How to Wait For All Tasks to Finish in the ThreadPoolExecutor

NOVEMBER 30, 2021 *by* **JASON BROWNLEE** *in* **THREADPOOLEXECUTOR (HTTPS://SUPERFASTPYTHON.COM/CATEGORY/THREADPOOLEXECUTOR/)**

You can wait for a task to finish in a **ThreadPoolExecutor (https://superfastpython.com/threadpoolexecutor-in-python/)** by calling the **wait()** module function.

In this tutorial you will discover how to wait for tasks to finish in a Python thread pool.

Let's get started.

Table of Contents

Need To Wait for All Tasks to Finish

The `ThreadPoolExecutor` in Python provides a pool of reusable threads for executing ad hoc tasks.

You can submit tasks to the thread pool by calling the `submit()` function and passing in the name of the function you wish to execute on another thread. Calling the `submit()` function will return a `Future` object that allows you to check on the status of the task and get the result from the task once it completes.

You can also submit tasks to the thread pool by calling the `map()` function and pass in the name of the function and the iterable of items that your function will be applied to asynchronously.

After submitting tasks to the thread pool, you may want to wait for the tasks to complete before continuing.

There are many reasons for this to be the case, for example:

- Perhaps you don't have the computational resources to perform the tasks and continue on with the program.
- Perhaps you need the results from all tasks.
- Perhaps you need the action performed by each task to have been performed before continuing.

How can you wait for tasks submitted to the `ThreadPoolExecutor` to complete?

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

How to Wait For All Tasks To Finish

There are a number of ways that you can wait for all tasks to complete in the `ThreadPoolExecutor`.

We will look at three approaches you can use, they are calling the `wait()` module function, getting the result from each task, and shutting down the thread pool.

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

Approach 1: Call the wait() Module Function

The most common approach is to call the `wait()` module function (https://docs.python.org/3/library/concurrent.futures.html) and pass in a collection of `Future` objects created when calling `submit()`.

The wait function allows you fine grained control over the specific tasks that you wish to wait to complete and also allows you to specify a timeout for how long you are willing to wait in seconds.

```
...
# wait for a collection of tasks to complete
wait(futures)
```

Approach 2: Get The Results From Each Task

Alternatively, you can enumerate the list of `Future` objects and attempt to get the result from each.

This iteration will complete when all results are available meaning that all tasks were completed.

```
...
# wait for all tasks to complete by getting all results
for future in futures:
    result = future.result()
    # do something with the result...
# all tasks are complete
```

A similar approach could be used with the `as_completed()` module function that will iterate over `Future` objects in the order that tasks are completed, rather than the order they were stored in the list.

For example:

```python
...
# wait for all tasks to complete by getting all results
for future in as_completed(futures):
    result = future.result()
    # do something with the result...
# all tasks are complete
```

Approach 3: Call shutdown()

Perhaps you don't have `Future` objects for your tasks because you submitted them to the thread pool by calling `map()`, or perhaps you wish to wait for all tasks in the thread pool to complete rather than a subset.

In this case, you can wait for tasks to complete while shutting down the thread pool.

You can shutdown the thread pool by calling the `shutdown()` function. We can set the `wait` argument to `True` so that the call will not return until all running tasks complete and set `cancel_futures` to `True` which will cancel all scheduled tasks.

```python
...
# shutdown the pool, cancels scheduled tasks, returns when running tasks complete
executor.shutdown(wait=True, cancel_futures=True)
```

You can also shutdown the pool and not cancel the scheduled tasks, yet still wait for all tasks to complete.

This will ensure all running and scheduled tasks are completed before the function returns. This is the default behavior of the shutdown function, but is a good idea to specify explicitly.

```python
...
# shutdown the pool, returns after all scheduled and running tasks complete
executor.shutdown(wait=True, cancel_futures=False)
```

Now that we have seen a few ways to wait for tasks to complete in the `ThreadPoolExecutor`, let's look at a worked example.

Confused by the ThreadPoolExecutor class API?

Download my FREE PDF cheat sheet (https://marvelous-writer-6152.ck.page/5fb5f69c42)

Example of Waiting For All Tasks To Complete With wait()

Let’s explore how to wait for tasks to complete in the `ThreadPoolExecutor` with an example.

First, let’s define a simple task that will sleep for a fraction of a second and report when it is completed.

```
# custom task that will sleep for a variable amount of time
def task(name):
    # sleep for less than a second
    sleep(random())
    print(f'Done: {name}')
```

Next, we can create a thread pool with 2 worker threads and issue ten tasks to the pool for execution by calling the `submit()` function for each task.

