Module 4 Part 3: Modifying a DataFrame, Data Aggregation and Grouping, Case Studies

This module consists of 3 parts:

- **Part 1** - Object-Oriented Programming with Python and Additional Python Functions
- **Part 2** - Introduction to pandas
- **Part 3** - Modifying a DataFrame, data aggregation and grouping, Case Studies

Each part is provided in a separate notebook file. It is recommended that you follow the order of the notebooks.

In this part we will work through the following topics:

1. We will start with a discussion on how to modify DataFrames — including how to add columns, delete rows, and remove entire columns of data.
2. We will discuss applying functions to columns of data and sorting DataFrames.
3. We will cover data aggregation and grouping. We will learn the split-apply-combine concept and how it is implemented in pandas. We will also discuss how to work with multi-index DataFrames.
4. Finally, to practice all the concepts we have learned in this module, we will work with two data sets in the Case Studies section.

Reading and Resources

The majority of the notebook content borrows from the recommended readings. We invite you to further supplement this notebook with the following recommended texts:

McKinney, W. (2017). *Python for Data Analysis*. O-Reilly: Boston

Table of Contents

Modifying DataFrames

Adding and removing columns, updating values

We will use the same data as we used in Part 2, federal support to all Canadian Provinces and Territories :

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

prov_support = pd.read_csv('pandas_ex1.csv',
                           sep=',',
                           skiprows=1,   # skipping one row
                           header=None,  # Set to None, since we are skipping the f
                           names=['province_name','province','2016','2017','2018'],
                           index_col='province')   # use column 'province' as the i

prov_support
```

Out[1]:

province	province_name	2016	2017	2018
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201
NB	New Brunswick	2741	2814	2956
QC	Quebec	21372	22720	23749
ON	Ontario	21347	21101	21420
MB	Manitoba	3531	3675	3965
SK	Saskatchewan	1565	1613	1673
AB	Alberta	5772	5943	6157
BC	British Columbia	6482	6680	6925
YT	Yukon	946	973	1006
NT	Northwest Territories	1281	1294	1319
NU	Nunavut	1539	1583	1634

We can add or remove columns from a `DataFrame`.

For example, we can create a new column, `'2016-2018 change'`. This column will be calculated based on two other columns, `'2016'` and `'2018'`. We are calculating a simple difference between two numbers, not a percent change. This is called a **vectorized operation** in `pandas`.

In [2]:
```
prov_support['2016-2018 change'] = prov_support['2018'] - prov_support['2016']
```

In [3]:
```
prov_support
```

Out[3]:

province	province_name	2016	2017	2018	2016-2018 change
NL	Newfoundland and Labrador	724	734	750	26
PE	Prince Edward Island	584	601	638	54
NS	Nova Scotia	3060	3138	3201	141
NB	New Brunswick	2741	2814	2956	215
QC	Quebec	21372	22720	23749	2377
ON	Ontario	21347	21101	21420	73
MB	Manitoba	3531	3675	3965	434
SK	Saskatchewan	1565	1613	1673	108
AB	Alberta	5772	5943	6157	385
BC	British Columbia	6482	6680	6925	443
YT	Yukon	946	973	1006	60
NT	Northwest Territories	1281	1294	1319	38
NU	Nunavut	1539	1583	1634	95

The new column is added to the `DataFrame` (always added on the right).

If we want to update a single value within the `DataFrame`, we can use a simple assignment operator. For example, if we want to update the `'2017'` value for Ontario from `21101` to `22222`, we can do the following:

In [4]:
```
prov_support.loc['ON', '2017'] = 22222
prov_support
```

Out[4]:

province	province_name	2016	2017	2018	2016-2018 change
NL	Newfoundland and Labrador	724	734	750	26
PE	Prince Edward Island	584	601	638	54
NS	Nova Scotia	3060	3138	3201	141
NB	New Brunswick	2741	2814	2956	215
QC	Quebec	21372	22720	23749	2377
ON	Ontario	21347	22222	21420	73
MB	Manitoba	3531	3675	3965	434
SK	Saskatchewan	1565	1613	1673	108
AB	Alberta	5772	5943	6157	385
BC	British Columbia	6482	6680	6925	443
YT	Yukon	946	973	1006	60
NT	Northwest Territories	1281	1294	1319	38
NU	Nunavut	1539	1583	1634	95

As you can see, the value is updated in the `DataFrame`.

We can get the same result by using the `at[]` field which provides access to a single value. We will now change the value of `22222` back to `21101` for Ontario in 2017:

In [5]:
```
prov_support.at['ON', '2017']   # get value of a cell
```

Out[5]: 22222

In [6]:
```
# set value of a data point back to 21101:

