SuperFastPython
making python developers awesome at concurrency (https://superfastpython.com/)

ThreadPoolExecutor in Python: The Complete Guide

JANUARY 20, 2022 *by* JASON BROWNLEE *in* THREADPOOLEXECUTOR (HTTPS://SUPERFASTPYTHON.COM/CATEGORY/THREADPOOLEXECUTOR/)

The **Python ThreadPoolExecutor** allows you to create and manage thread pools in Python.

Although the **ThreadPoolExecutor** has been available since Python 3.2, it is not widely used, perhaps because of misunderstandings of the capabilities and limitations of Threads in Python.

This guide provides a detailed and comprehensive review of the **ThreadPoolExecutor** in Python, including how it works, how to use it, common questions, and best practices.

This is a massive 23,000+ word guide. You may want to bookmark it so you can refer to it as you develop your concurrent programs.

Let's dive in.

Skip the tutorial. Master the ThreadPoolExecutor today. Learn more (https://superfastpython.com/ptpej-incontent)

Table of Contents

Python Threads and the Need for Thread Pools

So, what are threads and why do we care about thread pools?

What Are Python Threads

A thread (https://en.wikipedia.org/wiki/Thread_(computing)) refers to a thread of execution by a computer program.

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

Sometimes, we may need to create additional threads within our Python process to execute tasks concurrently.

Python provides real naive (system-level) threads via the **threading.Thread** class (https://docs.python.org/3/library/threading.html).

A task can be run in a new thread by creating an instance of the **Thread** class and specifying the function to run in the new thread via the target argument.

```
...
# create and configure a new thread to run a function
thread = Thread(target=task)
```

Once the thread is created, it must be started by calling the **start()** function.

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

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

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

We can demonstrate this with a complete example with a task that sleeps for a moment and prints a message.

The complete example of executing a target task function in a separate thread is listed below.

```
# SuperFastPython.com
# example of executing a target task function in a separate thread
from time import sleep
from threading import Thread

# a simple task that blocks for a moment and prints a message
def task():
    # block for a moment
    sleep(1)
    # display a message
    print('This is coming from another thread')

# create and configure a new thread to run a function
thread = Thread(target=task)
# start the task in a new thread
thread.start()
# display a message
print('Waiting for the new thread to finish...')
# wait for the task to complete
thread.join()
```

Running the example creates the thread object to run the **task()** function.

The thread is started and the **task()** function is executed in another thread. The task sleeps for a moment; meanwhile, in the main thread, a message is printed that we are waiting around and the main thread joins the new thread.

Finally, the new thread finishes sleeping, prints a message, and closes. The main thread then carries on and also closes as there are no more instructions to execute.

```
Waiting for the new thread to finish...
This is coming from another thread
```

You can learn more about Python threads in the guide:

- Threading in Python: The Complete Guide (https://superfastpython.com/threading-in-python/)

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

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

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

This can be achieved using a thread pool.

What Are Thread Pools

A thread pool (https://en.wikipedia.org/wiki/Thread_pool) is a programming pattern for automatically managing a pool of worker threads.

The pool is responsible for a fixed number of threads.

- It controls when the threads are created, such as just-in-time when they are needed.
- It also controls what threads should do when they are not being used, such as making them wait without consuming computational resources.

Each thread in the pool is called a worker or a worker thread. Each worker is agnostic to the type of tasks that are executed, along with the user of the thread pool to execute a suite of similar (homogeneous) or dissimilar tasks (heterogeneous) in terms of the function called, function arguments, task duration, and more.

Worker threads are designed to be re-used once the task is completed and provide protection against the unexpected failure of the task, such as raising an exception, without impacting the worker thread itself.

This is unlike a single thread that is configured for the single execution of one specific task.

The pool may provide some facility to configure the worker threads, such as running an initialization function and naming each worker thread using a specific naming convention.

Thread pools can provide a generic interface for executing ad hoc tasks with a variable number of arguments, but do not require that we choose a thread to run the task, start the thread, or wait for the task to complete.

It can be significantly more efficient to use a thread pool instead of manually starting, managing, and closing threads, especially with a large number of tasks.

Python provides a thread pool via the **ThreadPoolExecutor** class.

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

ThreadPoolExecutor for Thread Pools in Python

The **ThreadPoolExecutor** Python class is used to create and manage thread pools and is provided in the concurrent.futures module (https://docs.python.org/3/library/concurrent.futures.html).

The concurrent.futures module was introduced in Python 3.2 written by Brian Quinlan (http://sweetapp.com/) and provides both thread pools and process pools, although we will focus our attention on thread pools in this guide.

If you're interested, you can access the Python source code for the **ThreadPoolExecutor** class directly via thread.py (https://github.com/python/cpython/tree/3.10/Lib/concurrent/futures/thread.py). It may be interesting to dig into how the class works internally, perhaps after you are familiar with how it works from the outside.

The **ThreadPoolExecutor** extends the Executor class and will return **Future** objects when it is called.

- **Executor**: Parent class for the ThreadPoolExecutor that defines basic lifecycle operations for the pool.

- **Future**: Object returned when submitting tasks to the thread pool that may complete later.

Let’s take a closer look at **Executors**, **Futures**, and the lifecycle of using the **ThreadPoolExecutor** class.

What Are Executors

The **ThreadPoolExecutor** class extends the abstract **Executor** class.

The **Executor** class defines three methods used to control our thread pool; they are: **submit()**, **map()**, and **shutdown()**.

- **submit()**: Dispatch a function to be executed and return a future object.
- **map()**: Apply a function to an iterable of elements.
- **shutdown()**: Shut down the executor.

The **Executor** is started when the class is created and must be shut down explicitly by calling **shutdown()**, which will release any resources held by the **Executor**. We can also shut down automatically, but we will look at that a little later.

The **submit()** and **map()** functions are used to submit tasks to the Executor for asynchronous execution.

The **map()** function operates just like the **built-in map()** function and is used to apply a function to each element in an iterable object, like a list. Unlike the built-in **map()** function, each application of the function to an element will happen asynchronously instead of sequentially.

The **submit()** function takes a function, as well as any arguments, and will execute it asynchronously, although the call returns immediately and provides a **Future** object.

We will take a closer look at each of these three functions in a moment. Firstly, what is a Future?

What Are Futures

A future is an object that represents a delayed result for an asynchronous task (https://en.wikipedia.org/wiki/Futures_and_promises).

It is also sometimes called a promise or a delay. It provides a context for the result of a task that may or may not be executing and a way of getting a result once it is available.

In Python, the **Future** object is returned from an **Executor**, such as a **ThreadPoolExecutor** when calling the **submit()** function to dispatch a task to be executed asynchronously.

In general, we do not create **Future** objects; we only receive them and we may need to call functions on them.

There is always one **Future** object for each task sent into the **ThreadPoolExecutor** via a call to **submit()**.

The Future object provides a number of helpful functions for inspecting the status of the task such as: **cancelled()**, **running()**, and **done()** to determine if the task was cancelled, is currently running, or has finished execution.

- **cancelled()**: Returns **True** if the task was cancelled before being executed.
- **running()**: Returns **True** if the task is currently running.
- **done()**: Returns **True** if the task has completed or was cancelled.

A running task cannot be cancelled and a done task could have been cancelled.

A **Future** object also provides access to the result of the task via the **result()** function. If an exception was raised while executing the task, it will be re-raised when calling the **result()** function or can be accessed via the **exception()** function.

- **result()**: Access the result from running the task.
- **exception()**: Access any exception raised while running the task.

Both the **result()** and **exception()** functions allow a timeout to be specified as an argument, which is the number of seconds to wait for a return value if the task is not yet complete. If the timeout expires, then a **TimeoutError** will be raised.

Finally, we may want to have the thread pool automatically call a function once the task is completed.

This can be achieved by attaching a callback to the **Future** object for the task via the **add_done_callback()** function.

- **add_done_callback()**: Add a callback function to the task to be executed by the thread pool once the task is completed.

We can add more than one callback to each task and they will be executed in the order they were added. If the task has already completed before we add the callback, then the callback is executed immediately.

Any exceptions raised in the callback function will not impact the task or thread pool.

We will take a closer look at the **Future** object in a later section.

Now that we are familiar with the functionality of a **ThreadPoolExecutor** provided by the **Executor** class and of **Future** objects returned by calling **submit(),** let's take a closer look at the lifecycle of the **ThreadPoolExecutor** class.

Confused by the ThreadPoolExecutor class API?
Download my FREE PDF cheat sheet (https://marvelous-writer-6152.ck.page/5fb5f69c42)

LifeCycle of the ThreadPoolExecutor

The **ThreadPoolExecutor** provides a pool of generic worker threads.

The **ThreadPoolExecutor** was designed to be easy and straightforward to use.

If multithreading was like the transmission for changing gears in a car, then using threading.Thread is a manual transmission (e.g. hard to learn and and use) whereas **concurrency.futures.ThreadPoolExecutor** is an automatic transmission (e.g. easy to learn and use).

- **threading.Thread**: Manual threading in Python.

- **concurrency.futures.ThreadPoolExecutor**: Automatic or "*just work*" mode for threading in Python.

There are four main steps in the lifecycle of using the ThreadPoolExecutor class (https://superfastpython.com/threadpoolexecutor-quick-start-guide/); they are: create, submit, wait, and shut down.

- 1. Create: Create the thread pool by calling the constructor **ThreadPoolExecutor()**.
- 2. Submit: Submit tasks and get futures by calling **submit()** or **map()**.
- 3. Wait: Wait and get results as tasks complete (optional).
- 4. Shut down: Shut down the thread pool by calling **shutdown()**.

The following figure helps to picture the lifecycle of the ThreadPoolExecutor class.

THREADPOOLEXECUTOR LIFE-CYCLE

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

Step 1. Create the Thread Pool

First, a **ThreadPoolExecutor** instance must be created.

When an instance of a **ThreadPoolExecutor** is created, it must be configured with the fixed number of threads in the pool, a prefix used when naming each thread in the pool, and the name of a function to call when initializing each thread along with any arguments for the function.

The pool is created with one thread for each CPU in your system plus four. This is good for most purposes.

- Default Total Threads = (Total CPUs) + 4

For example, if you have 4 CPUs, each with hyperthreading (most modern CPUs have this), then Python will see 8 CPUs and will allocate (8 + 4) or 12 threads to the pool by default.

```
...
# create a thread pool with the default number of worker threads
executor = ThreadPoolExecutor()
```

It is a good idea to test your application in order to determine the number of threads that results in the best performance, anywhere from a few threads to hundreds of threads.

It is typically not a good idea to have thousands of threads as it may start to impact the amount of available RAM and results in a large amount of switching between threads, which may result in worse performance.

Well discuss tuning the number of threads for your pool more later on.

You can specify the number of threads to create in the pool via the max_workers argument; for example:

```
...
# create a thread pool with 10 worker threads
executor = ThreadPoolExecutor(max_workers=10)
```

Step 2. Submit Tasks to the Thread Pool

Once the thread pool has been created, you can submit tasks for asynchronous execution.

As discussed, there are two main approaches for submitting tasks defined on the Executor parent class. They are: map() and submit().

Step 2a. Submit Tasks With map()

The **map()** function is an asynchronous version of the built-in map() function (https://docs.python.org/3/library/functions.html#map) for applying a function to each element in an iterable, like a list.

You can call the map() function (https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.Executor.map) on the pool and pass it the name of your function and the iterable.

You are most likely to use **map()** when converting a for loop to run using one thread per loop iteration.

```
...
# perform all tasks in parallel
results = pool.map(my_task, my_items) # does not block
```

Where "**my_task**" is your function that you wish to execute and "**my_items**" is your iterable of objects, each to be executed by your "**my_task**" function.

The tasks will be queued up in the thread pool and executed by worker threads in the pool as they become available.

The **map()** function will return an iterable immediately. This iterable can be used to access the results from the target task function as they are available in the order that the tasks were submitted (e.g. order of the iterable you provided).

```
...
# iterate over results as they become available
for result in executor.map(my_task, my_items):
    print(result)
```

You can also set a timeout when calling **map()** via the "**timeout**" argument in seconds if you wish to impose a limit on how long you're willing to wait for each task to complete as you're iterating, after which a **TimeOut** error will be raised.

```
...
# perform all tasks in parallel
# iterate over results as they become available
for result in executor.map(my_task, my_items, timeout=5):
    # wait for task to complete or timeout expires
    print(result)
```

2a. Submit Tasks With submit()

The **submit()** function submits one task to the thread pool for execution.

The function takes the name of the function to call and all arguments to the function, then returns a **Future** object immediately.

The **Future** object is a promise to return the results from the task (if any) and provides a way to determine if a specific task has been completed or not.

```
...
# submit a task with arguments and get a future object
future = executor.submit(my_task, arg1, arg2) # does not block
```

Where "**my_task**" is the function you wish to execute and "**arg1**" and "**arg2**" are the first and second arguments to pass to the "**my_task**" function.

You can use the **submit()** function to submit tasks that do not take any arguments; for example:

```
...
# submit a task with no arguments and get a future object
future = executor.submit(my_task) # does not block
```

You can access the result of the task via the **result()** function on the returned **Future** object. This call will block until the task is completed.

```
...
# get the result from a future
result = future.result() # blocks
```

You can also set a timeout when calling **result()** via the "**timeout**" argument in seconds if you wish to impose a limit on how long you're willing to wait for each task to complete, after which a **TimeOut** error will be raised.

```
...
# wait for task to complete or timeout expires
result = future.result(timeout=5) # blocks
```

Step 3. Wait for Tasks to Complete (Optional)

The **concurrent.futures** module provides two module utility functions for waiting for tasks via their **Future** objects.

Recall that **Future** objects are only created when we call **submit()** to push tasks into the thread pool.

These wait functions are optional to use, as you can wait for results directly after calling **map()** or **submit()** or wait for all tasks in the thread pool to finish.

These two module functions are **wait()** for waiting for **Future** objects to complete and **as_completed()** for getting **Future** objects as their tasks complete.

- **wait()**: Wait on one or more **Future** objects until they are completed.
- **as_completed()**: Returns **Future** objects from a collection as they complete their execution.

You can use both functions with **Future** objects created by one or more thread pools, they are not specific to any given thread pool in your application. This is helpful if you want to perform waiting operations across multiple thread pools that are executing different types of tasks.

Both functions are useful to use with an idiom of dispatching multiple tasks into the thread pool via submit in a list compression; for example:

```
...
# dispatch tasks into the thread pool and create a list of futures
futures = [executor.submit(my_task, my_data) for my_data in my_datalist]
```

Here, my_task is our custom target task function, "**my_data**" is one element of data passed as an argument to "**my_task**", and "**my_datalist**" is our source of **my_data** objects.

We can then pass the **Future** objects to **wait()** or **as_completed()**.

Creating a list of **Future** objects in this way is not required, just a common pattern when converting for loops into tasks submitted to a thread pool.

Step 3a. Wait for Futures to Complete

The **wait()** function (https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.wait) can take one or more **Future** objects and will return when a specified action occurs, such as all tasks completing, one task completing, or one task raising an exception.

The function will return one set of **Future** objects that match the condition set via the "**return_when**". The second set will contain all of the futures for tasks that did not meet the condition. These are called the "**done**" and the "**not_done**" sets of futures.

It is useful for waiting on a large batch of work and to stop waiting when we get the first result.

This can be achieved via the **FIRST_COMPLETED** constant passed to the "**return_when**" argument.

```
...
# wait until we get the first result
done, not_done = wait(futures, return_when=concurrent.futures.FIRST_COMPLETED)
```

Alternatively, we can wait for all tasks to complete via the **ALL_COMPLETED** constant.

