Big Data Exercises

In these exercises we will work on data from a series of global weather monitoring stations used to measure climate trends to examine long-term trends in temperature for your home locality. This data comes from the Global Historical Climatology Network, and is the actual raw data provided by NOAA. The only changes I have made to this data are a few small formatting changes to help meet the learning goals of this exercise.

To do these excercises, first please download the data for this exercise from here. Note this is a big file (this is a big-data exercise, after all), so be patient.

(1) The data we'll be working with can be found in the file `ghcnd_daily.tar.gz` . It includes daily weather data from thousands of weather stations around the work over many decades.

Begin by unzipping the file and checking it's size -- it should come out to be *about* 4gb, but will expand to about 12 gb in RAM, which means there's just no way most students (who usually have, at most, 16gb of RAM) can import this dataset into pandas and manipulate it directly.

(Note: what we're doing can be applied to much bigger datasets, but they sometimes takes hours to work with, so we're working with data that's just a *little* big so we can get exercises done in reasonable time).

(2) Thankfully, we aren't going to be working with *all* the data today. Instead, everyone should pick three weather stations to examine during this analysis.

To pick your stations, we'll need to open the `ghcnd-stations.txt` file in the directory you've downloaded. It includes both station codes (which is what we'll find in the `ghcnd_daily.csv` data, as well as the name and location of each station).

When picking a weather station, make sure to pick one flagged as being in either GSN, HCN, or CRN (these designate more formalized stations that have been around a long time, ensuring you'll get a station with data that has been recorded over a longer period).

Note that Station IDs start with the two-letter code of the country in which they are located, and the "NAME" column often constains city names.

The `ghcnd-stations.txt` is a "fixed-width" dataset, meaning that instead of putting commas or tabs between observations, all columns have the same width (in terms of number of characters). So to import this data you'll have to (a) read the notes about the data in the project README.txt, and (b) read about how to read in fixed-width data in pandas. When entering column specifications, remember that normal people count from 1 and include end points, while Python counts from 0 and doesn't include end points (so if the readme says data is in columns 10-20, in Python that'd be 9 through 20).

In [ ]:
```
import pandas as pd
import numpy as np
import seaborn.objects as so
import warnings

warnings.filterwarnings("ignore")

pd.set_option("mode.copy_on_write", True)
```

In [ ]:
```
# picked three weather stations: ACW00011647, AE000041196, AEM00041194
```

In [ ]:
```
colspecs = [
    (0, 12),
    (13, 21),
    (22, 31),
    (32, 38),
    (39, 41),
    (42, 72),
    (73, 76),
    (77, 80),
    (81, 86),
]
df = pd.read_fwf(
    r"C:\Users\Asia\OneDrive\Pulpit\DUKE\Practicing Data Science\global_climate_dat
    colspecs=colspecs,
    names=[
        "ID",
        "LATITUDE",
        "LONGITUDE",
        "ELEVATION",
        "STATE",
        "NAME",
        "GSN FLAG",
        "HCN/CRN FLAG",
        "WMO ID",
    ],
)
```

In [ ]:
```
df.head()
```

Out[ ]:

	ID	LATITUDE	LONGITUDE	ELEVATION	STATE	NAME	GSN FLAG	HCN/CRN FLAG
0	ACW00011604	17.1167	-61.7833	10.1	NaN	T JOHNS COOLIDGE FLD	NaN	NaN
1	ACW00011647	17.1333	-61.7833	19.2	NaN	T JOHNS	NaN	NaN
2	AE000041196	25.3330	55.5170	34.0	NaN	HARJAH INTER. AIRP	SN	NaN
3	AEM00041194	25.2550	55.3640	10.4	NaN	UBAI INTL	NaN	NaN
4	AEM00041217	24.4330	54.6510	26.8	NaN	BU DHABI INTL	NaN	NaN

(3) Now that we something about the observations we want to work with, we can now turn to our actual weather data.

