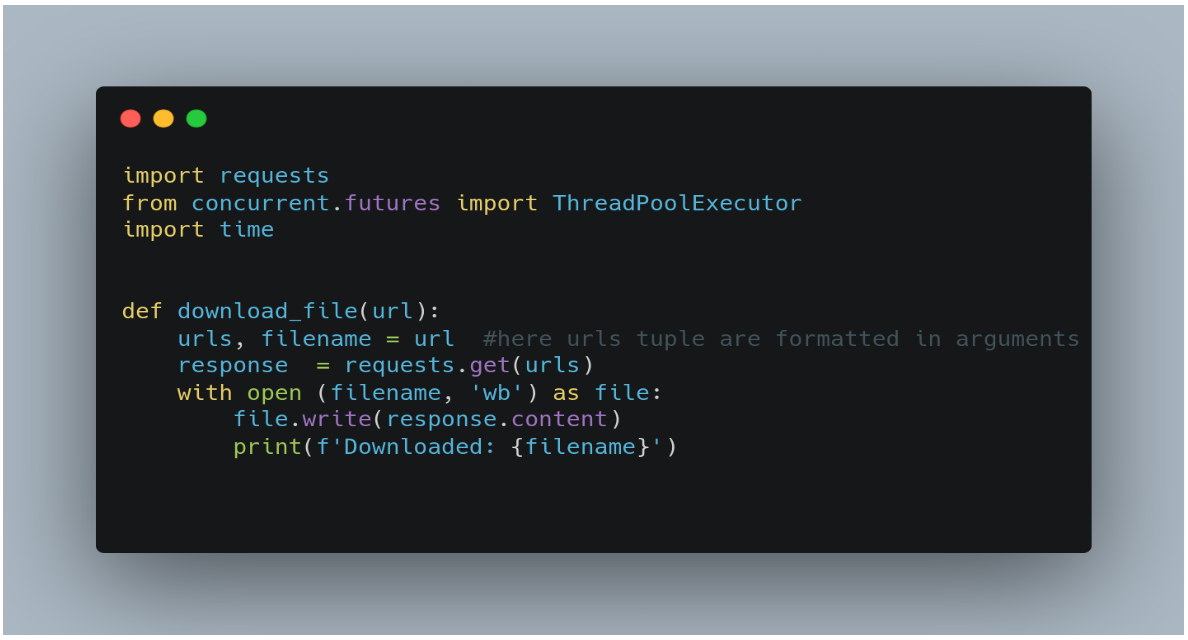
Multithreding:

Multithreading involves running multiple threads simultaneously within a single process. Each thread independently performs its designated tasks, and the OS schedules their executions.

Advantages:
- It is suitable for tasks mainly involving executing python bytecode.
- Multithreading in an interactive application enables a program to continue running even if a section is blocked or executing a lengthy process, increasing user responsiveness.

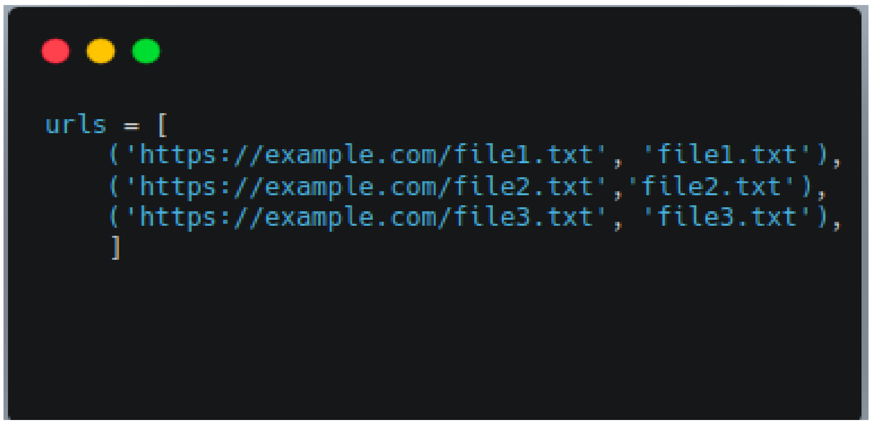
Disadvatages:
- It can consumes a large space of stocks of blocked threads.
- Increased complexity and potential for Deadlocks.
- Increasedcomplexity in Debugging and Testing

Here, first we import the necessary modules. We use concurrent.futures for the thread pool and requests for downloading files.