prov_support.at['ON', '2017'] = 21101
prov_support
```

Out[6]:

province	province_name	2016	2017	2018	2016-2018 change
NL	Newfoundland and Labrador	724	734	750	26
PE	Prince Edward Island	584	601	638	54
NS	Nova Scotia	3060	3138	3201	141
NB	New Brunswick	2741	2814	2956	215
QC	Quebec	21372	22720	23749	2377
ON	Ontario	21347	21101	21420	73
MB	Manitoba	3531	3675	3965	434
SK	Saskatchewan	1565	1613	1673	108
AB	Alberta	5772	5943	6157	385
BC	British Columbia	6482	6680	6925	443
YT	Yukon	946	973	1006	60
NT	Northwest Territories	1281	1294	1319	38
NU	Nunavut	1539	1583	1634	95

Data can be deleted from any axis, rows or columns. For example, if we need to delete a column, we need to use function `drop()` with parameter `axis=1` or `axis='columns'`:

In [7]:
```
# Deleting column '2016':

prov_support.drop('2016', axis = 1)
```

Out[7]:

province	province_name	2017	2018	2016-2018 change
NL	Newfoundland and Labrador	734	750	26
PE	Prince Edward Island	601	638	54
NS	Nova Scotia	3138	3201	141
NB	New Brunswick	2814	2956	215
QC	Quebec	22720	23749	2377
ON	Ontario	21101	21420	73
MB	Manitoba	3675	3965	434
SK	Saskatchewan	1613	1673	108
AB	Alberta	5943	6157	385
BC	British Columbia	6680	6925	443
YT	Yukon	973	1006	60
NT	Northwest Territories	1294	1319	38
NU	Nunavut	1583	1634	95

To remove some of the rows, we can use the same `drop()` function and specify a list of row labels that need to be deleted. Please note that parameter `axis = 0` is the default parameter, so we don't need to specify that we want to delete rows as `pandas` will do this by default:

In [8]:

```
# Deleting Ontario and Quebec from the DataFrame:

prov_support.drop(['ON', 'QC'])
```

Out[8]:

province	province_name	2016	2017	2018	2016-2018 change
NL	Newfoundland and Labrador	724	734	750	26
PE	Prince Edward Island	584	601	638	54
NS	Nova Scotia	3060	3138	3201	141
NB	New Brunswick	2741	2814	2956	215
MB	Manitoba	3531	3675	3965	434
SK	Saskatchewan	1565	1613	1673	108
AB	Alberta	5772	5943	6157	385
BC	British Columbia	6482	6680	6925	443
YT	Yukon	946	973	1006	60
NT	Northwest Territories	1281	1294	1319	38
NU	Nunavut	1539	1583	1634	95

When then `drop()` function is called, pandas creates a new DataFrame object, the original DataFrame is not modified. If we need to modify the original DataFrame, then we need to set a parameter `inplace = True`:

In [9]:
```
prov_support.drop('2016', axis = 1, inplace = True)
```

In [10]:
```
prov_support
```

Out[10]:

province	province_name	2017	2018	2016-2018 change
NL	Newfoundland and Labrador	734	750	26
PE	Prince Edward Island	601	638	54
NS	Nova Scotia	3138	3201	141
NB	New Brunswick	2814	2956	215
QC	Quebec	22720	23749	2377
ON	Ontario	21101	21420	73
MB	Manitoba	3675	3965	434
SK	Saskatchewan	1613	1673	108
AB	Alberta	5943	6157	385
BC	British Columbia	6680	6925	443
YT	Yukon	973	1006	60
NT	Northwest Territories	1294	1319	38
NU	Nunavut	1583	1634	95

Applying Functions to Columns

Sometimes we need to perform an operation on the column(s) of a `pandas` `DataFrame` which is not vectorizable. We might have a custom function that we need to apply to each row of data for one or more columns. To illustrate, we will create our own function to calculate percent change between the years 2017 and 2018.

In [2]:
```
# Here is the DataFrame that we start with:

