وزارة الاتصالات
وتكنولوجيا المعلومات

رواد مصر الرقمية

Data Wrangling with Pandas

13/07/2024

Adding and Removing Data

Before adding or removing data, note that most methods return a new DataFrame, but some change the data in place. If we want to avoid changing the original data, we should copy the dataframe before making modifications:

```
df_to_modify = df.copy()
```

We will once again be working with the earthquake data, but this time, we will only read in a subset of the columns:

```
>>> import pandas as pd

>>> df = pd.read_csv(
...     'data/earthquakes.csv',
...     usecols=[
...         'time', 'title', 'place', 'magType',
...         'mag', 'alert', 'tsunami'
...     ]
... )
```

Creating New Data

Creating new columns can be achieved in the same fashion as variable assignment.

```
>>> df['source'] = 'USGS API'
>>> df.head()
```

The new column is created to the right of the original columns, with a value of USGS API for every row:

	alert	mag	magType	place	time	title	tsunami	source
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API
2	NaN	3.42	ml	8km NE of Aguanga, CA	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	USGS API
3	NaN	0.44	ml	9km NE of Aguanga, CA	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	USGS API
4	NaN	2.16	md	10km NW of Avenal, CA	1539474716050	M 2.2 - 10km NW of Avenal, CA	0	USGS API

Important note

We cannot create the column with attribute notation (`df.source`) because the dataframe doesn't have that attribute yet, so we must use dictionary notation (`df['source']`).

Creating New Data

We aren't limited to broadcasting one value to the entire column; we can have the column hold the result of Boolean logic or a mathematical equation.

```
>>> df['mag_negative'] = df.mag < 0
>>> df.head()
```

Note that the new column has been added to the right:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False
2	NaN	3.42	ml	8km NE of Aguanga, CA	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	USGS API	False
3	NaN	0.44	ml	9km NE of Aguanga, CA	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	USGS API	False
4	NaN	2.16	md	10km NW of Avenal, CA	1539474716050	M 2.2 - 10km NW of Avenal, CA	0	USGS API	False

Creating New Data

We noted some data consistency issues in the place column, with multiple names for the same entity. For instance, earthquakes in California are marked as both CA and California.

```
>>> df.place.str.extract(r', (.*$)')[0].sort_values().unique()
array(['Afghanistan', 'Alaska', 'Argentina', 'Arizona',
       'Arkansas', 'Australia', 'Azerbaijan', 'B.C., MX',
       'Barbuda', 'Bolivia', ..., 'CA', 'California', 'Canada',
       'Chile', ..., 'East Timor', 'Ecuador', 'Ecuador region',
       ..., 'Mexico', 'Missouri', 'Montana', 'NV', 'Nevada',
       ..., 'Yemen', nan], dtype=object)
```

Creating New Data

We can use the replace() method to replace patterns in the place column as we see fit:

```
>>> df['parsed_place'] = df.place.str.replace(
...     r'.* of ', '', regex=True # remove <x> of <x>
... ).str.replace(
...     'the ', '' # remove "the "
... ).str.replace(
...     r'CA$', 'California', regex=True # fix California
... ).str.replace(
...     r'NV$', 'Nevada', regex=True # fix Nevada
... ).str.replace(
...     r'MX$', 'Mexico', regex=True # fix Mexico
... ).str.replace(
...     r' region$', '', regex=True # fix " region" endings
... ).str.replace(
...     'northern ', '' # remove "northern "
... ).str.replace(
...     'Fiji Islands', 'Fiji' # line up the Fiji places
... ).str.replace( # remove anything else extraneous from start
...     r'^.*, ', '', regex=True
... ).str.strip() # remove any extra spaces
```

Creating New Data

Notice that there is arguably still more to fix here with South Georgia and South Sandwich Islands and South Sandwich Islands. We could address this with another call to replace()

```
>>> df.parsed_place.sort_values().unique()
array([..., 'California', 'Canada', 'Carlsberg Ridge', ...,
       'Dominican Republic', 'East Timor', 'Ecuador',
       'El Salvador', 'Fiji', 'Greece', ...,
       'Mexico', 'Mid-Indian Ridge', 'Missouri', 'Montana',
       'Nevada', 'New Caledonia', ...,
       'South Georgia and South Sandwich Islands',
       'South Sandwich Islands', ..., 'Yemen'], dtype=object)
```

Important note

In practice, entity recognition can be an extremely difficult problem, where we may look to employ **natural language processing** (**NLP**) algorithms to help us. While this is well beyond the scope of this book, more information can be found at `https://www.kdnuggets.com/2018/12/introduction-named-entity-recognition.html`.

