Python ThreadPoolExecutor

Summary: in this tutorial, you'll learn how to use the Python `ThreadPoolExecutor` to develop multi-threaded programs.

Introduction to the Python ThreadPoolExecutor class

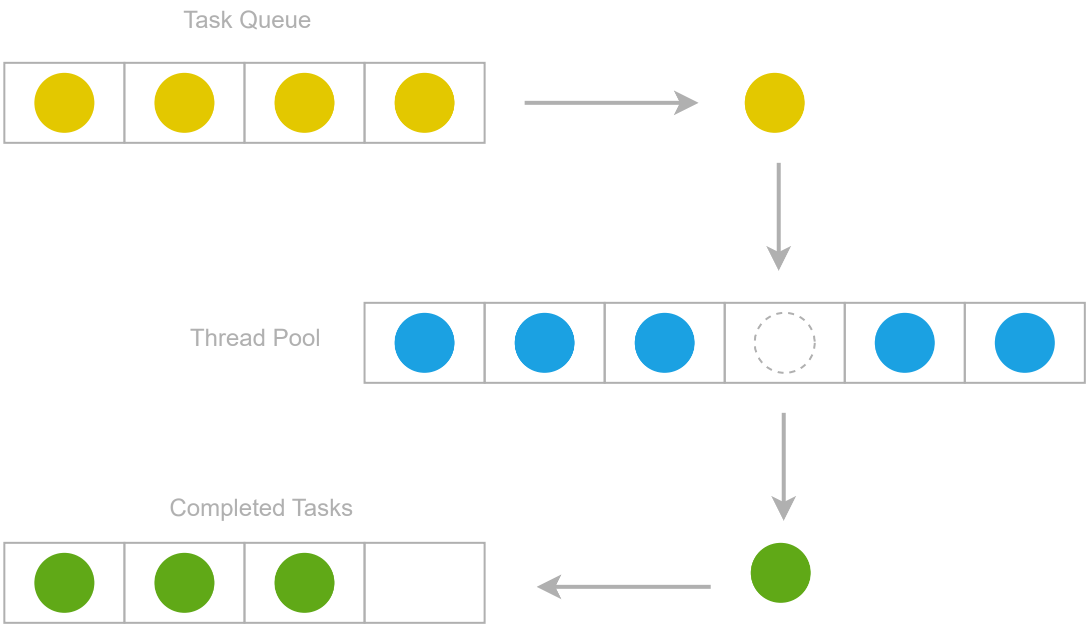
In the multithreading tutorial (https://www.pythontutorial.net/advanced-python/python-threading/) , you learned how to manage multiple threads in a program using the `Thread` class of the `threading` module. The `Thread` class is useful when you want to create threads manually.

However, manually managing threads is not efficient because creating and destroying many threads frequently are very expensive in terms of computational costs.

Instead of doing so, you may want to reuse the threads if you expect to run many ad-hoc tasks in the program. A thread pool allows you to achieve this.

Thread pool

A thread pool is a pattern for achieving concurrency of execution in a program. A thread pool allows you to automatically manage a pool of threads efficiently:



Each thread in the pool is called a worker thread or a worker. A thread pool allows you to reuse the worker threads once the tasks are completed. It also protects against unexpected failures such as exceptions (https://www.pythontutorial.net/python-oop/python-exceptions/) .

Typically, a thread pool allows you to configure the number of worker threads and provides a specific naming convention for each worker thread.

To create a thread pool, you use the `ThreadPoolExecutor` class from the `concurrent.futures` module.

ThreadPoolExecutor

The `ThreadPoolExecutor` class extends (https://www.pythontutorial.net/python-oop/python-inheritance/) the `Executor` class and returns a `Future` object.

Executor

The `Executor` class has three methods to control the thread pool:

- `submit()` – dispatch a function to be executed and return a `Future` object. The `submit()` method takes a function and executes it asynchronously.
- `map()` – execute a function asynchronously for each element in an iterable.
- `shutdown()` – shut down the executor.

When you create a new instance of the `ThreadPoolExecutor` class, Python starts the `Executor` .

Once completing working with the executor, you must explicitly call the `shutdown()` method to release the resource held by the executor. To avoid calling the `shutdown()` method explicitly, you can use the context manager (https://www.pythontutorial.net/advanced-python/python-context-managers/) .

Future object

A `Future` is an object that represents the eventual result of an asynchronous operation. The Future class has two useful methods:

- `result()` – return the result of an asynchronous operation.
- `exception()` – return the exception of an asynchronous operation in case an exception occurs.

Python ThreadPoolExecutor examples

The following program uses a single thread:

```
from time import sleep, perf_counter

def task(id):
    print(f'Starting the task {id}...')
    sleep(1)
    return f'Done with task {id}'

start = perf_counter()

print(task(1))
print(task(2))

finish = perf_counter()

print(f"It took {finish-start} second(s) to finish.")
```

Output:

```
Starting the task 1...
Done with task 1
Starting the task 2...
Done with task 2
It took 2.0144479 second(s) to finish.
```

How it works.

First, define the `task()` function that takes about one second to finish. The `task()` function calls the `sleep()` function to simulate a delay:

```
def task(id):
    print(f'Starting the task {id}...')
    sleep(1)
    return f'Done with task {id}'
```

Second, call the `task()` function twice and print out the result. Before and after calling the `task()` function, we use the `perf_counter()` to measure the start and finish time:

```
start = perf_counter()

print(task(1))
print(task(2))

finish = perf_counter()
```

Third, print out the time the program took to run:

```
print(f"It took {finish-start} second(s) to finish.")
```

Because the `task()` function takes one second, calling it twice will take about 2 seconds.

Using the submit() method example

To run the `task()` function concurrently, you can use the `ThreadPoolExecutor` class:

```python
from time import sleep, perf_counter
from concurrent.futures import ThreadPoolExecutor

def task(id):
    print(f'Starting the task {id}...')
    sleep(1)
    return f'Done with task {id}'

start = perf_counter()

with ThreadPoolExecutor() as executor:
    f1 = executor.submit(task, 1)
    f2 = executor.submit(task, 2)

    print(f1.result())
    print(f2.result())

finish = perf_counter()

print(f"It took {finish-start} second(s) to finish.")
```

Output:

```
Starting the task 1...
Starting the task 2...
Done with task 1
Done with task 2
It took 1.0177214 second(s) to finish.
```

The output shows that the program took about 1 second to finish.

How it works (we'll focus on the thread pool part):

First, import the `ThreadPoolExecutor` class from the `concurrent.futures` module:

```
from concurrent.futures import ThreadPoolExecutor
```

Second, create a thread pool using the `ThreadPoolExecutor` using a context manager:

```
with ThreadPoolExecutor() as executor:
```

Third, calling the `task()` function twice by passing it to the `submit()` method of the executor:

```
with ThreadPoolExecutor() as executor:
    f1 = executor.submit(task, 1)
    f2 = executor.submit(task, 2)

    print(f1.result())
    print(f2.result())
```

The `submit()` method returns a Future object. In this example, we have two Future objects `f1` and `f2` . To get the result from the Future object, we called its `result()` method.

Using the map() method example

The following program uses a `ThreadPoolExecutor` class. However, instead of using the `submit()` method, it uses the `map()` method to execute a function:

```
from time import sleep, perf_counter
from concurrent.futures import ThreadPoolExecutor


def task(id):
    print(f'Starting the task {id}...')
    sleep(1)
    return f'Done with task {id}'
```

```
start = perf_counter()

with ThreadPoolExecutor() as executor:
    results = executor.map(task, [1,2])
    for result in results:
        print(result)

finish = perf_counter()

print(f"It took {finish-start} second(s) to finish.")
```

How it works.

First, call the `map()` method of the executor object to run the task function for each id in the list [1,2]. The `map()` method returns an iterator that contains the result of the function calls.

```
results = executor.map(task, [1,2])
```

Second, iterate over the results and print them out:

```
for result in results:
    print(result)
```

Python ThreadPoolExecutor practical example

The following program downloads multiple images from Wikipedia using a thread pool:

```
from concurrent.futures import ThreadPoolExecutor
from urllib.request import urlopen
import time
import os
```

```
def download_image(url):
    image_data = None
    with urlopen(url) as f:
        image_data = f.read()

    if not image_data:
        raise Exception(f"Error: could not download the image from {url}")

    filename = os.path.basename(url)
    with open(filename, 'wb') as image_file:
        image_file.write(image_data)
        print(f'{filename} was downloaded...')

start = time.perf_counter()

urls = ['https://upload.wikimedia.org/wikipedia/commons/9/9d/Python_bivittatus_17
        'https://upload.wikimedia.org/wikipedia/commons/4/48/Python_Regius.jpg',
        'https://upload.wikimedia.org/wikipedia/commons/d/d3/Baby_carpet_python_c
        'https://upload.wikimedia.org/wikipedia/commons/f/f0/Rock_python_pratik.J
        'https://upload.wikimedia.org/wikipedia/commons/0/07/Dulip_Wilpattu_Pytho

with ThreadPoolExecutor() as executor:
      executor.map(download_image, urls)

finish = time.perf_counter()

print(f'It took {finish-start} second(s) to finish.')
```

How it works.

First, define a function `download_image()` that downloads an image from an URL and saves it into a file:

```
def download_image(url):
    image_data = None
    with urlopen(url) as f:
        image_data = f.read()

    if not image_data:
        raise Exception(f"Error: could not download the image from {url}")

    filename = os.path.basename(url)
    with open(filename, 'wb') as image_file:
        image_file.write(image_data)
        print(f'{filename} was downloaded...')
```

The `download_image()` function the `urlopen()` function from the `urllib.request` module to download an image from an URL.

Second, execute the `download_image()` function using a thread pool by calling the `map()` method of the `ThreadPoolExecutor` object:

```
with ThreadPoolExecutor() as executor:
      executor.map(download_image, urls)
```

Summary

- A thread pool is a pattern for managing multiple threads efficiently.
- Use `ThreadPoolExecutor` class to manage a thread pool in Python.
- Call the `submit()` method of the `ThreadPoolExecutor` to submit a task to the thread pool for execution. The `submit()` method returns a Future object.
- Call the `map()` method of the `ThreadPoolExecutor` to map to execute a function in a thread pool with each element in a list.