# Understanding Pandas Series and DataFrames

## Introduction ¶

In this lesson, we're digging into `Series` and `DataFrames`, the two main data types you'll work with in the pandas library.

## Objectives

You will be able to:

- Use the `.map()` and `.apply()` methods to apply a function to a pandas Series or DataFrame
- Perform operations to change the structure of pandas DataFrames
- Change the index of a pandas DataFrame
- Change data types of columns in pandas DataFrames

## Pandas Data Types vs. Base Python Data Types

Built-in Python data types such as lists, dictionaries, and sets can be powerful in limited settings, but they often require:

- Several lines of "boilerplate" code to accomplish common tasks, which opens up the possibility of mistakes
- Extra unnecessary memory space for storing data types. For example, if you have a Python `list` of 100 integers, you are also storing the fact that each one is an integer, and you store that same information again if you increase the length of the list by 1

Using pandas data types such as Series and DataFrames instead of built-in Python data types can address both of these issues. Series and DataFrames have a range of built-in methods which make standard practices and procedures streamlined. Some of these methods can result in dramatic performance gains. To read more about these methods, make sure to continuously reference the Pandas documentation (https://pandas.pydata.org/pandas-docs/stable/).

With built-in Python types, it is useful to know all of the available methods, since each of them is likely to come up at one point or another, and there aren't that many. **In pandas, by contrast, it is impossible to know every method at any given time, and you should not devote much time to memorization.** We will not deeply explain every pandas method in these upcoming lessons and labs. A critical part of every data scientist's job is to investigate documentation to learn about components of these tools on your own. When you are trying to do something new with your data, there will probably be a pandas method for it, and you'll work over time to get better at finding the appropriate method using the documentation, Google, and StackOverflow.

## Setup

This MTA turnstile dataset is a great place for us to get our hands dirty wrangling and cleaning some data! Here's the data dictionary if you want to know more about the dataset http://web.mta.info/developers/resources/nyct/turnstile/ts_Field_Description.txt (http://web.mta.info/developers/resources/nyct/turnstile/ts_Field_Description.txt)

Let's import the packages we need and load and preview the dataset.

## Import `pandas`

In [1]:
```
import pandas as pd
```

## Load and Preview Dataset

In [2]:
```
df = pd.read_csv('turnstile_180901.txt', dtype=str)
df
```

Out[2]:

| | C/A | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | DESC |
|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 00:00:00 | REGULAR |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 04:00:00 | REGULAR |
| **2** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 08:00:00 | REGULAR |
| **3** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 12:00:00 | REGULAR |
| **4** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 16:00:00 | REGULAR |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **197620** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 05:00:00 | REGULAR |
| **197621** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 09:00:00 | REGULAR |
| **197622** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 13:00:00 | REGULAR |
| **197623** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 17:00:00 | REGULAR |
| **197624** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 21:00:00 | REGULAR |