Creating New Data

Pandas also provides us with a way to make many new columns at once in one method call.

With the assign() method, the arguments are the names of the columns we want to create (or overwrite), and the values are the data for the columns.

```
>>> df.assign(
...     in_ca=df.parsed_place.str.endswith('California'),
...     in_alaska=df.parsed_place.str.endswith('Alaska')
... ).sample(5, random_state=0)
```

Creating New Data

Note that assign() doesn't change our original dataframe; instead, it returns a new DataFrame object with these columns added.

If we want to replace our original dataframe with this, we just use variable assignment to store the result of assign() in df (for example, df = df.assign(...)):

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place	in_ca	in_alaska
7207	NaN	4.80	mwr	73km SSW of Masachapa, Nicaragua	1537749595210	M 4.8 - 73km SSW of Masachapa, Nicaragua	0	USGS API	False	Nicaragua	False	False
4755	NaN	1.09	ml	28km NNW of Packwood, Washington	1538227540460	M 1.1 - 28km NNW of Packwood, Washington	0	USGS API	False	Washington	False	False
4595	NaN	1.80	ml	77km SSW of Kaktovik, Alaska	1538259609862	M 1.8 - 77km SSW of Kaktovik, Alaska	0	USGS API	False	Alaska	False	True
3566	NaN	1.50	ml	102km NW of Arctic Village, Alaska	1538464751822	M 1.5 - 102km NW of Arctic Village, Alaska	0	USGS API	False	Alaska	False	True
2182	NaN	0.90	ml	26km ENE of Pine Valley, CA	1538801713880	M 0.9 - 26km ENE of Pine Valley, CA	0	USGS API	False	California	True	False

Creating New Data

The assign() method also accepts lambda functions (anonymous functions usually defined in one line and for single use); assign() will pass the dataframe into the lambda function as x, and we can work from there.

```
>>> df.assign(
...     in_ca=df.parsed_place == 'California',
...     in_alaska=df.parsed_place == 'Alaska',
...     neither=lambda x: ~x.in_ca & ~x.in_alaska
... ).sample(5, random_state=0)
```

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place	in_ca	in_alaska	neither
7207	NaN	4.80	mwr	73km SSW of Masachapa, Nicaragua	1537749595210	M 4.8 - 73km SSW of Masachapa, Nicaragua	0	USGS API	False	Nicaragua	False	False	True
4755	NaN	1.09	ml	28km NNW of Packwood, Washington	1538227540460	M 1.1 - 28km NNW of Packwood, Washington	0	USGS API	False	Washington	False	False	True
4595	NaN	1.80	ml	77km SSW of Kaktovik, Alaska	1538259609862	M 1.8 - 77km SSW of Kaktovik, Alaska	0	USGS API	False	Alaska	False	True	False
3566	NaN	1.50	ml	102km NW of Arctic Village, Alaska	1538464751822	M 1.5 - 102km NW of Arctic Village, Alaska	0	USGS API	False	Alaska	False	True	False
2182	NaN	0.90	ml	26km ENE of Pine Valley, CA	1538801713880	M 0.9 - 26km ENE of Pine Valley, CA	0	USGS API	False	California	True	False	False

Creating New Data

Now that we have seen how to add new columns, let's take a look at adding new rows.

```
>>> tsunami = df[df.tsunami == 1]
>>> no_tsunami = df[df.tsunami == 0]

>>> tsunami.shape, no_tsunami.shape
((61, 10), (9271, 10))
```

To append rows to the bottom of our dataframe, we can either use pd.concat() or the append() method of the dataframe itself.

```
>>> pd.concat([tsunami, no_tsunami]).shape
(9332, 10) # 61 rows + 9271 rows
```

```
>>> tsunami.append(no_tsunami).shape
(9332, 10) # 61 rows + 9271 rows
```

