Exercise 4

1. Screen your code for optimization potential

1.1 Test which parts take the most time

Using the time taking functions from the previous exercise, find out which part of your script takes the longest. Make sure your toy example is large enough, that the longest section runs a few seconds (e.g., 10sec). Otherwise, it'll be hard to make differences visible.

```
from datetime import datetime

# take start time
start_execution_timestamp = datetime.now()

a_long_running_task() # replace this with your actual code

# take end time
end_execution_timestamp = datetime.now()
time_diff = end_execution_timestamp - start_execution_timestamp

# print status
print(f"this script took {time_diff}sec to execute")
```

Measuring runtime is part of a process called "profiling". There are more elaborate ways to do so, but for this context, this will do the trick. If you're interested, search for "Python profiling" on the internet to learn more about this topic.

1.2 Check if you run the same code repeatedly

Go through your code line by line and decide if a line gets called multiple time, e.g., in a function or a loop. If so, what does the line do? Can you spot parts that are called repeatedly, but the arguments always stay the same? If nothing changes, can the code be refactored, so that the line gets called only once? Take a look at the following example:

```
for index, row in dataframe.iterrows():
    other_dataset = load(filename)
    dataframe.loc[index, 'new_value'] = other_dataset[row.value]
```

In this code example, every row of the `dataframe` gets a new value that originates from `other_dataset`. While this would work, the `other_dataset` is loaded every time, for each row of the `dataframe`. This causes unnecessary I/O operations and slows down the execution of the code. Consider the following change:

```
other_dataset = load(filename)
for index, row in dataframe.iterrows():
    dataframe.loc[index, 'new_value'] = other_dataset[row.value]
```

Here, the result is the same. However, `other_dataset` is loaded only once, in the beginning of the code. Now, each iteration of the loop accesses the variable once, but the `other_dataset` is not loaded repeatedly. This works, because the content of `other_dataset` is not altered between iterations.

Optional suggestion: One instance where this logic could be applied is the section that computes the right index from the county coordinates for the NOAA CPC dataset. Although the same counties are used multiple times (once for each of the 365 days), the indexes are computed every time from scratch. Here, you could, for example, cache each index conversion for each county, so it needs to be computed only once.

1.3 Think about parallelization

As discussed in the lecture, parallelization is probably the most valuable concept to embrace. For you start, ask yourself which parts could profit from parallel execution? One easy answer is: **every time the same code is run on different, independent parts of data.** Let's understand what this means using an example:

In chapter 2, we augmented every sentiment score with a precipitation value. This process is repeated several thousand times, and each row is independent of another. However, the code to augment one row (read the county name, lookup the coordinates, and query the precipitation dataset) is always the same. In this case, it's easy to split the dataset into multiple parts and have different CPU cores work on different subsets.

2. Implement parallelization using joblib

Try parallelization for your own script. There are many libraries to assist you with this, from the native multiprocessing library, to ray, which allows you to easily distribute your code across multiple nodes. You can choose yourself how you want to do it, but we suggest trying `joblib`. It's an easy framework for working with multiple processes in Python. You'll find the documentation here: https://joblib.readthedocs.io/en/stable/, and a mini example in the course repo, called `joblib example.py`.

The central idea is to create a list of jobs in for of delayed function calls (function calls that are not executed right away, but only store the function call for later execution in an object). This list of calls can then be distributed across different processes. The operating system takes care that these processes are distributed across different CPU cores. Note: if you work on the FASRC, make sure your SLURM session requested multiple CPU cores, e.g., the command

```
srun --pty -p test --mem 1000 -t 0-01:00 /bin/bash
```

will only provide access to a single CPU core. Your script might spawn multiple processes, but you won't be able to leverage the multicore system. To request access to more CPU cores, add an argument to the SLURM request:

```
srun --pty -p test --mem 1000 -c 4 -t 0-01:00 /bin/bash
```

Tipp: When you are experimenting with using more resources, adjust your toy example to be large enough for differences to show up; e.g., run the analysis on data of the entire U.S. instead of only one or two states.

3. Run the precipitation analysis for the entire world

If you are confident your script runs quickly and can make use of the FASRC resources, try computing the sentiment change between rainy and non-rainy days on the data of the whole world for one entire year.

If that worked well, you can think of how to make use of multiple nodes, too. A very simple way could be starting several SLURM jobs, each for one year. Can you come up with a more elaborate way?

Which country is the most weather sensitive? Of one year? Of the last decade?

4. Further improvements

If you made it to this point but want to practice some more, here are a few ideas what you could improve next:

- Drop days with only very few tweets, e.g., having at least 10. Because if a county has only very few days for one day, the sentiment score is likely to be skewed, as it only depends on very few people.
- Similarly, you could include only such counties in your analysis, that have at least 20 rainy days. The rational is the same; if this number is very low, the difference between rainy and non-rainy depends on too few examples and is likely to be biased.
- Add a statistical analysis, such as effect sizes of the difference (e.g., Cohen's *d*), or significance testing (e.g., Welch's *p*-test). There are entire lectures to be held on multiple significance testing, but for the purpose of this workshop, it might be an interesting practice to explore this.

Probably the most complex addition is to only compare days with at least three days rain / no rain in a row. The idea here is that a single day of rainfall surely won't have a huge impact on people's mood. However, seeing only a dull, gray sky for days might have a stronger effect on how they feel. Hence, change the analysis procedure as such that it only considers those days, that have the same rain-condition for three or more consecutive days.

Try some or all of these additions yourself. If you want some inspiration, the file `global_precipitation_sentiment.py` implements all these ideas.

What you learned in this exercise:

- Identifying performance issues in your own code
- Applying parallelization to use multiple CPU cores at once
- How such performance improvements allow you to make use of the FASRC resources
- You learned some data science and analysis methods, and how to implement them, to draw more robust conclusions from your data