This can be helpful if you are using **submit()** to dispatch tasks and are looking for an easy way to wait for all work to be completed.

```
...
# wait for all tasks to complete
done, not_done = wait(futures, return_when=concurrent.futures.ALL_COMPLETED)
```

There is also an option to wait for the first exception via the **FIRST_EXCEPTION** constant.

```
...
# wait for the first exception
done, not_done = wait(futures, return_when=concurrent.futures.FIRST_EXCEPTION)
```

Step 3b. Wait for Futures as Completed

The beauty of performing tasks concurrently is that we can get results as they become available, rather than waiting for all tasks to be completed.

The as_completed() function (https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.as_completed) will return **Future** objects for tasks as they are completed in the thread pool.

We can call the function and provide it a list of **Future** objects created by calling **submit()** and it will return **Future** objects as they are completed in whatever order.

It is common to use the **as_completed()** function in a loop over the list of **Future** objects created when calling submit; for example:

```
...
# iterate over all submitted tasks and get results as they are available
for future in as_completed(futures):
    # get the result for the next completed task
    result = future.result() # blocks
```

Note: this is different from iterating over the results from calling **map()** in two ways. Firstly, **map()** returns an iterator over objects, not over **Future** objects. Secondly, **map()** returns results in the order that the tasks were submitted, not in the order that they are completed.

Step 4. Shutdown the Thread Pool

Once all tasks are completed, we can close down the thread pool, which will release each thread and any resources it may hold (e.g. the thread stack space).

```
...
# shutdown the thread pool
executor.shutdown() # blocks
```

The **shutdown()** function will wait for all tasks in the thread pool to complete before returning by default.

This behavior can be changed by setting the “**wait**” argument to **False** when calling **shutdown()**, in which case the function will return immediately. The resources used by thread pool will not be released until all current and queued tasks are completed.

```
...
# shutdown the thread pool
executor.shutdown(wait=False) # does not blocks
```

We can also instruct the pool to cancel all queued tasks to prevent their execution. This can be achieved by setting the “**cancel_futures**” argument to **True**. By default queued tasks are not cancelled when calling **shutdown()**.

```
...
# cancel all queued tasks
executor.shutdown(cancel_futures=True) # blocks
```

If we forget to close the thread pool, the thread pool will be closed automatically when we exit the main thread. If we forget to close the pool and there are still tasks executing, the main thread will not exit until all tasks in the pool and all queued tasks have executed.

ThreadPoolExecutor Context Manager

A preferred way to work with the **ThreadPoolExecutor** class is to use a context manager.

This matches the preferred way to work with other resources, such as files and sockets.

Using the **ThreadPoolExecutor** with a context manager involves using the "**with**" keyword to create a block in which you can use the thread pool to execute tasks and get results.

Once the block has completed, the thread pool is automatically shut down. Internally, the context manager will call the **shutdown()** function with the default arguments, waiting for all queued and executing tasks to complete before returning and carrying on.

Below is a code snippet to demonstrate creating a thread pool using the context manager.

```
...
# create a thread pool
with ThreadPoolExecutor(max_workers=10) as pool:
    # submit tasks and get results
    # ...
    # automatically shutdown the thread pool...
# the pool is shutdown at this point
```

This is a very handy idiom if you are converting a for loop to be multithreaded.

It is less useful if you want the thread pool to operate in the background while you perform other work in the main thread of your program, or if you wish to reuse the thread pool multiple times throughout your program.

Now that we are familiar with how to use the **ThreadPoolExecutor**, let's look at some worked examples.

Free Python ThreadPoolExecutor Course

ThreadPoolExecutor Example

In this section, we will look at a more complete example of using the **ThreadPoolExecutor**.

Perhaps the most common use case for the **ThreadPoolExecutor** is to download files (https://superfastpython.com/threadpoolexecutor-example/) from the internet concurrently.

It's a useful problem because there are many ways we can download files. We will use this problem as the basis to explore the different patterns with the **ThreadPoolExecutor** for downloading files concurrently.

First, let's develop a serial (non-concurrent) version of the program.

Download Files Serially

Consider the situation where we might want to have a local copy of some of the Python API documentation on concurrency for later view.

Perhaps we are taking a flight and won't have internet access and will need to refer to the documentation in HTML format as it appears on the docs.python.org website. It's a contrived scenario; Python is installed with docs and we also have the pydoc command, but go with me here.

We may want to download local copies of the following ten URLs that cover the extent of the Python concurrency APIs.

We can define these URLs as a list of strings for processing in our program.

```python
# python concurrency API docs
URLS = ['https://docs.python.org/3/library/concurrency.html',
        'https://docs.python.org/3/library/concurrent.html',
        'https://docs.python.org/3/library/concurrent.futures.html',
        'https://docs.python.org/3/library/threading.html',
        'https://docs.python.org/3/library/multiprocessing.html',
        'https://docs.python.org/3/library/multiprocessing.shared_memory.html',
        'https://docs.python.org/3/library/subprocess.html',
        'https://docs.python.org/3/library/queue.html',
        'https://docs.python.org/3/library/sched.html',
        'https://docs.python.org/3/library/contextvars.html']
```

URLs are reasonably easy to download in Python.

First, we can attempt to open a connection to the server using the urllib.request.urlopen() (https://docs.python.org/3/library/urllib.request.html) function and specify the URL and a reasonable timeout in seconds.

This will give a connection, which we can then call the **read()** function to read the contents of the file. Using the context manager for the connection will ensure it will be closed automatically, even if an exception is raised.

The **download_url()** function below implements this, taking a URL as a parameter and returning the contents of the file or **None** if the file cannot be downloaded for whatever reason. We will set a lengthy timeout of 3 seconds in case our internet connection is flaky for some reason.

```python
# download a url and return the raw data, or None on error
def download_url(url):
    try:
        # open a connection to the server
        with urlopen(url, timeout=3) as connection:
            # read the contents of the html doc
            return connection.read()
    except:
        # bad url, socket timeout, http forbidden, etc.
        return None
```

Once we have the data for a URL, we can save it as a local file.

First, we need to retrieve the filename of the file specified in the URL. There are a few ways to do this, but the **os.path.basename()** function is a common approach when working with paths. We can then use the os.path.join() (https://docs.python.org/3/library/os.path.html) function to construct an output path for saving the file, using a directory we specify and the filename.

We can then use the **open()** built-in function to open the file in write-binary mode and save the contents of the file, again using the context manager to ensure the file is closed once we are finished.

The **save_file()** function below implements this, taking the URL that was downloaded, the contents of the file that was downloaded, and the local output path where we wish to save downloaded files. It returns the output path that was used to save the file, in case we want to report progress to the user.

```
# save data to a local file
def save_file(url, data, path):
    # get the name of the file from the url
    filename = basename(url)
    # construct a local path for saving the file
    outpath = join(path, filename)
    # save to file
    with open(outpath, 'wb') as file:
        file.write(data)
    return outpath
```

Next, we can call the **download_url()** function for a given URL in our list then **save_file()** to save each downloaded file.

The **download_and_save()** function below implements this, reporting progress along the way, and handling the case of URLs that cannot be downloaded.

```
# download and save a url as a local file
def download_and_save(url, path):
    # download the url
    data = download_url(url)
    # check for no data
    if data is None:
        print(f'>Error downloading {url}')
        return
    # save the data to a local file
    outpath = save_file(url, data, path)
    # report progress
    print(f'>Saved {url} to {outpath}')
```

Finally, we need a function to drive the process.

First, the local output location where we will be saving files needs to be created, if it does not exist. We can achieve this using the os.makedirs() (https://docs.python.org/3/library/os.html) function.

We can iterate over a list of URLs and call our **download_and_save()** function for each.

The **download_docs()** function below implements this.

```
# download a list of URLs to local files
def download_docs(urls, path):
    # create the local directory, if needed
    makedirs(path, exist_ok=True)
    # download each url and save as a local file
    for url in urls:
        download_and_save(url, path)
```

And that’s it.

We can then call our **download_docs()** with our list of URLs and an output directory. In this case, we will use a ‘**docs/**‘ subdirectory of our current working directory (where the Python script is located) as the output directory.

Tying this together, the complete example of downloading files serially is listed below.

```python
# SuperFastPython.com
# download document files and save to local files serially
from os import makedirs
from os.path import basename
from os.path import join
from urllib.request import urlopen

# download a url and return the raw data, or None on error
def download_url(url):
    try:
        # open a connection to the server
        with urlopen(url, timeout=3) as connection:
            # read the contents of the html doc
            return connection.read()
    except:
        # bad url, socket timeout, http forbidden, etc.
        return None

# save data to a local file
def save_file(url, data, path):
    # get the name of the file from the url
    filename = basename(url)
    # construct a local path for saving the file
    outpath = join(path, filename)
    # save to file
    with open(outpath, 'wb') as file:
        file.write(data)
    return outpath

# download and save a url as a local file
def download_and_save(url, path):
    # download the url
    data = download_url(url)
    # check for no data
    if data is None:
        print(f'>Error downloading {url}')
        return
    # save the data to a local file
    outpath = save_file(url, data, path)
    # report progress
    print(f'>Saved {url} to {outpath}')

# download a list of URLs to local files
def download_docs(urls, path):
    # create the local directory, if needed
    makedirs(path, exist_ok=True)
    # download each url and save as a local file
    for url in urls:
        download_and_save(url, path)

# python concurrency API docs
URLS = ['https://docs.python.org/3/library/concurrency.html',
        'https://docs.python.org/3/library/concurrent.html',
        'https://docs.python.org/3/library/concurrent.futures.html',
        'https://docs.python.org/3/library/threading.html',
        'https://docs.python.org/3/library/multiprocessing.html',
        'https://docs.python.org/3/library/multiprocessing.shared_memory.html',
        'https://docs.python.org/3/library/subprocess.html',
        'https://docs.python.org/3/library/queue.html',
        'https://docs.python.org/3/library/sched.html',
        'https://docs.python.org/3/library/contextvars.html']
# local path for saving the files
PATH = 'docs'
# download all docs
download_docs(URLS, PATH)
```

Running the example iterates over the list of URLs and downloads each in turn.

Each file is then saved to a local file in the specified directory.

The process takes about 700 milliseconds to about one second (1,000 milliseconds) on my system.

Try running it a few times; how long does it take on your system?
Let me know in the comments.

```
>Saved https://docs.python.org/3/library/concurrency.html to docs/concurrency.html
>Saved https://docs.python.org/3/library/concurrent.html to docs/concurrent.html
>Saved https://docs.python.org/3/library/concurrent.futures.html to docs/concurrent.futures.html
>Saved https://docs.python.org/3/library/threading.html to docs/threading.html
>Saved https://docs.python.org/3/library/multiprocessing.html to docs/multiprocessing.html
>Saved https://docs.python.org/3/library/multiprocessing.shared_memory.html to docs/multiprocessing.sharec
>Saved https://docs.python.org/3/library/subprocess.html to docs/subprocess.html
>Saved https://docs.python.org/3/library/queue.html to docs/queue.html
>Saved https://docs.python.org/3/library/sched.html to docs/sched.html
>Saved https://docs.python.org/3/library/contextvars.html to docs/contextvars.html
```

Next, we can look at making the program concurrent using a thread pool.

Download Files Concurrently With submit()

Let's look at updating our program to make use of the **ThreadPoolExecutor** to download files concurrently.

A first thought might be to use **map()** as we just want to make a for-loop concurrent.

Unfortunately, the **download_and_save()** function that we call each iteration in the loop takes two parameters, only one of which is an iterable.

An alternate approach is to use submit() to call **download_and_save()** in a separate thread for each URL in the provided list.

We can do this by first configuring a thread pool with the number of threads equal to the number of URLs in the list. We'll use the context manager for the thread pool so that it will be closed automatically for us when we finish.

We can then call the **submit()** function for each URL using a list compression. We don't even need the Future objects returned from calling submit, as there is no result we're waiting for.

```
...
# create the thread pool
n_threads = len(urls)
with ThreadPoolExecutor(n_threads) as executor:
    # download each url and save as a local file
    _ = [executor.submit(download_and_save, url, path) for url in urls]
```

Once each thread has completed, the context manager will close the thread pool for us and we're done.

We don't even need to add an explicit call to wait, although we could if we wanted to make the code more readable; for example:

```
...
# create the thread pool
n_threads = len(urls)
with ThreadPoolExecutor(n_threads) as executor:
    # download each url and save as a local file
    futures = [executor.submit(download_and_save, url, path) for url in urls]
    # wait for all download tasks to complete
    _, _ = wait(futures)
```

But, adding this wait is not needed.

The updated version of our **download_docs()** function that downloads and saves the files concurrently is listed below.

```
# download a list of URLs to local files
def download_docs(urls, path):
    # create the local directory, if needed
    makedirs(path, exist_ok=True)
    # create the thread pool
    n_threads = len(urls)
    with ThreadPoolExecutor(n_threads) as executor:
        # download each url and save as a local file
        _ = [executor.submit(download_and_save, url, path) for url in urls]
```

Tying this together, the complete example is listed below.

```python
# SuperFastPython.com
# download document files and save to local files concurrently
from os import makedirs
from os.path import basename
from os.path import join
from urllib.request import urlopen
from concurrent.futures import ThreadPoolExecutor

# download a url and return the raw data, or None on error
def download_url(url):
    try:
        # open a connection to the server
        with urlopen(url, timeout=3) as connection:
            # read the contents of the html doc
            return connection.read()
    except:
        # bad url, socket timeout, http forbidden, etc.
        return None

# save data to a local file
def save_file(url, data, path):
    # get the name of the file from the url
    filename = basename(url)
    # construct a local path for saving the file
    outpath = join(path, filename)
    # save to file
    with open(outpath, 'wb') as file:
        file.write(data)
    return outpath

# download and save a url as a local file
def download_and_save(url, path):
    # download the url
    data = download_url(url)
    # check for no data
    if data is None:
        print(f'>Error downloading {url}')
        return
    # save the data to a local file
    outpath = save_file(url, data, path)
    # report progress
    print(f'>Saved {url} to {outpath}')

# download a list of URLs to local files
def download_docs(urls, path):
    # create the local directory, if needed
    makedirs(path, exist_ok=True)
    # create the thread pool
    n_threads = len(urls)
    with ThreadPoolExecutor(n_threads) as executor:
        # download each url and save as a local file
        _ = [executor.submit(download_and_save, url, path) for url in urls]

# python concurrency API docs
URLS = ['https://docs.python.org/3/library/concurrency.html',
        'https://docs.python.org/3/library/concurrent.html',
        'https://docs.python.org/3/library/concurrent.futures.html',
        'https://docs.python.org/3/library/threading.html',
        'https://docs.python.org/3/library/multiprocessing.html',
        'https://docs.python.org/3/library/multiprocessing.shared_memory.html',
        'https://docs.python.org/3/library/subprocess.html',
        'https://docs.python.org/3/library/queue.html',
        'https://docs.python.org/3/library/sched.html',
        'https://docs.python.org/3/library/contextvars.html']
# local path for saving the files
PATH = 'docs'
# download all docs
download_docs(URLS, PATH)
```

Running the example downloads and saves the files as before.

This time, the operation is complete in a fraction of a second. About 300 milliseconds in my case, which is less than half the time it took to download all files serially in the previous example.