Creating New Data

Suppose that we now want to work with some of the columns we ignored when we read in the data.

```
>>> additional_columns = pd.read_csv(
...     'data/earthquakes.csv', usecols=['tz', 'felt', 'ids']
... )
>>> pd.concat([df.head(2), additional_columns.head(2)], axis=1)
```

Since the indices of the dataframes align, the additional columns are placed to the right of our original columns:

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place	felt	ids	tz
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False	California	NaN	,ci37389218,	-480.0
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False	California	NaN	,ci37389202,	-480.0

Creating New Data

The concat() function uses the index to determine how to concatenate the values. If they don't align, this will generate additional rows because pandas won't know how to align them.

```
>>> additional_columns = pd.read_csv(
...     'data/earthquakes.csv',
...     usecols=['tz', 'felt', 'ids', 'time'],
...     index_col='time'
... )
>>> pd.concat([df.head(2), additional_columns.head(2)], axis=1)
```

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place	felt	ids	tz
0	NaN	1.35	ml	9km NE of Aguanga, CA	1.539475e+12	M 1.4 - 9km NE of Aguanga, CA	0.0	USGS API	False	California	NaN	NaN	NaN
1	NaN	1.29	ml	9km NE of Aguanga, CA	1.539475e+12	M 1.3 - 9km NE of Aguanga, CA	0.0	USGS API	False	California	NaN	NaN	NaN
1539475129610	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	,ci37389202,	-480.0
1539475168010	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	,ci37389218,	-480.0

Creating New Data

Say we want to concatenate the tsunami and no_tsunami dataframes, but the no_tsunami dataframe has an additional column (suppose we added a new column to it called type).

```
>>> pd.concat(
...     [
...         tsunami.head(2),
...         no_tsunami.head(2).assign(type='earthquake')
...     ],
...     join='inner'
... )
```

Creating New Data

Notice that the type column from the no_tsunami dataframe doesn't show up because it wasn't present in the tsunami dataframe.

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place
36	NaN	5.00	mww	165km NNW of Flying Fish Cove, Christmas Island	1539459504090	M 5.0 - 165km NNW of Flying Fish Cove, Christm...	1	USGS API	False	Christmas Island
118	green	6.70	mww	262km NW of Ozernovskiy, Russia	1539429023560	M 6.7 - 262km NW of Ozernovskiy, Russia	1	USGS API	False	Russia
0	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False	California
1	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False	California

Creating New Data

If the index is not meaningful, we can also pass in ignore_index to get sequential values in the index:

```
>>> pd.concat(
...     [
...         tsunami.head(2),
...         no_tsunami.head(2).assign(type='earthquake')
...     ],
...     join='inner', ignore_index=True
... )
```

	alert	mag	magType	place	time	title	tsunami	source	mag_negative	parsed_place
0	NaN	5.00	mww	165km NNW of Flying Fish Cove, Christmas Island	1539459504090	M 5.0 - 165km NNW of Flying Fish Cove, Christm...	1	USGS API	False	Christmas Island
1	green	6.70	mww	262km NW of Ozernovskiy, Russia	1539429023560	M 6.7 - 262km NW of Ozernovskiy, Russia	1	USGS API	False	Russia
2	NaN	1.35	ml	9km NE of Aguanga, CA	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0	USGS API	False	California
3	NaN	1.29	ml	9km NE of Aguanga, CA	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0	USGS API	False	California

Deleting Unwanted Data

Like adding data, we can use dictionary syntax to delete unwanted columns, just as we would when removing keys from a dictionary.

Both del df['<column_name>'] and df.pop('<column_name>') will work, provided that there is indeed a column with that name; otherwise, we will get a KeyError.

The difference here is that while del removes it right away, pop() will return the column that we are removing.

```
>>> del df['source']
>>> df.columns
Index(['alert', 'mag', 'magType', 'place', 'time', 'title',
       'tsunami', 'mag_negative', 'parsed_place'],
      dtype='object')
```

Deleting Unwanted Data

Note that if we aren't sure whether the column exists, we should put our column deletion code in a try...except block:

```
try:
    del df['source']
except KeyError:
    pass # handle the error here
```

We can use pop() to grab the series for the mag_negative column, which we can use as a Boolean mask later without having it in our dataframe:

```
>>> mag_negative = df.pop('mag_negative')
>>> df.columns
Index(['alert', 'mag', 'magType', 'place', 'time', 'title',
       'tsunami', 'parsed_place'],
      dtype='object')
```

