Section 2: Data Wrangling

To prepare our data for analysis, we need to perform data wrangling. In this section, we will learn how to clean and reformat data (e.g. renaming columns, fixing data type mismatches), restructure/reshape it, and enrich it (e.g. discretizing columns, calculating aggregations, combining data sources).

Data cleaning

In this section, we will take a look at creating, renaming, and dropping columns; type conversion; and sorting – all of which make our analysis easier. We will be working with the 2019 Yellow Taxi Trip Data provided by NYC Open Data.

In [1]:

```

```

Out[1]:

	vendorid	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	rate
0	2	2019-10-23T16:39:42.000	2019-10-23T17:14:10.000	1	7.93	
1	1	2019-10-23T16:32:08.000	2019-10-23T16:45:26.000	1	2.00	
2	2	2019-10-23T16:08:44.000	2019-10-23T16:21:11.000	1	1.36	
3	2	2019-10-23T16:22:44.000	2019-10-23T16:43:26.000	1	1.00	
4	2	2019-10-23T16:45:11.000	2019-10-23T16:58:49.000	1	1.96	

Source: NYC Open Data collected via SODA.

Dropping columns

Let's start by dropping the ID columns and the `store_and_fwd_flag` column, which we won't be using.

In [2]:

```

```

Out[2]:

```
Index(['vendorid', 'ratecodeid', 'store_and_fwd_flag', 'pulocationid',
       'dolocationid'],
      dtype='object')
```

In [3]:

Out[3]:

	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance	payment_type
0	2019-10-23T16:39:42.000	2019-10-23T17:14:10.000	1	7.93	1
1	2019-10-23T16:32:08.000	2019-10-23T16:45:26.000	1	2.00	1
2	2019-10-23T16:08:44.000	2019-10-23T16:21:11.000	1	1.36	1
3	2019-10-23T16:22:44.000	2019-10-23T16:43:26.000	1	1.00	1
4	2019-10-23T16:45:11.000	2019-10-23T16:58:49.000	1	1.96	1

Tip: Another way to do this is to select the columns we want to keep: `taxis.loc[:,~mask]` .

Renaming columns

Next, let's rename the datetime columns:

In [4]:

Out[4]:

	pickup	dropoff	passenger_count	trip_distance	payment_type	fare_amou
0	2019-10-23T16:39:42.000	2019-10-23T17:14:10.000	1	7.93	1	29
1	2019-10-23T16:32:08.000	2019-10-23T16:45:26.000	1	2.00	1	10
2	2019-10-23T16:08:44.000	2019-10-23T16:21:11.000	1	1.36	1	9
3	2019-10-23T16:22:44.000	2019-10-23T16:43:26.000	1	1.00	1	13
4	2019-10-23T16:45:11.000	2019-10-23T16:58:49.000	1	1.96	1	10

Important: This operation was performed in-place – be careful with in-place operations.

Type conversion

Notice anything off with the data types?

In [5]:

Out[5]:
```
pickup                     object
dropoff                    object
passenger_count             int64
trip_distance             float64
payment_type                int64
fare_amount               float64
extra                     float64
mta_tax                   float64
tip_amount                float64
tolls_amount              float64
improvement_surcharge     float64
total_amount              float64
congestion_surcharge      float64
dtype: object
```

Both `pickup` and `dropoff` should be stored as datetimes. Let's fix this:

In [6]:

Out[6]:
```
pickup                   datetime64[ns]
dropoff                  datetime64[ns]
passenger_count                   int64
trip_distance                   float64
payment_type                      int64
fare_amount                     float64
extra                           float64
mta_tax                         float64
tip_amount                      float64
tolls_amount                    float64
improvement_surcharge           float64
total_amount                    float64
congestion_surcharge            float64
dtype: object
```

Tip: There are other ways to perform type conversion. For numeric values, we can use the `pd.to_numeric()` function, and we will see the `astype()` method, which is a more generic method, a little later.