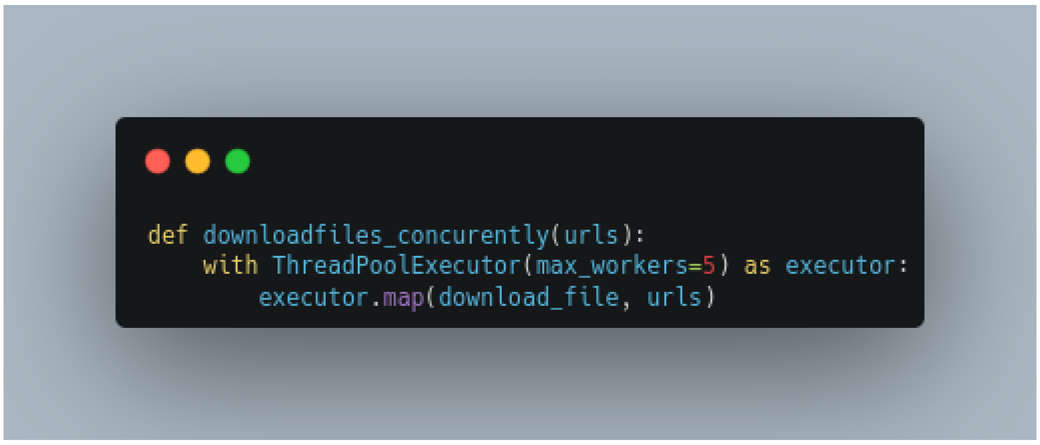
```
import requests
from concurrent.futures import ThreadPoolExecutor
import time


def download_file(url):
    urls, filename = url  #here urls tuple are formatted in arguments
    response  = requests.get(urls)
    with open (filename, 'wb') as file:
        file.write(response.content)
        print(f'Downloaded: {filename}')
```

Next, we define the function that will download a file from the provided urls. This function will be executed in in separate threads. And the urls are provided are jotted below:



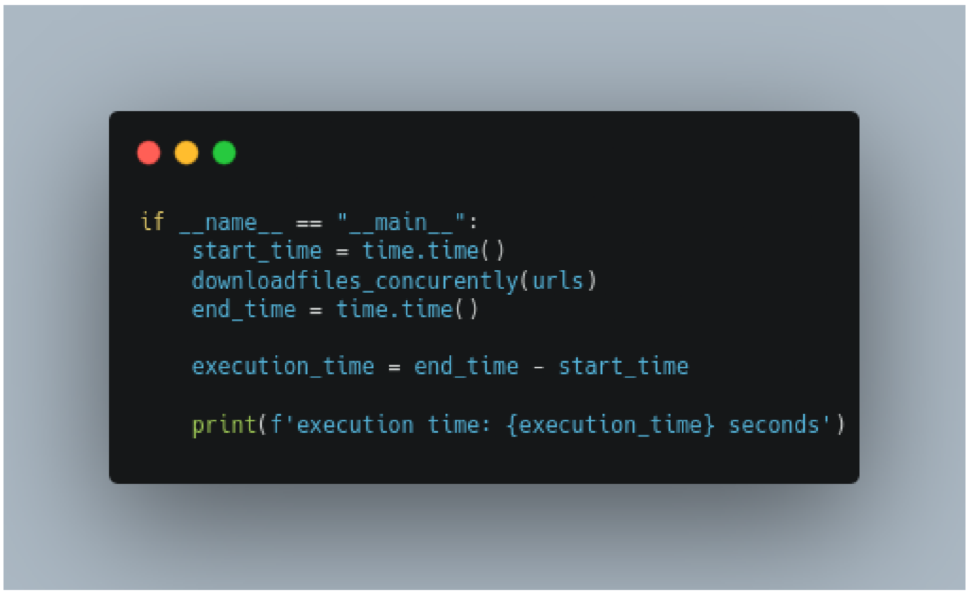
```
urls = [
    ('https://example.com/file1.txt', 'file1.txt'),
    ('https://example.com/file2.txt','file2.txt'),
    ('https://example.com/file3.txt', 'file3.txt'),
    ]
```

Now, we will use threadpoolexecutor to download all file concurrently. We will create thread pool with a specified number of threads and submit the download_file function to the executor for each url in our list.



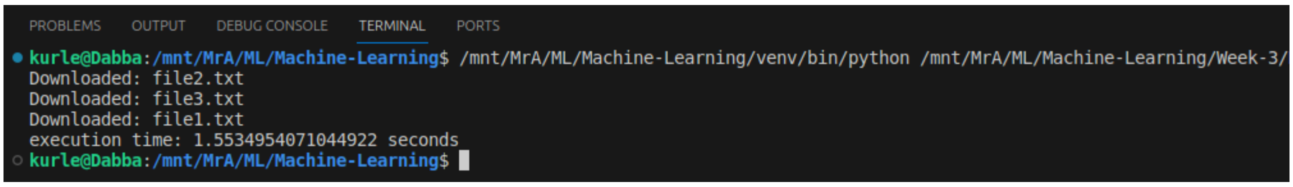
```
def downloadfiles_concurently(urls):
    with ThreadPoolExecutor(max_workers=5) as executor:
        executor.map(download_file, urls)
```

At last, we call our downloadfiles_concurently function by passing our urls to start download.



```
if __name__ == "__main__":
    start_time = time.time()
    downloadfiles_concurently(urls)
    end_time = time.time()

    execution_time = end_time - start_time

    print(f'execution time: {execution_time} seconds')
```

output:



```
● kurle@Dabba:/mnt/MrA/ML/Machine-Learning$ /mnt/MrA/ML/Machine-Learning/venv/bin/python /mnt/MrA/ML/Machine-Learning/Week-3/
  Downloaded: file2.txt
  Downloaded: file3.txt
  Downloaded: file1.txt
  execution time: 1.5534954071044922 seconds
○ kurle@Dabba:/mnt/MrA/ML/Machine-Learning$
```

This example demonstrates how to use multithreading in Python to perform multiple tasks concurrently, which can significantly speed up the execution time of your programs, especially when dealing with I/O-bound tasks like downloading files from the internet.

Multiprocessing :

Multiprocessing refers to the ability of a system to support more than one processor at the same time. Applications in a multiprocessing system are broken to smaller routines that run independently. The operating system allocates these threads to the processors improving performance of the system.

Advantages:

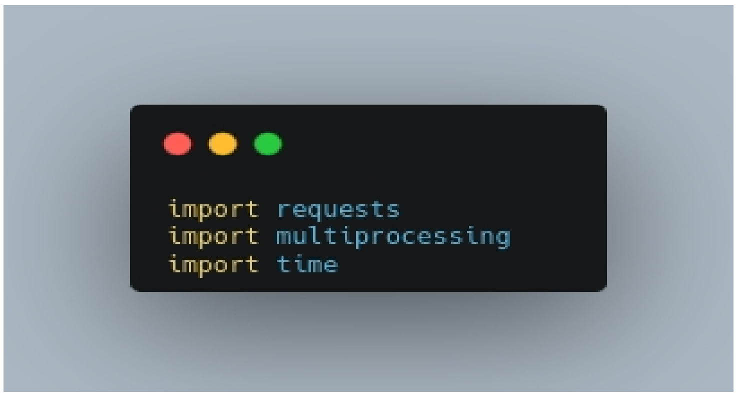
- These systems are fault-tolerant. Failure of a few processors does not bring the entire system to a halt.

Disadvantages:

- It is very difficult to balance the workload among processors rationally.
- Specialized synchronization schemes are necessary for managing multiple processors.

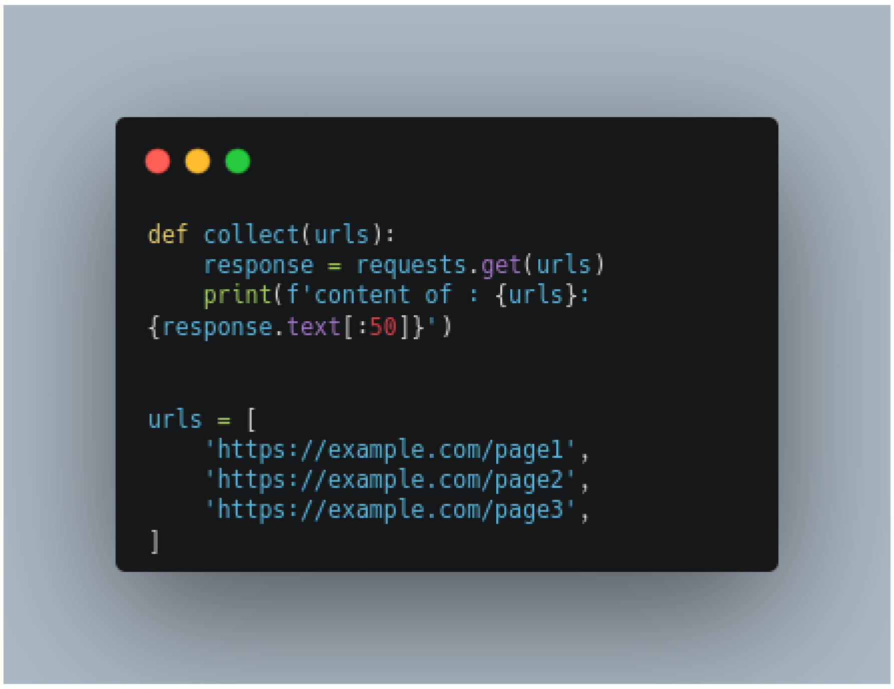
Example:

Here we import the necessary modules for use of multiprocessing.



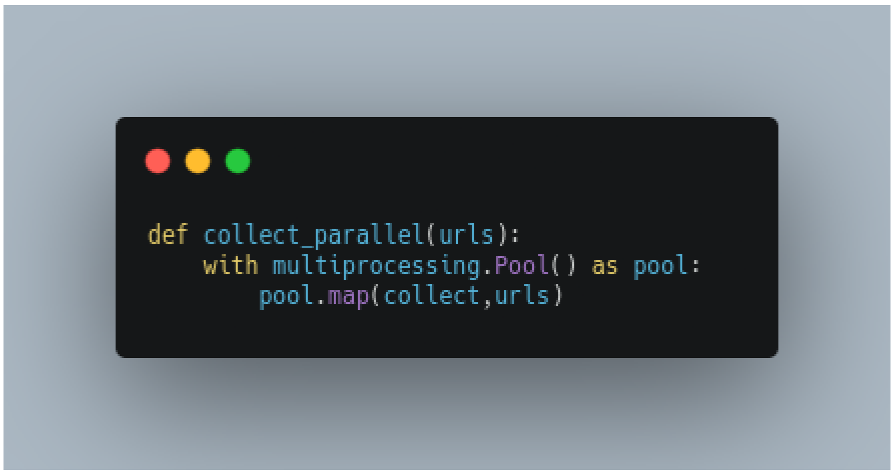
```python
import requests
import multiprocessing
import time
```

Now, we create collect function that will scrape data from the given listed urls which are passed into urls. The collect function used to download the content of the provided url and print out the text of urls.



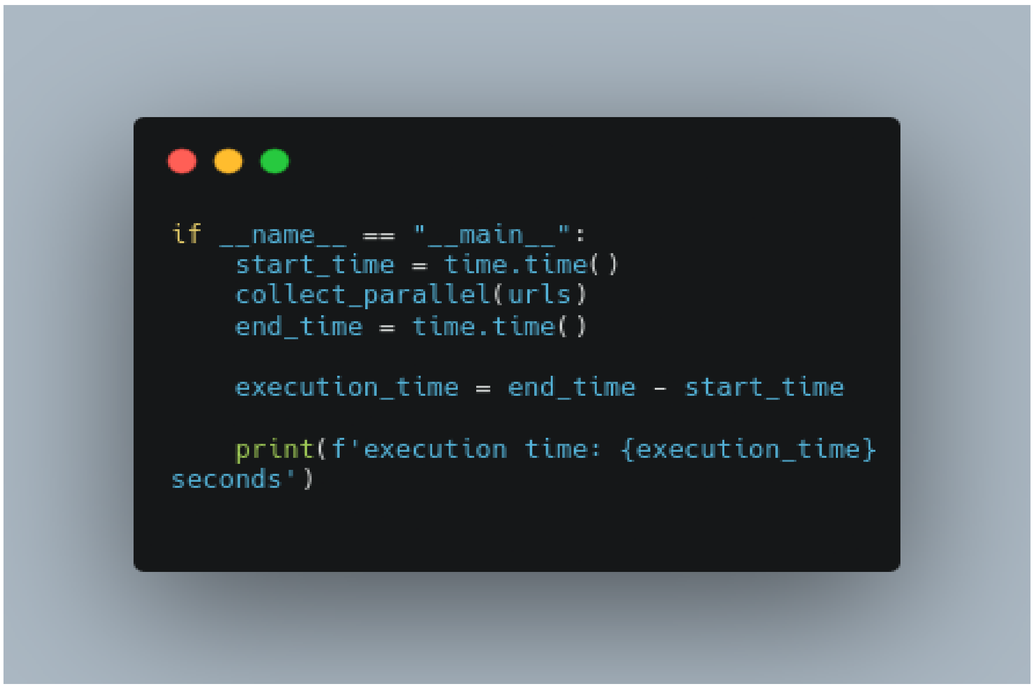
```python
def collect(urls):
    response = requests.get(urls)
    print(f'content of : {urls}:
{response.text[:50]}')


urls = [
    'https://example.com/page1',
    'https://example.com/page2',
    'https://example.com/page3',
]
```

