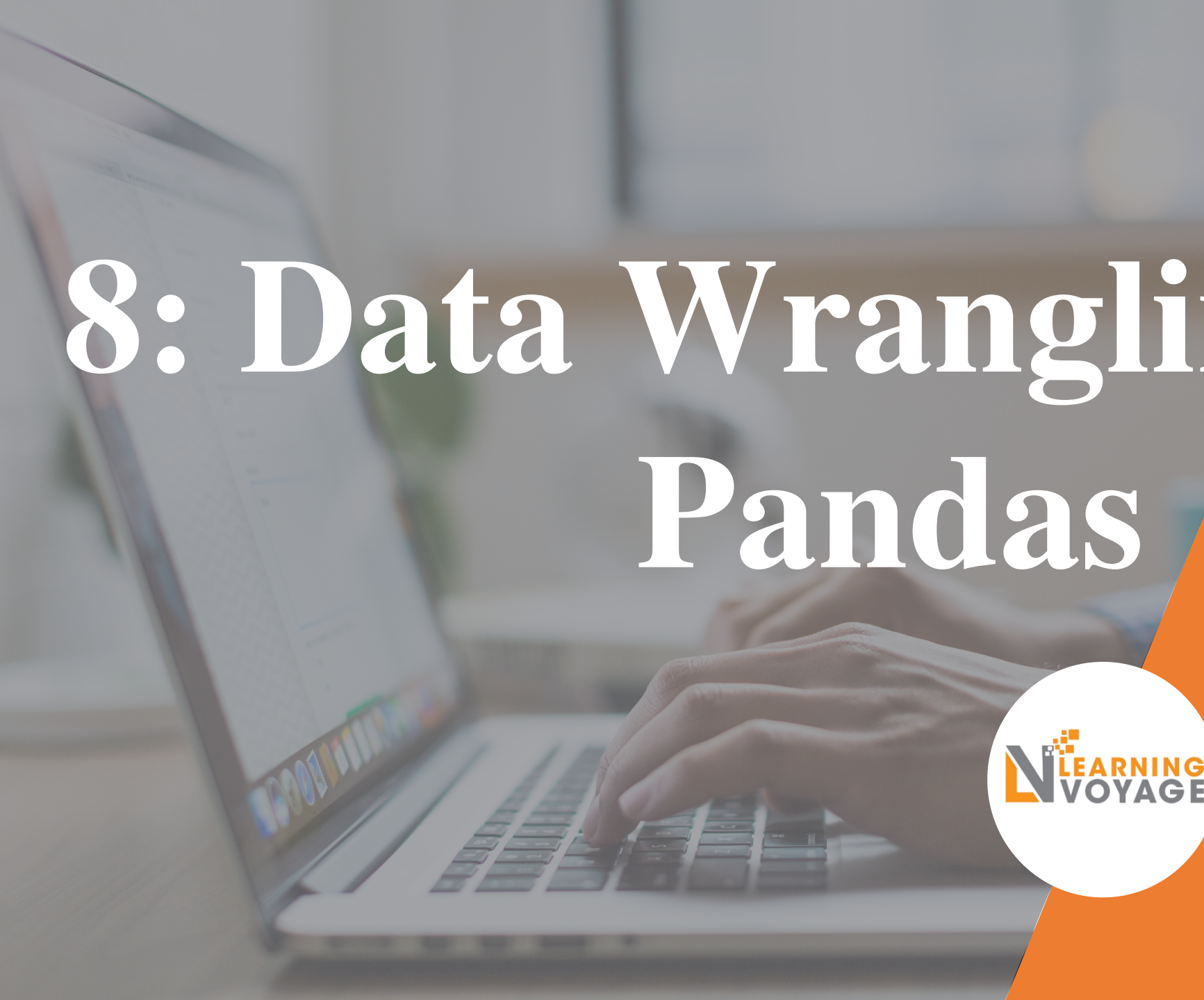
DAY 4

LEARNING VOYAGE



8: Data Wrangling with Pandas

LEARNING VOYAGE

Data Wrangling with Pandas

In this lesson, we will cover the following topics:

- Understanding data wrangling
- Exploring an API to find and collect temperature data
- Cleaning data
- Reshaping data
- Handling duplicate, missing, or invalid data

lesson materials

- The following files are in the data/ directory:

File	Description	Source
`bitcoin.csv`	Daily opening, high, low, and closing price of bitcoin, along with volume traded and market capitalization for 2017 through 2018.	CoinMarketCap
`dirty_data.csv`	2018 weather data for New York City, manipulated to introduce data issues.	Modified version of the data from the NCEI API's GHCND dataset.
`long_data.csv`	Long format temperature data for New York City in October 2018 from the Boonton 1 station, containing daily temperature at time of observation, minimum temperature, and maximum temperature.	The NCEI API's GHCND dataset
`nyc_temperatures.csv`	Temperature data for New York City in October 2018 measured from LaGuardia airport, containing daily minimum, maximum, and average temperature.	The NCEI API's GHCND dataset
`sp500.csv`	Daily opening, high, low, and closing price of the S&P 500 stock index, along with volume traded and adjusted close for 2017 through 2018.	The `stock_analysis` package (see *Chapter 7, Financial Analysis – Bitcoin and the Stock Market*).
`wide_data.csv`	Wide format temperature data for New York City in October 2018 from the Boonton 1 station, containing daily temperature at time of observation, minimum temperature, and maximum temperature.	The NCEI API's GHCND dataset

Understanding data wrangling

- When we perform data wrangling, we are taking our input data from its original state and putting it in a format where we can perform meaningful analysis on it.

- Data manipulation is another way to refer to this process.

- There is no set list of operations; the only goal is that the data post-wrangling is more useful to us than when we started.

Data cleaning

Some essential data cleaning tasks to master include the following:

- Renaming
- Sorting and reordering
- Data type conversions
- Handling duplicate data
- Addressing missing or invalid data
- Filtering to the desired subset of data

Data transformation

- Often, people will record and present data in wide format, but there are certain visualizations that require the data to be in long format:

variables

observations

	date	TMAX	TMIN	TOBS
0	2018-10-01	21.1	8.9	13.9
1	2018-10-02	23.9	13.9	17.2
2	2018-10-03	25.0	15.6	16.1
3	2018-10-04	22.8	11.7	11.7
4	2018-10-05	23.3	11.7	18.9
5	2018-10-06	20.0	13.3	16.1

repeated values for **date** column

variable names

variable values

	date	datatype	value
0	2018-10-01	TMAX	21.1
1	2018-10-01	TMIN	8.9
2	2018-10-01	TOBS	13.9
3	2018-10-02	TMAX	23.9
4	2018-10-02	TMIN	13.9
5	2018-10-02	TOBS	17.2

Data transformation

- read in the CSV files containing wide and long format data:

```
>>> import matplotlib.pyplot as plt
>>> import pandas as pd
>>> wide_df = \
...     pd.read_csv('data/wide_data.csv', parse_dates=['date'])
>>> long_df = pd.read_csv(
...     'data/long_data.csv',
...     usecols=['date', 'datatype', 'value'],
...     parse_dates=['date']
... )[['date', 'datatype', 'value']] # sort columns
```

Data transformation

- Let's look at the top six observations from the wide format data in wide_df:

```
>>> wide_df.head(6)
```

Data transformation

- Each column contains the top six observations of a specific class of temperature data in degrees Celsius—maximum temperature (TMAX), minimum temperature (TMIN), and temperature at the time of observation (TOBS)—at a daily frequency:

	date	TMAX	TMIN	TOBS
0	2018-10-01	21.1	8.9	13.9
1	2018-10-02	23.9	13.9	17.2
2	2018-10-03	25.0	15.6	16.1
3	2018-10-04	22.8	11.7	11.7
4	2018-10-05	23.3	11.7	18.9
5	2018-10-06	20.0	13.3	16.1

Data transformation

- When working with wide format data, we can easily grab summary statistics on this data by using the describe() method.

- Note that while older versions of pandas treated datetimes as categorical, pandas is moving toward treating them as numeric, so we pass datetime_is_numeric=True to suppress the warning:

```
>>> wide_df.describe(include='all', datetime_is_numeric=True)
```

Data transformation

- With hardly any effort on our part, we get summary statistics for the dates, maximum temperature, minimum temperature, and temperature at the time of observation:

	date	TMAX	TMIN	TOBS
count	31	31.000000	31.000000	31.000000
mean	2018-10-16 00:00:00	16.829032	7.561290	10.022581
min	2018-10-01 00:00:00	7.800000	-1.100000	-1.100000
25%	2018-10-08 12:00:00	12.750000	2.500000	5.550000
50%	2018-10-16 00:00:00	16.100000	6.700000	8.300000
75%	2018-10-23 12:00:00	21.950000	13.600000	16.100000
max	2018-10-31 00:00:00	26.700000	17.800000	21.700000
std	NaN	5.714962	6.513252	6.596550

Data transformation

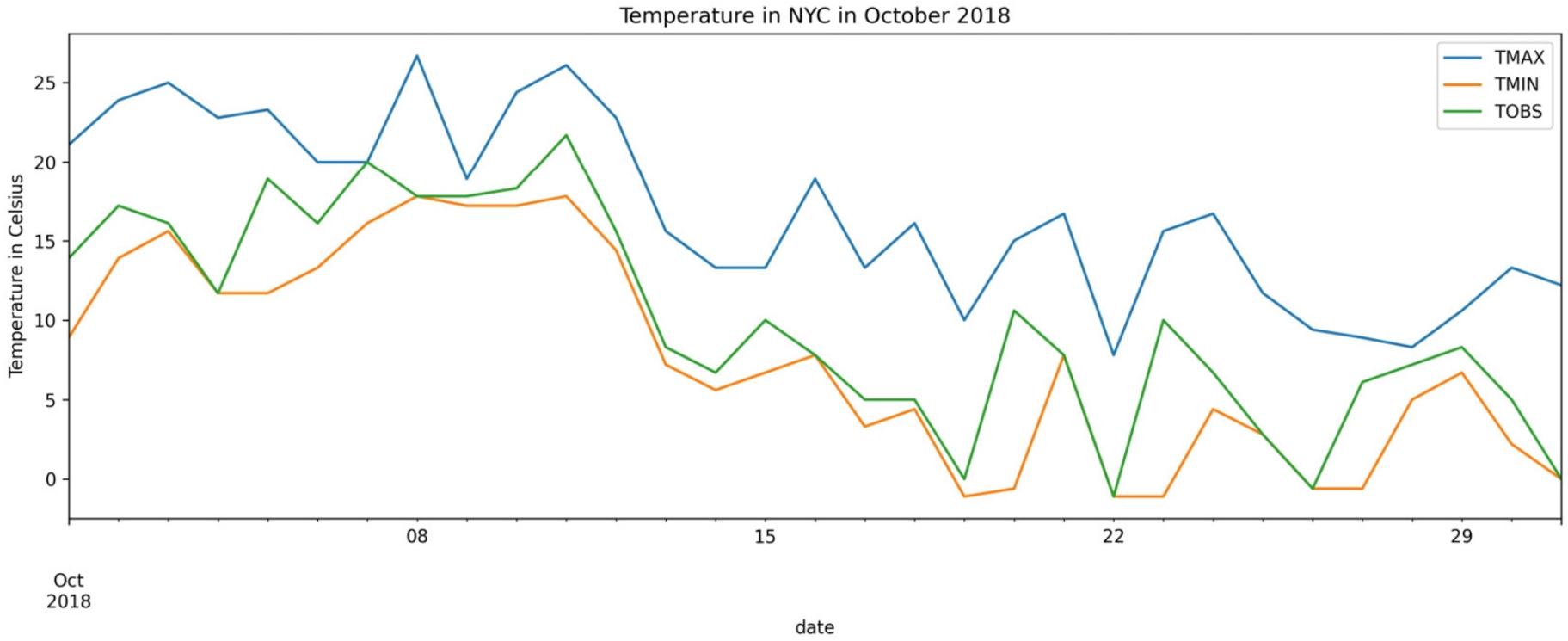
- As we discussed previously, the summary data in the preceding table is easy to obtain and is informative.