Creating new columns

Let's calculate the following for each row:

1. elapsed time of the trip
2. the tip percentage
3. the total taxes, tolls, fees, and surcharges
4. the average speed of the taxi

In [7]:

Our new columns get added to the right:

In [8]:

Out[8]:

	pickup	dropoff	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_t
0	2019-10-23 16:39:42	2019-10-23 17:14:10	1	7.93	1	29.5	1.0	(
1	2019-10-23 16:32:08	2019-10-23 16:45:26	1	2.00	1	10.5	1.0	(

Some things to note:

- We used `lambda` functions to 1) avoid typing `taxis` repeatedly and 2) be able to access the `cost_before_tip` column in the same method that we create it.
- To create a single new column, we can also use `df['new_col'] = <values>` .

Sorting by values

We can use the `sort_values()` method to sort based on any number of columns:

In [9]:

Out[9]:

	pickup	dropoff	passenger_count	trip_distance	payment_type	fare_amount	extra	m
5997	2019-10-23 15:55:19	2019-10-23 16:08:25	6	1.58	2	10.0	1.0	
443	2019-10-23 15:56:59	2019-10-23 16:04:33	6	1.46	2	7.5	1.0	
8722	2019-10-23 15:57:33	2019-10-23 16:03:34	6	0.62	1	5.5	1.0	
4198	2019-10-23 15:57:38	2019-10-23 16:05:07	6	1.18	1	7.0	1.0	
8238	2019-10-23 15:58:31	2019-10-23 16:29:29	6	3.23	2	19.5	1.0	

To pick out the largest/smallest rows, use `nlargest()` / `nsmallest()` instead. Looking at the 3 trips with the longest elapsed time, we see some possible data integrity issues:

In [10]:

```
```

Out[10]:

	pickup	dropoff	passenger_count	trip_distance	payment_type	fare_amount	extra	m
7576	2019-10-23 16:52:51	2019-10-24 16:51:44	1	3.75	1	17.5	1.0	
6902	2019-10-23 16:51:42	2019-10-24 16:50:22	1	11.19	2	39.5	1.0	
4975	2019-10-23 16:18:51	2019-10-24 16:17:30	1	0.70	2	7.0	1.0	

Working with the index

So far, we haven't really worked with the index because it's just been a row number; however, we can change the values we have in the index to access additional features of the `pandas` library.

Setting and sorting the index

Currently, we have a RangeIndex, but we can switch to a DatetimeIndex by specifying a datetime column when calling `set_index()` :

In [11]:

```

```

Out[11]:

pickup	dropoff	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_tax
2019-10-23 16:39:42	2019-10-23 17:14:10	1	7.93	1	29.5	1.0	0.5
2019-10-23 16:32:08	2019-10-23 16:45:26	1	2.00	1	10.5	1.0	0.5
2019-10-23 16:08:44	2019-10-23 16:21:11	1	1.36	1	9.5	1.0	0.5

Since we have a sample of the full dataset, let's sort the index to order by pickup time:

In [12]:

```

```

Tip: `taxis.sort_index(axis=1)` will sort the columns by name. The `axis` parameter is present throughout the `pandas` library: `axis=0` targets rows and `axis=1` targets columns.

We can now select ranges from our data based on the datetime the same way we did with row numbers:

In [13]:

```

```

Out[13]:

pickup	dropoff	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_ta
2019-10-23 07:48:58	2019-10-23 07:52:09	1	0.67	2	4.5	1.0	0.
2019-10-23 08:02:09	2019-10-24 07:42:32	1	8.38	1	32.0	1.0	0.
2019-10-23 08:18:47	2019-10-23 08:36:05	1	2.39	2	12.5	1.0	0.

When not specifying a range, we use `loc[]` :

In [14]:

```
```

Out[14]:

pickup	dropoff	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_ta
2019-10-23 08:02:09	2019-10-24 07:42:32	1	8.38	1	32.0	1.0	0.
2019-10-23 08:18:47	2019-10-23 08:36:05	1	2.39	2	12.5	1.0	0.

