FEATURED STORY

VISUALIZING CLIMATE CHANGE WITH PYTHON

Winfred Lam, CFA

Share this article

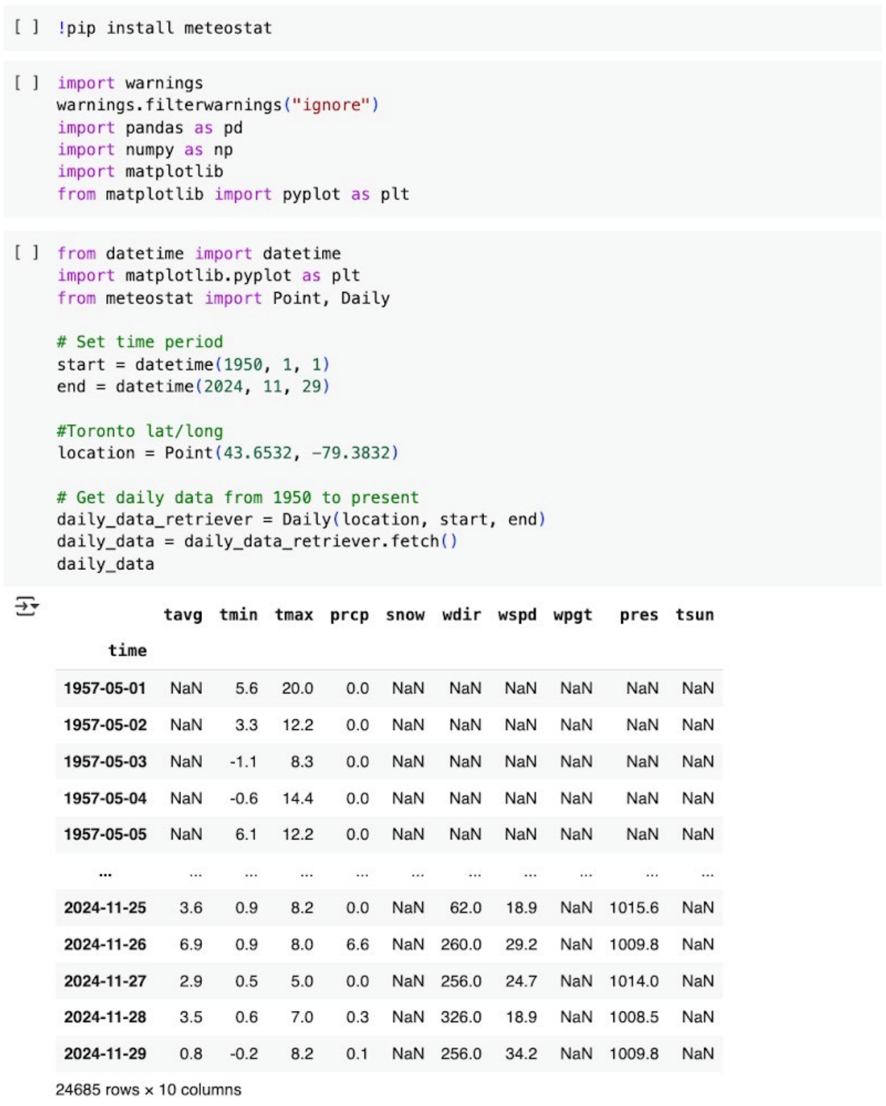
In a previous edition of *The Analyst*, we explored the power of Python's Pandas library for data analysis. Now, we're diving into the visualization and analysis of climate change trends—a topic of growing importance to financial institutions and the investment industry. This article focuses on transforming and visualizing data to unlock insights during exploratory data analysis (EDA).

Effective EDA uncovers hidden patterns and trends and highlights potential issues within data sets, making it an essential step in any analysis workflow. Additionally, we'll spotlight the JSON format—a cornerstone for storing and sharing web-based data and a key format for representing climate data.

Visualizing Toronto's temperature

We'll start by analyzing Toronto's temperature trends over time. The first step is identifying a reliable data set or an effective data application programming interface (API). One such option is the Meteostat API, which provides access to average temperature data for specific latitude and longitude coordinates.

Figure 1:



```
[ ] !pip install meteostat
```

```
[ ] import warnings
    warnings.filterwarnings("ignore")
    import pandas as pd
    import numpy as np
    import matplotlib
    from matplotlib import pyplot as plt
```

```
[ ] from datetime import datetime
    import matplotlib.pyplot as plt
    from meteostat import Point, Daily

    # Set time period
    start = datetime(1950, 1, 1)
    end = datetime(2024, 11, 29)

    #Toronto lat/long
    location = Point(43.6532, -79.3832)

    # Get daily data from 1950 to present
    daily_data_retriever = Daily(location, start, end)
    daily_data = daily_data_retriever.fetch()
    daily_data
```

	tavg	tmin	tmax	prcp	snow	wdir	wspd	wpgt	pres	tsun
time										
1957-05-01	NaN	5.6	20.0	0.0	NaN	NaN	NaN	NaN	NaN	NaN
1957-05-02	NaN	3.3	12.2	0.0	NaN	NaN	NaN	NaN	NaN	NaN
1957-05-03	NaN	-1.1	8.3	0.0	NaN	NaN	NaN	NaN	NaN	NaN
1957-05-04	NaN	-0.6	14.4	0.0	NaN	NaN	NaN	NaN	NaN	NaN
1957-05-05	NaN	6.1	12.2	0.0	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...
2024-11-25	3.6	0.9	8.2	0.0	NaN	62.0	18.9	NaN	1015.6	NaN
2024-11-26	6.9	0.9	8.0	6.6	NaN	260.0	29.2	NaN	1009.8	NaN
2024-11-27	2.9	0.5	5.0	0.0	NaN	256.0	24.7	NaN	1014.0	NaN
2024-11-28	3.5	0.6	7.0	0.3	NaN	326.0	18.9	NaN	1008.5	NaN
2024-11-29	0.8	-0.2	8.2	0.1	NaN	256.0	34.2	NaN	1009.8	NaN

24685 rows × 10 columns

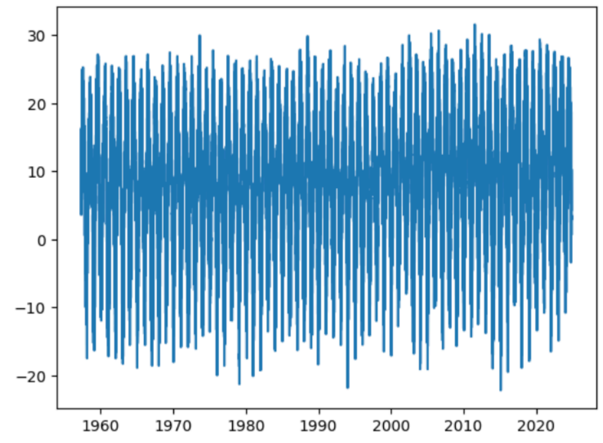
If we are interested in the average temperature (**tavg**), we note that some datapoints are unavailable (**NaN**), so start by cleaning the data set. We could drop all entries where tavg is NaN data, but say we want to go as far back as possible. We will instead fill NaN with the average of **tmin** and **tmax** (an admittedly imperfect but reasonable enough estimation) and do a quick line plot.