197625 rows × 11 columns

In [3]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 197625 entries, 0 to 197624
Data columns (total 11 columns):
 #   Column                                                          Non-
Null Count   Dtype
---  ------                                                          ----
----------   -----
 0   C/A                                                             1976
25 non-null  object
 1   UNIT                                                            1976
25 non-null  object
 2   SCP                                                             1976
25 non-null  object
 3   STATION                                                         1976
25 non-null  object
 4   LINENAME                                                        1976
25 non-null  object
 5   DIVISION                                                        1976
25 non-null  object
 6   DATE                                                            1976
25 non-null  object
 7   TIME                                                            1976
25 non-null  object
 8   DESC                                                            1976
25 non-null  object
 9   ENTRIES                                                         1976
25 non-null  object
 10  EXITS                                                           1976
25 non-null  object
dtypes: object(11)
memory usage: 16.6+ MB
```

# Using `.map()` to Transform Values

A standard data preparation step you might need to perform is "cleaning up" the values of a dataset so they follow your desired format. The `.map()` method is key for this task.

## Passing in a Dictionary

One of the most straightforward ways to use the `.map()` method on a pandas Series is with a **dictionary of values you want to use to replace other values**.

Let's say we want to look at the `DIVISION` column:

In [5]:
```
df['DIVISION'].value_counts()
```

Out[5]:
```
IRT    72198
IND    69274
BMT    41727
PTH    12788
SRT     1386
RIT      252
Name: DIVISION, dtype: int64
```

*If you have not seen `value_counts()` before, this would be a good time to check out the [documentation for it (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.value_counts.html)]! We use this method very frequently to understand the distribution of categorical data*

We look up some additional information, and locate the following mappings:

| Abbreviation | Full Name |
|---|---|
| IRT | Interborough Rapid Transit Company |
| IND | Independent Subway System |
| BMT | Brooklyn–Manhattan Transit Corporation |
| PTH | Port Authority Trans-Hudson (PATH) |
| SRT | Staten Island Rapid Transit |
| RIT | Roosevelt Island Tram |

To represent this in Python, let's use a dictionary with the abbreviations as keys and full names as values.

In [7]:
```
division_mapping = {
    "IRT": "Interborough Rapid Transit Company",
    "IND": "Independent Subway System",
    "BMT": "Brooklyn–Manhattan Transit Corporation",
    "PTH": "Port Authority Trans-Hudson (PATH)",
    "SRT": "Staten Island Rapid Transit",
    "RIT": "Roosevelt Island Tram"
}
```

Now we can call the `.map()` method to return a Series with the abbreviations transformed into full names:

In [8]:
```
df['DIVISION'].map(division_mapping)
```

Out[8]:
```
0         Brooklyn-Manhattan Transit Corporation
1         Brooklyn-Manhattan Transit Corporation
2         Brooklyn-Manhattan Transit Corporation
3         Brooklyn-Manhattan Transit Corporation
4         Brooklyn-Manhattan Transit Corporation
                          ...
197620                    Roosevelt Island Tram
197621                    Roosevelt Island Tram
197622                    Roosevelt Island Tram
197623                    Roosevelt Island Tram
197624                    Roosevelt Island Tram
Name: DIVISION, Length: 197625, dtype: object
```

Let's go ahead and replace the `DIVISION` column in `df` with these new, transformed values:

In [25]:
```
df['DIVISION'] = df['DIVISION'].map(division_mapping)
df['DIVISION'].value_counts()
```

Out[25]:
```
Interborough Rapid Transit Company        72198
Independent Subway System                 69274
Brooklyn-Manhattan Transit Corporation    41727
Port Authority Trans-Hudson (PATH)        12788
Staten Island Rapid Transit                1386
Roosevelt Island Tram                       252
Name: DIVISION, dtype: int64
```

## Passing in a Function

Another way to use the `.map()` method is by passing in a function.

Let's say we want to look at the `LINENAME` column:

In [26]:
```
df['LINENAME'].value_counts()
```

Out[26]:
```
1        24092
6        11263
7         9562
F         7146
25        6881
         ...
ACG        210
S          210
ND         209
S2345      168
23ACE      168
Name: LINENAME, Length: 113, dtype: int64
```

The `...` in the middle means this is a shortened version of the full value counts. `Length: 113` means there are 113 different categories present in the column.

Rather than substituting these values with some other values like we did with `DIVISION`, let's say we want a boolean (`True` or `False`) value representing whether or not the `LINENAME` contains the string `"N"` (i.e. whether or not the stop is an N line stop). We can do this with a function.

### Functions in Python Review

Let's review how to do this:

- In Python, we define a function using the `def` keyword. Afterwards, we give the function a name, followed by parentheses. Any required (or optional) parameters are specified within the parentheses (`()`), just as you would when you call a function.
- You then specify the function's behavior using a colon (`:`) and an indentation, much the same way you would a `for` loop or conditional block.
- Finally, if you want your function to return something (as with the `str.pop()` method) as opposed to a function that simply does something in the background but returns nothing (such as `list.append()`), you must use the `return` keyword. Note that as soon as a function hits a point in execution where something is returned, the function would terminate and no further commands would be executed. In other words the `return` command both returns a value and forces termination of the function.

Let's define a function that takes in a string and returns `True` if that string contains the letter `'N'`, and returns `False` otherwise.

In [27]:
```
def contains_n(text):
    if 'N' in text:
        return True
    else:
        return False

# Or the shorter, more pythonic way
# (this overwrites the previous function)
def contains_n(text):
    return 'N' in text
```

Then call the `.map()` method and pass in the function:

In [29]:
```
df['LINENAME'].map(contains_n)
```

Out[29]:
```
0          True
1          True
2          True
3          True
4          True
           ...
197620    False
197621    False
197622    False
197623    False
197624    False
Name: LINENAME, Length: 197625, dtype: bool
```

Note that for a pandas Series, the `.apply()` method can be used interchangeably with the `.map()` method when a function is provided (with somewhat different implementations "under the hood"):

In [30]:
```
df['LINENAME'].apply(contains_n)
```

Out[30]:
```
0          True
1          True
2          True
3          True
4          True
           ...
197620    False
197621    False
197622    False
197623    False
197624    False
Name: LINENAME, Length: 197625, dtype: bool
```

Rather than replacing `LINENAME` in the dataframe, let's create a new column to hold this result:

In [31]:
```
df['On_N_Line'] = df['LINENAME'].map(contains_n)
df
```

Out[31]:

| | C/A | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | DESC |
|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 00:00:00 | REGULAR |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 04:00:00 | REGULAR |
| **2** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 08:00:00 | REGULAR |
| **3** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 12:00:00 | REGULAR |
| **4** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 16:00:00 | REGULAR |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **197620** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 05:00:00 | REGULAR |
| **197621** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 09:00:00 | REGULAR |
| **197622** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 13:00:00 | REGULAR |
| **197623** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 17:00:00 | REGULAR |
| **197624** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 21:00:00 | REGULAR |

197625 rows × 12 columns

In [33]:
```
df['On_N_Line'].value_counts(normalize=True)
```

Out[33]:
```
False    0.870441
True     0.129559
Name: On_N_Line, dtype: float64
```

**Functions + `.map()` Explanation**

Above we used the `.map()` method for Pandas series (documentation here (https://pandas.pydata.org/docs/reference/api/pandas.Series.map.html)). This allows us to pass a function that will be applied to each and every data entry within the series. This line of Python code:

```
df['On_N_Line'] = df['LINENAME'].map(contains_n)
```

Is essentially the equivalent of this:

```
# Create an empty list
on_n_line = []
# Loop over every row in the dataframe
for _, row in df.iterrows():
    # Call the function to see if LINENAME contains N
    row_contains_n = contains_n(row['LINENAME'])
    # Append this result to a list
    on_n_line.append(row_contains_n)
# Add this list to the dataframe as a new column
df['On_N_Line'] = on_n_line
```

Note that the above snippet is much more complicated than the `.map()` syntax AND the code would run more slowly because it is less efficient. **If you ever find yourself trying to write a `for` loop that loops over all rows in a DataFrame, you are probably doing it wrong!**

As shorthand, since this function is only one line we could also pass a lambda function to determine whether or not each row was on the N line or not, rather than declaring a separate function:

```
df['On_N_Line'] = df['LINENAME'].map(lambda x: 'N' in x)
```

This is shorter and equivalent to the functions defined above. Lambda functions are often more convenient, but have less functionality than defining functions explicitly.

## Vectorized Pandas Logic for N Line

**Even better than using `.map()` with a custom function is using one of the highly efficient methods built into pandas**. These will exist for most common tasks, and checking whether a string contains another string is no exception. The best way to make the `On_N_Line` column is actually using `pandas.Series.str.contains` (documentation here (https://pandas.pydata.org/docs/reference/api/pandas.Series.str.contains.html)):