Our daily weather can be found in `ghcnd_daily.csv` , which you get by unzipping `ghcnd_daily.tar.gz` . Note that the README.txt talks about this being a fixed-width file. Since you've already dealt with one fixed-width file, I've just converted this to a CSV, and dropped all the data that isn't "daily max temperatures".

Let's start with the fun part. **SAVE YOUR NOTEBOOK AND ANY OTHER OPEN FILES!**. Then just try and import the data (`ghcnd_daily.csv`) while watching your Activity Monitor (Mac) or Resource Monitor (Windows) to see what happens.

If you have 8GB of RAM, this should fail miserably.

If you have 16GB of RAM, you might just get away with this. But if it *does* load, try sorting the data by year and see how things go.

(If you have 32GB of RAM: you're actually probably fine with data this size. Sorry -- datasets big enough to cause big problems for people with 32GB take a long time to chunk on an 8GB computer, and these exercises have to be fast enough to finish in a class period! There are some exercises at the bottom with a REALLY big dataset you can work with.)

You may have to kill your kernel, kill VS Code, and start over when this explodes...

In [ ]:

```
# daily_df = pd.read_csv(
#     r"C:\Users\Asia\OneDrive\Pulpit\DUKE\Practicing Data Science\global_climate_d
# )
```

(4) Now that we know that we can't work with this directly, it's good with these big datasets to just import ~200 lines so you can get a feel for the data. So load *just 200 lines* of `ghcnd_daily.csv` .

In [ ]:
```
daily_df = pd.read_csv(
    r"C:\Users\Asia\OneDrive\Pulpit\DUKE\Practicing Data Science\global_climate_dat
    nrows=200,
)
```

In [ ]:
```
daily_df.head()
```

Out[ ]:

	id	year	month	element	value1	mflag1	qflag1	sflag1	value2	mflag2
0	ACW00011604	1949	1	TMAX	289	NaN	NaN	X	289	NaN
1	ACW00011604	1949	2	TMAX	267	NaN	NaN	X	278	NaN
2	ACW00011604	1949	3	TMAX	272	NaN	NaN	X	289	NaN
3	ACW00011604	1949	4	TMAX	278	NaN	NaN	X	283	NaN
4	ACW00011604	1949	5	TMAX	283	NaN	NaN	X	283	NaN

5 rows × 128 columns

(5) Once you have a sense of the data, write code to chunk your data: i.e. code that reads in all blocks of the data that will fit in ram, keeps only the observations for the weather stations you've selected to focus on, and throws away everything else.

In addition to your own three weather stations, please also include station USC00050848 (a weather station from near my home!) so you can generate results that we can all compare (to check for accuracy).

Note you will probably have to play with your chunk sizes (probably while watching your RAM usage?). That's because small chunk sizes, while useful for debugging, are very slow.

Every time Python processes a chunk, there's a fixed processing cost, so in a dataset with, say, 10,000,000 rows, if you try to do chunks of 100 rows, that fixed processing cost has to be paid 100,000 times. Given that, the larger you can make your chunks the better, so long as your chunks don't use up all your RAM. Again, picking a chunk size then watching your RAM usage is a good way to see how close you are to the limits of your RAM.

In [ ]:
```
# ACW00011647, AE000041196, AEM00041194
our_data = r"C:\Users\Asia\OneDrive\Pulpit\DUKE\Practicing Data Science\global_clim
chunk_size = 10000
filtered_chunks = []
for chunk in pd.read_csv(our_data, chunksize=chunk_size):
    filtered_chunk = chunk[
        (chunk["id"] == "USC00050848")
        | (chunk["id"] == "ACW00011647")
        | (chunk["id"] == "AE000041196")
        | (chunk["id"] == "AEM00041194")
    ]
    filtered_chunks.append(filtered_chunk)

filtered_df = pd.concat(filtered_chunks)
```

In [ ]:
```
filtered_df_wo_missing = filtered_df.replace(-9999, np.nan)
filtered_df_wo_missing.head(20)
# print(len(filtered_df))
```

(6) Now, for each weather station, figure out the *earliest* year with data. Keep `USC00050848` and the two of the three weather stations you picked with the best data (i.e., you should have 3 total, two you picked and `USC00050848`).

In [ ]:
```
nick_site = filtered_df_wo_missing[
    filtered_df_wo_missing["id"] == "USC00050848"
].sort_values(by="year")
print(f"The earliest year with data for Nick's site is 1893.")
nick_site.head(5)
```

```
The earliest year with data for Nick's site is 1893.
```

Out[ ]:

	id	year	month	element	value1	mflag1	qflag1	sflag1	value2	m
7985564	USC00050848	1893	10	TMAX	61.0	NaN	NaN	6	139.0	
7985565	USC00050848	1893	11	TMAX	189.0	NaN	NaN	6	78.0	
7985566	USC00050848	1893	12	TMAX	167.0	NaN	NaN	6	133.0	
7985567	USC00050848	1894	1	TMAX	128.0	NaN	NaN	6	122.0	
7985568	USC00050848	1894	2	TMAX	-11.0	NaN	NaN	6	44.0	

5 rows × 128 columns

In [ ]:
```
st_john = filtered_df_wo_missing[
    filtered_df_wo_missing["id"] == "ACW00011647"
].sort_values(by="year")
print(f"The earliest year with data for St John's site is 1961.")
st_john.head(5)
```

```
The earliest year with data for St John's site is 1961.
```

Out[ ]:

	id	year	month	element	value1	mflag1	qflag1	sflag1	value2	mflag2
7	ACW00011647	1961	10	TMAX	272.0	NaN	NaN	X	NaN	NaN

1 rows × 128 columns

In [ ]:
```
sharjah = filtered_df_wo_missing[
    filtered_df_wo_missing["id"] == "AE000041196"
].sort_values(by="year")
print(f"The earliest year with data for Sharjah airport site is 1944.")
sharjah.head()
```

```
The earliest year with data for Sharjah airport site is 1944.
```

Out[ ]:

	id	year	month	element	value1	mflag1	qflag1	sflag1	value2	mflag2
8	AE000041196	1944	3	TMAX	NaN	NaN	NaN	NaN	NaN	NaN
9	AE000041196	1944	4	TMAX	258.0	NaN	NaN	I	263.0	NaN
10	AE000041196	1944	5	TMAX	335.0	NaN	NaN	I	363.0	NaN
11	AE000041196	1944	6	TMAX	374.0	NaN	NaN	I	396.0	NaN
12	AE000041196	1944	7	TMAX	396.0	NaN	NaN	I	380.0	NaN

5 rows × 128 columns

In [ ]:
```
dubai = filtered_df_wo_missing[
    filtered_df_wo_missing["id"] == "AEM00041194"
].sort_values(by="year")
print(f"The earliest year with data for Dubai airport site is 1983.")
dubai.head()
```

```
The earliest year with data for Dubai airport site is 1983.
```

Out[ ]:

	id	year	month	element	value1	mflag1	qflag1	sflag1	value2	mflag2
648	AEM00041194	1983	1	TMAX	276.0	NaN	NaN	S	302.0	NaN
659	AEM00041194	1983	12	TMAX	294.0	NaN	NaN	S	282.0	NaN
658	AEM00041194	1983	11	TMAX	335.0	NaN	NaN	S	NaN	NaN
657	AEM00041194	1983	10	TMAX	369.0	NaN	NaN	S	375.0	NaN
655	AEM00041194	1983	8	TMAX	378.0	NaN	NaN	S	400.0	NaN

5 rows × 128 columns

(7) Now calculate the average max temp for each weather station / month in the data. Note that in a few weeks, we'll have the skills to do this by reshaping our data so each row is a single day, rather than a month. But for the moment, just sum the columns, watching out for weird values.

To sum across the value columns, we can combine:

`weather_data.filter(like='value')`
(to just get the columns whose names start with "value") with `.mean(axis='columns')`
(which averages across columns (along rows) rather than the usual averaging across rows (along columns).