Resetting the index

We will be working with time series later this section, but sometimes we want to reset our index to row numbers and restore the columns. We can make `pickup` a column again with the `reset_index()` method:

In [15]:

```
```

Out[15]:

	pickup	dropoff	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_
0	2019-10-23 07:05:34	2019-10-23 08:03:16	3	14.68	1	50.0	1.0	
1	2019-10-23 07:48:58	2019-10-23 07:52:09	1	0.67	2	4.5	1.0	
2	2019-10-23 08:02:09	2019-10-24 07:42:32	1	8.38	1	32.0	1.0	
3	2019-10-23 08:18:47	2019-10-23 08:36:05	1	2.39	2	12.5	1.0	
4	2019-10-23 09:27:16	2019-10-23 09:33:13	2	1.11	2	6.0	1.0	

Reshaping data

The taxi dataset we have be working with is in a format conducive to an analysis. This isn't always the case. Let's now take a look at the TSA traveler throughput data, which compares 2021 throughput to the same day in 2020 and 2019:

In [16]:

```

```

Out[16]:

	Date	2021 Traveler Throughput	2020 Traveler Throughput	2019 Traveler Throughput
0	2021-05-14	1716561.0	250467	2664549
1	2021-05-13	1743515.0	234928	2611324
2	2021-05-12	1424664.0	176667	2343675
3	2021-05-11	1315493.0	163205	2191387
4	2021-05-10	1657722.0	215645	2512315

Source: TSA.gov

First, we will lowercase the column names and take the first word (e.g. `2021` for `2021 Traveler Throughput`) to make this easier to work with:

In [17]:

Out[17]:

	date	2021	2020	2019
0	2021-05-14	1716561.0	250467	2664549
1	2021-05-13	1743515.0	234928	2611324
2	2021-05-12	1424664.0	176667	2343675
3	2021-05-11	1315493.0	163205	2191387
4	2021-05-10	1657722.0	215645	2512315

Now, we can work on reshaping it.

Melting

Melting helps convert our data into long format. Now, we have all the traveler throughput numbers in a single column:

In [18]:

Out[18]:

	date	year	travelers
974	2020-09-12	2019	1879822.0
435	2021-03-05	2020	2198517.0
1029	2020-07-19	2019	2727355.0
680	2020-07-03	2020	718988.0
867	2020-12-28	2019	2500396.0

To convert this into a time series of traveler throughput, we need to replace the year in the `date` column with the one in the `year` column. Otherwise, we are marking prior years' numbers with the wrong year.

In [19]:

Out[19]:

	date	year	travelers
974	2019-09-12	2019	1879822.0
435	2020-03-05	2020	2198517.0
1029	2019-07-19	2019	2727355.0
680	2020-07-03	2020	718988.0
867	2019-12-28	2019	2500396.0

This leaves us with some null values (the dates that haven't yet occurred):

In [20]:

```
```

Out[20]:

	date	year	travelers
136	2021-12-29	2021	NaN
135	2021-12-30	2021	NaN
134	2021-12-31	2021	NaN

These can be dropped with the `dropna()` method:

In [21]:

```
```

Out[21]:

	date	year	travelers
2	2021-05-12	2021	1424664.0
1	2021-05-13	2021	1743515.0
0	2021-05-14	2021	1716561.0

Pivoting

Using the melted data, we can pivot the data to compare TSA traveler throughput on specific days across years:

In [22]:

```
```

Out[22]:

day_in_march year	1	2	3	4	5	6	7
2019	2257920.0	1979558.0	2143619.0	2402692.0	2543689.0	2156262.0	2485430.0
2020	2089641.0	1736393.0	1877401.0	2130015.0	2198517.0	1844811.0	2119867.0
2021	1049692.0	744812.0	826924.0	1107534.0	1168734.0	992406.0	1278557.0

Important: We aren't covering the `unstack()` and `stack()` methods, which are additional ways to pivot and melt, respectively. These come in handy when we have a multi-level index (e.g. if we ran `set_index()` with more than one column).

Transposing

The `T` attribute provides a quick way to flip rows and columns.

In [23]:

```

```

Out[23]:

year day_in_march	2019	2020	2021
1	2257920.0	2089641.0	1049692.0
2	1979558.0	1736393.0	744812.0
3	2143619.0	1877401.0	826924.0
4	2402692.0	2130015.0	1107534.0
5	2543689.0	2198517.0	1168734.0
6	2156262.0	1844811.0	992406.0
7	2485430.0	2119867.0	1278557.0
8	2378673.0	1909363.0	1119303.0
9	2122898.0	1617220.0	825745.0
10	2187298.0	1702686.0	974221.0

Merging

We typically observe changes in air travel around the holidays, so adding information about the dates in the TSA dataset provides more context. The `holidays.csv` file contains a few major holidays in the United States:

In [24]:

Out[24]:

	holiday
date	
2019-01-01	New Year's Day
2019-05-27	Memorial Day
2019-07-04	July 4th
2019-09-02	Labor Day
2019-11-28	Thanksgiving

Merging the holidays with the TSA traveler throughput data will provide more context for our analysis:

In [25]:

Out[25]:

	date	year	travelers	holiday
863	2019-01-01	2019	2126398.0	New Year's Day
862	2019-01-02	2019	2345103.0	NaN
861	2019-01-03	2019	2202111.0	NaN
860	2019-01-04	2019	2150571.0	NaN
859	2019-01-05	2019	1975947.0	NaN

Tip: There are many parameters for this method so be sure to check out the [documentation](). To append rows, take a look at `append()` and `pd.concat()`.

We can take this a step further by marking a few days before and after each holiday as part of the holiday. This would make it easier to compare holiday travel across years and look for any uptick in travel around the holidays:

In [26]:

Out[26]:

	date	year	travelers	holiday
899	2019-11-26	2019	1591158.0	Thanksgiving
898	2019-11-27	2019	1968137.0	Thanksgiving
897	2019-11-28	2019	2648268.0	Thanksgiving
896	2019-11-29	2019	2882915.0	Thanksgiving
873	2019-12-22	2019	1981433.0	Christmas Eve
872	2019-12-23	2019	1937235.0	Christmas Eve
871	2019-12-24	2019	2552194.0	Christmas Eve
870	2019-12-25	2019	2582580.0	Christmas Day
869	2019-12-26	2019	2470786.0	Christmas Day

Tip: Check out the documentation for the full list of functionality available with the `fillna()` method.

Aggregations and grouping

After reshaping and cleaning our data, we can perform aggregations to summarize it in a variety of ways. In this section, we will explore using pivot tables, crosstabs, and group by operations to aggregate the data.

Pivot tables

We can build a pivot table to compare holiday travel across the years in our dataset:

In [27]:

```
```

Out[27]:

holiday	Christmas Day	Christmas Eve	July 4th	Labor Day	Memorial Day	New Year's Day	New Year's Eve	Than
year								
2019	5053366.0	6470862.0	9414228.0	8314811.0	9720691.0	4471501.0	6535464.0	90
2020	1745242.0	3029810.0	2682541.0	2993653.0	1126253.0	4490388.0	3057449.0	33
2021	NaN	NaN	NaN	NaN	NaN	1998871.0	NaN	

We can use the `pct_change()` method on this result to see which holiday travel periods saw the biggest change in travel:

In [28]:

Out[28]:

holiday year	Christmas Day	Christmas Eve	July 4th	Labor Day	Memorial Day	New Year's Day	New Year's Eve	Thanksg
2019	NaN	NaN	NaN	NaN	NaN	NaN	NaN	
2020	-0.654638	-0.531776	-0.715055	-0.639961	-0.884139	0.004224	-0.532176	-0.62
2021	0.000000	0.000000	0.000000	0.000000	0.000000	-0.554856	0.000000	0.00

Let's make one last pivot table with column and row subtotals along with some formatting improvements. First, we set a display option for all floats:

In [29]:

Next, we group together Christmas Eve and Christmas Day, likewise for New Year's Eve and New Year's Day, and create the pivot table:

In [30]:

Out[30]:

holiday year	Christmas	July 4th	Labor Day	Memorial Day	New Year's	Thanksgiving	Total
2019	11,524,228	9,414,228	8,314,811	9,720,691	11,006,965	9,090,478	59,071,401
2020	4,775,052	2,682,541	2,993,653	1,126,253	7,547,837	3,364,358	22,489,694
2021	NaN	NaN	NaN	NaN	1,998,871	NaN	1,998,871
Total	16,299,280	12,096,769	11,308,464	10,846,944	20,553,673	12,454,836	83,559,966

Before moving on, let's reset the display option:

In [31]:

Tip: Read more about options in the documentation here.

Crosstabs

The `pd.crosstab()` function provides an easy way to create a frequency table:

In [32]:

Out[32]:

year	2019	2020	2021
travel_volume			
low	0	277	54
medium	42	44	80
high	323	44	0

Tip: The `pd.crosstab()` function supports other aggregations provided you pass in the data to aggregate as `values` and specify the aggregation with `aggfunc`. You can also add subtotals and normalize the data. See the documentation for more information.

