Python ProcessPoolExecutor

Summary: in this tutorial, you'll learn how to use the Python ProcessPoolExecutor to create and manage a process pool effectively.

Introduction to the Python ProcessPoolExecutor class

In the previous tutorial, you learned how to create running code in parallel by creating processes manually using the `Process` class from the `multiprocessing` module. However, manually creating processes is not efficient.

To manage the processes more efficiently, you can use a process pool. Like a thread pool (https://www.pythontutorial.net/advanced-python/python-threadpoolexecutor/) , a process pool is a pattern for managing processes automatically.

The `ProcessPoolExecutor` class from the `concurrent.futures` module allows you to create and manage a process pool.

For example, the `ProcessPoolExecutor` class uses the number of CPU cores for creating an optimized number of processes to create.

The `ProcessPoolExecutor` extends the `Executor` class that has three methods:

- `submit()` – dispatch a function to be executed by the process and return a Future object.
- `map()` – call a function to an iterable of elements.
- `shutdown()` – shut down the executor.

To release the resources held by the executor, you need to call the `shutdown()` method explicitly. To shut down the executor automatically, you can use a context manager (https://www.pythontutorial.net/advanced-python/python-context-managers/) .

The `Future` object represents an eventful result of an asynchronous operation. It has two main methods for getting the result:

- `result()` – return the result from the asynchronous operation.
- `exception()` – return an exception that occurred while running the asynchronous operation.

Python ProcessPoolExecutor example

The following program uses a process pool to create thumbnails for pictures in the `images` folder and save them to the `thumbs` folder.

```
import time
import os
from concurrent.futures import ProcessPoolExecutor
from PIL import Image, ImageFilter

filenames = [
    'images/1.jpg',
    'images/2.jpg',
    'images/3.jpg',
    'images/4.jpg',
    'images/5.jpg',
]

def create_thumbnail(filename, size=(50,50),thumb_dir ='thumbs'):
    img = Image.open(filename)
```

```
    img = img.filter(ImageFilter.GaussianBlur())
    img.thumbnail(size)
    img.save(f'{thumb_dir}/{os.path.basename(filename)}')
    print(f'{filename} was processed...')


if __name__ == '__main__':
    start = time.perf_counter()

    with ProcessPoolExecutor() as executor:
        executor.map(create_thumbnail, filenames)

    finish = time.perf_counter()

    print(f'It took {finish-start: .2f} second(s) to finish')
```

Output:

```
images/5.jpg was processed...
images/4.jpg was processed...
images/3.jpg was processed...
images/2.jpg was processed...
images/1.jpg was processed...
It took  0.79 second(s) to finish
```

Note that to run the program, you need to install the `Pillow` which is a popular library for image processing by running the pip command `pip install Pillow` .

How it works.

First, declare a list of files for creating thumbnails:

```
filenames = [
    'images/1.jpg',
```

```
    'images/2.jpg',
    'images/3.jpg',
    'images/4.jpg',
    'images/5.jpg',
]
```

Second, define a function that creates a thumbnail from an image file and saves the output to the thumbnail folder:

```
def create_thumbnail(filename, size=(50,50),thumb_dir ='thumbs'):
    img = Image.open(filename)
    img = img.filter(ImageFilter.GaussianBlur())
    img.thumbnail(size)
    img.save(f'{thumb_dir}/{os.path.basename(filename)}')
    print(f'{filename} was processed...')
```

Third, create a process pool and call the `create_thumbnail()` function for each picture specified in the filenames:

```
with ProcessPoolExecutor() as executor:
    executor.map(create_thumbnail, filenames)
```

Summary

- Use the Python ProcessPoolExecutor class to create and manage a process pool automatically.