Deleting Unwanted Data

DataFrame objects have a drop() method for removing multiple rows or columns either in-place or returning a new DataFrame object.

```
>>> df.drop([0, 1]).head(2)
```

Notice that the index starts at 2 because we dropped 0 and 1:

	alert	mag	magType	place	time	title	tsunami	parsed_place
2	NaN	3.42	ml	8km NE of Aguanga, CA	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0	California
3	NaN	0.44	ml	9km NE of Aguanga, CA	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0	California

Deleting Unwanted Data

By default, drop() assumes that we want to delete rows (axis=0). If we want to drop columns, we can either pass axis=1 or specify our list of column names using the columns argument.

```
>>> cols_to_drop = [
...     col for col in df.columns
...     if col not in [
...         'alert', 'mag', 'title', 'time', 'tsunami'
...     ]
... ]
>>> df.drop(columns=cols_to_drop).head()
```

	alert	mag	time	title	tsunami
0	NaN	1.35	1539475168010	M 1.4 - 9km NE of Aguanga, CA	0
1	NaN	1.29	1539475129610	M 1.3 - 9km NE of Aguanga, CA	0
2	NaN	3.42	1539475062610	M 3.4 - 8km NE of Aguanga, CA	0
3	NaN	0.44	1539474978070	M 0.4 - 9km NE of Aguanga, CA	0
4	NaN	2.16	1539474716050	M 2.2 - 10km NW of Avenal, CA	0

Deleting Unwanted Data

Whether we decide to pass axis=1 to drop() or use the columns argument, our result will be equivalent:

```
>>> df.drop(columns=cols_to_drop).equals(
...     df.drop(cols_to_drop, axis=1)
... )
True
```

By default, drop() will return a new DataFrame object; however, if we really want to remove the data from our original dataframe, we can pass in inplace=True

```
>>> df.drop(columns=cols_to_drop, inplace=True)
>>> df.head()
```

Data Wrangling with Pandas

In this section, 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

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.

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. In practice, there are three common tasks involved in the data wrangling process:

- Data cleaning
- Data transformation
- Data enrichment

It should be noted that there is no inherent order to these tasks, and it is highly probable that we will perform each many times throughout the data wrangling process.

Data Wrangling – Cleaning

An initial round of data cleaning will often give us the bare minimum we need to start exploring our data. 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 Wrangling – Transformation

In data transformation, we focus on changing our data's structure to facilitate our downstream analyses; this usually involves changing which data goes along the rows and which goes down the columns.

Most data we will find is either wide format or long format; each of these formats has its merits, and it's important to know which one we will need for our analysis.

Wide

variables (TMAX, TMIN, TOBS); observations (rows)

	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

Long

datatype: variable names; value: variable values; rows 0–2: repeated values for **date** column

	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 Wrangling – Transformation

First, we will import pandas and matplotlib (to help illustrate the strengths and weaknesses of each format when it comes to visualizations)

```
>>> 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
```

The Wide Data Format

With wide format data, we represent measurements of variables with their own columns, and each row represents an observation of those variables.

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

This makes it easy for us to compare variables across observations, get summary statistics, perform operations, and present our data;

However, some visualizations don't work with this data format because they may rely on the long format to split, size, and/or color the plot content.

