Data Wrangling with Pandas II

20/07/2024

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

Reordering, Reindexing, & Sorting Data

We aren't limited to sorting values; if we wish, we can even order the columns alphabetically and sort the rows by their index values.

For these tasks, we can use the sort_index() method.

```
>>> 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')
```

By default, sort_index() will target the rows, When we want to target columns, we must pass in axis=1.

Reordering, Reindexing, & Sorting Data

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

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

	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

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:

Reordering, Reindexing, & Sorting Data

The sort_index() method can also help us get an accurate answer when we're \ testing two dataframes for equality.

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

Important note

Both sort_index() and sort_values() return new DataFrame objects. We must pass in inplace=True to update the dataframe we are working with.

Reordering, Reindexing, & 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()
```

	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
date							
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, & Sorting Data

Setting the index to a datetime lets us take advantage of datetime slicing and indexing:

Both Inclusive

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

	datatype	station	flags	temp_C	temp_C_whole	temp_F	temp_F_whole
date							
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

رواد مصر الرقمية

Reordering, Reindexing, & Sorting Data

وزارة الاتصالات وتكنولوجيا المعلومات

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

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

Reshaping Data

Sometimes, we need to be able to restructure data into both wide and long formats, depending on the analysis we want to perform.
For many analyses, we will want wide format data so that we can look at the summary statistics easily and share our results in that format.

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

Long | Wide

	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
... )
```

	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

We will discuss transposing, pivoting, and melting our data.

Note that after reshaping the data, we will often revisit the data cleaning tasks as things may have changed, or we may need to change things we couldn't access easily before.

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):

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

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

It may not be immediately apparent how useful this can be, but we will see this a quite few times throughout this course.

Pivoting DataFrames

We **pivot** our data to go from long format to wide format. The pivot() method performs this restructuring of our DataFrame object.

To pivot, we need to tell pandas which column currently holds the values (with the values argument) and the column that contains what will become the column names in wide format (the columns argument).
Optionally, we can provide a new index (the index argument).

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

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, with the data in wide format, we can easily get meaningful summary statistics with the describe() method:

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

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

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):

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()
```

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

We can create an index from any number of columns with set_index(). 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

The pivot() method expects the data to only have one column to set as the index; if we have a multi-level index, we should use the unstack() method instead.

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

	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

Order matters here because, by default, unstack() will move the innermost level of the index to the columns; 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.

Pivoting DataFrames

The unstack() method has the added benefit of allowing us to specify how to fill in missing values that come into existence upon reshaping the data.

```
>>> 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']
```

		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()
```

	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

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

	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

To summarize, unstack() should be our method of choice when we have a multi level index and would like to move one or more of the levels to the columns; however, if we are simply using a single index, the pivot() method's syntax is likely to be easier to specify correctly since it's more apparent which data will end up where.

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()
```

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

We can use the melt() method for flexible reshaping—allowing us to turn this into long format, similar to what we got from the API. Melting requires that we specify the following:

- Which column(s) uniquely identify a row in the wide format data with the id_vars argument
- Which column(s) contain(s) the variable(s) with the value_vars argument

Optionally, we can also specify how to name the column containing the variable names in the long format data (var_name) and the name for the column containing their values (value_name). By default, these will be variable and value, respectively.

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()
```

	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

Just as we had an alternative way of pivoting data with the unstack() method, we also have another way of melting data with the stack() method.

This method will pivot the columns into the innermost level of the index (resulting in an index of type MultiIndex), so we need to double-check our index before calling it.

```
>>> 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.

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

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

Initially, we used set_index() to set the index to the date column because we didn't want to melt that; this formed the first level of the multi-level index.

Then, the stack() method moved the TMAX, TMIN, and TOBS columns into the second level of the index. 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

This is separated from the rest of the data cleaning discussion because it is an example where we will do some initial data cleaning, then reshape our data, and finally look to handle these potential issues.

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

The dirty_data.csv file contains wide format data from the weather API that has been altered to introduce many common data issues:
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

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

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

	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()
```

	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	264[illegible]294	-15.914379	8.632161	16.290909
std	10.002138	25.086077	[illegible]	274[illegible]281	24.242849	9.815054	9.489832
min	0.000000	0.000000	[illegible]	-1[illegible]000	-40.000000	-16.100000	1.800000
25%	0.000000	0.000000	[illegible]	1[illegible]000	-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

Useless

Unreliable.

Finding the Problematic Data

We can use the info() method to see if we have any missing values and check that our columns have the expected data types.

```
>>> 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
```

Null Values

Notice that the ? value that we saw for the station column when we used head() doesn't show up here—it's important to inspect our data from many different angles.

Not Boolean

Finding the Problematic Data

Now, let's track down those null values. Both Series and DataFrame objects provide two methods to do so: isnull() and isna().

```
>>> 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)
```

	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

If we look at the shape attribute of contain_nulls dataframe, we will see that every single row contains some null data.

Finding the Problematic Data

Tip

By default, the `sort_values()` method that we discussed earlier in this chapter will put any `NaN` values last. We can change this behavior (to put them first) by passing in `na_position='first'`, which can also be helpful when looking for patterns in the data when the sort columns have null values.

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
```

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
```

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}
```

Before we can decide on how to handle the infinite values, we should look at the summary statistics for snowfall (SNOW), which forms a big part of determining the snow depth (SNWD).

```
>>> 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

The snow depth was recorded as negative infinity when there was no snowfall; however, we can't be sure this isn't just a coincidence going forward.