- This format can easily be plotted with pandas as well, provided we tell it exactly what we want to plot:

```
>>> wide_df.plot(
...     x='date', y=['TMAX', 'TMIN', 'TOBS'], figsize=(15, 5),
...     title='Temperature in NYC in October 2018'
... ).set_ylabel('Temperature in Celsius')
>>> plt.show()
```

Data transformation

- Pandas plots the daily maximum temperature, minimum temperature, and temperature at the time of observation as their own lines on a single line plot:



Data transformation

- We can look at the top six rows of the long format data in long_df to see the difference between wide format and long format data:

```
>>> long_df.head(6)
```

Data transformation

- Notice that we now have three entries for each date, and the datatype column tells us what the data in the value column is for that row:

	date	datatype	value
0	2018-10-01	TMAX	21.1
1	2018-10-01	TMIN	8.9
2	2018-10-01	TOBS	13.9
3	2018-10-02	TMAX	23.9
4	2018-10-02	TMIN	13.9
5	2018-10-02	TOBS	17.2

Data transformation

- If we try to get summary statistics, like we did with the wide format data, the result isn't as helpful:

```
>>> long_df.describe(include='all', datetime_is_numeric=True)
```

Data transformation

- This means that this summary data is not very helpful:

	date	datatype	value
count	93	93	93.000000
unique	NaN	3	NaN
top	NaN	TOBS	NaN
freq	NaN	31	NaN
mean	2018-10-16 00:00:00	NaN	11.470968
min	2018-10-01 00:00:00	NaN	-1.100000
25%	2018-10-08 00:00:00	NaN	6.700000
50%	2018-10-16 00:00:00	NaN	11.700000
75%	2018-10-24 00:00:00	NaN	17.200000
max	2018-10-31 00:00:00	NaN	26.700000
std	NaN	NaN	7.362354

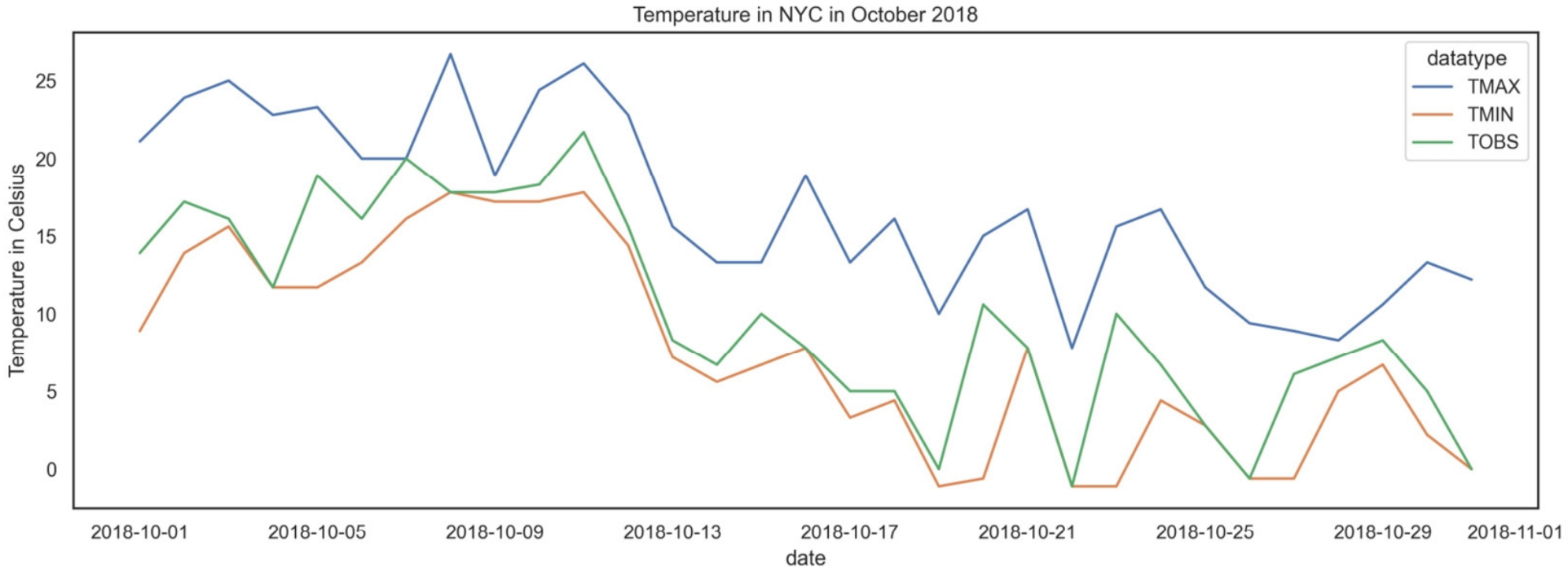
Data transformation

- Pandas expects its data for plotting to be in wide format, so, to easily make the same plot that we did with the wide format data, we must use another plotting library, called seaborn, which we will cover in Plotting with Seaborn and Customization Techniques:

```
>>> import seaborn as sns
>>> sns.set(rc={'figure.figsize': (15, 5)}, style='white')
>>> ax = sns.lineplot(
...     data=long_df, x='date', y='value', hue='datatype'
... )
>>> ax.set_ylabel('Temperature in Celsius')
>>> ax.set_title('Temperature in NYC in October 2018')
>>> plt.show()
```

Data transformation

- Seaborn can subset based on the datatype column to give us individual lines for the daily maximum temperature, minimum temperature, and temperature at the time of observation:



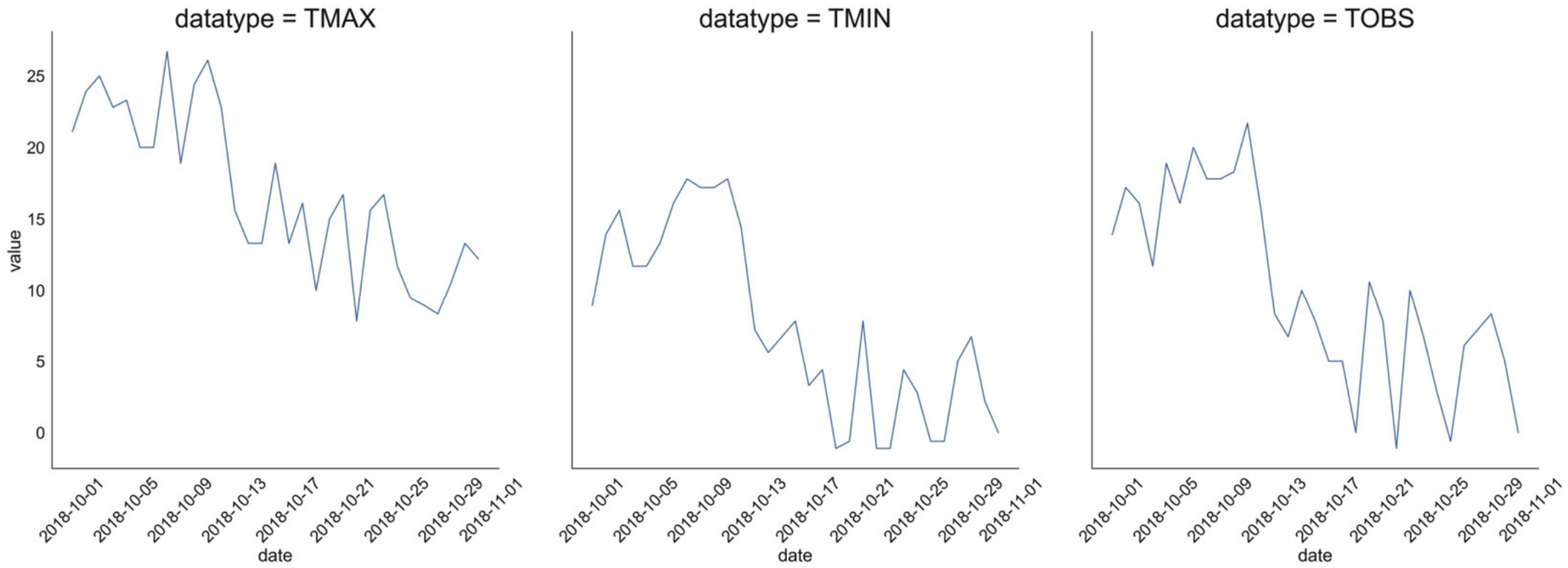
Data transformation

- Seaborn lets us specify the column to use for hue, which colored the lines in Figure by the temperature type.
- We aren't limited to this, though; with long format data, we can easily facet our plots:

```
>>> sns.set(
...     rc={'figure.figsize': (20, 10)},
...     style='white', font_scale=2
... )
>>> g = sns.FacetGrid(long_df, col='datatype', height=10)
>>> g = g.map(plt.plot, 'date', 'value')
>>> g.set_titles(size=25)
>>> g.set_xticklabels(rotation=45)
>>> plt.show()
```

Data transformation

- Seaborn can use long format data to create subplots for each distinct value in the datatype column:



Data enrichment

The following are ways to enhance our data using the original data:

- **Adding new columns**: Using functions on the data from existing columns to create new values.
- **Binning**: Turning continuous data or discrete data with many distinct values into buckets, which makes the column discrete while letting us control the number of possible values in the column.
- **Aggregating**: Rolling up the data and summarizing it.
- **Resampling**: Aggregating time series data at specific intervals.

Exploring an API to find and collect temperature

- For this section, we will be working in the 2-using_the_weather_api.ipynb notebook to request temperature data from the NCEI API.

- As we learned in Working with Pandas DataFrames, we can use the requests library to interact with APIs.

- In the following code block, we import the requests library and create a convenience function for making the requests to a specific endpoint, sending our token along.
- You need an email to do this lab: https://www.ncdc.noaa.gov/cdo-web/token

Exploring an API to find and collect temperature

- To use this function, we need to provide a token, as indicated in bold:

```
>>> import requests
>>> def make_request(endpoint, payload=None):
...     """
...     Make a request to a specific endpoint on the
...     weather API passing headers and optional payload.
...     Parameters:
...         - endpoint: The endpoint of the API you want to
...                     make a GET request to.
...         - payload: A dictionary of data to pass along
...                    with the request.
...
...     Returns:
...         A response object.
...     """
...     return requests.get(
...         'https://www.ncdc.noaa.gov/cdo-web/'
...         f'api/v2/{endpoint}',
...         headers={'token': 'PASTE_YOUR_TOKEN_HERE'},
...         params=payload
...     )
```

Exploring an API to find and collect temperature

- Let's check which datasets have data within the date range of October 1, 2018 through today:

```
>>> response = \
...     make_request('datasets', {'startdate': '2018-10-01'})
```

Exploring an API to find and collect temperature

- Remember that we check the status_code attribute to make sure the request was successful.

- Alternatively, we can use the ok attribute to get a Boolean indicator if everything went as expected:

```
>>> response.status_code
200
>>> response.ok
True
```

Exploring an API to find and collect temperature

- Once we have our response, we can use the json() method to get the payload.

- Then, we can use dictionary methods to determine which part we want to look at:

```
>>> payload = response.json()
>>> payload.keys()
dict_keys(['metadata', 'results'])
```

Exploring an API to find and collect temperature

- The metadata portion of the JSON payload tells us information about the result, while the results section contains the actual results.

- Let's see how much data we got back, so that we know whether we can print the results or whether we should try to limit the output:

```
>>> payload['metadata']
{'resultset': {'offset': 1, 'count': 11, 'limit': 25}}
```

Exploring an API to find and collect temperature

- We got back 11 rows, so let's see what fields are in the results portion of the JSON payload.
- The results key contains a list of dictionaries.
- If we select the first one, we can look at the keys to see what fields the data contains.

