Ironhack Student Portal

preview.my.ironhack.com/lms/courses/course-

Lesson Goals

- Understand the concepts of sequential vs. parallel processing.
- Introduce the Python multiprocessing library.
- Produce examples of sequential and parallel processing.
- Compare performance results and observe how much faster processes finish with multiprocessing.

Introduction

When working as a data analyst or data scientist, there will be times where you will have to apply the same function or operation to a large number of items. By default, Python processes each task you tell it to perform sequentially, which can take a substantial amount of time when either the number of tasks is large, the number of items on which you need to perform the task is large, or both.

When the information you need to compute upon is large or the computation is complex, your program can take a long time to run. In cases like these, it is often useful to divide the work among the computing resources available on your machine. This is known as multiprocessing or parallelization, and it can save you significant amounts of time.

Parallelization refers to the act of performing tasks in parallel. This is made possible by the existence of multiple CPUs or cores on a computer.

Python has a `multiprocessing` library that allows you to spread the work your programs need to perform over the number of cores your machine has. In this lesson, we will learn how to parallelize our code using the multiprocessing library. Along the way, we will practice parallelizing via a variety of examples. Let's go ahead and import it so that it's ready to be used.

```
import multiprocessing
```

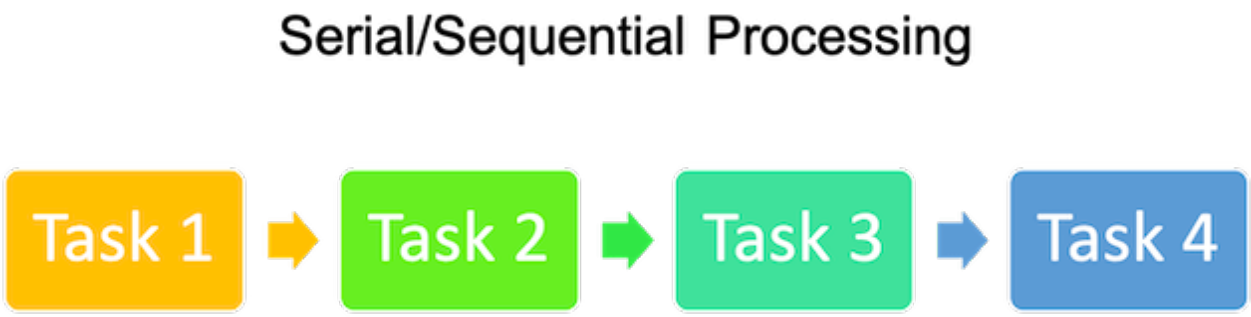
The multiprocessing library contains several tools for parallelizing your workflow, but the most commonly used is the `Pool.map` method. Just like the regular `map` function in Python, `Pool.map` applies a given function to an iterable such as a list or tuple. However, the main difference is that it does so in parallel. It is relatively straightforward to understand and should work for the majority of cases where you will need parallelization, so we will be focusing on this aspect of the multiprocessing library in this lesson.

Getting Started

The code examples in this lesson use the dataset `directories.zip` which you can download here. After downloading the file, extract the content. You will have a folder called `directories` that contains sub-folders and files.

Sequential Processing

When you perform a series of tasks in Python, it will execute them sequentially, or one at a time. When a task is performed this way, a subsequent computation does not occur until the one before it is complete.



For example, let's suppose you had a function that would square any number and a list of numbers between 0 and 99 that you wanted to square.

```
def square(x):
    return x*x

data = [x for x in range(100)]
```

You could get a list containing the square of each number by applying the `square` function to each number in the list and enclosing it in a list comprehension.

```
%%time
seq = [square(x) for x in data]
print(seq)
```

```
    [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484,
529, 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025, 3136,
3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184,
5329, 5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744,
7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]
CPU times: user 230 µs, sys: 63 µs, total: 293 µs
Wall time: 264 µs
```

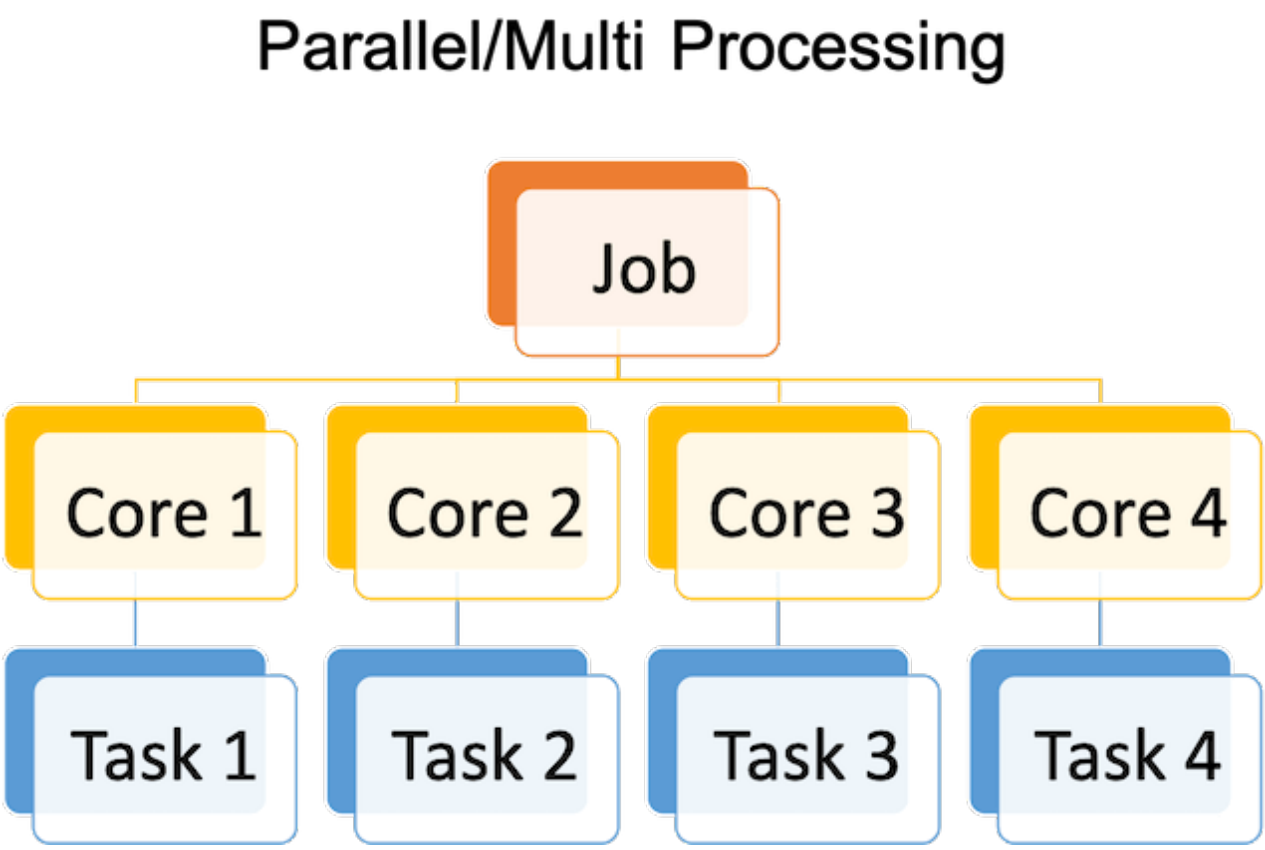
In this example, Python applied the function to each list element sequentially. In other words, first it squared 0, then it squared 1, then it squared 2, etc. It took a total of 293

milliseconds to run through the list squaring each number and then printing then printing the entire list.

Side Note: Note the use of the `%%time` magic command. When used at the beginning of a Jupyter Notebook cell, it calculates the amount of time it takes for the contents of that cell to run. This will help us compare execution times between sequentially and parallel processed cells.

Parallel Processing

If we wanted to speed up the squaring of these numbers, we could parallelize the operation. Doing so would split the job we have to do across the number of cores our machine has (ex. 4 cores) and assign each core a task that it would perform at the same time as the other cores.



