and its exit code.
- How to terminate and kill processes.
- How to get access to the current, parent and child processes.
- How to configure the start method and use a multiprocessing context.

Let's get started.

How to Configure New Processes

We can configure new processes by providing arguments to the `Process` class constructor.

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

Let's take a closer look at each.

Processes 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 process can be set via the `name` argument in the `Process` constructor.

Once created, the name of the process can be retrieved or changed again by the `name` attribute on a `Process` instance.

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

```
# SuperFastPython.com
# example of setting the process name in the constructor
from multiprocessing import Process
```

```
# protect the entry point
if __name__ == '__main__':
    # create a process with a custom name
    process = Process(name='MyProcess')
    # report process name
    print(process.name)
```

Running the example creates the child process with the custom name then reports the name of the process.

```
MyProcess
```

Processes can be configured to be *daemon* or *daemonic*, that is, they can be configured as background processes.

A parent process can only exit once all non-daemon child processes have exited.

This means that daemon child processes can run in the background and do not prevent the main process of a Python program from exiting when the main parts of the program have finished.

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

The daemon status of a `Process` instance can be queried via the `daemon` attribute. If the process has not yet started, the daemon attribute can be changed again.

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

```
# SuperFastPython.com
# example of setting a process to be a daemon
from multiprocessing import Process

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

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

```
True
```

Next, let's explore how we might query the status of a process via its `Process` instance.

How to Query the Status of a Process

An instance of the `Process` class provides a handle on a new instance of the Python interpreter.

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

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

Three more qualities of a process we may want to query in our programs include the process identifier (PID), whether the process is still running (or not) and the exit code of the process (if terminated).

Let's take a closer look at each.

Each process has a unique process identifier, called the PID, assigned by the operating system.

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

Knowing the PIDs for processes in our program can help with logging and with the general maintenance of the system. Other programs may query the PID of the process and even signal or terminate the process externally via the PID.

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

The example below creates an instance of a `Process` and reports the assigned PID.

```
# SuperFastPython.com
# example of reporting the native process identifier
from multiprocessing import Process

# protect the entry point
if __name__ == '__main__':
    # create the process
    process = Process()
    # report the process identifier
    print(process.pid)
    # start the process
    process.start()
    # report the process identifier
    print(process.pid)
```

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

The process is then started and the assigned PID is reported.

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

```
None
16302
```

A `Process` instance can be alive or dead.

An alive process means that the `run()` method of the `Process` instance is currently executing.

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

We can check if a process is alive via the `is_alive()` method on the `Process` instance.

The example below creates a `Process` instance then checks whether it is alive.

```
# SuperFastPython.com
# example of assessing whether a process is alive
from multiprocessing import Process

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

Running the example creates a new `Process` instance then reports that the process is not alive.

```
False
```

A child process will have an exit code once it has terminated.

An exit code provides an indication of whether processes completed successfully or not, and if not, the type of error that occurred that caused the termination.

Common exit codes include: 0 for a normal exit and 1 for an error or failure of some kind.

We can retrieve the exit code for a child process via the `exitcode` attribute.

A process will not have an exit code until the process is terminated. This means, checking the exit code of a process before it has started or while it is running will return a `None` value.

We can demonstrate this with a worked example that executes a custom target function that blocks for one second.

The complete example is listed below.

```
# SuperFastPython.com
# example of checking the exit status of a child process
from time import sleep
from multiprocessing import Process

# custom function to be executed in a child process
def task():
    # block for a moment
    sleep(1)

# protect the entry point
if __name__ == '__main__':
    # create the process
    process = Process(target=task)
    # report the exit status
    print(process.exitcode)
    # start the process
    process.start()
    # report the exit status
    print(process.exitcode)
    # wait for the process to finish
    process.join()
    # report the exit status
    print(process.exitcode)
```

Running the example first creates a new process to execute our custom task function.

The status code of the child process is reported, which is `None` as expected. The process is started and blocked for one second. While running, the exit code is reported again, which again is `None` as expected.

The parent process then blocks until the child process terminates. The exit code is reported, and in this case we can see that a value of zero is reported indicating a normal or successful exit.

```
None
None
0
```

The exit code for a process can be set by calling `sys.exit()` and specifying an integer value, such as 1 for a non-successful exit.

Next, let's explore how we might close or kill a child process.

How to Terminate or Kill a Process

A process will exit normally when its `run()` method exits, either normally or with an error.

A process may also be forcefully terminated or killed from another process. This involves raising a signal in the target process.

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

A process can be forcefully killed by calling the `terminate()` method on the `Process` instance. The call will only terminate the target process, not any child processes the target process may have.

For example:

```
...
# terminate the process
process.terminate()
```

The method will terminate the process using the `SIGTERM` (signal terminate) signal on most platforms, or the equivalent on windows.

A process may choose to handle a `SIGTERM` signal by registering a signal handler function via the `signal.signal()` function. As such, calling the `terminate()` method on a processes may be handled by that process.

Alternately, processes can be forcefully killed by calling the `kill()` method on a `Process` object.

The call will only terminate the target process, not child processes.

For example:

```
...
# kill the process
process.kill()
```

The method will terminate the process using the `SIGKILL` (signal kill) signal on most platforms, or the equivalent on windows.

This `SIGKILL` signal cannot be ignored and cannot be handled, unlike `SIGTERM` raised by the `terminate()` method. This makes calling `kill()` more forceful than calling `terminate()`.

Next, let's explore how we might get a `Process` instance for running processes within our program.

How to Get the Current, Parent, and Child Processes

The `multiprocessing` module provides a number of utility functions for getting `Process` instances for currently running processes.

This includes functions for getting the current process, the parent of the current process and a list of child processes.

Let's take a look at each in turn.

We can get a `Process` instance for the process running the current code.

This can be achieved via the `current_process()` module function that returns a `Process` instance.

The example below demonstrates how we can use this function to access the `Process` instance for the main process.

```
# SuperFastPython.com
# example of getting access to the current process
from multiprocessing import current_process

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

Running the example gets the `Process` instance for the currently running process.

The details are then reported, showing that we have accessed the main process that has no parent process.

```
<_MainProcess name='MainProcess' parent=None started>
```

We may need to access the parent process for a current child process.

This can be achieved via the `parent_process()` function. This will return a `Process` instance for the parent of the current process.

The main process does not have a parent, therefore attempting to get the parent of the main process will return `None`.

We can demonstrate this with a worked example.

```
# SuperFastPython.com
# example of getting the parent process
from multiprocessing import parent_process

# protect the entry point
if __name__ == '__main__':
    # get the parent process
    process = parent_process()
    # report details
    print(process)
```

Running the example attempts the `Process` instance for the main process.

The function returns `None`, as expected as the main process does not have a parent process.

