*SuperFast*Python
Guide

Parallel Loops in Python

Run Python Loops on All CPU Cores

Jason Brownlee

Parallel Loops in Python

Run Python Loops on All CPU Cores

Jason Brownlee

2022

Praise for *SuperFast*Python

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

– **Gabriele Berselli**, Python Developer.

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

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

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

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

"*Great as always!!!*"

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

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

– **Marius Rusu**, Python Developer.

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

– **Bilal B.**, Senior Data Engineer.

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

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

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

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

"*This is awesome*"

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

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.

Contents

Introduction

Welcome to **Parallel Loops in Python**!

This book will show you how to run your Python loops in parallel, *super fast*.

There are two main approaches to executing loops in parallel in Python, they are:

- **Thread-based concurrency** using the `threading` module.
- **Process-based concurrency** using the `multiprocessing` module.

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

Thread-based Concurrency

Thread-based concurrency is suited to I/O-bound tasks, such as reading and writing files, sockets, and interacting with devices like cameras.

Thread-based concurrency is not appropriate for CPU-bound tasks, such as calculating or modeling. This is because of the Global Interpreter Lock that prevents more than one thread from running at a time while the lock is held. The lock is not held in some cases, such as while performing I/O.

We can develop loops that execute in parallel with thread-based concurrency using one of three classes:

1. The `Thread` class.
2. The `ThreadPool` class.
3. The `ThreadPoolExecutor` class.

Process-based Concurrency

Process-based concurrency is ideally suited to CPU-bound tasks like calculating, parsing, encoding, and modeling.

We can also use process-based concurrency for I/O-bound tasks, but it is not well suited. The reason is that all data shared between processes must be pickled (serialized), which can be slow. Also, we may be limited in the maximum number of processes that can be created.

We can develop loops that execute in parallel with process-based concurrency using one of three classes:

1. The `Process` class.
2. The `Pool` class.
3. The `ProcessPoolExecutor` class.

Choosing Between Threads and Processes

If you would like to know more about how to choose between thread-based concurrency and process-based concurrency, I recommend the guide:

- How to Choose the Right Python Concurrency API
  https://SuperFastPython.com/python-concurrency-choose-api/

Source Code

This book provides a code example for each way to run loops in parallel.

You can use these code examples as templates in your own projects by replacing the body of the target function with your own code.

You can copy-paste the code examples from this ebook directly.

The Python `.py` files were provided with this book in a `src/` subdirectory.

You may also download the latest version of the code files from the public GitHub repository here:

- Parallel Loops In Python on GitHub
  https://github.com/SuperFastPython/ParallelLoopsInPython

Next, let's look at the first example of a parallel for-loop in Python.

Parallel Loop with the `Thread` Class

We can create a new thread for each iteration of the loop.

This can be achieved by creating a `Thread` object and setting the `target` argument to the name of the function to execute and pass any arguments via the `args` argument.

We can create all the required threads first using a list comprehension. Once created, all of the threads can be started at once by calling the `start()` method on each. Finally, we can wait for all the threads to finish by joining each in turn with the `join()` method.

The example below demonstrates a parallel for-loop using the `Thread` class.

```
# SuperFastPython.com
# example of a parallel for loop with the Thread class
from threading import Thread

# execute a task
def task(value):
    # add your work here...
    # ...
    # all done
    print(f'.done {value}')

# protect the entry point
if __name__ == '__main__':
    # create all tasks
    threads = [Thread(target=task, args=(i,)) for i in range(20)]
    # start all threads
    for thread in threads:
        thread.start()
    # wait for all threads to complete
    for thread in threads:
        thread.join()
    # report that all tasks are completed
    print('Done')
```

This approach is effective for a small number of tasks that all need to be run at once.

It is less effective if we have many more tasks than we can support concurrently because all of the tasks will run at the same time and could slow each other down.

It also does not allow results from tasks to be returned easily.

You can learn more about the `Thread` class in my free guide:

- Threading in Python: The Complete Guide
  https://SuperFastPython.com/threading-in-python/

Parallel Loop with the ThreadPool Class

We can create a pool of worker threads that can be reused for many tasks.

This can be achieved using the `ThreadPool` class that will create one worker for each logical CPU core in the system.

The `ThreadPool` class can be created using the context manager interface, which ensures that it is closed and all workers are released once we are finished with it.

We can call the same function many times with different arguments using the `map()` method on the `ThreadPool` class. Each call to the target function will be issued as a separate task.

The example below demonstrates a parallel for-loop with the `ThreadPool` class.

```
# SuperFastPython.com
# example of a parallel for loop with the ThreadPool class
from multiprocessing.pool import ThreadPool

# execute a task
def task(value):
    # add your work here...
    # ...
    # return a result, if needed
    return value

# protect the entry point
if __name__ == '__main__':
    # create the pool with the default number of workers
    with ThreadPool() as pool:
        # issue one task for each call to the function
        for result in pool.map(task, range(100)):
            # handle the result
            print(f'>got {result}')
    # report that all tasks are completed
    print('Done')
```

This approach is very effective for executing tasks that involve calling the same function many times with different arguments.

The `ThreadPool` class provides many variations of the `map()` function, such as lazy versions and a version that allows multiple arguments to the task function.

You can learn more about the `ThreadPool` class in my free tutorial:

- ThreadPool in Python: The Complete Guide
  https://SuperFastPython.com/threadpool-python/

Parallel Loop with the `ThreadPoolExecutor` Class

We can create a pool of worker threads using the `ThreadPoolExecutor` class with a modern executor interface.

This allows tasks to be issued as one-off tasks via the `submit()` method, returning `Future` object that provides a handle on the task. It also allows the same function to be called many times with different arguments via the `map()` method.