prov_support
```

Out[2]:

province	province_name	2016	2017	2018
NL	Newfoundland and Labrador	724	734	750
PE	Prince Edward Island	584	601	638
NS	Nova Scotia	3060	3138	3201
NB	New Brunswick	2741	2814	2956
QC	Quebec	21372	22720	23749
ON	Ontario	21347	21101	21420
MB	Manitoba	3531	3675	3965
SK	Saskatchewan	1565	1613	1673
AB	Alberta	5772	5943	6157
BC	British Columbia	6482	6680	6925
YT	Yukon	946	973	1006
NT	Northwest Territories	1281	1294	1319
NU	Nunavut	1539	1583	1634

Defining a custom function to calculate the percent change:

In [12]:
```
def percent_change(years):
    yr2017, yr2018 = years
    return (yr2018 - yr2017)/yr2017 * 100
```

The `percent_change()` function takes one parameter which is expected to be a pair of data values and assigns them to the `yr2017` and `yr2018` variables. The function returns the result of the calculation.

In [13]:
```
prov_support[['2017', '2018']].apply(percent_change, axis = 1)
```

Out[13]:
```
province
NL    2.179837
PE    6.156406
NS    2.007648
NB    5.046198
QC    4.529049
ON    1.511777
MB    7.891156
SK    3.719777
AB    3.600875
BC    3.667665
YT    3.391572
NT    1.931994
NU    3.221731
dtype: float64
```

Breaking down the line of code above:

1. `prov_support[['2017', '2018']]` : we want to apply the function only to the columns that are going to be used in the calculation: `'2017'` and `'2018'` .
2. These two columns are passed as a list to an index operation, hence we end up with a new `DataFrame` which will contain these two columns only.
3. Next, we apply the function `percent_change()` to the new `DataFrame` , column-wise. In order to specify that we want the `apply()` function to work on columns, we use parameter `axis = 1` . This means that `apply()` will take one value per column and pass the list of the values as an argument called `years` to the `percent_change()` function.

NOTE: The default value for the `axis` parameter is `axis = 0` . If this parameter is used, the `apply()` function will take an entire column and will use all values from the column as the `years` argument for the function `percent_change()` — which is not what we want to do. The function will return an error. That's why we need to explicitly specify what axis we want to use, `axis = 1` .

In [14]:
```
# We can create a new column with the values calculated above and add it to the Dat