Group by operations

Rather than perform aggregations, like `mean()` or `describe()`, on the full dataset at once, we can perform these calculations per group by first calling `groupby()`:

In [33]:

Out[33]:

								tr
	count	mean	std	min	25%	50%	75%	
year								
2019	365.0	2.309482e+06	285061.490784	1534386.0	2091116.0	2358007.0	2538384.00	288
2020	365.0	8.818674e+05	639775.194297	87534.0	507129.0	718310.0	983745.00	250
2021	134.0	1.112632e+06	338040.673782	468933.0	807156.0	1117391.0	1409377.75	174

Groups can also be used to perform separate calculations per subset of the data. For example, we can find the highest-volume travel day per year using `rank()`:

In [34]:

Out[34]:

	date	year	travelers	holiday	travel_volume_rank
896	2019-11-29	2019	2882915.0	Thanksgiving	1.0
456	2020-02-12	2020	2507588.0	NaN	1.0
1	2021-05-13	2021	1743515.0	NaN	1.0

The previous two examples called a single method on the grouped data, but using the `agg()` method we can specify any number of them:

In [35]:

Out[35]:

	travelers		holiday_travelers		non_holiday_trav	
	mean	std	mean	std	mean	
year						
2019	2.309482e+06	285061.490784	2.271977e+06	303021.675751	2.312359e+06	283906.22
2020	8.818674e+05	639775.194297	8.649882e+05	489938.240989	8.831619e+05	650399.77
2021	1.112632e+06	338040.673782	9.994355e+05	273573.249680	1.114347e+06	339479.29

In addition, we can specify which aggregations to perform on each column:

In [36]:

Out[36]:

	holiday_travelers		holiday	
	mean	std	nunique	count
year				
2019	2.271977e+06	303021.675751	8	26
2020	8.649882e+05	489938.240989	8	26
2021	9.994355e+05	273573.249680	1	2

We are only scratching the surface; some additional functionalities to be aware of include the following:

- We can group by multiple columns – this creates a hierarchical index.
- Groups can be excluded from calculations with the `filter()` method.
- We can group on content in the index using the `level` or `name` parameters e.g. `groupby(level=0)` or `groupby(name='year')`.
- We can group by date ranges if we use a `pd.Grouper()` object.

Be sure to check out the documentation for more details.

Time series

When working with time series data, `pandas` provides us with additional functionality to not just compare the observations in our dataset, but to use their relationship in time to analyze the data. In this section, we will see a few such operations for selecting date/time ranges, calculating changes over time, performing window calculations, and resampling the data to different date/time intervals.

Selecting based on date and time

Let's switch back to the `taxis` dataset, which has timestamps of pickups and dropoffs. First, we will set the `dropoff` column as the index and sort the data:

In [37]:

```

```

We saw earlier that we can slice on the datetimes:

In [38]:

```

```

Out[38]:

	pickup	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_tax
dropoff							
2019-10-24 12:30:08	2019-10-23 13:25:42	4	0.76	2	5.0	1.0	0.5
2019-10-24 12:42:01	2019-10-23 13:34:03	2	1.58	1	7.5	1.0	0.5

We can also represent this range with shorthand. Note that we must use `loc[]` here:

In [39]:

```

```

Out[39]:

dropoff	pickup	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_tax
2019-10-24 12:30:08	2019-10-23 13:25:42	4	0.76	2	5.0	1.0	0.5
2019-10-24 12:42:01	2019-10-23 13:34:03	2	1.58	1	7.5	1.0	0.5

However, if we want to look at this time range across days, we need another strategy.

We can pull out the dropoffs that happened between a certain time range on *any* day with the `between_time()` method:

In [40]:

Out[40]:

dropoff	pickup	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_tax
2019-10-23 12:53:49	2019-10-23 12:35:27	5	2.49	1	13.5	1.0	0.5
2019-10-24 12:30:08	2019-10-23 13:25:42	4	0.76	2	5.0	1.0	0.5
2019-10-24 12:42:01	2019-10-23 13:34:03	2	1.58	1	7.5	1.0	0.5

Tip: The `at_time()` method can be used to extract all entries at a given time (e.g. 12:35:27).