The example below demonstrates parallel for-loops with the `ThreadPoolExecutor` class.

```
# SuperFastPython.com
# example of a parallel for loop with the ThreadPoolExecutor class
import concurrent.futures

# execute a task
def task(value):
    # add your work here...
    # return a result, if needed
    return value

# protect the entry point
if __name__ == '__main__':
    # create the pool with the default number of workers
    with concurrent.futures.ThreadPoolExecutor() as exe:
        # issue some tasks and collect futures
        futures = [exe.submit(task, i) for i in range(50)]
        # handle results as tasks are completed
        for future in concurrent.futures.as_completed(futures):
            print(f'>got {future.result}')
        # issue one task for each call to the function
        for result in exe.map(task, range(50)):
            print(f'>got {result}')
    # report that all tasks are completed
    print('Done')
```

This is the preferred approach for modern parallel for-loops.

This approach is effective for issuing one-off tasks as well as calling the same function many times with different arguments.

The `ThreadPoolExecutor` provides a modern approach for executing parallel for-loops in Python.

You can learn more about the `ThreadPoolExecutor` class in my free guide:

- ThreadPoolExecutor in Python: The Complete Guide
  https://SuperFastPython.com/threadpoolexecutor-in-python/

Parallel Loop with the `Process` Class

We can create a new child process for each iteration of the loop.

This can be achieved by creating a `Process` object and setting the `target` argument to the name of the function to execute and pass any arguments via the `args` argument.

We can create all the required processes first using a list comprehension. Once created, all of the processes can be started at once by calling the `start()` method on each. Finally, we can wait for all the processes to finish by joining each in turn with the `join()` method.

The example below demonstrates a parallel for-loop using the `Process` class.

```
# SuperFastPython.com
# example of a parallel for loop with the Process class
from multiprocessing import Process

# execute a task
def task(value):
    # add your work here...
    # ...
    # all done
    print(f'.done {value}', flush=True)

# protect the entry point
if __name__ == '__main__':
    # create all tasks
    processes = [Process(target=task, args=(i,)) for i in range(20)]
    # start all processes
    for process in processes:
        process.start()
    # wait for all processes to complete
    for process in processes:
        process.join()
    # report that all tasks are completed
    print('Done')
```

This approach is effective for a small number of tasks that all need to be run at once.

It is less effective if we have many more tasks than we have CPU cores because all of the tasks will run at the same time and slow each other down.

It also does not allow results from tasks to be returned easily.

You can learn more about the `Process` class in my free guide:

- Multiprocessing in Python: The Complete Guide
  https://SuperFastPython.com/multiprocessing-in-python/

Parallel Loop with the `Pool` Class

We can create a pool of worker processes that can be reused for many tasks.

This can be achieved using the `Pool` class that will create one worker for each logical CPU core in the system.

The `Pool` class can be created using the context manager interface, which ensures that it is closed and all workers are released once we are finished with it.

We can call the same function many times with different arguments using the `map()` method on the `Pool` class. Each call to the target function will be issued as a separate task.

The example below demonstrates a parallel for-loop with the `Pool` class.

```
# SuperFastPython.com
# example of a parallel for loop with the Pool class
from multiprocessing import Pool

# execute a task
def task(value):
    # add your work here...
    # ...
    # return a result, if needed
    return value

# protect the entry point
if __name__ == '__main__':
    # create the pool with the default number of workers
    with Pool() as pool:
        # issue one task for each call to the function
        for result in pool.map(task, range(100)):
            # handle the result
            print(f'>got {result}')
    # report that all tasks are completed
    print('Done')
```

This approach is very effective for executing tasks that involve calling the same function many times with different arguments.

The `Pool` class provides many variations of the `map()` function, such as lazy versions and a version that allows multiple arguments to the task function.

You can learn more about the `Pool` class in my free guide:

- Multiprocessing Pool in Python: The Complete Guide
  https://SuperFastPython.com/multiprocessing-pool-python/

Parallel Loop with the `ProcessPoolExecutor` Class

We can create a pool of worker processes using the `ProcessPoolExecutor` class with a modern executor interface.

This allows tasks to be issued as one-off tasks via the `submit()` method, returning `Future` object that provides a handle on the task. It also allows the same function to be called many times with different arguments via the `map()` method.

The example below demonstrates parallel for-loops with the `ProcessPoolExecutor` class.

```
# SuperFastPython.com
# example of a parallel for loop with the ProcessPoolExecutor class
import concurrent.futures

# execute a task
def task(value):
    # add your work here...
    # return a result, if needed
    return value

# protect the entry point
if __name__ == '__main__':
    # create the pool with the default number of workers
    with concurrent.futures.ProcessPoolExecutor() as exe:
        # issue some tasks and collect futures
        futures = [exe.submit(task, i) for i in range(50)]
        # process results as tasks are completed
        for future in concurrent.futures.as_completed(futures):
            print(f'>got {future.result}')
        # issue one task for each call to the function
        for result in exe.map(task, range(50)):
            print(f'>got {result}')
    # report that all tasks are completed
    print('Done')
```

This is the preferred approach for modern parallel for-loops.

This approach is effective for issuing one-off tasks as well as calling the same function many times with different arguments.

The `ProcessPoolExecutor` provides a modern approach for executing parallel for-loops in Python.

You can learn more about the `ProcessPoolExecutor` class in my free guide:

- ProcessPoolExecutor in Python: The Complete Guide
  https://SuperFastPython.com/processpoolexecutor-in-python/

Thank-You

Thank you for letting me help you develop parallel loops in Python.

If you ever have any questions about this guide or Python concurrency in general, please reach out.

You can contact me directly here:

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

Kind Regards,

Jason Brownlee
SuperFastPython.com
Making python developers awesome at concurrency