*SuperFast*Python

7-Day Course

Python Multiprocessing Pool Jump-Start

Run Your Python Functions In Parallel With Just A Few Lines Of Code

Jason Brownlee

Python Multiprocessing Pool Jump-Start

Run Your Python Functions In Parallel With Just A Few Lines Of Code

Jason Brownlee

2022

Praise for *SuperFast*Python

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

– **Gabriele Berselli**, Python Developer.

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

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

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

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

"*Great as always!!!*"

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

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

– **Marius Rusu**, Python Developer.

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

– **Bilal B.**, Senior Data Engineer.

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

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

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

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

"*This is awesome*"

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

Copyright

Disclaimer

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

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

Preface

Python concurrency is deeply misunderstood.

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

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

Perhaps one of the most common use cases for Python concurrency is to convert a for-loop to run in parallel. Such as calling the same function multiple times with different arguments in a loop.

The multiprocessing pool is the easiest way to convert a for-loop to run in parallel.

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

After completing all seven lessons, you will know how to bring process-based concurrency with the multiprocessing pool 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 pool allows you to create and manage process pools in Python.

Process pools are a design pattern that let you execute and manage heterogeneous, discrete, and ad hoc tasks.

This book provides a jump-start guide to get you productive with the Python multiprocessing pool 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 a result.

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 `Pool` class for CPU-bound tasks.

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 pool 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**: Processes and Multiprocessing Pools.
- **Lesson 02**: Configure the Multiprocessing Pool.
- **Lesson 03**: Execute Tasks in Parallel and Wait.
- **Lesson 04**: Execute Tasks in Parallel and Not Wait.
- **Lesson 05**: Execute Tasks in Parallel and Be Responsive.
- **Lesson 06**: Callbacks and `AsyncResults` for Async Tasks.
- **Lesson 07**: Case Study: Parallel Primality Testing.

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 ten minutes to one hour, although most lessons can be completed within about 20-to-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 if you like. 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/pmpj-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 exercise is 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 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.

The 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 pool 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 pool.

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

- How to create a multiprocessing pool to execute CPU-bound tasks in parallel and in the background with process-based concurrency.
- How to configure the multiprocessing pool including how to set the number of child worker processes, the number of tasks per worker before they are replaced, and how to initialize worker processes.
- How to execute ad hoc tasks in the multiprocessing pool and block until the results are available.
- How to execute tasks asynchronously in the multiprocessing pool, allowing the caller to continue on with other work.
- How to use lazy functions to execute many tasks that use less memory and provide task return values in a way that is more responsive.
- How to work with asynchronous tasks, including how to execute callback functions, how to wait for and get results, and how to check on their status.
- How to build upon what you have learned to speed-up the CPU-bound task of testing if numbers are prime to execute in parallel using the multiprocessing pool.

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, discover process-based concurrency and how to create and run a multiprocessing pool.

Lesson 01: Processes and Multiprocessing Pools

In this lesson, you will discover process-based concurrency in Python and the need for the multiprocessing pool.

After completing this lesson, you will know:

- What is process-based concurrency.
- What are multiprocessing pools and how to create one.
- When to use multiprocessing pools.

Let's get started.

Process-Based Concurrency for Full Parallelism

A process is a computer program.

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

Sometimes we may need to create new processes to run additional tasks concurrently.

Python provides real native (system-level) processes in the `multiprocessing` module via the `Process` class.

The `multiprocessing` module and the `Process` class provide process-based concurrency, as opposed to thread-based concurrency.

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

Process-based concurrency in Python provides true parallelism, that is the ability for a Python program to execute code using more than one CPU core at the same time. This is

unlike threads in Python that only support parallelism when the global interpreter lock (GIL) is released, such as during blocking IO.

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

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

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

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

We can then wait around for the task to complete by joining the process; for example:

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

Whenever we create new processes, it is a good practice to protect the entry point of the program.

When using the `'spawn'` start method, which is the default on Windows and MacOS, not protecting the entry point of the program when using multiprocessing will result in an error. As such, it is a good practice to use the *main module* idiom whenever we are using the `multiprocessing` module.

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

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

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

# 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__':
    # define a task to run in a new process
    process = Process(target=task)
    # start the task in a new process
```

```
    process.start()
    # wait for the child process to terminate
    process.join()
```

Running the example creates an instance of the `Process` class configured to run the `task()` function.

The process is then started by calling the `start()` method. The main process then blocks until the child process terminates.

The child process executes the `task()` function then terminates.

The main process then continues on and the program ends.

```
This is 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.

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, such as 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, which we will look at next.

Process Pools Provide Reusable Workers

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

Python provides a process pool via the `Pool` class, which will take a closer look at next.

Discover the Multiprocessing `Pool` Class

The `Pool` class in the `multiprocessing` module provides a process pool in Python.

Technically, the `Pool` class is in the `multiprocessing.pool` namespace, although an alias allows us to import it directly from the `multiprocessing` namespace.

The `Pool` class allows tasks to be submitted as functions to child worker processes to be executed concurrently.

This section will give just a first taste for the multiprocessing pool, we will spend subsequent lessons looking at each aspect of the pool in detail.

Create a Pool

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

For example:

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

We will learn more about how to configure the multiprocessing pool in Lesson 02.

Issue Tasks and Get Results

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

For example:

```
...
# execute a function asynchronously
async_result = pool.apply_async(task)
```

This returns an `AsyncResult` object that provides a handle on the running task. We can get the result from the running task once it is ready, or simply wait for the task to complete.

```
    ...
# wait for an asynchronous task to complete
async_result.wait()
```

We will learn more about how to issue tasks in Lesson 03, Lesson 04 and Lesson 05.

Close the Pool

Once we have finished with the multiprocessing pool, it can be closed and resources used by the pool can be released.

For example:

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

Once closed, the pool can be joined which is a blocking call that will not return until all issued tasks have completed and all worker processes have closed.

```
...
# join the multiprocessing pool
pool.join()
```

Alternatively, we can forcefully terminate all child worker processes and all running tasks immediately.

```
...
# terminate the multiprocessing pool
pool.terminate()
```

Context Manager Interface

Finally, 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.

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

Example

A complete example of creating a multiprocessing pool to run an ad hoc task function is listed below.

```
# SuperFastPython.com
# example running a function in the multiprocessing pool
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 the task
        async_result = pool.apply_async(task)
        # wait for the task to finish
        async_result.wait()
    # close the multiprocessing pool automatically
```

Running the example first creates a multiprocessing `Pool` object using the default configuration via the context manager interface.

The `Pool` object creates a default number of child workers internally that wait for work to be submitted.

A one-off task is issued to the `Pool` to execute the `task()` function asynchronously and an `AsyncResult` object is returned as a handle on the task.

The main process then waits on the `AsyncResult` object for the issued task to complete.

A worker child process executes the `task()` function and a message is reported.

The main process then continues on and the `Pool` is close automatically by the context manager interface.

```
This is another process
```

A lot happened in this small example, don't worry too much about the details yet as we will dive into each aspect in later lessons.

Nevertheless, this is a helpful template for running ad hoc functions asynchronously in child worker processes.

Next, let's consider when we should use the multiprocessing pool in our Python programs.

When to Use the Multiprocessing Pool

The multiprocessing pool is powerful and flexible, although is not suited for all situations where we need to run a background task or apply a function to each item in an iterable in parallel.

Use the multiprocessing pool when:

- Your tasks can be defined by a pure function that has no state or side effects.
- Your task can fit within a single Python function, likely making it simple and easy to understand.
- You need to perform the same task many times with different arguments, e.g. homogeneous tasks.
- You need to apply the same function to each object in a collection in a for-loop.

Process pools work best when applying the same pure function on a set of different data, e.g. homogeneous tasks, heterogeneous data. This makes code easier to read and debug. This

is a recommendation, not a rule.

Use the Multiprocessing Pool for CPU-Bound Tasks

You should use processes for CPU-bound tasks.

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

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 include mathematical operations such as calculating the points of a fractal, estimating pi, and factoring primes. It is also appropriate for computational intensive operations such as parsing text documents, encoding images or video and running simulations.

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

Using processes and multiprocessing pools via the `Pool` class in Python is probably the best path toward achieving this end.

Lesson Review

Takeaways

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

- You are now familiar with process-based concurrency.
- You know that process pools are a helpful pattern for executing ad hoc tasks.
- You know how to create a multiprocessing pool and issue a task.
- You know that multiprocessing pools are appropriate for CPU-bound tasks.

Exercise

Your task for this lesson is to list examples of tasks where you think the multiprocessing pool may be helpful.

Some tips to help you:

- Recall that the multiprocessing pool is suited to CPU-bound tasks.
- Recall that the pool supports both one-off tasks as well as calling the same function many times with different data.
- Consider general computer science tasks.
- Consider aspects of common business logic.
- Consider other examples shared online.

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)
- CPU-bound, Wikipedia.
  https://en.wikipedia.org/wiki/CPU-bound
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html

Next

In the next lesson, we will discover how to configure the `Pool` class.

Lesson 02: Configure the Multiprocessing Pool

In this lesson, you will discover how to configure the multiprocessing pool.

After completing this lesson, you will know:

- How to configure the number of child worker processes.
- How to configure the worker initializer function and process start method.
- How to configure the maximum tasks per child worker process.

Let's get started.

Configure the Pool via the Constructor

We can configure a `Pool` object by specifying arguments to the object constructor.

There are many aspects of the multiprocessing pool that we can configure via arguments, such as:

- `processes`: Maximum number of worker processes to use in the pool.
- `initializer`: Function executed after each worker process is created.
- `initargs`: Arguments to the worker process initialization function.
- `maxtasksperchild`: Limit the maximum number of tasks executed by each worker process.
- `context`: Configure the multiprocessing context such as the process start method.

Next, let's look at the default configuration for a `Pool` object.

How to Get the `Pool` Default Configuration

A `Pool` object can be constructed without any arguments.

For example:

```
...
# create a default multiprocessing pool
pool = Pool()
```

This will create a multiprocessing pool that will use a number of worker processes that matches the number of logical CPU cores in our system.

- Default Number Worker Processes = Total Logical CPU Cores in Your System

For example, if we had 2 physical CPUs in our system and each CPU has hyperthreading (common in modern CPUs) then we would have 2 physical and 4 logical CPUs. Python would see 4 CPUs. The default number of child worker processes on our system would then be 4.

By default, the multiprocessing pool will not call a function that initializes the worker processes when they are created.

Each worker process will be able to execute an unlimited number of tasks within the pool.

Finally, the default multiprocessing context will be used, along with the currently configured or default start method for the system.

The example below creates a multiprocessing pool, then prints its details.

This provides an easy way to see the default number of worker processes created for multiprocessing pools on our system.

```
# SuperFastPython.com
# example reporting the details of a default pool
from multiprocessing import Pool

# protect the entry point
if __name__ == '__main__':
    # create a multiprocessing pool
    pool = Pool()
    # report the status of the multiprocessing pool
    print(pool)
    # close the multiprocessing pool
    pool.close()
```

Running the example creates a multiprocessing pool, then reports the number of workers created by default.

In this case, we can see that 8 workers were created on my system. This is because I have a system with 4 physical CPU cores with hyperthreading, resulting in 8 logical CPU cores.

```
<multiprocessing.pool.Pool state=RUN pool_size=8>
```

Next, let's discover how we might configure each aspect of the `Pool` class.

How to Configure the Number of Worker Processes

We can configure the number of worker processes in the `Pool` by setting the `processes` argument in the constructor.

We can set the `processes` argument to specify the number of child processes to create and use as workers in the multiprocessing pool.

For example:

```
...
# create a multiprocessing pool with 4 workers
pool = Pool(processes=4)
```

The `processes` argument is the first argument in the constructor and does not need to be specified by name to be set, for example:

```
...
# create a multiprocessing pool with 4 workers
pool = Pool(4)
```

If we are using the context manager to create the multiprocessing pool so that it is automatically shutdown, then we can configure the number of processes in the same manner.

For example:

```
...
# create a multiprocessing pool with 4 workers
with Pool(4):
    # ...
```

You should probably set the number of processes to be equal to the number of logical CPU cores in our system, e.g. the default.

If we are expecting to perform computational work in the main process in addition to the multiprocessing pool, consider setting the number of processes in the pool to be equal to the number of logical CPUs in our system minus one, to allow the main process to execute.

If we have particularly CPU intensive tasks, consider configuring the number of processes to be equal to the number of physical CPUs instead of the number of logical CPUs.

Next, let's look at how we might configure the worker process initialization function.

How to Configure the Worker Process Initialization

We can configure worker processes in the multiprocessing pool to execute an initialization function prior to executing tasks.

This can be achieved by setting the `initializer` argument when configuring the multiprocessing pool via the class constructor.

The `initializer` argument can be set to the name of a function that will be called to initialize the worker processes.

For example:

```
# worker process initialization function
def worker_init():
    # ...

...
# create a multiprocessing pool and initialize workers
pool = Pool(initializer=worker_init)
```

If our worker process initialization function takes arguments, they can be specified to the multiprocessing pool constructor via the `initargs` argument, which takes an ordered list or tuple of arguments for the custom initialization function.

For example:

```
# worker process initialization function
def init(arg1, arg2):
    # ...

...
# create a multiprocessing pool and initialize workers
pool = Pool(initializer=init, initargs=(arg1, arg2))
```

We can explore an example of calling a custom function to initialize each process in the multiprocessing pool.

In this example we will define a task to simulate work that will report a message and block for a moment. We will also define a simple worker process initialization function that will simply report a message. We will then configure a multiprocessing pool to initialize the workers with our initialization function and execute a number of tasks.

The complete example is listed below.

```
# SuperFastPython.com
# example initializing worker processes in the pool
from time import sleep
from multiprocessing import Pool

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

# initialize a worker in the multiprocessing pool
def initialize_worker():
    # report a message
```

```
    print('Initializing worker...', flush=True)

# protect the entry point
if __name__ == '__main__':
    # create and configure the multiprocessing pool
    with Pool(2, initializer=initialize_worker) as pool:
        # issue tasks to the multiprocessing pool
        for _ in range(4):
            pool.apply_async(task)
        # close the multiprocessing pool
        pool.close()
        # wait for all tasks to complete
        pool.join()
```

Running the example first creates and configures the multiprocessing pool.

Tasks are issued to the pool and worker processes are created as needed to execute the tasks.

After the worker processes are created they are initialized, then start executing the issued tasks.

Importantly, each worker process is initialized once and only before it begins consuming and completing tasks in the pool.

```
Initializing worker...
Worker executing task...
Initializing worker...
Worker executing task...
Worker executing task...
Worker executing task...
```

Next, let's look at how we might configure the maximum tasks per child worker process.

How to Configure the Maximum Tasks Per Worker

The multiprocessing pool allows us to specify the number of tasks that each worker may execute before being replaced with a new worker process.

This can be achieved by setting the `maxtasksperchild` argument in the `Pool` class constructor when configuring a new multiprocessing pool.

For example:

```
...
# create pool and limit number of tasks for each worker
pool = Pool(maxtasksperchild=5)
```

The `maxtasksperchild` takes a positive integer number of tasks that may be completed by

a child worker process, after which the process will be terminated and a new child worker process will be created to replace it.

By default the `maxtasksperchild` argument is set to `None`, which means each child worker process will run for the lifetime of the multiprocessing pool.

Next, let's look at how we might configure the multiprocessing context for the pool.

How to Configure the Context Used to Create Workers

We can set the context for the multiprocessing pool via the `context` argument to the `Pool` class constructor.

The `context` is an instance of a multiprocessing context configured with a start method, created via the `get_context()` function.

By default, `context` is `None`, which uses the current default context and start method configured for the application.

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.

The `'spawn'` start method is the default on Windows and MacOS, whereas `'fork'` is the default for Linux and may not be supported on Windows.

Multiprocessing contexts provide a more flexible way to manage process start methods directly within a program, and may be a preferred approach to changing start methods in general, especially within a Python library.

A new context can be created with a given start method and passed to the multiprocessing pool.

For example:

```
...
# create a process context
ctx = get_context('fork')
# create a multiprocessing pool with a given context
pool = Pool(context=ctx)
```

Lesson Review

Takeaways

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

- You discovered how to configure the number of child worker processes.
- You discovered how to initialize the child worker processes.
- You discovered how to configure the number of tasks executed by each child worker process before being replaced.
- You discovered how to configure the method used to start child worker processes.

Exercise

Your task for this lesson is to create a small example that uses the multiprocessing pool where the pool is configured with the number of physical CPU cores in your system.

This will require that you know how many CPU cores are in your system which might require checking the system properties of your operating system.

For bonus points, confirm that the default number of worker processes created in the multiprocessing pool is double the number of physical CPU cores. This can be achieved by reporting the number of active child processes in your program via the `active_children()` function from the `multiprocessing` module when using a multiprocessing pool with the default configuration.

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.

- Hyper-threading, Wikipedia.
  https://en.wikipedia.org/wiki/Hyper-threading
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html
- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html

Next

In the next lesson, we will discover how to execute tasks synchronously in the multiprocessing pool.

Lesson 03: Execute Tasks in Parallel and Wait

In this lesson, you will discover how to execute tasks synchronously with the multiprocessing pool.

After completing this lesson, you will know:

- How to run one-off tasks and wait for them to finish.
- How to run multiple calls to the same function with different arguments, such as in a loop.
- How to run multiple calls to the same function with multiple arguments.

Let's get started.

Issue Tasks to the Pool Synchronously (blocking)

An important consideration when issuing tasks is whether the function used to issue the task blocks until the tasks are complete or not.

Recall that a blocking call does not return until the call is complete. This means the caller cannot perform any actions until all tasks are issued and finished.

Executing tasks synchronously is helpful when we want to wait for the tasks to complete then process their results.

Blocking method calls for executing tasks on the `Pool` class include:

- `apply()`: For executing one-off tasks.
- `map()`: For calling a function many times with different arguments.
- `starmap()`: For calling a function many times with multiple different arguments.

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

How to Execute One-Off Tasks with `apply()`

We can execute one-off tasks with a `Pool` object using the `apply()` method.

The `apply()` method takes the name of the function to execute by a worker process. The call will block until the function is executed by a worker process, after which time it will return.

For example:

```
...
# execute a task in the multiprocessing pool
pool.apply(task)
```

The `apply()` method is a parallel version of the now deprecated built-in `apply()` function.

The `apply()` method can execute a function that takes no arguments or an arbitrary number of arguments. Arguments to the target function can be provided as a tuple to the `args` keyword or a dictionary to the `kwds` argument.

For example:

```
...
# execute a task in the pool with multiple arguments
pool.apply(task, args=(arg1, arg2, arg3))
```

In summary, the capabilities of the `apply()` method are as follows:

- Issues a single task to the multiprocessing pool.
- Supports multiple arguments to the target function.
- Blocks until the call to the target function is complete.

The example below demonstrates how to execute a task using the `apply()` method.

```
# SuperFastPython.com
# example of executing a one-off task and waiting
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 and wait for it to complete
        pool.apply(task)
```

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
```

Next, let's take a look at how to use the `map()` method.

How to Execute Many Tasks with `map()`

The `Pool` class provides a parallel version of the built-in `map()` function for issuing tasks.

The `map()` method takes the name of a target function and an iterable. A task is created 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 `Pool` object's `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.

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.

In summary, the capabilities of the `map()` method are as follows:

- Issue multiple tasks to the multiprocessing pool all at once.
- Returns an iterable over return values.
- Supports a single argument to the target function.
- Blocks until all issued tasks are completed.
- Allows tasks to be grouped and executed in batches by workers.

The example below demonstrates how to execute multiple tasks using the `map()` method.

```
# SuperFastPython.com
# example of executing multiple tasks and waiting
from multiprocessing import Pool

# custom function to be executed in a child process
def task(arg):
    # report a message
    print(f'From another process {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)):
            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.

```
From another process 0
From another process 1
From another process 2
From another process 3
From another process 4
From another process 5
From another process 6
From another process 7
From another process 8
From another process 9
0
2
4
6
8
10
12
14
16
18
```

Next, let's take a look at how to use the `starmap()` method.

How to Execute Many Tasks with Multiple Arguments

We can issue multiple tasks to the multiprocessing pool using the `starmap()` method.

The `starmap()` method takes the name of a target function and an iterable. A task is created to call the target function for each item in the provided iterable. Each item in the iterable is

itself an iterable, allowing multiple arguments to be provided to the target function.

It returns an iterable over the return values from each call to the target function. The iterable is first traversed and all tasks are issued at once.

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)
```

Like the `map()` method, a `chunksize` argument can be specified to split the tasks into groups which may be sent to each worker process to be executed in a batch.

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, chunksize=10):
    print(result)
```

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

In summary, the capabilities of the `starmap()` method are as follows:

- Issue multiple tasks to the multiprocessing pool all at once.
- Returns an iterable over return values.
- Supports multiple arguments to the target function.
- Blocks until all issued tasks are completed.
- Allows tasks to be grouped and executed in batches by workers.

The example below demonstrates how to execute multiple tasks using the `starmap()` method.

```
# SuperFastPython.com
# example of multiple tasks with multiple arguments
from multiprocessing import Pool

# custom function to be executed in a child process
def task(arg1, arg2, arg3):
    # report a message
    print(f'From another process {arg1}, {arg2}, {arg3}',
        flush=True)
    # return a value
    return arg1 + arg2 + arg3
```

```
# protect the entry point
if __name__ == '__main__':
    # create the multiprocessing pool
    with Pool() as pool:
        # prepare task arguments
        args = [(i, i*2, i*3) for i in range(10)]
        # issue multiple tasks and process return values
        for result in pool.starmap(task, args):
            print(result)
```

Running the example first prepares a list of tuples, each item in the list representing the arguments for a single task executed in the multiprocessing pool.

Then, 10 tasks are issued to the multiprocessing pool, each with multiple arguments.

The tasks are executed as workers become available and report a message with their arguments.

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

```
From another process 0, 0, 0
From another process 1, 2, 3
From another process 2, 4, 6
From another process 3, 6, 9
From another process 4, 8, 12
From another process 5, 10, 15
From another process 6, 12, 18
From another process 7, 14, 21
From another process 8, 16, 24
From another process 9, 18, 27
0
6
12
18
24
30
36
42
48
54
```

Lesson Review

Takeaways

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

- You know how to execute ad hoc tasks to the multiprocessing pool with `apply()`.
- You know how to execute a function for each item in an iterable with `map()`.
- You know how to execute a function with multiple arguments multiple times with `starmap()`.

Exercise

Your task for this lesson is to execute a for loop in parallel using one of the synchronous methods described above.