prov_support['per_change'] = prov_support[['2017', '2018']].apply(percent_change, a
prov_support
```

Out[14]:

province	province_name	2017	2018	2016-2018 change	per_change
NL	Newfoundland and Labrador	734	750	26	2.179837
PE	Prince Edward Island	601	638	54	6.156406
NS	Nova Scotia	3138	3201	141	2.007648
NB	New Brunswick	2814	2956	215	5.046198
QC	Quebec	22720	23749	2377	4.529049
ON	Ontario	21101	21420	73	1.511777
MB	Manitoba	3675	3965	434	7.891156
SK	Saskatchewan	1613	1673	108	3.719777
AB	Alberta	5943	6157	385	3.600875
BC	British Columbia	6680	6925	443	3.667665
YT	Yukon	973	1006	60	3.391572
NT	Northwest Territories	1294	1319	38	1.931994
NU	Nunavut	1583	1634	95	3.221731

Pandas has a function `applymap()` which will apply another function to every element in the selected DataFrame. For example, formatting all number columns as floating point numbers. We can also use the `lambda` function for this operation:

In [15]:
```
prov_support.loc[:,'2017':'per_change'].applymap(lambda x: '%.2f' % x)
```

Out[15]:

province	2017	2018	2016-2018 change	per_change
NL	734.00	750.00	26.00	2.18
PE	601.00	638.00	54.00	6.16
NS	3138.00	3201.00	141.00	2.01
NB	2814.00	2956.00	215.00	5.05
QC	22720.00	23749.00	2377.00	4.53
ON	21101.00	21420.00	73.00	1.51
MB	3675.00	3965.00	434.00	7.89
SK	1613.00	1673.00	108.00	3.72
AB	5943.00	6157.00	385.00	3.60
BC	6680.00	6925.00	443.00	3.67
YT	973.00	1006.00	60.00	3.39
NT	1294.00	1319.00	38.00	1.93
NU	1583.00	1634.00	95.00	3.22

However, if we want to operate on a single column only, `per_change`, we won't be able to use the `applymap()` method. Slicing the `DataFrame` and selecting a single column will return a `Series` object. `Series` has a method `map()` for applying element-wise functions to a `Series`:

In [16]:
```
prov_support['per_change'].map(lambda x: '%.2f' % x)
```

Out[16]:
```
province
NL    2.18
PE    6.16
NS    2.01
NB    5.05
QC    4.53
ON    1.51
MB    7.89
SK    3.72
AB    3.60
BC    3.67
YT    3.39
NT    1.93
NU    3.22
Name: per_change, dtype: object
```

Sorting a DataFrame

We can sort values in the `DataFrame` by values in a column or by index. Let's look at both scenarios.

For example, we can sort values in the `DataFrame` by the province full name, which is the `province_name` column. For this operation, `pandas` has the `sort_values()` method:

In [17]:
```
# the values are sorted in alphabetical order
prov_support.sort_values('province_name')
```

Out[17]:

	province_name	2017	2018	2016-2018 change	per_change
province					
AB	Alberta	5943	6157	385	3.600875
BC	British Columbia	6680	6925	443	3.667665
MB	Manitoba	3675	3965	434	7.891156
NB	New Brunswick	2814	2956	215	5.046198
NL	Newfoundland and Labrador	734	750	26	2.179837
NT	Northwest Territories	1294	1319	38	1.931994
NS	Nova Scotia	3138	3201	141	2.007648
NU	Nunavut	1583	1634	95	3.221731
ON	Ontario	21101	21420	73	1.511777
PE	Prince Edward Island	601	638	54	6.156406
QC	Quebec	22720	23749	2377	4.529049
SK	Saskatchewan	1613	1673	108	3.719777
YT	Yukon	973	1006	60	3.391572

In [18]:
```
# using the 'ascending' parameter we can control sort order
prov_support.sort_values('province_name', ascending=False)
```

Out[18]:

province	province_name	2017	2018	2016-2018 change	per_change
YT	Yukon	973	1006	60	3.391572
SK	Saskatchewan	1613	1673	108	3.719777
QC	Quebec	22720	23749	2377	4.529049
PE	Prince Edward Island	601	638	54	6.156406
ON	Ontario	21101	21420	73	1.511777
NU	Nunavut	1583	1634	95	3.221731
NS	Nova Scotia	3138	3201	141	2.007648
NT	Northwest Territories	1294	1319	38	1.931994
NL	Newfoundland and Labrador	734	750	26	2.179837
NB	New Brunswick	2814	2956	215	5.046198
MB	Manitoba	3675	3965	434	7.891156
BC	British Columbia	6680	6925	443	3.667665
AB	Alberta	5943	6157	385	3.600875

We can sort by the values in other columns. For example, we can use the `per_change` column and sort values in descending order so that the provinces with the highest percent change in financial support appear at the top of the table:

In [19]:
```
prov_support.sort_values('per_change', ascending=False)
```

Out[19]:

province	province_name	2017	2018	2016-2018 change	per_change
MB	Manitoba	3675	3965	434	7.891156
PE	Prince Edward Island	601	638	54	6.156406
NB	New Brunswick	2814	2956	215	5.046198
QC	Quebec	22720	23749	2377	4.529049
SK	Saskatchewan	1613	1673	108	3.719777
BC	British Columbia	6680	6925	443	3.667665
AB	Alberta	5943	6157	385	3.600875
YT	Yukon	973	1006	60	3.391572
NU	Nunavut	1583	1634	95	3.221731
NL	Newfoundland and Labrador	734	750	26	2.179837
NS	Nova Scotia	3138	3201	141	2.007648
NT	Northwest Territories	1294	1319	38	1.931994
ON	Ontario	21101	21420	73	1.511777

Function `sort_index()` will sort a `DataFrame` by index. The default behaviour is `axis=0` which means that the `DataFrame` will be sorted by the rows index, in our case by the province abbreviation in ascending order:

In [20]:

```
prov_support.sort_index()
```

Out[20]:

province	province_name	2017	2018	2016-2018 change	per_change
AB	Alberta	5943	6157	385	3.600875
BC	British Columbia	6680	6925	443	3.667665
MB	Manitoba	3675	3965	434	7.891156
NB	New Brunswick	2814	2956	215	5.046198
NL	Newfoundland and Labrador	734	750	26	2.179837
NS	Nova Scotia	3138	3201	141	2.007648
NT	Northwest Territories	1294	1319	38	1.931994
NU	Nunavut	1583	1634	95	3.221731
ON	Ontario	21101	21420	73	1.511777
PE	Prince Edward Island	601	638	54	6.156406
QC	Quebec	22720	23749	2377	4.529049
SK	Saskatchewan	1613	1673	108	3.719777
YT	Yukon	973	1006	60	3.391572

When setting parameter `axis=1` for method `sort_index()`, the `DataFrame` will be sorted by column index, in lexicographical order.

In [21]:
```
prov_support.sort_index(axis=1)
```

Out[21]:

province	2016-2018 change	2017	2018	per_change	province_name
NL	26	734	750	2.179837	Newfoundland and Labrador
PE	54	601	638	6.156406	Prince Edward Island
NS	141	3138	3201	2.007648	Nova Scotia
NB	215	2814	2956	5.046198	New Brunswick
QC	2377	22720	23749	4.529049	Quebec
ON	73	21101	21420	1.511777	Ontario
MB	434	3675	3965	7.891156	Manitoba
SK	108	1613	1673	3.719777	Saskatchewan
AB	385	5943	6157	3.600875	Alberta
BC	443	6680	6925	3.667665	British Columbia
YT	60	973	1006	3.391572	Yukon
NT	38	1294	1319	1.931994	Northwest Territories
NU	95	1583	1634	3.221731	Nunavut

Data Aggregation and Grouping

The term **split-apply-combine**, which describes a "strategy, where you break up a big problem into manageable pieces, operate on each piece independently and then put all the pieces back together," was first introduced by Hadley Wickham in 2011 (Wickham, 2011). Hadley Wickman is an author of many popular packages for the R programming language.

The paper describes an implementation of the concept using one of R's packages, however `pandas` provides similar support for this concept. The image below helps to illustrate it (McKinney, 2017):

split-apply-combine

Picture 10. Split-apply-combine (McKinney, 2017).

1. First, the data is split into **groups** based on one or more **keys**. The keys for splitting are based on one of the axes of the DataFrame, either rows (`axis=0`) or columns (`axis=1`). In the image above, the `DataFrame` is split and the data is grouped by the key with the values `A` , `B` , and `C` .

2. The next step is **apply** when a function is *applied* to each group. The function can perform aggregate, transformation, or filtering operations. It is possible to specify different functions for different groups of data. In the example above, the data within the groups is summarized.

3. In the **combine** step, the results of the **apply** function(s) are merged into a result object which can be an `array` , `DataFrame` , or `Series` , depending on what operations were performed with the data.

Examples of the functions for the *apply* step include:

- **Aggregation**: compute a summary statistic for each group. For example, compute sum or mean, or compute size for each group.

- **Transformation**: perform group-specific computation(s). For example, standardize the data within a group, or replace NAs within a group based on a value calculated from the data within this group (could be a mean or sum or any other calculation).

- **Filtration**: discard some groups, based on a group-wise computation that evaluates to True or False. Some examples: filter out data based on a condition or based on the group sum / mean.

For more information, please refer to the pandas documentation.

GroupBy

To demonstrate `pandas` capabilities for grouping data, we will use one of the most well-known datasets, **Iris Flower Data Set** published originally in 1936. You can read about it on this Wikipedia page, and the dataset's home page in the UCI Machine Learning Repository.

Please download the data file from the repository https://archive.ics.uci.edu/ml/machine-learning-databases/iris/, file name: `iris.data`. The code in this notebook will assume that the data file is in the same folder as the Jupyter notebook. However, you can save the data file anywhere on your file system, and update the code below to point to the right folder.

Let's read the file and create a `DataFrame`. Before we do that, let's talk a little bit about the data. The description of the dataset can be found on its home page:

- the file contains data for 3 classes of irises (Iris Setosa, Iris Versicolour, and Iris Virginica), each class is of 50 instances.

- there are 5 attributes of the data in the set:

  - sepal length in cm
  - sepal width in cm
  - petal length in cm
  - petal width in cm
  - class
- the attribute `class` refers to a type of iris plant

- the attributes are separated by commas

- there is no header row in the data file, which means we need to add names for the columns

In [3]:
```
iris = pd.read_csv('iris.data', sep=',',
                   header=None,  # the data file does not contain a header
                   names=['sepal length','sepal width','petal length','petal width'
                  )
```

```
iris.head()
```

Out[3]:

	sepal length	sepal width	petal length	petal width	class
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Quick investigation of the data:

In [23]:
```
iris.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
sepal length    150 non-null float64
sepal width     150 non-null float64
petal length    150 non-null float64
petal width     150 non-null float64
class           150 non-null object
dtypes: float64(4), object(1)
memory usage: 5.9+ KB
```

As expected, there are:

- 5 columns of data,
- 4 columns contain numeric data,
- the column `class` is of the string data type,
- there are no Null values,
- and there are 3 classes of irises with 50 rows of data each for a total of 150 rows.

Since the `class` attribute contains repeating values, we can use it as a key to group the data by class. Suppose we want to compute means of all attributes for each class of the flowers which will become our **key**.

`pandas`' `groupby()` method is used to split the data into groups:

In [4]:
```
iris.groupby('class')
```

Out[4]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x000002C31C353510>
```

The result of the `groupby()` operation is a `GroupBy` object. No calculation has been performed yet — the DataFrame was simply split into 3 groups. By default, the `groupby()` groups on `axis=0`.

The `GroupBy` object has several attributes that we can examine. For example, we can validate which groups are contained within the object:

In [6]:
```
# For simplicity, let's create a new variable for the GroupBy object

iris_grouped = iris.groupby('class')
iris_grouped.groups
```

Out[6]:
```
{'Iris-setosa': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 3
9, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49], 'Iris-versicolor': [50, 51, 52, 53, 5
4, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74,
75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 9
5, 96, 97, 98, 99], 'Iris-virginica': [100, 101, 102, 103, 104, 105, 106, 107, 10
8, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124,
125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 14
1, 142, 143, 144, 145, 146, 147, 148, 149]}
```

As expected, there are 3 groups, by the 3 values of the key `class`.