- We can then reduce the output to the fields we care about:

```
>>> payload['results'][0].keys()
dict_keys(['uid', 'mindate', 'maxdate', 'name',
        'datacoverage', 'id'])
```

Exploring an API to find and collect temperature

- For our purposes, we want to look at the IDs and names of the datasets, so let's use a list comprehension to look at those only:

```
>>> [(data['id'], data['name']) for data in payload['results']]
[('GHCND', 'Daily Summaries'),
 ('GSOM', 'Global Summary of the Month'),
 ('GSOY', 'Global Summary of the Year'),
 ('NEXRAD2', 'Weather Radar (Level II)'),
 ('NEXRAD3', 'Weather Radar (Level III)'),
 ('NORMAL_ANN', 'Normals Annual/Seasonal'),
 ('NORMAL_DLY', 'Normals Daily'),
 ('NORMAL_HLY', 'Normals Hourly'),
 ('NORMAL_MLY', 'Normals Monthly'),
 ('PRECIP_15', 'Precipitation 15 Minute'),
 ('PRECIP_HLY', 'Precipitation Hourly')]
```

Exploring an API to find and collect temperature

- Here, we can print the JSON payload since it isn't that large (only nine entries):

```
>>> response = make_request(
...     'datacategories', payload={'datasetid': 'GHCND'}
... )
>>> response.status_code
200
>>> response.json()['results']
[{'name': 'Evaporation', 'id': 'EVAP'},
 {'name': 'Land', 'id': 'LAND'},
 {'name': 'Precipitation', 'id': 'PRCP'},
 {'name': 'Sky cover & clouds', 'id': 'SKY'},
 {'name': 'Sunshine', 'id': 'SUN'},
 {'name': 'Air Temperature', 'id': 'TEMP'},
 {'name': 'Water', 'id': 'WATER'},
 {'name': 'Wind', 'id': 'WIND'},
 {'name': 'Weather Type', 'id': 'WXTYPE'}]
```

Exploring an API to find and collect temperature

- We will use a list comprehension once again to only print the names and IDs; this is still a rather large list, so the output has been abbreviated:

```
>>> response = make_request(
...     'datatypes',
...     payload={'datacategoryid': 'TEMP', 'limit': 100}
... )
>>> response.status_code
200
>>> [(datatype['id'], datatype['name'])
...  for datatype in response.json()['results']]
[('CDSD', 'Cooling Degree Days Season to Date'),
 ...,
 ('TAVG', 'Average Temperature.'),
 ('TMAX', 'Maximum temperature'),
 ('TMIN', 'Minimum temperature'),
 ('TOBS', 'Temperature at the time of observation')]
```

Exploring an API to find and collect temperature

- To determine a value for locationcategoryid, we must use the locationcategories endpoint:

```
>>> response = make_request(
...     'locationcategories', payload={'datasetid': 'GHCND'}
... )
>>> response.status_code
200
```

Exploring an API to find and collect temperature

- Note that we can use pprint from the Python standard library (https://docs.python.org/3/library/pprint.html) to print our JSON payload in an easier-to-read format:

```
>>> import pprint
>>> pprint.pprint(response.json())
{'metadata': {
     'resultset': {'count': 12, 'limit': 25, 'offset': 1}},
 'results': [{'id': 'CITY', 'name': 'City'},
             {'id': 'CLIM_DIV', 'name': 'Climate Division'},
             {'id': 'CLIM_REG', 'name': 'Climate Region'},
             {'id': 'CNTRY', 'name': 'Country'},
             {'id': 'CNTY', 'name': 'County'},
             ...,
             {'id': 'ST', 'name': 'State'},
             {'id': 'US_TERR', 'name': 'US Territory'},
             {'id': 'ZIP', 'name': 'Zip Code'}]}
```

Exploring an API to find and collect temperature

- Each time, we are slicing the data in half, so when we grab the middle entry to test, we are moving closer to the value we seek:

```
>>> def get_item(name, what, endpoint, start=1, end=None):
...     """
...     Grab the JSON payload using binary search.
...
...     Parameters:
...         - name: The item to look for.
...         - what: Dictionary specifying what item `name` is.
...         - endpoint: Where to look for the item.
...         - start: The position to start at. We don't need
...           to touch this, but the function will manipulate
...           this with recursion.
...         - end: The last position of the items. Used to
...           find the midpoint, but like `start` this is not
...           something we need to worry about.
...
...     Returns: Dictionary of the information for the item
...              if found, otherwise an empty dictionary.
...     """
...     # find the midpoint to cut the data in half each time
...     mid = (start + (end or 1)) // 2
```

Exploring an API to find and collect temperature

```
...
...     # lowercase the name so this is not case-sensitive
...     name = name.lower()
...     # define the payload we will send with each request
...     payload = {
...         'datasetid': 'GHCND', 'sortfield': 'name',
...         'offset': mid, # we'll change the offset each time
...         'limit': 1 # we only want one value back
...     }
...
...     # make request adding additional filters from `what`
...     response = make_request(endpoint, {**payload, **what})
...
...     if response.ok:
...         payload = response.json()
...
...         # if ok, grab the end index from the response
...         # metadata the first time through
...         end = end or \
...             payload['metadata']['resultset']['count']
```

Exploring an API to find and collect temperature

```
...
...         # grab the lowercase version of the current name
...         current_name = \
...             payload['results'][0]['name'].lower()
...
...         # if what we are searching for is in the current
...         # name, we have found our item
...         if name in current_name:
...             # return the found item
...             return payload['results'][0]
...         else:
...             if start >= end:
...                 # if start index is greater than or equal
...                 # to end index, we couldn't find it
...                 return {}
...             elif name < current_name:
...                 # name comes before the current name in the
...                 # alphabet => search further to the left
...                 return get_item(name, what, endpoint,
...                                 start, mid - 1)
...             elif name > current_name:
...                 # name comes after the current name in the
...                 # alphabet => search further to the right
...                 return get_item(name, what, endpoint,
...                                 mid + 1, end)
...     else:
...         # response wasn't ok, use code to determine why
...         print('Response not OK, '
...               f'status: {response.status_code}')
```

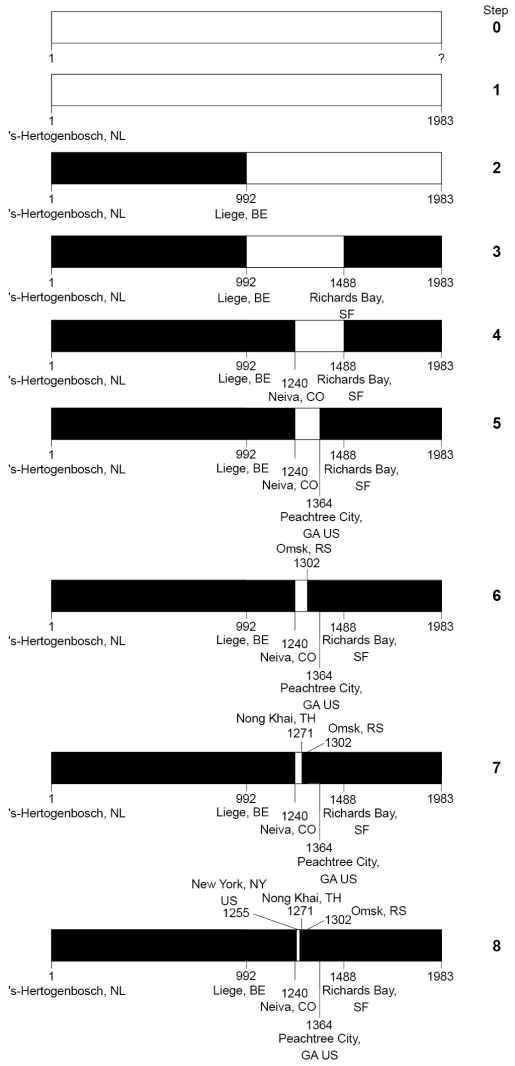
Exploring an API to find and collect temperature

- Now, let's use the binary search implementation to find the ID for New York City, which will be the value we will use for locationid in subsequent queries:

```
>>> nyc = get_item(
...     'New York', {'locationcategoryid': 'CITY'}, 'locations'
... )
>>> nyc
{'mindate': '1869-01-01',
 'maxdate': '2021-01-14',
 'name': 'New York, NY US',
 'datacoverage': 1,
 'id': 'CITY:US360019'}
```

Exploring an API to find and collect temperature

- In the following diagram, we can see how binary search eliminates sections of the list of locations systematically, which is represented by black on the number line (white means it is still possible that the desired value is in that section):



Exploring an API to find and collect temperature

- Using binary search again, we can grab the station ID for the Central Park station:

```
>>> central_park = get_item(
...     'NY City Central Park',
...     {'locationid': nyc['id']}, 'stations'
... )
>>> central_park
{'elevation': 42.7,
 'mindate': '1869-01-01',
 'maxdate': '2020-01-13',
 'latitude': 40.77898,
 'name': 'NY CITY CENTRAL PARK, NY US',
 'datacoverage': 1,
 'id': 'GHCND:USW00094728',
 'elevationUnit': 'METERS',
 'longitude': -73.96925}
```

Exploring an API to find and collect temperature

- Now, let's request NYC's temperature data in Celsius for October 2018, recorded from Central Park.
- For this, we will use the data endpoint and provide all the parameters we picked up throughout our exploration of the API:

```
>>> response = make_request(
...     'data',
...     {'datasetid': 'GHCND',
...      'stationid': central_park['id'],
...      'locationid': nyc['id'],
...      'startdate': '2018-10-01',
...      'enddate': '2018-10-31',
...      'datatypeid': ['TAVG', 'TMAX', 'TMIN'],
...      'units': 'metric',
...      'limit': 1000}
... )
>>> response.status_code
200
```

Exploring an API to find and collect temperature

- Lastly, we will create a DataFrame object; since the results portion of the JSON payload is a list of dictionaries, we can pass it directly to pd.DataFrame():

```
>>> import pandas as pd
>>> df = pd.DataFrame(response.json()['results'])
>>> df.head()
```

Exploring an API to find and collect temperature

- We get back long format data.
- The datatype column is the temperature variable being measured, and the value column contains the measured temperature:

	date	datatype	station	attributes	value
0	2018-10-01T00:00:00	TMAX	GHCND:USW00094728	,,W,2400	24.4
1	2018-10-01T00:00:00	TMIN	GHCND:USW00094728	,,W,2400	17.2
2	2018-10-02T00:00:00	TMAX	GHCND:USW00094728	,,W,2400	25.0
3	2018-10-02T00:00:00	TMIN	GHCND:USW00094728	,,W,2400	18.3
4	2018-10-03T00:00:00	TMAX	GHCND:USW00094728	,,W,2400	23.3

Exploring an API to find and collect temperature

- We asked for TAVG, TMAX, and TMIN, but notice that we didn't get TAVG.
- This is because the Central Park station isn't recording average temperature, despite being listed in the API as offering it—real-world data is dirty:

```
>>> df.datatype.unique()
array(['TMAX', 'TMIN'], dtype=object)
>>> if get_item(
...     'NY City Central Park',
...     {'locationid': nyc['id'], 'datatypeid': 'TAVG'},
...     'stations'
... ):
...     print('Found!')
Found!
```

Cleaning data

- For this section, we will be using the nyc_temperatures.csv file, which contains the maximum daily temperature (TMAX), minimum daily temperature (TMIN), and the average daily temperature (TAVG) from the LaGuardia Airport station in New York City for October 2018:

```
>>> import pandas as pd
>>> df = pd.read_csv('data/nyc_temperatures.csv')
>>> df.head()
```

Cleaning data

- We retrieved long format data from the API; for our analysis, we want wide format data, but we will address that in the Pivoting DataFrames section, later in this lesson:

	date	datatype	station	attributes	value
0	2018-10-01T00:00:00	TAVG	GHCND:USW00014732	H,,S,	21.2
1	2018-10-01T00:00:00	TMAX	GHCND:USW00014732	,,W,2400	25.6
2	2018-10-01T00:00:00	TMIN	GHCND:USW00014732	,,W,2400	18.3
3	2018-10-02T00:00:00	TAVG	GHCND:USW00014732	H,,S,	22.7
4	2018-10-02T00:00:00	TMAX	GHCND:USW00014732	,,W,2400	26.1

Renaming columns

- Since the API endpoint we used could return data of any units and category, it had to call that column value.

- We only pulled temperature data in Celsius, so all our observations have the same units.

- This means that we can rename the value column so that it's clear what data we are working with:

```
>>> df.columns
Index(['date', 'datatype', 'station', 'attributes', 'value'],
      dtype='object')
```

Renaming columns

- The DataFrame class has a rename() method that takes a dictionary mapping the old column name to the new column name.

- In addition to renaming the value column, let's rename the attributes column to flags since the API documentation mentions that that column contains flags for information about data collection:

```
>>> df.rename(
...     columns={'value': 'temp_C', 'attributes': 'flags'},
...     inplace=True
... )
```

Renaming columns

- Most of the time, pandas will return a new DataFrame object; however, since we passed in inplace=True, our original dataframe was updated instead.

- Always be careful with in-place operations, as they might be difficult or impossible to undo.

- Our columns now have their new names:

```
>>> df.columns
Index(['date', 'datatype', 'station', 'flags', 'temp_C'],
      dtype='object')
```

Renaming columns

- We can also do transformations on the column names with rename().
- For instance, we can put all the column names in uppercase:

```
>>> df.rename(str.upper, axis='columns').columns
Index(['DATE', 'DATATYPE', 'STATION', 'FLAGS', 'TEMP_C'],
      dtype='object')
```

Type conversion

- Note that, sometimes, we may have data that we believe should be a certain type, such as a date, but it is stored as a string; this could be for a very valid reason—data could be missing.
- In the case of missing data encoded as text (for example, ? or N/A), pandas will store it as a string when reading it in to allow for this data.
- It will be marked as object when we use the dtypes attribute on our dataframe.