In [ ]:
```
# we choose stations 'dubai' and 'sharjah' for analysis

# DUBAI
dubai["Avg monthly temp"] = dubai.filter(like="value").mean(axis="columns")
dubai["Avg monthly temp in C"] = dubai["Avg monthly temp"] / 10
dubai.head()
```

Out[ ]:

	id	year	month	element	value1	mflag1	qflag1	sflag1	value2	mflag2
648	AEM00041194	1983	1	TMAX	276.0	NaN	NaN	S	302.0	NaN
659	AEM00041194	1983	12	TMAX	294.0	NaN	NaN	S	282.0	NaN
658	AEM00041194	1983	11	TMAX	335.0	NaN	NaN	S	NaN	NaN
657	AEM00041194	1983	10	TMAX	369.0	NaN	NaN	S	375.0	NaN
655	AEM00041194	1983	8	TMAX	378.0	NaN	NaN	S	400.0	NaN

5 rows × 130 columns

In [ ]:
```
# SHARJAH
sharjah["Avg monthly temp"] = sharjah.filter(like="value").mean(axis="columns")
sharjah["Avg monthly temp in C"] = sharjah["Avg monthly temp"] / 10
sharjah.head()
```

Out[ ]:

	id	year	month	element	value1	mflag1	qflag1	sflag1	value2	mflag2
8	AE000041196	1944	3	TMAX	NaN	NaN	NaN	NaN	NaN	NaN
9	AE000041196	1944	4	TMAX	258.0	NaN	NaN	I	263.0	NaN
10	AE000041196	1944	5	TMAX	335.0	NaN	NaN	I	363.0	NaN
11	AE000041196	1944	6	TMAX	374.0	NaN	NaN	I	396.0	NaN
12	AE000041196	1944	7	TMAX	396.0	NaN	NaN	I	380.0	NaN

5 rows × 130 columns

In [ ]:
```
# NICK's
```

```
nick_site["Avg monthly temp"] = nick_site.filter(like="value").mean(axis="columns")
nick_site["Avg monthly temp in C"] = nick_site["Avg monthly temp"] / 10
nick_site.head()
```

Out[ ]:

	id	year	month	element	value1	mflag1	qflag1	sflag1	value2	m
7985564	USC00050848	1893	10	TMAX	61.0	NaN	NaN	6	139.0	
7985565	USC00050848	1893	11	TMAX	189.0	NaN	NaN	6	78.0	
7985566	USC00050848	1893	12	TMAX	167.0	NaN	NaN	6	133.0	
7985567	USC00050848	1894	1	TMAX	128.0	NaN	NaN	6	122.0	
7985568	USC00050848	1894	2	TMAX	-11.0	NaN	NaN	6	44.0	

5 rows × 130 columns

(6) Now for each weather station, generate a separate plot of the daily temperatures over time. You should end up with a plot that looks something like this:

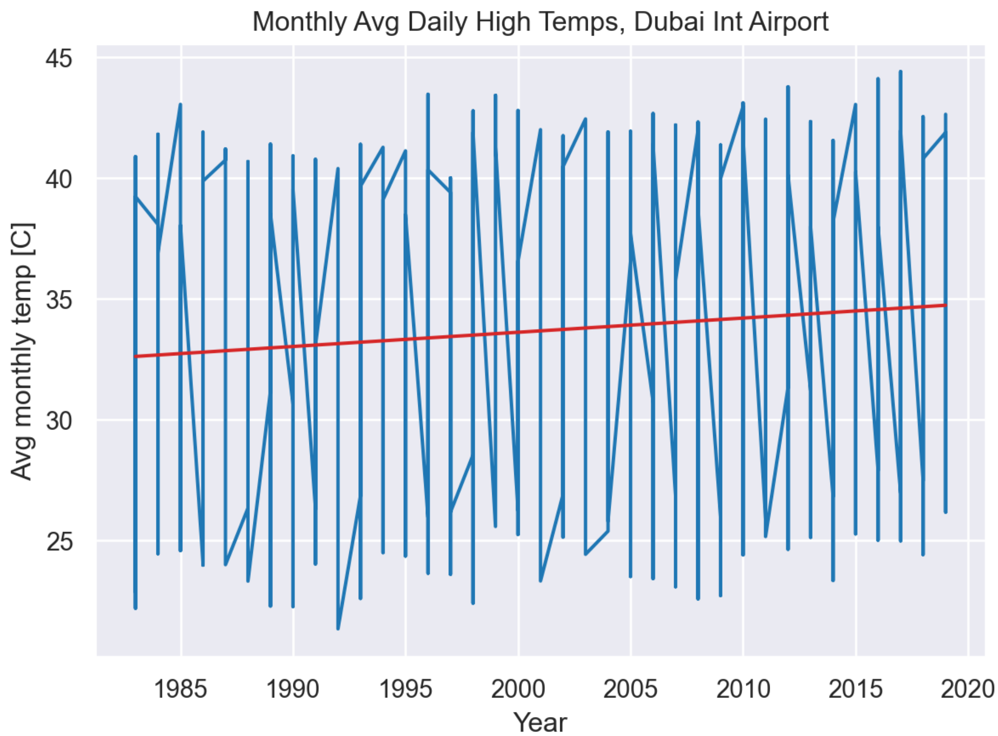
NOTE: If your plot has little horizontal lines at the tops and bottoms of the temperature plots connecting perfectly vertical temperature lines, it means you made a mistake in how you plotted your data!

