02613 Python and High Performance Computing

Exercises for Week 7

For these exercises, we will use weather data from public DMI (Danmarks Metereologiske Institut) from January 2023. You can read more about DMI's free data (in Danish) here. For information about the data records we will be working with (in English), see the official documentation. We have prepared the relevant data in a CSV file you can find at `/dtu/projects/02613_2025/data/dmi/2023_01.csv.zip`.

Each row of the CSV file represents a measurement from one of the measurement stations DMI operates in Denmark, Greenland and the Faroe Islands. The stations measure several different parameters including wind speed, precipitation, humidity, etc., and each row is a measurement of one parameter. The columns are:

1. `coordsx` First coordinate of measurement station.
2. `coordsy` Second coordinate of measurement station.
3. `created` Time measurement was entered in data base.
4. `observed` Time measurement was observed.
5. `parameterId` Measured parameter. See also the official documentation.
6. `stationId` ID of measurement station.
7. `value` Measured value.

1. Storage and Reading Files with Pandas

The goal of this exercise is to compare different method of loading and representing dataframes with pure Pandas.

1. The CSV file is stored as a zip file. Compare the time it takes for the following two approaches:

   a) First unpacking the zip file with the `unzip` command and then reading the CSV file with `read_csv`.

   b) Read the zip directly with `read_csv`.

   Which one is faster?

2. **Autolab** Load the data to a pandas dataframe (use `pd.read_csv()`). How much memory does the data frame occupy? Create a function `df_memsize` that takes a Pandas DataFrame as input and returns its size in bytes.

3. Check the columns and list all the changes you can do to save memory. Hint: check sections 7.1.2 and 7.1.3 on Fast Python for inspiration. You can use the following function summarize the dataframe:

```
def summarize_columns(df):
print(pd.DataFrame([
    (
        c,
        df[c].dtype,
        len(df[c].unique()),
        df[c].memory_usage(deep=True) // (1024**2)
    ) for c in df.columns
], columns=['name', 'dtype', 'unique', 'size (MB)']))
print('Total size:', df.memory_usage(deep=True).sum() / 1024**2, '
    MB')
```

4. **Autolab** Perform the changes to reduce the memory usage. What is the new size of the dataframe? Create a function `reduce_dmi_df` that takes a Pandas DataFrame as input and returns a transformed DataFrame where at least 3 columns have had their memory use reduced.

2. Reading Files with Arrow

The goal of this exercise is to compare the time and memory size we get when using PyArrow to read the files instead of pandas. Check section 7.4 on Fast Python for help on this exercise.

1. **Autolab** Load the DMI data from January 2023, as we did in the last exercise, but this time using PyArrow. Compute the time it took to read the CSV, what was the speed up when comparing with pandas? Create a function `pyarrow_load` that takes a path to a CSV file as input, loads the file with PyArrow and returns the PyArrow table.
2. **Autolab** Now convert the PyArrow table to a pandas data frame, how long does it take? Is the total time (time to load the data with PyArrow + the time to convert it to a pandas data frame) faster or slower than pure Pandas? Modify your previous function to return a Pandas DataFrame instead of a PyArrow table.
3. What was the size of the PyArrow table you got after loading the data? What was the size of the data frame after the conversion? Why are they different?
4. Modify your loading code in order to perform (as many as possible of) the memory reducing operations you did in exercise 1.4. What is the new size of the PyArrow table?

3. Parquet Files

Parquet files offer a highly efficient representation for columnar data, optimizing storage, and query performance in data processing workflows. The data is stored in a binary file allowing to represent numbers in a much more compactly form than in text, as it is done for csv files. In this exercise we will explore the advantages of using Parquet files to store data. Check section 8.2 in Fast Python for help on this exercise.

1. **Autolab** Make a Python program that receives the path to a CSV file as a command line argument, loads the CSV file and finally saves it again as a Parquet file. What is the difference in size between the original CSV file and the Parquet file?
2. Measure the times to read and write the DMI data to a parquet files and compare it to reading and writing CSV.

4. Pandas - Fast Operations

Our goal with this exercise is to compute the total amount of precipitation (i.e., rain) across all measurement stations. We will do this by summing all records with `parameterId` equal to `precip_past10min`. These hold the total precipitation in kg/m^2 or mm for this station over the past 10 minutes. Check sections 7.2.2 and 7.3.1 on Fast Python for help on this exercise.

A raw python implementation (i.e., not using pandas- or NumPy-based approaches) for could be:

```
def total_precip(df):
    total = 0.0
    for i in range(len(df)):
        row = df.iloc[i]
        if row['parameterId'] == 'precip_past10min':
            total += row['value']
    return total
```

1. Run the raw python implementation and time it. *Note:* Use the Pandas DataFrame method `sample` to subsample a smaller DataFrame, to avoid exessive wait times. How long did it take to execute?
2. Build a pandas-based implementation using the `apply()` method to perform the same operation and time it. What was the speed up?
3. Now write a pure Pandas version using a vectorized approach. How long did it take to execute? What was the speed up in comparison with the two previous implementations?
4. **Autolab** Turn your previous implementation into a Python program that receives the path to a CSV file as a command line argument, loads the CSV, computes the total precipitation and finally

prints the result (and only the result).

5. A significant portion of the work in the vectorized version is doing the indexing to extract the relevant rows. Rewrite the vectorized version to first use the `parameterId` column as an index. Excluding the time it takes to build the index, how long does it take to compute the total precipitation now? Including the time to build the index? When would one prefer one to the other? Hint: see section 7.2.1 in Fast Python.