Finally, `head()` and `tail()` limit us to a number of rows, but we may be interested in rows within the first/last 2 hours (or any other time interval) of the data, in which case, we should use `first()` / `last()` :

In [41]:

Out[41]:

dropoff	pickup	passenger_count	trip_distance	payment_type	fare_amount	extra	mta_tax
2019-10-23 07:52:09	2019-10-23 07:48:58	1	0.67	2	4.5	1.0	0.5
2019-10-23 08:03:16	2019-10-23 07:05:34	3	14.68	1	50.0	1.0	0.5
2019-10-23 08:36:05	2019-10-23 08:18:47	1	2.39	2	12.5	1.0	0.5
2019-10-23 09:33:13	2019-10-23 09:27:16	2	1.11	2	6.0	1.0	0.5
2019-10-23 09:49:31	2019-10-23 09:47:25	2	0.47	2	52.0	4.5	0.5

For the rest of this section, we will be working with the TSA traveler throughput data. Let's start by setting the index to the `date` column:

In [42]:

```
```

Calculating change over time

In [43]:

```
```

Out[43]:

date	year	travelers	holiday	one_day_change	seven_day_change
2020-01-01	2020	2311732.0	New Year's Day	NaN	NaN
2020-01-02	2020	2178656.0	New Year's Day	-133076.0	NaN
2020-01-03	2020	2422272.0	NaN	243616.0	NaN
2020-01-04	2020	2210542.0	NaN	-211730.0	NaN
2020-01-05	2020	1806480.0	NaN	-404062.0	NaN
2020-01-06	2020	1815040.0	NaN	8560.0	NaN
2020-01-07	2020	2034472.0	NaN	219432.0	NaN
2020-01-08	2020	2072543.0	NaN	38071.0	-239189.0
2020-01-09	2020	1687974.0	NaN	-384569.0	-490682.0
2020-01-10	2020	2183734.0	NaN	495760.0	-238538.0

Tip: To perform operations other than subtraction, take a look at the `shift()` method. It also makes it possible to perform operations across columns.

Resampling

We can use resampling to aggregate time series data to a new frequency:

In [44]:

```
tsa_melted_holiday_travel['2019':'2021-Q1'].resample('Q').agg(['sum', 'mean',
```

Out[44]:

	travelers		
	sum	mean	std
date			
2019-03-31	189281658.0	2.103130e+06	282239.618354
2019-06-30	221756667.0	2.436886e+06	212600.697665
2019-09-30	220819236.0	2.400209e+06	260140.242892
2019-12-31	211103512.0	2.294603e+06	260510.040655
2020-03-31	155354148.0	1.726157e+06	685094.277420
2020-06-30	25049083.0	2.752646e+05	170127.402046
2020-09-30	63937115.0	6.949686e+05	103864.705739
2020-12-31	77541248.0	8.428397e+05	170245.484185
2021-03-31	86094635.0	9.566071e+05	280399.809061

Window calculations

Window calculations are similar to group by calculations except the group over which the calculation is performed isn't static – it can move or expand. Pandas provides functionality for constructing a variety of windows, including moving/rolling windows, expanding windows (e.g. cumulative sum or mean up to the current date in a time series), and exponentially weighted moving windows (to weight closer observations higher than further ones). We will only look at rolling and expanding calculations here.

Performing a window calculation is very similar to a group by calculation: we first define the window, and then we specify the aggregation:

In [45]:

```

```

Out[45]:

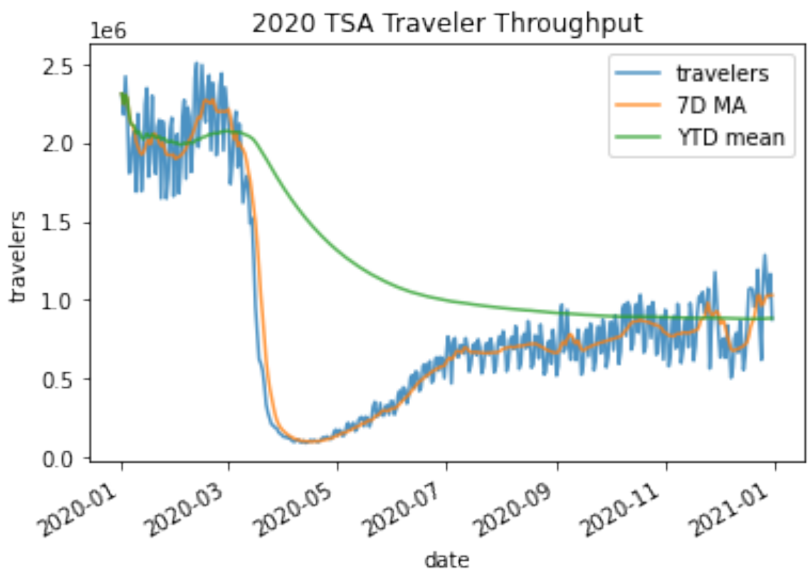
	year	travelers	holiday	7D MA	YTD mean
date					
2020-01-01	2020	2311732.0	New Year's Day	2.311732e+06	2.311732e+06
2020-01-02	2020	2178656.0	New Year's Day	2.245194e+06	2.245194e+06
2020-01-03	2020	2422272.0	NaN	2.304220e+06	2.304220e+06
2020-01-04	2020	2210542.0	NaN	2.280800e+06	2.280800e+06
2020-01-05	2020	1806480.0	NaN	2.185936e+06	2.185936e+06
2020-01-06	2020	1815040.0	NaN	2.124120e+06	2.124120e+06
2020-01-07	2020	2034472.0	NaN	2.111313e+06	2.111313e+06
2020-01-08	2020	2072543.0	NaN	2.077144e+06	2.106467e+06
2020-01-09	2020	1687974.0	NaN	2.007046e+06	2.059968e+06
2020-01-10	2020	2183734.0	NaN	1.972969e+06	2.072344e+06

To understand what's happening, it's best to visualize the original data and the result, so here's a sneak peek of plotting with `pandas` :

In [46]:

Out[46]:

```
<AxesSubplot:title={'center':'2020 TSA Traveler Throughput'}, xlabel='date', y
label='travelers'>
```



Other types of windows:

- exponentially weighted moving: use the `ewm()` method
- custom: create a subclass of `pandas.api.indexers.BaseIndexer` or use a pre-built one in `pandas.api.indexers`

Up Next: Data Visualization

Let's take a 25-minute break for some exercises to check your understanding:

1. Read in the meteorite data from the `Meteorite_Landings.csv` file.
2. Rename the `mass (g)` column to `mass`, and drop all the latitude and longitude columns.
3. Update the `year` column to only contain the year, and create a new column indicating if the year is unknown. Hint: Use `year.str.slice()` to grab a substring.
4. There's a data entry error in the `year` column. Can you find it? (Don't spend too much time on this.)
5. Compare summary statistics of the `mass` column for the meteorites that were found versus observed falling.
6. Create a pivot table that shows both the number of meteorites and the 95th percentile of meteorite mass for those that were found versus observed falling per year from 1990 to 2000 (inclusive).
7. Using the `taxis` data from earlier this section, resample the data to an hourly frequency based on the dropoff time. Calculate the total `trip_distance`, `fare_amount`, `tolls_amount`, and `tip_amount`, then find the 5 hours with the most tips.

Exercises

1. Read in the meteorite data from the `Meteorite_Landings.csv` file:

In [ ]:
```

```

2. Rename the `mass (g)` column to `mass`, and drop all the latitude and longitude columns.

In [ ]:
```

```

3. Update the `year` column to only contain the year, and create a new column indicating if the year is unknown. Hint: Use `year.str.slice()` to grab a substring.

In [ ]:

4. There's a data entry error in the `year` column. Can you find it?

In [ ]:

Oops! This meteorite actually was found in 2010 (more information here).

5. Compare summary statistics of the `mass` column for the meteorites that were found versus observed falling.

In [ ]:

6. Create a pivot table that shows both the number of meteorites and the 95th percentile of meteorite mass for those that were found versus observed falling per year from 1990 to 2000 (inclusive).

In [ ]:

7. Using the `taxis` data from earlier this section, resample the data to an hourly frequency based on the dropoff time. Calculate the total `trip_distance`, `fare_amount`, `tolls_amount`, and `tip_amount`, then find the 5 hours with the most tips.

In [ ]: