Doing More with Data: `pandas`

Introduction to Python

Data Sciences Institute, University of Toronto
Instructor: A Mahfouz | TA: Kaylie Lau
July 2022

Contents:

Setup

Anaconda

Install `pandas` if you haven't already. If you're not sure, go through the steps below.

1. Open Anaconda Prompt. You may have to do this in admin mode.
2. Type `conda install pandas` and press enter to run.
3. Press `enter` to answer Y to any prompts.

We will also need `openpyxl`, which can be installed by running the command `conda install openpyxl` in Anaconda Prompt.

Google Colab

If you're using Colab, you're all set! `pandas` and `openpyxl` are already installed.

Data

This module uses four datasets: **bike thefts**, **TTC subway delays and subway delay reason codes**, and **neighbourhood profiles**. All four are available in the course repo, and originally come from Toronto Open Data.

pandas

What is `pandas`?

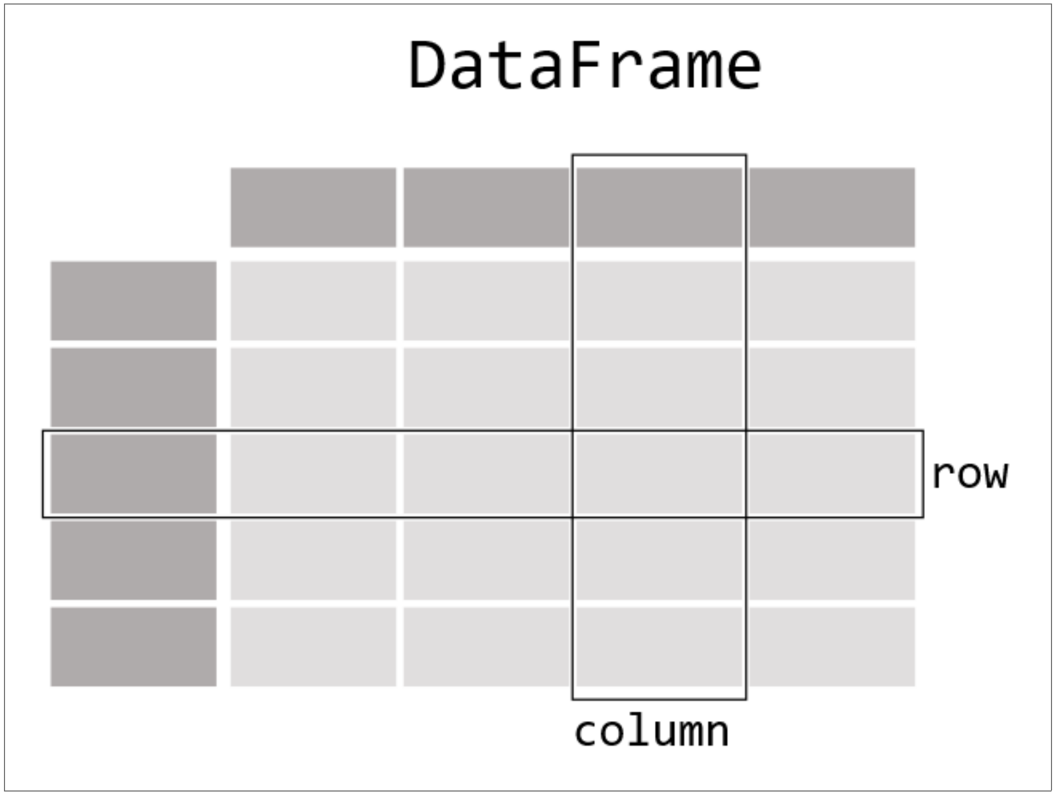
`pandas` is a package for data analysis and manipulation. (The name is a reference to panel data, not the animal.) It gives us data frames, which represent data in a table of columns and rows, and functions to manipulate and plot them. `pandas` also provides a slew of functions for reading and writing data to a variety of sources, including files, SQL databases, and compressed binary formats.

In [1]:
```
import numpy as np
# pd is the conventional alias for pandas
import pandas as pd

# display all columns
pd.set_option("display.max_columns", None)
```

DataFrames

Columns are labeled with their names. Rows also have a label, or *index*. If row labels are not specified, `pandas` uses numbers as the default. Each column is a *Series*, or one-dimensional array, where values share a data type. Unlike `numpy` arrays, DataFrames can have columns of different data types. However, like arrays and lists, **DataFrames are mutable** -- this means that if more than one variable refers to the same DataFrame, updating one updates them all!


DataFrame
row
column

Getting data

We can create a DataFrame manually with `DataFrame()` constructor. If a dictionary is passed to `DataFrame()`, the keys become column names, and the values become the rows. Calling just `DataFrame()` creates an empty DataFrame to which data can be added later.

```
In [2]: trees = pd.DataFrame({
            'name': ['sugar maple', 'black oak', 'white ash', 'douglas fir'],
     'avg_lifespan': [300, 100, 260, 450],
     'quantity': [53, 207, 178, 93]
})
trees
```

```
Out[2]:
```

	name	avg_lifespan	quantity
0	sugar maple	300	53
1	black oak	100	207
2	white ash	260	178
3	douglas fir	450	93

We can create an individual column with `Series()`. The `name` argument corresponds to a column name.

```
In [3]: tree_types = pd.Series(['deciduous', 'deciduous', 'deciduous', 'evergreen'],
                               name='foliage')
tree_types
```

```
Out[3]:

0    deciduous
1    deciduous
2    deciduous
3    evergreen
Name: foliage, dtype: object
```

Data from csv

Of course, we're more likely to load data into a DataFrame than to create DataFrames manually. `pandas` has read functions for different file formats. To read data from a csv or other delimited file, we use `pd.read_csv()`, then pass in the local file path or the URL of the csv to read. `pandas` will infer the data type of each column based on the values in the first chunk of the file loaded.

```
In [4]: thefts = pd.read_csv('../data/bicycle-thefts - 4326.csv')
```

Profiling and initial data cleaning

We got our data, but now we need to understand what's in it. We can start to understand the DataFrame by checking out its `dtypes` and `shape` attributes, which give column data types and row by column dimensions, respectively. Note that `object` is `pandas`' way of saying values are represented as string data.

In [5]:
```
thefts.shape
```

Out[5]:

```
(25569, 33)
```

In [6]:
```
thefts.dtypes
```

Out[6]:

```
_id                         int64
OBJECTID                    int64
event_unique_id            object
Primary_Offence            object
Occurrence_Date            object
Occurrence_Year             int64
Occurrence_Month           object
Occurrence_DayOfWeek       object
Occurrence_DayOfMonth       int64
Occurrence_DayOfYear        int64
Occurrence_Hour             int64
Report_Date                object
Report_Year                 int64
Report_Month               object
Report_DayOfWeek           object
Report_DayOfMonth           int64
Report_DayOfYear            int64
Report_Hour                 int64
Division                   object
City                       object
Hood_ID                    object
NeighbourhoodName          object
Location_Type              object
Premises_Type              object
Bike_Make                  object
Bike_Model                 object
Bike_Type                  object
Bike_Speed                  int64
Bike_Colour                object
Cost_of_Bike              float64
Status                     object
ObjectId2                   int64
geometry                   object
dtype: object
```

`head()`s and `tail()`s

To check out the first few rows, we can call the DataFrame `head()` method. Similarly, we can see the last few rows with the `tail()` method. Five rows are shown by default, but we can change that by passing an integer as an argument.

```
In [7]: thefts.head()
```

Out[7]:

	_id	OBJECTID	event_unique_id	Primary_Offence	Occurrence_Date	Occurrence_Year	Occurrence_Mont
0	1	17744	GO-20179016397	THEFT UNDER	2017-10-03T00:00:00	2017	Octobe
1	2	17759	GO-20172033056	THEFT UNDER - BICYCLE	2017-11-08T00:00:00	2017	Novembe
2	3	17906	GO-20189030822	THEFT UNDER - BICYCLE	2018-09-14T00:00:00	2018	Septembe
3	4	17962	GO-2015804467	THEFT UNDER	2015-05-07T00:00:00	2015	May
4	5	17963	GO-20159002781	THEFT UNDER	2015-05-16T00:00:00	2015	May

_id	OBJECTID	event_unique_id	Primary_Offence	Occurrence_Date	Occurrence_Year	Occurrence_Month

In [8]:
```python
# last 3
thefts.tail(3)
```

Out[8]:

	_id	OBJECTID	event_unique_id	Primary_Offence	Occurrence_Date	Occurrence_Year	Occurre
25566	25567	11462	GO-20169005434	THEFT UNDER	2016-06-04T00:00:00	2016	
25567	25568	11695	GO-20161170896	THEFT UNDER	2016-07-04T00:00:00	2016	
25568	25569	11883	GO-20169007653	THEFT UNDER - BICYCLE	2016-07-22T00:00:00	2016	

Renaming columns

Most, but not all, of the bike theft columns follow the same naming convention. For convenience's sake, though, let's convert the column names to all lowercase. We can do this with the DataFrame `rename()` method. `rename()` accepts either a dictionary with current column names as the keys and new names as the values, or the name of a function to transform names. Let's write a function.

In [9]:
```
# notice that we do not add () to the function name
thefts = thefts.rename(columns=str.lower)
```

Let's also rename `cost_of_bike` so it follows the pattern of the other bike attribute columns.

```
In [10]: thefts = thefts.rename(columns={'cost_of_bike':'bike_cost'})

# view column names
print(list(thefts))
```

```
['_id', 'objectid', 'event_unique_id', 'primary_offence', 'occurrence_date', 'occurrence_year', '
occurrence_month', 'occurrence_dayofweek', 'occurrence_dayofmonth', 'occurrence_dayofyear', 'occu
rrence_hour', 'report_date', 'report_year', 'report_month', 'report_dayofweek', 'report_dayofmont
h', 'report_dayofyear', 'report_hour', 'division', 'city', 'hood_id', 'neighbourhoodname', 'locat
ion_type', 'premises_type', 'bike_make', 'bike_model', 'bike_type', 'bike_speed', 'bike_colour',
'bike_cost', 'status', 'objectid2', 'geometry']
```

Profiling columns

It can be useful to focus on a subset of columns, particularly to understand value sets. To select a single column in a DataFrame, we can supply the name of the column in square brackets, just like we did when accessing values in a dictionary. `pandas` will return the column as a Series. To get unique values, we can use the `unique()` Series method. If we want to count how many times each value appears, we can use the `value_counts()` method.

In [11]:
```
thefts['status']
```

Out[11]:

```
0           STOLEN
1        RECOVERED
2           STOLEN
3           STOLEN
4           STOLEN
           ...
25564       STOLEN
25565       STOLEN
25566       STOLEN
25567       STOLEN
25568       STOLEN
Name: status, Length: 25569, dtype: object
```

In [12]:
```
thefts['status'].unique()
```

Out[12]:

```
array(['STOLEN', 'RECOVERED', 'UNKNOWN'], dtype=object)
```

In [13]:
```
thefts['status'].value_counts()
```

Out[13]:

```
STOLEN       24807
UNKNOWN        454
RECOVERED      308
Name: status, dtype: int64
```

We can summarize numeric Series much like we did with `numpy` functions.

```
In [14]: thefts['bike_cost'].median()
```

```
Out[14]:

600.0
```

```
In [15]: thefts['bike_cost'].quantile(0.9)
```

```
Out[15]:

2000.0
```

`info()`

We can get an overview of the DataFrame by profiling it with the `info()` method. `info()` prints a lot of information about a DataFrame, including:

- the `shape` as the number of rows and columns
- column names and their `dtype`
- the number of non-null values in each column
- how big the DataFrame is in terms of memory usage

The bicycle theft data looks quite complete, though some records are missing bike descriptors like bike_make, bike_model, bike_colour, and bike_cost.
Most of the column dtypes make sense. We'll want to convert the dates to proper dates. We may also want to convert string columns with limited value sets, like status, to categorical data.

```
In [16]: thefts.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25569 entries, 0 to 25568
Data columns (total 33 columns):
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   _id                    25569 non-null  int64
 1   objectid               25569 non-null  int64
 2   event_unique_id        25569 non-null  object
 3   primary_offence        25569 non-null  object
 4   occurrence_date        25569 non-null  object
 5   occurrence_year        25569 non-null  int64
 6   occurrence_month       25569 non-null  object
 7   occurrence_dayofweek   25569 non-null  object
 8   occurrence_dayofmonth  25569 non-null  int64
 9   occurrence_dayofyear   25569 non-null  int64
 10  occurrence_hour        25569 non-null  int64
 11  report_date            25569 non-null  object
 12  report_year            25569 non-null  int64
 13  report_month           25569 non-null  object
 14  report_dayofweek       25569 non-null  object
 15  report_dayofmonth      25569 non-null  int64
 16  report_dayofyear       25569 non-null  int64
 17  report_hour            25569 non-null  int64
 18  division               25569 non-null  object
 19  city                   25569 non-null  object
 20  hood_id                25569 non-null  object
 21  neighbourhoodname      25569 non-null  object
 22  location_type          25569 non-null  object
 23  premises_type          25569 non-null  object
 24  bike_make              25448 non-null  object
 25  bike_model             15923 non-null  object
 26  bike_type              25569 non-null  object
 27  bike_speed             25569 non-null  int64
 28  bike_colour            23508 non-null  object
 29  bike_cost              23825 non-null  float64
 30  status                 25569 non-null  object
 31  objectid2              25569 non-null  int64
 32  geometry               25569 non-null  object
```

```
dtypes: float64(1), int64(12), object(20)
memory usage: 6.4+ MB
```

Changing data types

Before exploring the bike theft data further, let's fix the date and categorical columns. To convert a column to datetime, we use the `pd.to_datetime()` function, passing in the column to convert, and reassign the output back to the column we're converting. `pandas` knows how to convert the dates in the bike thefts data, but for less common formats, it is necessary to use the `format` keyword argument to specify how dates should be parsed. `format` strings use `strftime` codes. See **https://strftime.org/** for a cheat sheet.

In [17]:
```
thefts['occurrence_date'] = pd.to_datetime(thefts['occurrence_date'],
                                           format='%Y-%m-%d')
thefts['occurrence_date']
```

Out[17]:

```
0        2017-10-03
1        2017-11-08
2        2018-09-14
3        2015-05-07
4        2015-05-16
            ...
25564    2015-04-01
25565    2016-05-16
25566    2016-06-04
25567    2016-07-04
25568    2016-07-22
Name: occurrence_date, Length: 25569, dtype: datetime64[ns]
```

In [18]:
```python
# convert report_date without the format argument
thefts['report_date'] = pd.to_datetime(thefts['report_date'])
thefts['report_date']
```

Out[18]:

```
0        2017-10-03
1        2017-11-08
2        2018-09-17
3        2015-05-14
4        2015-05-16
            ...
25564    2015-04-01
25565    2016-05-16
25566    2016-06-07
25567    2016-07-04
25568    2016-07-23
Name: report_date, Length: 25569, dtype: datetime64[ns]
```

All other data type conversions can be done with the `astype()` method. If we were converting to a number, `pd.to_numeric()` provides an easy way to convert without having to pick a specific numeric data type.

```
In [19]: thefts['status'] = thefts['status'].astype('category')
         thefts['status']
```

Out[19]:

```
0           STOLEN
1        RECOVERED
2           STOLEN
3           STOLEN
4           STOLEN
           ...
25564       STOLEN
25565       STOLEN
25566       STOLEN
25567       STOLEN
25568       STOLEN
Name: status, Length: 25569, dtype: category
Categories (3, object): ['RECOVERED', 'STOLEN', 'UNKNOWN']
```

We can select and convert multiple columns at once by passing a list of columns in the square brackets., then using `.astype()`.

In [20]:
```
thefts[['location_type', 'premises_type']] = thefts[['location_type','premises_type']].astype('category')

# check data types
thefts[['location_type', 'premises_type']].dtypes
```

Out[20]:

```
location_type    category
premises_type    category
dtype: object
```

`describe()`

To get a sense of the values in a DataFrame, we can use the `describe()` method. `describe()` summarizes only numeric columns by default. Passing the `include='all'` argument will produce summary statistics for other columns as well.

In [21]:
```
thefts.describe(include='all',
                datetime_is_numeric=True)  # silence warning about upcoming pandas change
```

Out[21]:

	_id	objectid	event_unique_id	primary_offence	occurrence_date	occurre
count	25569.000000	25569.000000	25569	25569	25569	25569.0
unique	NaN	NaN	22771	66	NaN	
top	NaN	NaN	GO-20201550944	THEFT UNDER	NaN	
freq	NaN	NaN	14	11904	NaN	
mean	12785.000000	12909.173218	NaN	NaN	2017-09-04 03:39:28.321013504	2017.1
min	1.000000	1.000000	NaN	NaN	2009-09-01 00:00:00	2009.0
25%	6393.000000	6456.000000	NaN	NaN	2016-01-06 00:00:00	2016.0
50%	12785.000000	12918.000000	NaN	NaN	2017-09-05 00:00:00	2017.0
75%	19177.000000	19360.000000	NaN	NaN	2019-06-20 00:00:00	2019.0
max	25569.000000	25806.000000	NaN	NaN	2020-12-30 00:00:00	2020.0

	_id	objectid	event_unique_id	primary_offence	occurrence_date	occurre
std	7381.278853	7448.318562	NaN	NaN	NaN	1.9

Practice

1. Explore unique valuesets for some of the other columns in `thefts`. Identify at least one that may make sense as categorical variables and convert it.
2. Count values for `location_type`. Where did most reported thefts happen?
3. Find the 10th percentile of `bike_cost`.

Wrangling and Plotting

Combining datasets: concatenation

Just as `pandas` has `read_csv()` for flat files, there is a `read_excel()` function to load Excel files.
The TTC publishes subway delay data as a multi-sheet Excel workbook, with a month's worth of data per sheet. `read_excel()` loads just the first sheet in an Excel file by default. To load all sheets, pass in the keyword argument `sheet_name=None`. The result is a dictionary, where each key is the sheet name and each value is a DataFrame with the contents of the sheet.

In [23]:
```
delays_url = '../data/ttc-subway-delay-data-2021.xlsx'
delays = pd.read_excel(delays_url, sheet_name=None)
```

In [24]:
```
type(delays)
```

Out[24]:

```
dict
```

To combine them all, we create an empty DataFrame, then loop through the dictionary items and use `pd.concat()` to append data. `concat()` takes a list of DataFrames to combine. Since we did not specify an index, row labels are numbers: the first row of each sheet has an index of 0, and so on. To reset row labels so that they are sequential again, we set `ignore_index=True`.

In [25]:
```python
# create an empty DataFrame
all_delays = pd.DataFrame()

for sheet_name, values in delays.items():
    # print the number of rows
    print(f'Adding {values.shape[0]} rows from {sheet_name}')
    # add each sheet to all_delays
    all_delays = pd.concat([all_delays, values],
                           axis=0,  # concatenate rows
                           ignore_index=True)  # reset row labels

all_delays.shape
```

```
Adding 1216 rows from January21
Adding 1245 rows from Feb 21
Adding 1167 rows from March '21
Adding 1170 rows from April '21
Adding 1168 rows from May '21
Adding 1265 rows from June 21
Adding 1244 rows from July 21
Adding 1273 rows from August 21
Adding 1433 rows from Sept 21
Adding 1560 rows from Oct 21
Adding 1771 rows from Nov 21
Adding 1858 rows from December21
```

Out[25]:

```
(16370, 10)
```

In [26]:
```python
all_delays.head()
```

Out[26]:

	Date	Time	Day	Station	Code	Min Delay	Min Gap	Bound	Line	Vehicle
0	2021-01-01	00:33	Friday	BLOOR STATION	MUPAA	0	0	N	YU	6046
1	2021-01-01	00:39	Friday	SHERBOURNE STATION	EUCO	5	9	E	BD	5250
2	2021-01-01	01:07	Friday	KENNEDY BD STATION	EUCD	5	9	E	BD	5249
3	2021-01-01	01:41	Friday	ST CLAIR STATION	MUIS	0	0	NaN	YU	0
4	2021-01-01	02:04	Friday	SHEPPARD WEST STATION	MUIS	0	0	NaN	YU	0

Practice

1. The TTC delays data includes a reason code for the delay. Code definitions, however, are in a separate Excel file, `ttc-subway-delay-codes.xlsx`. This file has been modified slightly to make it easier to work with. Codes are split between two tabs. Load both to a DataFrame, `delay_reasons`.
2. Rename the columns in both `all_delays` and `delay_reasons` so that all letters are lowercase. **Challenge**: rename the columns to replace spaces with underscores as well as convert all letters to lowercase.
3. Explore `all_delays`. Do any columns have missing values? What was the median `Min Delay`? How many unique lines are in the data?

Combining datasets: merging

Ideally, the delays data would include code descriptions. We can get descriptions into `all_delays` by *merging* in `delay_reasons`. Merging is analagous to joining in SQL databases. To merge two DataFrames, we pass them as arguments to the `pd.merge()`. Then, we specify `how` to merge the two DataFrames and what column names to merge `on`.

Let's review the `all_delays` and `delay_reasons` DataFrames. `code` is equivalent to `rmenu_code`. If we pass in `all_delays` as the first DataFrame, then it will be the left frame, and `delay_reasons` the right one. We want to keep all the delay records, even if there isn't a matching code in `delay_reasons`, so we will perform a left join.

```
In [29]: all_delays.head(2)
```

Out[29]:

	date	time	day	station	code	min_delay	min_gap	bound	line	vehicle
0	2021-01-01	00:33	Friday	BLOOR STATION	MUPAA	0	0	N	YU	6046
1	2021-01-01	00:39	Friday	SHERBOURNE STATION	EUCO	5	9	E	BD	5250

```
In [30]: delay_reasons.head(2)
```

Out[30]:

	rmenu_code	code_description	sub_or_srt
0	EUAC	Air Conditioning	SUB
1	EUAL	Alternating Current	SUB

```
In [31]: delays_w_reasons = pd.merge(all_delays,
                                     delay_reasons,
                             how='left',
                             left_on='code',
                             right_on='rmenu_code')
         delays_w_reasons.head(3)
```

Out[31]:

	date	time	day	station	code	min_delay	min_gap	bound	line	vehicle	rmenu_co
0	2021-01-01	00:33	Friday	BLOOR STATION	MUPAA	0	0	N	YU	6046	MUPA
1	2021-01-01	00:39	Friday	SHERBOURNE STATION	EUCO	5	9	E	BD	5250	EUC
2	2021-01-01	01:07	Friday	KENNEDY BD STATION	EUCD	5	9	E	BD	5249	EUC

Practice

Try performing an `inner` join with `all_delays` and `delay_reasons`. Assign the result to a different variable name. Compare the shapes of `delays_w_reasons` and this new DataFrame. Did we get the same number of rows with the inner join?

`drop()`

The resulting DataFrame has both our join columns, which is redundant. We can drop one with the `drop()` DataFrame method, passing in the column name(s) we want to drop in the `columns` keyword argument.

```
In [32]: delays_w_reasons = delays_w_reasons.drop(columns='rmenu_code')
         delays_w_reasons.head(3)
```

Out[32]:

	date	time	day	station	code	min_delay	min_gap	bound	line	vehicle	code_de
0	2021-01-01	00:33	Friday	BLOOR STATION	MUPAA	0	0	N	YU	6046	Pas Ass Acti No
1	2021-01-01	00:39	Friday	SHERBOURNE STATION	EUCO	5	9	E	BD	5250	C
2	2021-01-01	01:07	Friday	KENNEDY BD STATION	EUCD	5	9	E	BD	5249	Consec Del Dela

Creating new columns

Adding a column to a DataFrame looks like adding a key-value pair to a dictionary. At its simplest, we can assign a single value to repeat down a column.

In [33]:
```
delays_w_reasons['year'] = 2021
delays_w_reasons['year'].unique()
```

Out[33]:

```
array([2021], dtype=int64)
```

We can also write an expression and store the resulting values in a new column.

In [34]:
```
delays_w_reasons['hour_delay'] = round(delays_w_reasons['min_delay'] / 60, 2)
delays_w_reasons[['min_delay', 'hour_delay']].head()
```

Out[34]:

	min_delay	hour_delay
0	0	0.00
1	5	0.08
2	5	0.08
3	0	0.00
4	0	0.00

It is also possible to extract parts of a datetime column with the `dt` accessor.

```
In [35]: delays_w_reasons['month'] = delays_w_reasons['date'].dt.month
         delays_w_reasons['month']
```

```
Out[35]:

0         1
1         1
2         1
3         1
4         1
         ..
16365    12
16366    12
16367    12
16368    12
16369    12
Name: month, Length: 16370, dtype: int64
```

Practice

- Create a new integer column, `hour`, that contains the hour in which a delay occurred.

  Hint: Use `pd.to_datetime()`, specifying that the format is hour:minutes.
- Drop the `year` column.

Filtering and selecting data

Let's take another look at the TTC subway delay data. There are only 4 subway lines in Toronto, but `describe()` reported 17 unique values.

In [37]:
```
delays_w_reasons['line'].unique()
```

Out[37]:

```
array(['YU', 'BD', 'SHP', 'SRT', 'YU/BD', nan, 'YONGE/UNIVERSITY/BLOOR',
       'YU / BD', 'YUS', '999', 'SHEP', '36 FINCH WEST', 'YUS & BD',
       'YU & BD LINES', '35 JANE', '52', '41 KEELE', 'YUS/BD'],
      dtype=object)
```

Looks like some of the line values should be updated (YU/BD variants) and others should be dropped (e.g., 36 FINCH WEST, NaNs). Luckily there don't seem to be too many affected records, though the NaNs are not shown.

In [38]:
```
delays_w_reasons['line'].value_counts()
```

Out[38]:

```
YU                         8880
BD                         5734
SHP                         657
SRT                         656
YU/BD                       346
YUS                          18
YU / BD                      17
YU & BD LINES                 1
41 KEELE                      1
52                            1
35 JANE                       1
999                           1
YUS & BD                      1
36 FINCH WEST                 1
SHEP                          1
YONGE/UNIVERSITY/BLOOR        1
YUS/BD                        1
Name: line, dtype: int64
```

`.loc[]` and `isna()`

To find the records with no line, we can use `.loc[]`, which lets us access rows and columns with either a boolean array or row/column labels.
In this case, the boolean array is the product of the `isna()` Series method.

In [39]:
```python
# access rows of data where line is NA
delays_w_reasons.loc[delays_w_reasons['line'].isna()]
```

Out[39]:

	date	time	day	station	code	min_delay	min_gap	bound	line	vehi
495	2021-01-13	15:22	Wednesday	FINCH WEST STATION	MUSAN	3	6	S	NaN	57
513	2021-01-13	22:08	Wednesday	EGLINTON WEST STATION	PUMEL	0	0	NaN	NaN	
1044	2021-01-27	22:00	Wednesday	YONGE-UNIVERSITY AND B	MUO	0	0	NaN	NaN	
1045	2021-01-27	23:00	Wednesday	FINCH STATION	MUO	0	0	NaN	NaN	
1362	2021-02-04	01:45	Thursday	LAWRENCE STATION	TUSC	0	0	S	NaN	55
1679	2021-02-11	01:12	Thursday	GREENWOOD CARHOUSE	MUIE	0	0	NaN	NaN	
2179	2021-02-22	08:27	Monday	BICHMOUNT DIVISION	MUIE	0	0	NaN	NaN	
2204	2021-02-22	22:33	Monday	BLOOR STATION	SUAP	4	9	N	NaN	60
2206	2021-02-22	23:36	Monday	EGLINTON	MUO	0	0	NaN	NaN	

	date	time	day	station	code	min_delay	min_gap	bound	line	vehi
				STATION						
3039	2021-03-17	05:15	Wednesday	INGLIS BUILDING	PUMEL	0	0	NaN	NaN	
3330	2021-03-24	19:13	Wednesday	INGLIS BUILDING	PUMEL	0	0	NaN	NaN	
3407	2021-03-26	09:03	Friday	WILSON YARD (SOUTH TAI	PUTO	0	0	NaN	NaN	
3557	2021-03-30	00:36	Tuesday	INGLIS BUILDING	PUMEL	0	0	NaN	NaN	
3944	2021-04-08	23:45	Thursday	DAVISVILLE YARD	MUIE	0	0	NaN	NaN	
4097	2021-04-13	10:57	Tuesday	SPADINA STATION	SUAE	0	0	NaN	NaN	
4119	2021-04-13	22:00	Tuesday	YONGE-UNIVERSITY AND B	MUO	0	0	NaN	NaN	
4336	2021-04-19	23:00	Monday	SHEPPARD WEST TO LAWRE	MUO	0	0	NaN	NaN	
4748	2021-04-29	22:00	Thursday	YONGE UNIVERSITY SPADI	MUO	0	0	NaN	NaN	

	date	time	day	station	code	min_delay	min_gap	bound	line	vehi
5312	2021-05-15	05:05	Saturday	SPADINA BD STATION	MUNCA	0	0	NaN	NaN	
5448	2021-05-18	20:15	Tuesday	VAUGHAN MC STATION	MUWR	0	0	NaN	NaN	
5484	2021-05-19	18:11	Wednesday	QUEEN'S QUAY STATION	PUMEL	0	0	NaN	NaN	
5642	2021-05-23	23:19	Sunday	ST ANDREW STATION	SUDP	0	0	NaN	NaN	
5685	2021-05-25	00:19	Tuesday	DUNDA WEST STATION	SUO	0	0	E	NaN	
6042	2021-06-02	22:28	Wednesday	WARDEN STATION	MUIRS	0	0	NaN	NaN	
6046	2021-06-02	00:56	Wednesday	BAY STATION	SUO	0	0	NaN	NaN	
6540	2021-06-14	22:43	Monday	YONGE BD STATION	MUIS	0	0	NaN	NaN	
6560	2021-06-15	07:15	Tuesday	SUBWAY OPERATIONS BUIL	PUMEL	0	0	NaN	NaN	

	date	time	day	station	code	min_delay	min_gap	bound	line	vehi
7137	2021-06-28	01:03	Monday	COXWELL STATION	MUNCA	0	0	NaN	NaN	
7766	2021-07-14	03:51	Wednesday	TRANSIT CONTROL CENTRE	PUSO	0	0	NaN	NaN	
8889	2021-08-11	07:46	Wednesday	TRANSIT CONTROL	MUIE	0	0	NaN	NaN	
9628	2021-08-29	15:49	Sunday	YORKDALE STATION	SUPOL	0	0	NaN	NaN	
9629	2021-08-29	16:13	Sunday	YORK MILLS STATION	MUO	0	0	NaN	NaN	
9780	2021-09-01	20:35	Wednesday	MAIN STREET AND UNION	MUO	0	0	NaN	NaN	
9789	2021-09-01	22:14	Wednesday	UNION AND KENNEDY STAT	MUO	0	0	NaN	NaN	
10336	2021-09-13	17:20	Monday	MCBRIEN BUILDING	SUO	0	0	NaN	NaN	
10951	2021-09-26	15:50	Sunday	WILSON STATION	PUOPO	0	0	N	NaN	54
11223	2021-10-01	00:33	Friday	WELLESLEY STATION	SUDP	0	0	NaN	NaN	
12533	2021-10-28	14:18	Thursday	VICTORIA	MUIS	0	0	NaN	NaN	

	date	time	day	station	code	min_delay	min_gap	bound	line	vehi
				PARK STATION						
12826	2021-11-02	12:22	Tuesday	GREENWOOD SHOP	MUIE	0	0	NaN	NaN	
13007	2021-11-05	08:59	Friday	BLOOR STATION	MUIRS	0	0	S	NaN	
13080	2021-11-06	18:41	Saturday	KENNEDY BD STATION	MUO	0	0	NaN	NaN	
13273	2021-11-10	16:25	Wednesday	SUMMERHILL STATION	TUS	3	6	N	NaN	65
13402	2021-11-12	20:42	Friday	CLOSURES BUILDING	MUIE	0	0	NaN	NaN	
13410	2021-11-12	00:02	Friday	TRANSIT CONTROL	MUO	0	0	NaN	NaN	
14177	2021-11-25	21:14	Thursday	WILSON CARHOUSE	MUIE	0	0	NaN	NaN	
14371	2021-11-29	05:10	Monday	GO PROTOCOL	MUO	0	0	NaN	NaN	

	date	time	day	station	code	min_delay	min_gap	bound	line	vehi
14935	2021-12-08	06:00	Wednesday	TORONTO TRANSIT COMMIS	MUO	0	0	NaN	NaN	
14952	2021-12-08	13:58	Wednesday	KIPLING STATION	MUIS	0	0	NaN	NaN	
14967	2021-12-08	17:14	Wednesday	QUEEN'S PARK STATION	MUO	0	0	NaN	NaN	
15581	2021-12-19	00:42	Sunday	WILSON CARHOUSE	MUO	0	0	NaN	NaN	
15623	2021-12-20	16:23	Monday	YONGE-SHEPPARD (LINE 4	MUIRS	0	0	NaN	NaN	
16332	2021-12-31	14:34	Friday	GO PROTOCOL	MUO	0	0	NaN	NaN	

`.loc[]` also lets us access data by label, with row conditions first and column conditions second.

```
In [40]: (delays_w_reasons.loc[delays_w_reasons['line'].isna(),  # filter rows
                               ['time', 'station', 'line']]  # get columns
                   .head()) # first 5 lines to save space
```

Out[40]:

	time	station	line
495	15:22	FINCH WEST STATION	NaN
513	22:08	EGLINTON WEST STATION	NaN
1044	22:00	YONGE-UNIVERSITY AND B	NaN
1045	23:00	FINCH STATION	NaN
1362	01:45	LAWRENCE STATION	NaN

`query()`

Alternatively, we can use the DataFrame `query()` method, which takes a filter condition as a string, and returns a DataFrame of records that met the condition. `query()` is slower than `loc[]`, but it can be easier to read.

In [41]:
```python
# slower than .loc, but can be easier to read
delays_w_reasons.query('line.isna()').head()
```

Out[41]:

	date	time	day	station	code	min_delay	min_gap	bound	line	vehicle
495	2021-01-13	15:22	Wednesday	FINCH WEST STATION	MUSAN	3	6	S	NaN	5751
513	2021-01-13	22:08	Wednesday	EGLINTON WEST STATION	PUMEL	0	0	NaN	NaN	0
1044	2021-01-27	22:00	Wednesday	YONGE-UNIVERSITY AND B	MUO	0	0	NaN	NaN	0
1045	2021-01-27	23:00	Wednesday	FINCH STATION	MUO	0	0	NaN	NaN	0
1362	2021-02-04	01:45	Thursday	LAWRENCE STATION	TUSC	0	0	S	NaN	5596

`dropna()`

In this case, the number of records without lines is relatively small. Most do not have delay durations. Some appear to be at rail yards, i.e. not on a rail line. For our analysis, we may drop them with the `dropna()` DataFrame method. We can drop rows missing lines by passing a `subset`.

In [42]:
```
delays_w_reasons = delays_w_reasons.dropna(subset='line')
```

Filtering data with `.loc[]` and `isin()`

We can use `.loc[]` to create a delays DataFrame without the invalid lines. To to this, we first create a list of values to exclude, then pass the list to the Series `isin()` method. Finally, we negate the expression, and assign the output back to `delays_w_reasons`.
Note: The negation operator here is `~`, not `!`. The `and` and `or` operators are different as well: `&` and `|` respectively.

In [43]:
```
# set up filter list
filter_list = ['999', '36 FINCH WEST', '35 JANE', '52', '41 KEELE']
```

In [44]:
```python
# filter out records with invalid lines
delays_w_reasons = delays_w_reasons.loc[~delays_w_reasons['line'].isin(filter_list)]
delays_w_reasons['line'].unique()
```

Out[44]:

```
array(['YU', 'BD', 'SHP', 'SRT', 'YU/BD', 'YONGE/UNIVERSITY/BLOOR',
       'YU / BD', 'YUS', 'SHEP', 'YUS & BD', 'YU & BD LINES', 'YUS/BD'],
      dtype=object)
```

Replacing values with `str.replace()`

To standardize the YU/BD values, we can replace the less common ones. One way to do this is by selecting the line Series and using `str.replace()`, like below for "YUS".

```
In [45]: delays_w_reasons['line'] = (delays_w_reasons['line']
                                     .str.replace('YUS', 'YU'))
delays_w_reasons['line'].unique()
```

```
Out[45]:

array(['YU', 'BD', 'SHP', 'SRT', 'YU/BD', 'YONGE/UNIVERSITY/BLOOR',
       'YU / BD', 'SHEP', 'YU & BD', 'YU & BD LINES'], dtype=object)
```

Another is to assign "YU/BD" to values selected by `.loc[]`

```
In [46]: yubd_list = ['YONGE/UNIVERSITY/BLOOR',
                      'YU / BD',
              'YU & BD',
              'YU & BD LINES']

# check the .loc[] selection
delays_w_reasons.loc[delays_w_reasons['line'].isin(yubd_list), 'line']
```

```
Out[46]:

590      YONGE/UNIVERSITY/BLOOR
852                     YU / BD
1137                    YU / BD
1628                    YU / BD
1672                    YU / BD
1700                    YU / BD
6725                    YU / BD
7469                    YU / BD
8034                    YU & BD
8301                    YU / BD
8341                    YU / BD
8463                    YU / BD
9164                    YU / BD
9541              YU & BD LINES
9839                    YU / BD
10792                   YU / BD
11119                   YU / BD
11299                   YU / BD
12128                   YU / BD
15574                   YU / BD
Name: line, dtype: object
```

In [47]:
```
delays_w_reasons.loc[delays_w_reasons['line'].isin(yubd_list), 'line'] = 'YU/BD'
delays_w_reasons['line'].unique()
```

Out[47]:

```
array(['YU', 'BD', 'SHP', 'SRT', 'YU/BD', 'SHEP'], dtype=object)
```

Practice

- Select the date, time, and station for all delays on the Yonge-University ("YU") and Bloor-Danforth ("BD") lines.
- Create a new DataFrame, `nonzero_delays`, containing delays with a `min_time` greater than zero.

Grouping

A core workflow in `pandas` is *split-apply-combine*:

- **splitting** data into groups
- **applying** a function to each group, such calculating group sums, standardizing data, or filtering out some groups
- **combining** the results into a data structure

This workflow starts by grouping data by calling the `groupby()` method. We'll pass in a column name or list of names to group by.

```
In [48]: line_groups = delays_w_reasons.groupby('line')
```

`groupby()` returns a grouped DataFrame that we can use to calculate groupwise statistics. The grouping column values become indexes, or row labels. **Note that this grouped DataFrame still references the original, so mutating one affects the other.**

In [49]:
```
# how many hours of delays did each line have in 2021?
line_groups['hour_delay'].sum()
```

Out[49]:

```
line
BD        329.47
SHEP        0.00
SHP        28.43
SRT        57.82
YU        477.50
YU/BD       0.00
Name: hour_delay, dtype: float64
```

We can group by more than one column by passing a list into `groupby()`. Data is grouped in the order of column names.

```
In [50]: # group by line first, then reason code description
         line_code_groups = delays_w_reasons.groupby(['line', 'code_description'])
```

Chaining methods and `unstack()`ing

We can *chain* methods together for convenience and code readability. Here, we calculate the `size()` of each group, then `unstack()` the resulting Series by the first part of the row label, line. The `tail()` method is added to the end so that the output takes less screen space.

In [51]:
```python
# view the number of delays by reason and line
line_code_groups.size().unstack(0).tail()
```

Out[51]:

line	BD	SHEP	SHP	SRT	YU	YU/BD
code_description						
Work Refusal	4.0	NaN	1.0	NaN	12.0	NaN
Work Vehicle	3.0	NaN	NaN	NaN	7.0	NaN
Work Zone Problems - Signals	4.0	NaN	4.0	NaN	5.0	NaN
Work Zone Problems - Track	12.0	NaN	NaN	NaN	29.0	NaN
Yard/Carhouse Related Problems	17.0	NaN	NaN	NaN	15.0	NaN

`agg()` regating

So far, we have applied one function at a time. The `agg()` DataFrame method lets us apply multiple functions on different columns at once.

`agg()`'s argument syntax is a little unusual. It follows this pattern:

```
DataFrame.agg(agg_colname=('column_to_aggregate', 'aggregation_function_name'),
              agg_colname2=('col_to_agg2', 'agg_func_name'))
```

```
In [52]: delay_summary = (delays_w_reasons
                          .groupby('date')
                  .agg(delay_count=('station', 'count'),
                       total_delay_min=('min_delay', 'sum'),
                       mean_delay_min=('min_delay', 'mean')))

delay_summary.head()
```

Out[52]:

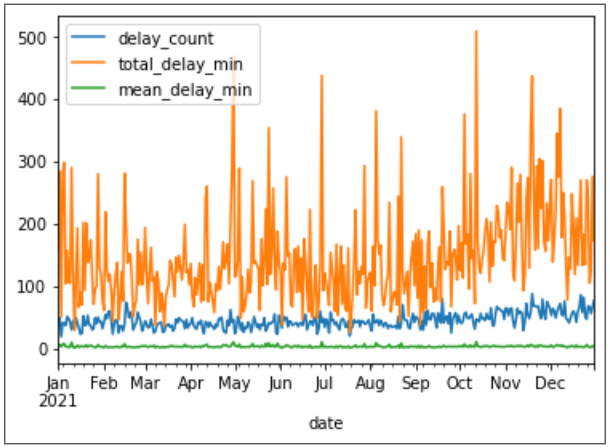
	delay_count	total_delay_min	mean_delay_min
date			
2021-01-01	36	159	4.416667
2021-01-02	49	284	5.795918
2021-01-03	19	51	2.684211
2021-01-04	41	284	6.926829
2021-01-05	40	298	7.450000

Plotting

The summary table we just generated can be easily plotted within `pandas`. Since the index contains dates, `pandas` automatically knows to plot values as time series data, with the dates in the x-axis -- we just have to call the `plot()` method.

```
In [53]: delay_summary.plot()
```

```
Out[53]:
<AxesSubplot:xlabel='date'>
```