```
df['On_N_Line'] = df['LINENAME'].str.contains('N', regex=False)
```

Sometimes, like with this example, the naming is slightly different between base Python and pandas. In base Python we ask whether one string is `in` another, whereas in pandas we ask whether one `.contains` another. Try browsing the available methods on the left side menu of the pandas documentation to find what you're looking for in cases like this.

Whether you use `.map()` or `.str.contains()` will matter more as the dataframe size increases. If you are working with a relatively small dataframe, you may have an easier time if you focus on figuring out something that *works* rather than worrying too much about finding the optimal technique.

# Transforming Columns

## Cleaning Column Names

Sometimes, you have messy column names. Let's look at what we currently have:

In [9]:
```
df.columns
```

Out[9]:
```
Index(['C/A', 'UNIT', 'SCP', 'STATION', 'LINENAME', 'DIVISION', 'DATE', 'TIME',
       'DESC', 'ENTRIES',
       'EXITS                                                               '],
      dtype='object')
```

You might notice that the `EXITS` column has a lot of annoying whitespace following it.

We can quickly use a list comprehension to clean up all of the column names.

In [10]:
```
[col.strip() for col in df.columns]
```

Out[10]:
```
['C/A',
 'UNIT',
 'SCP',
 'STATION',
 'LINENAME',
 'DIVISION',
 'DATE',
 'TIME',
 'DESC',
 'ENTRIES',
 'EXITS']
```

Because there are relatively few column names, a list comprehension like that is usually sufficient. However you can use similar techniques to the ones described above if you need to:

In [35]:
```
df.columns.str.strip()
```

Out[35]:
```
Index(['C/A', 'UNIT', 'SCP', 'STATION', 'LINENAME', 'DIVISION', 'DATE', 'TIME',
       'DESC', 'ENTRIES', 'EXITS', 'On_N_Line'],
      dtype='object')
```

In [37]:
```
df.columns.map(lambda col: col.strip())
```

Out[37]:
```
Index(['C/A', 'UNIT', 'SCP', 'STATION', 'LINENAME', 'DIVISION', 'DATE', 'TIME',
       'DESC', 'ENTRIES', 'EXITS', 'On_N_Line'],
      dtype='object')
```

Note that none of these have actually modified the columns so far:

In [38]:
```
df.columns
```

Out[38]:
```
Index(['C/A', 'UNIT', 'SCP', 'STATION', 'LINENAME', 'DIVISION', 'DATE', 'TIME',
       'DESC', 'ENTRIES',
       'EXITS                                                               ',
       'On_N_Line'],
      dtype='object')
```

We need to reassign `df.columns` for this to happen:

In [39]:
```
# Even though this is assigning a list of strings, it
# will be cast to an Index automatically
df.columns = [col.strip() for col in df.columns]
df.columns
```

Out[39]:
```
Index(['C/A', 'UNIT', 'SCP', 'STATION', 'LINENAME', 'DIVISION', 'DATE', 'TIME',
       'DESC', 'ENTRIES', 'EXITS', 'On_N_Line'],
      dtype='object')
```

## Renaming Columns

You can also rename columns using dictionaries. Unlike `.map()`, which will replace values with NaN if they do not have an associated key in the dictionary, `.rename()` will only replace values that appear in the dictionary. This is useful if you only want to replace some values.

Let's say we want to rename `C/A` to `CONTROL_AREA` (the data dictionary indicates that this is what it stands for).

In [11]:
```
df.rename(columns={'C/A' : 'CONTROL_AREA'})
```

Out[11]:

| | CONTROL_AREA | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | |
|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 00:00:00 | RE |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 04:00:00 | RE |
| **2** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 08:00:00 | RE |
| **3** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 12:00:00 | RE |
| **4** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 16:00:00 | RE |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **197620** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 05:00:00 | RE |
| **197621** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 09:00:00 | RE |
| **197622** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 13:00:00 | RE |
| **197623** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 17:00:00 | RE |
| **197624** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | RIT | 08/31/2018 | 21:00:00 | RE |

197625 rows × 11 columns

Again, note that the dataframe was not automatically transformed by doing this. If we look at it now, `C/A` is still there:

In [42]:
```
df
```

Out[42]:

| | C/A | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | DESC |
|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 00:00:00 | REGULAR |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 04:00:00 | REGULAR |
| **2** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 08:00:00 | REGULAR |
| **3** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 12:00:00 | REGULAR |
| **4** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 16:00:00 | REGULAR |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| **197620** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 05:00:00 | REGULAR |
| **197621** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 09:00:00 | REGULAR |
| **197622** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 13:00:00 | REGULAR |
| **197623** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 17:00:00 | REGULAR |
| **197624** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 21:00:00 | REGULAR |

197625 rows × 12 columns

If we want the change to "stick", one way to do that is to use `inplace=True` :

In [43]:
```
df.rename(columns={'C/A' : 'CONTROL_AREA'}, inplace=True)
```

Now the value has actually been changed:

In [44]:
```
df
```

Out[44]:

| | CONTROL_AREA | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | |
|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 00:00:00 | I |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 04:00:00 | I |
| **2** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 08:00:00 | I |
| **3** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 12:00:00 | I |
| **4** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 16:00:00 | I |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **197620** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 05:00:00 | I |
| **197621** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 09:00:00 | I |
| **197622** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 13:00:00 | I |
| **197623** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 17:00:00 | I |
| **197624** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 21:00:00 | I |

197625 rows × 12 columns

Note that this behavior (not changing the contents of the dataframe unless you use `inplace=True` or reassign the variable) is not a mistake or oversight in pandas. It is a useful feature that lets you preview the outcome of an operation before permanently applying it! This is especially important if you are dropping data or transforming it in a way that is not reversible.

## Dropping Columns

Let's say we have determined that the `DESC` column doesn't matter. We can test out dropping it like this:

In [45]:
```
df.drop('DESC', axis=1)
```

Out[45]:

| | CONTROL_AREA | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | |
|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 00:00:00 | ( |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 04:00:00 | ( |
| **2** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 08:00:00 | ( |
| **3** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 12:00:00 | ( |
| **4** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08/25/2018 | 16:00:00 | ( |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **197620** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 05:00:00 | ( |
| **197621** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 09:00:00 | ( |
| **197622** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 13:00:00 | ( |
| **197623** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 17:00:00 | ( |
| **197624** | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 08/31/2018 | 21:00:00 | ( |

197625 rows × 11 columns

Note the `axis=1` argument. By default, `df.drop()` tries to drop rows ( `axis=0` ) with the specified index, e.g.:

In [13]:
```
df.drop(3).head()
```

Out[13]:

| | C/A | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | DESC | ENTRIES |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 00:00:00 | REGULAR | 0006736067 |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 04:00:00 | REGULAR | 0006736087 |
| **2** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 08:00:00 | REGULAR | 0006736105 |
| **4** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 16:00:00 | REGULAR | 0006736349 |
| **5** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 20:00:00 | REGULAR | 0006736562 |

If you are trying to drop a column and you forget the `axis=1` , you'll get an error message like this:

In [14]:
```
df.drop('DESC')
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
/tmp/ipykernel_296/3582495597.py in <module>
----> 1 df.drop('DESC')