If we need to look at one of the groups, we can use the `get_group()` method:

In [7]:
```
# This method of the GroupBy object will output the content of a single group

iris_grouped.get_group('Iris-versicolor')
```

Out[7]:

	sepal length	sepal width	petal length	petal width	class
50	7.0	3.2	4.7	1.4	Iris-versicolor
51	6.4	3.2	4.5	1.5	Iris-versicolor
52	6.9	3.1	4.9	1.5	Iris-versicolor
53	5.5	2.3	4.0	1.3	Iris-versicolor
54	6.5	2.8	4.6	1.5	Iris-versicolor
55	5.7	2.8	4.5	1.3	Iris-versicolor
56	6.3	3.3	4.7	1.6	Iris-versicolor
57	4.9	2.4	3.3	1.0	Iris-versicolor
58	6.6	2.9	4.6	1.3	Iris-versicolor
59	5.2	2.7	3.9	1.4	Iris-versicolor
60	5.0	2.0	3.5	1.0	Iris-versicolor
61	5.9	3.0	4.2	1.5	Iris-versicolor
62	6.0	2.2	4.0	1.0	Iris-versicolor
63	6.1	2.9	4.7	1.4	Iris-versicolor
64	5.6	2.9	3.6	1.3	Iris-versicolor
65	6.7	3.1	4.4	1.4	Iris-versicolor
66	5.6	3.0	4.5	1.5	Iris-versicolor
67	5.8	2.7	4.1	1.0	Iris-versicolor
68	6.2	2.2	4.5	1.5	Iris-versicolor
69	5.6	2.5	3.9	1.1	Iris-versicolor
70	5.9	3.2	4.8	1.8	Iris-versicolor
71	6.1	2.8	4.0	1.3	Iris-versicolor
72	6.3	2.5	4.9	1.5	Iris-versicolor
73	6.1	2.8	4.7	1.2	Iris-versicolor
74	6.4	2.9	4.3	1.3	Iris-versicolor
75	6.6	3.0	4.4	1.4	Iris-versicolor
76	6.8	2.8	4.8	1.4	Iris-versicolor
77	6.7	3.0	5.0	1.7	Iris-versicolor
78	6.0	2.9	4.5	1.5	Iris-versicolor
79	5.7	2.6	3.5	1.0	Iris-versicolor

	sepal length	sepal width	petal length	petal width	class
80	5.5	2.4	3.8	1.1	Iris-versicolor
81	5.5	2.4	3.7	1.0	Iris-versicolor
82	5.8	2.7	3.9	1.2	Iris-versicolor
83	6.0	2.7	5.1	1.6	Iris-versicolor
84	5.4	3.0	4.5	1.5	Iris-versicolor
85	6.0	3.4	4.5	1.6	Iris-versicolor
86	6.7	3.1	4.7	1.5	Iris-versicolor
87	6.3	2.3	4.4	1.3	Iris-versicolor
88	5.6	3.0	4.1	1.3	Iris-versicolor
89	5.5	2.5	4.0	1.3	Iris-versicolor
90	5.5	2.6	4.4	1.2	Iris-versicolor
91	6.1	3.0	4.6	1.4	Iris-versicolor
92	5.8	2.6	4.0	1.2	Iris-versicolor
93	5.0	2.3	3.3	1.0	Iris-versicolor
94	5.6	2.7	4.2	1.3	Iris-versicolor
95	5.7	3.0	4.2	1.2	Iris-versicolor
96	5.7	2.9	4.2	1.3	Iris-versicolor
97	6.2	2.9	4.3	1.3	Iris-versicolor
98	5.1	2.5	3.0	1.1	Iris-versicolor
99	5.7	2.8	4.1	1.3	Iris-versicolor

Aggregation

Now we can select an aggregate function to apply to each group in the `GroupBy` object. For example, we can compute the mean for each column within each group:

In [8]:
```
iris_grouped.mean()
```

Out[8]:

	sepal length	sepal width	petal length	petal width
class				
Iris-setosa	5.006	3.418	1.464	0.244
Iris-versicolor	5.936	2.770	4.260	1.326
Iris-virginica	6.588	2.974	5.552	2.026

Or, we can compute the `median()` :

In [9]:
```
iris_grouped.median()
```

Out[9]:

	sepal length	sepal width	petal length	petal width
class				
Iris-setosa	5.0	3.4	1.50	0.2
Iris-versicolor	5.9	2.8	4.35	1.3
Iris-virginica	6.5	3.0	5.55	2.0

The result of the aggregate operation is a new `DataFrame` object with the **key** as an index, which in our case is `'class'` .