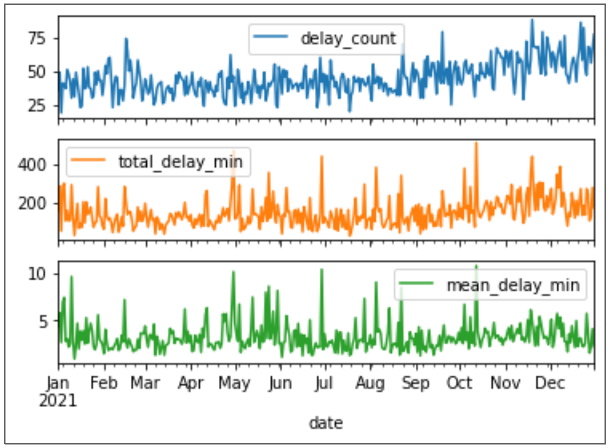


To create a separate plot for each column, we can specify `subplots=True`

```
In [54]: delay_summary.plot(subplots=True)
```

```
Out[54]:

array([<AxesSubplot:xlabel='date'>, <AxesSubplot:xlabel='date'>,
       <AxesSubplot:xlabel='date'>], dtype=object)
```

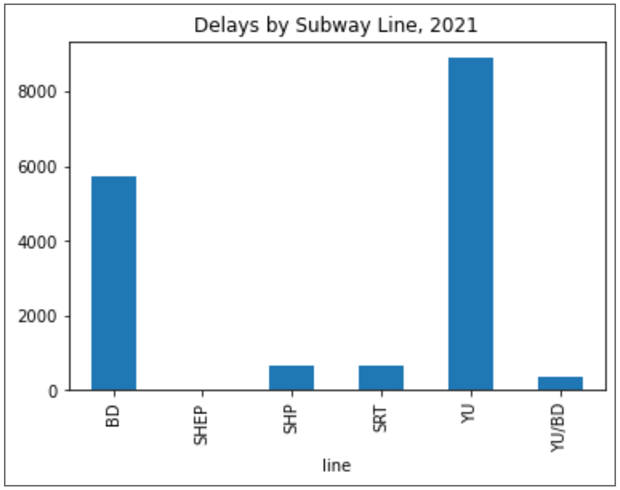


We can plot other aggregations too. Below, we use `line_groups` and calculate the size of each group, i.e., the number of delays reported on each line. Then we plot the data, telling `pandas` that the plot `kind` should be a bar graph, with TTC lines should in the `x`-axis. We also pass in a title.

In [55]:
```
(line_groups
    .size()
 .plot(x='line',
       kind='bar',
       title='Delays by Subway Line, 2021'))
```

Out[55]:

```
<AxesSubplot:title={'center':'Delays by Subway Line, 2021'}, xlabel='line'>
```



Practice

- Sum up and plot the total delay time, in hours, by line.
- Re-create `delay_summary`, but group by date and line this time. Add an additional aggregation column.
    - Experiment with the `unstack()` and `reset_index()` methods. Try passing in 0 or 1 as an argument. How does `delay_summary` change?

Writing to file

Exporting DataFrames

DataFrames have `to_[file format]()` methods, analagous to `pandas` read functions. The counterpart to `pd.read_csv()`, for instance, is `DataFrame.to_csv()`. The export methods generally take a file path to save to as their first argument. Additional arguments vary a bit by export format, but `index` is a common useful one. It takes a boolean of whether to write the index to file -- set it to `False` if the index is the numbered default.
Two additional useful parameters in `DataFrame.to_csv()` and `DataFrame.to_excel()` are `na_rep`, which takes a string to use for null values, and `columns`, which lets us write out a subset of columns.

In [57]:
```
# write delays_w_reasons to an Excel file
delays_w_reasons.to_excel('ttc_subway_delays_w_reasons.xlsx', index=False)
```

More wrangling

Neighbourhood Profiles

The bike theft data includes neighbourhood identifiers. These neighbourhoods are designated by City of Toronto, which publishes neighbourhood demographic profiles. Let's get this data to start investigating if neighbourhoods with more bike theft reports simply have higher populations. In the process, we will reinforce what we learned so far. We will also learn about two last ways to reshape data: `melt()`, which rearranges data from a wide format to a long format; and `pivot()`, which reorganizes data based on index and column values.

Getting data

Let's load the neighbourhood data and explore it.

In [58]:
```
profiles = pd.read_csv('../data/neighbourhood-profiles-2016-140-model.csv')
```

In [59]:
```
profiles.shape
```

Out[59]:

```
(2383, 146)
```

The neighbourhood profiles are in an unusual format. Neighbourhood names are in the columns, while attribute fields are rows, and there are thousands of attributes.

```
In [60]: profiles.head()
```

Out[60]:

	_id	Category	Topic	Data Source	Characteristic	City of Toronto	Agincourt North	Sou
0	1	Neighbourhood Information	Neighbourhood Information	City of Toronto	Neighbourhood Number	NaN	129	
1	2	Neighbourhood Information	Neighbourhood Information	City of Toronto	TSNS2020 Designation	NaN	No Designation	Des
2	3	Population	Population and dwellings	Census Profile 98-316-X2016001	Population, 2016	2,731,571	29,113	
3	4	Population	Population and dwellings	Census Profile 98-316-X2016001	Population, 2011	2,615,060	30,279	
4	5	Population	Population and dwellings	Census Profile 98-316-X2016001	Population Change 2011-2016	4.50%	-3.90%	

Because of the layout and formatting characters, all of the numeric values have been read in as text data. It also looks like the characteristics are not unique.

In [61]:
```
profiles.dtypes.value_counts()
```

Out[61]:

```
object    145
int64       1
dtype: int64
```

In [62]:
```
len(profiles['Characteristic'].unique())
```

Out[62]:

```
1651
```

Removing extra whitespace

The characteristic values contain extra whitespace. Let's remove the whitespace up with `str.strip()`.

In [63]:
```
# the whitespace is easier to see in a list than a Series
list(profiles['Characteristic'][95:100])
```

Out[63]:

```
['    Female parent',
 '    Male parent',
 'Couple census families in private households',
 '  Couples with children',
 '    1 child']
```

In [64]:
```
profiles['Characteristic'] = profiles['Characteristic'].str.strip()

# get the first 10 characteristics
list(profiles['Characteristic'][95:100])
```

Out[64]:

```
['Female parent',
 'Male parent',
 'Couple census families in private households',
 'Couples with children',
 '1 child']
```

Subsetting data

1651 characteristics is still a lot. Let's check out the categories to understand the areas covered.

```
In [65]: profiles['Category'].unique()
```

```
Out[65]:

array(['Neighbourhood Information', 'Population',
       'Families, households and marital status', 'Language', 'Income',
       'Immigration and citizenship', 'Visible minority', 'Ethnic origin',
       'Aboriginal peoples', 'Education', 'Housing', 'Language of work',
       'Labour', 'Journey to work', 'Mobility'], dtype=object)
```

"Journey to work" sounds relevant. We can use `.loc[]` to get the rows in that category, then select the Topic column and get its unique values.

```
In [66]: profiles.loc[profiles['Category'] == 'Journey to work']['Topic'].unique()
```

```
Out[66]:
```

```
array(['Commuting destination', 'Main mode of commuting',
       'Commuting duration', 'Time leaving for work'], dtype=object)
```

The "Population and dwellings" topic we saw in the DataFrame head looked promising as well. Let's check out the Characteristics in that topic.

```
In [67]: profiles.loc[profiles['Topic'] == 'Population and dwellings']['Characteristic'].unique()
```

```
Out[67]:

array(['Population, 2016', 'Population, 2011',
       'Population Change 2011-2016', 'Total private dwellings',
       'Private dwellings occupied by usual residents',
       'Population density per square kilometre',
       'Land area in square kilometres'], dtype=object)
```

Now that we know what topics we're interested in, let's create a subset DataFrame limited to them. We'll use the `copy()` DataFrame method to leave the original data untouched.

