Concurrency in Python: Multithreading and Multiprocessing

Introduction

Concurrency is a concept that allows multiple tasks to run simultaneously. In Python, we can achieve concurrency using two main approaches: multithreading and multiprocessing.

Multithreading with ThreadPoolExecutor

The ThreadPoolExecutor is part of the concurrent.futures module, which provides a high-level interface for asynchronously executing callables. It allows you to manage a pool of threads and submit tasks to be executed in parallel.

```
from concurrent.futures import ThreadPoolExecutor
import time

def print_number(number):
    time.sleep(1)
    return f"Number: {number}"

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3]

with ThreadPoolExecutor(max_workers=3) as executor:
    results = executor.map(print_number, numbers)

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

In this code snippet, we define a function `print_number` that simulates a delay using `time.sleep(1)` and returns a formatted string. We create a list of numbers and use a ThreadPoolExecutor to execute the `print_number` function concurrently. The `max_workers` parameter specifies the number of threads to use.

Multiprocessing with ProcessPoolExecutor

The ProcessPoolExecutor is similar to ThreadPoolExecutor but uses separate processes instead of threads. This is useful for CPU-bound tasks, as it allows you to bypass the Global Interpreter Lock (GIL).

```
from concurrent.futures import ProcessPoolExecutor
import time

def square_number(number):
    time.sleep(2)
    return f"Square: {number * number}"

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

```
if __name__ == '__main__':
    with ProcessPoolExecutor(max_workers=3) as executor:
        results = executor.map(square_number, numbers)

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

In this code snippet, we define a function `square_number` that calculates the square of a given number after a delay of 2 seconds. We create a list of numbers and use a ProcessPoolExecutor to execute the `square_number` function concurrently. The `if __ name__ == '__main__':` guard is necessary to ensure that the code runs correctly when using multiprocessing.

Conclusion

Both multithreading and multiprocessing are powerful tools in Python for achieving concurrency. Choosing between them depends on the nature of the tasks: use multithreading for I/O-bound tasks and multiprocessing for CPU-bound tasks.

References

1. Python Official Documentation: https://docs.python.org/3/library/concurrent.futures.html

2. Real Python: https://realpython.com/python-concurrency/