We might also want to use a custom function to calculate values based on the grouped data.

For example, we can create a function that will select and return a record from the group for the flower which will have the longest petal length:

In [10]:
```
def longest_petal(g):
    return g.loc[g['petal length'].idxmax()]

iris_grouped.apply(longest_petal)
```

Out[10]:

	sepal length	sepal width	petal length	petal width	class
class					
Iris-setosa	4.8	3.4	1.9	0.2	Iris-setosa
Iris-versicolor	6.0	2.7	5.1	1.6	Iris-versicolor
Iris-virginica	7.7	2.6	6.9	2.3	Iris-virginica

We can select a particular column from the original `DataFrame` while grouping:

In [30]:
```
iris.groupby('class')['petal length'].mean()
```

Out[30]:
```
class
Iris-setosa        1.464
Iris-versicolor    4.260
Iris-virginica     5.552
Name: petal length, dtype: float64
```

This returns a `Series` object.

Pandas GroupBy also allows us to compute multiple aggregate functions. For example, we can calculate `min()`, `max()` and `mean()` for `'petal length'` and `'sepal length'`, for each group of flowers based on the flower class:

In [31]:
```
iris.groupby('class')[['petal length','sepal length']].aggregate(['min', np.mean, m
```

Out[31]:

	petal length			sepal length		
	min	**mean**	**max**	**min**	**mean**	**max**
class						
Iris-setosa	1.0	1.464	1.9	4.3	5.006	5.8
Iris-versicolor	3.0	4.260	5.1	4.9	5.936	7.0
Iris-virginica	4.5	5.552	6.9	4.9	6.588	7.9

In [32]:
```
# This syntax will return the same result:

iris.groupby('class')[['petal length','sepal length']].aggregate(['min', 'mean', 'm
```

Out[32]:

	petal length			sepal length		
	min	**mean**	**max**	**min**	**mean**	**max**
class						
Iris-setosa	1.0	1.464	1.9	4.3	5.006	5.8
Iris-versicolor	3.0	4.260	5.1	4.9	5.936	7.0
Iris-virginica	4.5	5.552	6.9	4.9	6.588	7.9

Grouping Multi-Index DataFrame

DataFrames can be split / grouped by multiple indexes. Let's use a simple `DataFrame` with abstract index columns and random data for this exercise, just to demonstrate the concept. Later in this module we will apply these concepts to the 311 New York dataset.

In [17]:
```
# Creating sample DataFrame to demonstrate the concepts

import numpy as np
```

```
df = pd.DataFrame({'A' : ['foo', 'bar', 'foo', 'bar',
                          'foo', 'bar', 'foo', 'foo'],
                   'B' : ['one', 'one', 'two', 'three',
                          'two', 'two', 'one', 'three'],
                   'C' : np.random.randn(8),
                   'D' : np.random.randn(8)})
df
```

Out[17]:

	A	B	C	D
0	foo	one	0.505317	0.374228
1	bar	one	0.848091	-0.769484
2	foo	two	2.190786	-0.055230
3	bar	three	0.890705	-0.328650
4	foo	two	1.248886	1.675356
5	bar	two	0.556773	0.618285
6	foo	one	-0.202389	0.912904
7	foo	three	0.071095	-0.089613

We can use columns `A` and `B` as keys and split/group the `DataFrame` `df` by both keys:

In [18]:

```
grouped_df = df.groupby(['A', 'B'])
grouped_df.groups
```

Out[18]:

```
{('bar', 'one'): [1], ('bar', 'three'): [3], ('bar', 'two'): [5], ('foo', 'one'):
[0, 6], ('foo', 'three'): [7], ('foo', 'two'): [2, 4]}
```

It is interesting to note that when we group by multiple keys, we see the tuples of key values as group indexes. We can iterate over groups and print the group name and group data as follows:

In [19]:

```
for (key1, key2), group in df.groupby(['A', 'B']):
    print((key1, key2))
    print(group)
```

```
('bar', 'one')
     A    B         C         D
1  bar  one  0.848091 -0.769484
('bar', 'three')
     A      B         C        D
3  bar  three  0.890705 -0.32865
('bar', 'two')
     A    B         C         D
5  bar  two  0.556773  0.618285
('foo', 'one')
     A    B         C         D
0  foo  one  0.505317  0.374228
6  foo  one -0.202389  0.912904
('foo', 'three')
     A      B         C         D
7  foo  three  0.071095 -0.089613
('foo', 'two')
     A    B         C         D
2  foo  two  2.190786 -0.055230
4  foo  two  1.248886  1.675356
```

And now we can calculate the means of the grouped data:

In [21]:
```
grouped_df.mean()
```

Out[21]:

A	B	C	D
bar	**one**	0.848091	-0.769484
	three	0.890705	-0.328650
	two	0.556773	0.618285
foo	**one**	0.151464	0.643566
	three	0.071095	-0.089613
	two	1.719836	0.810063

The resulting object is a `DataFrame` with a hierarchical index consisting of the unique pairs of keys.

