CM4125 Topic 7: Time Series Data + Bar Chart Race

Lecture objectives

Lecture objectives

1. Understand the concept of time series data and what does it imply

Lecture objectives

1. Understand the concept of time series data and what does it imply

2. Process time series datasets into effective dataframes

Lecture objectives

1. Understand the concept of time series data and what does it imply

2. Process time series datasets into effective dataframes

3. Produce a bar chart race

What is Time Series Data (TSD)?

Simply put, datasets that describe/contain data which varies over time

Simply put, datasets that describe/contain data which varies over time

They are usually characterised by a fixed and constant aggregation of data

```
Readings from sensors
COVID cases
Trends in economics
```

Simply put, datasets that describe/contain data which varies over time

They are usually characterised by a fixed and constant aggregation of data

```
Readings from sensors
COVID cases
Trends in economics
```

The art of using these datasets resides in knowing how to shape them and using the right features

To practice, we will use the dataset called `bitcoin.csv`

To practice, we will use the dataset called `bitcoin.csv`

This dataset contains the price of the Bitcoin cryptocurrency on the US-based exchange Gemini, minute-by-minute for all of 2017

In [1]:
```
import pandas as pd
import numpy as np
bitcoin = pd.read_csv('https://www.dropbox.com/s/kxjc92oia6gcoy8/bitcoin.csv?r
bitcoin
```

Out[1]:

	Open	High	Low	Close	Volume
Date					
2017-01-01 00:00:00	974.55	974.55	974.55	974.55	0.000000
2017-01-01 00:01:00	974.55	974.55	974.55	974.55	0.000000
2017-01-01 00:02:00	974.55	974.55	970.00	970.00	0.417679
2017-01-01 00:03:00	970.00	970.00	970.00	970.00	0.000514
2017-01-01 00:04:00	970.00	970.00	970.00	970.00	0.000000
...	...	...	...	...	...
2017-12-31 23:55:00	13782.87	13782.87	13771.00	13771.00	1.657980
2017-12-31 23:56:00	13771.00	13775.00	13770.95	13775.00	7.538428
2017-12-31 23:57:00	13775.00	13815.37	13775.00	13815.37	18.437304
2017-12-31 23:58:00	13815.37	13825.00	13815.37	13825.00	3.547593
2017-12-31 23:59:00	13825.00	13825.00	13804.68	13820.26	2.610719

525600 rows × 5 columns

Note that the `Date` column contains entries formatted as a date-time such as `2017-01-01 00:00:00`

Note that the `Date` column contains entries formatted as a date-time such as `2017-01-01 00:00:00`

First, let's check the type of this data by examining the first item

Note that the `Date` column contains entries formatted as a date-time such as `2017-01-01 00:00:00`

First, let's check the type of this data by examining the first item

In [2]:
```
first_index_value = bitcoin.index[0]
print(first_index_value)
print(type(first_index_value))
```

```
2017-01-01 00:00:00
<class 'str'>
```

We can avoid this in one go by parsing the dates when loading

We can avoid this in one go by parsing the dates when loading

In [3]:
```
bitcoin = pd.read_csv('https://www.dropbox.com/s/kxjc92oia6gcoy8/bitcoin.csv?r
                      parse_dates=['Date'], index_col=0)
bitcoin
```

Out[3]:

Date	Open	High	Low	Close	Volume
2017-01-01 00:00:00	974.55	974.55	974.55	974.55	0.000000
2017-01-01 00:01:00	974.55	974.55	974.55	974.55	0.000000
2017-01-01 00:02:00	974.55	974.55	970.00	970.00	0.417679
2017-01-01 00:03:00	970.00	970.00	970.00	970.00	0.000514
2017-01-01 00:04:00	970.00	970.00	970.00	970.00	0.000000
...	...	...	...	...	...
2017-12-31 23:55:00	13782.87	13782.87	13771.00	13771.00	1.657980
2017-12-31 23:56:00	13771.00	13775.00	13770.95	13775.00	7.538428
2017-12-31 23:57:00	13775.00	13815.37	13775.00	13815.37	18.437304
2017-12-31 23:58:00	13815.37	13825.00	13815.37	13825.00	3.547593
2017-12-31 23:59:00	13825.00	13825.00	13804.68	13820.26	2.610719

525600 rows × 5 columns

If we check the column now, it is marked as `Timestamp`

If we check the column now, it is marked as `Timestamp`

In [4]:
```
first_index_value = bitcoin.index[0]
print(first_index_value)
print(type(first_index_value))
```

```
2017-01-01 00:00:00
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

Exploring TSD

This dataset contains the following columns:

This dataset contains the following columns:

Price Data:

- Open - The price of the commodity *(bitcoin)* at the **start** of the time period *(minute)*
- High - The **highest** value the price reached during the time period
- Low - The **lowest** value the price reached during the time period
- Close - The price at the **end** of the time period

Trade Data:

- Volume - The total amount of the commodity bought/sold in the time period

Trade Data:

- Volume - The total amount of the commodity bought/sold in the time period

Other time series datasets with have other columns, but this is common for financial data

Even though the index is stored as `Timestamp`, we can retrieve rows by querying with a string

Even though the index is stored as `Timestamp`, we can retrieve rows by querying with a string

For example during the minute `2017-05-27 19:30:00`, we can query using `2017-05-27 19:30:00` which matches the data

Even though the index is stored as `Timestamp`, we can retrieve rows by querying with a string

For example during the minute `2017-05-27 19:30:00`, we can query using `2017-05-27 19:30:00` which matches the data

However, since the dataset has only minute-by-minute data, the seconds don't matter, so we can exclude the seconds in our query and search for `2017-05-27 19:30`

In [5]:
```
bitcoin.loc['2017-05-27 19:30']
```

Out[5]:
```
Open      2144.68000
High      2144.68000
Low       2141.26000
Close     2142.53000
Volume       0.05406
Name: 2017-05-27 19:30:00, dtype: float64
```

In [5]:
```
bitcoin.loc['2017-05-27 19:30']
```

Out[5]:
```
Open      2144.68000
High      2144.68000
Low       2141.26000
Close     2142.53000
Volume       0.05406
Name: 2017-05-27 19:30:00, dtype: float64
```

The price started this minute at `2144.68` USD, dropped to `2142.53` USD at the end of the minute

During this time the minimum value was 2141.26 USD and the maximum value was 2144.68 USD

During this time the minimum value was `2141.26` USD and the maximum value was `2144.68` USD

`0.05406` bitcoin was bought/sold during this minute

During this time the minimum value was `2141.26` USD and the maximum value was `2144.68` USD

`0.05406` bitcoin was bought/sold during this minute

Let's check the next minute `2017-05-27 19:31`

In [6]:
```
bitcoin.loc['2017-05-27 19:31']
```

Out[6]:
```
Open      2142.5300
High      2142.5300
Low       2141.2700
Close     2141.2700
Volume       0.2069
Name: 2017-05-27 19:31:00, dtype: float64
```

In [6]:
```
bitcoin.loc['2017-05-27 19:31']
```

Out[6]:
```
Open      2142.5300
High      2142.5300
Low       2141.2700
Close     2141.2700
Volume       0.2069
Name: 2017-05-27 19:31:00, dtype: float64
```

The previous minute **closed** at `2142.53` USD and this minute **opened** at `2142.53` USD

In [6]:
```
bitcoin.loc['2017-05-27 19:31']
```

Out[6]:
```
Open      2142.5300
High      2142.5300
Low       2141.2700
Close     2141.2700
Volume       0.2069
Name: 2017-05-27 19:31:00, dtype: float64
```

The previous minute **closed** at `2142.53` USD and this minute **opened** at `2142.53` USD

This is expected since the end of one minute is the same as the start of the next minute!

If we want to check the open and close price for the **day**, we could check the open price for the first minute and close price for the last minute

If we want to check the open and close price for the **day**, we could check the open price for the first minute and close price for the last minute

In [7]:
```
print("Date: 2017-05-27")
print("Open: ", bitcoin.loc['2017-05-27 00:00'].at["Open"])
print("Close: ", bitcoin.loc['2017-05-27 23:59'].at["Close"])
```