Figure 2:

```
[ ] daily_data["tavg"] = daily_data["tavg"].fillna((daily_data["tmin"]+daily_data["tmax"])/2)
    daily_data.head()
```

```
             tavg  tmin  tmax  prcp  snow  wdir  wspd  wpgt  pres  tsun
time
1957-05-01  12.80   5.6  20.0   0.0   NaN   NaN   NaN   NaN   NaN   NaN
1957-05-02   7.75   3.3  12.2   0.0   NaN   NaN   NaN   NaN   NaN   NaN
1957-05-03   3.60  -1.1   8.3   0.0   NaN   NaN   NaN   NaN   NaN   NaN
1957-05-04   6.90  -0.6  14.4   0.0   NaN   NaN   NaN   NaN   NaN   NaN
1957-05-05   9.15   6.1  12.2   0.0   NaN   NaN   NaN   NaN   NaN   NaN
```

```
[ ] daily_data.dropna(subset=["tavg"],inplace=True)
    plt.plot(daily_data.index, daily_data["tavg"])
    plt.show()
```



With Python, we can easily compare temperatures of different regions by visualizing them on a map. These tools can be an invaluable way to understand the structure for highly complex JSON data

Seasonal data

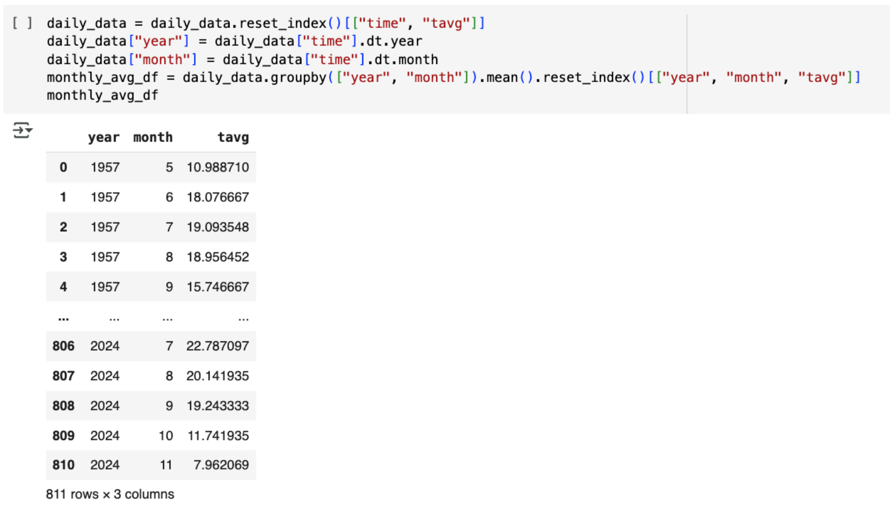
The above plot is complex to decipher but already displays one of the common challenges when analyzing time series data: seasonality. To account for it properly (i.e., to compare apples to apples), we will perform some light data manipulation. These are the steps:

1) Reset **index** so that the "**time**" column becomes just another column, then parse the year and month of each date into separate columns.

2) Group by **Year, Month** pairing. We will average the grouped values.

3) Select only the **year**, **month**, and **tavg** data points. Drop the rest.

Figure 3:



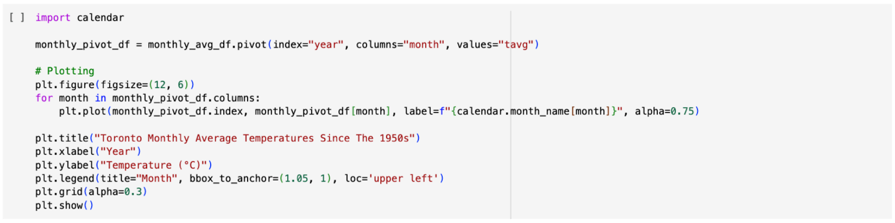
```
[ ] daily_data = daily_data.reset_index()[["time", "tavg"]]
    daily_data["year"] = daily_data["time"].dt.year
    daily_data["month"] = daily_data["time"].dt.month
    monthly_avg_df = daily_data.groupby(["year", "month"]).mean().reset_index()[["year", "month", "tavg"]]
    monthly_avg_df
```

	year	month	tavg
0	1957	5	10.988710
1	1957	6	18.076667
2	1957	7	19.093548
3	1957	8	18.956452
4	1957	9	15.746667
...	...	...	...
806	2024	7	22.787097
807	2024	8	20.141935
808	2024	9	19.243333
809	2024	10	11.741935
810	2024	11	7.962069

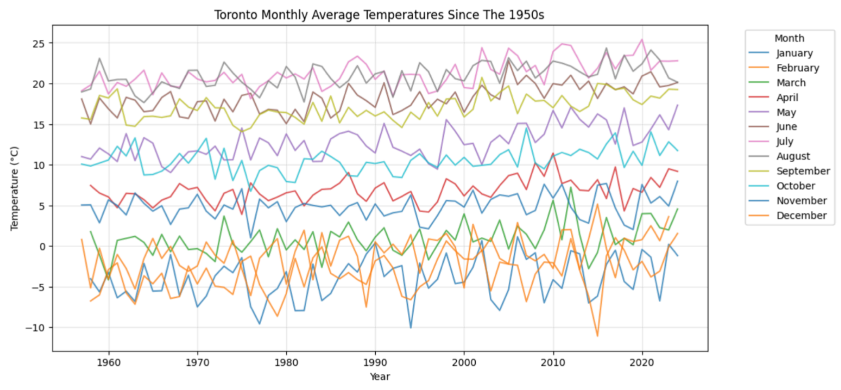
811 rows × 3 columns

Below, for each month, we plot the monthly average over the last 80 years.