To do this in Python, all we would have to do is call the `Pool` class from the multiprocessing library and then simply `map` the function to the data. This would apply the function in a parallel fashion across however many cores your computer has at a time. Once that has happened, we call the `terminate` method to stop and shutdown the job and then the `join` method to prevent any "zombie" processes that may result from a child process continuing after a parent process has terminated.

```
%%time
pool = multiprocessing.Pool()
result = pool.map(square, data)
pool.terminate()
pool.join()
print(result)
```

```
 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361, 400, 441, 484,
529, 576, 625, 676, 729, 784, 841, 900, 961, 1024, 1089, 1156, 1225, 1296, 1369, 1444, 1521, 1600,
1681, 1764, 1849, 1936, 2025, 2116, 2209, 2304, 2401, 2500, 2601, 2704, 2809, 2916, 3025, 3136,
3249, 3364, 3481, 3600, 3721, 3844, 3969, 4096, 4225, 4356, 4489, 4624, 4761, 4900, 5041, 5184,
5329, 5476, 5625, 5776, 5929, 6084, 6241, 6400, 6561, 6724, 6889, 7056, 7225, 7396, 7569, 7744,
7921, 8100, 8281, 8464, 8649, 8836, 9025, 9216, 9409, 9604, 9801]
CPU times: user 16.5 ms, sys: 19.9 ms, total: 36.4 ms
Wall time: 40.3 ms
```

We can see that this took a total of 36.4 milliseconds, which means it finished in 3.4% of the time it took to perform the task sequentially.

Use Case: File System Traversal

The example in the previous section was a simple one intended to show how easy it can be to parallelize operations using Python's multiprocessing library. In this section, we will take a look at a real-world use case where parallelization would be helpful and see how we can use the previous example as a guide in parallelizing our tasks.

Suppose we had a directory with a hundred folders and hundreds of files in each folder. There are various file types contained within each folder, and we are specifically interested in knowing how many markdown files are in our directory and where they are located. These files can be identified by a .md file extension.

To start, we will get a list of the directories using the `listdir` method from the `os` library. We are going to remove a hidden file called .DS_Store that was automatically created in the directory because it is not a folder and we want just a list of folders that we can search through.

```
import os

dirs = os.listdir('./directories')
dirs.remove('.DS_Store')
```

This list is an iterable, much like the list of numbers we used in the previous example.

Next, we will define a function that will search through a given directory folder and extract a list of files that end with the .md file extension.

```
def find_markdowns(dir):
    files = os.listdir(dir)
    markdowns = [dir +'/' + file for file in files if file.endswith('.md')]
    return markdowns
```

This function can be applied to the iterable list just like our `square` function could be applied to the list of numbers in our earlier example.

Let's apply this function to our directory list in a sequential fashion, time it the execution, and see how fast it can identify all the markdown files. To do this, we will apply the function inside a list comprehension to every directory in our directory list. This will produce a list of lists, which we will then flatten into a single list containing the location of all the markdown files as elements.

```
%%time
result = [find_markdowns(dir) for dir in dirs]
flatten = [file for sublist in result for file in sublist]
```

```
    CPU times: user 22.2 ms, sys: 49.3 ms, total: 71.4 ms
Wall time: 70.8 ms
```

We can see that the total time it took for this process to run sequentially was 71.4 milliseconds.

Next, let's see if we can achieve any performance gains by applying the function to the iterable in parrallel. As before, we are going to call the `Pool` class and then map the function to the iterable. We are also going to flatten the results here just like we did for the sequential run.

```
%%time
pool = multiprocessing.Pool()
result = pool.map(find_markdowns, dirs)
pool.terminate()
pool.join()
flatten = [file for sublist in result for file in sublist]
```

```
    CPU times: user 15.5 ms, sys: 23.1 ms, total: 38.6 ms
Wall time: 133 ms
```

This time, it only took 38.6 milliseconds to run, which is almost twice as fast as it took to run the process sequentially.

Summary

In this lesson, we introduced the concepts of sequential and parallel processing. We looked at a basic example of sequential processing and then parallel processing using

the Python multiprocessing library and witnessed a significant improvement in performance using multiprocessing. We then used what we learned from the basic example and applied it to a real-world use case where we wanted to extract a certain type of file from a directory with a hundred folders and hundreds of files in each folder. We saw first-hand the performance gains that are possible with parallelization. As you work through the rest of the program, keep these concepts in mind and try to think about ways to implement multiprocessing when you have a time-consuming process that involves a large amount of data, very complex computations, or both.