How long did it take to download all files on your system?

```
>Saved https://docs.python.org/3/library/concurrent.html to docs/concurrent.html
>Saved https://docs.python.org/3/library/multiprocessing.shared_memory.html to docs/multiprocessing.shared
>Saved https://docs.python.org/3/library/concurrency.html to docs/concurrency.html
>Saved https://docs.python.org/3/library/sched.html to docs/sched.html
>Saved https://docs.python.org/3/library/contextvars.html to docs/contextvars.html
>Saved https://docs.python.org/3/library/queue.html to docs/queue.html
>Saved https://docs.python.org/3/library/concurrent.futures.html to docs/concurrent.futures.html
>Saved https://docs.python.org/3/library/threading.html to docs/threading.html
>Saved https://docs.python.org/3/library/subprocess.html to docs/subprocess.html
>Saved https://docs.python.org/3/library/multiprocessing.html to docs/multiprocessing.html
```

This is one approach to making the program concurrent, but let's look at some alternatives.

Download Files Concurrently With submit() and as_completed()

Perhaps we want to report the progress of downloads as they are completed.

The thread pool allows us to do this by storing the **Future** objects returned from calls to **submit()** and then calling the **as_completed()** on the collection of **Future** objects.

Also, consider that we are doing two things in the task. The first is a downloading from a remote server, which is an IO-bound operation that we can make concurrently. The second is saving the contents of the file to the local hard drive, which is another IO-bound operation that we cannot make concurrently as most hard drives can only save one file at a time.

Therefore, perhaps a better design is to only make the file downloading part of the program a concurrent task and the file saving part of the program serial.

This will require more changes to the program.

We can call the **download_url()** function for each URL and this can be our concurrent task submitted to the thread pool.

When we call **result()** on each **Future** object, it will give us the data that was downloaded, but we won't know what URL the data was downloaded from. The **Future** object won't know.

Therefore, we can update the **download_url()** to return both the data that was downloaded and the URL that was provided as an argument.

The updated version of the **download_url()** function that returns a tuple of data and the input URL is listed below.

```python
# download a url and return the raw data, or None on error
def download_url(url):
    try:
        # open a connection to the server
        with urlopen(url, timeout=3) as connection:
            # read the contents of the html doc
            return (connection.read(), url)
    except:
        # bad url, socket timeout, http forbidden, etc.
        return (None, url)
```

We can then submit a call to this function to each URL to the thread pool to give us a **Future** object.

```python
...
# download each url and save as a local file
futures = [executor.submit(download_url, url) for url in urls]
```

So far, so good.

Now, we want to save local files and report progress as the files are downloaded.

This requires that we cannibalize the **download_and_save()** function and move it back into the **download_docs()** function used to drive the program.

We can iterate over the futures via the **as_completed()** function that will return **Future** objects in the order that the downloads are completed, not the order that we dispatched them into the thread pool.

We can then retrieve the data and URL from the **Future** object.

```python
...
# process each result as it is available
for future in as_completed(futures):
    # get the downloaded url data
    data, url = future.result()
```

We can check if the download was unsuccessful and report an error, otherwise save the file and report progress as per normal. A direct copy-paste from the **download_and_save()** function.

```
...
# check for no data
if data is None:
    print(f'>Error downloading {url}')
    continue
# save the data to a local file
outpath = save_file(url, data, path)
# report progress
print(f'>Saved {url} to {outpath}')
```

The updated version of our **download_docs()** function that will only download files concurrently then save the files serially as the files are downloaded is listed below.

```
# download a list of URLs to local files
def download_docs(urls, path):
    # create the local directory, if needed
    makedirs(path, exist_ok=True)
    # create the thread pool
    n_threads = len(urls)
    with ThreadPoolExecutor(n_threads) as executor:
        # download each url and save as a local file
        futures = [executor.submit(download_url, url) for url in urls]
        # process each result as it is available
        for future in as_completed(futures):
            # get the downloaded url data
            data, url = future.result()
            # check for no data
            if data is None:
                print(f'>Error downloading {url}')
                continue
            # save the data to a local file
            outpath = save_file(url, data, path)
            # report progress
            print(f'>Saved {url} to {outpath}')
```

Tying this together, the complete example is listed below.

```python
# SuperFastPython.com
# download document files concurrently and save the files locally serially
from os import makedirs
from os.path import basename
from os.path import join
from urllib.request import urlopen
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed

# download a url and return the raw data, or None on error
def download_url(url):
    try:
        # open a connection to the server
        with urlopen(url, timeout=3) as connection:
            # read the contents of the html doc
            return (connection.read(), url)
    except:
        # bad url, socket timeout, http forbidden, etc.
        return (None, url)

# save data to a local file
def save_file(url, data, path):
    # get the name of the file from the url
    filename = basename(url)
    # construct a local path for saving the file
    outpath = join(path, filename)
    # save to file
    with open(outpath, 'wb') as file:
        file.write(data)
    return outpath

# download a list of URLs to local files
def download_docs(urls, path):
    # create the local directory, if needed
    makedirs(path, exist_ok=True)
    # create the thread pool
    n_threads = len(urls)
    with ThreadPoolExecutor(n_threads) as executor:
        # download each url and save as a local file
        futures = [executor.submit(download_url, url) for url in urls]
        # process each result as it is available
        for future in as_completed(futures):
            # get the downloaded url data
            data, url = future.result()
            # check for no data
            if data is None:
                print(f'>Error downloading {url}')
                continue
            # save the data to a local file
            outpath = save_file(url, data, path)
            # report progress
            print(f'>Saved {url} to {outpath}')

# python concurrency API docs
URLS = ['https://docs.python.org/3/library/concurrency.html',
        'https://docs.python.org/3/library/concurrent.html',
        'https://docs.python.org/3/library/concurrent.futures.html',
        'https://docs.python.org/3/library/threading.html',
        'https://docs.python.org/3/library/multiprocessing.html',
        'https://docs.python.org/3/library/multiprocessing.shared_memory.html',
        'https://docs.python.org/3/library/subprocess.html',
        'https://docs.python.org/3/library/queue.html',
        'https://docs.python.org/3/library/sched.html',
        'https://docs.python.org/3/library/contextvars.html']
# local path for saving the files
PATH = 'docs'
# download all docs
download_docs(URLS, PATH)
```

Running the program, the files are downloaded and saved as before, perhaps a few milliseconds faster.

Looking at the output of the program, we can see that the order of saved files is different.

Smaller files like "**sched.html**" that were dispatched nearly last were downloaded sooner (e.g. less bytes to download) and in turn were saved to local files sooner.

This confirms that we are indeed processing downloads in their order of task completion and not the order the tasks were submitted.

```
>Saved https://docs.python.org/3/library/concurrent.html to docs/concurrent.html
>Saved https://docs.python.org/3/library/sched.html to docs/sched.html
>Saved https://docs.python.org/3/library/concurrency.html to docs/concurrency.html
>Saved https://docs.python.org/3/library/contextvars.html to docs/contextvars.html
>Saved https://docs.python.org/3/library/queue.html to docs/queue.html
>Saved https://docs.python.org/3/library/multiprocessing.shared_memory.html to docs/multiprocessing.shared
>Saved https://docs.python.org/3/library/threading.html to docs/threading.html
>Saved https://docs.python.org/3/library/concurrent.futures.html to docs/concurrent.futures.html
>Saved https://docs.python.org/3/library/subprocess.html to docs/subprocess.html
>Saved https://docs.python.org/3/library/multiprocessing.html to docs/multiprocessing.html
```

Now that we have seen some examples, let's look at some common usage patterns when using the **ThreadPoolExecutor**.

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

ThreadPoolExecutor Usage Patterns

The **ThreadPoolExecutor** provides a lot of flexibility for executing concurrent tasks in Python.

Nevertheless, there are a handful of common usage patterns that will fit most program scenarios.

This section lists the common usage patterns (https://superfastpython.com/threadpoolexecutor-usage-patterns/) with worked examples that you can copy-and-paste into your own project and adapt as needed.

The patterns we will look at are as follows:

- Map and Wait Pattern
- Submit and Use as Completed Pattern

- Submit and Use Sequentially Pattern
- Submit and Use Callback Pattern
- Submit and Wait for All Pattern
- Submit and Wait for First Pattern

We will use a contrived task in each example that will sleep for a random amount of time less than one second. You can easily replace this example task with your own task in each pattern.

Also, recall that each Python program has one thread by default called the main thread where we do our work. We will create the thread pool in the main thread in each example and may reference actions in the main thread in some of the patterns, as opposed to actions in threads in the thread pool.

Map and Wait Pattern

Perhaps the most common pattern when using the **ThreadPoolExecutor** is to convert a for loop that executes a function on each item in a collection to use threads.

It assumes that the function has no side effects, meaning it does not access any data outside of the function and does not change the data provided to it. It takes data and produces a result.

These types of for loops can be written explicitly in Python; for example:

```
...
# apply a function to each element in a collection
for item in mylist:
    result = task(item)
```

A better practice is to use the built-in **map()** function that applies the function to each item in the iterable for you.

```
...
# apply the function to each element in the collection
results = map(task, mylist)
```

This does not perform the **task()** function to each item until we iterate the results, so-called lazy evaluation:

```
...
# iterate the results from map
for result in results:
    print(result)
```

Therefore, it is common to see this operation consolidated to the following:

```
...
# iterate the results from map
for result in map(task, mylist):
    print(result)
```

We can perform this same operation using the thread pool, except each application of the function to an item in the list is a task that is executed asynchronously. For example:

```
...
# iterate the results from map
for result in executor.map(task, mylist):
    print(result)
```

Although the tasks are executed concurrently, the results are iterated in the order of the iterable provided to the map() function, the same as the built-in **map()** function.

In this way, we can think of the thread pool version of **map()** as a concurrent version of the **map()** function and is ideal if you are looking to update your for loop to use threads.

The example below demonstrates using the map and wait pattern with a task that will sleep a random amount of time less than one second and return the provided value.

```
# SuperFastPython.com
# example of the map and wait pattern for the ThreadPoolExecutor
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())
    return name

# start the thread pool
with ThreadPoolExecutor(10) as executor:
    # execute tasks concurrently and process results in order
    for result in executor.map(task, range(10)):
        # retrieve the result
        print(result)
```

Running the example, we can see that the results are reported in the order that the tasks were created and sent into the thread pool.

```
0
1
2
3
4
5
6
7
8
9
```

The **map()** function supports target functions that take more than one argument by providing more than iterable as arguments to the call to **map()**.

For example, we can define a target function for map that takes two arguments, then provide two iterables of the same length to the call to map.

The complete example is listed below.

```python
# SuperFastPython.com
# example of calling map with two iterables
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(value1, value2):
    # sleep for less than a second
    sleep(random())
    return (value1, value2)

# start the thread pool
with ThreadPoolExecutor() as executor:
    # submit all tasks
    for result in executor.map(task, ['1', '2', '3'], ['a', 'b', 'c']):
        print(result)
```

Running the example executes the tasks as expected, providing two arguments to map and reporting a result that combines both arguments.

```
('1', 'a')
('2', 'b')
('3', 'c')
```

A call to the map function will issue all tasks to the thread pool immediately, even if you do not iterate the iterable of results.

This is unlike the built-in **map()** function that is lazy and does not compute each call until you ask for the result during iteration.

The example below confirms this by issuing all tasks with a map and not iterating the results.

```python
# SuperFastPython.com
# example of calling map and not iterating the results
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(value):
    # sleep for less than a second
    sleep(random())
    print(f'Done: {value}')
    return value

# start the thread pool
with ThreadPoolExecutor() as executor:
    # submit all tasks
    executor.map(task, range(5))
print('All done!')
```

Running the example, we can see that the tasks are sent into the thread pool and executed without having to explicitly pass over the iterable of results that was returned.

The use of the context manager ensured that the thread pool did not shutdown until all tasks were complete.

```
Done: 0
Done: 2
Done: 1
Done: 3
Done: 4
All done!
```

Submit and Use as Completed

Perhaps the second most common pattern when using the **ThreadPoolExecutor** is to submit tasks and use the results as they become available.

This can be achieved using the **submit()** function to push tasks into the thread pool that returns **Future** objects, then calling the module method **as_completed()** on the list of Future objects that will return each **Future** object as it's task is completed.

The example below demonstrates this pattern, submitting the tasks in order from 0 to 9 and showing results in the order that they were completed.

```python
# SuperFastPython.com
# example of the submit and use as completed pattern for the ThreadPoolExecutor
from time import sleep
from random import random
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed

# custom task that will sleep for a variable amount of time
def task(name):
    # sleep for less than a second
    sleep(random())
    return name

# start the thread pool
with ThreadPoolExecutor(10) as executor:
    # submit tasks and collect futures
    futures = [executor.submit(task, i) for i in range(10)]
    # process task results as they are available
    for future in as_completed(futures):
        # retrieve the result
        print(future.result())
```

Running the example, we can see that the results are retrieved and printed in the order that the tasks completed, not the order that the tasks were submitted to the thread pool.

```
5
9
6
1
0
7
3
8
4
2
```

Submit and Use Sequentially

We may require the results from tasks in the order that the tasks were submitted.

This may be because the tasks have a natural ordering.

We can implement this pattern by calling **submit()** for each task to get a list of **Future** objects, then iterating over the **Future** objects in the order that the tasks were submitted and retrieving the results.

The main difference from the "*as completed*" pattern is that we enumerate the list of futures directly, instead of calling the **as_completed()** function.

```python
...
# process task results in the order they were submitted
for future in futures:
    # retrieve the result
    print(future.result())
```

The example below demonstrates this pattern, submitting the tasks in order from 0 to 9 and showing the results in the order that they were submitted.

```python
# SuperFastPython.com
# example of the submit and use sequentially pattern for the ThreadPoolExecutor
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())
    return name

# start the thread pool
with ThreadPoolExecutor(10) as executor:
    # submit tasks and collect futures
    futures = [executor.submit(task, i) for i in range(10)]
    # process task results in the order they were submitted
    for future in futures:
        # retrieve the result
        print(future.result())
```

Running the example, we can see that the results are retrieved and printed in the order that the tasks were submitted, not the order that the tasks were completed.

```
0
1
2
3
4
5
6
7
8
9
```

Submit and Use Callback

We may not want to explicitly process the results once they are available; instead, we want to call a function on the result.

Instead of doing this manually, such as in the as completed pattern above, we can have the thread pool call the function for us with the result automatically.

This can be achieved by setting a callback on each **Future** object by calling the **add_done_callback()** function and passing the name of the function.

The thread pool will then call the callback function (https://superfastpython.com/threadpoolexecutor-add-callback/) as each task completes, passing in **Future** objects for the task.

The example below demonstrates this pattern, registering a custom callback function to be applied to each task as it is completed.

```python
# SuperFastPython.com
# example of the submit and use a callback pattern for the ThreadPoolExecutor
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())
    return name

# custom callback function called on tasks when they complete
def custom_callback(fut):
    # retrieve the result
    print(fut.result())

# start the thread pool
with ThreadPoolExecutor(10) as executor:
    # submit tasks and collect futures
    futures = [executor.submit(task, i) for i in range(10)]
    # register the callback on all tasks
    for future in futures:
        future.add_done_callback(custom_callback)
    # wait for tasks to complete...
```

Running the example, we can see that results are retrieved and printed in the order they are completed, not the order that tasks were completed.

```
8
0
7
1
4
6
5
3
2
9
```

We can register multiple callbacks on each **Future** object; it is not limited to a single callback.

The callback functions are called in the order in which they were registered on each **Future** object.