/opt/conda/lib/python3.9/site-packages/pandas/util/_decorators.py in wrapper(*a
rgs, **kwargs)
    309                     stacklevel=stacklevel,
    310                 )
--> 311             return func(*args, **kwargs)
    312
    313         return wrapper

/opt/conda/lib/python3.9/site-packages/pandas/core/frame.py in drop(self, label
s, axis, index, columns, level, inplace, errors)
   4899                 weight  1.0     0.8
   4900         """
-> 4901         return super().drop(
   4902             labels=labels,
   4903             axis=axis,

/opt/conda/lib/python3.9/site-packages/pandas/core/generic.py in drop(self, lab
els, axis, index, columns, level, inplace, errors)
   4145         for axis, labels in axes.items():
   4146             if labels is not None:
-> 4147                 obj = obj._drop_axis(labels, axis, level=level, errors=
errors)
   4148
   4149         if inplace:

/opt/conda/lib/python3.9/site-packages/pandas/core/generic.py in _drop_axis(sel
f, labels, axis, level, errors)
   4180                 new_axis = axis.drop(labels, level=level, errors=errors
)
   4181             else:
-> 4182                 new_axis = axis.drop(labels, errors=errors)
   4183             result = self.reindex(**{axis_name: new_axis})
   4184

/opt/conda/lib/python3.9/site-packages/pandas/core/indexes/base.py in drop(sel
f, labels, errors)
   6016         if mask.any():
   6017             if errors != "ignore":
-> 6018                 raise KeyError(f"{labels[mask]} not found in axis")
   6019             indexer = indexer[~mask]
   6020         return self.delete(indexer)

KeyError: "['DESC'] not found in axis"
```

Let's go ahead and permanently drop that column:

In [15]:
```
df = df.drop('DESC', axis=1)
df.head()
```

Out[15]:

| | C/A | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | ENTRIES | EXITS |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 00:00:00 | 0006736067 | 000228318 |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 04:00:00 | 0006736087 | 000228318 |
| **2** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 08:00:00 | 0006736105 | 000228322 |
| **3** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 12:00:00 | 0006736180 | 000228331 |
| **4** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | BMT | 08/25/2018 | 16:00:00 | 0006736349 | 000228338 |

## Changing Column Types

Another common data munging technique can be reformatting column types. We first previewed column types above using the `df.info()` method, which we'll repeat here.

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 197625 entries, 0 to 197624
Data columns (total 10 columns):
 #   Column                                                        Non-
Null Count   Dtype
---  ------                                                        ----
----------   -----
 0   C/A                                                           1976
25 non-null  object
 1   UNIT                                                          1976
25 non-null  object
 2   SCP                                                           1976
25 non-null  object
 3   STATION                                                       1976
25 non-null  object
 4   LINENAME                                                      1976
25 non-null  object
 5   DIVISION                                                      1976
25 non-null  object
 6   DATE                                                          1976
25 non-null  object
 7   TIME                                                          1976
25 non-null  object
 8   ENTRIES                                                       1976
25 non-null  object
 9   EXITS                                                         1976
25 non-null  object
dtypes: object(10)
memory usage: 15.1+ MB
```