In [68]:
```
topics = ['Neighbourhood Information', 'Population and dwellings', 'Main mode of commuting']

# make sure it's an independent copy
profiles_subset = profiles.copy()

# get just the topics we're interested in
profiles_subset = profiles_subset.loc[profiles['Topic'].isin(topics)]
profiles_subset.shape
```

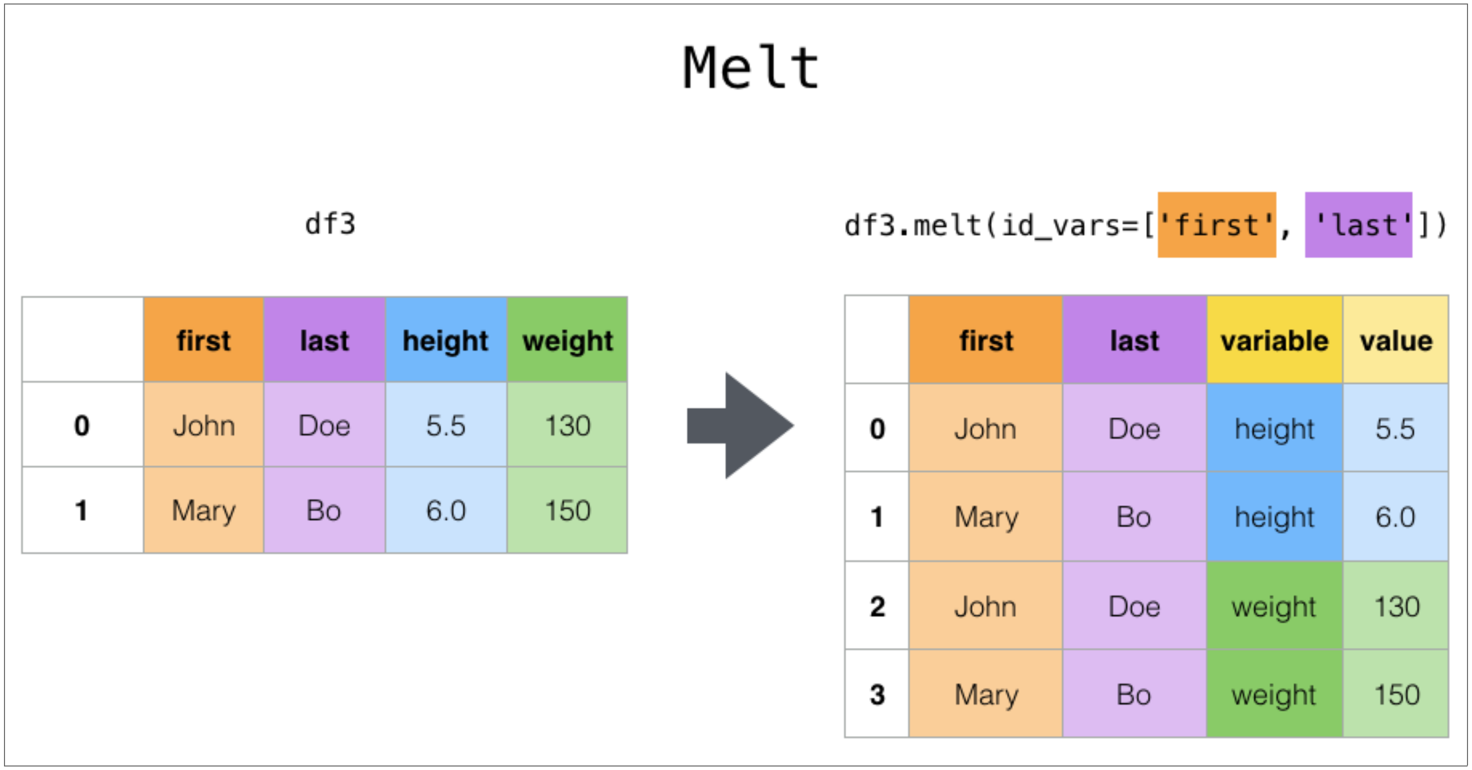
Out[68]:

```
(16, 146)
```

Reshaping data with `melt()`

Now we're ready to reshape our data. We can `drop()` the ID, data source, and category columns now.
To `melt()` a DataFrame, we specify `id_vars` -- the columns to keep as identifiers. All other columns are 'melted' into a new `variable` column. The values at DataFrame[id_vars, variable_col] move into a `value` column. We can change the names of the variable and value columns with the `var_name` and `value_name` arguments.

The pandas documentation provides an **illustrative example**:



Let's `melt()` the profiles subset. We'll keep `Topic` and `Characteristic` as our `id_vars`. This will melt the neighbourhood names into the `variable` column, which we'll rename `Neighbourhood`.

```
In [69]: profiles_melt = (profiles_subset
                          .drop(columns=['_id', 'Data Source', 'Category'])
                    .melt(id_vars=['Topic', 'Characteristic'],
                          var_name='Neighbourhood'))
profiles_melt.head()
```

Out[69]:

	Topic	Characteristic	Neighbourhood	value
0	Neighbourhood Information	Neighbourhood Number	City of Toronto	NaN
1	Neighbourhood Information	TSNS2020 Designation	City of Toronto	NaN
2	Population and dwellings	Population, 2016	City of Toronto	2,731,571
3	Population and dwellings	Population, 2011	City of Toronto	2,615,060
4	Population and dwellings	Population Change 2011-2016	City of Toronto	4.50%

Reshaping data with `pivot()`

The profile data is looking much closer to what we want! The next step is to make the Topic/Characteristic the column header, `pivot()` ing the values. To do this, we specify the column(s) to use as the `index`, or row labels; the column(s) whose values we should use as `column` names, and which column our `values` come from.
Pivoting on two columns creates a multi-level column header, so we then drop the top `Topic` level with `droplevel()`. Finally, we `reset_index()` to make neighbourhood names a regular column.

```
In [70]: neighbourhoods = (profiles_melt.pivot(index='Neighbourhood',
                                              columns=['Topic', 'Characteristic'],
                                       values='value')
                                 .droplevel(0, axis=1) # remove topic col header
                                 .reset_index()) # make Neighbourhood a regular column
         neighbourhoods.head()
```

Out[70]:

Characteristic	Neighbourhood	Neighbourhood Number	TSNS2020 Designation	Population, 2016	Population, 2011	Population Change 2011-2016	Total private dwellings	Privat dwelling occupie by usu resident
0	Agincourt North	129	No Designation	29,113	30,279	-3.90%	9,371	9,12
1	Agincourt South-	128	No Designation	23,757	21,988	8.00%	8,535	8,13

	Malvern West							
2	Alderwood	20	No Designation	12,054	11,904	1.30%	4,732	4,61
3	Annex	95	No Designation	30,526	29,177	4.60%	18,109	15,93
4	Banbury-Don Mills	42	No Designation	27,695	26,918	2.90%	12,473	12,12

Renaming all columns

Much better! These column names could be shorter, though. Let's rename them to be easier to work with. We could use the `rename()` DataFrame method, passing in a dictionary of old and new names. Since there isn't an easy renaming function, and some of the current names are very long, we will instead reassign a list of new names to the `columns` attribute of our DataFrame.

In [71]:
```python
# rename all columns
neighbourhoods.columns = ['neighbourhood',
                          'n_id',
                          'designation',
                          'pop_2016',
                          'pop_2011',
                          'pop_change',
                          'private_dwellings',
                          'occupied_dwllings',
                          'pop_dens',
                          'area',
                          'total_commuters',
                          'drive',
                          'car_passenger',
                          'transit',
                          'walk',
                          'bike',
                          'other']
neighbourhoods.columns
```

Out[71]:

```
Index(['neighbourhood', 'n_id', 'designation', 'pop_2016', 'pop_2011',
       'pop_change', 'private_dwellings', 'occupied_dwllings', 'pop_dens',
       'area', 'total_commuters', 'drive', 'car_passenger', 'transit', 'walk',
       'bike', 'other'],
      dtype='object')
```

Replacing values in multiple columns

All of the values in our neighbourhood data are text right now. Part of the problem is that numbers contain characters like commas and percentage signs. We can remove these from everywhere in our data with the DataFrame `replace()` method, which takes a string to look for and a replacement string. Normally, `replace()` looks for a perfect, full-string match. Since we're only looking for a substring match, we set `regex=True`.

In [72]:
```
# for those comfortable with regex, ',|%' and '[,%]' also work
neighbourhoods = (neighbourhoods.replace(',', '', regex=True)
                                .replace('%', '', regex=True))
neighbourhoods.head(2)
```

Out[72]:

	neighbourhood	n_id	designation	pop_2016	pop_2011	pop_change	private_dwellings	occupied_dwllings	pop_c
0	Agincourt North	129	No Designation	29113	30279	-3.90	9371	9120	39
1	Agincourt South-Malvern West	128	No Designation	23757	21988	8.00	8535	8136	30

`apply()`ing a function to multiple columns

Now the numbers look like numbers, but they are still strings. We can convert them with `pd.to_numeric()`, which takes a Series and returns it as the most appropriate numeric data type. Doing this for columns one-by-one would be tedious. Instead, we can use the `apply()` DataFrame method to run a function on every column in a DataFrame. `apply()` takes the name of the function to apply and any arguments needed to run that function. We only want to convert from `pop_2016` onwards, so we'll use `.loc[]` to select the correct columns.

```python
In [73]: # select all rows, columns from pop_2016 to end
         neighbourhoods.loc[:, 'pop_2016':] = neighbourhoods.loc[:, 'pop_2016':].apply(pd.to_numeric)
neighbourhoods.head()
```

Out[73]:

	neighbourhood	n_id	designation	pop_2016	pop_2011	pop_change	private_dwellings	occupied_dwllings	pop_c
0	Agincourt North	129	No Designation	29113	30279	-3.9	9371	9120	39
1	Agincourt South-Malvern West	128	No Designation	23757	21988	8.0	8535	8136	30
2	Alderwood	20	No Designation	12054	11904	1.3	4732	4616	24
3	Annex	95	No Designation	30526	29177	4.6	18109	15934	108
4	Banbury-Don Mills	42	No Designation	27695	26918	2.9	12473	12124	27

In [74]:
```
# confirm dtypes
neighbourhoods.dtypes
```

Out[74]:

```
neighbourhood          object
n_id                   object
designation            object
pop_2016                int64
pop_2011                int64
pop_change            float64
private_dwellings       int64
occupied_dwllings       int64
pop_dens                int64
area                  float64
total_commuters         int64
drive                   int64
car_passenger           int64
transit                 int64
walk                    int64
bike                    int64
other                   int64
dtype: object
```

Calculating more columns

Let's fix the population change column and calculate the percentage of commuters who bike.

In [75]:
```
neighbourhoods['pop_change'] = neighbourhoods['pop_change'] / 100
neighbourhoods['pct_bike'] = round(neighbourhoods['bike'] / neighbourhoods['total_commuters'], 3)
neighbourhoods.head()
```

Out[75]:

	neighbourhood	n_id	designation	pop_2016	pop_2011	pop_change	private_dwellings	occupied_dwllings	pop_c
0	Agincourt North	129	No Designation	29113	30279	-0.039	9371	9120	39
1	Agincourt South-Malvern West	128	No Designation	23757	21988	0.080	8535	8136	30
2	Alderwood	20	No Designation	12054	11904	0.013	4732	4616	24
3	Annex	95	No Designation	30526	29177	0.046	18109	15934	108
4	Banbury-Don Mills	42	No Designation	27695	26918	0.029	12473	12124	27

`merge()`ing

The profile are now ready to merge into the bike thefts data!

In [76]:
```
thefts_demo = pd.merge(thefts,
                        neighbourhoods,
                 how='left',
                 left_on='hood_id',
                 right_on='n_id')
thefts_demo
```

Out[76]:

	_id	objectid	event_unique_id	primary_offence	occurrence_date	occurrence_year	occurrence_month
0	1	17744	GO-20179016397	THEFT UNDER	2017-10-03	2017	October
1	2	17759	GO-20172033056	THEFT UNDER - BICYCLE	2017-11-08	2017	November
2	3	17906	GO-20189030822	THEFT UNDER - BICYCLE	2018-09-14	2018	September
3	4	17962	GO-2015804467	THEFT UNDER	2015-05-07	2015	May

	_id	objectid	event_unique_id	primary_offence	occurrence_date	occurrence_year	occurrence_month
4	5	17963	GO-20159002781	THEFT UNDER	2015-05-16	2015	May
...	...	...	...	...	...	...	...
25564	25565	9361	GO-2015543181	MISCHIEF UNDER	2015-04-01	2015	April
25565	25566	11318	GO-20169004589	THEFT UNDER	2016-05-16	2016	May
25566	25567	11462	GO-20169005434	THEFT UNDER	2016-06-04	2016	June
25567	25568	11695	GO-20161170896	THEFT UNDER	2016-07-04	2016	July

	_id	objectid	event_unique_id	primary_offence	occurrence_date	occurrence_year	occurrence_month
25568	25569	11883	GO-20169007653	THEFT UNDER - BICYCLE	2016-07-22	2016	July

25569 rows × 51 columns

Grouping and plotting

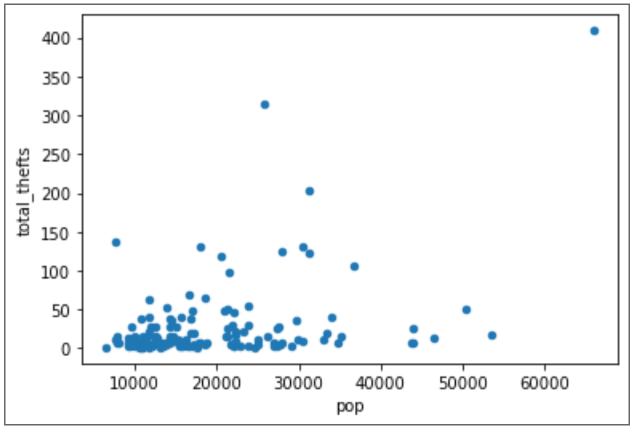
With the datasets joned, we can aggregate and plot the data. We can start using statistical methods, like `corr()` to check for relationships between variables as well.

```
In [77]: thefts_2016_grouped = (thefts_demo
                                  .query('occurrence_year == 2016')
                         .groupby(['neighbourhood']))
```

In [81]:
```
# thefts counts vs population
(thefts_2016_grouped
 .agg(total_thefts=('_id', 'count'),
      pop=('pop_2016', 'median'),
      pct_bike=('pct_bike', 'mean'))
 .reset_index()
 .plot(kind='scatter', y='total_thefts', x='pop'))
```

Out[81]:

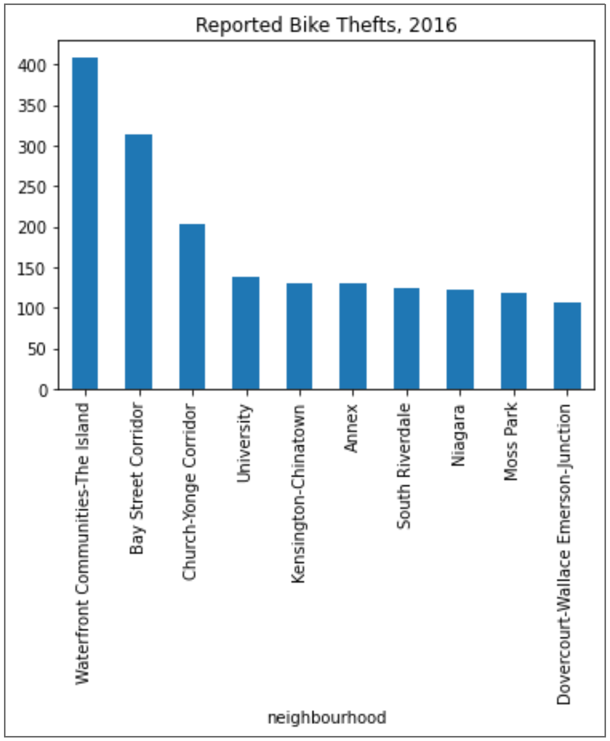
```
<AxesSubplot:xlabel='pop', ylabel='total_thefts'>
```



In [82]:
```python
(thefts_2016_grouped
        .size()
 .sort_values(ascending=False)
 .head(10)
 .plot(kind='bar', title='Reported Bike Thefts, 2016'))
```

Out[82]:

```
<AxesSubplot:title={'center':'Reported Bike Thefts, 2016'}, xlabel='neighbourhood'>
```



In [83]:
```
# a quick correlation check
        (thefts_2016_grouped
 .agg(total_thefts=('_id', 'count'),
     pop=('pop_2016', 'median'),
     dens=('pop_dens', 'median'),
     pct_bike=('pct_bike', 'mean'))
 .corr('spearman'))
```

Out[83]:

	total_thefts	pop	dens	pct_bike
total_thefts	1.000000	0.267761	0.485556	0.650101
pop	0.267761	1.000000	0.020082	-0.218934
dens	0.485556	0.020082	1.000000	0.603434
pct_bike	0.650101	-0.218934	0.603434	1.000000

References

Programming

- pandas development team. *API reference*. **https://pandas.pydata.org/pandas-docs/stable/reference/index.html**
- pandas development team. *User guide*. **https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html**
- *Python strftime cheatsheet*. **https://strftime.org/**

Data Sources

- Open Data Toronto. *Neighbourhood Profiles*. **https://open.toronto.ca/dataset/neighbourhood-profiles/**
- Open Data Toronto. *TTC Subway Delay Data*. **https://open.toronto.ca/dataset/ttc-subway-delay-data/**
- Open Data Toronto. *Bicyle Thefts*. **https://open.toronto.ca/dataset/bicycle-thefts/**