In [ ]:

```
# DUBAI

so.Plot(
    dubai,
    y="Avg monthly temp in C",
    x="year",
).add(so.Line(color="tab:blue")).label(
    x="Year",
    y="Avg monthly temp [C]",
    title=f"Monthly Avg Daily High Temps, Dubai Int Airport",
).add(
    so.Line(color="tab:red"), so.PolyFit(order=1)
)
```

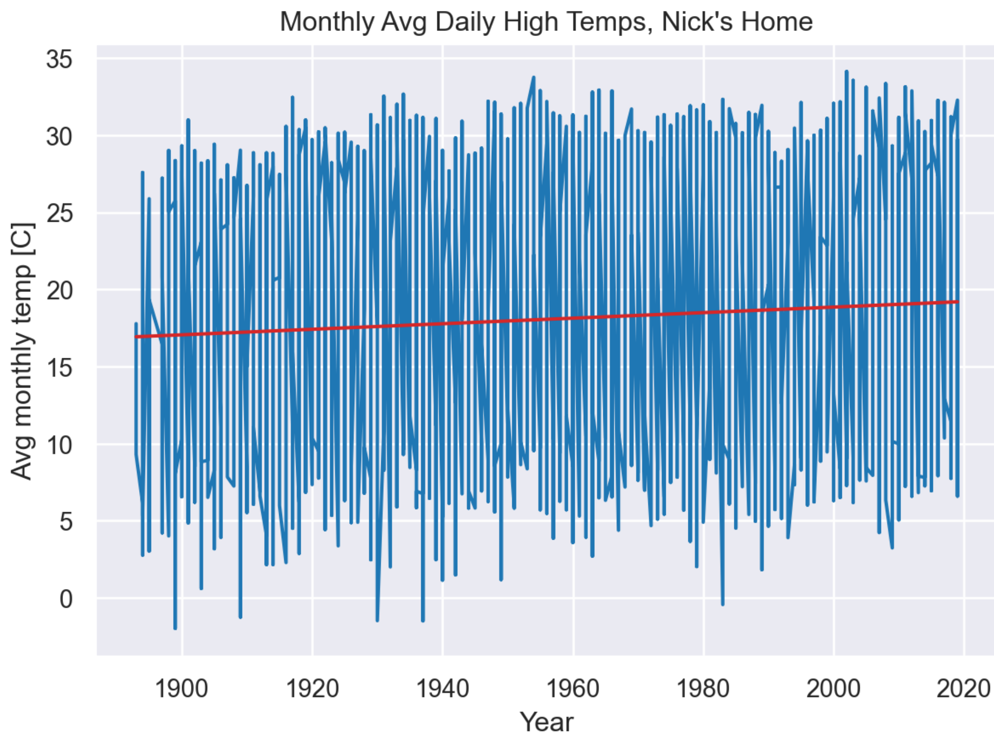
Out[ ]:



In [ ]:

```
# NICK SIDE
so.Plot(
    nick_site,
    y="Avg monthly temp in C",
    x="year",
).add(so.Line(color="tab:blue")).label(
    x="Year",
    y="Avg monthly temp [C]",
    title=f"Monthly Avg Daily High Temps, Nick's Home",
).add(
    so.Line(color="tab:red"), so.PolyFit(order=1)
)
```

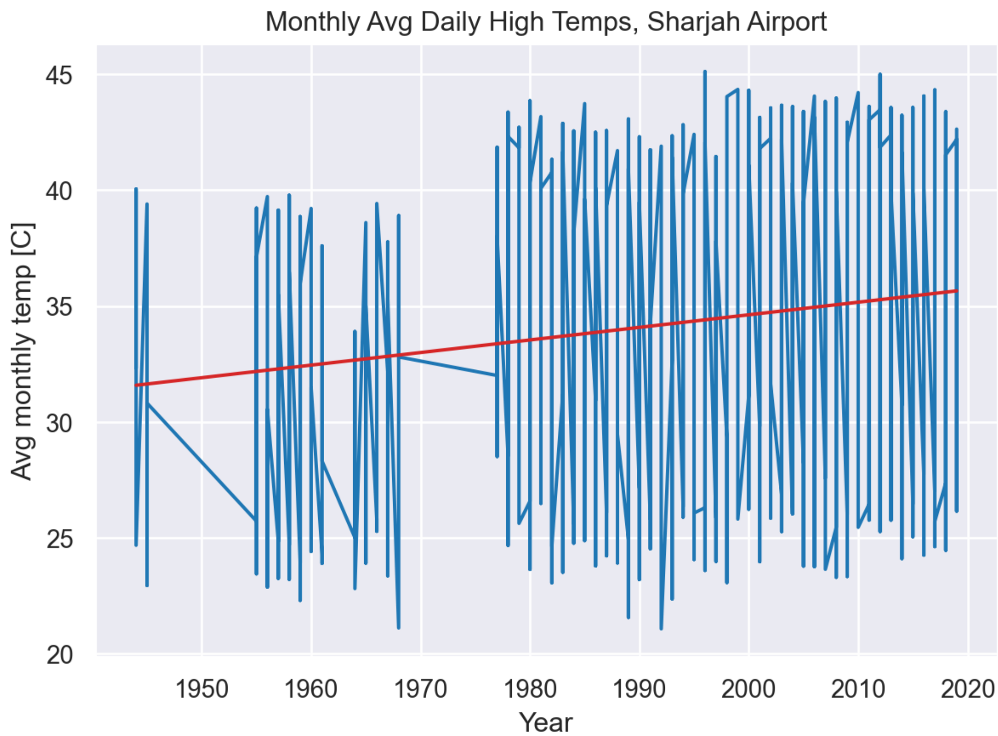
Out[ ]:



In [ ]:

```
# SHARJAH
so.Plot(
    sharjah,
    y="Avg monthly temp in C",
    x="year",
).add(so.Line(color="tab:blue")).label(
    x="Year",
    y="Avg monthly temp [C]",
    title=f"Monthly Avg Daily High Temps, Sharjah Airport",
).add(
    so.Line(color="tab:red"), so.PolyFit(order=1)
)
```

Out[ ]:



Want More Practice?

If you *really* want a challenge, the file `ghcnd_daily_30gb.tar.gz` will decompress into `ghcnd_daily.dat`, the full version of the GHCND daily data. It contains not only daily high temps, but also daily low temps, preciptionation, etc. Moreover, it is still in fixed-width format, and is about 30gb in raw form.

Importing and chunking this data (with moderate optimizations) took about 2 hours on my computer.

If you're up for it, it's a great dataset to wrestling with data in weird formats and chunking.

Pro-tip: strings take up *way* more space in RAM than numbers, so some columns can be converted to keep the memory footprint of the data down.