	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

The Wide Data Format

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

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

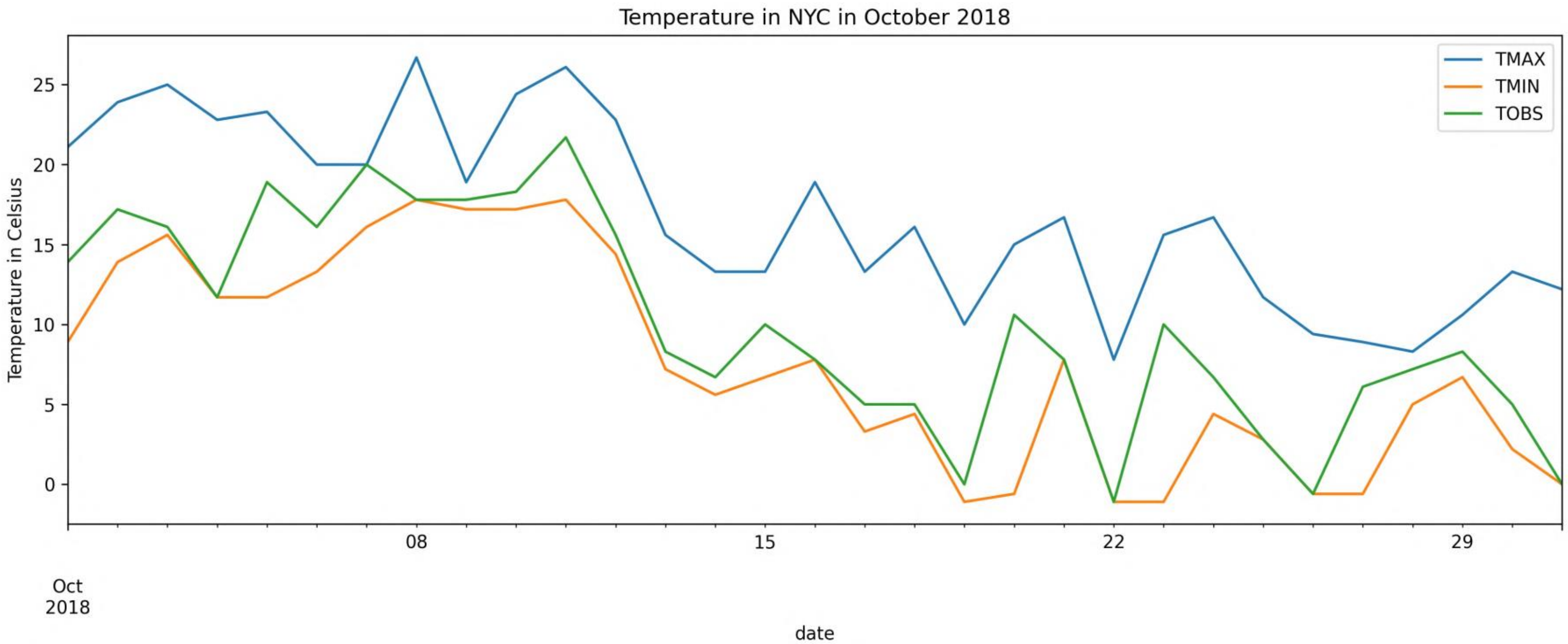
With hardly any effort on our part, we get summary statistics for the dates, maximum temperature, minimum temperature, and temperature at the time of observation:

	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

The Wide Data Format

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()
```



The Long Data Format

Long format data will have a row for each observation of a variable; this means that, if we have three variables being measured daily, we will have three rows for each day we record observations.

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

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

The Long Data Format

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)
```

The value column provides summary statistics, but it combines daily maximum temperatures, minimum temperatures, and temperatures at the time of observation.

The maximum reflects the highest daily maximum temperatures, and the minimum reflects the lowest daily minimum temperatures, making this summary data less useful.

	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

The Long Data Format

This format is not very easy to digest and certainly shouldn't be how we present data;

However, it makes it easy to create visualizations where our plotting library can color lines by the name of the variable, size the points by the values of a certain variable, and perform splits for faceting.