The following example demonstrates having two callbacks on each **Future**.

```
# SuperFastPython.com
# example of the submit and use multiple callbacks for the ThreadPoolExecutor
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())
    return name

# custom callback function called on tasks when they complete
def custom_callback1(fut):
    # retrieve the result
    print(f'Callback 1: {fut.result()}')

# custom callback function called on tasks when they complete
def custom_callback2(fut):
    # retrieve the result
    print(f'Callback 2: {fut.result()}')

# start the thread pool
with ThreadPoolExecutor(10) as executor:
    # submit tasks and collect futures
    futures = [executor.submit(task, i) for i in range(10)]
    # register the callbacks on all tasks
    for future in futures:
        future.add_done_callback(custom_callback1)
        future.add_done_callback(custom_callback2)
    # wait for tasks to complete...
```

Running the example, we can see that results are reported in the order that tasks were completed and that the two callback functions are called for each task in the order that we registered them with each **Future** object.

```
Callback 1: 3
Callback 2: 3
Callback 1: 9
Callback 2: 9
Callback 1: 7
Callback 2: 7
Callback 1: 2
Callback 2: 2
Callback 1: 0
Callback 2: 0
Callback 1: 5
Callback 2: 5
Callback 1: 1
Callback 2: 1
Callback 1: 8
Callback 2: 8
Callback 1: 4
Callback 2: 4
Callback 1: 6
Callback 2: 6
```

Submit and Wait for All

It is common to submit all tasks and then wait for all tasks in the thread pool to complete.

This pattern may be useful when tasks do not return a result directly, such as if each task stores the result in a resource directly like a file.

There are two ways that we can wait for tasks to complete: by calling the **wait()** module function or by calling **shutdown()**.

The most likely case is you want to explicitly wait for a set or subset of tasks in the thread pool to complete.

You can achieve this by passing the list of tasks to the **wait()** function, which, by default, will wait for all tasks to complete.

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

We can explicitly specify to wait for all tasks by setting the "**return_when**" argument to the **ALL_COMPLETED** constant; for example:

```
...
# wait for all tasks to complete
wait(futures, return_when=ALL_COMPLETED)
```

The example below demonstrates this pattern. Note that we are intentionally ignoring the return from calling **wait()** as we have no need to inspect it in this case.

```
# SuperFastPython.com
# example of the submit and wait for all pattern for the ThreadPoolExecutor
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())
    # display the result
    print(name)

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

Running the example, we can see that results are handled by each task as the tasks complete. Importantly, we can see that the main thread waits until all tasks are completed before carrying on and printing a message.

```
3
9
0
8
4
6
2
1
5
7
All tasks are done!
```

An alternative approach would be to shut down the thread pool and wait for all executing and queued tasks to complete before moving on.

This might be preferred when we don't have a list of **Future** objects or when we only intend to use the thread pool once for a set of tasks.

We can implement this pattern using the context manager; for example:

```
# SuperFastPython.com
# example of the submit and wait for all with shutdown pattern for the ThreadPoolExecutor
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())
    # display the result
    print(name)

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

Running the example, we can see that the main thread does not move on and print the message until all tasks are completed, after the thread pool has been automatically shut down by the context manager.

```
1
2
8
4
5
3
9
0
7
6
All tasks are done!
```

The context manager automatic shutdown pattern might be confusing to developers not used to how thread pools work, hence the comment at the end of the context manager block in the previous example.

We can achieve the same effect without the context manager and an explicit call to shutdown.

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

Recall that the **shutdown()** function will wait for all tasks to complete by default and will not cancel any queued tasks, but we can make this explicit by setting the “**wait**” argument to **True** and the “**cancel_futures**” argument to **False**; for example:

```
...
# wait for all tasks to complete and close the pool
executor.shutdown(wait=True, cancel_futures=False)
```

The example below demonstrates the pattern of waiting for all tasks in the thread pool to complete by calling **shutdown()** before moving on.

```
# SuperFastPython.com
# example of the submit and wait for all with shutdown pattern for the ThreadPoolExecutor
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())
    # display the result
    print(name)

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

Running the example, we can see that all tasks report their result as they complete and that the main thread does not move on until all tasks have completed and the thread pool has been shut down.

```
3
5
2
6
8
9
7
1
4
0
All tasks are done!
```

Submit and Wait for First

It is common to issue many tasks and only be concerned with the first result returned.

That is, not the result of the first task, but a result from any task that happens to be the first to complete its execution.

This may be the case if you are trying to access the same resource from multiple locations, like a file or some data.

This pattern can be achieved using the **wait()** module function and setting the “**return_when**” argument to the **FIRST_COMPLETED** constant.

```
...
# wait until any task completes
done, not_done = wait(futures, return_when=FIRST_COMPLETED)
```

We must also manage the thread pool manually by constructing it and calling **shutdown()** manually so that we can continue on with the execution of the main thread without waiting for all of the other tasks to complete.

The example below demonstrates this pattern and will stop waiting as soon as the first task is completed.

```
# SuperFastPython.com
# example of the submit and wait for first the ThreadPoolExecutor
from time import sleep
from random import random
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait
from concurrent.futures import FIRST_COMPLETED

# custom task that will sleep for a variable amount of time
def task(name):
    # sleep for less than a second
    sleep(random())
    return name

# start the thread pool
executor = ThreadPoolExecutor(10)
# submit tasks and collect futures
futures = [executor.submit(task, i) for i in range(10)]
# wait until any task completes
done, not_done = wait(futures, return_when=FIRST_COMPLETED)
# get the result from the first task to complete
print(done.pop().result())
# shutdown without waiting
executor.shutdown(wait=False, cancel_futures=True)
```

Running the example will wait for any of the tasks to complete, then retrieve the result of the first completed task and shut down the thread pool.

Importantly, the tasks will continue to execute in the thread pool in the background and the main thread will not close until all tasks have completed.

```
1 9
```

Now that we have seen some common usage patterns for the **ThreadPoolExecutor**, let’s look at how we might customize the configuration of the thread pool.

How to Configure ThreadPoolExecutor

We can customize the configuration of the thread pool when constructing a **ThreadPoolExecutor** instance.

There are three aspects of the thread pool we may wish to customize for our application; they are the number of worker threads, the names of threads in the pool, and the initialization of each thread in the pool.

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

Configure the Number of Threads

The number of threads in the thread pool (https://superfastpython.com/threadpoolexecutor-number-of-threads/) can be configured by the “**max_workers**” argument.

It takes a positive integer and defaults to the number of CPUs in your system plus four.

- Total Number Worker Threads = (CPUs in Your System) + 4

For example, if you had 2 physical CPUs in your system and each CPU has hyperthreading (common in modern CPUs) then you would have 2 physical and 4 logical CPUs. Python would see 4 CPUs. The default number of worker threads on your system would then be (4 + 4) or 8.

If this number comes out to be more than 32 (e.g. 16 physical cores, 32 logical cores, plus four), the default will clip the upper bound to 32 threads.

It is common to have more threads than CPUs (physical or logical) in your system.

The reason for this is because threads are used for IO-bound tasks, not CPU-bound tasks. This means that threads are used for tasks that wait for relatively slow resources to respond, like hard drives, DVD drives, printers, and network connections, and much more. We will discuss the best application of threads in a later section.

Therefore, it is not uncommon to have tens, hundreds, and even thousands of threads in your application, depending on your specific needs. It is unusual to have more than one or a few thousand threads. If you require this many threads, then alternative solutions may be preferred, such as AsyncIO. We will discuss Threads vs. AsyncIO in a later section.

First, let's check how many threads are created for thread pools on your system.

Looking at the source code for the **ThreadPoolExecutor** (https://github.com/python/cpython/blob/3.10/Lib/concurrent/futures/thread.py), we can see that the number of worker threads chosen by default is stored in the **_max_workers** property, which we can access and report after a thread pool is created.

Note: "**_max_workers**" is a protected member and may change in the future.

The example below reports the number of default threads in a thread pool on your system.

```python
# SuperFastPython.com
# report the default number of worker threads on your system
from concurrent.futures import ThreadPoolExecutor
# create a thread pool with the default number of worker threads
pool = ThreadPoolExecutor()
# report the number of worker threads chosen by default
print(pool._max_workers)
```

Running the example reports the number of worker threads used by default on your system.

I have four physical CPU cores and eight logical cores; therefore the default is 8 + 4 or 12 threads.

```
12
```

How many worker threads are allocated by default on your system?
Let me know in the comments below.

We can specify the number of worker threads directly, and this is a good idea in most applications.

The example below demonstrates how to configure 500 worker threads.

```
# SuperFastPython.com
# configure and report the default number of worker threads
from concurrent.futures import ThreadPoolExecutor
# create a thread pool with a large number of worker threads
pool = ThreadPoolExecutor(500)
# report the number of worker threads
print(pool._max_workers)
```

Running the example configures the thread pool to use 500 threads and confirms that it will create 500 threads.

```
500
```

How Many Threads Should You Use?

If you have hundreds of tasks, you should probably set the number of threads to be equal to the number of tasks.

If you have thousands of tasks, you should probably cap the number of threads at hundreds or 1,000.

If your application is intended to be executed multiple times in the future, you can test different numbers of threads and compare overall execution time, then choose a number of threads that gives approximately the best performance. You may want to mock the task in these tests with a random sleep operation.

Configure Thread Names

Each thread in Python has a name.

The main thread has the name "**MainThread**". You can access the main thread via a call to the **main_thread()** function in the threading module and then access the name member. For example:

```
# access the name of the main thread
from threading import main_thread
# access the main thread
thread = main_thread()
# report the thread name
print(thread.name)
```

Running the example accesses the main thread and reports its name.

```
MainThread
```

Names are unique by default.

This can be helpful when debugging a program with multiple threads. Log messages can report the thread that is performing a specific step or a debugging can be used to trace a thread with a specific name.

When creating threads in the thread pool, each thread has the name "**ThreadPoolExecutor-%d_%d**" where the first **%d** indicates the thread pool number and the second **%d** indicates the thread number, both in the order that thread pools and threads are created.

We can see this if we access the threads directly inside the pool after allocating some work so that all threads are created.

We can enumerate all threads in a Python program (process) via the enumerate() function in the threading module, then report the name for each.

The example below creates a thread pool with the default number of threads, allocates work to the pool to ensure the threads are created, then reports the names of all threads in the program.

```python
# SuperFastPython.com
# report the default name of threads in the thread pool
import threading
from concurrent.futures import ThreadPoolExecutor

# a mock task that does nothing
def task(name):
    pass

# create a thread pool
executor = ThreadPoolExecutor()
# execute asks
executor.map(task, range(10))
# report all thread names
for thread in threading.enumerate():
    print(thread.name)
# shutdown the thread pool
executor.shutdown()
```

Running the example reports the names of all threads in the system, showing first the name of the main thread and the name of four threads in the pool.

In this case, only 4 threads were created as the tasks were executed so quickly. Recall that worker threads are used after they finish executing their tasks. This ability to reuse workers is a major benefit of using thread pools.

```
MainThread
ThreadPoolExecutor-0_0
ThreadPoolExecutor-0_1
ThreadPoolExecutor-0_2
ThreadPoolExecutor-0_3
```

The "**ThreadPoolExecutor-%d**" is a prefix for all threads in the thread pool and we can customize it with a name that may be meaningful in the application for the types of tasks executed by the pool.

The thread name prefix can be set (https://superfastpython.com/threadpoolexecutor-thread-names/) via the "**thread_name_prefix**" argument when constructing the thread pool.

The example below sets the prefix to be "**TaskPool**", which is prepended to the name of each thread created in the pool.

```
# SuperFastPython.com
# set a custom thread name prefix for all threads in the pool
import threading
from concurrent.futures import ThreadPoolExecutor

# a mock task that does nothing
def task(name):
    pass

# create a thread pool with a custom name prefix
executor = ThreadPoolExecutor(thread_name_prefix='TaskPool')
# execute asks
executor.map(task, range(10))
# report all thread names
for thread in threading.enumerate():
    print(thread.name)
# shutdown the thread pool
executor.shutdown()
```

Running the example reports the name of the main thread as before, but in this case, the names of threads in the thread pool with the custom thread name prefix.

```
MainThread
TaskPool_0
TaskPool_1
TaskPool_2
TaskPool_3
```

Configure the Initializer

Worker threads can call a function before they start processing tasks.

This is called an initializer function (https://superfastpython.com/threadpoolexecutor-initializer/) and can be specified via the "**initializer**" argument when creating a thread pool. If the initializer function takes arguments, they can be passed in via the "initargs" argument to the thread pool, which is a tuple of arguments to pass to the initializer function.

By default, there is no initializer function.

We might choose to set an initializer function for worker threads if we would like each thread to set up resources specific to the thread.

Examples might include a thread-specific log file or a thread-specific connection to a remote resource like a server or database. The resource would then be available to all tasks executed by the thread, rather than being created and discarded or opened and closed for each task.

These thread-specific resources can then be stored somewhere where the worker thread can reference, like a global variable, or in a thread local variable. Care must be taken to correctly close these resources once you are finished with the thread pool.

The example below will create a thread pool with two threads and use a custom initialization function. In this case, the function does nothing other than print the worker thread name. We then complete ten tasks with the thread pool.

```python
# SuperFastPython.com
# example of a custom worker thread initialization function
from time import sleep
from random import random
from threading import current_thread
from concurrent.futures import ThreadPoolExecutor

# function for initializing the worker thread
def initializer_worker():
    # get the unique name for this thread
    name = current_thread().name
    # store the unique worker name in a thread local variable
    print(f'Initializing worker thread {name}')

# a mock task that sleeps for a random amount of time less than one second
def task(identifier):
    sleep(random())
    # get the unique name
    return identifier

# create a thread pool
with ThreadPoolExecutor(max_workers=2, initializer=initializer_worker) as executor:
    # execute asks
    for result in executor.map(task, range(10)):
        print(result)
```

Running the example, we can see that the two threads are initialized before running any tasks, then all ten tasks are completed successfully.

```
Initializing worker thread ThreadPoolExecutor-0_0
Initializing worker thread ThreadPoolExecutor-0_1
0
1
2
3
4
5
6
7
8
9
```

Now that we are familiar with how to configure the thread pools, let's learn more about how to check and manipulate tasks via Future objects.

How to Use Future Objects in Detail

Future objects (https://docs.python.org/3/library/concurrent.futures.html#future-objects) are created when we call **submit()** to send tasks into the **ThreadPoolExecutor** to be executed asynchronously.

Future objects provide the capability to check the status of a task (e.g. is it running?) and to control the execution of the task (e.g. cancel).

In this section, we will look at some examples of checking and manipulating **Future** objects created by our thread pool.

Specifically, we will look at the following:

- How to Check the Status of Futures
- How to Get Results From Futures
- How to Cancel Futures
- How to Add a Callback to Futures
- How to Get Exceptions From Futures

First, let's take a closer look at the lifecycle of a future object (https://superfastpython.com/threadpoolexecutor-futures/).

Life-Cycle of a Future Object

A **Future** object is created when we call **submit()** for a task on a **ThreadPoolExecutor**.

While the task is executing, the **Future** object has the status “*running*“.

When the task completes, it has the status “*done*” and if the target function returns a value, it can be retrieved.

Before a task is running, it will be inserted into a queue of tasks for a worker thread to take and start running. In this “*pre-running*” state, the task can be cancelled and has the “*cancelled*” state. A task in the “*running*” state cannot be cancelled.

A “*cancelled*” task is always also in the “*done*” state.

While a task is running, it can raise an uncaught exception, causing the execution of the task to stop. The exception will be stored and can be retrieved directly or will be re-raised if the result is attempted to be retrieved.

The figure below summarizes the lifecycle of a **Future** object.

OVERVIEW OF THE LIFE-CYCLE OF A PYTHON FUTURE OBJECT.

Now that we are familiar with the lifecycle of a **Future** object, let's look at how we might use check and manipulate it.

How to Check the Status of Futures

There are two types of normal status of a **Future** object that we might want to check: running and done.

Each has its own function that returns a **True** if the **Future** object is in that state or **False** otherwise; for example:

- **running()**: Returns **True** if the task is currently running.
- **done()**: Returns **True** if the task has completed or was cancelled.

We can develop simple examples to demonstrate how to check the status of a **Future** object.

In this example, we can start a task and then check that it's running and not done, wait for it to complete, then check that it is done and not running.