We can also check the data type of an individual column, rather than listing all of them:

In [29]:
```
print(df['ENTRIES'].dtype)
```

```
object
```

In this case we specified `dtype=str` when we opened the file, telling pandas to treat all of the columns as strings initially. So currently every column except for `On_N_Line` is dtype `object`.

A common transformation needed is converting numbers stored as text (dtype `object`) to *float* or *integer* representations.

Let's look more closely at `ENTRIES`:

In [30]:
```
df.loc[:5, 'ENTRIES']
```

Out[30]:
```
0    0006736067
1    0006736087
2    0006736105
3    0006736180
4    0006736349
5    0006736562
Name: ENTRIES, dtype: object
```

Those seem like integers. Let's try converting the type:

In [31]:
```
df.loc[:5, 'ENTRIES'].astype(int)
```

Out[31]:
```
0    6736067
1    6736087
2    6736105
3    6736180
4    6736349
5    6736562
Name: ENTRIES, dtype: int64
```

Note that again, we could use `.map()` instead:

In [32]:
```
# int is a built-in function, so we do not
# need to declare a helper function here
df.loc[:5, 'ENTRIES'].map(int)
```

Out[32]:
```
0    6736067
1    6736087
2    6736105
3    6736180
4    6736349
5    6736562
Name: ENTRIES, dtype: int64
```

That looks good, so let's change the type of that column:

In [33]:
```
df['ENTRIES'] = df['ENTRIES'].astype(int)
```

In [34]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 197625 entries, 0 to 197624
Data columns (total 11 columns):
 #   Column        Non-Null Count   Dtype 
---  ------        --------------   ----- 
 0   CONTROL_AREA  197625 non-null  object
 1   UNIT          197625 non-null  object
 2   SCP           197625 non-null  object
 3   STATION       197625 non-null  object
 4   LINENAME      197625 non-null  object
 5   DIVISION      197625 non-null  object
 6   DATE          197625 non-null  object
 7   TIME          197625 non-null  object
 8   ENTRIES       197625 non-null  int64 
 9   EXITS         197625 non-null  object
 10  On_N_Line     197625 non-null  bool  
dtypes: bool(1), int64(1), object(9)
memory usage: 15.3+ MB
```

Attempting to convert a string column to int or float will produce errors if there are actually non-numeric characters. For example, `LINENAME`:

In [35]:
```
df['LINENAME'] = df['LINENAME'].astype(int)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-35-606443ef165a> in <module>
----> 1 df['LINENAME'] = df['LINENAME'].astype(int)

//anaconda3/envs/learn-env/lib/python3.8/site-packages/pandas/core/generic.py i
n astype(self, dtype, copy, errors)
   5544         else:
   5545             # else, only a single dtype is given
-> 5546             new_data = self._mgr.astype(dtype=dtype, copy=copy, errors=
errors,)
   5547             return self._constructor(new_data).__finalize__(self, metho
d="astype")
   5548

//anaconda3/envs/learn-env/lib/python3.8/site-packages/pandas/core/internals/ma
nagers.py in astype(self, dtype, copy, errors)
    593         self, dtype, copy: bool = False, errors: str = "raise"
    594     ) -> "BlockManager":
--> 595         return self.apply("astype", dtype=dtype, copy=copy, errors=erro
rs)
    596
    597     def convert(

//anaconda3/envs/learn-env/lib/python3.8/site-packages/pandas/core/internals/ma
nagers.py in apply(self, f, align_keys, **kwargs)
    404                 applied = b.apply(f, **kwargs)
    405             else:
--> 406                 applied = getattr(b, f)(**kwargs)
    407             result_blocks = _extend_blocks(applied, result_blocks)
    408

//anaconda3/envs/learn-env/lib/python3.8/site-packages/pandas/core/internals/bl
ocks.py in astype(self, dtype, copy, errors)
    593             vals1d = values.ravel()
    594             try:
--> 595                 values = astype_nansafe(vals1d, dtype, copy=True)
    596             except (ValueError, TypeError):
    597                 # e.g. astype_nansafe can fail on object-dtype of strin
gs