	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

If we are just going to be working with this fixed date range, we can treat that as having a depth of 0 or NaN because it didn't snow.

Unfortunately, we can't really make any assumptions with the positive infinity entries. So, we can't decide what they should be, it's probably best to leave them alone or not look at this column.

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')
```

	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

In 765 rows of data, the date column only has 324 unique values, with some dates being present as many as eight times (freq).

There are only two unique values for the station column, with the most frequent being GHCND:USC00280907.

Since we saw some station IDs with the value of ? when we used head() earlier, we know that is the other value; however, we can use unique() to see all the unique values if we hadn't.

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

Finding the Problematic Data

Upon seeing that we had 765 rows of data and two distinct values for the station ID, we might have assumed that each day had two entries—one per station. However, this would only account for 730 rows, and we also now know that we are missing some dates.

Let's see whether we can find any duplicate data that could account for this. We can use the result of the duplicated() method as a Boolean mask to find the duplicate rows:

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

Counts all

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

Finding the Problematic Data

There is also a subset argument (first positional argument), which allows us to focus just on the duplicates of certain columns.

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

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

	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

Some rows are repeated at least three times. Remember that the default behavior of duplicated() is to not show the first occurrence

Mitigating the Issues

Our data is in an unsatisfactory state, and improving it isn't always straightforward. The simplest approach might be to remove duplicate rows, but we must consider the impact on our analysis. If our data was part of a larger dataset with additional columns, the remaining data might still be duplicated for other reasons. We need to consult the data source and available documentation to understand this

Since both stations are for New York City, we can drop the station column.
If we remove duplicate rows based on the date column and keep data from the non-? station, we will lose all WESF data because only the ? station reports 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:

1. Perform type conversion on the date column:

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

2. Save the WESF column as a series:

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

3. 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

4. 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')
```

5. 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

6. Update the WESF column using the combine_first() method to coalesce (just as in SQL for those coming from a SQL background) the values to the first non-null entry;

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

This means that if we had data from both stations, we would first take the value provided by the station with an ID, and if (and only if) that station was null would we take the value from the station without an ID (?).

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

Mitigating the Issues

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

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

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
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

Now, let's deal with the null data. We can choose to drop it, replace it with some arbitrary value, or impute it using surrounding data.

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)
```

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

We can also use a subset of columns to determine what to drop. Say we wanted to look at snow data:

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

Note that this operation can also be performed along the columns, and that we can provide a threshold for the number of null values that must be observed to drop the data with the thresh argument. 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

Since we have a lot of null values, we will likely be more interested in keeping these values, and perhaps finding a better way to represent them.

To fill in null values with other data, we use the fillna() method, which gives us the option of specifying a value or a strategy for how to perform the filling.

The WESF column contains mostly null values, it is a measurement in milliliters that takes on the value of NaN when there is no water equivalent of snowfall, we can fill in the nulls with zeros.

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

Mitigating the Issues

The WESF column no longer contains NaN values:

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
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

At this point, we have done everything we can without distorting the data. We know that we are missing dates, but if we reindex, we don't know how to fill in the resulting NaN values.

With the weather data, we can't assume that because it snowed one day that it will snow the next, or that the temperature will be the same.

For this reason, note that the following examples are just for illustrative purposes only—just because we can do something doesn't mean we should. The right solution will most likely depend on the domain and the problem we are looking to solve.

Mitigating the Issues

We know that when TMAX is the temperature of the Sun, it must be because there was no measured value, so let's replace it with NaN. We will also do so for TMIN, which currently uses -40°C for its placeholder.

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

We will also make an assumption that the temperature won't change drastically from day to day. Note that this is actually a big assumption, but it will allow us to understand how the fillna() method works when we provide a strategy through the method parameter: 'ffill' to forward-fill or 'bfill' to back-fill.

Mitigating the Issues

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()
```

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
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 to learn from this data:

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

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
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

This approach isn't suitable for our use case. For -np.inf, we can set SNWD to 0 since there was no snowfall on those days.
However, np.inf and large positive numbers make the data harder to interpret

Mitigating the Issues

Depending on the data we are working with, we may choose to use the clip() method as an alternative to the np.nan_to_num() function. The clip() method makes it possible to cap values at a specific minimum and/or maximum threshold.

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

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

Our last strategy is imputation. When we replace a missing value with a new value derived from the data, using summary statistics or data from other observations.

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

Notice from the changes to the data for January 1st and 4th that the median maximum and minimum temperatures were 14.4°C and 5.6°C, respectively.
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

If we want to run the same calculation on all the columns, we should use the apply() method instead of assign(), since it saves us the redundancy of having to write the same calculation for each of the columns.

```
>>> df_deduped.apply(lambda x:
...     # Rolling 7-day median
...     # we set min_periods (# of p
...     # calculation) to 0 so we al
...     x.fillna(x.rolling(7, min_pe
... ).head(10)
```

	PRCP	SNOW	SNWD	TMAX	TMIN	TOBS	WESF	inclement_weather
date								
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

Another way of imputing missing data is to have pandas calculate what the values should be with the interpolate() method.

By default, it will perform linear interpolation, making the assumption that all the rows are evenly spaced. Our data is daily data, although some days are missing, so it is just a matter of reindexing first. 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

Different strategies for interpolation can be specified via the method argument; check out the interpolate() method documentation to view the available options.

رواد مصر الرقمية

وزارة الاتصالات وتكنولوجيا المعلومات

Q&A

Questions and answers

Thanks!