After that, collect_paraller function is create and urls are passed as paramters which include multiprocessing method. It creates a pool of workers processes. The map method applies to the scrape data function to each url in the urls list, distributing the tasks among the worker processes in the pool.



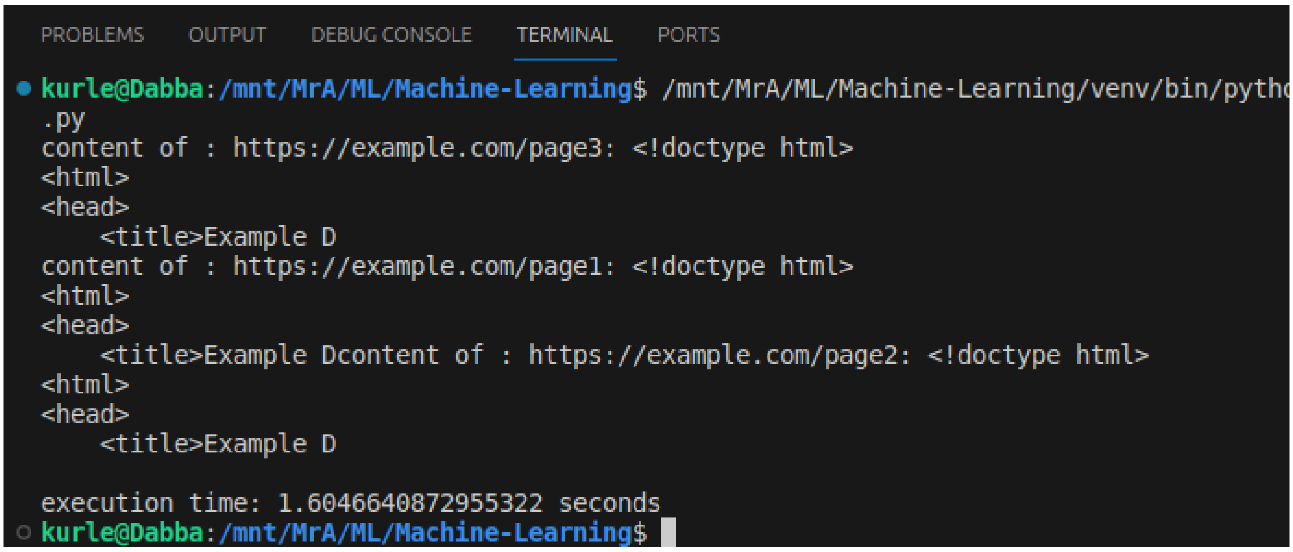
```python
def collect_parallel(urls):
    with multiprocessing.Pool() as pool:
        pool.map(collect,urls)
```

The output of the program can be taken by call this collect_parallel function and also the execution time is measured to show the difference of the process.



```python
if __name__ == "__main__":
    start_time = time.time()
    collect_parallel(urls)
    end_time = time.time()

    execution_time = end_time - start_time

    print(f'execution time: {execution_time}
seconds')
```

Final, output with execution-time:



```
● kurle@Dabba:/mnt/MrA/ML/Machine-Learning$ /mnt/MrA/ML/Machine-Learning/venv/bin/pytho
  .py
  content of : https://example.com/page3: <!doctype html>
  <html>
  <head>
      <title>Example D
  content of : https://example.com/page1: <!doctype html>
  <html>
  <head>
      <title>Example Dcontent of : https://example.com/page2: <!doctype html>
  <html>
  <head>
      <title>Example D

  execution time: 1.6046640872955322 seconds
○ kurle@Dabba:/mnt/MrA/ML/Machine-Learning$ 
```