//anaconda3/envs/learn-env/lib/python3.8/site-packages/pandas/core/dtypes/cast.
py in astype_nansafe(arr, dtype, copy, skipna)
    970         # work around NumPy brokenness, #1987
    971         if np.issubdtype(dtype.type, np.integer):
--> 972             return lib.astype_intsafe(arr.ravel(), dtype).reshape(arr.s
hape)
    973
    974         # if we have a datetime/timedelta array of objects

pandas/_libs/lib.pyx in pandas._libs.lib.astype_intsafe()

ValueError: invalid literal for int() with base 10: 'NQR456W'
```

# Converting Dates

A slightly more complicated data type transformation is creating *date* or *datetime* objects. These are pandas data types that have useful information such as being able to quickly calculate the time between two days, or extracting the day of the week from a given date. However, if we look at our current date column, we will notice it is simply a dtype `object` (all strings).

In [36]:
```
df['DATE'].head()
```

Out[36]:
```
0    08/25/2018
1    08/25/2018
2    08/25/2018
3    08/25/2018
4    08/25/2018
Name: DATE, dtype: object
```

## `pd.to_datetime()`

This is the handiest of methods when converting strings to datetime objects.

Often you can simply pass the series into this function, but it is good practice to preview the results first to prevent overwriting data if some error occurs.

In [37]:
```
pd.to_datetime(df['DATE']).head()
```

Out[37]:
```
0   2018-08-25
1   2018-08-25
2   2018-08-25
3   2018-08-25
4   2018-08-25
Name: DATE, dtype: datetime64[ns]
```

That worked!

Note that the `dtype` has changed from `object` to `datetime64[ns]`.

Sometimes the above won't work and you'll have to explicitly pass an argument describing how the date is formatted.
To do that, you have to use some datetime codes. Here's a preview of some of the most common ones:

| Code | Meaning | Example |
|---|---|---|
| %a | Weekday as locale's abbreviated name. | Mon |
| %A | Weekday as locale's full name. | Monday |
| %w | Weekday as a decimal number, where 0 is Sunday and 6 is Saturday. | 1 |
| %d | Day of the month as a zero-padded decimal number. | 30 |
| %-d | Day of the month as a decimal number. (Platform specific) | 30 |
| %b | Month as locale's abbreviated name. | Sep |
| %B | Month as locale's full name. | September |
| %m | Month as a zero-padded decimal number. | 09 |
| %-m | Month as a decimal number. (Platform specific) | 9 |
| %y | Year without century as a zero-padded decimal number. | 13 |
| %Y | Year with century as a decimal number. | 2013 |
| %H | Hour (24-hour clock) as a zero-padded decimal number. | 07 |
| %-H | Hour (24-hour clock) as a decimal number. (Platform specific) | 7 |
| %I | Hour (12-hour clock) as a zero-padded decimal number. | 07 |
| %-I | Hour (12-hour clock) as a decimal number. (Platform specific) | 7 |
| %p | Locale's equivalent of either AM or PM. | AM |
| %M | Minute as a zero-padded decimal number. | 06 |
| %-M | Minute as a decimal number. (Platform specific) | 6 |

To explicitly pass formatting parameters, start by previewing your dates to understand their current format as strings.

In [38]:
```
# Selecting just the first date entry
df['DATE'].iloc[0]
```

Out[38]: '08/25/2018'

Based on that, it looks like we have:

- `08` : a month code with zero padding. So that's `%m` in the table above
- `/` : a delimiter
- `25` : a day of the month. It's not clear that it's zero-padded but we'll go ahead and say it's a `%d` in the table above
- `/` : another delimiter

- `2018` : a year with the century (it would just be `18` without the century). So that's `%Y` in the table above

All together, `%m + / + %d + / + %Y = %m/%d/%Y` , so we'll use that as the format.

In [39]:
```
pd.to_datetime(df['DATE'], format='%m/%d/%Y').head()
```

Out[39]:
```
0   2018-08-25
1   2018-08-25
2   2018-08-25
3   2018-08-25
4   2018-08-25
Name: DATE, dtype: datetime64[ns]
```

This has the equivalent behavior for this particular dataset as when we skipped the `format` argument, since pandas was able to detect the format correctly, automatically.

Now let's actually change the whole dataframe's `DATE` to a datetime (skipping the format since we didn't actually need it here):