Type conversion

- That being said, let's examine the data types in our temperature data.

- Note that the date column isn't actually being stored as a datetime:

```
>>> df.dtypes
date        object
datatype    object
station     object
flags       object
temp_C     float64
dtype: object
```

Type conversion

- We can use the pd.to_datetime() function to convert it into a datetime:

```
>>> df.loc[:,'date'] = pd.to_datetime(df.date)
>>> df.dtypes
date        datetime64[ns]
datatype            object
station             object
flags               object
temp_C             float64
dtype: object
```

Type conversion

- This is much better.

- Now, we can get useful information when we summarize the date column:

```
>>> df.date.describe(datetime_is_numeric=True)
count                     93
mean     2018-10-16 00:00:00
min      2018-10-01 00:00:00
25%      2018-10-08 00:00:00
50%      2018-10-16 00:00:00
75%      2018-10-24 00:00:00
max      2018-10-31 00:00:00
Name: date, dtype: object
```

Type conversion

- For example, when working with a DatetimeIndex object, if we need to keep track of time zones, we can use the tz_localize() method to associate our datetimes with a time zone:

```
>>> pd.date_range(start='2018-10-25', periods=2, freq='D')\
...     .tz_localize('EST')
DatetimeIndex(['2018-10-25 00:00:00-05:00',
               '2018-10-26 00:00:00-05:00'],
              dtype='datetime64[ns, EST]', freq=None)
```

Type conversion

- This also works with Series and DataFrame objects that have an index of type DatetimeIndex.
- We can read in the CSV file again and, this time, specify that the date column will be our index and that we should parse any dates in the CSV file into datetimes:

```
>>> eastern = pd.read_csv(
...     'data/nyc_temperatures.csv',
...     index_col='date', parse_dates=True
... ).tz_localize('EST')
>>> eastern.head()
```

Type conversion

- Note that we have added the Eastern Standard Time offset (-05:00 from UTC) to the datetimes in the index:

date	datatype	station	attributes	value
2018-10-01 00:00:00-05:00	TAVG	GHCND:USW00014732	H,,S,	21.2
2018-10-01 00:00:00-05:00	TMAX	GHCND:USW00014732	,,W,2400	25.6
2018-10-01 00:00:00-05:00	TMIN	GHCND:USW00014732	,,W,2400	18.3
2018-10-02 00:00:00-05:00	TAVG	GHCND:USW00014732	H,,S,	22.7
2018-10-02 00:00:00-05:00	TMAX	GHCND:USW00014732	,,W,2400	26.1

Type conversion

- We can use the tz_convert() method to change the time zone into a different one.
- Let's change our data into UTC:

```
>>> eastern.tz_convert('UTC').head()
```

Type conversion

- Now, the offset is UTC (+00:00), but note that the time portion of the date is now 5 AM; this conversion took into account the -05:00 offset:

date	datatype	station	attributes	value
2018-10-01 05:00:00+00:00	TAVG	GHCND:USW00014732	H,,S,	21.2
2018-10-01 05:00:00+00:00	TMAX	GHCND:USW00014732	,,W,2400	25.6
2018-10-01 05:00:00+00:00	TMIN	GHCND:USW00014732	,,W,2400	18.3
2018-10-02 05:00:00+00:00	TAVG	GHCND:USW00014732	H,,S,	22.7
2018-10-02 05:00:00+00:00	TMAX	GHCND:USW00014732	,,W,2400	26.1

Type conversion

- This is because the underlying data for a PeriodIndex object is stored as a PeriodArray object:

```
>>> eastern.tz_localize(None).to_period('M').index
PeriodIndex(['2018-10', '2018-10', ..., '2018-10', '2018-10'],
            dtype='period[M]', name='date', freq='M')
```

Type conversion

- We can use the to_timestamp() method to convert our PeriodIndex object into a DatetimeIndex object; however, the datetimes all start at the first of the month now:

```
>>> eastern.tz_localize(None)\
...     .to_period('M').to_timestamp().index
DatetimeIndex(['2018-10-01', '2018-10-01', '2018-10-01', ...,
               '2018-10-01', '2018-10-01', '2018-10-01'],
              dtype='datetime64[ns]', name='date', freq=None)
```

Type conversion

- Here, we will create a new one.
- Note that our original conversion of the dates modified the column, so, in order to illustrate that we can use assign(), we need to read our data in once more:

```
>>> df = pd.read_csv('data/nyc_temperatures.csv').rename(
...     columns={'value': 'temp_C', 'attributes': 'flags'}
... )
>>> new_df = df.assign(
...     date=pd.to_datetime(df.date),
...     temp_F=(df.temp_C * 9/5) + 32
... )
>>> new_df.dtypes
date        datetime64[ns]
datatype            object
station             object
flags               object
temp_C             float64
temp_F             float64
dtype: object
>>> new_df.head()
```

Type conversion

- We now have datetimes in the date column and a new column, temp_F:

	date	datatype	station	flags	temp_C	temp_F
0	2018-10-01	TAVG	GHCND:USW00014732	H,,S,	21.2	70.16
1	2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	78.08
2	2018-10-01	TMIN	GHCND:USW00014732	,,W,2400	18.3	64.94
3	2018-10-02	TAVG	GHCND:USW00014732	H,,S,	22.7	72.86
4	2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	78.98

Type conversion

- It is very common (and useful) to use lambda functions with assign():

```
>>> df = df.assign(
...     date=lambda x: pd.to_datetime(x.date),
...     temp_C_whole=lambda x: x.temp_C.astype('int'),
...     temp_F=lambda x: (x.temp_C * 9/5) + 32,
...     temp_F_whole=lambda x: x.temp_F.astype('int')
... )
>>> df.head()
```

Type conversion

- If all the numbers are whole, they will be converted into integers (obviously, we will still get errors if the data isn't numeric at all):

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
0	2018-10-01	TAVG	GHCND:USW00014732	H,,S,	21.2	21	70.16	70
1	2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
2	2018-10-01	TMIN	GHCND:USW00014732	,,W,2400	18.3	18	64.94	64
3	2018-10-02	TAVG	GHCND:USW00014732	H,,S,	22.7	22	72.86	72
4	2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78

Type conversion

- We can use the astype() method to cast these into categories and look at the summary statistics:

```
>>> df_with_categories = df.assign(
...     station=df.station.astype('category'),
...     datatype=df.datatype.astype('category')
... )
>>> df_with_categories.dtypes
date             datetime64[ns]
datatype               category
station                category
flags                    object
temp_C                  float64
temp_C_whole              int64
temp_F                  float64
temp_F_whole              int64
dtype: object
>>> df_with_categories.describe(include='category')
```

Type conversion

- The summary statistics for categories are just like those for strings.
- We can see the number of non-null entries (count), the number of unique values (unique), the mode (top), and the number of occurrences of the mode (freq):

	datatype	station
count	93	93
unique	3	1
top	TAVG	GHCND:USW00014732
freq	31	93

Type conversion

- The categories we just made don't have any order to them, but pandas does support this:

```
>>> pd.Categorical(
...     ['med', 'med', 'low', 'high'],
...     categories=['low', 'med', 'high'],
...     ordered=True
... )
['med', 'med', 'low', 'high']
Categories (3, object): ['low' < 'med' < 'high']
```

Reordering, reindexing, and sorting data

- We will often find the need to sort our data by the values of one or many columns.
- Say we wanted to find the days that reached the highest temperatures in New York City during October 2018; we could sort our values by the temp_C (or temp_F) column in descending order and use head() to select the number of days we wanted to see.
- To accomplish this, we can use the sort_values() method.

- Let's look at the top 10 days:

```
>>> df[df.datatype == 'TMAX']\
...     .sort_values(by='temp_C', ascending=False).head(10)
```

Reordering, reindexing, and sorting data

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
19	2018-10-07	TMAX	GHCND:USW00014732	,,W,2400	27.8	27	82.04	82
28	2018-10-10	TMAX	GHCND:USW00014732	,,W,2400	27.8	27	82.04	82
31	2018-10-11	TMAX	GHCND:USW00014732	,,W,2400	26.7	26	80.06	80
10	2018-10-04	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78
4	2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78
1	2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
25	2018-10-09	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
7	2018-10-03	TMAX	GHCND:USW00014732	,,W,2400	25.0	25	77.00	77
13	2018-10-05	TMAX	GHCND:USW00014732	,,W,2400	22.8	22	73.04	73
22	2018-10-08	TMAX	GHCND:USW00014732	,,W,2400	22.8	22	73.04	73

Reordering, reindexing, and sorting data

- The sort_values() method can be used with a list of column names to break ties.
- The order in which the columns are provided will determine the sort order, with each subsequent column being used to break ties.

- As an example, let's make sure the dates are sorted in ascending order when breaking ties:

```
>>> df[df.datatype == 'TMAX'].sort_values(
...     by=['temp_C', 'date'], ascending=[False, True]
... ).head(10)
```

Reordering, reindexing, and sorting data

- Notice how October 2nd is now above October 4th, despite both having the same temperature reading:

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
19	2018-10-07	TMAX	GHCND:USW00014732	,,W,2400	27.8	27	82.04	82
28	2018-10-10	TMAX	GHCND:USW00014732	,,W,2400	27.8	27	82.04	82
31	2018-10-11	TMAX	GHCND:USW00014732	,,W,2400	26.7	26	80.06	80
4	2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78
10	2018-10-04	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78
1	2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
25	2018-10-09	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
7	2018-10-03	TMAX	GHCND:USW00014732	,,W,2400	25.0	25	77.00	77
13	2018-10-05	TMAX	GHCND:USW00014732	,,W,2400	22.8	22	73.04	73
22	2018-10-08	TMAX	GHCND:USW00014732	,,W,2400	22.8	22	73.04	73

Reordering, reindexing, and sorting data

- Let's grab the top 10 days by average temperature this time:

```
>>> df[df.datatype == 'TAVG'].nlargest(n=10, columns='temp_C')
```

Reordering, reindexing, and sorting data

- We get the warmest days (on average) in October:

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
27	2018-10-10	TAVG	GHCND:USW00014732	H,,S,	23.8	23	74.84	74
30	2018-10-11	TAVG	GHCND:USW00014732	H,,S,	23.4	23	74.12	74
18	2018-10-07	TAVG	GHCND:USW00014732	H,,S,	22.8	22	73.04	73
3	2018-10-02	TAVG	GHCND:USW00014732	H,,S,	22.7	22	72.86	72
6	2018-10-03	TAVG	GHCND:USW00014732	H,,S,	21.8	21	71.24	71
24	2018-10-09	TAVG	GHCND:USW00014732	H,,S,	21.8	21	71.24	71
9	2018-10-04	TAVG	GHCND:USW00014732	H,,S,	21.3	21	70.34	70
0	2018-10-01	TAVG	GHCND:USW00014732	H,,S,	21.2	21	70.16	70
21	2018-10-08	TAVG	GHCND:USW00014732	H,,S,	20.9	20	69.62	69
12	2018-10-05	TAVG	GHCND:USW00014732	H,,S,	20.3	20	68.54	68

Reordering, reindexing, and sorting data

- For instance, the sample() method will give us randomly selected rows, which will lead to a jumbled index, so we can use sort_index() to order them afterward:

```
>>> df.sample(5, random_state=0).index
Int64Index([2, 30, 55, 16, 13], dtype='int64')
>>> df.sample(5, random_state=0).sort_index().index
Int64Index([2, 13, 16, 30, 55], dtype='int64')
```

Reordering, reindexing, and sorting data

- Let's use this knowledge to sort the columns of our dataframe alphabetically:

```
>>> df.sort_index(axis=1).head()
```

Reordering, reindexing, and sorting data

- Having our columns in alphabetical order can come in handy when using loc[] because we can specify a range of columns with similar names; for example, we could now use df.loc[:,'station':'temp_F_whole'] to easily grab all of our temperature columns, along with the station information:

	datatype	date	flags	station	temp_C	temp_C_whole	temp_F	temp_F_whole
0	TAVG	2018-10-01	H,,S,	GHCND:USW00014732	21.2	21	70.16	70
1	TMAX	2018-10-01	,,W,2400	GHCND:USW00014732	25.6	25	78.08	78
2	TMIN	2018-10-01	,,W,2400	GHCND:USW00014732	18.3	18	64.94	64
3	TAVG	2018-10-02	H,,S,	GHCND:USW00014732	22.7	22	72.86	72
4	TMAX	2018-10-02	,,W,2400	GHCND:USW00014732	26.1	26	78.98	78

Reordering, reindexing, and sorting data

- We must sort the index to see that they are the same:

```
>>> df.equals(df.sort_values(by='temp_C'))
False
>>> df.equals(df.sort_values(by='temp_C').sort_index())
True
```

Reordering, reindexing, and sorting data

- Sometimes, we don't care too much about the numeric index, but we would like to use one (or more) of the other columns as the index instead.
- In this case, we can use the set_index() method.