Firstly, develop a small program that executes a task in a loop, such as calling the same function multiple times with different arguments.

The task should perform an operation that is slow and would benefit from being sped up. See examples from Lesson 01 to give you ideas.

Once you have developed your example, update it to execute in parallel using one of the synchronous methods for executing tasks in the multiprocessing pool, e.g. one of `apply()`, `map()` or `starmap()`.

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
- `apply` built-in function API.
  https://docs.python.org/2/library/functions.html#apply
- `map` built-in function API.
  https://docs.python.org/3/library/functions.html#map
- `itertools.starmap` API.
  https://docs.python.org/3/library/itertools.html#itertools.starmap

Next

In the next lesson, we discover how to execute tasks asynchronously with the multiprocessing pool.

Lesson 04: Execute Tasks in Parallel and Not Wait

In this lesson, you will discover how to execute tasks asynchronously in the multiprocessing pool.

After completing this lesson, you will know:

- How to issue one-off tasks asynchronously.
- How to issue multiple calls to the same function with different arguments asynchronously.
- How to issue multiple calls to the same function with multiple arguments asynchronously.

Let's get started.

Issue Tasks to the Pool Asynchronously (non-blocking)

We typically want to execute tasks with the multiprocessing pool without blocking.

Recall that a non-blocking call returns immediately, not waiting for the called function to complete and return.

When we execute tasks asynchronously in the multiprocessing pool, it provides a hook or mechanism to check the status of the tasks and get the results later.

The caller can issue tasks and carry on with the program, fire-and-forget or fire-and-monitor.

Non-blocking method calls on the `Pool` class include:

- Use `apply_async()`: For executing one-off tasks.
- Use `map_async()`: For calling a function many times with different arguments.
- Use `starmap_async()`: For calling a function many times with multiple different arguments.

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

How to Issue One-Off Tasks Asynchronously

We can execute asynchronous one-off tasks on a `Pool` object using the `apply_async()` method.

The `apply_async()` method takes the name of the function to execute in a worker process and returns immediately with a `AsyncResult` object for the task.

For example:

```
...
# issue a task asynchronously to the pool
async_result = pool.apply_async(task)
```

Like the `apply()` method, the `apply_async()` method can execute a target function that takes no arguments or an arbitrary number of arguments. Arguments to the target function can be provided as a tuple to the `args` keyword or a dictionary to the `kwds` argument.

For example:

```
...
# execute a task in the pool with multiple arguments
async_result = pool.apply_async(task, args=(arg1, arg2))
```

Later the status of the issued task may be checked or retrieved.

For example:

```
...
# get the result from the issued task
value = async_result.get()
```

We will learn more about checking the status of asynchronous tasks and getting results in Lesson 06.

In summary, the capabilities of the `apply_async()` method are as follows:

- Issues a single task to the multiprocessing pool asynchronously.
- Supports multiple arguments to the target function.
- Does not block, instead returns a `AsyncResult`.
- Supports callback for the return value and any raised errors.

We will explore callback functions for return values and raised errors in Lesson 06.

The example below demonstrates how to execute multiple tasks using the `apply_async()` method.

```
# SuperFastPython.com
# example of executing a one-off task asynchronously
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 issued task returns an `AsyncResult` object immediately, providing a handle on the asynchronous task.

The caller is then free to continue on.

The caller then calls the `wait()` method on the `AsyncResult` object to explicitly wait for the issued task to complete.

```
This is another process
```

Next, let's take a look at how to use the `map_async()` method.

How to Issue Many Tasks Asynchronously

The `Pool` class provides an asynchronous version of the `map()` method for executing tasks called `map_async()`.

The `map_async()` method takes the name of a target function and an iterable. A task is created to call the target function for each item in the provided iterable. It does not block and returns immediately with an `AsyncResult` object that may be used to access the results.

The iterable argument is first traversed and all tasks are issued at once.

For example:

```
...
# issue tasks to the multiprocessing pool asynchronously
async_result = pool.map_async(task, items)
```

Like the `map()` method, a `chunksize` argument can be specified to split the tasks into groups which may be sent to each worker process to be executed in a batch.

For example:

```
...
# issue tasks to the multiprocessing pool asynchronously
async_result = pool.map_async(task, items, chunksize=10)
```

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

Later the status of the tasks can be checked and the return values from each call to the target function may be iterated.

For example:

```
...
# iterate over return values from the issued tasks
for value in async_result.get():
    # ...
```

In summary, the capabilities of the `map_async()` method are as follows:

- Issue multiple tasks to the multiprocessing pool all at once asynchronously.
- Supports a single argument to the target function.
- Does not block, instead returns a `AsyncResult` for accessing results later.
- Allows tasks to be grouped and executed in batches by workers.
- Supports callback for the return value and any raised errors.

The example below demonstrates how to execute multiple tasks using the `map_async()` method.

```
# SuperFastPython.com
# example executing multiple tasks asynchronously
from multiprocessing import Pool

# custom function to be executed in a child process
def task(arg):
    # report a message
    print(f'From another process {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 to the pool
        async_result = pool.map_async(task, range(10))
        # process return values once all tasks completed
        for result in async_result.get():
            print(result)
```

Running the example first issues 10 tasks to the multiprocessing pool and returns an `AsyncResult` object immediately.

The caller is then free to continue on.

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

The caller then calls the `get()` method on the `AsyncResult` object which will return an

iterable of return values once all tasks have completed. The iterable is then traversed, reporting each return value.

```
From another process 0
From another process 1
From another process 2
From another process 3
From another process 4
From another process 5
From another process 6
From another process 7
From another process 8
From another process 9
0
2
4
6
8
10
12
14
16
18
```

Next, let's take a look at how to use the `starmap_async()` method.

How to Issue Tasks with Multiple Arguments Asynchronously

We can issue multiple tasks asynchronously to the multiprocessing pool using the `starmap_async()` method.

The `starmap_async()` method takes the name of a target function and an iterable. A task is created to call the target function for each item in the provided iterable.

Each item in the iterable itself must be an iterable, allowing multiple arguments to be provided to the target function.

It does not block and returns immediately with an `AsyncResult` that may be used to access the return values of the target function.

The iterable is first traversed and all tasks are issued all at once.

For example:

```
...
# prepare an iterable of iterables for each task
items = [(1,2), (3,4), (5,6)]
```

```
# issue tasks to the multiprocessing pool asynchronously
async_result = pool.starmap_async(task, items)
```

Later the status of the tasks can be checked and the return values from each call to the target function may be iterated.

For example:

```
...
# iterate over return values from the issued tasks
for value in async_result.get():
    # ...
```

In summary, the capabilities of the `starmap_async()` method are as follows:

- Issue multiple tasks to the multiprocessing pool all at once.
- Supports multiple arguments to the target function.
- Does not block, instead returns a `AsyncResult` for accessing results later.
- Allows tasks to be grouped and executed in batches by workers.
- Supports callback for the return value and any raised errors.

The example below demonstrates how to execute multiple tasks using the `starmap_async()` method.

```
# SuperFastPython.com
# example many tasks multiple arguments asynchronously
from multiprocessing import Pool

# custom function to be executed in a child process
def task(arg1, arg2, arg3):
    # report a message
    print(f'From another process {arg1}, {arg2}, {arg3}',
        flush=True)
    # return a value
    return arg1 + arg2 + arg3

# protect the entry point
if __name__ == '__main__':
    # create the multiprocessing pool
    with Pool() as pool:
        # prepare task arguments
        args = [(i, i*2, i*3) for i in range(10)]
        # issue multiple tasks to the pool
        async_result = pool.starmap_async(task, args)
        # process return values
        for result in async_result.get():
            print(result)
```

Running the example first prepares a list of tuples, each item in the list representing the arguments for a single task executed in the multiprocessing pool.

Then, 10 tasks are issued to the multiprocessing pool, each with multiple arguments. The call returns immediately with an `AsyncResult` object.

The caller is then free to continue on.

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

The caller then calls the `get()` method on the `AsyncResult` object which will return an iterable of return values once all tasks have completed. The iterable is then traversed, reporting each return value.

```
From another process 0, 0, 0
From another process 1, 2, 3
From another process 2, 4, 6
From another process 3, 6, 9
From another process 4, 8, 12
From another process 5, 10, 15
From another process 6, 12, 18
From another process 7, 14, 21
From another process 8, 16, 24
From another process 9, 18, 27
0
6
12
18
24
30
36
42
48
54
```

Lesson Review

Takeaways

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

- You discovered how to execute an ad hoc function asynchronously with the multiprocessing pool.
- You discovered how to execute multiple calls to the same function with different arguments asynchronously.
- You discovered how to execute multiple calls to the same function with multiple different arguments asynchronously.

Exercise

Your task for this lesson is to execute a for loop in parallel using one of the asynchronous methods described above.

Firstly, develop a small program that executes a task in a loop, such as calling the same function multiple times with different arguments.

The example must be devised in a way that the caller can perform some other operation while the tasks are executing.

The task should perform an operation that is slow and would benefit from being sped up. See examples from Lesson 01 to give you ideas.

Once you have developed your example, update it to execute in parallel using one of the asynchronous methods for executing tasks in the multiprocessing pool, e.g. one of `apply_async()`, `map_async()` or `starmap_async()`.

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

In the next lesson, we will discover lazy versions of the `map()` method for executing tasks on a `Pool`.

Lesson 05: Execute Tasks in Parallel and Be Responsive

In this lesson, you will discover the limitations of the `map()` method for executing tasks in the multiprocessing pool and a lazy version of the `map()` method called the `imap()` method that overcomes these limitations.

After completing this lesson, you will know:

- The limitations of the `map()` method.
- How to issue tasks as needed and process results as they become available.
- How to process results from issued tasks in the order they are completed.

Let's get started.

How to Overcome the Limitations of `map()`

A problem with the `map()` method on the `Pool` 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. This is because the multiprocessing pool will then have hundreds, thousands, or millions of tasks sitting idly waiting to execute, using large amounts of main memory unnecessarily.

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 in a lazy manner.

Specifically:

- Argument items are yielded from the iterable as workers become available, rather than of all at once.
- Return values are yielded in order as they are completed, rather than after all tasks are completed.

This is great, but we can go one step further.

A limitation of `imap()` method is that it yields return value results in the order that the tasks were issued to the multiprocessing pool. Although results are yielded as tasks are completed,

the caller may not be as responsive as it could be if later tasks complete before earlier tasks, holding up the progress of the iterable of return values.

The `imap_unordered()` addresses this limitation. It returns an iterable of return values. The return values are yielded in the order that tasks are completed, not the order that the tasks were issued to the multiprocessing pool.

Both `imap()` and `imap_unordered()` are blocking calls for issuing tasks to the multiprocessing pool, but are more lazy than the `map()` method.

Let's take a closer look at each method in turn starting with the `imap()` method.

How to Issue Tasks One-by-One with `imap()`

We can issue tasks to a `Pool` object one-by-one via the `imap()` method.

The `imap()` method takes the name of a target function and an iterable. A task is created 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. The iterable will yield return values as tasks are completed, in the order that tasks were issued.

The `imap()` method is lazy in that it traverses the provided iterable and issues tasks to the multiprocessing pool one by one as space becomes available in the multiprocessing pool.

For example:

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

Like the `map()` method, a `chunksize` argument can be set to batch tasks into groups for each child worker process, potentially offering a speed-up if there are a large number of tasks.

In summary, the capabilities of the `imap()` method are as follows:

- Issue multiple tasks to the multiprocessing pool, one-by-one.
- Returns an iterable over return values, results are yielded as tasks complete.
- Supports a single argument to the target function.
- Blocks until each task is completed in order they were issued.
- Allows tasks to be grouped and executed in batches by workers.

Worked Example

The example below demonstrates how to execute multiple tasks using the `imap()` method.

```
# SuperFastPython.com
# example of executing multiple tasks one-by-one
from time import sleep
```

```
from random import random
from multiprocessing import Pool

# custom function to be executed in a child process
def task(arg):
    # block for a random fraction of a second
    sleep(random())
    # report a message
    print(f'From another process {arg}', flush=True)
    # return a value
    return arg * 2

# protect the entry point
if __name__ == '__main__':
    # create the multiprocessing pool
    with Pool(4) as pool:
        # issue multiple tasks and process return values
        for result in pool.imap(task, range(10)):
            print(result)
```

Running the example first starts the multiprocessing pool.

Tasks are issued to the multiprocessing pool one-by-one as worker processes become available.

As tasks are completed, return values are yielded by the returned iterable reported by the caller in the main process.

In this case, we can see that messages reported by tasks are interleaved with messages with return values by the caller. Messages with return values are in the order that the tasks were issued.

This is different from the `map()` method that would issue all tasks at once, then wait for all issued tasks to complete returning an iterable of return values.

```
From another process 3
From another process 2
From another process 1
From another process 4
From another process 0
0
2
4
6
8
From another process 6
From another process 9
From another process 5
```

```
10
12
From another process 8
From another process 7
14
16
18
```

Next, let's take a look at how to use the `imap_unordered()` method.

How to Get Results in Task Completion Order

We can issue tasks to the multiprocessing pool one-by-one via the `imap_unordered()` method.

The `imap_unordered()` method takes the name of a target function and an iterable. A task is created 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. The iterable will yield return values as tasks are completed, in the order that tasks were completed, not the order they were issued.

The `imap_unordered()` method is lazy in that it traverses the provided iterable and issues tasks to the multiprocessing pool one by one as space becomes available in the multiprocessing pool.

For example:

```
...
# iterate results as tasks complete, in order completed
for result in pool.imap_unordered(task, items):
    # ...
```

Like `map()` and `imap()`, the `imap_unordered()` supports a `chunksize` argument that allows tasks to be grouped into batches sent to each child worker process.

In summary, the capabilities of the `imap_unordered()` method are as follows:

- Issue multiple tasks to the multiprocessing pool, one-by-one.
- Returns an iterable over return values, results are yielded as tasks complete.
- Supports a single argument to the target function.
- Blocks until each task is completed in the order they are completed.
- Allows tasks to be grouped and executed in batches by workers.

The example below demonstrates how to execute multiple tasks using the `imap_unordered()` method.

```
# SuperFastPython.com
# example getting many task results in completion order
from time import sleep
```

```
from random import random
from multiprocessing import Pool

# custom function to be executed in a child process
def task(arg):
    # block for a random fraction of a second
    sleep(random())
    # report a message
    print(f'From another process {arg}', flush=True)
    # return a value
    return arg * 2

# protect the entry point
if __name__ == '__main__':
    # create the multiprocessing pool
    with Pool(4) as pool:
        # issue multiple tasks and process return values
        for rs in pool.imap_unordered(task, range(10)):
            print(rs)
```

Running the example first starts the multiprocessing pool.

Tasks are issued to the multiprocessing pool one-by-one as worker processes become available.

As tasks are completed, return values are yielded by the returned iterable reported by the caller in the main process.

In this case, we can see that messages reported by tasks are interleaved with messages with return values by the caller. Messages with return values are not in the order that the tasks were issued.

Return values are yielded and reported in the order that issued tasks were completed. This is unlike the `imap()` method that always yields return values in the order that tasks were issued.

```
From another process 1
2
From another process 0
0
From another process 2
4
From another process 3
6
From another process 7
14
From another process 6
12
```

```
From another process 5
10
From another process 4
8
From another process 8
16
From another process 9
18
```

Lesson Review

Takeaways

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

- You discovered two key limitations of the `map()` method for executing tasks in the multiprocessing pool, namely all tasks are issued at once and return values are yielded in order.
- You discovered how to execute tasks in the multiprocessing pool with a lazy version of `map()` called `imap()`.
- You discovered how to process return values in the order that tasks are completed using `imap_unordered()`.

Exercise

Your task for this lesson is to develop an example to demonstrate a key capability of the `imap()` or `imap_unordered()` methods.

Recall that both methods overcome limitations of the `map()` method, specifically by issuing tasks one-by-one using less memory and yielding return values as tasks complete being more responsive.

Develop an example using either method to highlight one of these key benefits.

You can use one of the examples above as a starting point, but try to develop a program that does something useful, such as performing a computationally expensive calculation, parsing or encoding raw data, or similar ideas from Lesson 01.

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
- `itertools.imap` API.
  https://docs.python.org/2/library/itertools.html#itertools.imap
- `random` - Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html

Next

In the next lesson, we will discover how to use callbacks for asynchronous tasks and how to query the status of asynchronous tasks in the multiprocessing pool.

Lesson 06: Callbacks and `AsyncResults` for Async Tasks

In this lesson, you will discover how to use callbacks with asynchronous tasks and how to check on the status of asynchronous tasks using the `AsyncResult` object.

After completing this lesson, you will know:

- How to add callback functions to asynchronous tasks.
- How to wait for and get results from asynchronous tasks.
- How to check the status of asynchronous tasks.

Let's get started.

How to Configure a Callback Function

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 methods: `apply_async()`, `map_async()` or `starmap_async()`.

Callback functions are called in two situations:

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

Use a Callback Function To Handle Results

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()` method 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)
```

Use a Callback Function for Unhandled Errors in Tasks

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.

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 callback(error):
    print(error, flush=True)

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

Example of Using a Result Callback Function

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 callback function for a one-off async 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(identifier):
    # generate a value
    value = random()
    # report a message
    print(f'Task {identifier} executing 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 executing with 0.2998712545246843
Callback got: 0.2998712545246843
```

Next, let's explore how we can wait for and get results from asynchronous tasks.

How to Manage Asynchronous Tasks with `AsyncResult`

An `AsyncResult` object is returned when issuing tasks to `Pool` the multiprocessing pool asynchronously.

This can be achieved via any of the following methods on the multiprocessing pool asynchronously: `apply_async()`, `map_async()`, or `starmap_async()`.

Once we have a `AsyncResult`, we can use it to interact with the task in two ways:

- Get the result of the issued task or tasks.
- Wait for the issued task or all issued tasks to complete.

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

Retrieve the Results

We can get the result of an issued task by calling the `get()` method.

This will return the result of the specific function called to issue the task, such as a single return value in the case of the `apply_async()` method or an iterable of return values otherwise.

For example:

```
...
# get the result of the task or tasks
value = async_result.get()
```

If the issued tasks have not yet completed, then `get()` will block until the tasks are finished.

A `timeout` argument can be specified. If the tasks are still running and do not complete within the specified number of seconds, a `TimeoutError` is raised.

For example:

```
...
try:
    # get the task result with a timeout
    value = async_result.get(timeout=10)
```

```
except TimeoutError as e:
    # ...
```

If an issued task raises an exception, the exception will be re-raised once the issued tasks are completed and we get the return value.

We may need to handle this case explicitly if we expect a task to raise an exception on failure.

For example:

```
...
try:
    # get the task result that might raise an exception
    value = async_result.get()
except Exception as e:
    # ...
```

Wait for Tasks to Finish

We can wait for all tasks to complete via the `wait()` method.

This will block until all issued tasks are completed.

Completed means that the task or tasks completed successfully, or failed with an unhandled error or exception.

For example:

```
...
# wait for issued task to complete
async_result.wait()
```

If the tasks have already completed, then the `wait()` method will return immediately.

A `timeout` argument can be specified to set a limit in seconds for how long the caller is willing to wait.

If the timeout expires before the tasks are complete, the `wait()` method will return.

When using a timeout, the `wait()` method does not give an indication that it returned because tasks completed or because the timeout elapsed. Therefore, we can check if the tasks completed via the `ready()` method (covered later in this lesson).

For example:

```
...
# wait for issued task to complete with a timeout
async_result.wait(timeout=10)
# check if the tasks are all done
if async_result.ready()
    print('All Done')
    ...
```

```
else :
    print('Not Done Yet')
    ...
```

Next, let's look at how we can check the status of an asynchronous task via the `AsyncResult` object.

How to Check the Status of Tasks with `AsyncResult`

We can check on the status of an asynchronous task via its `AsyncResult`.

There are three aspects of an asynchronous task that we can check via the `AsyncResult`.

They are:

- Check if the task or tasks have been completed.
- Check if all tasks were successful.

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

Check if Tasks Are Completed

We can check if the issued tasks are completed via the `ready()` method.

It returns `True` if the tasks have completed, successfully or otherwise, or `False` if the tasks are still running.

For example:

```
...
# check if tasks are still running
if async_result.ready():
    print('Tasks are done')
else:
    print('Tasks are not done')
```

Check if Tasks Finished Normally

We can check if the issued tasks completed successfully via the `successful()` method.

Issued tasks are successful if no tasks raised an exception.

If at least one issued task raised an exception, then the call was not successful and the `successful()` method will return `False`.

This method should be called after it is known that the tasks have completed, e.g. `ready()` returns `True`.

For example:

```
...
# check if the tasks have completed
if async_result.ready():
    # check if the tasks were successful
    if async_result.successful():
        print('Successful')
    else:
        print('Unsuccessful')
```

If the issued tasks are still running, a `ValueError` is raised and may need to be handled.

For example:

```
...
try:
    # check if the tasks were successful
    if async_result.successful():
        print('Successful')
except ValueError as e:
    print('Tasks still running')
```

Worked Example of Checking the Status of a Task

We can explore how to handle an asynchronous task via its `AsyncResult` object.

In this example we will define a long running task to run asynchronously. The task will loop 10 times and each iteration it will generate a random number between 0 and 1, block for a fraction of a second then report the value that was generated.

The main process will execute the task asynchronously then loop, waiting for the task to complete. It will check if the task has completed via the `ready()` method, and wait for the task to complete via the `wait()` method with a timeout.

The timeout allows the status of the task to be explicitly checked often, called a busy-wait loop that can be helpful if the caller wants to give up after a while.

Once the task is completed, the caller will check if the task was completed successfully, without an unhandled error or exception.

The complete example is listed below.

```
# SuperFastPython.com
# example check status and handle result of async task
from time import sleep
from random import random
from multiprocessing import Pool

# custom function to be executed in a child process
```

```
def task():
    # loop a few times to simulate a slow task
    for i in range(10):
        # generate a random value between 0 and 1
        value = random()
        # block for a fraction of a second
        sleep(value)
        # report a message
        print(f'>{i} got {value}', 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 until the task is complete
        while not async_result.ready():
            # report a message
            print('Main process waiting...')
            # block for a moment
            async_result.wait(timeout=1)
        # report if the task was successful
        if async_result.successful():
            print('Task was successful.')
```

Running the example executes the task asynchronously.

The caller then checks to see if the task is completed or not, and if not to block for a second and check again. This is called a busy-wait loop and is a helpful pattern in concurrency programming. It allows the caller to do other things and give up waiting if it chooses.

The task loops, generating random numbers and reporting progress along the way.

Once the asynchronous task completes, the caller exits its busy wait loop and then reports a message if the task completed successfully.

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

```
Main process waiting...
>0 got 0.5898448969350131
>1 got 0.22306653764618534
Main process waiting...
>2 got 0.803522628622837
Main process waiting...
>3 got 0.6576819630495541
>4 got 0.5732127322896557
```

```
Main process waiting...
>5 got 0.5960402721165015
Main process waiting...
>6 got 0.9680748920499497
>7 got 0.057583828378944046
Main process waiting...
>8 got 0.785156301006927
>9 got 0.2663906981882074
Task was successful.
```

Lesson Review

Takeaways

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

- You discovered how to use callback and error callback functions with asynchronous tasks.
- You discovered how to wait for and get results from asynchronous tasks
- You discovered how to check the status of asynchronous tasks.

Exercise

Your task for this lesson is to update the above example to cause exceptions and to handle those exceptions.

In the loop of the `task()` function, add an if-statement to cause the task to raise an exception if a number above a threshold is generated. The specific exception raised and the specific threshold do not matter.

For example:

```
# check if the generated value is above a threshold
if value > 0.7:
    raise Exception('Something bad happened')
```

Add an error callback function to the asynchronous task to report the exception.

Update the example to report a message if the task completed unsuccessfully.

For bonus points, update the task function to return a value, have the caller get and report the result of the task and correctly handle the case if an exception was raised.

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
- `random` - Generate pseudo-random numbers.
  https://docs.python.org/3/library/random.html
- `time` - Time access and conversions.
  https://docs.python.org/3/library/time.html

Next

In the next lesson, we will discover how to build upon our understanding of the multiprocessing pool with an end-to-end worked example.

Lesson 07: Case Study: Parallel Primality Testing

In this lesson, you will discover how to develop an end-to-end example of using the multiprocessing pool to execute a computationally intensive task concurrently.

Specifically, this lesson will cover:

- How to develop a function to check if a number is prime.
- How to check each number in a list is prime, one-by-one.
- How to speedup primality testing of a list of numbers using the multiprocessing pool.

Let's get started.

How to Perform Primality Testing in Python

A number is prime if it is positive and only divisible by itself and one.

The Wikipedia page for primality testing (linked in the further reading section at the end of the lesson) lays out a naive algorithm that we can use to test if a number is prime.

The steps can be summarized as follows:

- If the number is less than 2, it is not prime.
- If the number is 2, prime.
- If the number can be divided by 2 with no remainder, not prime.
- If the number has a divisor between 3 and `sqrt(n)`, not prime
- Otherwise, prime.

Let's take a closer look at these checks.

Use Modulo To Check for a Divisor

Firstly, we can check if a target number is divisible by another number with no remainder by using the modulo operator.

Recall that this operator returns the remainder of a division.

For example, we can check if a number is divisible with no remainder as follows:

```
# check if divisible with no remainder
def is_divisible(target, number):
    return
```

Check Divisibility for a Series Of Numbers

To test if a number is prime, we can see if it is divisible by all numbers between 2 and itself minus one with no remainder.

For example:

```
...
# check if a target is divisible
for i in range(2, target):
    if target % i == 0:
        # not prime
# otherwise prime
```

In fact, we can just check if the number is divisible by 2 with no remainder and skip all positive numbers because they too are all divisible by 2.

```
...
# check if divisible by 2, rule out all even divisors
if target % 2 == 0:
    # not prime
```

This leaves only the odd numbers between 3 and the target number minus one.

Recall, the `range()` function will take a step size as a third parameter, therefore we can write something like:

```
...
# check if a target is divisible
for i in range(3, target, 2):
    if target % i == 0:
        # not prime
# otherwise prime
```

When looking at divisors for a number, we see symmetry between `[1 to n/2]` and `[n/2 to n]`, where `n` is the number we are considering.

We can push this further (according to Wikipedia), and only consider divisors between 3 and the square root of the target, e.g. `math.sqrt(target)`.

We might be able to write something like:

```
...
# check if a target is divisible
for i in range(3, sqrt(target), 2):
    if target % i == 0:
```

```
        # not prime
# otherwise prime
```

So far so good.

Correct Upper-Limit

A minor issue is that the square root may not result in an integer, therefore we need to round it in some way.

We can take the `math.floor()` of the number which will just keep the integer and drop any fractional values.

```
...
# calculate the upper limit on divisors to check
limit = floor(sqrt(target))
```

Finally, we want to check the integer that is the square root of the target, not stop at one minus this value, which is the default behavior of `range()`.

Therefore, we can add one to this upper limit of divisors to check.

```
...
# limit divisors to sqrt(n)+1 so range will reach it
limit = floor(sqrt(number)) + 1
```

And that's about it.

Complete `is_prime()` Function

Tying all of this together, the `is_prime()` function below will take a number and return `True` if it is prime or `False` otherwise.

It's not the fastest algorithm for primality testing, but it will work.

```
# returns True if prime, False otherwise
def is_prime(number):
    # 1 is a special case of not prime
    if number <= 1:
        return False
    # 2 is a special case of a prime
    if number == 2:
        return True
    # check if the number divides by 2 with no remainder
    if number % 2 == 0:
        return False
    # limit divisors to sqrt(n)+1 so range will reach it
    limit = floor(sqrt(number)) + 1
    # check all odd numbers in range
```

```
    for i in range(3, limit, 2):
        # check if number is divisible with no remainder
        if number % i == 0:
            # number is divisible and is not a prime
            return False
    # number is probably prime
    return True
```

Next, let's explore how we might use this function to test a series of numbers to see if they are prime.

How to Check if Numbers are Prime Sequentially (slow)

We can now test out our function on some candidate primes.

Check Many Numbers

Let's define a function `check_numbers_are_prime()` that takes a list of numbers and tests each in turn, reporting if they are prime, and ignoring them if not.

This is a typical use case, where we are interested in what numbers are prime, not whether a given number is prime or not.

The function is listed below.

```
# check if a series of numbers are prime or not
def check_numbers_are_prime(numbers):
    # check each number in turn
    for number in numbers:
        if is_prime(number):
            print(f'{number} is prime')
```

Next, let's test it out on some known primes.

Test with Known Large Primes

Let's consider testing a dozen examples of such large numbers.

We can take these numbers from the Wikipedia page "*List of prime numbers*" (linked at the end of the lesson) that lists some very large prime numbers. We can also spice up the list with variations of known primes that are not primes, e.g. remove, add, or change a digit.

Below is a list of 17 candidate primes we can test, 14 of which are actually prime.

```
...
# define some numbers to check
NUMS = [17977, 10619863, 106198, 6620830889,
    80630964769, 228204732751, 1171432692373,
```

```
    1398341745571, 10963707205259, 15285151248481,
    99999199999, 304250263527209, 30425026352720,
    10657331232548839, 10657331232548830,
    44560482149, 1746860020068409]
```

Complete Example

The complete example with this updated list of numbers to check for primality is listed below.

```
# SuperFastPython.com
# check if large numbers are prime
from math import sqrt
from math import floor

# returns True if prime, False otherwise
def is_prime(number):
    # 1 is a special case of not prime
    if number <= 1:
        return False
    # 2 is a special case of a prime
    if number == 2:
        return True
    # check if the number divides by 2 with no remainder
    if number % 2 == 0:
        return False
    # limit divisors to sqrt(n)+1 so range will reach it
    limit = floor(sqrt(number)) + 1
    # check all odd numbers in range
    for i in range(3, limit, 2):
        # check if number is divisible with no remainder
        if number % i == 0:
            # number is divisible and is not a prime
            return False
    # number is probably prime
    return True

# check if a series of numbers are prime or not
def check_numbers_are_prime(numbers):
    # check each number in turn
    for number in numbers:
        if is_prime(number):
            print(f'{number} is prime')

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

```
    # define some numbers to check
    NUMS = [17977, 10619863, 106198, 6620830889,
        80630964769, 228204732751, 1171432692373,
        1398341745571, 10963707205259, 15285151248481,
        99999199999, 304250263527209, 30425026352720,
        10657331232548839, 10657331232548830,
        44560482149, 1746860020068409]
    # check whether each number is a prime
    check_numbers_are_prime(NUMS)
```

Running the example tests if each number is prime.

We can see that the 14 primes are identified correctly.

Importantly, the process is relatively slow, taking about 7.1 seconds on modern hardware.

This will get progressively worse as we add more large numbers or numbers even larger.

```
17977 is prime
10619863 is prime
6620830889 is prime
80630964769 is prime
228204732751 is prime
1171432692373 is prime
1398341745571 is prime
10963707205259 is prime
15285151248481 is prime
99999199999 is prime
304250263527209 is prime
10657331232548839 is prime
44560482149 is prime
1746860020068409 is prime
```

How to Check if Numbers are Prime in Parallel (fast)

We can update our primality checking program to use the multiprocessing pool directly with very little change.

Making the prime testing example concurrent is a natural case for the `map()` method.

This is because:

- We want to apply the `is_prime()` function to each number in a list.
- We want to process results in order that tasks were issued.

Traverse Results With Numbers Using `zip()`

We also want to report the number along with whether it is prime.

This can be achieved using the built-in `zip()` function that allows two iterables to be traversed in tandem.

We could adapt the example to use the multiprocessing pool with the `map()` method by first getting the iterable of return values from map, then iterating the zip of the return values and the numbers.

For example:

```
...
# create multiprocessing pool
with Pool() as pool:
    # issue the tasks
    results = pool.map(is_prime, numbers)
    # report results as completed in order
    for number, isprime in zip(numbers, results):
        if isprime:
            print(f'{number} is prime')
```

Report Results As Tasks Complete with `imap()`

A limitation of this approach is that the `map()` method does not return until all issued tasks have completed.

Another limitation is that the iterable of all numbers are issued immediately to the multiprocessing pool. This could cause memory problems if the source of numbers to check contained millions or billions of items.

An alternate approach is to use the `imap()` method.

This method can be used in the same way as `map()` method, except that it will yield tasks to the pool as workers become available and will yield return values as tasks are completed.

This will use less memory and make the program more responsive, showing results along the way.

For example:

```
...
# issue the tasks
results = pool.imap(is_prime, numbers)
# report results as completed in order
for number, isprime in zip(numbers, results):
    if isprime:
        print(f'{number} is prime')
```

Complete Example

Tying this together, the updated version of checking numbers for primality concurrently using the multiprocessing pool is listed below.

```
# SuperFastPython.com
# check if large numbers are prime concurrently
from math import sqrt
from math import floor
from multiprocessing import Pool

# returns True if prime, False otherwise
def is_prime(number):
    # 1 is a special case of not prime
    if number <= 1:
        return False
    # 2 is a special case of a prime
    if number == 2:
        return True
    # check if the number divides by 2 with no remainder
    if number % 2 == 0:
        return False
    # limit divisors to sqrt(n)+1 so range will reach it
    limit = floor(sqrt(number)) + 1
    # check all odd numbers in range
    for i in range(3, limit, 2):
        # check if number is divisible with no remainder
        if number % i == 0:
            # number is divisible and is not a prime
            return False
    # number is probably prime
    return True

# check if a series of numbers are prime or not
def check_numbers_are_prime(numbers):
    # create multiprocessing pool
    with Pool() as pool:
        # issue the tasks
        results = pool.imap(is_prime, numbers)
        # report results as completed in order
        for number, isprime in zip(numbers, results):
            if isprime:
                print(f'{number} is prime')

# protect the entry point
```

```
if __name__ == '__main__':
    # define some numbers to check
    NUMS = [17977, 10619863, 106198, 6620830889,
        80630964769, 228204732751, 1171432692373,
        1398341745571, 10963707205259, 15285151248481,
        99999199999, 304250263527209, 30425026352720,
        10657331232548839, 10657331232548830,
        44560482149, 1746860020068409]
    # check whether each number is a prime
    check_numbers_are_prime(NUMS)
```

Running the example tests the list of numbers for primality as we did before.

We see the same results, which is as expected.

In this case, the program finishes in about half the time or about 4.4 seconds on my system. That is about a 1.61x speedup over the serial case.

```
17977 is prime
10619863 is prime
6620830889 is prime
80630964769 is prime
228204732751 is prime
1171432692373 is prime
1398341745571 is prime
10963707205259 is prime
15285151248481 is prime
99999199999 is prime
304250263527209 is prime
10657331232548839 is prime
44560482149 is prime
1746860020068409 is prime
```

Lesson Review

Takeaways

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

- You discovered how to check if a number is prime or not prime in Python.
- You discovered how to check a list of numbers for primality, sequentially one-by-one.
- You discovered how to update the primality testing program to use the multiprocessing pool and execute tasks in parallel.

Exercise

Your task for this lesson is to extend the above example for primality testing.

1. Update the example to test all numbers between 1 and 100 and confirm the results are correct using the "*List of prime numbers*" Wikipedia page.
2. Update the example to test all numbers between 1 and 1,000 and use `chunksize` to send tasks in batch to child worker processes. Tune the chunksize to get the performance.
3. Update the example to use `imap_unordered()` to make the program more responsive.

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.

- Prime number, Wikipedia.
  https://en.wikipedia.org/wiki/Prime_number
- Primality test, Wikipedia.
  https://en.wikipedia.org/wiki/Primality_test
- List of prime numbers, Wikipedia.
  https://en.wikipedia.org/wiki/List_of_prime_numbers
- Python Built-in Functions.
  https://docs.python.org/3/library/functions.html
- `multiprocessing` API - Process-based parallelism.
  https://docs.python.org/3/library/multiprocessing.html
- `math` - Mathematical functions.
  https://docs.python.org/3/library/math.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 process-based concurrency, the need for process pools and how to create a multiprocessing pool to execute CPU-bound tasks in parallel.
- You discovered how to configure the multiprocessing pool including how to set the number of child worker processes, the number of tasks per worker before they are replaced, and how to initialize worker processes.
- You discovered how to execute ad hoc tasks in the multiprocessing pool and block until the results are available.
- You discovered how to execute tasks asynchronously in the multiprocessing pool, allowing the caller to continue on with other work.
- You discovered how to use lazy functions to execute many tasks that use less memory and provide task return values in a way that is more responsive.
- You discovered how to work with asynchronous tasks, including how to execute callback functions, how to wait for and get results, and how to check on their status.
- You discovered how to build upon what you have learned to speed-up the CPU-bound task of testing if numbers are prime to execute in parallel using the multiprocessing pool.

You now know how to use the multiprocessing pool 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

Save days of debugging with step-by-step jump-start guides.

Python Threading Jump-Start.
https://SuperFastPython.com/ptj

Python ThreadPool Jump-Start.
https://SuperFastPython.com/ptpj

Python ThreadPoolExecutor Jump-Start.
https://SuperFastPython.com/ptpej

Python Multiprocessing Jump-Start.
https://SuperFastPython.com/pmj

Python Multiprocessing Pool Jump-Start.
https://SuperFastPython.com/pmpj

Python ProcessPoolExecutor Jump-Start.
https://SuperFastPython.com/pppej

Python Asyncio Jump-Start.
https://SuperFastPython.com/paj