In [40]:
```
df['DATE'] = pd.to_datetime(df['DATE'])
df.head(2)
```

Out[40]:

| | CONTROL_AREA | UNIT | SCP | STATION | LINENAME | DIVISION | DATE | TIME | ENTRIES | |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 2018-08-25 | 00:00:00 | 6736067 | 000 |
| **1** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 2018-08-25 | 04:00:00 | 6736087 | 000 |

In [41]:
```
# Make a sample of rows so we can see various dates
date_sample = df['DATE'].sample(n=10, random_state=0)
date_sample
```

Out[41]:
```
91546    2018-08-30
75472    2018-08-31
151239   2018-08-30
77535    2018-08-25
73591    2018-08-27
10204    2018-08-28
51946    2018-08-27
129569   2018-08-26
10655    2018-08-25
11334    2018-08-30
Name: DATE, dtype: datetime64[ns]
```

## Applying Datetime Methods

Now that we have converted the `DATE` field to a datetime object we can use some handy built-in methods.

For example, finding the name of the day of the week:

In [42]:
```
# .dt stores all the pandas datetime methods (only works for datetime columns)
date_sample.dt.day_name()
```

Out[42]:
```
91546     Thursday
75472       Friday
151239    Thursday
77535     Saturday
73591       Monday
10204      Tuesday
51946       Monday
129569      Sunday
10655     Saturday
11334     Thursday
Name: DATE, dtype: object
```

Or, rounding to the nearest 7 days:

In [43]:
```
date_sample.dt.round('7D')
```

Out[43]:
```
91546    2018-08-30
75472    2018-08-30
151239   2018-08-30
77535    2018-08-23
73591    2018-08-30
10204    2018-08-30
51946    2018-08-30
129569   2018-08-23
10655    2018-08-23
11334    2018-08-30
Name: DATE, dtype: datetime64[ns]
```

## Setting a New Index

It can also be helpful to set one of the columns as the index of the DataFrame, such as when graphing.

In [44]:
```
df = df.set_index('DATE')
df.head()
```

Out[44]:

| DATE | CONTROL_AREA | UNIT | SCP | STATION | LINENAME | DIVISION | TIME | ENTRIES | E |
|---|---|---|---|---|---|---|---|---|---|
| **2018-08-25** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 00:00:00 | 6736067 | 000228 |
| **2018-08-25** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 04:00:00 | 6736087 | 000228 |
| **2018-08-25** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08:00:00 | 6736105 | 000228 |
| **2018-08-25** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 12:00:00 | 6736180 | 000228 |
| **2018-08-25** | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 16:00:00 | 6736349 | 000228 |

Or the opposite, resetting the index so that the current index becomes a column and a new index is created:

In [45]:
```
df.reset_index()
```

Out[45]:

| | DATE | CONTROL_AREA | UNIT | SCP | STATION | LINENAME | DIVISION | TIME | ENTR |
|---|---|---|---|---|---|---|---|---|---|
| **0** | 2018-08-25 | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 00:00:00 | 6736 |
| **1** | 2018-08-25 | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 04:00:00 | 6736 |
| **2** | 2018-08-25 | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 08:00:00 | 6736 |
| **3** | 2018-08-25 | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 12:00:00 | 6736 |
| **4** | 2018-08-25 | A002 | R051 | 02-00-00 | 59 ST | NQR456W | Brooklyn–Manhattan Transit Corporation | 16:00:00 | 6736 |
| **...** | ... | ... | ... | ... | ... | ... | ... | ... | |
| **197620** | 2018-08-31 | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 05:00:00 | 5 |
| **197621** | 2018-08-31 | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 09:00:00 | 5 |
| **197622** | 2018-08-31 | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 13:00:00 | 5 |
| **197623** | 2018-08-31 | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 17:00:00 | 5 |
| **197624** | 2018-08-31 | TRAM2 | R469 | 00-05-01 | RIT-ROOSEVELT | R | Roosevelt Island Tram | 21:00:00 | 5 |

197625 rows × 11 columns

# Summary

We've seen in this lesson the differences between Pandas (Series and DataFrames) and base Python (Dictionaries and Lists) data types. Then we walked through transforming the values in a pandas Series, modifying the columns of a pandas DataFrame, and finally modifying the DataFrame index.