- Let's set the date column as our index:

```
>>> df.set_index('date', inplace=True)
>>> df.head()
```

Reordering, reindexing, and sorting data

- Notice that the date column has moved to the far left where the index goes, and we no longer have the numeric index:

date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
2018-10-01	TAVG	GHCND:USW00014732	H,,S,	21.2	21	70.16	70
2018-10-01	TMAX	GHCND:USW00014732	,,W,2400	25.6	25	78.08	78
2018-10-01	TMIN	GHCND:USW00014732	,,W,2400	18.3	18	64.94	64
2018-10-02	TAVG	GHCND:USW00014732	H,,S,	22.7	22	72.86	72
2018-10-02	TMAX	GHCND:USW00014732	,,W,2400	26.1	26	78.98	78

Reordering, reindexing, and sorting data

- These can also be combined to build ranges.
- Note that loc[] is optional when using ranges:

```
>>> df['2018-10-11':'2018-10-12']
```

Reordering, reindexing, and sorting data

- This gives us the data from October 11, 2018 through October 12, 2018 (inclusive of both endpoints):

date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
2018-10-11	TAVG	GHCND:USW00014732	H,,S,	23.4	23	74.12	74
2018-10-11	TMAX	GHCND:USW00014732	,,W,2400	26.7	26	80.06	80
2018-10-11	TMIN	GHCND:USW00014732	,,W,2400	21.7	21	71.06	71
2018-10-12	TAVG	GHCND:USW00014732	H,,S,	18.3	18	64.94	64
2018-10-12	TMAX	GHCND:USW00014732	,,W,2400	22.2	22	71.96	71
2018-10-12	TMIN	GHCND:USW00014732	,,W,2400	12.2	12	53.96	53

Reordering, reindexing, and sorting data

- We can use the reset_index() method to restore the date column:

```
>>> df['2018-10-11':'2018-10-12'].reset_index()
```

Reordering, reindexing, and sorting data

- Our index now starts at 0, and the dates are now in a column called date.
- This is especially useful if we have data that we don't want to lose in the index, such as the date, but need to perform an operation as if it weren't in the index:

	date	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
0	2018-10-11	TAVG	GHCND:USW00014732	H,,S,	23.4	23	74.12	74
1	2018-10-11	TMAX	GHCND:USW00014732	,,W,2400	26.7	26	80.06	80
2	2018-10-11	TMIN	GHCND:USW00014732	,,W,2400	21.7	21	71.06	71
3	2018-10-12	TAVG	GHCND:USW00014732	H,,S,	18.3	18	64.94	64
4	2018-10-12	TMAX	GHCND:USW00014732	,,W,2400	22.2	22	71.96	71
5	2018-10-12	TMIN	GHCND:USW00014732	,,W,2400	12.2	12	53.96	53

Reordering, reindexing, and sorting data

- Let's read it in, setting the date column as the index and parsing the dates:

```
>>> sp = pd.read_csv(
...     'data/sp500.csv', index_col='date', parse_dates=True
... ).drop(columns=['adj_close']) # not using this column
```

Reordering, reindexing, and sorting data

- In this case, it's day_name():

```
>>> sp.head(10)\
...     .assign(day_of_week=lambda x: x.index.day_name())
```

Reordering, reindexing, and sorting data

- Since the stock market is closed on the weekend (and holidays), we only have data for weekdays:

date	high	low	open	close	volume	day_of_week
2017-01-03	2263.879883	2245.129883	2251.570068	2257.830078	3770530000	Tuesday
2017-01-04	2272.820068	2261.600098	2261.600098	2270.750000	3764890000	Wednesday
2017-01-05	2271.500000	2260.449951	2268.179932	2269.000000	3761820000	Thursday
2017-01-06	2282.100098	2264.060059	2271.139893	2276.979980	3339890000	Friday
2017-01-09	2275.489990	2268.899902	2273.590088	2268.899902	3217610000	Monday
2017-01-10	2279.270020	2265.270020	2269.719971	2268.899902	3638790000	Tuesday
2017-01-11	2275.320068	2260.830078	2268.600098	2275.320068	3620410000	Wednesday
2017-01-12	2271.780029	2254.250000	2271.139893	2270.439941	3462130000	Thursday
2017-01-13	2278.679932	2271.510010	2272.739990	2274.639893	3081270000	Friday
2017-01-17	2272.080078	2262.810059	2269.139893	2267.889893	3584990000	Tuesday

Reordering, reindexing, and sorting data

- The bitcoin data also contains OHLC data and volume traded, but it comes with a column called market_cap that we don't need, so we have to drop that first:

```
>>> bitcoin = pd.read_csv(
...     'data/bitcoin.csv', index_col='date', parse_dates=True
... ).drop(columns=['market_cap'])
```

Reordering, reindexing, and sorting data

- For example, each day's closing price will be the sum of the closing price of the S&P 500 and the closing price of bitcoin:

```
# every day's closing price = S&P 500 close + Bitcoin close
# (same for other metrics)
>>> portfolio = pd.concat([sp, bitcoin], sort=False)\
...     .groupby(level='date').sum()
>>> portfolio.head(10).assign(
...     day_of_week=lambda x: x.index.day_name()
... )
```

Reordering, reindexing, and sorting data

- Now, if we examine our portfolio, we will see that we have values for every day of the week; so far, so good:

date	high	low	open	close	volume	day_of_week
2017-01-01	1003.080000	958.700000	963.660000	998.330000	147775008	Sunday
2017-01-02	1031.390000	996.700000	998.620000	1021.750000	222184992	Monday
2017-01-03	3307.959883	3266.729883	3273.170068	3301.670078	3955698000	Tuesday
2017-01-04	3432.240068	3306.000098	3306.000098	3425.480000	4109835984	Wednesday
2017-01-05	3462.600000	3170.869951	3424.909932	3282.380000	4272019008	Thursday
2017-01-06	3328.910098	3148.000059	3285.379893	3179.179980	3691766000	Friday
2017-01-07	908.590000	823.560000	903.490000	908.590000	279550016	Saturday
2017-01-08	942.720000	887.250000	908.170000	911.200000	158715008	Sunday
2017-01-09	3189.179990	3148.709902	3186.830088	3171.729902	3359486992	Monday
2017-01-10	3194.140020	3166.330020	3172.159971	3176.579902	3754598000	Tuesday

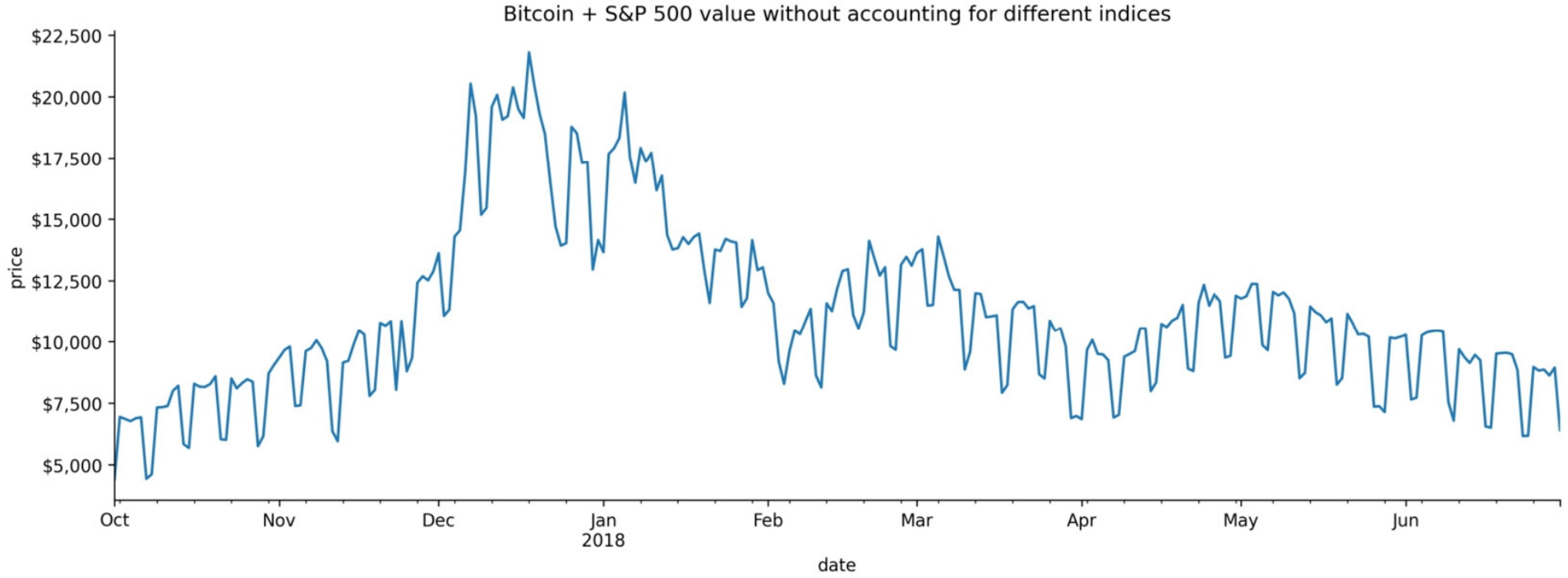
Reordering, reindexing, and sorting data

- However, there is a problem with this approach, which is much easier to see with a visualization. Plotting will be covered in depth in Visualizing Data with Pandas and Matplotlib, and Plotting with Seaborn and Customization Techniques, so don't worry about the details for now:

```
>>> import matplotlib.pyplot as plt # module for plotting
>>> from matplotlib.ticker import StrMethodFormatter
# plot the closing price from Q4 2017 through Q2 2018
>>> ax = portfolio['2017-Q4':'2018-Q2'].plot(
...     y='close', figsize=(15, 5), legend=False,
...     title='Bitcoin + S&P 500 value without accounting '
...           'for different indices'
... )
# formatting
>>> ax.set_ylabel('price')
>>> ax.yaxis\
...     .set_major_formatter(StrMethodFormatter('${x:,.0f}'))
>>> for spine in ['top', 'right']:
...     ax.spines[spine].set_visible(False)
# show the plot
>>> plt.show()
```

Reordering, reindexing, and sorting data

- Notice how there is a cyclical pattern here? It is dropping every day the market is closed because the aggregation only had bitcoin data to sum for those days:



Reordering, reindexing, and sorting data

- Forward-filling seems to be the best option, but since we aren't sure, we will see how this works on a few rows of the data first:

```
>>> sp.reindex(bitcoin.index, method='ffill').head(10)\
...     .assign(day_of_week=lambda x: x.index.day_name())
```

Reordering, reindexing, and sorting data

- Notice any issues with this? Well, the volume traded (volume) column makes it seem like the days we used forward-filling for are actually days when the market is open:

date	high	low	open	close	volume	day_of_week
2017-01-01	NaN	NaN	NaN	NaN	NaN	Sunday
2017-01-02	NaN	NaN	NaN	NaN	NaN	Monday
2017-01-03	2263.879883	2245.129883	2251.570068	2257.830078	3.770530e+09	Tuesday
2017-01-04	2272.820068	2261.600098	2261.600098	2270.750000	3.764890e+09	Wednesday
2017-01-05	2271.500000	2260.449951	2268.179932	2269.000000	3.761820e+09	Thursday
2017-01-06	2282.100098	2264.060059	2271.139893	2276.979980	3.339890e+09	Friday
2017-01-07	2282.100098	2264.060059	2271.139893	2276.979980	3.339890e+09	Saturday
2017-01-08	2282.100098	2264.060059	2271.139893	2276.979980	3.339890e+09	Sunday
2017-01-09	2275.489990	2268.899902	2273.590088	2268.899902	3.217610e+09	Monday
2017-01-10	2279.270020	2265.270020	2269.719971	2268.899902	3.638790e+09	Tuesday

Reordering, reindexing, and sorting data

- Lastly, we can use the np.where() function for the remaining columns, which allows us to build a vectorized if...else.

- It takes the following form:

np.where(boolean condition, value if True, value if False)

Reordering, reindexing, and sorting data

- Since these come after the close column gets worked on, we will have the forward-filled value for close to use for the other columns where necessary:

```
>>> import numpy as np
>>> sp_reindexed = sp.reindex(bitcoin.index).assign(
...     # volume is 0 when the market is closed
...     volume=lambda x: x.volume.fillna(0),
...     # carry this forward
...     close=lambda x: x.close.fillna(method='ffill'),
...     # take the closing price if these aren't available
...     open=lambda x: \
...         np.where(x.open.isnull(), x.close, x.open),
...     high=lambda x: \
...         np.where(x.high.isnull(), x.close, x.high),
...     low=lambda x: np.where(x.low.isnull(), x.close, x.low)
... )
>>> sp_reindexed.head(10).assign(
...     day_of_week=lambda x: x.index.day_name()
... )
```

Reordering, reindexing, and sorting data

- The OHLC prices are all equal to the closing price on Friday, the 6th:

date	high	low	open	close	volume	day_of_week
2017-01-01	NaN	NaN	NaN	NaN	0.000000e+00	Sunday
2017-01-02	NaN	NaN	NaN	NaN	0.000000e+00	Monday
2017-01-03	2263.879883	2245.129883	2251.570068	2257.830078	3.770530e+09	Tuesday
2017-01-04	2272.820068	2261.600098	2261.600098	2270.750000	3.764890e+09	Wednesday
2017-01-05	2271.500000	2260.449951	2268.179932	2269.000000	3.761820e+09	Thursday
2017-01-06	2282.100098	2264.060059	2271.139893	2276.979980	3.339890e+09	Friday
2017-01-07	2276.979980	2276.979980	2276.979980	2276.979980	0.000000e+00	Saturday
2017-01-08	2276.979980	2276.979980	2276.979980	2276.979980	0.000000e+00	Sunday
2017-01-09	2275.489990	2268.899902	2273.590088	2268.899902	3.217610e+09	Monday
2017-01-10	2279.270020	2265.270020	2269.719971	2268.899902	3.638790e+09	Tuesday

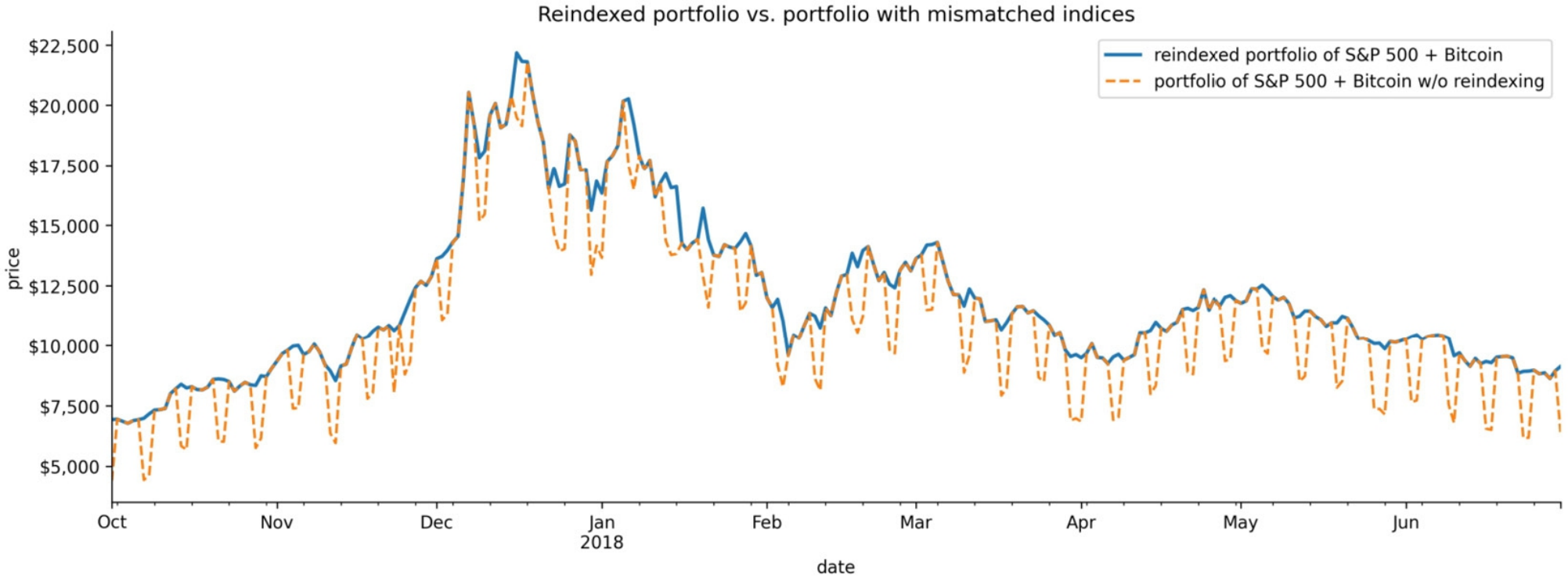
Reordering, reindexing, and sorting data

- Now, let's recreate the portfolio with the reindexed S&P 500 data and use a visualization to compare it with the previous attempt (again, don't worry about the plotting code, which will be covered in Visualizing Data with Pandas and Matplotlib, and Plotting with Seaborn and Customization Techniques):

```
# every day's closing price = S&P 500 close adjusted for
# market closure + Bitcoin close (same for other metrics)
>>> fixed_portfolio = sp_reindexed + bitcoin
# plot the reindexed portfolio's close (Q4 2017 - Q2 2018)
>>> ax = fixed_portfolio['2017-Q4':'2018-Q2'].plot(
...     y='close', figsize=(15, 5), linewidth=2,
...     label='reindexed portfolio of S&P 500 + Bitcoin',
...     title='Reindexed portfolio vs.'
...           'portfolio with mismatched indices'
... )
# add line for original portfolio for comparison
>>> portfolio['2017-Q4':'2018-Q2'].plot(
...     y='close', ax=ax, linestyle='--',
...     label='portfolio of S&P 500 + Bitcoin w/o reindexing'
... )
# formatting
>>> ax.set_ylabel('price')
>>> ax.yaxis\
...     .set_major_formatter(StrMethodFormatter('${x:,.0f}'))
>>> for spine in ['top', 'right']:
...     ax.spines[spine].set_visible(False)
# show the plot
>>> plt.show()
```

Reordering, reindexing, and sorting data

- Keep this strategy in mind for the exercises in Financial Analysis – Bitcoin and the Stock Market:



Reshaping data

- However, this isn't always as black and white as going from long format to wide format or vice versa.
- Consider the following data from the Exercises section:

	ticker	date	high	low	open	close	volume
0	AAPL	2018-01-02	43.075001	42.314999	42.540001	43.064999	102223600
0	AMZN	2018-01-02	1190.000000	1170.510010	1172.000000	1189.010010	2694500
0	FB	2018-01-02	181.580002	177.550003	177.679993	181.419998	18151900
0	GOOG	2018-01-02	1066.939941	1045.229980	1048.339966	1065.000000	1237600
0	NFLX	2018-01-02	201.649994	195.419998	196.100006	201.070007	10966900

Reshaping data

- We will begin by importing pandas and reading in the long_data.csv file, adding the temperature in Fahrenheit column (temp_F), and performing some of the data cleaning we just learned about:

```
>>> import pandas as pd
>>> long_df = pd.read_csv(
...     'data/long_data.csv',
...     usecols=['date', 'datatype', 'value']
... ).rename(columns={'value': 'temp_C'}).assign(
...     date=lambda x: pd.to_datetime(x.date),
...     temp_F=lambda x: (x.temp_C * 9/5) + 32
... )
```

Reshaping data

- Our long format data looks like this:

	datatype	date	temp_C	temp_F
0	TMAX	2018-10-01	21.1	69.98
1	TMIN	2018-10-01	8.9	48.02
2	TOBS	2018-10-01	13.9	57.02
3	TMAX	2018-10-02	23.9	75.02
4	TMIN	2018-10-02	13.9	57.02

Transposing DataFrames

- While we will be pretty much only working with wide or long formats, pandas provides ways to restructure our data as we see fit, including taking the transpose (flipping the rows with the columns), which we may find useful to make better use of our display area when we're printing parts of our dataframe:

```
>>> long_df.set_index('date').head(6).T
```

Transposing DataFrames

- Notice that the index is now in the columns, and that the column names are in the index:

date	2018-10-01	2018-10-01	2018-10-01	2018-10-02	2018-10-02	2018-10-02
datatype	TMAX	TMIN	TOBS	TMAX	TMIN	TOBS
temp_C	21.10	8.90	13.90	23.90	13.90	17.20
temp_F	69.98	48.02	57.02	75.02	57.02	62.96

Pivoting DataFrames

- Let's pivot into wide format, where we have a column for each of the temperature measurements in Celsius and use the dates as the index:

```
>>> pivoted_df = long_df.pivot(
...     index='date', columns='datatype', values='temp_C'
... )
>>> pivoted_df.head()
```

Pivoting DataFrames

- Finally, the values for each combination of date and datatype are the corresponding temperatures in Celsius since we passed in values='temp_C':

datatype	TMAX	TMIN	TOBS
date			
2018-10-01	21.1	8.9	13.9
2018-10-02	23.9	13.9	17.2
2018-10-03	25.0	15.6	16.1
2018-10-04	22.8	11.7	11.7
2018-10-05	23.3	11.7	18.9

Pivoting DataFrames

- As we discussed at the beginning of this lesson, with the data in wide format, we can easily get meaningful summary statistics with the describe() method:

```
>>> pivoted_df.describe()
```

Pivoting DataFrames

- We can see that we have 31 observations for all three temperature measurements and that this month has a wide range of temperatures (highest daily maximum of 26.7°C and lowest daily minimum of -1.1°C):

datatype	TMAX	TMIN	TOBS
count	31.000000	31.000000	31.000000
mean	16.829032	7.561290	10.022581
std	5.714962	6.513252	6.596550
min	7.800000	-1.100000	-1.100000
25%	12.750000	2.500000	5.550000
50%	16.100000	6.700000	8.300000
75%	21.950000	13.600000	16.100000
max	26.700000	17.800000	21.700000

Pivoting DataFrames

- We lost the temperature in Fahrenheit, though.
- If we want to keep it, we can provide multiple columns to values:

```
>>> pivoted_df = long_df.pivot(
...     index='date', columns='datatype',
...     values=['temp_C', 'temp_F']
... )
>>> pivoted_df.head()
```

Pivoting DataFrames

- However, we now get an extra level above the column names. This is called a hierarchical index:

	temp_C			temp_F		
datatype	TMAX	TMIN	TOBS	TMAX	TMIN	TOBS
date						
2018-10-01	21.1	8.9	13.9	69.98	48.02	57.02
2018-10-02	23.9	13.9	17.2	75.02	57.02	62.96
2018-10-03	25.0	15.6	16.1	77.00	60.08	60.98
2018-10-04	22.8	11.7	11.7	73.04	53.06	53.06
2018-10-05	23.3	11.7	18.9	73.94	53.06	66.02

Pivoting DataFrames

- With this hierarchical index, if we want to select TMIN in Fahrenheit, we will first need to select temp_F and then TMIN:

```
>>> pivoted_df['temp_F']['TMIN'].head()
date
2018-10-01    48.02
2018-10-02    57.02
2018-10-03    60.08
2018-10-04    53.06
2018-10-05    53.06
Name: TMIN, dtype: float64
```

Pivoting DataFrames

- This gives us an index of type MultiIndex, where the outermost level corresponds to the first element in the list provided to set_index():

```
>>> multi_index_df = long_df.set_index(['date', 'datatype'])
>>> multi_index_df.head().index
MultiIndex([('2018-10-01', 'TMAX'),
            ('2018-10-01', 'TMIN'),
            ('2018-10-01', 'TOBS'),
            ('2018-10-02', 'TMAX'),
            ('2018-10-02', 'TMIN')],
           names=['date', 'datatype'])
>>> multi_index_df.head()
```

Pivoting DataFrames

- Notice that we now have two levels in the index—date is the outermost level and datatype is the innermost:

		temp_C	temp_F
date	**datatype**		
2018-10-01	**TMAX**	21.1	69.98
	TMIN	8.9	48.02
	TOBS	13.9	57.02
2018-10-02	**TMAX**	23.9	75.02
	TMIN	13.9	57.02

Pivoting DataFrames

- To unstack a different level, simply pass in the index of the level to unstack, where 0 is the leftmost and -1 is the rightmost, or the name of the level (if it has one).

- Here, we will use the default:

```
>>> unstacked_df = multi_index_df.unstack()
>>> unstacked_df.head()
```

Pivoting DataFrames

- In Aggregating Pandas DataFrames, we will discuss a way to squash this back into a single level of columns:

	temp_C			temp_F		
datatype	TMAX	TMIN	TOBS	TMAX	TMIN	TOBS
date						
2018-10-01	21.1	8.9	13.9	69.98	48.02	57.02
2018-10-02	23.9	13.9	17.2	75.02	57.02	62.96
2018-10-03	25.0	15.6	16.1	77.00	60.08	60.98
2018-10-04	22.8	11.7	11.7	73.04	53.06	53.06
2018-10-05	23.3	11.7	18.9	73.94	53.06	66.02

Pivoting DataFrames

- We could append this to long_df and set our index to the date and datatype columns, as we did previously:

```
>>> extra_data = long_df.append([{
...     'datatype': 'TAVG',
...     'date': '2018-10-01',
...     'temp_C': 10,
...     'temp_F': 50
... }]).set_index(['date', 'datatype']).sort_index()
>>> extra_data['2018-10-01':'2018-10-02']
```

Pivoting DataFrames

- We now have four temperature measurements for October 1, 2018, but only three for the remaining days:

		temp_C	temp_F
date	**datatype**		
2018-10-01	**TAVG**	10.0	50.00
	TMAX	21.1	69.98
	TMIN	8.9	48.02
	TOBS	13.9	57.02
2018-10-02	**TMAX**	23.9	75.02
	TMIN	13.9	57.02
	TOBS	17.2	62.96

Pivoting DataFrames

- Using unstack(), as we did previously, will result in NaN values for most of the TAVG data:

```
>>> extra_data.unstack().head()
```

Pivoting DataFrames

- Take a look at the TAVG columns after we unstack:

	temp_C				temp_F			
datatype	**TAVG**	**TMAX**	**TMIN**	**TOBS**	**TAVG**	**TMAX**	**TMIN**	**TOBS**
date								
2018-10-01	10.0	21.1	8.9	13.9	50.0	69.98	48.02	57.02
2018-10-02	NaN	23.9	13.9	17.2	NaN	75.02	57.02	62.96
2018-10-03	NaN	25.0	15.6	16.1	NaN	77.00	60.08	60.98
2018-10-04	NaN	22.8	11.7	11.7	NaN	73.04	53.06	53.06
2018-10-05	NaN	23.3	11.7	18.9	NaN	73.94	53.06	66.02

Pivoting DataFrames

- To address this, we can pass in an appropriate fill_value. However, we are restricted to passing in a value for this, not a strategy (as we saw when we discussed reindexing), so while there is no good value for this case, we can use -40 to illustrate how this works:

```
>>> extra_data.unstack(fill_value=-40).head()
```

Pivoting DataFrames

- Note that, in practice, it is better to be explicit about the missing data if we are sharing this with others and leave the NaN values:

	temp_C				temp_F			
datatype	TAVG	TMAX	TMIN	TOBS	TAVG	TMAX	TMIN	TOBS
date								
2018-10-01	10.0	21.1	8.9	13.9	50.0	69.98	48.02	57.02
2018-10-02	-40.0	23.9	13.9	17.2	-40.0	75.02	57.02	62.96
2018-10-03	-40.0	25.0	15.6	16.1	-40.0	77.00	60.08	60.98
2018-10-04	-40.0	22.8	11.7	11.7	-40.0	73.04	53.06	53.06
2018-10-05	-40.0	23.3	11.7	18.9	-40.0	73.94	53.06	66.02

Melting DataFrames

- To go from wide format to long format, we need to melt the data.

- Melting undoes a pivot.

- For this example, we will read in the data from the wide_data.csv file:

```
>>> wide_df = pd.read_csv('data/wide_data.csv')
>>> wide_df.head()
```

Melting DataFrames

- Our wide data contains a column for the date and a column for each temperature measurement we have been working with:

	date	TMAX	TMIN	TOBS
0	2018-10-01	21.1	8.9	13.9
1	2018-10-02	23.9	13.9	17.2
2	2018-10-03	25.0	15.6	16.1
3	2018-10-04	22.8	11.7	11.7
4	2018-10-05	23.3	11.7	18.9

Melting DataFrames

- Now, let's use the melt() method to turn the wide format data into long format:

```
>>> melted_df = wide_df.melt(
...     id_vars='date', value_vars=['TMAX', 'TMIN', 'TOBS'],
...     value_name='temp_C', var_name='measurement'
... )
>>> melted_df.head()
```

Melting DataFrames

- We now have just three columns; the date, the temperature reading in Celsius (temp_C), and a column indicating which temperature measurement is in that row's temp_C cell (measurement):

	date	measurement	temp_C
0	2018-10-01	TMAX	21.1
1	2018-10-02	TMAX	23.9
2	2018-10-03	TMAX	25.0
3	2018-10-04	TMAX	22.8
4	2018-10-05	TMAX	23.3

Melting DataFrames

- We can do the following to get a similar output to the melt() method:

```
>>> wide_df.set_index('date', inplace=True)
>>> stacked_series = wide_df.stack() # put datatypes in index
>>> stacked_series.head()
date
2018-10-01  TMAX    21.1
            TMIN     8.9
            TOBS    13.9
2018-10-02  TMAX    23.9
            TMIN    13.9
dtype: float64
```

Melting DataFrames

- Notice that the result came back as a Series object, so we will need to create the DataFrame object once more.

- We can use the to_frame() method and pass in a name to use for the column once it is a dataframe:

```
>>> stacked_df = stacked_series.to_frame('values')
>>> stacked_df.head()
```

Melting DataFrames

- Now, we have a dataframe with a multi-level index, containing date and datatype, with values as the only column.
- Notice, however, that only the date portion of our index has a name:

		values
date		
2018-10-01	TMAX	21.1
	TMIN	8.9
	TOBS	13.9
2018-10-02	TMAX	23.9
	TMIN	13.9

Melting DataFrames

- However, this level was never named, so it shows up as None, but we know that the level should be called datatype:

```
>>> stacked_df.head().index
MultiIndex([('2018-10-01', 'TMAX'),
            ('2018-10-01', 'TMIN'),
            ('2018-10-01', 'TOBS'),
            ('2018-10-02', 'TMAX'),
            ('2018-10-02', 'TMIN')],
           names=['date', None])
```

Melting DataFrames

- We can use the set_names() method to address this:

```
>>> stacked_df.index\
...     .set_names(['date', 'datatype'], inplace=True)
>>> stacked_df.index.names
FrozenList(['date', 'datatype'])
```

Handling duplicate, missing, or invalid data

- We will be working in the 5-handling_data_issues.ipynb notebook and using the dirty_data.csv file.

- Let's start by importing pandas and reading in the data:

```
>>> import pandas as pd
>>> df = pd.read_csv('data/dirty_data.csv')
```

Handling duplicate, missing, or invalid data

It contains the following fields:

- PRCP: Precipitation in millimeters
- SNOW: Snowfall in millimeters
- SNWD: Snow depth in millimeters
- TMAX: Maximum daily temperature in Celsius
- TMIN: Minimum daily temperature in Celsius
- TOBS: Temperature at the time of observation in Celsius
- WESF: Water equivalent of snow in millimeters

Finding the problematic data

- In Working with Pandas DataFrames, we learned the importance of examining our data when we get it; it's not a coincidence that many of the ways to inspect the data will help us find these issues.

- Examining the results of calling head() and tail() on the data is always a good first step:

```
>>> df.head()
```

Finding the problematic data

- Lastly, we can observe many NaN values in several columns, including the inclement_weather column, which appears to also contain Boolean values:

	date	station	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
0	2018-01-01T00:00:00	?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
1	2018-01-01T00:00:00	?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
2	2018-01-01T00:00:00	?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
3	2018-01-02T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-8.3	-16.1	-12.2	NaN	False
4	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False

Finding the problematic data

- Using describe(), we can see if we have any missing data and look at the 5-number summary to spot potential issues:

```
>>> df.describe()
```

Finding the problematic data

- If unknowns were encoded with another value, say 40°C, we couldn't be sure it wasn't actual data:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF
count	765.000000	577.000000	577.0	765.000000	765.000000	398.000000	11.000000
mean	5.360392	4.202773	NaN	2649.175294	-15.914379	8.632161	16.290909
std	10.002138	25.086077	NaN	2744.156281	24.242849	9.815054	9.489832
min	0.000000	0.000000	-inf	-11.700000	-40.000000	-16.100000	1.800000
25%	0.000000	0.000000	NaN	13.300000	-40.000000	0.150000	8.600000
50%	0.000000	0.000000	NaN	32.800000	-11.100000	8.300000	19.300000
75%	5.800000	0.000000	NaN	5505.000000	6.700000	18.300000	24.900000
max	61.700000	229.000000	inf	5505.000000	23.900000	26.100000	28.700000

Finding the problematic data

```
>>> df.info()
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 765 entries, 0 to 764
Data columns (total 10 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   date               765 non-null    object
 1   station            765 non-null    object
 2   PRCP               765 non-null    float64
 3   SNOW               577 non-null    float64
 4   SNWD               577 non-null    float64
 5   TMAX               765 non-null    float64
 6   TMIN               765 non-null    float64
 7   TOBS               398 non-null    float64
 8   WESF               11 non-null     float64
 9   inclement_weather  408 non-null    object
dtypes: float64(7), object(3)
memory usage: 59.9+ KB
```

Finding the problematic data

- This means that we will need to combine checks for each of the columns with the | (bitwise OR) operator:

```
>>> contain_nulls = df[
...     df.SNOW.isna() | df.SNWD.isna() | df.TOBS.isna()
...     | df.WESF.isna() | df.inclement_weather.isna()
... ]
>>> contain_nulls.shape[0]
765
>>> contain_nulls.head(10)
```

Finding the problematic data

- Looking at the top 10 rows, we can see some NaN values in each of these rows:

	date	station	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
0	2018-01-01T00:00:00	?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
1	2018-01-01T00:00:00	?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
2	2018-01-01T00:00:00	?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
3	2018-01-02T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-8.3	-16.1	-12.2	NaN	False
4	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
5	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
6	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
7	2018-01-04T00:00:00	?	20.6	229.0	inf	5505.0	-40.0	NaN	19.3	True
8	2018-01-04T00:00:00	?	20.6	229.0	inf	5505.0	-40.0	NaN	19.3	True
9	2018-01-05T00:00:00	?	0.3	NaN	NaN	5505.0	-40.0	NaN	NaN	NaN

Finding the problematic data

- Note that we can't check whether the value of the column is equal to NaN because NaN is not equal to anything:

```
>>> import numpy as np
>>> df[df.inclement_weather == 'NaN'].shape[0] # doesn't work
0
>>> df[df.inclement_weather == np.nan].shape[0] # doesn't work
0
```

Finding the problematic data

- We must use the aforementioned options (isna()/isnull()):

```
>>> df[df.inclement_weather.isna()].shape[0] # works
357
```

Finding the problematic data

- Note that inf and -inf are actually np.inf and -np.inf.
- Therefore, we can find the number of rows with inf or -inf values by doing the following:

```
>>> df[df.SNWD.isin([-np.inf, np.inf])].shape[0]
577
```

Finding the problematic data

- This only tells us about a single column, though, so we could write a function that will use a dictionary comprehension to return the number of infinite values per column in our dataframe:

```
>>> def get_inf_count(df):
...     """Find the number of inf/-inf values per column"""
...     return {
...         col: df[
...             df[col].isin([np.inf, -np.inf])
...         ].shape[0] for col in df.columns
...     }
```

Finding the problematic data

- Using our function, we find that the SNWD column is the only column with infinite values, but the majority of the values in the column are infinite:

```
>>> get_inf_count(df)
{'date': 0, 'station': 0, 'PRCP': 0, 'SNOW': 0, 'SNWD': 577,
 'TMAX': 0, 'TMIN': 0, 'TOBS': 0, 'WESF': 0,
 'inclement_weather': 0}
```

Finding the problematic data

- In addition, we will use the T attribute to transpose the data for easier viewing:

```
>>> pd.DataFrame({
...     'np.inf Snow Depth':
...         df[df.SNWD == np.inf].SNOW.describe(),
...     '-np.inf Snow Depth':
...         df[df.SNWD == -np.inf].SNOW.describe()
... }).T
```

Finding the problematic data

- They most certainly aren't that, but we can't decide what they should be, so it's probably best to leave them alone or not look at this column:

	count	mean	std	min	25%	50%	75%	max
np.inf Snow Depth	24.0	101.041667	74.498018	13.0	25.0	120.5	152.0	229.0
-np.inf Snow Depth	553.0	0.000000	0.000000	0.0	0.0	0.0	0.0	0.0

Finding the problematic data

- We are working with a year of data, but somehow, we have 765 rows, so we should check why.
- The only columns we have yet to inspect are the date and station columns.
- We can use the describe() method to see the summary statistics for them:

```
>>> df.describe(include='object')
```

Finding the problematic data

- We also know that ? occurs 367 times (765 - 398), without the need to use value_counts():

	date	station	inclement_weather
count	765	765	408
unique	324	2	2
top	2018-07-05T00:00:00	GHCND:USC00280907	False
freq	8	398	384

Finding the problematic data

- We can use the result of the duplicated() method as a Boolean mask to find the duplicate rows:

```
>>> df[df.duplicated()].shape[0]
284
```

Finding the problematic data

- However, if we pass in keep=False, we will get all the rows that are present more than once, not just each additional appearance they make:

```
>>> df[df.duplicated(keep=False)].shape[0]
482
```

Finding the problematic data

- However, we don't know if this is actually a problem:

```
>>> df[df.duplicated(['date', 'station'])].shape[0]
284
```

- Now, let's examine a few of the duplicated rows:

```
>>> df[df.duplicated()].head()
```

Finding the problematic data

- Just looking at the first five rows shows us that some rows are repeated at least three times.
- Remember that the default behavior of duplicated() is to not show the first occurrence, which means that rows 1 and 2 have another matching value in the data (same for rows 5 and 6):

	date	station	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
1	2018-01-01T00:00:00	?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
2	2018-01-01T00:00:00	?	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
5	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
6	2018-01-03T00:00:00	GHCND:USC00280907	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
8	2018-01-04T00:00:00	?	20.6	229.0	inf	5505.0	-40.0	NaN	19.3	True

Mitigating the issues

- If we then decide to remove duplicate rows using the date column and keep the data for the station that wasn't ?, in the case of duplicates, we will lose all data we have for the WESF column because the ? station is the only one reporting WESF measurements:

```
>>> df[df.WESF.notna()].station.unique()
array(['?'], dtype=object)
```

Mitigating the issues

- One satisfactory solution in this case may be to carry out the following actions:

- Perform type conversion on the date column:

```
>>> df.date = pd.to_datetime(df.date)
```

- Save the WESF column as a series:

```
>>> station_qm_wesf = df[df.station == '?']\
...     .drop_duplicates('date').set_index('date').WESF
```

Mitigating the issues

- Sort the dataframe by the station column in descending order to put the station with no ID (?) last:

```
>>> df.sort_values(
...     'station', ascending=False, inplace=True
... )
```

Mitigating the issues

- Remove rows that are duplicated based on the date, keeping the first occurrences, which will be ones where the station column has an ID (if that station has measurements).

```
>>> df_deduped = df.drop_duplicates('date')
```

Mitigating the issues

- Drop the station column and set the index to the date column (so that it matches the WESF data):

```
>>> df_deduped = df_deduped.drop(columns='station')\
...     .set_index('date').sort_index()
```

Mitigating the issues

- Since both df_deduped and station_qm_wesf are using the date as the index, the values are properly matched to the appropriate date:

```
>>> df_deduped = df_deduped.assign(WESF=
...     lambda x: x.WESF.combine_first(station_qm_wesf)
... )
```

Mitigating the issues

- Let's take a look at the result using the aforementioned implementation:

```
>>> df_deduped.shape
(324, 8)
>>> df_deduped.head()
```

Mitigating the issues

- We are now left with 324 rows—one for each unique date in our data.
- We were able to save the WESF column by putting it alongside the data from the other station:

date	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	-inf	5505.0	-40.0	NaN	NaN	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.1	-12.2	NaN	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.9	-13.3	NaN	False
2018-01-04	20.6	229.0	inf	5505.0	-40.0	NaN	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.9	-13.9	NaN	True

Mitigating the issues

- To drop all the rows with any null data (this doesn't have to be true for all the columns of the row, so be careful), we use the dropna() method; in our case, this leaves us with just 4 rows:

```
>>> df_deduped.dropna().shape
(4, 8)
```

Mitigating the issues

- We can change the default behavior to only drop a row if all the columns are null with the how argument, except this doesn't get rid of anything:

```
>>> df_deduped.dropna(how='all').shape # default is 'any'
(324, 8)
```

Mitigating the issues

- This can be achieved with the subset argument:

```
>>> df_deduped.dropna(
...     how='all', subset=['inclement_weather', 'SNOW', 'SNWD']
... ).shape
(293, 8)
```

Mitigating the issues

- For example, if we say that at least 75% of the rows must be null to drop the column, we will drop the WESF column:

```
>>> df_deduped.dropna(
...     axis='columns',
...     thresh=df_deduped.shape[0] * .75 # 75% of rows
... ).columns
Index(['PRCP', 'SNOW', 'SNWD', 'TMAX', 'TMIN', 'TOBS',
       'inclement_weather'],
      dtype='object')
```

Mitigating the issues

- Note that this can be done in-place (again, as a general rule of thumb, we should use caution with in-place operations):

```
>>> df_deduped.loc[:,'WESF'].fillna(0, inplace=True)
>>> df_deduped.head()
```

Mitigating the issues

- The WESF column no longer contains NaN values:

date	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	-inf	5505.0	-40.0	NaN	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	inf	5505.0	-40.0	NaN	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.9	-13.9	0.0	True

Mitigating the issues

- We will also do so for TMIN, which currently uses -40°C for its placeholder, despite the coldest temperature ever recorded in NYC being -15°F (-26.1°C) on February 9, 1934 (https://www.weather.gov/media/okx/Climate/CentralPark/extremes.pdf):

```
>>> df_deduped = df_deduped.assign(
...     TMAX=lambda x: x.TMAX.replace(5505, np.nan),
...     TMIN=lambda x: x.TMIN.replace(-40, np.nan)
... )
```

Mitigating the issues

- Notice we don't have the 'nearest' option, like we did when we were reindexing, which would have been the best option; so, to illustrate how this works, let's use forward-filling:

```
>>> df_deduped.assign(
...     TMAX=lambda x: x.TMAX.fillna(method='ffill'),
...     TMIN=lambda x: x.TMIN.fillna(method='ffill')
... ).head()
```

Mitigating the issues

- Take a look at the TMAX and TMIN columns on January 1st and 4th.
- Both are NaN on the 1st because we don't have data before then to bring forward, but the 4th now has the same values as the 3rd:

date	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	-inf	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	inf	-4.4	-13.9	NaN	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.9	-13.9	0.0	True

Mitigating the issues

- If we want to handle the nulls and infinite values in the SNWD column, we can use the np.nan_to_num() function; it turns NaN into 0 and inf/-inf into very large positive/negative finite numbers, making it possible for machine learning models (discussed in Getting Started with Machine Learning in Python) to learn from this data:

```
>>> df_deduped.assign(
...     SNWD=lambda x: np.nan_to_num(x.SNWD)
... ).head()
```

Mitigating the issues

- However, we don't know what to do with np.inf, and the large positive numbers, arguably, make this more confusing to interpret:

date	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	-1.797693e+308	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	-1.797693e+308	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	-1.797693e+308	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	1.797693e+308	NaN	NaN	NaN	19.3	True
2018-01-05	14.2	127.0	1.797693e+308	-4.4	-13.9	-13.9	0.0	True

Mitigating the issues

- To show how the upper bound works, we will use the snowfall (SNOW) as an estimate:

```
>>> df_deduped.assign(
...     SNWD=lambda x: x.SNWD.clip(0, x.SNOW)
... ).head()
```

Mitigating the issues

- The values of SNWD for January 1st through 3rd are now 0 instead of -inf, while the values of SNWD for January 4th and 5th went from inf to that day's value for SNOW:

date	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	0.0	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	0.0	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	0.0	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	229.0	NaN	NaN	NaN	19.3	True
2018-01-05	14.2	127.0	127.0	-4.4	-13.9	-13.9	0.0	True

Mitigating the issues

- We can combine imputation with the fillna() method.
- As an example, let's fill in the NaN values for TMAX and TMIN with their medians and TOBS with the average of TMIN and TMAX (after imputing them):

```
>>> df_deduped.assign(
...     TMAX=lambda x: x.TMAX.fillna(x.TMAX.median()),
...     TMIN=lambda x: x.TMIN.fillna(x.TMIN.median()),
...     # average of TMAX and TMIN
...     TOBS=lambda x: x.TOBS.fillna((x.TMAX + x.TMIN) / 2)
... ).head()
```

Mitigating the issues

- This means that when we impute TOBS and also don't have TMAX and TMIN in the data, we get 10°C:

date	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	-inf	14.4	5.6	10.0	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.1	-12.2	0.0	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.9	-13.3	0.0	False
2018-01-04	20.6	229.0	inf	14.4	5.6	10.0	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.9	-13.9	0.0	True

Mitigating the issues

- We will cover rolling calculations and apply() in Aggregating Pandas DataFrames, so this is just a preview:

```
>>> df_deduped.apply(lambda x:
...     # Rolling 7-day median (covered in chapter 4).
...     # we set min_periods (# of periods required for
...     # calculation) to 0 so we always get a result
...     x.fillna(x.rolling(7, min_periods=0).median())
... ).head(10)
```

Mitigating the issues

- In reality, it was slightly warmer that day (around -3°C):

date	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	-inf	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.30	-16.1	-12.20	0.0	False
2018-01-03	0.0	0.0	-inf	-4.40	-13.9	-13.30	0.0	False
2018-01-04	20.6	229.0	inf	-6.35	-15.0	-12.75	19.3	True
2018-01-05	14.2	127.0	inf	-4.40	-13.9	-13.90	0.0	True
2018-01-06	0.0	0.0	-inf	-10.00	-15.6	-15.00	0.0	False
2018-01-07	0.0	0.0	-inf	-11.70	-17.2	-16.10	0.0	False
2018-01-08	0.0	0.0	-inf	-7.80	-16.7	-8.30	0.0	False
2018-01-10	0.0	0.0	-inf	5.00	-7.8	-7.80	0.0	False
2018-01-11	0.0	0.0	-inf	4.40	-7.8	1.10	0.0	False

Mitigating the issues

- Let's combine this with the apply() method to interpolate all of our columns at once:

```
>>> df_deduped.reindex(
...     pd.date_range('2018-01-01', '2018-12-31', freq='D')
... ).apply(lambda x: x.interpolate()).head(10)
```

Mitigating the issues

- Check out January 9th, which we didn't have previously—the values for TMAX, TMIN, and TOBS are the average of the values for the day prior (January 8th) and the day after (January 10th):

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
2018-01-01	0.0	0.0	-inf	NaN	NaN	NaN	0.0	NaN
2018-01-02	0.0	0.0	-inf	-8.3	-16.10	-12.20	0.0	False
2018-01-03	0.0	0.0	-inf	-4.4	-13.90	-13.30	0.0	False
2018-01-04	20.6	229.0	inf	-4.4	-13.90	-13.60	19.3	True
2018-01-05	14.2	127.0	inf	-4.4	-13.90	-13.90	0.0	True
2018-01-06	0.0	0.0	-inf	-10.0	-15.60	-15.00	0.0	False
2018-01-07	0.0	0.0	-inf	-11.7	-17.20	-16.10	0.0	False
2018-01-08	0.0	0.0	-inf	-7.8	-16.70	-8.30	0.0	False
2018-01-09	0.0	0.0	-inf	-1.4	-12.25	-8.05	0.0	NaN
2018-01-10	0.0	0.0	-inf	5.0	-7.80	-7.80	0.0	False

Summary

- In this lesson, we learned more about what data wrangling is (aside from a data science buzzword) and got some firsthand experience with cleaning and reshaping our data.

- Utilizing the requests library, we once again practiced working with APIs to extract data of interest; then, we used pandas to begin our introduction to data wrangling, which we will continue in the next lesson.

- Finally, we learned how to deal with duplicate, missing, and invalid data points in various ways and discussed the ramifications of those decisions.

"Complete Exercises in lab8.md for Homework"

"Complete Lab 8 as a workalong with the instructor"