```
# SuperFastPython.com
# check the status of a Future object for task executed by a thread pool
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# mock task that will sleep for a moment
def work():
    sleep(0.5)

# create a thread pool
with ThreadPoolExecutor() as executor:
    # start one thread
    future = executor.submit(work)
    # confirm that the task is running
    running = future.running()
    done = future.done()
    print(f'Future running={running}, done={done}')
    # wait for the task to complete
    wait([future])
    # confirm that the task is done
    running = future.running()
    done = future.done()
    print(f'Future running={running}, done={done}')
```

Running the example, we can see that immediately after the task is submitted, it is marked as running, and that after the task is completed, we can confirm that it is done.

```
Future running=True, done=False
Future running=False, done=True
```

How to Get Results From Futures

When a task is completed, we can retrieve the result from the task (https://superfastpython.com/threadpoolexecutor-get-results/) by calling the **result()** function on the **Future**.

This returns the result from the return function of the task we executed or **None** if the function did not return a value.

The function will block until the task completes and a result can be retrieved. If the task has already been completed, it will return a result immediately.

The example below demonstrates how to retrieve a result from a **Future** object.

```
# SuperFastPython.com
# get the result from a completed future task
from time import sleep
from concurrent.futures import ThreadPoolExecutor

# mock task that will sleep for a moment
def work():
    sleep(1)
    return "all done"

# create a thread pool
with ThreadPoolExecutor() as executor:
    # start one thread
    future = executor.submit(work)
    # get the result from the task, wait for task to complete
    result = future.result()
    print(f'Got Result: {result}')
```

Running the example submits the task then attempts to retrieve the result, blocking until the result is available, then reports the result that was received.

```
Got Result: all done
```

We can also set a timeout for how long we wish to wait for a result in seconds.

If the timeout elapses before we get a result, a **TimeoutError** is raised.

The example below demonstrates the timeout, showing how to give up waiting before the task has completed.

```python
# SuperFastPython.com
# set a timeout when getting results from a future
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import TimeoutError

# mock task that will sleep for a moment
def work():
    sleep(1)
    return "all done"

# create a thread pool
with ThreadPoolExecutor() as executor:
    # start one thread
    future = executor.submit(work)
    # get the result from the task, wait for task to complete
    try:
        result = future.result(timeout=0.5)
        print(f'Got Result: {result}')
    except TimeoutError:
        print('Gave up waiting for a result')
```

Running the example shows that we gave up waiting for a result after half a second.

```
Gave up waiting for a result
```

How to Cancel Futures

We can also cancel a task (https://superfastpython.com/threadpoolexecutor-cancel-task/) that has not yet started running.

Recall that when we put tasks into the pool with **submit()** or **map()** that the tasks are added to an internal queue of work from which worker threads can remove the tasks and execute them.

While a task is in the queue and before it has been started, we can cancel it by calling **cancel()** on the **Future** object associated with the task. The **cancel()** function will return **True** if the task was cancelled, **False** otherwise.

Let’s demonstrate this with a worked example.

We can create a thread pool with one thread, then start a long running task, then submit a second task, request that it is cancelled, then confirm that it was indeed cancelled.

```python
# SuperFastPython.com
# example of cancelling a task via it's future
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# mock task that will sleep for a moment
def work(sleep_time):
    sleep(sleep_time)

# create a thread pool
with ThreadPoolExecutor(1) as executor:
    # start a long running task
    future1 = executor.submit(work, 2)
    running = future1.running()
    print(f'First task running={running}')
    # start a second
    future2 = executor.submit(work, 0.1)
    running = future2.running()
    print(f'Second task running={running}')
    # cancel the second task
    was_cancelled = future2.cancel()
    print(f'Second task was cancelled: {was_cancelled}')
    # wait for the second task to finish, just in case
    wait([future2])
    # confirm it was cancelled
    running = future2.running()
    cancelled = future2.cancelled()
    done = future2.done()
    print(f'Second task running={running}, cancelled={cancelled}, done={done}')
    # wait for the long running task to finish
    wait([future1])
```

Running the example, we can see that the first task is started and is running normally.

The second task is scheduled and is not yet running because the thread pool is occupied with the first task. We then cancel the second task and confirm that it is indeed not running; it was cancelled and is done.

```
First task running=True
Second task running=False
Second task was cancelled: True
Second task running=False, cancelled=True, done=True
```

Cancel a Running Future

Now, let’s try to cancel a task that has already completed running.

```python
# SuperFastPython.com
# example of trying to cancel a running task via its future
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# mock task that will sleep for a moment
def work(sleep_time):
    sleep(sleep_time)

# create a thread pool
with ThreadPoolExecutor(1) as executor:
    # start a long running task
    future = executor.submit(work, 2)
    running = future.running()
    print(f'Task running={running}')
    # try to cancel the task
    was_cancelled = future.cancel()
    print(f'Task was cancelled: {was_cancelled}')
    # wait for the task to finish
    wait([future])
    # check if it was cancelled
    running = future.running()
    cancelled = future.cancelled()
    done = future.done()
    print(f'Task running={running}, cancelled={cancelled}, done={done}')
```

Running the example, we can see that the task was started as per normal.

We then tried to cancel the task, but this was not successful, as we expected, as the task was already running.

We then wait for the task to complete and then check its status. We can see that the task is no longer running and was not cancelled, as we expect, but is marked as not done.

This is surprising because the task was completed successfully. This is likely to indicate the situation where the cancel request was received but not acted upon.

```
Task running=True
Task was cancelled: False
Task running=False, cancelled=False, done=True
```

Cancel a Done Future

Consider what would happen if we tried to cancel a task that was already done.

We might expect that canceling a task that is already done has no effect, and this happens to be the case.

This can be demonstrated with a short example.

We start and run a task as per normal, then wait for it to complete and report its status. We then attempt to cancel the task.

```python
# SuperFastPython.com
# example of trying to cancel a done task via its future
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# mock task that will sleep for a moment
def work(sleep_time):
    sleep(sleep_time)

# create a thread pool
with ThreadPoolExecutor(1) as executor:
    # start a long running task
    future = executor.submit(work, 2)
    running = future.running()
    # wait for the task to finish
    wait([future])
    # check the status
    running = future.running()
    cancelled = future.cancelled()
    done = future.done()
    print(f'Task running={running}, cancelled={cancelled}, done={done}')
    # try to cancel the task
    was_cancelled = future.cancel()
    print(f'Task was cancelled: {was_cancelled}')
    # check if it was cancelled
    running = future.running()
    cancelled = future.cancelled()
    done = future.done()
    print(f'Task running={running}, cancelled={cancelled}, done={done}')
```

Running the example confirms that the task runs and is marked done, as per normal.

The attempt to cancel the task fails and checking the status after the attempt to cancel confirms that the task was not impacted by the attempt.

```
Task running=False, cancelled=False, done=True
Task was cancelled: False
Task running=False, cancelled=False, done=True
```

How to Add a Callback to Futures

We have already seen above how to add a callback to a **Future** (https://superfastpython.com/threadpoolexecutor-add-callback/). Nevertheless, let's look at some more examples for completeness including some edge cases.

We can register one or more callback functions on a **Future** object by calling the **add_done_callback()** function and specifying the name of the function to call.

The callbacks functions will be called with the **Future** object as an argument immediately after the completion of the task. If more than one callback function is registered, then they will be called in the order they were registered and any exceptions within each callback function will be caught, logged and ignored.

The callback will be called by the worker thread that executed the task.

The example below demonstrates how to add a callback function to a **Future** object.

```python
# SuperFastPython.com
# add a callback option to a future object
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# callback function to call when a task is completed
def custom_callback(future):
    print('Custom callback was called')

# mock task that will sleep for a moment
def work():
    sleep(1)
    print('Task is done')

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute the task
    future = executor.submit(work)
    # add the custom callback
    future.add_done_callback(custom_callback)
    # wait for the task to complete
    wait([future])
```

Running the example, we can see that the task is completed first, then the callback is executed as we expected.

```
Task is done
Custom callback was called
```

Common Error When Using Future Callbacks

A common error is to forget to add the **Future** object as an argument to the custom callback.

For example:

```python
# callback function to call when a task is completed
def custom_callback():
    print('Custom callback was called')
```

If you register this function and try to run the code, you will get a **TypeError** as follows:

```
Task is done
exception calling callback for <Future at 0x104482b20 state=finished returned NoneType>
...
TypeError: custom_callback() takes 0 positional arguments but 1 was given
```

The message in the **TypeError** makes it clear how to fix the issue: add a single argument to the function for the future object, even if you don't intend on using it in your callback.

Callbacks Execute When Cancelling a Future

We can also see the effect of callbacks on **Future** objects for tasks that are cancelled.

The effect does not appear to be documented in the API, but we might expect for the callback to always be executed, whether the task is run normally or whether it is cancelled. And this happens to be the case.

The example below demonstrates this.

First, a thread pool is created with a single thread. A long running task is issued that occupies the entire pool, then we send in a second task, add a callback to the second task, cancel it, and wait for all tasks to finish.

```python
# SuperFastPython.com
# example of a callback for a cancelled task via the future object
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# callback function to call when a task is completed
def custom_callback(future):
    print('Custom callback was called')

# mock task that will sleep for a moment
def work(sleep_time):
    sleep(sleep_time)

# create a thread pool
with ThreadPoolExecutor(1) as executor:
    # start a long running task
    future1 = executor.submit(work, 2)
    running = future1.running()
    print(f'First task running={running}')
    # start a second
    future2 = executor.submit(work, 0.1)
    running = future2.running()
    print(f'Second task running={running}')
    # add the custom callback
    future2.add_done_callback(custom_callback)
    # cancel the second task
    was_cancelled = future2.cancel()
    print(f'Second task was cancelled: {was_cancelled}')
    # explicitly wait for all tasks to complete
    wait([future1, future2])
```

Running the example, we can see that the first task is started as we expect.

The second task is scheduled but does not get a chance to run before we cancel it. The callback is run immediately after we cancel the task, then we report in the main thread that indeed the task was cancelled correctly.

```
First task running=True
Second task running=False
Custom callback was called
Second task was cancelled: True
```

How to Get Exceptions From Futures

A task may raise an exception during execution.

If we can anticipate the exception, we can wrap parts of our task function in a try-except block and handle the exception within the task.

If an unexpected exception occurs within our task, the task will stop executing.

We cannot know based on the task status whether an exception was raised, but we can check for an exception directly.

We can then access the exception via the **exception()** function. Alternately, the exception will be re-raised when calling the **result()** function when trying to get a result.

We can demonstrate this with an example.

The example below will raise a **ValueError** within the task that will not be caught but instead will be caught by the thread pool for us to access later.

```
# SuperFastPython.com
# example of handling an exception raised within a task
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# mock task that will sleep for a moment
def work():
    sleep(1)
    raise Exception('This is Fake!')
    return "never gets here"

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute our task
    future = executor.submit(work)
    # wait for the task to complete
    wait([future])
    # check the status of the task after it has completed
    running = future.running()
    cancelled = future.cancelled()
    done = future.done()
    print(f'Task running={running}, cancelled={cancelled}, done={done}')
    # get the exception
    exception = future.exception()
    print(f'Exception={exception}')
    # get the result from the task
    try:
        result = future.result()
    except Exception:
        print('Unable to get the result')
```

Running the example starts the task normally, which sleeps for one second.

The task then raises an exception that is caught by the thread pool. The thread pool stores the exception and the task is completed.

We can see that after the task is completed, it is marked as not running, not cancelled, and done.

We then access the exception from the task, which matches the exception we intentionally raise.

Attempting to access the result via the result() function fails and we catch the same exception raised in the task.

```
Task running=False, cancelled=False, done=True
Exception=This is Fake!
Unable to get the result
```

Callbacks Are Still Called if a Task Raises an Exception

We might wonder if we register a callback function with a **Future** whether it will still execute if the task raises an exception.

As we might expect, the callback is executed even if the task raises an exception.

We can test this by updating the previous example to register a callback function before the task fails with an exception.

```python
# SuperFastPython.com
# example of handling an exception raised within a task that has a callback
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# callback function to call when a task is completed
def custom_callback(future):
    print('Custom callback was called')

# mock task that will sleep for a moment
def work():
    sleep(1)
    raise Exception('This is Fake!')
    return "never gets here"

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute our task
    future = executor.submit(work)
    # add the custom callback
    future.add_done_callback(custom_callback)
    # wait for the task to complete
    wait([future])
    # check the status of the task after it has completed
    running = future.running()
    cancelled = future.cancelled()
    done = future.done()
    print(f'Task running={running}, cancelled={cancelled}, done={done}')
    # get the exception
    exception = future.exception()
    print(f'Exception={exception}')
    # get the result from the task
    try:
        result = future.result()
    except Exception:
        print('Unable to get the result')
```

Running the example starts the task as before, but this time registers a callback function.

When the task fails with an exception, the callback is called immediately. The main thread then reports the status of the failed task and the details of the exception.