Figure 4:

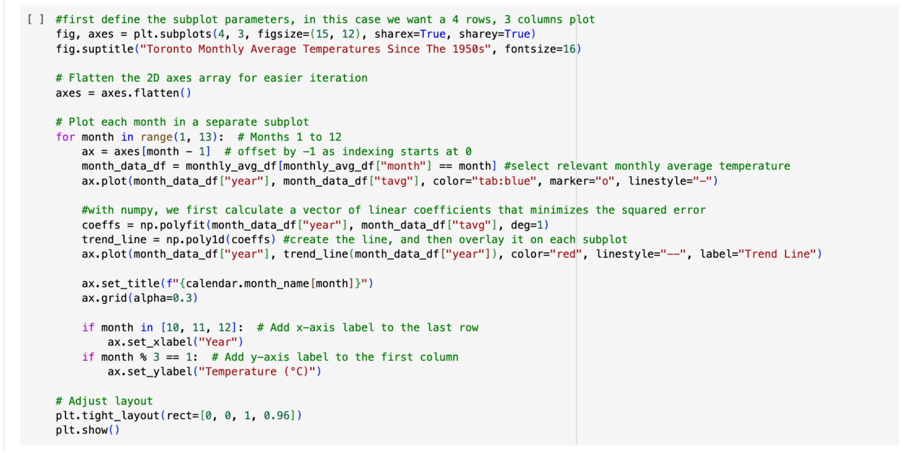


```
[ ] import calendar

    monthly_pivot_df = monthly_avg_df.pivot(index="year", columns="month", values="tavg")

    # Plotting
    plt.figure(figsize=(12, 6))
    for month in monthly_pivot_df.columns:
        plt.plot(monthly_pivot_df.index, monthly_pivot_df[month], label=f"{calendar.month_name[month]}", alpha=0.75)

    plt.title("Toronto Monthly Average Temperatures Since The 1950s")
    plt.xlabel("Year")
    plt.ylabel("Temperature (°C)")
    plt.legend(title="Month", bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.grid(alpha=0.3)
    plt.show()
```

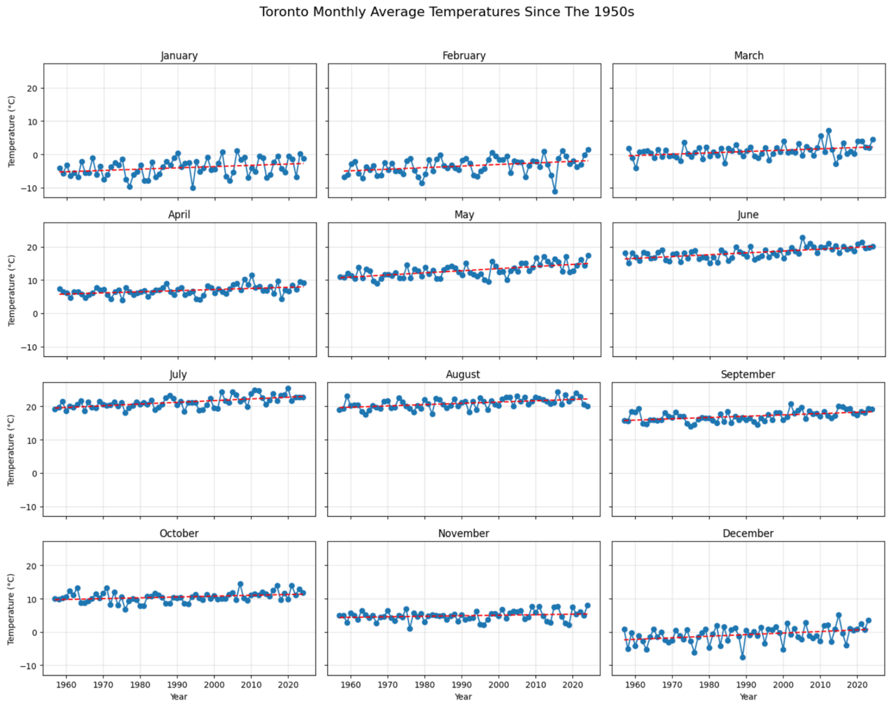


This analysis is clearer and appears to showcase some upward trend for each month. However, the chart looks very busy. With Python, we can easily create subplots and isolate the data series of each month, all with a few lines of code. To make our results more conclusive, we will also add a line of best fit through each subplot with the help of the Numpy library.

Figure 5:



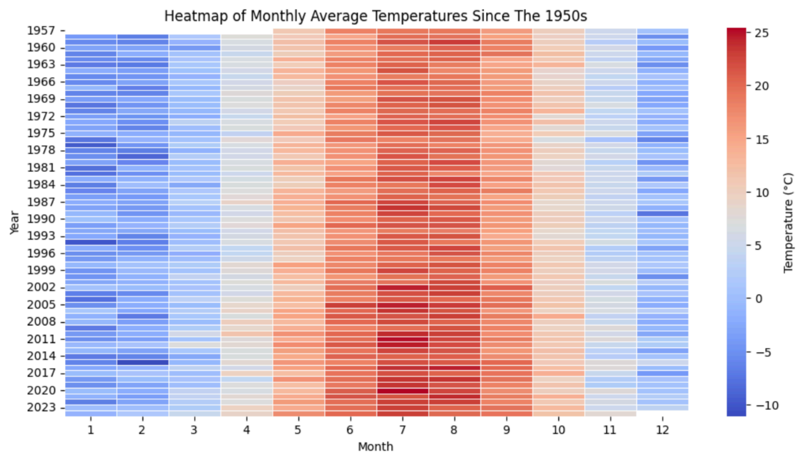
```
[ ] #first define the subplot parameters, in this case we want a 4 rows, 3 columns plot
    fig, axes = plt.subplots(4, 3, figsize=(15, 12), sharex=True, sharey=True)
    fig.suptitle("Toronto Monthly Average Temperatures Since The 1950s", fontsize=16)

    # Flatten the 2D axes array for easier iteration
    axes = axes.flatten()

    # Plot each month in a separate subplot
    for month in range(1, 13):  # Months 1 to 12
        ax = axes[month - 1]  # offset by -1 as indexing starts at 0
        month_data_df = monthly_avg_df[monthly_avg_df["month"] == month] #select relevant monthly average temperature
        ax.plot(month_data_df["year"], month_data_df["tavg"], color="tab:blue", marker="o", linestyle="-")

        #with numpy, we first calculate a vector of linear coefficients that minimizes the squared error
        coeffs = np.polyfit(month_data_df["year"], month_data_df["tavg"], deg=1)
        trend_line = np.poly1d(coeffs) #create the line, and then overlay it on each subplot
        ax.plot(month_data_df["year"], trend_line(month_data_df["year"]), color="red", linestyle="--", label="Trend Line")

        ax.set_title(f"{calendar.month_name[month]}")
        ax.grid(alpha=0.3)

        if month in [10, 11, 12]:  # Add x-axis label to the last row
            ax.set_xlabel("Year")
        if month % 3 == 1:  # Add y-axis label to the first column
            ax.set_ylabel("Temperature (°C)")

    # Adjust layout
    plt.tight_layout(rect=[0, 0, 1, 0.96])
    plt.show()
```



Another popular and compact way to visualize temperature changes is to use a heat map. We will rely on Seaborn, yet another popular library for visualization that is built on top of matplotlib and integrates very well with Pandas dataframes.

Figure 6:

```
[ ] import seaborn as sns

    plt.figure(figsize=(12, 6))
    sns.heatmap(
        monthly_pivot_df,
        cmap="coolwarm",
        annot=False,
        linewidths=0.5,
        cbar_kws={"label": "Temperature (°C)"}
    )
    plt.title("Heatmap of Monthly Average Temperatures Since The 1950s")
    plt.xlabel("Month")
    plt.ylabel("Year")
    plt.show()
```



While more definite conclusions should be supported by deeper statistical analysis, the above visualizations showcase Toronto's rising temperature trends over the years.

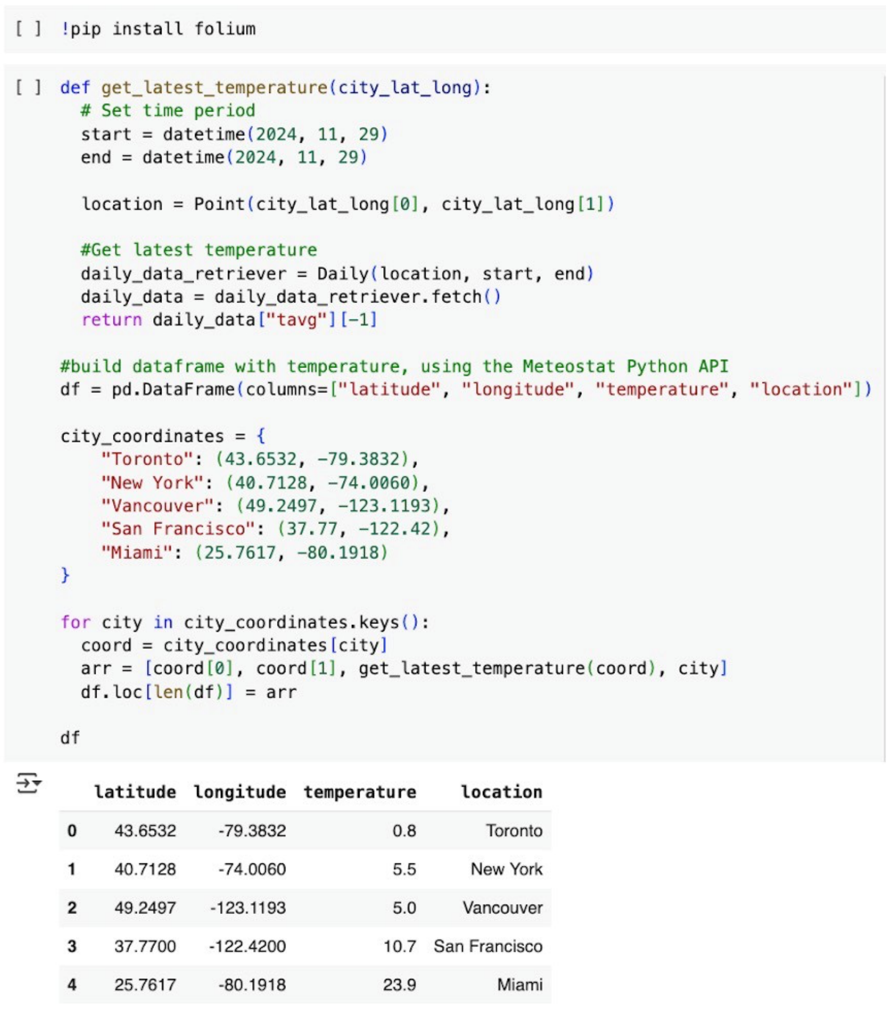
Visualizing data on interactive maps

Suppose we want to compare Toronto's temperature to other regions worldwide. We could overlay even more data series onto the above charts, but let us take advantage of another type of visualization: interactive maps.

With Python, we can easily compare temperatures of different regions by visualizing them on a map. Several libraries allow us to do this, namely Folium, Leaflet, and Mapbox. We will focus on Folium, which allows for easy overlay of labels and markets on a map using Python.

Here, we will start with using the Folium library to overlay the temperature of several major North American cities, once again using Meteostat to pull the underlying temperature data.

Figure 7:



```
[ ] !pip install folium
```

```
[ ] def get_latest_temperature(city_lat_long):
      # Set time period
      start = datetime(2024, 11, 29)
      end = datetime(2024, 11, 29)

      location = Point(city_lat_long[0], city_lat_long[1])

      #Get latest temperature
      daily_data_retriever = Daily(location, start, end)
      daily_data = daily_data_retriever.fetch()
      return daily_data["tavg"][-1]

    #build dataframe with temperature, using the Meteostat Python API
    df = pd.DataFrame(columns=["latitude", "longitude", "temperature", "location"])

    city_coordinates = {
        "Toronto": (43.6532, -79.3832),
        "New York": (40.7128, -74.0060),
        "Vancouver": (49.2497, -123.1193),
        "San Francisco": (37.77, -122.42),
        "Miami": (25.7617, -80.1918)
    }

    for city in city_coordinates.keys():
      coord = city_coordinates[city]
      arr = [coord[0], coord[1], get_latest_temperature(coord), city]
      df.loc[len(df)] = arr

    df
```

	latitude	longitude	temperature	location
0	43.6532	-79.3832	0.8	Toronto
1	40.7128	-74.0060	5.5	New York
2	49.2497	-123.1193	5.0	Vancouver
3	37.7700	-122.4200	10.7	San Francisco
4	25.7617	-80.1918	23.9	Miami

Unlike tabular data, climate data is often stored in GeoJSON format—a structure designed to capture geographic information such as weather patterns, temperature zones, and flood risks tied to specific coordinates. In the final section of this article, we will delve into effectively utilizing GeoJSON for mapping and data analysis.

JSON supports various data types, including strings, numbers, arrays, and nested objects, allowing for the representation of complex data structures.

JSON and GeoJson data.

Financial analysts are no strangers to tabular data, which is arguably the most common method of representation (e.g., Excel, SQL databases) and often the first thing that comes to mind when people think of "data". However, the data is often returned in JSON format when working with web APIs. JSON (JavaScript Object Notation) is a widely adopted format for exchanging data between servers and web applications. It organizes information as "key-value" pairs, making it both human-readable and easily parsed by machines. JSON supports various data types, including strings, numbers, arrays, and nested objects, allowing for the representation of complex data structures. This flexibility makes JSON a go-to format in modern web development and data science workflows.

For location-based data, a standard extension of JSON is GeoJSON, a specialized format designed to store geographic and spatial information. GeoJSON builds upon the JSON structure by introducing standardized keys such as "**type**", "**geometry**", and " **properties**", which describe spatial features like points, lines, and polygons. These features can represent anything from a single location (e.g., a city) to extensive geographic regions (e.g., countries or natural landscapes). GeoJSON is especially useful for visualizing and analyzing geographic data in mapping tools and Geographic Information System (GIS) applications.

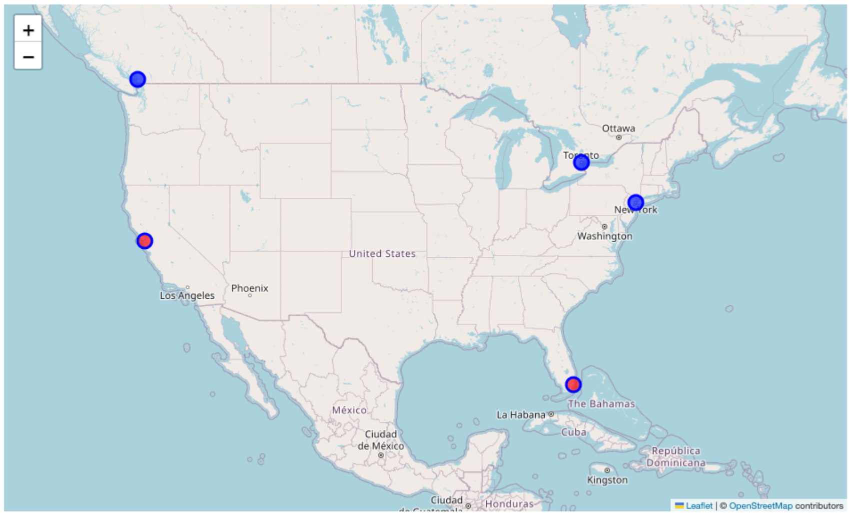
One of GeoJSON's key advantages is its compatibility with a wide range of tools, including Python libraries such as Folium. By combining the structured nature of JSON with the spatial focus of GeoJSON, we can seamlessly exchange and analyze geographic data across diverse systems.

Suppose we are interested in mapping and comparing global temperatures across countries worldwide. To do this, we first need to overlay country borders, available in GeoJSON format at the URL below.

We can use Python's Requests library to retrieve the data and return it in JSON format. JSON data can be complex, and navigating through it in Python may initially seem challenging (we'll demonstrate this below). To help with this, we also include an image of the tree structure generated using an online JSON viewer (https://jsoneditoronline.org/). These tools can be an invaluable way to understand the structure for highly complex JSON data.

Figure 8:

```
[ ] import folium
    # Base map centred on mean of latitude/longitude of dataframe
    m = folium.Map(location=[df["latitude"].mean(), df["longitude"].mean()], zoom_start=5)

    # Add markers with temperature data
    for _, row in df.iterrows():
        folium.CircleMarker(
            location=(row["latitude"], row["longitude"]),
            radius=8,  # Circle size
            color="blue",
            fill=True,
            fill_color="red" if row["temperature"] > 10 else "blue",
            fill_opacity=0.7,
            popup=folium.Popup(f"{row['location']}: {row['temperature']}°C", parse_html=True)
        ).add_to(m)

    m
```



Remember that JSON stores data in a hierarchical, tree-like structure, which can be represented as a nested dictionary in Python. Let's unpack it to understand the data better.

At the first layer, we see "**type**" and "**features**". This is a common way to conceptualize GeoJSON data, as GeoJSON is structured around a FeatureCollection, which contains multiple features, each representing a spatial object like a point, line, or polygon. Each feature typically has:

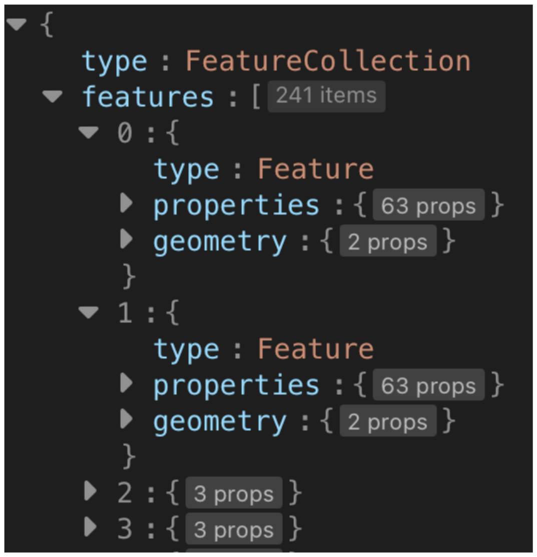
1. **Geometry**: Describes the shape (e.g., Point, LineString, Polygon) and coordinates.
2. **Properties**: Key-value pairs that describe metadata or attributes associated with the geometry (e.g., country names, population).

We get the "**features**" data set via below and see that it is a Python list of length 241. This aligns with what we saw in the image above.

Figure 9:

```
[ ] import requests
    countries_geojson_url = "http://geojson.xyz/naturalearth-3.3.0/ne_50m_admin_0_countries.geojson"

    response = requests.get(countries_geojson_url)
    geojson_data = response.json()
    geojson_data
```

Show hidden output



```
▼ {
    type : FeatureCollection
  ▼ features : [ 241 items
    ▼ 0 : {
        type : Feature
      ▶ properties : { 63 props }
      ▶ geometry : { 2 props }
      }
    ▼ 1 : {
        type : Feature
      ▶ properties : { 63 props }
      ▶ geometry : { 2 props }
      }
    ▶ 2 : { 3 props }
    ▶ 3 : { 3 props }
```

Referencing the first value of the list, we see that it is yet another dictionary with three key-value pairs: "**type**", "**properties**", and "**geometry**", just as shown in the image above. We would be particularly interested in the "name" of the country, which is likely under "properties", and the "geometry" key, as it contains the coordinates.

Figure 10:

```
[ ] print(geojson_data.keys()) #the first layer has two keys: "type" and "features"
    print(type(geojson_data["features"]), len(geojson_data["features"]))
```

```
dict_keys(['type', 'features'])
<class 'list'> 241
```

Referencing "properties" under the variable first_country_data, we see that one of the keys is in fact "**name**". Let's retrieve the associated value. It is Aruba!

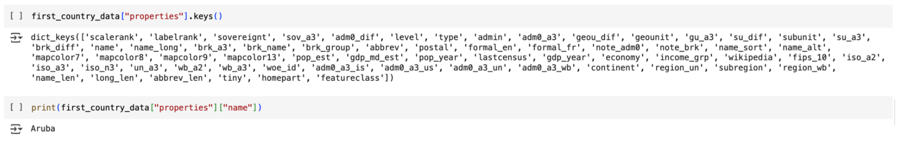
Figure 11:

```
[ ] first_country_data = geojson_data["features"][0]
    print(type(first_country_data))
    print(first_country_data.keys())
```

```
<class 'dict'>
dict_keys(['type', 'properties', 'geometry'])
```

As for "geometry", we note that the dictionary retrieval contains the key "**coordinates**", which will return a list of coordinates. These coordinates make up the border of Aruba.

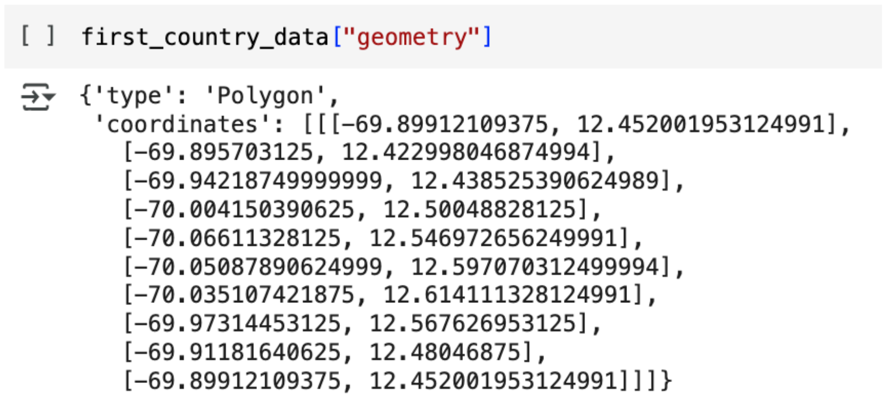
Figure 12:



```
[ ] first_country_data["properties"].keys()

dict_keys(['scalerank', 'labelrank', 'sovereignt', 'sov_a3', 'adm0_dif', 'level', 'type', 'admin', 'adm0_a3', 'geou_dif', 'geounit', 'gu_a3', 'su_dif', 'subunit', 'su_a3',
'brk_diff', 'name', 'name_long', 'brk_a3', 'brk_name', 'brk_group', 'abbrev', 'postal', 'formal_en', 'formal_fr', 'note_adm0', 'note_brk', 'name_sort', 'name_alt',
'mapcolor7', 'mapcolor8', 'mapcolor9', 'mapcolor13', 'pop_est', 'gdp_md_est', 'pop_year', 'lastcensus', 'gdp_year', 'economy', 'income_grp', 'wikipedia', 'fips_10', 'iso_a2',
'iso_a3', 'iso_n3', 'un_a3', 'wb_a2', 'wb_a3', 'woe_id', 'adm0_a3_is', 'adm0_a3_us', 'adm0_a3_un', 'adm0_a3_wb', 'continent', 'region_un', 'subregion', 'region_wb',
'name_len', 'long_len', 'abbrev_len', 'tiny', 'homepart', 'featureclass'])

[ ] print(first_country_data["properties"]["name"])

Aruba
```

We have now found the pattern to access all countries' names and border coordinates in the data set. We want to get the border coordinates of Canada and plot it on the map. Folium's built-in **GeoJSON()** method allows us to easily map GeoJSON data onto the interactive map.

Figure 13:



```
[ ] first_country_data["geometry"]

{'type': 'Polygon',
 'coordinates': [[[-69.89912109375, 12.452001953124991],
   [-69.895703125, 12.422998046874994],
   [-69.94218749999999, 12.438525390624989],
   [-70.004150390625, 12.50048828125],
   [-70.06611328125, 12.546972656249991],
   [-70.05087890624999, 12.597070312499994],
   [-70.035107421875, 12.614111328124991],
   [-69.97314453125, 12.567626953125],
   [-69.91181640625, 12.48046875],
   [-69.89912109375, 12.452001953124991]]]}
```

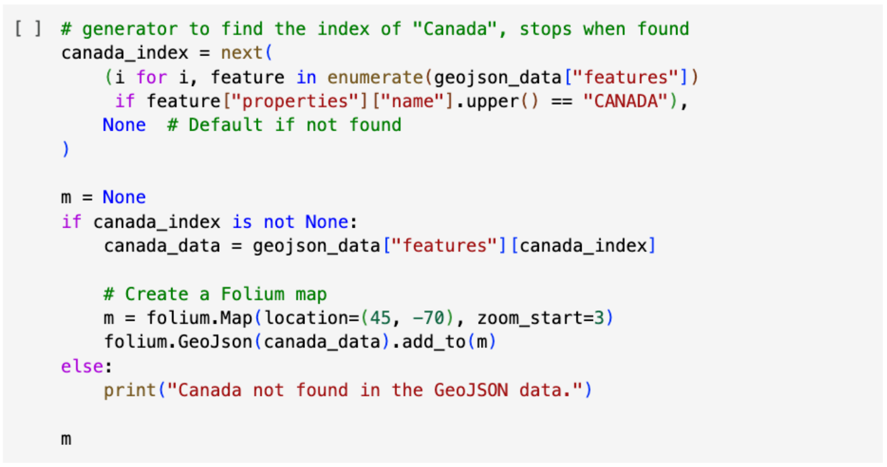
Choropleth map

Defining country borders on the map is crucial, as it establishes the framework for layering additional data. With these boundaries in place, we can now create a choropleth map, which uses variations in color or shading to represent data values across geographical areas to visualize and compare temperature changes in countries over time.

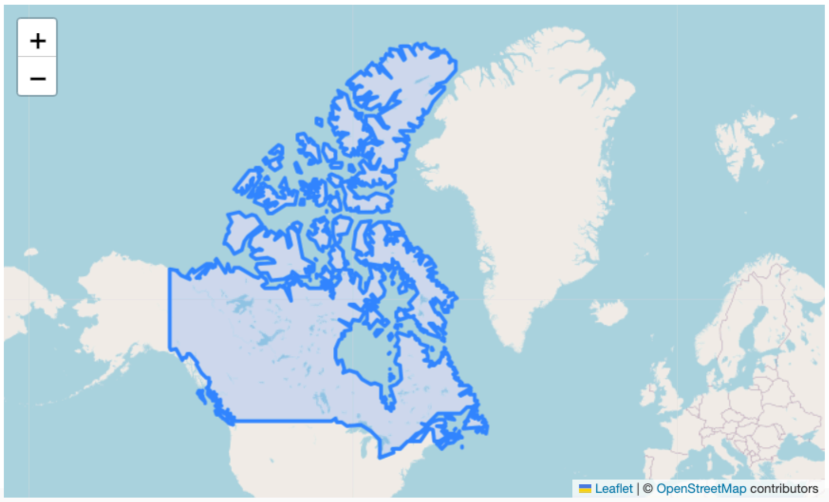
Let us first read in global historical temperature. The data set can be viewed on Kaggle.

We will apply some initial transformations to the data, such as dropping NaNs and converting the date type.

Figure 14:



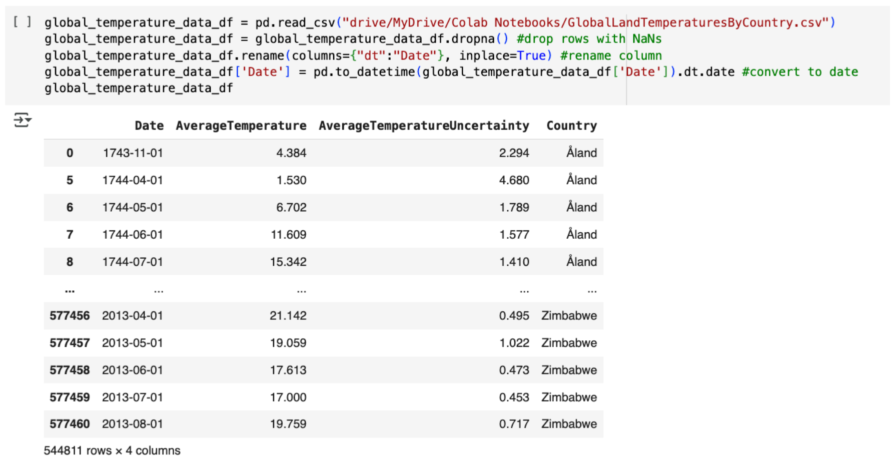
```
[ ] # generator to find the index of "Canada", stops when found
    canada_index = next(
        (i for i, feature in enumerate(geojson_data["features"])
         if feature["properties"]["name"].upper() == "CANADA"),
        None  # Default if not found
    )

    m = None
    if canada_index is not None:
        canada_data = geojson_data["features"][canada_index]

        # Create a Folium map
        m = folium.Map(location=(45, -70), zoom_start=3)
        folium.GeoJson(canada_data).add_to(m)
    else:
        print("Canada not found in the GeoJSON data.")

    m
```



Although the data set goes back to the year 1743, we cannot assume that 1743 data is available for all countries. Let us find the maximum range of dates for which we would have data for all countries.

For each country, we find the most recent data points available.

Figure 15:

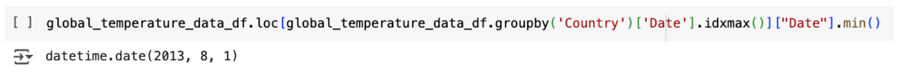


```
[ ] global_temperature_data_df = pd.read_csv("drive/MyDrive/Colab Notebooks/GlobalLandTemperaturesByCountry.csv")
    global_temperature_data_df = global_temperature_data_df.dropna() #drop rows with NaNs
    global_temperature_data_df.rename(columns={"dt":"Date"}, inplace=True) #rename column
    global_temperature_data_df['Date'] = pd.to_datetime(global_temperature_data_df['Date']).dt.date #convert to date
    global_temperature_data_df
```

```
              Date  AverageTemperature  AverageTemperatureUncertainty   Country
     0  1743-11-01               4.384                          2.294     Åland
     5  1744-04-01               1.530                          4.680     Åland
     6  1744-05-01               6.702                          1.789     Åland
     7  1744-06-01              11.609                          1.577     Åland
     8  1744-07-01              15.342                          1.410     Åland
   ...         ...                 ...                            ...       ...
577456  2013-04-01              21.142                          0.495  Zimbabwe
577457  2013-05-01              19.059                          1.022  Zimbabwe
577458  2013-06-01              17.613                          0.473  Zimbabwe
577459  2013-07-01              17.000                          0.453  Zimbabwe
577460  2013-08-01              19.759                          0.717  Zimbabwe
544811 rows × 4 columns
```

We also find the earliest start date for which we would have data on all countries.

Figure 16:

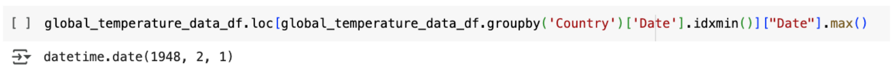


```
[ ] global_temperature_data_df.loc[global_temperature_data_df.groupby('Country')['Date'].idxmax()]["Date"].min()
```

```
datetime.date(2013, 8, 1)
```

From above, we see that for select countries, data is not available until 1948. So, let's use 1948 as the starting point. To keep it simple, let's compute the temperature change in August from 1948 to 2013 for each country.

Figure 17:

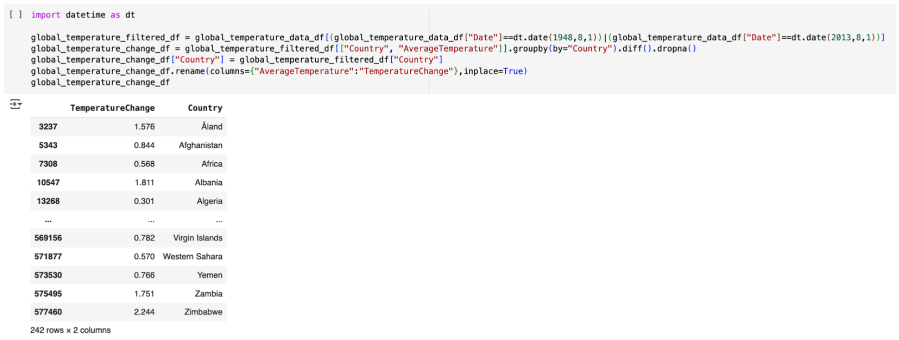


```
[ ] global_temperature_data_df.loc[global_temperature_data_df.groupby('Country')['Date'].idxmin()]["Date"].max()
```

```
datetime.date(1948, 2, 1)
```

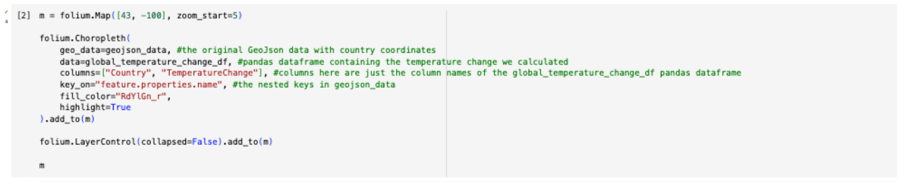
We are now ready to overlay the data onto the Folium map using the **folium.Choropleth()** method. The method has many optional arguments, but the key arguments to note in this case are **geo_data**, **data**, **columns**, and **key_on**, all of which are needed to properly overlay and join the two data sets.

Figure 18:

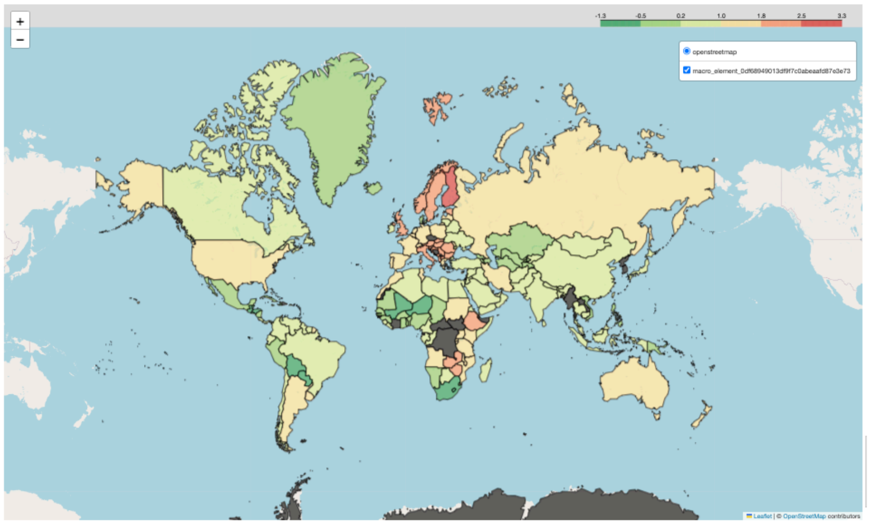


```
[ ] import datetime as dt

    global_temperature_filtered_df = global_temperature_data_df[(global_temperature_data_df["Date"]==dt.date(1948,8,1))|(global_temperature_data_df["Date"]==dt.date(2013,8,1))]
    global_temperature_change_df = global_temperature_filtered_df[["Country", "AverageTemperature"]].groupby(by="Country").diff().dropna()
    global_temperature_change_df["Country"] = global_temperature_filtered_df["Country"]
    global_temperature_change_df.rename(columns={"AverageTemperature":"TemperatureChange"},inplace=True)
    global_temperature_change_df
```

```
        TemperatureChange         Country
3237                1.576           Åland
5343                0.844     Afghanistan
7308                0.568          Africa
10547               1.811         Albania
13268               0.301         Algeria
...                   ...             ...
569156              0.782  Virgin Islands
571877              0.570  Western Sahara
573530              0.766           Yemen
575495              1.751          Zambia
577460              2.244        Zimbabwe
242 rows × 2 columns
```



```
[2] m = folium.Map([43, -100], zoom_start=5)

    folium.Choropleth(
        geo_data=geojson_data, #the original GeoJson data with country coordinates
        data=global_temperature_change_df, #pandas dataframe containing the temperature change we calculated
        columns=["Country", "TemperatureChange"], #columns here are just the column names of the global_temperature_change_df pandas dataframe
        key_on="feature.properties.name", #the nested keys in geojson_data
        fill_color="RdYlGn_r",
        highlight=True
    ).add_to(m)

    folium.LayerControl(collapsed=False).add_to(m)

    m
```


-1.3
-0.5
0.2
1.0
1.8
2.5
3.3
openstreetmap
macro_element_0df68949013df9f7c0abeaafd87e3e73
Leaflet | © OpenStreetMap contributors

Folium allows us to overlay various data onto maps, making it easier to identify trends and clusters at a glance. For instance, a financial institution might overlay flood zone data in coastal cities with insurance claims to better inform property insurance pricing.

Python's extensive ecosystem of libraries, as well as data APIs written for Python, simplifies data procurement, transformation, and visualization. At the same time, Folium enables the creation of interactive maps to visualize trends and patterns. Python's seamless handling of GeoJSON data, essential for representing geographic information, makes integrating spatial data into analyses straightforward. As such, Python stands out as a versatile and powerful tool for visualizing climate data.

Winfred Lam, CFA, is a Manager at BMO Corporate Treasury responsible for Balance Sheet Management. He holds a master's degree in computer science from the University of Pennsylvania and is a volunteer member of CFA Society Toronto's Member Communications Committee.