```
Date: 2017-05-27
Open:  2267.37
Close:  2096.77
```

We can select an interval of time the same way we do with numbers

We can select an interval of time the same way we do with numbers

With the following code, we select all of the values in May of 2017

We can select an interval of time the same way we do with numbers

With the following code, we select all of the values in May of 2017

In [8]:
```
bitcoin[(bitcoin.index >= '2017-05') & (bitcoin.index < '2017-06')]
```

Out[8]:

Date	Open	High	Low	Close	Volume
2017-05-01 00:00:00	1362.63	1362.63	1359.05	1359.05	0.129530
2017-05-01 00:01:00	1359.05	1359.05	1359.05	1359.05	0.015945
2017-05-01 00:02:00	1359.05	1359.05	1359.05	1359.05	0.430274
2017-05-01 00:03:00	1359.05	1359.05	1359.05	1359.05	0.024931
2017-05-01 00:04:00	1359.05	1359.05	1359.05	1359.05	0.000000
...	...	...	...	...	...
2017-05-31 23:55:00	2301.80	2301.80	2301.80	2301.80	0.000000
2017-05-31 23:56:00	2301.80	2301.80	2301.80	2301.80	0.065392
2017-05-31 23:57:00	2301.80	2301.80	2301.80	2301.80	0.000000
2017-05-31 23:58:00	2301.80	2301.80	2301.80	2301.80	0.000000
2017-05-31 23:59:00	2301.80	2303.62	2301.80	2303.62	13.587863

44640 rows × 5 columns

Selecting all of the values for 1st May before noon

Selecting all of the values for 1st May before noon

In [9]:
```
bitcoin[(bitcoin.index >= '2017-05-01') & (bitcoin.index < '2017-05-01 12:00')
```

Out[9]:

Date	Open	High	Low	Close	Volume
2017-05-01 00:00:00	1362.63	1362.63	1359.05	1359.05	0.129530
2017-05-01 00:01:00	1359.05	1359.05	1359.05	1359.05	0.015945
2017-05-01 00:02:00	1359.05	1359.05	1359.05	1359.05	0.430274
2017-05-01 00:03:00	1359.05	1359.05	1359.05	1359.05	0.024931
2017-05-01 00:04:00	1359.05	1359.05	1359.05	1359.05	0.000000
...	...	...	...	...	...
2017-05-01 11:55:00	1376.43	1376.43	1376.40	1376.43	0.064194
2017-05-01 11:56:00	1376.43	1377.36	1376.43	1377.36	0.013068
2017-05-01 11:57:00	1377.36	1377.36	1377.35	1377.35	0.023829
2017-05-01 11:58:00	1377.35	1378.25	1377.35	1378.24	17.260743
2017-05-01 11:59:00	1378.24	1378.24	1378.24	1378.24	0.000000

720 rows × 5 columns

Calculations with TSD

We may want to calculate other values such as taking the middle value between the `Low` and `High` values, this will allow us to calculate an estimate for the value of the bitcoin traded

We may want to calculate other values such as taking the middle value between the `Low` and `High` values, this will allow us to calculate an estimate for the value of the bitcoin traded

It will only be an **estimate**, since we don't have the exact list of trades available in this dataset!

We may want to calculate other values such as taking the middle value between the `Low` and `High` values, this will allow us to calculate an estimate for the value of the bitcoin traded

It will only be an **estimate**, since we don't have the exact list of trades available in this dataset!

We can perform calculations on the columns in exactly the same we we have with any other dataset before

In [10]:

```
bitcoin['Middle'] = (bitcoin['High'] + bitcoin['Low']) / 2
bitcoin['USD Volume'] = bitcoin['Middle'] * bitcoin['Volume']
bitcoin
```

Out[10]:

Date	Open	High	Low	Close	Volume	Middle	USD V
2017-01-01 00:00:00	974.55	974.55	974.55	974.55	0.000000	974.550	0.
2017-01-01 00:01:00	974.55	974.55	974.55	974.55	0.000000	974.550	0.
2017-01-01 00:02:00	974.55	974.55	970.00	970.00	0.417679	972.275	406.
2017-01-01 00:03:00	970.00	970.00	970.00	970.00	0.000514	970.000	0.
2017-01-01 00:04:00	970.00	970.00	970.00	970.00	0.000000	970.000	0.
...	...	...	...	...	...	...	...
2017-12-31 23:55:00	13782.87	13782.87	13771.00	13771.00	1.657980	13776.935	22841.
2017-12-31 23:56:00	13771.00	13775.00	13770.95	13775.00	7.538428	13772.975	103826.
2017-12-31 23:57:00	13775.00	13815.37	13775.00	13815.37	18.437304	13795.185	254346.
2017-12-31 23:58:00	13815.37	13825.00	13815.37	13825.00	3.547593	13820.185	49028.
2017-12-31 23:59:00	13825.00	13825.00	13804.68	13820.26	2.610719	13814.840	36066.

Resampling TSD

Sometimes we are interested in sampling the data in larger chunks than it is presented

Sometimes we are interested in sampling the data in larger chunks than it is presented

For instance, this data has one line per **minute**; we may want data for **days** instead

Sometimes we are interested in sampling the data in larger chunks than it is presented

For instance, this data has one line per **minute**; we may want data for **days** instead

The method to resample the data in Python is called `resample`, and takes a time period as an argument

For example, `1D` means 1 day (you can see more options here)

For example, `1D` means 1 day (you can see more options here)

In [11]:
```
# Not very useful output, right?
bitcoin.resample('1D')
```

Out[11]:
```
<pandas.core.resample.DatetimeIndexResampler object at 0x0000027A1CC5
2750>
```

If we call the `mean` method, it will average out the values of this new object by day

If we call the `mean` method, it will average out the values of this new object by day

In [12]:
```
bitcoin.resample('1D').mean()
```

Out[12]:

Date	Open	High	Low	Close	Volume	
2017-01-01	980.776378	980.898135	980.663396	980.793059	0.743065	
2017-01-02	1016.424132	1016.618785	1016.247319	1016.439174	1.006385	1
2017-01-03	1023.624340	1023.743431	1023.514653	1023.634618	1.596420	1
2017-01-04	1082.933979	1083.422104	1082.550431	1083.009069	2.258822	1
2017-01-05	1056.542243	1057.577083	1055.411271	1056.446187	4.190673	1
...	...	...	...	...	...	
2017-12-27	15639.538896	15655.889035	15621.917583	15639.252847	6.427652	15
2017-12-28	14261.058215	14285.193507	14237.371021	14260.412146	7.639668	14
2017-12-29	14535.871354	14549.550931	14522.446875	14535.931340	4.438627	14
2017-12-30	13476.331806	13494.028972	13454.185951	13475.012840	6.065731	13
2017-12-31	13274.262319	13285.760785	13262.565792	13275.057167	5.678043	13

365 rows × 7 columns

The method takes the values in the column `Volume` and rows `2017-01-01 00:00` to `2017-01-01 23:59` and averages them

The method takes the values in the column `Volume` and rows `2017-01-01 00:00` to `2017-01-01 23:59` and averages them

We can also get the total by adding the values

The method takes the values in the column `Volume` and rows `2017-01-01 00:00` to `2017-01-01 23:59` and averages them

We can also get the total by adding the values

In [13]:
```
# The number of rows (365) hint that it was correct
bitcoin.resample('1D').sum()
```

Out[13]:

Date	Open	High	Low	Close	Volume
2017-01-01	1.412318e+06	1.412493e+06	1412155.29	1.412342e+06	1070.013119
2017-01-02	1.463651e+06	1.463931e+06	1463396.14	1.463672e+06	1449.193752
2017-01-03	1.474019e+06	1.474191e+06	1473861.10	1.474034e+06	2298.844903
2017-01-04	1.559425e+06	1.560128e+06	1558872.62	1.559533e+06	3252.704150
2017-01-05	1.521421e+06	1.522911e+06	1519792.23	1.521283e+06	6034.569411
...	...	...	...	...	...
2017-12-27	2.252094e+07	2.254448e+07	22495561.32	2.252052e+07	9255.819566
2017-12-28	2.053592e+07	2.057068e+07	20501814.27	2.053499e+07	11001.121472
2017-12-29	2.093165e+07	2.095135e+07	20912323.50	2.093174e+07	6391.623337
2017-12-30	1.940592e+07	1.943140e+07	19374027.77	1.940402e+07	8734.652933
2017-12-31	1.911494e+07	1.913150e+07	19098094.74	1.911608e+07	8176.382068

365 rows × 7 columns

This operation was more useful for the `Volume` column, as it now sums up all of the entries for that day

This operation was more useful for the `Volume` column, as it now sums up all of the entries for that day

We can see that `1070.013119` bitcoin (about $1,061,594 at the time) were traded on 01/01

In fact there was another way to get this!