```
Custom callback was called
Task running=False, cancelled=False, done=True
Exception=This is Fake!
Unable to get the result
```

When to Use the ThreadPoolExecutor

The **ThreadPoolExecutor** is powerful and flexible, although it is not suited for all situations where you need to run a background task.

In this section, we will look at some general cases where it is a good fit, and where it isn't, then we'll look at broad classes of tasks and why they are or are not appropriate for the **ThreadPoolExecutor**.

Use ThreadPoolExecutor 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, e.g. homogeneous tasks.
- You need to apply the same function to each object in a collection in a for-loop.

Thread 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 not a rule, just a gentle suggestion.

Use Multiple ThreadPoolExecutor When…

- You need to perform groups of different types of tasks; one thread pool could be used for each task type.
- You need to perform a pipeline of tasks or operations; one thread pool can be used for each step.

Thread pools can operate on tasks of different types (e.g. heterogeneous tasks), although it may make the organization of your program and debugging easy if a separate thread pool is responsible for each task type. This is not a rule, just a gentle suggestion.

Don't Use ThreadPoolExecutor When…

- You have a single task; consider using the **Thread** class with the target argument.
- You have long-running tasks, such as monitoring or scheduling; consider extending the **Thread** class.
- Your task functions require state; consider extending the **Thread** class.
- Your tasks require coordination; consider using a **Thread** and patterns like a **Barrier** or **Semaphore**.

- Your tasks require synchronization; consider using a **Thread** and **Locks**.
- You require a thread trigger on an event; consider using the **Thread** class.

The sweet spot for thread pools is in dispatching many similar tasks, the results of which may be used later in the program. Tasks that don't fit neatly into this summary are probably not a good fit for thread pools. This is not a rule, just a gentle suggestion.

Do you know any other good or bad cases where using a ThreadPoolExecutor?
Let me know in the comments below.

Use Threads for IO-Bound Tasks

You should use threads for IO-bound tasks.

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

The operations involve input and output (IO), and the speed of these operations is bound by the device, hard drive, or network connection. This is why these tasks are referred to as IO-bound.

CPUs are really fast. Modern CPUs, like a 4GHz, can execute 4 billion instructions per second, and you likely have more than one CPU in your system.

Doing IO is very slow compared to the speed of CPUs.

Interacting with devices, reading and writing files and socket connections involves calling instructions in your operating system (the kernel), which will wait for the operation to complete. If this operation is the main focus for your CPU, such as executing in the main thread of your Python program, then your CPU is going to wait many milliseconds or even many seconds doing nothing.

That is potentially billions of operations prevented from executing.

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

There's more to it under the covers, but this is the gist.

Therefore, the tasks we execute with a **ThreadPoolExecutor** should be tasks that involve IO operations.

Examples include:

- Reading or writing a file from the hard drive.
- Reading or writing to standard output, input or error (stdin, stdout, stderr).
- Printing a document.
- Downloading or uploading a file.
- Querying a server.
- Querying a database.
- Taking a photo or recording a video.
- And so much more.

If your task is not IO-bound, perhaps threads and using a thread pool is not appropriate.

Don't Use Threads for CPU-Bound Tasks

You should probably not use threads 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:

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

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

Using threads and thread pools via the ThreadPoolExecutor class in Python is probably not a path toward achieving this end.

This is because of a technical reason behind the way that the Python interpreter was implemented. The implementation prevents two Python operations executing at the same time inside the interpreter and it does this with a master lock that only one thread can hold at a time. This is called the global interpreter lock, or GIL.

The GIL is not evil and is not frustrating; it is a design decision in the python interpreter that we must be aware of and consider in the design of our applications.

I said that you "*probably*" should not use threads for CPU-bound tasks.

You can and are free to do so, but your code will not benefit from concurrency because of the GIL. It will likely perform worse because of the additional overhead of context switching (the CPU jumping from one thread of execution to another) introduced by using threads.

Additionally, the GIL is a design decision that affects the reference implementation of Python, which you download from Python.org. If you use a different implementation of the Python interpreter (such as PyPy, IronPython, Jython, and perhaps others), then you may not be subject to the GIL and can use threads for CPU bound tasks directly.

Python provides a multiprocessing module (https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing) for multi-core task execution as well as a sibling of the **ThreadPoolExecutor** that uses processes called the **ProcessPoolExecutor** (https://docs.python.org/3/library/concurrent.futures.html#concurrent.futures.ProcessPoolExecutor) that can be used for concurrency of CPU-bound tasks.

ThreadPoolExecutor Exception Handling

Exception handling (https://superfastpython.com/threadpoolexecutor-exception-handling/) is an important consideration when using threads.

Code will raise an exception when something unexpected happens and the exception should be dealt with by your application explicitly, even if it means logging it and moving on.

Python threads are well suited for use with IO-bound tasks, and operations within these tasks often raise exceptions, such as if a server cannot be reached, if the network goes down, if a file cannot be found, and so on.

There are three points you may need to consider exception handling when using the **ThreadPoolExecutor**; they are:

- Exception Handling During Thread Initialization
- Exception Handling During Task Execution
- Exception Handling During Task Completion Callbacks

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

Exception Handling During Thread Initialization

You can specify a custom initialization function when configuring your **ThreadPoolExecutor**.

This can be set via the "**initializer**" argument to specify the function name and "**initargs**" to specify a tuple of arguments to the function.

Each thread started by the thread pool will call your initialization function before starting the thread.

If your initialization function raises an exception, it will break your thread pool.

All current tasks and any future tasks executed by the thread pool will not run and will raise a **BrokenThreadPool** exception.

We can demonstrate this with an example of a contrived initializer function that raises an exception.

```python
# SuperFastPython.com
# example of an exception in a thread pool initializer function
from time import sleep
from random import random
from threading import current_thread
from concurrent.futures import ThreadPoolExecutor

# function for initializing the worker thread
def initializer_worker():
    # raise an exception
    raise Exception('Something bad happened!')

# a mock task that sleeps for a random amount of time less than one second
def task(identifier):
    sleep(random())
    # get the unique name
    return identifier

# create a thread pool
with ThreadPoolExecutor(max_workers=2, initializer=initializer_worker) as executor:
    # execute tasks
    for result in executor.map(task, range(10)):
        print(result)
```

Running the example fails with an exception, as we expected.

The thread pool is created as per normal, but as soon as we try to execute tasks, new worker threads are created, the custom worker thread initialization function is called, and raises an exception.

Multiple threads attempted to start, and in turn, multiple threads failed with an Exception. Finally, the thread pool itself logged a message that the pool is broken and cannot be used any longer.

```
Exception in initializer:
Traceback (most recent call last):
...
    raise Exception('Something bad happened!')
Exception: Something bad happened!
Exception in initializer:
Traceback (most recent call last):
...
    raise Exception('Something bad happened!')
Exception: Something bad happened!
Traceback (most recent call last):
...
concurrent.futures.thread.BrokenThreadPool: A thread initializer failed, the thread pool is not usable any
```

This highlights that if you use a custom initializer function, you must carefully consider the exceptions that may be raised and perhaps handle them, otherwise risk all tasks that depend on the thread pool.

Exception Handling During Task Execution

An exception may occur while executing your task.

This will cause the task to stop executing, but will not break the thread pool. Instead, the exception will be caught by the thread pool and will be available via the **Future** object associated with the task via the **exception()** function.

Alternately, the exception will be re-raised if you call **result()** in the **Future** in order to get the result. This will impact both calls to **submit()** and **map()** when adding tasks to the thread pool.

It means that you have two options for handling exceptions in tasks; they are:

- 1. Handle exceptions within the task function.
- 2. Handle exceptions when getting results from tasks.

Handle Exception Within the Task

Handling the exception within the task means that you need some mechanism to let the recipient of the result know that something unexpected happened.

This could be via the return value from the function, e.g. **None**.

Alternatively, you can re-raise an exception and have the recipient handle it directly. A third option might be to use some broader state or global state, perhaps passed by reference into the call to the function.

The example below defines a work task that will raise an exception, but will catch the exception and return a result indicating a failure case.

```
# SuperFastPython.com
# example of handling an exception raise within a task
from time import sleep
from concurrent.futures import ThreadPoolExecutor

# mock task that will sleep for a moment
def work():
    sleep(1)
    try:
        raise Exception('Something bad happened!')
    except Exception:
        return 'Unable to get the result'
    return "never gets here"

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute our task
    future = executor.submit(work)
    # get the result from the task
    result = future.result()
    print(result)
```

Running the example starts the thread pool as per normal, issues the task, then blocks waiting for the result.

The task raises an exception and the result received is an error message.

This approach is reasonably clean for the recipient code and would be appropriate for tasks issued by both **submit()** and **map()**. It may require special handling of a custom return value for the failure case.

```
Unable to get the result
```

Handle Exception by the Recipient of the Task Result

An alternative to handling the exception in the task is to leave the responsibility to the recipient of the result.

This may feel like a more natural solution, as it matches the synchronous version of the same operation, e.g. if we were performing the function call in a for-loop.

It means that the recipient must be aware of the types of errors that the task may raise and handle them explicitly.

The example below defines a simple task that raisees an **Exception**, which is then handled by the recipient when attempting to get the result from the function call.

```python
# SuperFastPython.com
# example of handling an exception raised within a task
from time import sleep
from concurrent.futures import ThreadPoolExecutor

# mock task that will sleep for a moment
def work():
    sleep(1)
    raise Exception('Something bad happened!')
    return "never gets here"

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute our task
    future = executor.submit(work)
    # get the result from the task
    try:
        result = future.result()
    except Exception:
        print('Unable to get the result')
```

Running the example creates the thread pool and submits the work as per normal. The task fails with an error, the thread pool catches the exception, stores it, then re-raises it when we call the **result()** function in the **Future**.

The recipient of the result accepts the exception and catches it, reporting a failure case.

```
Unable to get the result
```

We can also check for the exception directly via a call to the **exception()** function on the **Future** object. This function blocks until an exception occurs and takes a timeout, just like a call to **result()**.

If an exception never occurs and the task is cancelled or completes successfully, then **exception()** will return a value of **None**.

We can demonstrate the explicit checking for an exceptional case in the task in the example below.

```python
# SuperFastPython.com
# example of handling an exception raised within a task
from time import sleep
from concurrent.futures import ThreadPoolExecutor

# mock task that will sleep for a moment
def work():
    sleep(1)
    raise Exception('Something bad happened!')
    return "never gets here"

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute our task
    future = executor.submit(work)
    # get the result from the task
    exception = future.exception()
    # handle exceptional case
    if exception:
         print(exception)
    else:
        result = future.result()
        print(result)
```

Running the example creates and submits the work per normal.

The recipient checks for the exceptional case, which blocks until an exception is raised or the task is completed. An exception is received and is handled by reporting it.

```
Something bad happened!
```

We cannot check the exception() function of the **Future** object for each task, as **map()** does not provide access to **Future** objects.

Worse still, the approach of handling the exception in the recipient cannot be used when using **map()** to submit tasks, unless you wrap the entire iteration.

We can demonstrate this by submitting one task with **map()** that happens to raise an Exception.

The complete example is listed below.

```python
# SuperFastPython.com
# example of handling an exception raised within a task
from time import sleep
from concurrent.futures import ThreadPoolExecutor

# mock task that will sleep for a moment
def work(value):
    sleep(1)
    raise Exception('Something bad happened!')
    return f'Never gets here {value}'

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute our task
    for result in executor.map(work, [1]):
        print(result)
```

Running the example submits the single task (a bad use for **map()**) and waits for the first result.

The task raises an exception and the main thread exits, as we expected.

```
Traceback (most recent call last):
...
    raise Exception('Something bad happened!')
Exception: Something bad happened!
```

This highlights that if **map()** is used to submit tasks to the thread pool, then the tasks should handle their own exceptions or be simple enough that exceptions are not expected.

Exception Handling in Callbacks

A final case we must consider for exception handling when using the **ThreadPoolExecutor** is in callback functions.

When issuing tasks to the thread pool with a call to **submit()**, we receive a **Future** object in return on which we can register callback functions to call when the task completes via the **add_done_callback()** function.

This allows one or more callback functions to be registered that will be executed in the order in which they are registered.

These callbacks are always called, even if the task is cancelled or fails itself with an exception.

A callback can fail with an exception and it will not impact other callback functions that have been registered or the task.

The exception is caught by the thread pool, logged as an exception type message, and the procedure moves on. In a sense, callbacks are able to fail silently.

We can demonstrate this with a worked example with multiple callback functions, the first of which will raise an exception.

```python
# SuperFastPython.com
# add callbacks to a future, one of which raises an exception
from time import sleep
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# callback function to call when a task is completed
def custom_callback1(future):
    raise Exception('Something bad happened!')
    # never gets here
    print('Callback 1 called.')

# callback function to call when a task is completed
def custom_callback2(future):
    print('Callback 2 called.')

# mock task that will sleep for a moment
def work():
    sleep(1)
    return 'Task is done'

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute the task
    future = executor.submit(work)
    # add the custom callbacks
    future.add_done_callback(custom_callback1)
    future.add_done_callback(custom_callback2)
    # wait for the task to complete and get the result
    result = future.result()
    # wait for callbacks to finish
    sleep(0.1)
    print(result)
```

Running the example starts the thread pool as per normal and executes the task.

When the task completes, the first callback is called, which fails with an exception. The exception is logged and reported on the console (the default behavior for logging).

The thread pool is not broken and carries on.

The second callback is called successfully, and finally, the main thread gets the result of the task.

```
exception calling callback for <Future at 0x101d76730 state=finished returned str>
Traceback (most recent call last):
...
    raise Exception('Something bad happened!')
Exception: Something bad happened!
Callback 2 called.
Task is done
```

This highlights that if callbacks are expected to raise an exception, that it must be handled explicitly and checked for if you wish to have the failure impact the task itself.

How Does ThreadPoolExecutor Work Internally

It is important to pause for a moment and look at how the **ThreadPoolExecutor** works internally (https://superfastpython.com/how-does-threadpoolexecutor-work/).

The internal workings of the class impact how we use the thread pool and the behavior we can expect, specifically around cancelling tasks.

Without this knowledge, some of the behavior of the thread pool may appear confusing from the outside.

You can see the source code for the **ThreadPoolExecutor** and the base class here:

- cpython/Lib/concurrent/futures/thread.py (https://github.com/python/cpython/blob/3.10/Lib/concurrent/futures/thread.py)
- cpython/Lib/concurrent/futures/_base.py (https://github.com/python/cpython/blob/3.10/Lib/concurrent/futures/_base.py)

There is a lot we could learn about how the thread pool works internally, but we will limit ourselves to the most critical aspects.

There are two aspects that you need to consider about the internal operation of the ThreadPoolExecutor class: how tasks are sent into the pool and how worker threads are created.

Tasks Are Added to an Internal Queue

Tasks are sent into the thread pool by adding them to an internal queue.

Recall that a queue is a data structure where items are added to one end and retrieved from the other in a first-in, first-out (FIFO) manner by default.

The queue is a **SimpleQueue** object (https://docs.python.org/3/library/queue.html#queue.SimpleQueue), which is a thread-safe **Queue** implementation. This means we can add work to the pool from any thread and the **Queue** of work will not become corrupt from concurrent **put()** and **get()** operations.

The use of a task queue explains the distinction between tasks that have been added or scheduled but are not yet running, and that these tasks can be cancelled.

Recall that the thread pool has a fixed number of worker threads. The number of tasks on the queue may exceed the current number of threads, or the current number of available threads. In which case, tasks may sit in a scheduled state for some time, allowing them to be canceled either directly or en masse when shutting down the pool.

A task is wrapped in an internal object called a **_WorkItem**. This captures the details such as the function to call, the arguments, the associated **Future** object, and handling of exceptions if they occur during task execution.

This explains how an exception within a task does not bring down the entire thread pool, but can be checked for and accessed after the task has completed.

When **_WorkItem** object is retrieved from the queue by a worker thread, it will check if the task has been cancelled before it is executed. If so, it will return immediately and not execute the content of the task.

This explains internally how cancellation is implemented by the thread pool and why we cannot cancel a running task.

Worker Threads Are Created as Needed

Worker threads are not created when the thread pool is created.

Instead, worker threads are created on demand or just-in-time.

Each time a task is added to the internal queue, the thread pool will check if the number of active threads is less than the upper limit of threads supported by the thread pool. If so, an additional thread is created to handle the new work.

Once a thread has completed a task, it will wait on the queue for new work to arrive. As new work arrives, all threads waiting on the queue will be notified and one will consume the unit of work and start executing it.

These two points show how the pool will only ever create new threads until the limit is reached and how threads will be reused, waiting for new tasks without consuming computational resources.

It also shows that the thread pool will not release threads after a fixed number of units of work. Perhaps this would be a nice addition to the API in the future.

Now that we understand how work is injected into the thread pool and how the pool manages threads, let’s look at some best practices to consider when using the ThreadPoolExecutor.

ThreadPoolExecutor Best Practices

Now that we know how the **ThreadPoolExecutor** works and how to use it, let’s review some best practices to consider (https://superfastpython.com/threadpoolexecutor-best-practices/) when bringing thread pools into our Python programs.

To keep things simple, there are five best practices; they are:

- 1. Use the Context Manager
- 2. Use map() for Asynchronous For-Loops
- 3. Use submit() with as_completed()
- 4. Use Independent Functions as Tasks
- 5. Use for IO-Bound Tasks (probably)

Use the Context Manager

Use the context manager when using thread pools and handle all task dispatching to the thread pool and processing results within the manager.

For example:

```
...
# create a thread pool via the context manager
with ThreadPoolExecutor(10) as executor:
    # ...
```

Remember to configure your thread pool when creating it in the context manager, specifically by setting the number of threads to use in the pool.

Using the context manager avoids the situation where you have explicitly instantiated the thread pool and forget to shut it down manually by calling **shutdown()**.

It is also less code and better grouped than managing instantiation and shutdown manually; for example:

```
...
# create a thread pool manually
executor = ThreadPoolExecutor(10)
# ...
executor.shutdown()
```

Don't use the context manager when you need to dispatch tasks and get results over a broader context (e.g. multiple functions) and/or when you have more control over the shutdown of the pool.

Use map() for Asynchronous For-Loops

If you have a for-loop that applies a function to each item in a list, then use the **map()** function to dispatch the tasks asynchronously.

For example, you may have a for-loop over a list that calls **myfunc()** for each item:

```
...
# apply a function to each item in an iterable
for item in mylist:
    result = myfunc(item)
    # do something...
```

Or, you may already be using the built-in map function:

```
...
# apply a function to each item in an iterable
for result in map(myfinc, mylist):
    # do something...
```

Both of these cases can be made asynchronous using the **map()** function on the thread pool.