Each call to submit will return a Future object which we will collect into a list.

```
...
# start the thread pool
with ThreadPoolExecutor(2) as executor:
    # submit tasks and collect futures
    futures = [executor.submit(task, i) for i in range(10)]
```

Next, we can call the `wait()` module function and pass in the list of the ten Future objects we collected when calling `submit()`.

```
...
# wait for all tasks to complete
print('Waiting for tasks to complete...')
wait(futures)
```

This call will return once all tasks associated with the Future objects in the collection have completed.

We can then report that all tasks are completed.

```
...
print('All tasks are done!')
```

Tying this together, the complete example of starting tasks and waiting for them all to complete before continuing on is listed below.

```python
# SuperFastPython.com
# example of waiting for tasks to complete
from time import sleep
from random import random
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# custom task that will sleep for a variable amount of time
def task(name):
    # sleep for less than a second
    sleep(random())
    print(f'Done: {name}')

# start the thread pool
with ThreadPoolExecutor(2) as executor:
    # submit tasks and collect futures
    futures = [executor.submit(task, i) for i in range(10)]
    # wait for all tasks to complete
    print('Waiting for tasks to complete...')
    wait(futures)
    print('All tasks are done!')
```

Running the example first submits all tasks into the thread pool.

We then start waiting for the tasks to complete, reporting a message.

The tasks complete, reporting a message as they do.

Finally, all tasks are completed and we are free to carry on.

```
Waiting for tasks to complete...
Done: 0
Done: 1
Done: 2
Done: 3
Done: 4
Done: 6
Done: 7
Done: 5
Done: 8
Done: 9
All tasks are done!
```

Example of Waiting For All Tasks To Complete With shutdown()

Let’s look at an alternative approach for waiting for all tasks to complete.

Perhaps we have many tasks in the thread pool and we don’t have `Future` objects for them.

This might happen if we add tasks to the pool using the `map()` function and choose not to enumerate the results for the tasks.

```python
...
# submit tasks to the thread pool
executor.map(task, range(10))
```

We can wait for all of the scheduled and running tasks to complete by calling the `shutdown()` function explicitly and setting `wait=True` and `cancel_futures=False`, the default arguments.

```python
...
# shutdown the thread pool and wait for all tasks to complete
executor.shutdown()
```

Because we are using the context manager, the thread pool will be closed automatically using the default parameters.

Therefore, we don’t need to add any extra code in order to wait for all tasks in the thread pool to complete before continuing on.

We can demonstrate this with a worked example listed below.

```python
# SuperFastPython.com
# example of waiting for tasks to complete via a pool shutdown
from time import sleep
from random import random
from concurrent.futures import ThreadPoolExecutor

# custom task that will sleep for a variable amount of time
def task(name):
    # sleep for less than a second
    sleep(random())
    print(f'Done: {name}')

# start the thread pool
with ThreadPoolExecutor(2) as executor:
    # submit tasks
    executor.map(task, range(10))
    # wait for all tasks to complete
    print('Waiting for tasks to complete...')
print('All tasks are done!')
```

Running the example, we see that all tasks are submitted to the pool using the `map()` function and we do not have `Future` objects associated with each task.

We then wait for the tasks to complete at the end of the context manager. Tasks report their progress as they complete.

Finally, all tasks are completed, the context manager closes the thread pool, and the program ends.

```
Waiting for tasks to complete...
Done: 0
Done: 1
Done: 2
Done: 4
Done: 3
Done: 6
Done: 7
Done: 5
Done: 8
Done: 9
All tasks are done!
```

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

Further Reading

This section provides additional resources that you may find helpful.

Books

- ThreadPoolExecutor Jump-Start (https://amzn.to/3Evr8Sa), Jason Brownlee, 2022 (**my book!**).
- Concurrent Futures API Interview Questions (https://amzn.to/3E4TTnh)
- ThreadPoolExecutor Class API Cheat Sheet (https://superfastpython.gumroad.com/l/nyfpaz)

I also recommend specific chapters from the following books:

- Effective Python (https://amzn.to/3GpopJ1), Brett Slatkin, 2019.
    - See *Chapter 7: Concurrency and Parallelism*

- Python in a Nutshell (https://amzn.to/3m7SLGD), Alex Martelli, et al., 2017.
  - See: *Chapter: 14: Threads and Processes*

Guides

- ThreadPoolExecutor: The Complete Guide (https://superfastpython.com/threadpoolexecutor-in-python/)

APIs

- concurrent.futures - Launching parallel tasks (https://docs.python.org/3/library/concurrent.futures.html)