```
Multiply bitcoin per minute rate from before (0.743065) times the number of minute in a day (1440)
```

In fact there was another way to get this!

```
Multiply bitcoin per minute rate from before (0.743065) times the number of minute in a day (1440)
```

In [14]:
```
1440*0.743065
```

Out[14]:
```
1070.0136
```

Resampling TSD by Column

If we want the lowest or highest price for the whole day, we don't need an average or a sum!

If we want the lowest or highest price for the whole day, we don't need an average or a sum!

We can tell Pandas to use `max` on these values, and also we can use `min` for `Low`

If we want the lowest or highest price for the whole day, we don't need an average or a sum!

We can tell Pandas to use `max` on these values, and also we can use `min` for `Low`

In [15]:
```
bitcoin.resample('1D').max()
```

Out[15]:

Date	Open	High	Low	Close	Volume	Middle	USD V
2017-01-01	1003.54	1003.54	1003.54	1003.54	110.126689	1003.540	1.1029
2017-01-02	1035.89	1035.90	1035.88	1035.89	116.627861	1035.885	1.1971
2017-01-03	1038.26	1038.26	1038.26	1038.26	405.000039	1038.260	4.1504
2017-01-04	1150.00	1155.54	1149.97	1150.00	167.000000	1149.980	1.8391
2017-01-05	1167.16	1167.16	1167.16	1167.16	918.000000	1167.160	8.8229
...	...	...	...	...	...	...	
2017-12-27	16500.00	16500.00	16499.58	16500.00	115.322939	16499.790	1.7044
2017-12-28	15495.96	15499.99	15493.84	15495.96	143.885194	15494.420	2.0054
2017-12-29	15077.92	15115.10	15060.75	15077.92	172.666517	15087.925	2.4483
2017-12-30	14607.75	14607.77	14577.60	14607.75	168.027243	14592.685	2.2051
2017-12-31	14180.00	14191.20	14163.09	14180.00	106.797334	14171.545	1.4359

365 rows × 7 columns

If we want the lowest or highest price for the whole day, we don't need an average or a sum!

We can tell Pandas to use `max` on these values, and also we can use `min` for `Low`

In [15]:
```
bitcoin.resample('1D').max()
```

Out[15]:

Date	Open	High	Low	Close	Volume	Middle	USD V
2017-01-01	1003.54	1003.54	1003.54	1003.54	110.126689	1003.540	1.1029
2017-01-02	1035.89	1035.90	1035.88	1035.89	116.627861	1035.885	1.1971
2017-01-03	1038.26	1038.26	1038.26	1038.26	405.000039	1038.260	4.1504
2017-01-04	1150.00	1155.54	1149.97	1150.00	167.000000	1149.980	1.8391
2017-01-05	1167.16	1167.16	1167.16	1167.16	918.000000	1167.160	8.8229
...	...	...	...	...	...	...	
2017-12-27	16500.00	16500.00	16499.58	16500.00	115.322939	16499.790	1.7044
2017-12-28	15495.96	15499.99	15493.84	15495.96	143.885194	15494.420	2.0054
2017-12-29	15077.92	15115.10	15060.75	15077.92	172.666517	15087.925	2.4483
2017-12-30	14607.75	14607.77	14577.60	14607.75	168.027243	14592.685	2.2051
2017-12-31	14180.00	14191.20	14163.09	14180.00	106.797334	14171.545	1.4359

365 rows × 7 columns

In [16]:
```
bitcoin.resample('1D').min()
```

Out[16]:

Date	Open	High	Low	Close	Volume	Middle	USD Volume
2017-01-01	965.00	965.00	965.00	965.00	0.0	965.000	0.0

To generate a new dataframe with both changes applied to the respective column in one line of code, we will use the `resample`, `lambda` and `apply` methods:

- For `High`, we will `max` the values of `High` for each minute
- For `Low`, we will use `min`
- For `Volume` and `USD Volume`, we will use `sum`
- For `Open`, we will use the **first** item (index `[0]`)
- For `Close`, we will use the **last** item (index `[-1]`)

In [17]:

```
daily = bitcoin.resample('1D').apply({'Open'       : lambda x: x[0],
                                      'High'       : np.max,
                                      'Low'        : np.min,
                                      'Close'      : lambda x: x[-1],
                                      'Volume'     : np.sum,
                                      'USD Volume' : np.sum})
daily
```

```
C:\Users\CM8738\AppData\Local\Temp\ipykernel_21260\1017184179.py:1: Fu
tureWarning: Series.__getitem__ treating keys as positions is deprecat
ed. In a future version, integer keys will always be treated as labels
(consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  daily = bitcoin.resample('1D').apply({'Open'       : lambda x: x[0],
C:\Users\CM8738\AppData\Local\Temp\ipykernel_21260\1017184179.py:1: Fu
tureWarning: The provided callable <function max at 0x0000027A7F8B1F80
> is currently using SeriesGroupBy.max. In a future version of pandas,
the provided callable will be used directly. To keep current behavior
pass the string "max" instead.
  daily = bitcoin.resample('1D').apply({'Open'       : lambda x: x[0],
C:\Users\CM8738\AppData\Local\Temp\ipykernel_21260\1017184179.py:1: Fu
tureWarning: The provided callable <function min at 0x0000027A7F8B20C0
> is currently using SeriesGroupBy.min. In a future version of pandas,
the provided callable will be used directly. To keep current behavior
pass the string "min" instead.
  daily = bitcoin.resample('1D').apply({'Open'       : lambda x: x[0],
C:\Users\CM8738\AppData\Local\Temp\ipykernel_21260\1017184179.py:4: Fu
tureWarning: Series.__getitem__ treating keys as positions is deprecat
ed. In a future version, integer keys will always be treated as labels
(consistent with DataFrame behavior). To access a value by position, u
se `ser.iloc[pos]`
  'Close'      : lambda x: x[-1],
C:\Users\CM8738\AppData\Local\Temp\ipykernel_21260\1017184179.py:1: Fu
tureWarning: The provided callable <function sum at 0x0000027A7F8B18A0
> is currently using SeriesGroupBy.sum. In a future version of pandas,
the provided callable will be used directly. To keep current behavior
pass the string "sum" instead.
  daily = bitcoin.resample('1D').apply({'Open'       : lambda x: x[0],
```

Out[17]:

	Open	High	Low	Close	Volume	USD Volume

Notice that the `Close` of one day still matches the `Open` of the next

Notice that the `Close` of one day still matches the `Open` of the next

Moreover, we have deliberately left out `Middle` since we want to recalculate it on a daily basis, as it will be different from what we calculate on a minute basis!

Notice that the `Close` of one day still matches the `Open` of the next

Moreover, we have deliberately left out `Middle` since we want to recalculate it on a daily basis, as it will be different from what we calculate on a minute basis!

We will leave the `USD Volume` based on minute data since that will give a more accurate estimate and this column is not exact since we don't have the trades list

Let's recalculate `Middle` on this new daily data

Let's recalculate `Middle` on this new daily data

In [18]:
```
daily['Middle'] = (daily['High'] + daily['Low']) / 2
daily
```

Out[18]:

Date	Open	High	Low	Close	Volume	USD Volume
2017-01-01	974.55	1003.54	965.00	998.57	1070.013119	1.061594e+06
2017-01-02	998.57	1035.90	995.00	1020.23	1449.193752	1.476088e+06
2017-01-03	1020.23	1038.26	1014.06	1035.03	2298.844903	2.356241e+06
2017-01-04	1035.03	1155.54	1035.02	1143.16	3252.704150	3.548748e+06
2017-01-05	1143.16	1167.16	879.00	1004.84	6034.569411	5.998187e+06
...	...	...	...	...	...	...
2017-12-27	15830.86	16500.00	14521.00	15418.95	9255.819566	1.431533e+08
2017-12-28	15418.95	15499.99	13514.81	14488.61	11001.121472	1.565894e+08
2017-12-29	14488.61	15115.10	13949.50	14574.99	6391.623337	9.279851e+07
2017-12-30	14574.99	14607.77	12481.44	12675.68	8734.652933	1.158837e+08
2017-12-31	12675.68	14191.20	12511.00	13820.26	8176.382068	1.082456e+08

365 rows × 7 columns

Plotting TSD

Typically, the first plot used for this type of data is a line plot

Typically, the first plot used for this type of data is a line plot

In [19]:
```
import plotly.express as px
fig = px.line(daily.reset_index(), x='Date', y='Middle')
fig.show()
```

In [41]:

```
import plotly.graph_objects as go
import pandas as pd
from datetime import datetime

daily.reset_index(inplace=True)

fig = go.Figure(data=[go.Candlestick(x=daily['Date'],
                open=daily['Open'],
                high=daily['High'],
                low=daily['Low'],
                close=daily['Close'])])

fig.show()
```


20k
15k
10k
5k
0
Jan 2017
Mar 2017
May 2017
Jul 2017
Sep 2017
Nov 2017

Loading a TSD in Tablaeu

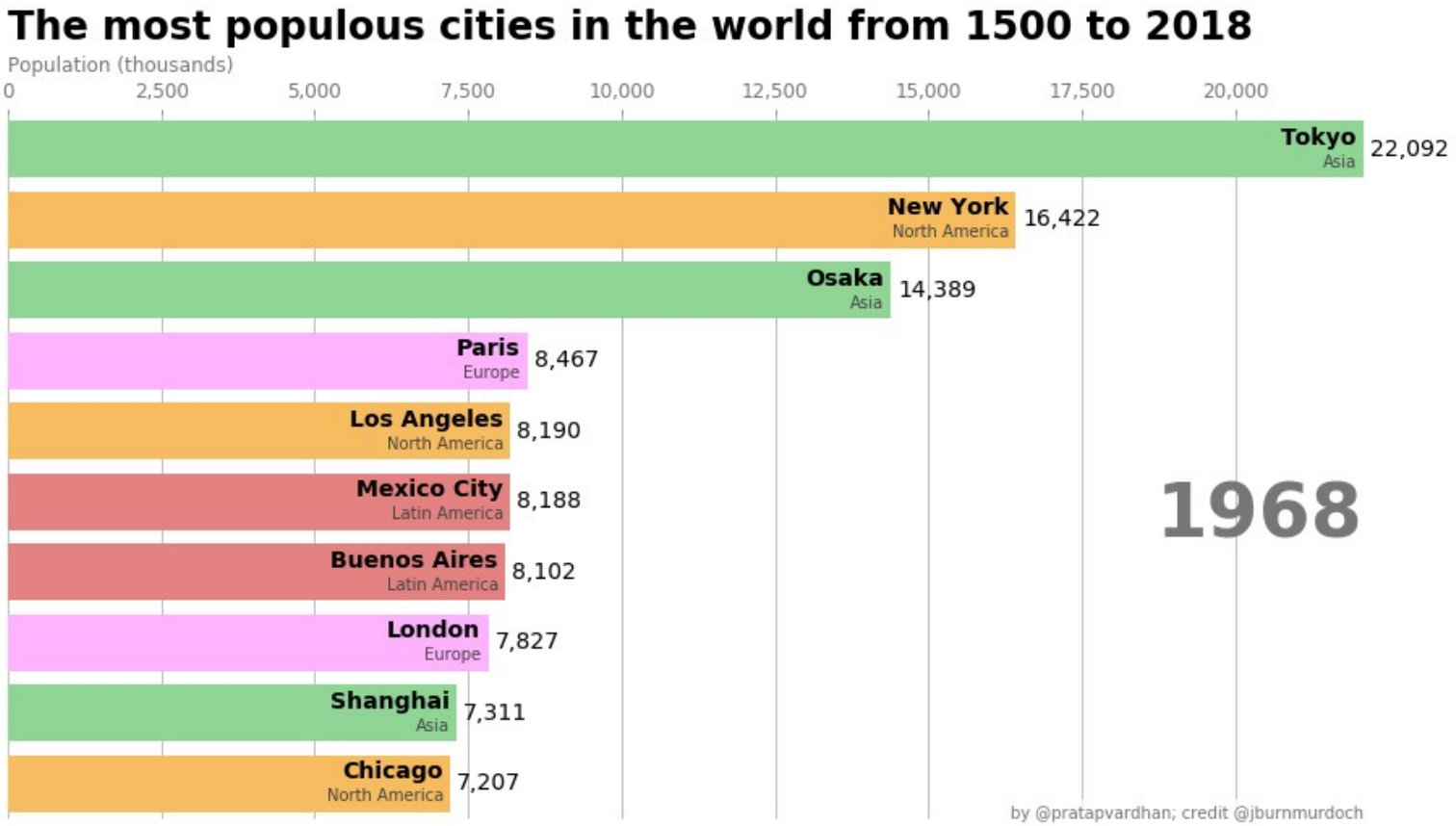
How does it look in Tableau? How to do it Option 1 and Option 2

Bart Chart Race

However, if we had the data of one more cryptocurrency (yes, there's more crypto other than bitcoin!), we could compare their trends and see which one is trading higher!

And that is why people came up with... the BAR CHART RACE (BCR)!

And that is why people came up with... the BAR CHART RACE (BCR)!



Although recently, I've seen more LINE races! Like this one

Let's see how this example was produced using Python

Let's see how this example was produced using Python

We will use a file called `city_populations.txt` which you can download from here

Let's see how this example was produced using Python

We will use a file called `city_populations.txt` which you can download from here

This file contains population data of different world cities through time

In [21]:
```
df = pd.read_csv('https://gist.githubusercontent.com/johnburnmurdoch/4199dbe55
df
```

Out[21]:

	name	group	year	value	subGroup	city_id	lastValue	lat
0	Agra	India	1575	200.0	India	Agra - India	200.0	27.18333
1	Agra	India	1576	212.0	India	Agra - India	200.0	27.18333
2	Agra	India	1577	224.0	India	Agra - India	212.0	27.18333
3	Agra	India	1578	236.0	India	Agra - India	224.0	27.18333
4	Agra	India	1579	248.0	India	Agra - India	236.0	27.18333
...	...	...	...	...	...	...	...	...
6247	Vijayanagar	India	1561	480.0	India	Vijayanagar - India	480.0	15.33500
6248	Vijayanagar	India	1562	480.0	India	Vijayanagar - India	480.0	15.33500
6249	Vijayanagar	India	1563	480.0	India	Vijayanagar - India	480.0	15.33500
6250	Vijayanagar	India	1564	480.0	India	Vijayanagar - India	480.0	15.33500
6251	Vijayanagar	India	1565	480.0	India	Vijayanagar - India	480.0	15.33500

6252 rows × 9 columns

Let's clean this a little bit...

Let's clean this a little bit...

In [22]:
```
# These are the only columns we need
df = df[['name','group','year','value']]
df
```

Out[22]:

	name	group	year	value
0	Agra	India	1575	200.0
1	Agra	India	1576	212.0
2	Agra	India	1577	224.0
3	Agra	India	1578	236.0
4	Agra	India	1579	248.0
...	...	...	...	...
6247	Vijayanagar	India	1561	480.0
6248	Vijayanagar	India	1562	480.0
6249	Vijayanagar	India	1563	480.0
6250	Vijayanagar	India	1564	480.0
6251	Vijayanagar	India	1565	480.0

6252 rows × 4 columns

We will change the names of the columns to something more coherent!

We will change the names of the columns to something more coherent!

In [23]:
```
df = df.rename(columns={"name": "City",
                        "group": "Continent",
                        "year": "Year",
                        "value":"Population"})
df
```

Out[23]:

	City	Continent	Year	Population
0	Agra	India	1575	200.0
1	Agra	India	1576	212.0
2	Agra	India	1577	224.0
3	Agra	India	1578	236.0
4	Agra	India	1579	248.0
...	...	...	...	...
6247	Vijayanagar	India	1561	480.0
6248	Vijayanagar	India	1562	480.0
6249	Vijayanagar	India	1563	480.0
6250	Vijayanagar	India	1564	480.0
6251	Vijayanagar	India	1565	480.0

6252 rows × 4 columns

Notice that in the `Continent` column we have India for some cities

Notice that in the `Continent` column we have India for some cities

If I'm correct, India is a sub-continent rather than a continent!

Notice that in the `Continent` column we have India for some cities

If I'm correct, India is a sub-continent rather than a continent!

Let's see the continents listed here...

Notice that in the `Continent` column we have India for some cities

If I'm correct, India is a sub-continent rather than a continent!

Let's see the continents listed here...

In [24]:
```
set(df['Continent'])
```

Out[24]:
```
{'Asia', 'Europe', 'India', 'Latin America', 'Middle East', 'North Am
erica'}
```

Notice that in the `Continent` column we have India for some cities

If I'm correct, India is a sub-continent rather than a continent!

Let's see the continents listed here...

In [24]:
```
set(df['Continent'])
```

Out[24]:
```
{'Asia', 'Europe', 'India', 'Latin America', 'Middle East', 'North Am
erica'}
```

Middle East and Latin America are also listed as continents!

In [25]:
```
# Cities in the "Indian continent"
print(set(df[df['Continent']=='India']['City']))
```

```
{'Kolkatta', 'Ahmedabad', 'Agra', 'Gauda', 'Bijapur', 'Mumbai', 'Delhi
', 'Lucknow', 'Vijayanagar', 'Cuttack'}
```

In [25]:
```
# Cities in the "Indian continent"
print(set(df[df['Continent']=='India']['City']))
```

```
{'Kolkatta', 'Ahmedabad', 'Agra', 'Gauda', 'Bijapur', 'Mumbai', 'Delhi
', 'Lucknow', 'Vijayanagar', 'Cuttack'}
```

In [26]:
```
# Cities with "Middle East" continent
print(set(df[df['Continent']=='Middle East']['City']))
```

```
{'Mashhad', 'Esfahan', 'Cairo', 'Fez', 'Tabriz'}
```

In [25]:
```
# Cities in the "Indian continent"
print(set(df[df['Continent']=='India']['City']))
```

```
{'Kolkatta', 'Ahmedabad', 'Agra', 'Gauda', 'Bijapur', 'Mumbai', 'Delhi
', 'Lucknow', 'Vijayanagar', 'Cuttack'}
```

In [26]:
```
# Cities with "Middle East" continent
print(set(df[df['Continent']=='Middle East']['City']))
```

```
{'Mashhad', 'Esfahan', 'Cairo', 'Fez', 'Tabriz'}
```

In [27]:
```
# Cities with "Latin America" continent
print(set(df[df['Continent']=='Latin America']['City']))
```

```
{'Sao Paulo', 'Rio de Janeiro', 'Buenos Aires', 'Mexico City'}
```

Let's first replace `India -> Asia`

Let's first replace `India -> Asia`

In [28]:
```
df['Continent'] = df['Continent'].replace('India', 'Asia')
```

With the other two is slightly different, as these categories contain cities from different continents!

With the other two is slightly different, as these categories contain cities from different continents!

In fact, I may need to change "one by one"

In [29]:

```
import warnings;
warnings.simplefilter('ignore')
df['Continent'][df['City']=='Fez']='Africa'
df['Continent'][df['City']=='Cairo']='Africa'
df['Continent'] = df['Continent'].replace('Middle East', 'Asia')
df['Continent'][df['City']=='Mexico City']='North America'
df['Continent'] = df['Continent'].replace('Latin America', 'South America')
```

In [30]:
```
# Show that the replacement has been successful
df[df['City']=='Cairo']
```

Out[30]:

	City	Continent	Year	Population
1055	Cairo	Africa	1500	400.0
1056	Cairo	Africa	1501	399.2
1057	Cairo	Africa	1502	398.4
1058	Cairo	Africa	1503	397.6
1059	Cairo	Africa	1504	396.8
...	...	...	...	...
1193	Cairo	Africa	2016	19131.2
1194	Cairo	Africa	2017	19490.4
1195	Cairo	Africa	2018	19849.6
1196	Cairo	Africa	2019	20208.8
1197	Cairo	Africa	2020	20568.0

143 rows × 4 columns

I will save the new dataset locally

I will save the new dataset locally

In [31]:
```
df.to_csv(r'city_populations_updated.csv',index=False)
```

Now, let's try to do our own BCR with `Python` and the `matplotlib` package

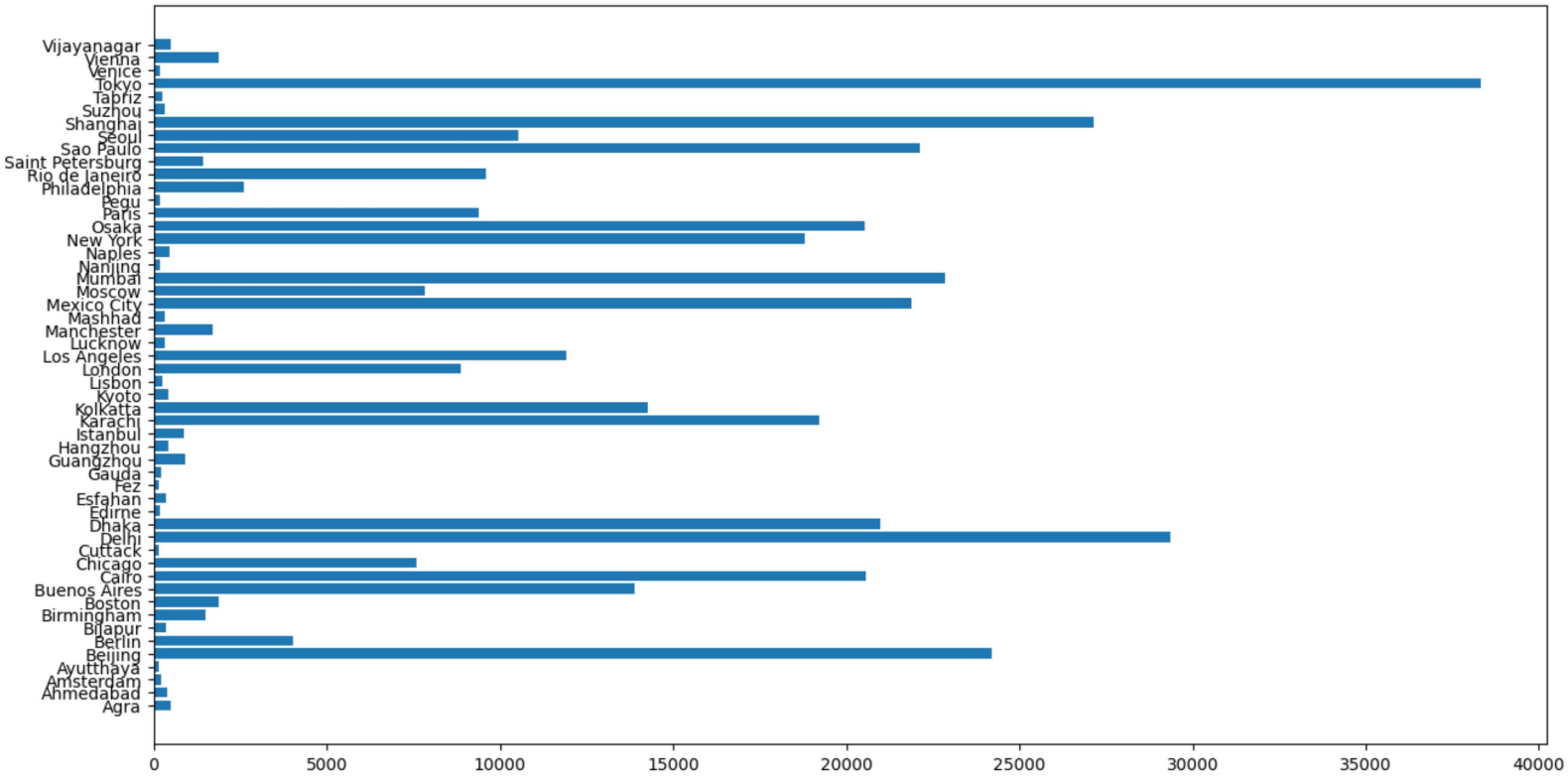
Now, let's try to do our own BCR with `Python` and the `matplotlib` package

In [32]:
```
# Importing the necessary packages
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
```

If we just plot **city vs population** using `ax.barh`, we will see the last value of population for each of the cities

In [33]:
```
fig, ax = plt.subplots(figsize=(15, 8))
ax.barh(df['City'], df['Population'])
```

Out[33]:
```
<BarContainer object of 6252 artists>
```



To do a BCR, we need to define a given year and sort the cities from most to least populated (for that year)

To do a BCR, we need to define a given year and sort the cities from most to least populated (for that year)

In [34]:
```
current_year = 1500
df2 = (df[df['Year'].eq(current_year)]
       .sort_values(by='Population',ascending=False))
df2
```

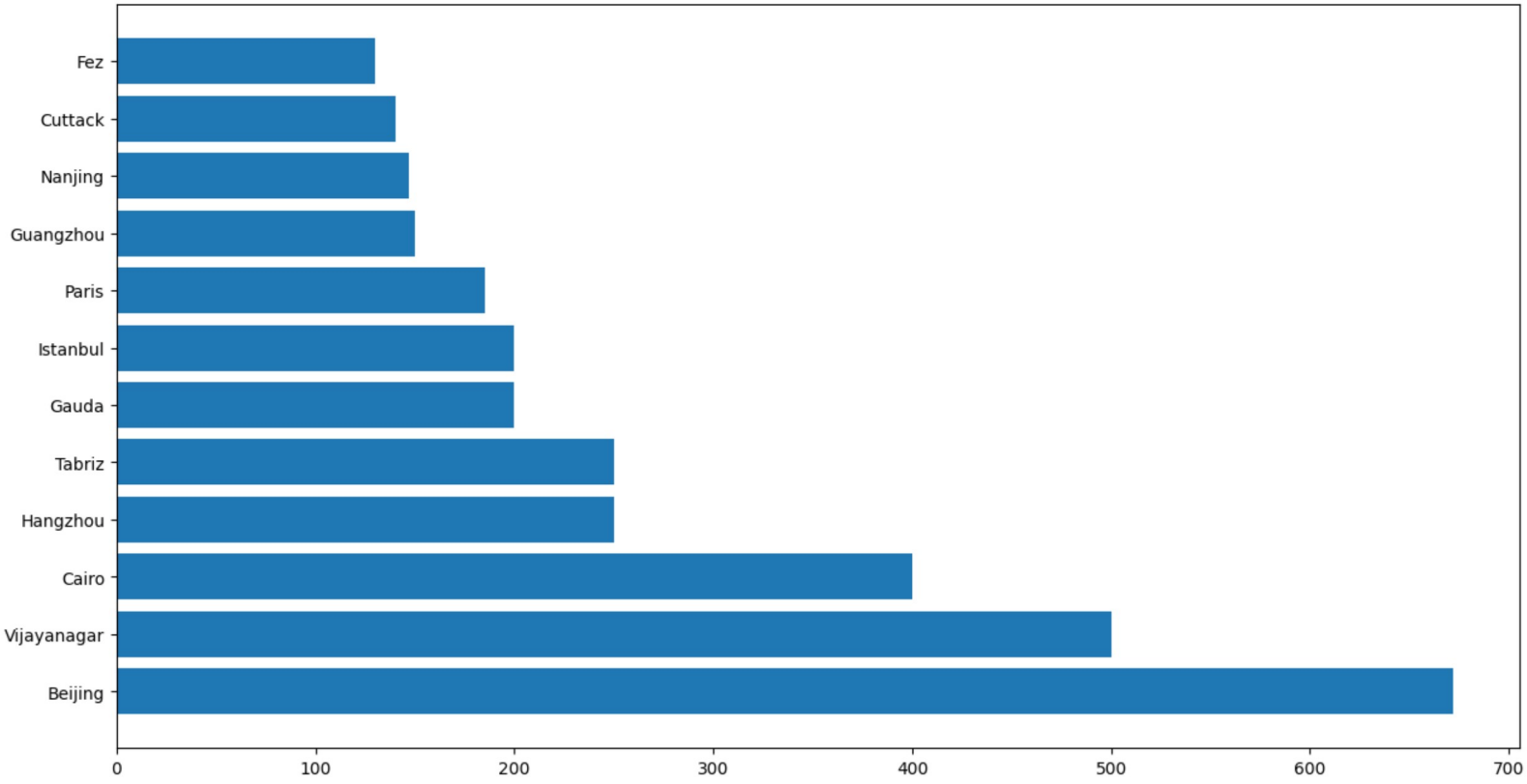
Out[34]:

	City	Continent	Year	Population
276	Beijing	Asia	1500	672.0
6186	Vijayanagar	Asia	1500	500.0
1055	Cairo	Africa	1500	400.0
1791	Hangzhou	Asia	1500	250.0
5595	Tabriz	Asia	1500	250.0
1503	Gauda	Asia	1500	200.0
2142	Istanbul	Europe	1500	200.0
4682	Paris	Europe	1500	185.0
1537	Guangzhou	Asia	1500	150.0
3751	Nanjing	Asia	1500	147.0
1289	Cuttack	Asia	1500	140.0
1496	Fez	Africa	1500	130.0

Notice that in this dataset, we don't have all cities for each year so we would get different charts

In [35]:
```
# Plotting a basic bar chart
fig, ax = plt.subplots(figsize=(15, 8))
ax.barh(df2['City'], df2['Population'])
```

Out[35]:
```
<BarContainer object of 12 artists>
```



To add colours and labels (for each continent), we will use a dictionary that pairs cities with their respective continent

In [36]:

```
colours = dict(zip(
    ['Europe', 'Asia', 'South America', 'North America', 'Africa'],
    ['#adb0ff', '#90d595','#aafbff', '#f7bb5f', '#eafb50']))
group_lk = df.set_index('City')['Continent'].to_dict()
group_lk
```

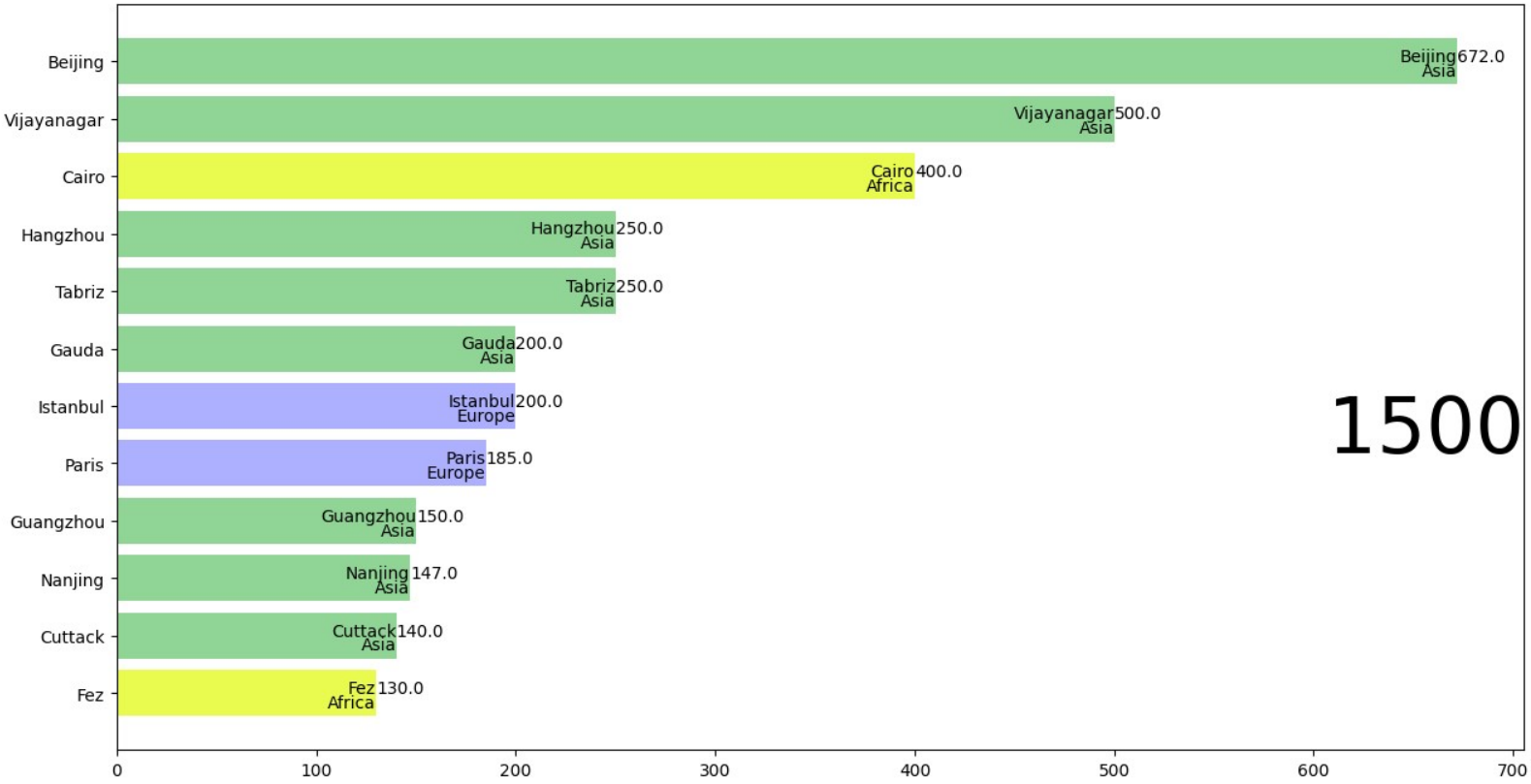
Out[36]:

```
{'Agra': 'Asia',
 'Ahmedabad': 'Asia',
 'Amsterdam': 'Europe',
 'Ayutthaya': 'Asia',
 'Beijing': 'Asia',
 'Berlin': 'Europe',
 'Bijapur': 'Asia',
 'Birmingham': 'Europe',
 'Boston': 'North America',
 'Buenos Aires': 'South America',
 'Cairo': 'Africa',
 'Chicago': 'North America',
 'Cuttack': 'Asia',
 'Delhi': 'Asia',
 'Dhaka': 'Asia',
 'Edirne': 'Europe',
 'Esfahan': 'Asia',
 'Fez': 'Africa',
 'Gauda': 'Asia',
 'Guangzhou': 'Asia',
 'Hangzhou': 'Asia',
 'Istanbul': 'Europe',
 'Karachi': 'Asia',
 'Kolkatta': 'Asia',
 'Kyoto': 'Asia',
 'Lisbon': 'Europe',
 'London': 'Europe',
 'Los Angeles': 'North America',
 'Lucknow': 'Asia',
 'Manchester': 'Europe',
 'Mashhad': 'Asia',
```

Now we can use this pairing to get some colours in `df2`

In [37]:
```
fig, ax = plt.subplots(figsize=(15, 8))
df2 = df2[::-1]   # flip values from top to bottom
# pass colours values to `color=`
ax.barh(df2['City'], df2['Population'], color=[colours[group_lk[x]] for x in d
# iterate over the values to plot labels and values (Tokyo, Asia, 38194.2)
for i, (value, name) in enumerate(zip(df2['Population'], df2['City'])):
    ax.text(value, i,     name,            ha='right')
    ax.text(value, i-.25, group_lk[name],  ha='right')
    ax.text(value, i,     value,           ha='left')
# Add year right middle portion of canvas
ax.text(1, 0.4, current_year, transform=ax.transAxes, size=46, ha='right')
```

Out[37]:
```
Text(1, 0.4, '1500')
```

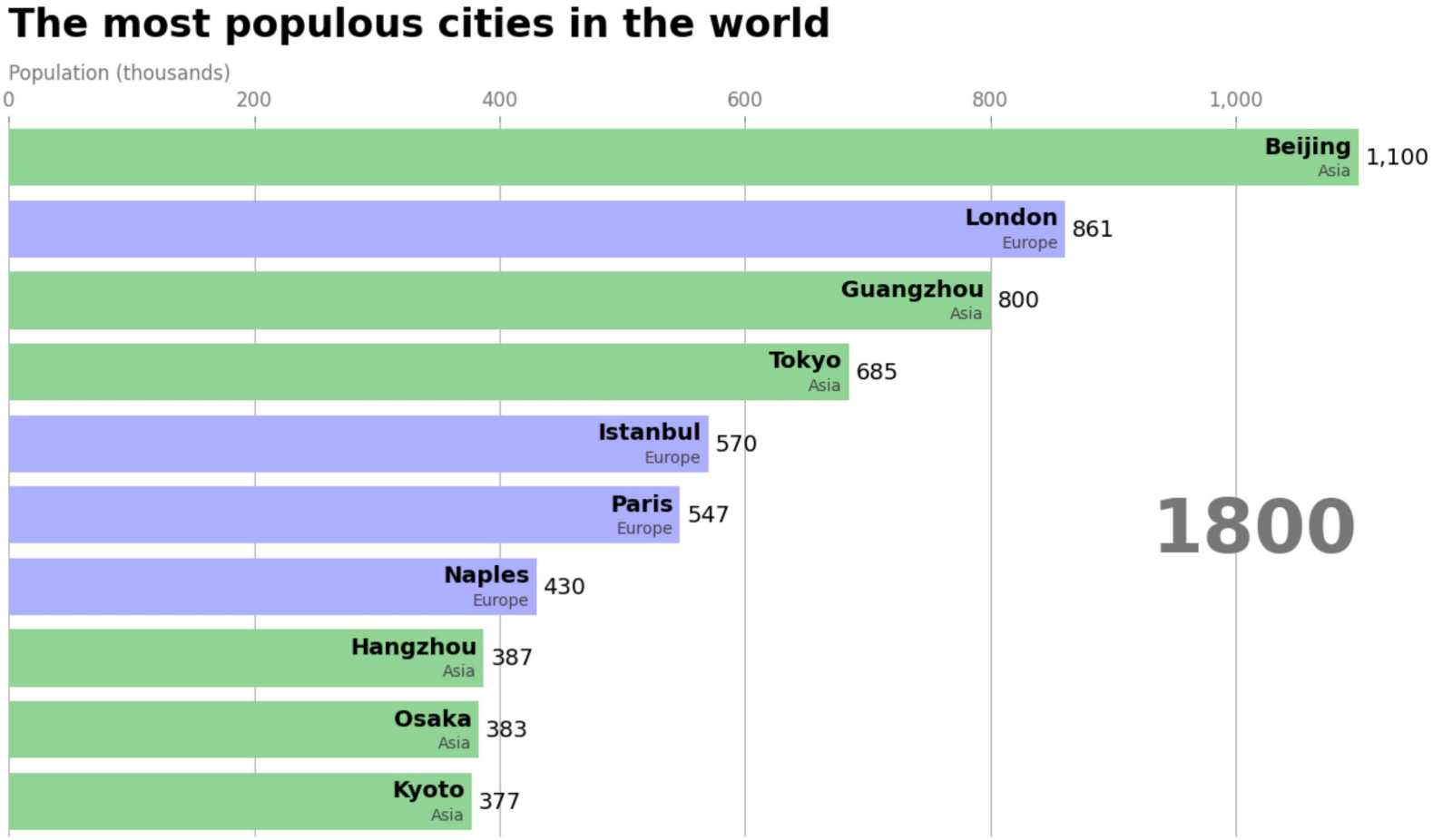

Beijing
Vijayanagar
Cairo
Hangzhou
Tabriz
Gauda
Istanbul
Paris
Guangzhou
Nanjing
Cuttack
Fez
Beijing 672.0
Asia
Vijayanagar 500.0
Asia
Cairo 400.0
Africa
Hangzhou 250.0
Asia
Tabriz 250.0
Asia
Gauda 200.0
Asia
Istanbul 200.0
Europe
Paris 185.0
Europe
Guangzhou 150.0
Asia
Nanjing 147.0
Asia
Cuttack 140.0
Asia
Fez 130.0
Africa
1500
0
100
200
300
400
500
600
700

This function will draw a more "stylish" function to create the last chart

In [38]:

```
# Frankly, I don't know what half of it does...
def draw_barchart(year):
    dff = df[df['Year'].eq(year)].sort_values(by='Population', ascending=True)
    ax.clear()
    ax.barh(dff['City'], dff['Population'], color=[colours[group_lk[x]] for x 
    dx = dff['Population'].max() / 200
    for i, (value, name) in enumerate(zip(dff['Population'], dff['City'])):
        ax.text(value-dx, i,     name,           size=14, weight=600, ha='righ
        ax.text(value-dx, i-.25, group_lk[name], size=10, color='#444444', ha=
        ax.text(value+dx, i,     f'{value:,.0f}',  size=14, ha='left',  va='ce
    # ... polished styles
    ax.text(1, 0.4, year, transform=ax.transAxes, color='#777777', size=46, ha
    ax.text(0, 1.06, 'Population (thousands)', transform=ax.transAxes, size=12
    ax.xaxis.set_major_formatter(ticker.StrMethodFormatter('{x:,.0f}'))
    ax.xaxis.set_ticks_position('top')
    ax.tick_params(axis='x', colors='#777777', labelsize=12)
    ax.set_yticks([])
    ax.margins(0, 0.01)
    ax.grid(which='major', axis='x', linestyle='-')
    ax.set_axisbelow(True)
    ax.text(0, 1.12, 'The most populous cities in the world ',
            transform=ax.transAxes, size=24, weight=600, ha='left')
    plt.box(False)
```

In [39]:
```
# Now we execute the function
fig, ax = plt.subplots(figsize=(15, 8))
draw_barchart(1800)
```

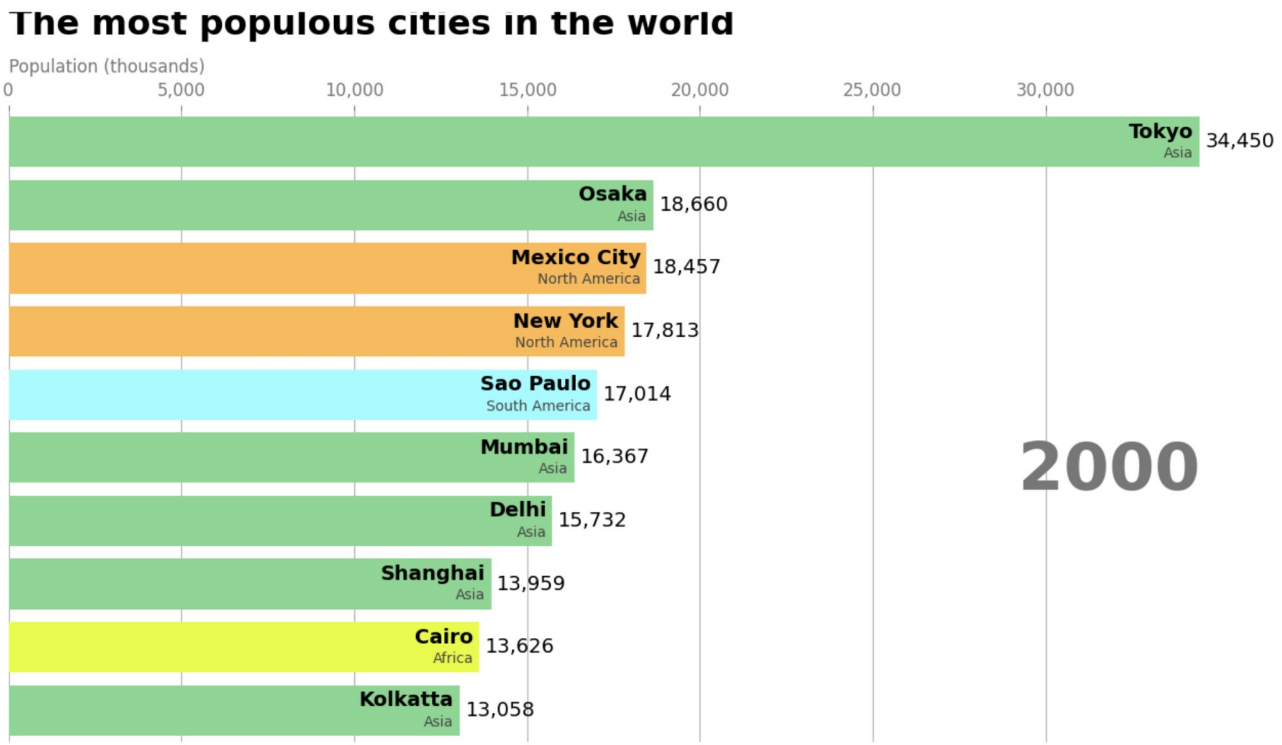


The most important part is to animate the plot for a range of years

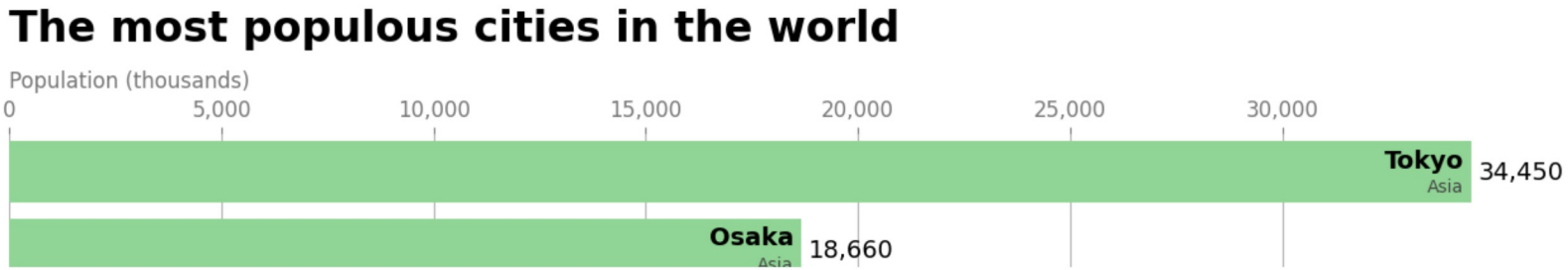
In [40]:

```
import matplotlib.animation as animation
from IPython.display import HTML
fig, ax = plt.subplots(figsize=(15, 8))
animator = animation.FuncAnimation(fig, draw_barchart, frames=range(2000,2020)
HTML(animator.to_jshtml())
```

Out[40]:



Once Loop Reflect



Lab

There are two parts, one for TSD and another for BCR

There are two parts, one for TSD and another for BCR

The first one consists on following the instructions to treat TSD data

The second one has the code seen in class to do the BCR

For that one, I want you to get creative!

- Can you change colours, styles, names, etc.?
- Can you skip some years?
- Can you get rid of the static plot that gets printed after the animation?
- Can you show not only the top 12 cities?
- Can you show all cities at the same time?
- Can you put the years of the range in the title of the chart?
- Can you run it from 1500 to 2020?
- Can you make the chart go back in time?

Alternatively you can download the dataset from Moodle and see if other tools are able to these things!

- You can try to install this Python Module which in fact looks easier and better than my proposed method!
    - Although it doesn't allow you to include the continent to colour the bars!
- Using an app by Microsoft
- Using online services such as Fluorish or Highcharts
- Or with Excel!