```
...
# apply a function to each item in a iterable asynchronously
for result in executor.map(myfunc, mylist):
    # do something...
```

Do not use the **map()** function if your target task function has side effects.

Do not use the **map()** function if your target task function has no arguments or more than one argument.

Do not use the **map()** function if you need control over exception handling for each task, or if you would like to get results to tasks in the order that tasks are completed.

Use submit() with as_completed()

If you would like to process results in the order that tasks are completed, rather than the order that tasks are submitted, then use **submit()** and **as_completed()**.

The **submit()** function is on the thread pool and is used to push tasks into the pool for execution and returns immediately with a Future object for the task. The **as_completed()** function is a module method that will take an iterable of **Future** objects, like a list, and will return **Future** objects as the tasks are completed.

For example:

```
...
# submit all tasks and get future objects
futures = [executor.submit(myfunc, item) for item in mylist]
# process results from tasks in order of task completion
for future in as_completed(futures):
    # get the result
    result = future.result()
    # do something...
```

Do not use the **submit()** and **as_completed()** combination if you need to process the results in the order that the tasks were submitted to the thread pool.

Do not use the **submit()** and **as_completed()** combination if you need results from all tasks to continue; you may be better off using the **wait()** module function.

Do not use the **submit()** and **as_completed()** combination for a simple asynchronous for-loop; you may be better off using **map()**.

Use Independent Functions as Tasks

Use the **ThreadPoolExecutor** if your tasks are independent.

This means that each task is not dependent on other tasks that could execute at the same time. It also may mean tasks that are not dependent on any data other than data provided via function arguments to the task.

The **ThreadPoolExecutor** is ideal for tasks that do not change any data, e.g. have no side effects, so-called pure functions (https://en.wikipedia.org/wiki/Pure_function).

Thread pools can be organized into data flows and pipelines for linear dependence between tasks, perhaps with one thread pool per task type.

The thread pool is not designed for tasks that require coordination; you should consider using the **Thread** class and coordination patterns like the **Barrier** and **Semaphore**.

Thread pools are not designed for tasks that require synchronization; you should consider using the **Thread** class and locking patterns like **Lock** and **RLock**.

Use for IO-Bound Tasks (probably)

Use **ThreadPoolExecutor** for IO-bound tasks only.

These are tasks that may involve interacting with an external device such as a peripheral (e.g. a camera or a printer), a storage device (e.g. a storage device or a hard drive), or another computer (e.g. socket communication).

Threads and thread pools like the **ThreadPoolExecutor** are not probably appropriate for CPU-bound tasks, like computation on data in memory.

This is because of design decisions within the Python interpreter that makes use of a master lock called the Global Interpreter Lock (GIL) that prevents more than one Python instruction executing at the same time.

This design decision was made within the reference implementation of the Python interpreter (from Python.org), but may not impact other interpreters (such as PyPy, Iron Python, and Jython).

Common Errors When Using ThreadPoolExecutor

There are a number of common errors when using the **ThreadPoolExecutor** (https://superfastpython.com/threadpoolexecutor-common-errors/).

These errors are typically made because of bugs introduced by copy-and-pasting code, or from a slight misunderstanding in how the **ThreadPoolExecutor** works.

We will take a closer look at some of the more common errors made when using the **ThreadPoolExecutor**.

Do you have an error using the ThreadPoolExecutor?
Let me know in the comments so I can recommend a fix and add the case to this section.

Using a Function Call in submit()

A common error is to call your function when using the **submit()** function.

For example:

```
...
# submit the task
future = executor.submit(task())
```

A complete example with this error is listed below.

```python
# SuperFastPython.com
# example of calling submit with a function call
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():
    # sleep for less than a second
    sleep(random())
    return 'all done'

# start the thread pool
with ThreadPoolExecutor() as executor:
    # submit the task
    future = executor.submit(task())
    # get the result
    result = future.result()
    print(result)
```

Running this example will fail with an error.

```
Traceback (most recent call last):
...
    result = self.fn(*self.args, **self.kwargs)
TypeError: 'str' object is not callable
```

You can fix the error by updating the call to **submit()** to take the name of your function and any arguments, instead of calling the function in the call to submit.

For example:

```python
...
# submit the task
future = executor.submit(task)
```

Using a Function Call in map()

A common error is to call your function when using the **map()** function.

For example:

```python
...
# submit all tasks
for result in executor.map(task(), range(5)):
    print(result)
```

A complete example with this error is listed below.

```python
# SuperFastPython.com
# example of calling map with a function call
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(value):
    # sleep for less than a second
    sleep(random())
    return value

# start the thread pool
with ThreadPoolExecutor() as executor:
    # submit all tasks
    for result in executor.map(task(), range(5)):
        print(result)
```

Running the example results in a **TypeError**

```
Traceback (most recent call last):
...
TypeError: task() missing 1 required positional argument: 'value'
```

This error can be fixed by changing the call to **map()** to pass the name of the target task function instead of a call to the function.

```python
...
# submit all tasks
for result in executor.map(task, range(5)):
    print(result)
```

Incorrect Function Signature for map()

Another common error when using **map()** is to provide no second argument to function, e.g. the iterable.

For example:

```python
...
# submit all tasks
for result in executor.map(task):
    print(result)
```

A complete example with this error is listed below.

```python
# SuperFastPython.com
# example of calling map without an iterable
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(value):
    # sleep for less than a second
    sleep(random())
    return value

# start the thread pool
with ThreadPoolExecutor() as executor:
    # submit all tasks
    for result in executor.map(task):
        print(result)
```

Running the example does not issue any tasks to the thread pool as there was no iterable for the **map()** function to iterate over.

In this case, no output is displayed.

The fix involves providing an iterable in the call to **map()** along with your function name.

```python
...
# submit all tasks
for result in executor.map(task, range(5)):
    print(result)
```

Incorrect Function Signature for Future Callbacks

Another common error is to forget to include the **Future** in the signature for the callback function registered with a **Future** object.

For example:

```python
...
# callback function to call when a task is completed
def custom_callback():
    print('Custom callback was called')
```

A complete example with this error is listed below.

```python
# SuperFastPython.com
# example of the wrong signature on the callback function
from time import sleep
from concurrent.futures import ThreadPoolExecutor

# callback function to call when a task is completed
def custom_callback():
    print('Custom callback was called')

# mock task that will sleep for a moment
def work():
    sleep(1)
    return 'Task is done'

# create a thread pool
with ThreadPoolExecutor() as executor:
    # execute the task
    future = executor.submit(work)
    # add the custom callback
    future.add_done_callback(custom_callback)
    # get the result
    result = future.result()
    print(result)
```

Running this example will result in an error when the callback is called by the thread pool.

```
Task is done
exception calling callback for <Future at 0x10a05f190 state=finished returned str>
Traceback (most recent call last):
...
TypeError: custom_callback() takes 0 positional arguments but 1 was given
```

Fixing this error involves updating the signature of your callback function to include the **Future** object.

```python
...
# callback function to call when a task is completed
def custom_callback(future):
    print('Custom callback was called')
```

Common Questions When Using the ThreadPoolExecutor

This section answers common questions asked by developers when using the **ThreadPoolExecutor**.

Do you have a question about the ThreadPoolExecutor?

Ask your question in the comments below and I will do my best to answer it and perhaps add it to this list of questions.

How Do You Stop a Running Task?

A task in the **ThreadPoolExecutor** can be cancelled before it has started running.

In this case, the task must have been sent into the pool by calling **submit()**, which returns a **Future** object. You can then call the **cancel()** function in the future.

If the task is already running it cannot be canceled, stopped, or terminated by the thread pool.

Instead, you must add this functionality to your task.

One approach might be to use a thread-safe flag, like an **threading.Event** that, if set, will indicate that all tasks must stop running as soon as they can (https://superfastpython.com/threadpoolexecutor-stop-tasks/). You can then update your target task function or functions to check the state of this flag frequently.

It may require that you change the structure of your task.

For example, if your task reads data from a file or a socket, you may need to change the read operation to be performed in blocks of data in a loop so that each iteration of the loop you can check the status of the flag.

The example below provides a template you can use for adding an event flag to your target task function to check for a stop condition to shut down all currently running tasks.

The example below demonstrates this with a worked example.

```python
# SuperFastPython.com
# example of stopping running tasks using an event
from time import sleep
from threading import Event
from concurrent.futures import ThreadPoolExecutor

# mock target task function
def work(event):
    # pretend read data for a long time
    for _ in range(100):
        # pretend to read some data
        sleep(1)
        # check the status of the flag
        if event.is_set():
            # shut down this task now
            print("Not done, asked to stop")
            return
    return "All done!"

# create an event to shut down all running tasks
event = Event()
# create a thread pool
executor = ThreadPoolExecutor(5)
# execute all of our tasks
futures = [executor.submit(work, event) for _ in range(50)]
# wait a moment
print('Tasks are running...')
sleep(2)
# cancel all scheduled tasks
print('Cancelling all scheduled tasks...')
for future in futures:
    future.cancel()
# stop all currently running tasks
print('Trigger all running tasks to stop...')
event.set()
# shutdown the thread pool and wait for all tasks to complete
print('Shutting down...')
executor.shutdown()
```

Running the example first creates a thread pool with 5 worker threads and schedules 50 tasks.

An event object is created and passed to each task where it is checked each iteration to see if it has been set and if so to bail out of the task.

The first 5 tasks start executing for a few seconds, then we decide to shut everything down.

First, we cancel all scheduled tasks that are not yet running so that if they make it off the queue into a worker thread, they will not start running.

We then mark set the event to trigger all running tasks to stop.

The thread pool is then shut down and we wait for all running threads to complete their execution.

The five running threads check the status of the event in their next loop iteration and bail out, printing a message.

```
Tasks are running...
Cancelling all scheduled tasks...
Trigger all running tasks to stop...
Shutting down...
Not done, asked to stop
Not done, asked to stop
Not done, asked to stop
Not done, asked to stop
Not done, asked to stop
```

How Do You Wait for All Tasks to Complete?

There are a few ways to wait for all tasks to complete in a **ThreadPoolExecutor** (https://superfastpython.com/threadpoolexecutor-wait-all-tasks/).

Firstly, if you have a **Future** object for all tasks in the thread pool because you called **submit()**, then you can provide the collection of tasks to the **wait()** module function. By default, this function will return when all provided **Future** objects have completed.

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

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()
# all tasks are complete
```

Another approach is to shut down the thread pool. We can set "**cancel_futures**" to **True**, which will cancel all scheduled tasks and wait for all currently running tasks to complete.

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

You can also shut down 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.

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

How Do You Dynamically Change the Number of Threads?

You cannot dynamically increase or decrease the number of threads in a **ThreadPoolExecutor**.

The number of threads is fixed when the **ThreadPoolExecutor** is configured in the call to the object constructor. For example:

```
...
# configure a thread pool
executor = ThreadPoolExecutor(20)
```

How Do You Log From a Task?

Your target task functions are executed by worker threads in the thread pool and you may be concerned whether logging from those task functions is thread safe.

That is, will the log become corrupt if two threads attempt at the same time. The answer is no, the log will not become corrupt.

The Python logging functionality is thread-safe by default.

For example, see this quote from the logging module API documentation:

> *The logging module is intended to be thread-safe without any special work needing to be done by its clients. It achieves this though using threading locks; there is one lock to serialize access to the module's shared data, and each handler also creates a lock to serialize access to its underlying I/O.*
>
> — **LOGGING FACILITY FOR PYTHON, THREAD SAFETY (HTTPS://DOCS.PYTHON.ORG/3/LIBRARY/LOGGING.HTML#THREAD-SAFETY).**

Therefore, you can log from your target task functions directly (https://superfastpython.com/threadpoolexecutor-logging/).

How Do You Unit Tasks and Thread Pools?

You can unit test your target task functions directly, perhaps mocking any external resources required.

You can unit test your usage of the thread pool with mock tasks that do not interact with external resources.

Unit testing of tasks and the thread pool itself must be considered as part of your design and may require that connection to the IO resource be configurable so that it can be mocked, and that the target task function called by your thread pool is configurable so that it can be mocked.

How Do You Compare Serial to Parallel Performance?

You can compare the performance of your program with and without the thread pool.

This can be a useful exercise to confirm that making use of the **ThreadPoolExecutor** in your program has resulted in a speed-up.

Perhaps the simplest approach is to manually record the start and end time of your code and subtract the end from the start time to report the total execution time. Then record the time with and without the use of the thread pool.

```python
# SuperFastPython.com
# example of recording the execution time of a program
import time

# record the start time
start_time = time.time()
# do work with or without a thread pool
# ....
time.sleep(3)
# record the end time
end_time = time.time()
# report execution time
total_time = end_time - start_time
print(f'Execution time: {total_time:.1f} seconds.')
```

Using an average program execution time might give a more stable program timing than a one-off run.

You can record the execution time 3 or more times for your program without the thread pool then calculate the average as the sum of times divided by the total runs. Then repeat this exercise to calculate the average time with the thread pool.

This would probably only be appropriate if the running time of your program is minutes rather than hours.

You can then compare the serial vs. parallel version by calculating the speed up multiple as:

- Speed-Up Multiple = Serial Time / Parallel Time

For example, if the serial run of a program took 15 minutes (900 seconds) and the parallel version with a **ThreadPoolExecutor** took 5 minutes (300 seconds), then the percentage multiple up would be calculated as:

- Speed-Up Multiple = Serial Time / Parallel Time
- Speed-Up Multiple = 900 / 300
- Speed-Up Multiple = 3

That is, the parallel version of the program with the **ThreadPoolExecutor** is 3 times faster or 3x faster.

You can multiply the speed-up multiple by 100 to give a percentage

- Speed-Up Percentage = Speed-Up Multiple * 100

In this example, the parallel version is 300% faster than the serial version.

How Do You Set chunksize in map()?

The **map()** function on the **ThreadPoolExecutor** takes a parameter called “**chunksize**” which defaults to 1.

The chunksize parameter is not used by the **ThreadPoolExecutor**; it is only used by the **ProcessPoolExecutor**, therefore you can safely ignore it.

Setting this parameter does nothing when using the **ThreadPoolExecutor**.

How Do You Submit a Follow-up Task?

Some tasks require that a second task be executed that makes use of the result from the first task in some way.

We might call this the need to execute a follow-up task (https://superfastpython.com/threadpoolexecutor-followup-task/) for each task that is submitted, which might be conditional on the result in some way.

There are a few ways to submit a follow-up task.

One approach would be to submit the follow-up task as we are processing the results from the first task.

For example, we can process the result from each of the first tasks as they complete, then manually call submit() for each follow-up task when needed and store the new future object in a second list for later use.

We can make this example of submitting follow-up tasks concrete with a full example.

```
# SuperFastPython.com
# example of submitting follow-up tasks
from time import sleep
from random import random
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed

# mock test that works for moment
def task1():
    value = random()
    sleep(value)
    print(f'Task 1: {value}')
    return value

# mock test that works for moment
def task2(value1):
    value2 = random()
    sleep(value2)
    print(f'Task 2: value1={value1}, value2={value2}')
    return value2

# start the thread pool
with ThreadPoolExecutor(5) as executor:
    # send in the first tasks
    futures1 = [executor.submit(task1) for _ in range(10)]
    # process results in the order they are completed
    futures2 = list()
    for future1 in as_completed(futures1):
        # get the result
        result = future1.result()
        # check if we should trigger a follow-up task
        if result > 0.5:
            future2 = executor.submit(task2, result)
            futures2.append(future2)
    # wait for all follow-up tasks to complete
```

Running the example starts a thread pool with 5 worker threads and submits 10 tasks.