```
None
```

We can get a list of all active child processes for a parent process.

This can be achieved via the `active_children()` function that returns a list of all child processes that are currently running.

We can develop a small example that first starts a number of children processes then has each child process block for a moment, then have the parent process get and report the list of active child processes.

The complete example is listed below

```
# SuperFastPython.com
# example of getting a list of active child processes
from time import sleep
from multiprocessing import active_children
from multiprocessing import Process

# custom function to be executed in a child process
def task():
    # block for a moment
    sleep(1)

# protect the entry point
if __name__ == '__main__':
    # create a number of child processes
    processes = [Process(target=task) for _ in range(5)]
    # start the child processes
    for process in processes:
        process.start()
    # get a list of all active child processes
    children = active_children()
    # report a count of active children
    print(f'Active Children Count: {len(children)}')
    # report each in turn
    for child in children:
        print(child)
```

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

Each process blocks for a moment, giving the main process time to query active child processes.

A list of all active child processes is then retrieved in the main process. The count is reported, which is shown as five, as we expect, then the details of each process are then reported.

This highlights how we can access all active child processes from a parent process.

It also shows that child processes are assigned names automatically based on the order they were created, e.g. *Process-1*, *Process-2*.

NOTE: The PIDs and parent PIDs will vary each time the program is run because they are allocated by the operating system.

```
Active Children Count: 5
<Process name='Process-3' pid=16853 parent=16849 ...
<Process name='Process-4' pid=16854 parent=16849 ...
<Process name='Process-2' pid=16852 parent=16849 ...
<Process name='Process-1' pid=16851 parent=16849 ...
<Process name='Process-5' pid=16855 parent=16849 ...
```

Next, let's explore how we might configure the method used to start child processes.

How to Configure the Start Method

A start method is the technique used to start child processes in Python.

There are three start methods, they are:

- `'spawn'`: start a new Python process.
- `'fork'`: copy a Python process from an existing process.
- `'forkserver'`: new process from which future forked processes will be copied.

Each platform has a default start method, e.g. Windows and MacOS use `'spawn'`, whereas Linux uses `'fork'`.

Not all platforms support all start methods., e.g. windows does not support `'fork'` or `'forkserver'`.

The `multiprocessing` module provides functions for getting and setting the start method for creating child processes.

The list of supported start methods can be retrieved via the `get_all_start_methods()` function.

The function returns a list of string values, each representing a supported start method.

For example:

```
...
# get supported start methods
methods = get_all_start_methods()
```

The current start method can be retrieved via the `get_start_method()` function.

The function returns a string value for the currently configured start method.

For example:

```
...
# get the current start method
method = get_start_method()
```

The start method can be set via the `set_start_method()` function.

The function takes a string argument indicating the start method to use.

This must be one of the start methods supported on your platform.

For example:

```
...
# set the start method
set_start_method('spawn')
```

It is a best practice, and required on most platforms that the start method be set first, prior to any other code, and to be done so within a `if __name__ == '__main__'` check called a protected entry point or top-level code environment.

For example:

```
...
# protect the entry point
if __name__ == '__main__':
    # set the start method
    set_start_method('spawn')
```

We can use different start methods throughout our program by creating processes from different multiprocessing contexts.

We can create a multiprocessing context configured to use a specific start method via the `get_context()` function. This function takes the name of the start method as an argument, then returns a multiprocessing context that can be used to create new child processes.

For example:

```
...
# get a context configured with a start method
context = get_context('fork')
```

The context can then be used to create a child process directly, for example:

```
...
# create a child process via a context
process = context.Process(...)
```

In fact, a multiprocessing context provides access to the entire `multiprocessing` API, allowing other multiprocessing objects like a `Lock` or `Semaphore` be created in a way that is compatible with the configured start method.

Lesson Review

Takeaways

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

- You now know how to configure the name of a process and whether it is a daemon.
- You now know how to query the status of a process such as if it is running and its exit code.
- You now know how to terminate and kill processes.
- You now know how to get access to the current, parent and child processes.
- You now know how to configure the start method and use a multiprocessing context.

Exercise

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

Create a small program that creates and starts one or more processes performing some arbitrary task.

Then from the main process, wait for some time or a trigger and terminate one or more of the processes you have created.

Extend the example so that the child processes handle the `SIGTERM` signal and perform a clean-up operation for the task before being terminated.

Forcefully killing parallel tasks in child processes is a requirement of many programs that use concurrency. This exercise is good practice for when you need to add this capability to your own concurrent programs in the future.

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.

- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html
- `sys` - System-specific parameters and functions.
  https://docs.python.org/3/library/sys.html

- `signal` - Set handlers for asynchronous events.
  https://docs.python.org/3/library/signal.html
- `__main__` - Top-level code environment.
  https://docs.python.org/3/library/__main__.html

Next

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

Lesson 04: Synchronize and Coordinate Processes

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

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 processes using an event.
- How to coordinate action with wait and notify using a condition variable.
- How to coordinate multiple processes at one point 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 processes (or 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 processes 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 processes via the `Lock` class.

An instance of the lock can be created and then acquired by processes before accessing a critical section, and released after the critical section.

For example:

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

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

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

We can also use the lock via the context manager protocol via the with statement, allowing the critical section to be a block within the usage of the lock and for the lock to be released automatically once the block has completed.

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 multiprocessing import Process
from multiprocessing import Lock

# custom function to be executed in a child process
def task(shared_lock, ident, value):
    # acquire the lock
    with shared_lock:
        # report a message
        print(f'>{ident} got lock, sleeping {value}',
            flush=True)
        # 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 processes with different args
    processes = [Process(target=task,
        args=(lock, i, random())) for i in range(10)]
    # start the processes
    for process in processes:
        process.start()
    # wait for all processes to finish
    for process in processes:
        process.join()
```

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

The child processes are then started and the main process blocks until all child processes finish.

Each child process attempts to acquire the lock within the `task()` function.

Only one process 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 process 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.

```
>2 got lock, sleeping 0.1537300967240235
>0 got lock, sleeping 0.15033927763482824
>3 got lock, sleeping 0.06852384740143935
>4 got lock, sleeping 0.6128370934678512
>1 got lock, sleeping 0.6095906057434888
>5 got lock, sleeping 0.9096664066862853
>6 got lock, sleeping 0.3292580160207148
>7 got lock, sleeping 0.576156511368414
>8 got lock, sleeping 0.7539760059829592
>9 got lock, sleeping 0.26463763659870543
```

The `multiprocessing` module also provides a reentrant lock via the `RLock` class. This mutex allows a process 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 processes 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 processes that can acquire the lock before additional processes will block. Once full, new processes can only acquire access on the semaphore once an existing process 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 processes via the `Semaphore` class.

The `Semaphore` instance must be configured when it is created to set the limit on the internal counter. This limit will match the number of concurrent processes 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 the semaphore has no available access in which case, processes attempting to acquire it must block until access 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 process 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 processes 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 processes, allowing some processes 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 multiprocessing import Process
from multiprocessing import Semaphore

# custom function to be executed in a child process
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'Process {ident} got {val}', flush=True)

# protect the entry point
if __name__ == '__main__':
    # create the shared semaphore
    semaphore = Semaphore(2)
    # create processes
    processes = [Process(target=task,
        args=(semaphore, i)) for i in range(10)]
    # start child processes
    for process in processes:
        process.start()
    # wait for child processes to finish
    for process in processes:
        process.join()
```

Running the example first creates the shared semaphore instance then starts ten child processes.

All ten processes attempt to acquire the semaphore, but only two processes are granted access at a time. The processes 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 process to acquire access and perform its simulated calculation, all the while allowing only two of the processes to be running within the critical section at any one time, even though all ten processes are executing their run methods.

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

```
Process 3 got 0.28471035400619005
Process 1 got 0.7548211067378485
Process 0 got 0.7285818407567202
Process 6 got 0.3527937627550165
Process 4 got 0.98181495633406
Process 2 got 0.5985443390280271
Process 5 got 0.3897305303458548
Process 8 got 0.2794555552459089
Process 9 got 0.44061600851138194
Process 7 got 0.9779313084749204
```

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

How to Signal Between Processes Using an Event

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

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

An `Event` class wraps a boolean variable that can either be *set* (`True`) or *not set* (`False`). Processes sharing the event instance 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 processes 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 processes waiting on the event to be set will be notified.

For example:

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

Finally, processes 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 process 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 processes that each will perform some processing and report a message. All processes will use an event to wait to be set before starting their work. The main process will set the event and trigger the child processes to start work.

```
# SuperFastPython.com
# example of using an event object with processes
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing import Event

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

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

Running the example first creates and starts five child processes.

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

The main process blocks for a moment, allowing all child processes to begin and start waiting on the event.

The main process then sets the event. This triggers all five child processes 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.

```
Main process blocking...
Process 1 waiting...
Process 0 waiting...
Process 4 waiting...
Process 3 waiting...
Process 2 waiting...
Process 0 got 0.6141449597363285
Process 4 got 0.654083699410865
Process 2 got 0.7031834341766182
Process 1 got 0.7116801625105117
Process 3 got 0.9641420413449788
```

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

How to Coordinate Using a Condition Variable

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

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

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

The waiting process 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 process 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 process to stop blocking after a time limit in seconds.

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

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

For example:

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

The notified process 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 processes waiting on the condition via the `notify_all()` method.

```
...
# acquire the condition
with condition:
    # notify all processes 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 child process to simulate performing some work that the main process is dependent upon. Once prepared, the child process will notify the waiting main process, then the main process will continue on.

The complete example is listed below.

```
# SuperFastPython.com
# example of wait/notify with a condition for processes
from time import sleep
from multiprocessing import Process
from multiprocessing import Condition

# custom function to be executed in a child process
def task(shared_condition):
    # block for a moment
    sleep(1)
    # notify a waiting process that the work is done
    print('Child sending notification...', flush=True)
    with shared_condition:
        shared_condition.notify()
```

```python
# protect the entry point
if __name__ == '__main__':
    # create a condition
    condition = Condition()
    # acquire the condition
    print('Main process waiting for data...')
    with condition:
        # create a new process to execute the task
        worker = Process(target=task, args=(condition,))
        # start the new child process
        worker.start()
        # wait to be notified by the child process
        condition.wait()
    # we know the data is ready
    print('Main process all done')
```

Running the example first creates the condition variable.

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

The child process blocks for a moment to simulate work, then notifies the waiting main process.

Meanwhile the main process waits to be notified by the child process, then once notified it continues on.

```
Main process waiting for data...
Child sending notification...
Main process all done
```

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

How to Coordinate at a Point Using a Barrier

A barrier is a synchronization primitive.

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

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

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

Python provides a barrier via the `Barrier` class.

A barrier instance must first be created and configured via the constructor specifying the number of parties (processes) 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 processes 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 process once all processes reach the barrier but before the processes are released.

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

Once configured, the barrier instance can be shared between processes and used.

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

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

This is a blocking call and will return once all other processes (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 processes, each required to perform some blocking calculation. We will use a `Barrier` to coordinate all processes after they have finished their work and perform some action in the main process.

The complete example is listed below.

```
# SuperFastPython.com
# example of using a barrier with processes
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing import Barrier

# custom function to be executed in a child process
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'Process {ident} got: {value}', flush=True)
    # wait for all other processes to complete
    shared_barrier.wait()

# protect the entry point
if __name__ == '__main__':
    # create a barrier for (5 workers + 1 main process)
    barrier = Barrier(5 + 1)
    # create the worker processes
    workers = [Process(target=task,
        args=(barrier, i)) for i in range(5)]
    # start the worker processes
    for worker in workers:
        # start process
        worker.start()
    # wait for all worker processes to finish
    print('Main process waiting on all results...')
    barrier.wait()
    # report once all processes are done
    print('All processes have their result')
```

Running the example first creates the shared `Barrier` then creates and starts the worker processes.

Each worker process performs its calculation and then waits on the barrier for all other processes to finish.

Finally, the processes finish and are all released, including the main process, reporting a final message.

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

```
Main process waiting on all results...
Process 0 got: 1.5797718210997858
Process 1 got: 5.153409616555768
Process 2 got: 6.194883621010085
Process 3 got: 6.664785986540583
Process 4 got: 7.594772825174695
All processes 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 processes using an event.
- You know how to coordinate action with wait and notify using a condition variable.
- You know how to coordinate multiple processes at one point using a barrier.

Exercise

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

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

You could have 10 processes that add one to a balance many times each in a loop and 10 processes do the same by subtracting one from the same shared variable (such as a shared ctype from the next lesson).

For example:

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

# subtract from the balance
def subtract(variable):
    for i in range(1000000):
        variable.value -= 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 process-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)
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html
- `random` - Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html

Next

In the next lesson, we will discover how to share data between processes.

Lesson 05: Share Data Between Processes

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

After completing this lesson, you will know:

- How forked processes can inherit global variables from parent processes.
- How to use process-safe shared `ctypes` to share primitives.
- How to send and receive data between processes using pipes.
- How to create producer and consumer processes with a shared queue.

Let's get started.

How to Inherit Global Variables from Parent Processes

A forked child process can inherit global variables from a parent process.

Recall that a global variable is a variable defined and assigned within a module, e.g. outside of a function. They are different from variables defined and assigned within a function, which are referred to as local variables.

Global variables can be accessed and assigned within a function if they are defined as being global, using the `global` keyword before they are used.

A global variable can be inherited by a child process.

This means that we can define and assign a global variable in a parent process, then access and assign values to it in a function executed by a child process.

Importantly, changes made to the global variable in the child process will not propagate back up to the parent process.

Global variables can only be shared or inherited by child processes that are forked from the parent process.

Specifically, this means that we must create child processes using the `'fork'` start method.

Recall that the `'fork'` start method is the default on Unix platforms, and can be set prior to creating child processes via the `set_start_method()` function.

For example:

```
...
# set the start method to fork
set_start_method('fork')
```

The fork start method is not available on all platforms, e.g. it may not be available on Windows.

Although the global variables are inherited, changes to the variable are not propagated.

Specifically:

- Changes to a global variable in the parent process are not propagated to the child processes.
- Changes to the global variable in a child process are not propagated back to the parent process or to other child processes.

This means that the forked child process gets a snapshot of the global variables from the parent process at the time the child process was created.

You cannot inherit global variables when using the `'spawn'` method to start child processes.

Recall that the `'spawn'` start method is the default on MacOS and Windows.

Attempting to access and use a global variable in a child process started with the `'spawn'` start method will result in an error.

We can explore how child processes forked from a parent process will inherit the parents global variables.

The parent process will explicitly use the fork start method. It will then define a global variable and assign a value to the variable and report the global variable to confirm it was assigned. A child process will then run a custom function that will attempt to access and report the same global variable defined in the parent process.

The complete example is listed below.

```
# SuperFastPython.com
# example of sharing global between forked processes
from time import sleep
from multiprocessing import Process
from multiprocessing import set_start_method

# custom function to be executed in a child process
def task():
    # declare global state
    global data
    # report global state
    print(f'child process before: {data}', flush=True)
    # change global state
    data = 'hello hello!'
    # report global state
    print(f'child process after: {data}', flush=True)

# protect the entry point
if __name__ == '__main__':
    # set the start method to fork
    set_start_method('fork')
    # define global state
    data = 'Hello there'
    # report global state
    print(f'main process: {data}')
    # start a child process
    process = Process(target=task)
    process.start()
    # wait for the child to terminate
    process.join()
    # report global state
    print(f'main process: {data}')
```

Running the example first sets the start method to `'fork'`.

It then defines a global variable and assigns it the value *Hello there* and reports the variable to confirm that this is the current value.

A child process is then started using the `'fork'` start method and executes our custom function. The main process then blocks until the child process terminates.

The child process declares the `data` variable as being global.

It then reports the `data` global variable. This works as expected, as the variable was inherited from the parent process. The value is the same as that assigned in the parent value, specifically *Hello there*.

The child process then changes the value of the global variable, then reports the value again. This shows that the value was changed to *hello hello!*.

The child process terminates and the parent process wakes up. It then reports the value of the global variable again, which is unchanged with the value *Hello there*.

This highlights that indeed a forked child process will inherit global variables, but changes to the global variable are not propagated from the child back to the parent.

NOTE: This example will not work on platforms without support for the `'fork'` start method, such as Windows.

```
main process: Hello there
child process before: Hello there
child process after: hello hello!
main process: Hello there
```

Although a forked child process can inherit global variables from a parent process, it should be avoided.

The most important reason is because the functionality is only supported on those platforms that support the `'fork'` spawn method, e.g. not Windows.

Next, let's explore how we can use shared `ctypes` between processes.

How to Share Python Primitives with `ctypes`

The `ctypes` module in Python provides tools for working with C primitive data types, such as floats, integers, and characters.

This is helpful because the data types used in C are somewhat standardized across many platforms.

The `ctypes` module allows Python code to read, write, and generally interoperate with data using standard C data types.

Python provides the capability to share `ctypes` between processes on one system.

This is primarily achieved via the following classes:

- The `Value` class is used to share a `ctype` of a given type among multiple processes.
- The `Array` class is used to share an array of `ctypes` of a given type among multiple processes.

Share `ctypes` provide a simple and easy to use way of sharing data between processes.

For example, a shared `ctype` value can be defined in a parent process, then shared with multiple child processes. All child processes and the parent process can then safely read and modify the data within the shared object.

This can be helpful in a number of use cases, such as:

- A counter shared among multiple processes.
- Returning data from a child process to a parent process.
- Sharing results of computation among processes.

Let's take a closer look at sharing one primitive value between processes with the `Value` class.

The `Value` class will create a shared `ctype` with a specified data type and initial value.

For example:

```
...
# create a value
value = Value(...)
```

The first argument defines the data type for the value. It may be a string type code or a Python `ctype` class. The second argument may be an initial value.

For example, we can define a signed integer type with the `'i'` type code and an initial value of zero as follows:

```
...
# create a integer value
variable = Value('i', 0)
```

The most common data types we are likely to use are: `'i'` for signed integer, `'f'` for single floating point value, and `'c'` for character.

We can define the same signed integer shared `ctype` using the `ctypes.c_int` class.

For example:

```
...
# create a integer value
variable = Value(ctypes.c_int, 0)
```

Once defined, the value can then be shared and used within multiple processes, such as between a parent and a child process.

Internally, the `Value` makes use of a reentrant mutex lock (`RLock`) that ensures that access and modification of the data inside the class is mutually exclusive, e.g. process-safe.

This means that only one process at a time can access or change the data within the `Value` object.

The data within the `Value` object can be accessed via the `value` attribute.

For example:

```
...
# get the data
data = variable.value
```

The data within the `Value` can be changed by the same `value` attribute.

For example:

```
...
# change the data
variable.value = 100
```

We can explore how to share a floating point value between processes.

In this example we will define a `Value` as a floating point variable and an initial value. The value will be shared with a child process and modified with a random value. The parent process will then access and report the variable, confirming that the change took effect.

The complete example is listed below.

```
# SuperFastPython.com
# example of shared ctype accessed in multiple processes
from random import random
from multiprocessing import Value
from multiprocessing import Process

# custom function to be executed in a child process
def task(shared_var):
    # generate a single floating point value
    generated = random()
    # store value
    shared_var.value = generated
    # report progress
    print(f'Wrote: {shared_var.value}', flush=True)

# protect the entry point
if __name__ == '__main__':
    # create shared variable
    variable = Value('f', 0.0)
    # create a child process process
    process = Process(target=task, args=(variable,))
    # start the process
    process.start()
    # wait for the process to finish
    process.join()
    # read the value
    data = variable.value
    # report the value
    print(f'Read: {data}')
```

Running the example first creates the shared `Value` instance to hold a float.

The child process is configured and started and the main process blocks until the child process terminates.

The child process generates a random value. It then stores the generated value in the shared Value instance for the parent process to access and reports the value that was stored.

The child process terminates and the main process unblocks.

The parent process then reports the number stored in the shared Value instance.

We can see that the generated value in the child process matches the value accessed by the parent process, showing that indeed that the variable is shared and updated among the two processes.

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

```
Wrote: 0.17691104226938958
Read: 0.17691104114055634
```

Next, let's explore how we might send and receive data between processes using a pipe.

How to Send Data to Processes with Pipes

In multiprocessing, a pipe is a connection between two processes in Python.

It is used to send data from one process which is received by another process.

Under the covers, a pipe is implemented using a pair of connection objects, provided by the `Connection` class.

Creating a pipe will create two `Connection` objects, one for sending data and one for receiving data. A pipe can also be configured to be duplex so that each connection object can both send and receive data.

Python provides a simple pipe in the `Pipe` class.

A pipe can be created by calling the constructor of the `Pipe` class, which returns two `Connection` objects.

For example:

```
...
# create a pipe
conn1, conn2 = Pipe()
```

By default, the first connection (`conn1`) can only be used to receive data, whereas the second connection (`conn2`) can only be used to send data.

The connection objects can be made duplex or bidirectional.

This can be achieved by setting the `duplex` argument to the constructor to `True`.

For example:

```
...
# create a duplex pipe
conn1, conn2 = Pipe(duplex=True)
```

In this case, both connections can be used to send and receive data.

Objects can be shared between processes using the `Pipe`.

The `send()` method can be used to send objects from one process to another.

The objects sent must be picklable.

For example:

```
...
# send an object
conn2.send('Hello world')
```

The `recv()` method can be used to receive objects in one process sent by another.

The objects received will be automatically un-pickled.

For example:

```
...
# receive an object
obj = conn1.recv()
```

The method call will block until an object is received.

We can explore how to use a `Pipe` to share data between processes.

In this example we will create a sender process that will generate random numbers and send them to another process via the `Pipe`. We will also create a receiver process that will receive numbers sent from the other process and report them.

```
# SuperFastPython.com
# example of using a pipe between processes
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing import Pipe

# custom function generate work items (sender)
def sender(connection):
    print('Sender: Running', flush=True)
    # generate work
    for _ in range(10):
        # generate a value
        value = random()
        # block
        sleep(value)
        # send data
        connection.send(value)
    # all done, signal to expect no further messages
    connection.send(None)
    print('Sender: Done', flush=True)

# custom function to consume work items (receiver)
def receiver(connection):
    print('Receiver: Running', flush=True)
    # consume work
    while True:
        # get a unit of work
        item = connection.recv()
        # report
        print(f'>receiver got {item}', flush=True)
        # check for stop
        if item is None:
            break
    # all done
    print('Receiver: Done', flush=True)

# protect the entry point
if __name__ == '__main__':
    # create the pipe
    conn1, conn2 = Pipe()
    # start the sender
    sender_p = Process(target=sender, args=(conn2,))
    sender_p.start()
    # start the receiver
    receiver_p = Process(target=receiver, args=(conn1,))
    receiver_p.start()
    # wait for all processes to finish
    sender_p.join()
    receiver_p.join()
```

Running the example first creates the pipe, then creates and starts both child processes.

The main process then blocks until the child process is finished.

The sender child process then runs in a loop, generating and sending ten random values along the pipe. Once all values are generated and sent, the sender process sends a special value (called a sentinel value) to indicate that no further values should be expected, then terminates.

The child process loops, receiving objects from the pipe each iteration. It blocks until an object appears each iteration. Received values are reported, and the loop is broken once the sentinel value is received.

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

```
Sender: Running
Receiver: Running
>receiver got 0.948078639569654
>receiver got 0.74436836170323
>receiver got 0.5616808907659429
>receiver got 0.4215810585838159
>receiver got 0.5722781037208557
>receiver got 0.46904197077762444
```

```
>receiver got 0.6680454030764481
>receiver got 0.008521317435038922
>receiver got 0.9341184741199164
Sender: Done
>receiver got 0.47239922120411093
>receiver got None
Receiver: Done
```

Next, let's explore how we might share data between processes using a process-safe queue.

How to Use Producers and Consumers with 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()`.

Python provides a process-safe queue in the `Queue` class.

The multiprocessing queues are a process-safe version of the process-safe queues provided in the `queue` module.

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 `multiprocessing` module provides other types of process-safe queues such as the `SimpleQueue` that does not allow the capacity of the queue to be limited and the `JoinableQueue` that allows a caller to wait to be notified that all items in the queue have been processed.

Let's look at how we can use the `Queue` class.

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 queue can be created with a size limit by specifying the `maxsize` argument to a value larger than zero.

For example:

```
...
# created a size limited queue
queue = Queue(maxsize=100)
```

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

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

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

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 and will not use a timeout.

The number of items in the queue can be checked by the `qsize()` method.

For example:

```
...
# check the size of the queue
size = queue.qsize()
```

We can check if the queue contains no values via the `empty()` method.

For example:

```
...
# check if the queue is empty
if queue.empty():
    # ...
```

We may also check if the queue is full, if it is size limited when configured.

For example:

```
...
# check if the queue is full
if queue.full():
    # ...
```

We can explore how to use the `Queue` class with a worked example.

In this case, we will create a producer child process that will generate ten random numbers and put them on the queue. We will also create a consumer child process that will get numbers from the queue and report their values.

```
# SuperFastPython.com
# example of producer and consumer processes with queue
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing import Queue

# custom function for generating work (producer)
def producer(shared_queue):
    print('Producer: Running', flush=True)
    # generate work
    for _ in range(10):
        # generate a value
        value = random()
        # block
        sleep(value)
        # add to the queue
        shared_queue.put(value)
    # all done
    shared_queue.put(None)
    print('Producer: Done', flush=True)

# custom function for consuming work (consumer)
def consumer(shared_queue):
    print('Consumer: Running', flush=True)
    # consume work
    while True:
        # get a unit of work
        item = shared_queue.get()
        # check for stop
        if item is None:
            break
        # report
        print(f'>got {item}', flush=True)
    # all done
    print('Consumer: Done', flush=True)

# protect the entry point
if __name__ == '__main__':
    # create the shared queue
    queue = Queue()
    # start the consumer
    consumer_p = Process(target=consumer, args=(queue,))
    consumer_p.start()
    # start the producer
    producer_p = Process(target=producer, args=(queue,))
    producer_p.start()
    # wait for all processes to finish
    producer_p.join()
    consumer_p.join()
```

Running the example first creates the shared `Queue` instance.

Next, the consumer process is created and passed the queue instance. Then the producer process is started and the main process blocks until the worker processes terminate.

The producer process generates a new random value each iteration of the task, blocks and adds it to the queue. The consumer process waits on the queue for items to arrive, then consumes them one at a time, reporting their value.

Finally, the producer task finishes, a `None` value is put on the queue and the process terminates. The consumer process gets the `None` value, breaks its loop and also terminates.

This highlights how the `Queue` can be used to share data easily between producer and consumer processes.

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

```
Consumer: Running
Producer: Running
>got 0.42670034948416746
>got 0.6437306997304052
>got 0.5854279118232432
>got 0.6983355928616997
>got 0.27820873726406803
>got 0.3787863412350282
>got 0.7840846305336705
>got 0.9030905887595716
>got 0.06845335982079348
Producer: Done
>got 0.6508966920630934
Consumer: Done
```

Lesson Review

Takeaways

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

- You know how forked processes can inherit global variables from parent processes.
- You know how to use process-safe shared `ctypes` to share primitives.
- You know how to send and receive data between processes using pipes.
- You know how to create producer and consumer processes with a shared queue.

Exercise

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

Develop a small program where a task is split into subtasks and executed by child worker processes. The results of the task must be collected in the main process 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 process and sum their values.

Implement the program in a few different ways to explore the different approaches to sharing data between processes. For example, try using a pipe, try using a queue, and if you feel confident, also try using an Array.

Sharing data between processes is central to almost all programs that achieve parallelism via the `multiprocessing` 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.

- Pipeline (Unix), Wikipedia.
  https://en.wikipedia.org/wiki/Pipeline_(Unix)
- Queue (abstract data type), Wikipedia.
  https://en.wikipedia.org/wiki/Queue_(abstract_data_type)
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html
- `ctypes` - A foreign function library for Python.
  https://docs.python.org/3/library/ctypes.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 processes.

Lesson 06: Run Tasks with Reusable Workers in Pools

In this lesson, you will discover how you can execute ad hoc tasks with worker processes in the multiprocessing pool.

After completing this lesson, you will know:

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

Let's get started.

What Are Process Pools

The `Process` 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 process, although it becomes cumbersome when we have many tasks to run.

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

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

This can be achieved using a process pool.

A process pool is a programming pattern for automatically managing a pool of worker child processes.

The pool is responsible for a fixed number of processes.

- It controls when they are created, such as 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.

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

How to Use Multiprocessing Pools in Python

Python provides a process pool via the `Pool` class.

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

To use the process pool, we must first create and configure an instance of the class.

For example:

```
...
# create a process pool
pool = Pool(...)
```

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 process pool, it can be closed and resources used by the pool can be released.

For example:

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

Alternatively, we may want to forcefully terminate all child worker processes, regardless of whether they are executing tasks or not.

This can be achieved via the `terminate()` method.

For example:

```
...
# forcefully close all worker processes
pool.terminate()
```

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 multiprocessing pool 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 multiprocessing pool
with Pool() as pool:
    # issue tasks to the pool
    # ...
# close the pool automatically
```

Next, let's explore how we might configure the multiprocessing pool.

How to Configure the Multiprocessing Pool

The `Pool` is configured via the class constructor.

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

For example:

```
...
# create a process pool with a default configuration
pool = Pool()
```

The first argument is `processes` that specifies the number of workers 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 process pool.

It is a good idea to set the number of worker processes to the number of logical or the number of physical CPU cores in our system. Experiment and discover what works best for a given program.

For example:

```
...
# create a process pool with a given number of workers
pool = Pool(processes=4)
```

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

For example:

```
...
# create a process pool with a given number of workers
pool = Pool(4)
```

Each worker process in the pool is a separate child process.

It is possible for child processes to become unstable or accumulate resources without releasing them, such as if there are subtle bugs in the tasks that are being executed.

As such, it is a good practice to limit the number of tasks executed by each worker process and create a new replacement worker process once the limit on the number of tasks has been reached.

The number of tasks that may be executed by each worker process can be set via the `maxtasksperchild` argument, which defaults to no limit.

For example:

```
...
# create a process pool limiting each worker to 10 tasks
pool = Pool(maxtasksperchild=10)
```

We can also configure the pool to initialize each worker process with a custom initialization function via the `initializer` argument, and provide the multiprocessing context to use when creating the worker processes via the `context` 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 multiprocessing pool 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 process pool
result = pool.apply(task, args=(arg1, arg2))
```

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

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 import Pool

# custom function to be executed in a child process
def task():
    # report a message
    print('This is another process', flush=True)

# protect the entry point
if __name__ == '__main__':
    # create the multiprocessing pool
    with Pool() 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 multiprocessing pool.

The caller blocks until the task is complete.

```
This is another process
```

The multiprocessing pool provides a parallel version of the built-in `map()` method for issuing tasks.

The `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 multiprocessing pool `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 process 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 in parallel with a worked example, listed below.

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

# custom function to be executed in a child process
def task(arg):
    # report a message
    print(f'Worker task got {arg}', flush=True)
    # return a value
    return arg * 2

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

Running the example first issues 10 tasks to the multiprocessing pool.

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 process.

```
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 multiprocessing pool immediately.

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

As an alternative, the multiprocessing pool 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 Callback Functions to Process Results

The `Pool` supports custom callback functions.

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

Callbacks are only supported in the multiprocessing pool 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, flush=True)

...
# 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 multiprocessing pool.

```
# result callback function
def result_callback(result):
    # iterate all results
    for value in result:
        print(value, flush=True)

...
# 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, flush=True)

...
# issue a single task
result = apply_async(..., error_callback=custom_callback)
```

We can explore how to use a result callback with the multiprocessing 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 import Pool

# result callback function
def result_callback(return_value):
    # report a message
    print(f'Callback got: {return_value}', flush=True)

# custom function to be executed in a child process
def task(ident):
    # generate a value
    value = random()
    # report a message
    print(f'Task {ident} with {value}', flush=True)
    # block for a moment
    sleep(value)
    # return the generated value
    return value

# protect the entry point
if __name__ == '__main__':
    # create and configure the multiprocessing pool
    with Pool() as pool:
        # issue tasks to the multiprocessing pool
        result = pool.apply_async(task, args=(0,),
            callback=result_callback)
        # close the multiprocessing pool
        pool.close()
        # wait for all tasks to complete
        pool.join()
```

Running the example first starts the multiprocessing pool with the default configuration.

Then the task is issued to the multiprocessing pool. The main process 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 process 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 0.4982746119315874
Callback got: 0.4982746119315874
```

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 process pool.

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 process.

Lesson Review

Takeaways

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

- You know what multiprocessing pools are and when to use them in your program.
- You know how to create and configure new multiprocessing pools.
- You know how to execute single and multiple tasks using multiprocessing pools.
- You know how to use callback functions to process results and handle errors asynchronously.
- You know how to interact with tasks issued asynchronously in multiprocessing pools.

Exercise

Your task for this lesson is to use your knowledge of multiprocessing 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 multiprocessing pool via the `map()` method.

Update the example to use the `imap()` method and make the main process 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 process report results in the order that tasks are completed.

Multiprocessing pools are a helpful way of executing ad hoc tasks in parallel in your Python programs. With a few lines of code you can easily transform a for-loop or list comprehension to execute on all CPU cores. 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
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html

Next

In the next and final lesson, we will discover how we can create and share access to centralized Python objects among child processes using managers.

Lesson 07: Share Centralized Objects with Managers

In this lesson, you will discover how we can create a centralized version of a Python object in a server process and use it in multiple processes with a manager.

After completing this lesson, you will know:

- What managers are and how they can be used to centralize Python objects in a server process.
- How to create and use a manager to create centralized objects and shareable proxy objects.
- How to create a centralized Python data structure and share it among multiple processes.
- How to create a centralized concurrency primitive to be shared and used in worker processes.

Let's get started.

What Is a Manager

The multiprocessing manager provides a way of creating centralized Python objects that can be shared safely among processes.

Manager objects create a server process which is used to host Python objects. Managers then return proxy objects used to interact with the hosted objects.

The proxy objects automatically ensure process-safety and serialize data to and from the centralized objects.

This means that the proxy objects can be shared among multiple processes allowing the centralized object to be used in parallel seamlessly in a process-safe manner.

Why Use a Manager

Managers provide three key capabilities for process-based concurrency, they are:

- **Centralized**: A single instance of a shared object is maintained in a separate server process.

- **Process-safety**: Proxy objects ensure that access to the centralized object is process-safe in order to avoid race conditions.
- **Pickability**: Proxy objects can be pickled and shared with child processes such as arguments in process pools and items in queues.

In addition to providing safe access to a shared object across processes on one system, managers allow the same object to be accessed safely across processes on other systems via network access.

When to Use a Manager

A manager should be used when an object needs to be shared among processes.

Sometimes the object can be shared directly without problem, other times it cannot and a manager is required for those cases, which include:

- When the shared object cannot be pickled and needs to be pickled in order to be shared.
- When the shared object is not process-safe and needs to be used in multiple processes simultaneously.

If the object to be shared meets either or both of these requirements, then it is a good candidate for being hosted in a manager server process and shared via proxy objects.

There are a number of common use cases where you may need to use a manager.

They include:

1. When sharing a concurrency primitive (e.g. `Lock`, `Semaphore`, `Event`, etc.) or `Queue` with a `Pool` or `ProcessPoolExecutor`. This is because concurrency primitives cannot be pickled and all arguments to process pools must be pickled.
2. When sharing an object that cannot be pickled with processes via a `Queue`. This is because the `Queue` requires that all objects put on the queue be pickled.

Next, let's look at how we might create and use a manager.

How to Use A Manager

Using a manager involves four steps, they are:

1. Create the manager instance.
2. Start the manager.
3. Create one or more hosted objects to share.
4. Shutdown the manager.

Step 1. Create a Manager

A manager can be created and configured via an instance of the `Manager` class.

For example:

```
...
# create a manager
manager = Manager()
```

Step 2. Start the Manager

After the manager is created, it must be started.

This can be achieved by calling the `start()` method.

For example:

```
...
# start the manager
manager.start()
```

This will start the server process used to host centralized versions of Python objects.

Step 3. Create Hosted Objects

Once the manager is started, it can be used to create hosted objects and return proxy objects for those hosted objects.

Only objects that have been registered with the manager, e.g. those objects that the manager knows about, can be created.

For example:

```
...
# create a hosted object and get a proxy object
proxy_object = manager.Lock()
```

This creates a centralized version of an object in the `Manager`'s server process, in this case a mutex `Lock` class, and returns a proxy for interacting with the hosted object.

Step 4. Shutdown the Manager

Once we are finished with the shared objects, we can shutdown the manager.

This will release all resources used by the manager, including the server process.

This can be achieved by calling the `close()` method.

For example:

```
...
# shutdown the manager
manager.close()
```

Context Manager Interface

An alternate usage pattern for the `Manager` is to use the context manager interface.

This ensures that the `Manager` instance is always closed, in case we forget to call `close()` or in case the `close()` method is not reached due to an error.

For example:

```
...
# create and start the manager
with Manager() as manager:
    # create a hosted object and get a proxy object
    proxy_object = manager.Lock()
    # ...
# manager is closed automatically
```

The manager is created and started at the beginning of the context manager block, all usage of the manager is restricted to the block, then the manager is closed as soon as the block is exited, normally or otherwise.

This is the preferred way to use the `Manager` in your program, if possible.

Next, let's look at the types of Python objects we can create with a manager and how they may be created.

What Objects Do Managers Support

Managers are provided via the `Manager` class, which creates and returns a `SyncManager` instance.

The `SyncManager` allows a suite of Python objects to be created and managed by default, including:

- Data structures.
- Shared `ctypes`.
- Concurrency primitives.
- Queues.
- Other Objects.

Let's take a closer look at each group:

Managed Data Structures

This includes Python objects we may want to share, such as:

- `dict`
- `list`

For example:

```
...
# create and start the manager
with Manager() as manager:
    # create a dict
    proxy_dict = manager.dict()
    # ...
```

Managed Shared ctypes

This include shared `ctypes` for primitive values, such as:

- `Value`
- `Array`

For example:

```
...
# create and start the manager
with Manager() as manager:
    # create a shared value
    proxy_value = manager.Value()
    # ...
```

Managed Concurrency Primitives

This includes concurrency primitives for synchronizing and coordinating processes, such as:

- `Lock`
- `Event`
- `Condition`
- `Semaphore`
- `BoundedSemaphore`
- `Barrier`

For example:

```
...
# create and start the manager
with Manager() as manager:
    # create a shared event
    proxy_event = manager.Event()
    # ...
```

Managed Queues

This includes process-safe queues for sharing data between processes, such as:

- `Queue`
- `JoinableQueue`

For example:

```
...
# create and start the manager
with Manager() as manager:
    # create a shared queue
    proxy_queue = manager.Queue()
    # ...
```

Managed Other Objects

Other objects are provided by the `SyncManager` class, some of which are not documented, such as:

- `Namespace`
- `Pool`

A `Namespace` provides a workspace for sharing Python primitive objects, such as integers, floating point values, strings, and so on. A `Pool` is a pool of worker processes.

For example:

```
...
# create and start the manager
with Manager() as manager:
    # create a shared namespace
    proxy_namespace = manager.Namespace()
    # ...
```

Details of Hosted Objects

In fact, if we look at the source code for the `SyncManager` class, we can see that the actual classes that are hosted are from the `threading` module, not the `multiprocessing` module.

Likely this is because the hosted objects are never used outside of the `Manager`'s server process. They are only interacted with indirectly via proxy objects which manage the inter-process communication and serialization for process-safety.

Now that we are familiar with how to use a `Manager`, let's look at some worked examples.

How to Use a Manager to Share a Data Structure

We can use a `Manager` to share access to a Python data structure like a list or dict among multiple processes.

The shared data structure can be read and modified from multiple processes in a safe manner, meaning that it will not become corrupt, inconsistent or suffer data loss by concurrent modifications.

In this example we will create a `list` on the manager and share it among a number of processes that will concurrently add objects.

The complete example is listed below.

```python
# SuperFastPython.com
# example of shared list among processes using a manager
from time import sleep
from random import random
from multiprocessing import Process
from multiprocessing import Manager

# custom function to be executed in a child process
def task(number, shared_list):
    # generate a number between 0 and 1
    value = random()
    # block for a fraction of a second
    sleep(value)
    # store the value in the shared list
    shared_list.append((number, value))

# protect the entry point
if __name__ == '__main__':
    # create the manager
    with Manager() as manager:
        # create the shared list
        managed_list = manager.list()
        # create many child processes
        processes = [Process(target=task,
            args=(i, managed_list)) for i in range(50)]
        # start all processes
        for process in processes:
            process.start()
        # wait for all processes to complete
        for process in processes:
            process.join()
        # report the number of items stored
        print(f'List: {len(managed_list)}')
```

Running the example first creates the Manager instance using the context manager interface.

Next, the `Manager` is used to create the shared `list`.

This creates a centralized version of the list on the `Manager`'s server process and returns proxy objects for interacting with the list.

Next, 50 child processes are created and configured to call our custom `task()` function, then the processes are started. The main process then blocks until all child processes complete.

Each process generates a random number, blocks for a fraction of a second then appends a tuple with the task number and the generated value to the shared list.

All tasks complete and the main process unblocks and reports the total number of items in the shared list which matches the same number of processes that were started.

If the list was not created using a manager in this case, then changes to the list in each child process would not be shared. Each worker would be operating on a local copy of the list and changes would not be propagated to other child processes or to the main process.

```
List: 50
```

Next, let's explore how we might use a manager to create a shared concurrency primitive.

How to Use a Manager to Share a Concurrency Primitive

We can use a `Manager` to create a centralized concurrency primitive that can be shared and used safely among multiple processes.

In this example we will define a task that is constrained by a `Semaphore` so that only two instances of the task can run in parallel at any one time. The tasks will be executed by workers in the `Pool` and a `Manager` will be used to create a centralized `Semaphore` that can be used safely in the child worker processes of the pool.

The complete example is listed below.

```python
# SuperFastPython.com
# example of shared semaphore using a manager
from time import sleep
from random import random
from multiprocessing import Manager
from multiprocessing import Pool

# custom function to be executed in a child process
def task(number, shared_semaphore):
    # acquire the shared semaphore
    with shared_semaphore:
        # generate a number between 0 and 1
        value = random()
        # block for a fraction of a second
        sleep(value)
        # report the generated value
        print(f'{number} got {value}')

# protect the entry point
if __name__ == '__main__':
    # create the manager
    with Manager() as manager:
        # create the shared semaphore
        managed_sem = manager.Semaphore(2)
        # create the shared pool
        with Pool() as pool:
            # prepare arguments for task
            args = [(i,managed_sem) for i in range(10)]
            # issue many tasks to the process pool
            pool.starmap(task, args)
```

Running the example first creates the `Manager` instance using the context manager interface.

Next, the `Manager` is used to create the shared `Semaphore` instance with two positions, returning a proxy object for the centralized object that can be shared among processes.

A `Pool` is then created with the default number of workers. A list of tuples is then created as arguments for the task function, one tuple for each call to the task function.

The calls to the `task()` function with arguments are then issued to the `Pool` using the `starmap()` method and the main process blocks until the tasks are completed.

Each task attempts to acquire the semaphore, but only two tasks are able to acquire it at a time. Tasks generate a random number, block for a fraction of a second then report their generated number.

All tasks complete and the `Pool` is shutdown automatically via the context manager interface, followed by the `Manager`.

If the semaphore was not created using the `Manager`, an error would be raised by the `Pool` indicating that a `Semaphore` cannot be passed directly to child worker processes.

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

```
4 got 0.6477163269260953
3 got 0.36257010063693895
5 got 0.5121752319871548
2 got 0.795369090090284
6 got 0.4784123199328061
7 got 0.480959275085588
1 got 0.6841684456946335
9 got 0.5539105856832549
```

Lesson Review

Takeaways

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

- You know what managers are and how they can be used to centralize Python objects in a server process.
- You know how to create and use a manager to create centralized objects and shareable proxy objects.
- You know how to create a centralized Python data structure and share it among multiple processes.
- You know how to create a centralized concurrency primitive to be shared and used in worker processes.

Exercise

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

Develop an example that requires a centralized object be accessed and used within multiple tasks. This could be a shared primitive like a dict, a shared concurrency primitive or a shared pool or queue.

The shared object must be created by the manager and the proxy objects shared with child processes executing each task.

Bonus points if you can develop an example that fails if the manager is not used.

Managers are a little used corner of the `multiprocessing` module and critical for solving the problem of sharing objects with child processes that cannot be pickled or require automatic underlying process-safety. This exercise ensures you are able to bring managers into your future projects, when needed.

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.

- `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 `multiprocessing` module.
- You discovered how to execute your own ad hoc functions in parallel using the `Process` class.
- You discovered how to identify the main process, about parent and child processes, and the life-cycle of processes in Python.
- You discovered how to configure a new child process and access `Process` instance for running processes, kill them, and query their status such as their exit code and whether they are still running.
- You discovered how to coordinate and synchronize Python processes using mutex locks, semaphores, condition variables and the full suite of concurrency primitives.
- You discovered how to inherit and use global variables from parent processes when using the fork start method.
- You discovered how to share data between processes using shared `ctypes`, how to send and receive data using pipes and how to create producer and consumer processes using queues.
- You discovered how to create and configure multiprocessing pools to execute ad hoc tasks using reusable child worker processes.
- You discovered how to process results, handle errors, and query the status of asynchronous tasks executed in multiprocessing pools.
- You discovered how to create centralized Python objects that can be accessed and used in a process-safe manner using proxy objects.

You now know how to use the `multiprocessing` module and bring process-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