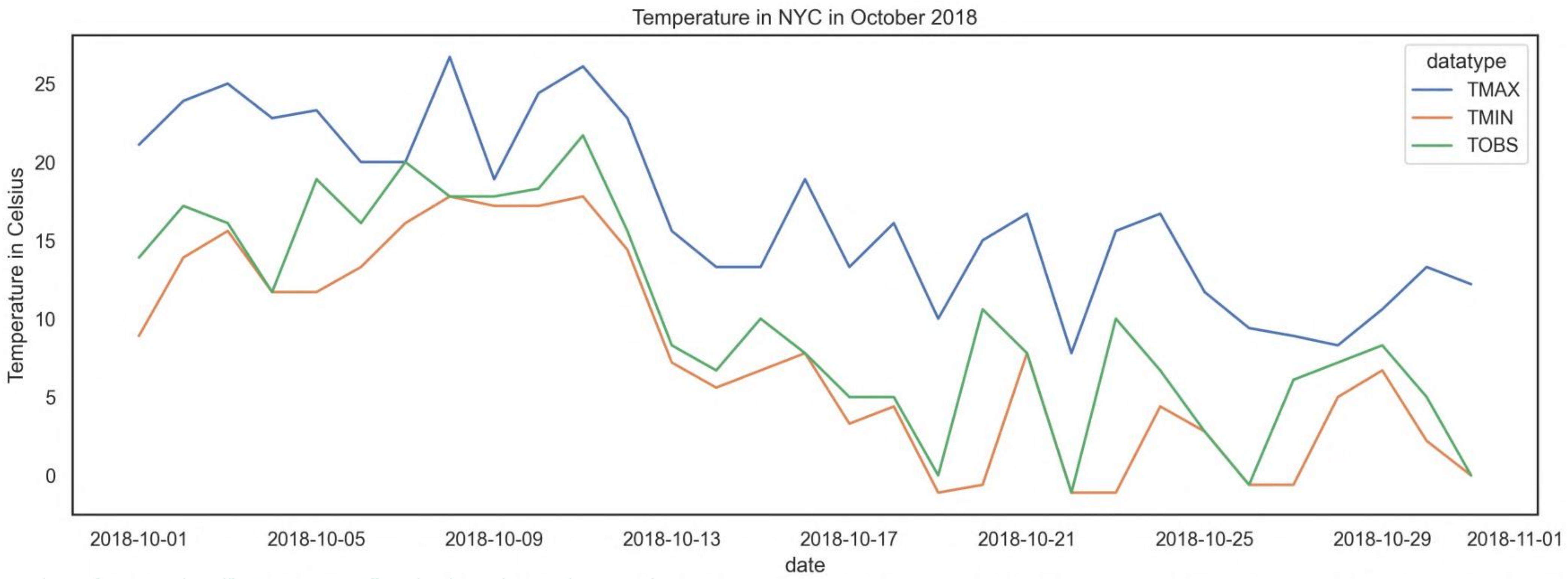
```
>>> import seaborn as sns

>>> 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()
```

The Long Data Format

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:



The Long Data Format

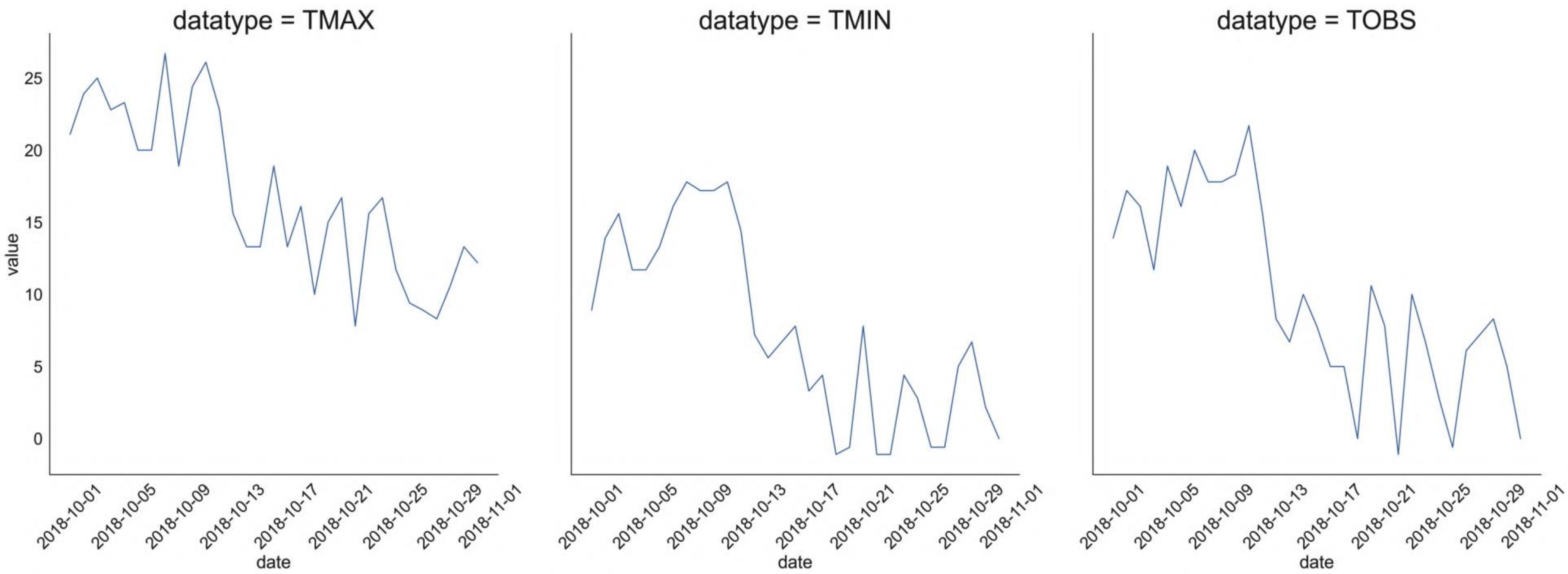
Seaborn lets us specify the column to use for hue, which colored the lines in Figure by the temperature type.