We then process the results for the tasks as they are completed. If a result from the first round of tasks requires a follow-up task, we submit the follow-up task and keep track of the **Future** object in a second list.

These follow-up tasks are submitted as needed, rather than waiting until all first round tasks are completed, which is a nice benefit of using the **as_completed()** function with a list of Future objects.

We can see that in this case, five first round tasks resulted in follow-up tasks.

```
Task 1: 0.021868594663798424
Task 1: 0.07220684891621587
Task 1: 0.1889059597524675
Task 1: 0.4044025009726221
Task 1: 0.5377728619737125
Task 1: 0.5627604576510364
Task 1: 0.19590409149609522
Task 1: 0.8350495785309672
Task 2: value1=0.8350495785309672, value2=0.21472292885893007
Task 2: value1=0.5377728619737125, value2=0.6180101068687799
Task 1: 0.9916368220002719
Task 1: 0.6688307514083958
Task 2: value1=0.6688307514083958, value2=0.2691774622597396
Task 2: value1=0.5627604576510364, value2=0.859736361909423
Task 2: value1=0.9916368220002719, value2=0.642060404763057
```

You might like to use a separate thread pool for follow-up tasks, to keep things separate.

I would not recommend submitting new tasks from within a task.

This would require access to the thread pool either as a global variable or by being passed in and would break the idea of tasks being pure functions that don't have side effects, a good practice when using thread pools.

How Do You Store Local State for Each Thread?

You can use thread local variables for worker threads in the **ThreadPoolExecutor**.

A common pattern would be to use a custom initializer function for each worker thread to set up a thread local variable specific to each worker thread.

This thread local variables can then be used by each thread within each task, requiring that the task be aware of the thread local mechanism.

We can demonstrate this with a worked example.

First, we can define a custom initializer function that takes a thread local context and sets up a custom variable named "key" with a unique value between 0.0 and 1.0 for each worker thread.

```python
# function for initializing the worker thread
def initializer_worker(local):
    # generate a unique value for the worker thread
    local.key = random()
    # store the unique worker key in a thread local variable
    print(f'Initializing worker thread {local.key}')
```

We can then define our target task function to take the same thread local context and to access the thread local variable for the worker thread and make use of it.

```
# a mock task that sleeps for a random amount of time less than one second
def task(local):
    # access the unique key for the worker thread
    mykey = local.key
    # make use of it
    sleep(mykey)
    return f'Worker using {mykey}'
```

We can then configure our new **ThreadPoolExecutor** instance to use the initializer with the required local argument.

```
...
# get the local context
local = threading.local()
# create a thread pool
executor = ThreadPoolExecutor(max_workers=2, initializer=initializer_worker, initargs=(local,))
```

Then dispatch tasks into the thread pool with the same thread local context.

```
...
# dispatch asks
futures = [executor.submit(task, local) for _ in range(10)]
```

Tying this together, the complete example of using a **ThreadPoolExecutor** with thread local storage is listed below.

```
# SuperFastPython.com
# example of thread local storage for worker threads
from time import sleep
from random import random
import threading
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# function for initializing the worker thread
def initializer_worker(local):
    # generate a unique value for the worker thread
    local.key = random()
    # store the unique worker key in a thread local variable
    print(f'Initializing worker thread {local.key}')

# a mock task that sleeps for a random amount of time less than one second
def task(local):
    # access the unique key for the worker thread
    mykey = local.key
    # make use of it
    sleep(mykey)
    return f'Worker using {mykey}'

# get the local context
local = threading.local()
# create a thread pool
executor = ThreadPoolExecutor(max_workers=2, initializer=initializer_worker, initargs=(local,))
# dispatch asks
futures = [executor.submit(task, local) for _ in range(10)]
# wait for all threads to complete
for future in futures:
    result = future.result()
    print(result)
# shutdown the thread pool
executor.shutdown()
print('done')
```

Running the example first configures the thread pool to use our custom initializer function, which sets up a thread local variable for each worker thread with a unique value, in this case two threads each with a value between 0 and 1.

Each worker thread then works on tasks in the queue, all ten of them, each using the specific value of the thread local variable setup for the thread in the initialization function.

```
Initializing worker thread 0.9360961457279074
Initializing worker thread 0.9075641843481475
Worker using 0.9360961457279074
Worker using 0.9075641843481475
Worker using 0.9075641843481475
Worker using 0.9360961457279074
Worker using 0.9075641843481475
Worker using 0.9360961457279074
Worker using 0.9075641843481475
Worker using 0.9360961457279074
Worker using 0.9075641843481475
Worker using 0.9360961457279074
done
```

How Do You Show Progress of All Tasks?

There are many ways to show progress for tasks (https://superfastpython.com/threadpoolexecutor-progress/) that are being executed by the **ThreadPoolExecutor**.

Perhaps the simplest is to use a callback function that updates a progress indicator. This can be achieved by defining the progress indicator function and registering it with the Future object for each task via the **add_done_callback()** function.

The simplest progress indicator is to print a dot to the screen for each task that completes.

The example below demonstrates this simple progress indicator.

```python
# SuperFastPython.com
# example of a simple progress indicator
from time import sleep
from random import random
from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import wait

# simple progress indicator callback function
def progress_indicator(future):
    print('.', end='', flush=True)

# mock test that works for moment
def task(name):
    sleep(random())

# start the thread pool
with ThreadPoolExecutor(2) as executor:
    # send in the tasks
    futures = [executor.submit(task, i) for i in range(20)]
    # register the progress indicator callback
    for future in futures:
        future.add_done_callback(progress_indicator)
    # wait for all tasks to complete
print('\nDone!')
```

Running the example starts a thread pool with two worker threads and dispatches 20 tasks.

A progress indicator callback function is registered with each **Future** object that prints one dot as each task is completed, ensuring that the standard output is flushed with each call to **print()** and that no new line is printed.

This ensures that each we see the dot immediately regardless of the thread that prints and that all dots appear on one line.

Each task will work for a variable amount of time less than one second and a dot is printed once the task is completed.

```
....................
Done!
```

A more elaborate progress indicator must know the total number of tasks and will use a thread safe counter to update the status of the number of tasks completed out of all tasks to be completed.

Do We Need to Have a Check for __main__?

You do not need to have a check for **__main__** when using the **ThreadPoolExecutor**.

You do need a check for **__main__** when using the **Process** version of the pool, called a **ProcessPoolExecutor**. This may be the source of confusion.

How Do You Get a Future Object for Tasks Added With map()?

When you call **map()**, it does create a **Future** object for each task.

Internally, **submit()** is called for each item in the iterable provided to the call to **map()**.

Nevertheless, there is no clean way to access the **Future** object for tasks sent into the thread pool via **map()**.

Each task on the **ThreadPoolExecutor** object's internal work queue is an instance of a **_WorkItem** that does have a reference to the **Future** object for the task. You can access the **ThreadPoolExecutor** object's internal queue, but you have no safe way of enumerating items in the queue without removing them.

If you need a **Future** object for a task, call **submit()**.

Can I Call shutdown() From Within the Context Manager?

You can call **shutdown()** within the context manager (https://superfastpython.com/threadpoolexecutor-context-manager/), but there are not many use cases.

It does not cause an error that I can see.

You may want to call **shutdown()** explicitly if you wish to cancel all scheduled tasks and you don't have access to the **Future** objects, and you wish to do other clean-up before waiting for all running tasks to stop.

It would be strange if you found yourself in this situation.

Nevertheless, here is an example of calling **shutdown()** from within the context manager.

```python
# SuperFastPython.com
# example of shutting down within a context manager
from time import sleep
from concurrent.futures import ThreadPoolExecutor

# mock test that works for moment
def task(name):
    sleep(2)
    print(f'Done: {name}')

# start the thread pool
with ThreadPoolExecutor(1) as executor:
    # send some tasks into the thread pool
    print('Sending in tasks...')
    futures = [executor.submit(task, i) for i in range(10)]
    # explicitly shutdown within the context manager
    print('Shutting down...')
    executor.shutdown(wait=False, cancel_futures=True)
    # shutdown called again here when context manager exited
    print('Waiting...')
print('Doing other things...')
```

Running the example starts a thread pool with one worker thread and then sends in ten tasks to execute.

We then explicitly call shutdown on the thread pool and cancel all scheduled tasks without waiting.

We then exit the context manager and wait for all tasks to complete. This second shutdown works as expected, waiting for the one running task to complete before returning.

```
Sending in tasks...
Shutting down...
Waiting...
Done: 0
Doing other things...
```

Common Objections to Using ThreadPoolExecutor

The **ThreadPoolExecutor** may not be the best solution for all multithreading problems in your program.

That being said, there may also be some misunderstandings that are preventing you from making full and best use of the capabilities of the **ThreadPoolExecutor** in your program.

In this section, we review some of the common objections seen by developers when considering using the **ThreadPoolExecutor** in their code.

What About the Global Interpreter Lock (GIL)?

The Global Interpreter Lock, or GIL for short, is a design decision with the reference Python interpreter.

It refers to the fact that the implementation of the Python interpreter makes use of a master lock that prevents more than one Python instruction executing at the same time.

This prevents more than one thread of execution within Python programs, specifically within each Python process, that is each instance of the Python interpreter.

The implementation of the GIL means that Python threads may be concurrent, but cannot run in parallel. Recall that concurrent means that more than one task can be in progress at the same time; parallel means more than one task actually executing at the same time. Parallel tasks are concurrent, concurrent tasks may or may not execute in parallel.

It is the reason behind the heuristic that Python threads should only be used for IO-bound tasks, and not CPU-bound tasks, as IO-bound tasks will wait in the operating system kernel for remote resources to respond (not executing Python instructions), allowing other Python threads to run and execute Python instructions.

Put another way, the GIL does not mean we cannot use threads in Python, only that some use cases for Python threads are viable or appropriate.

This design decision was made within the reference implementation of the Python interpreter (from Python.org), but may not impact other interpreters (such as PyPy, Iron Python, and Jython) that allow multiple Python instructions to be executed concurrently and in parallel.

You can learn more about this topic here:

- ThreadPoolExecutor vs. the Global Interpreter Lock (GIL) (https://superfastpython.com/threadpoolexecutor-vs-gil/)

Are Python Threads “Real Threads”?

Yes.

Python makes use of real system-level threads, also called kernel-level threads, a capability provided by modern operating systems like Windows, Linux, and MacOS.

Python threads are not software-level threads, sometimes called user-level threads or green threads (https://en.wikipedia.org/wiki/Green_threads).

Aren't Python Threads Buggy?

No.

Python threads are not buggy.

Python threading is a first class capability of the Python platform and has been for a very long time.

Isn't Python a Bad Choice for Concurrency?

Developers love python for many reasons, most commonly because it is easy to use and fast for development.

Python is commonly used for glue code, one-off scripts, but more and more for large-scale software systems.

If you are using Python and then you need concurrency, then you work with what you have. The question is moot.

If you need concurrency and you have not chosen a language, perhaps another language would be more appropriate, or perhaps not. Consider the full scope of functional and non-functional requirements (or user needs, wants, and desires) for your project and the capabilities of different development platforms.

Why Not Always Use ProcessPoolExecutor Instead?

The **ProcessPoolExecutor** supports pools of processes, unlike the **ThreadPoolExecutor** that supports pools of threads.

Threads and Processes are quite different and choosing one over the other is intentional.

A Python program is a process that has a main thread. You can create many additional threads in a Python process. You can also fork or spawn many Python processes, each of which will have one thread, and may spawn additional threads.

More broadly, threads are lightweight and can share memory (data and variables) within a process, whereas processes are heavyweight and require more overhead and impose more limits on sharing memory (data and variables).

Typically, processes are used for CPU-bound tasks and threads are used for IO-bound tasks, and this is a good heuristic, but this does not have to be the case.

Perhaps **ProcessPoolExecutor** is a better fit for your specific problem. Perhaps try it and see.

You can learn more about this topic here:

- ThreadPoolExecutor vs. ProcessPoolExecutor in Python (https://superfastpython.com/threadpoolexecutor-vs-processpoolexecutor/)

Why Not Use threading.Thread instead?

The **ThreadPoolExecutor** is like the "*auto mode*" for Python threading.

If you have a more sophisticated use case, you may need to use the **threading.Thread** class directly.

This may be because you require more synchronization between threads with locking mechanisms, and/or more coordination between threads with barriers and semaphores.

It may be that you have a simpler use case, such as a single task, in which case perhaps a thread pool would be too heavy a solution.

That being said, if you find yourself using the **Thread** class with the target keyword for pure functions (functions that don't have side effects), perhaps you would be better suited to using the **ThreadPoolExecutor**.

You can learn more about this topic here:

- ThreadPoolExecutor vs. Thread in Python (https://superfastpython.com/threadpoolexecutor-vs-threads/)

Why Not Use AsyncIO?

AsyncIO can be an alternative to using a **ThreadPoolExecutor**.

AsyncIO is designed to support large numbers of IO operations, perhaps thousands to tens of thousands, all within a single Thread.

It requires an alternate programming paradigm, called reactive programming (https://en.wikipedia.org/wiki/Reactive_programming), which can be challenging for beginners.

Nevertheless, it may be a better alternative to using a thread pool for many applications.

You can learn more about this topic here:

- ThreadPoolExecutor vs. AsyncIO in Python (https://superfastpython.com/threadpoolexecutor-vs-asyncio/)

Further Reading

This section lists helpful additional resources on the topic.

Books

- ThreadPoolExecutor Class API Cheat Sheet (https://superfastpython.gumroad.com/l/nyfpaz)
- Concurrent Futures API Interview Questions (https://superfastpython.com/pcfiq-sidebar)
- ThreadPoolExecutor Jump-Start (https://superfastpython.com/ptpej-further-reading) (my 7-day course)

APIs

- Python Built-in Functions (https://docs.python.org/3/library/functions.html)
- concurrent.futures — Launching parallel tasks (https://docs.python.org/3/library/concurrent.futures.html)
- cpython/Lib/concurrent/futures/thread.py Source Code (https://github.com/python/cpython/blob/3.10/Lib/concurrent/futures/thread.py)
- cpython/Lib/concurrent/futures/_base.py Source Code (https://github.com/python/cpython/blob/3.10/Lib/concurrent/futures/_base.py)
- threading — Thread-based parallelism (https://docs.python.org/3/library/threading.html)
- Multiprocessing — Process-based parallelism (https://docs.python.org/3/library/multiprocessing.html)
- queue — A synchronized queue class (https://docs.python.org/3/library/queue.html)
- logging — Logging facility for Python (https://docs.python.org/3/library/logging.html)

References

- Thread (computing), Wikipedia (https://en.wikipedia.org/wiki/Thread_(computing)).
- Brian Quinlan Homepage (http://sweetapp.com/).
- Futures and promises, Wikipedia (https://en.wikipedia.org/wiki/Futures_and_promises).
- Pure function, Wikipedia (https://en.wikipedia.org/wiki/Pure_function).

Conclusions

This is a large guide, and you have discovered in great detail how the **ThreadPoolExecutor** works and how to best use it on your project.

Did you find this guide useful?
I'd love to know, please share a kind word in the comments below.

Have you used the ThreadPoolExecutor on a project?
I'd love to hear about it, please let me know in the comments.

Do you have any questions?
Leave your question in a comment below and I will reply fast with my best advice.