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()
```

The Long Data Format

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



Wide vs. Long Data Formats

In the Reshaping data section, we will cover how to transform our data from wide to long format by melting, and from long to wide format by pivoting.

Additionally, we will learn how to transpose data, which flips the columns and the rows.

Data Enrichment

Once our data is cleaned and formatted for analysis, we may need to enrich it. Data enrichment improves quality by adding relevant information, which is crucial for modeling and machine learning as part of feature engineering.

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 Data

To begin, we will start by exploring the weather API that's provided by the NCEI. Then, in the next section, we will learn about data wrangling using temperature data that was previously obtained from this API.

```
>>> 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 Data

To use the make_request() function, we need to learn how to form our request. The NCEI has a helpful getting started page (https://www.ncdc.noaa.gov/cdo-web/webservices/v2#gettingStarted) that shows us how to form requests.

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

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

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

Exploring an API to Find and Collect Temperature Data

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'])
```

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 Data

Now, 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()
```

	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 Data

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)
```

Time for plan B: let's use LaGuardia Airport as the station instead of Central Park for the remainder of this section.

Alternatively, we could have grabbed data for all the stations that cover New York City; however, since this would give us multiple entries per day for some of the temperature measurements, we won't do so here—we would need skills that will be covered later.

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()
```

We retrieved long format data from the API; for our analysis, we want wide format data, but we will address that later.

	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

Cleaning Data

For now, we will focus on making little tweaks to the data that will make it easier for us to use: renaming columns, converting each column into the most appropriate data type, sorting, and reindexing.

Often, this will be the time to filter the data down, but we did that when we worked on requesting data from the API.

Cleaning Data - 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.

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

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

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

Cleaning Data - 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.

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

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')
```

Cleaning Data - Type Conversion

With type conversion, we aim to reconcile what the current data types are with what we believe they should be; we will be changing how our data is represented.

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
```

Cleaning Data - 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
```

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
```

Cleaning Data - Type Conversion

We can use the assign() method to handle any type conversions by passing the column names as named parameters and their new values as the value for that argument to the method call.

```
>>> 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()
```

Cleaning Data - 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

Cleaning Data - Type Conversion

Additionally, we can use the astype() method to convert one column at a time.

```
>>> 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()
```

	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

Cleaning Data - Type Conversion

It's also important to mention that we don't have to know whether to convert the column into a float or an integer: we can use pd.to_numeric(), which will convert the data into floats if it sees decimals. 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).

Lastly, we have two columns with data currently being stored as strings that can be represented in a better way for this dataset.

The station and datatype columns only have one and three distinct values, respectively, meaning that we aren't being efficient with our memory use since we are storing them as strings.

Cleaning Data - Type Conversion

We only have one value for the station column and only three distinct values for the datatype column (TAVG, TMAX, TMIN).

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')
```

Cleaning Data - 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

Reordering, Reindexing, & 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, & Sorting Data

The result is like:

	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, & 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.

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

	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, & Sorting Data

Tip

In `pandas`, the index is tied to the rows—when we drop rows, filter, or do anything that returns only some of the rows, our index may look out of order (as we saw in the previous examples). At the moment, the index just represents the row number in our data, so we may be interested in changing the values so that we have the first entry at index `0`. To have `pandas` do so automatically, we can pass `ignore_index=True` to `sort_values()`.

Reordering, Reindexing, & Sorting Data

Pandas also provides an additional way to look at a subset of the sorted values; we can use nlargest() to grab the n rows with the largest values according to specific criteria and nsmallest() to grab the n smallest rows, without the need to sort the data beforehand.

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

	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

رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

Q&A

Questions and answers

Thanks!