In [15]:
```
# Other methods are also supported, for example, max() or min(), or count(), etc.:

grouped_df.count()
```

Out[15]:

		C	D
A	**B**		
bar	**one**	1	1
	three	1	1
	two	1	1
foo	**one**	2	2
	three	1	1
	two	2	2

If the `DataFrame` has a hierarchical index, we can group by one of the levels of the hierarchy. In order to demonstrate this scenario, we will update the `df` `DataFrame` by setting the index to be a two-level `MultiIndex` :

In [23]:
```
df.set_index(['A','B'], inplace=True)
df
```

Out[23]:

		C	D
A	**B**		
foo	**one**	0.505317	0.374228
bar	**one**	0.848091	-0.769484
foo	**two**	2.190786	-0.055230
bar	**three**	0.890705	-0.328650
foo	**two**	1.248886	1.675356
bar	**two**	0.556773	0.618285
foo	**one**	-0.202389	0.912904
	three	0.071095	-0.089613

In [24]:
```
df.index  # to confirm that we have a DataFrame with MultiIndex
```

Out[24]:
```
MultiIndex([('foo',   'one'),
            ('bar',   'one'),
            ('foo',   'two'),
            ('bar', 'three'),
            ('foo',   'two'),
            ('bar',   'two'),
            ('foo',   'one'),
            ('foo', 'three')],
           names=['A', 'B'])
```

Now we can group by one or both levels:

In [25]:
```
df.groupby(level=['A','B']).count()
```

Out[25]:

A	B	C	D
bar	one	1	1
	three	1	1
	two	1	1
foo	one	2	2
	three	1	1
	two	2	2

Understanding stack() and unstack()

In the example above, when we grouped the `DataFrame` by 2 keys and calculated the average values for each of the group, the resulting `DataFrame` is a `MultiIndex` object. Let's quickly review how we can operate with this data.

Here is the `DataFrame` we had:

In [26]:
```
df_means = grouped_df.mean()
df_means
```

Out[26]:

A	B	C	D
bar	one	0.848091	-0.769484
	three	0.890705	-0.328650
	two	0.556773	0.618285
foo	one	0.151464	0.643566
	three	0.071095	-0.089613
	two	1.719836	0.810063

In [27]:
```
df_means.index
```

Out[27]:
```
MultiIndex([('bar',   'one'),
            ('bar', 'three'),
            ('bar',   'two'),
            ('foo',   'one'),
            ('foo', 'three'),
            ('foo',   'two')],
           names=['A', 'B'])
```

How do we access data in `MultiIndex` DataFrames? Let's review. If we need to select only one subgroup of data, for example, `bar` or `C`, we can do it as follows:

In [28]:
```
df_means['C']
```

Out[28]:
```
A    B    
bar  one      0.848091
     three    0.890705
     two      0.556773
foo  one      0.151464
     three    0.071095
     two      1.719836
Name: C, dtype: float64
```

In [29]:
```
df_means.loc['bar']
```

Out[29]:

B	C	D
one	0.848091	-0.769484
three	0.890705	-0.328650
two	0.556773	0.618285

When we select column `'C'`, we get back a `Series` with the same `MultiIndex` as the original `DataFrame`.

When we select a group of rows with the `'bar'` index, we get back a `DataFrame` which has a single-level index.

Pandas has two functions for reshaping the `DataFrame` and changing the index: `stack()` and `unstack()`. The `stack()` function "compresses" a level in the DataFrame's columns to produce either:

- A `Series`, in the case of a simple column Index.
- A `DataFrame`, in the case of `MultiIndex` columns.

If the columns have a `MultiIndex`, we can choose which level to stack. The stacked level becomes the new lowest level in a `MultiIndex` on the columns. For more information, please refer to the pandas documentation: https://pandas.pydata.org/pandas-docs/stable/reshaping.html.

Let's take the `DataFrame` `df_means` and demonstrate:

In [30]:
```
df_means
```

Out[30]:

A	B	C	D
bar	**one**	0.848091	-0.769484
	three	0.890705	-0.328650
	two	0.556773	0.618285
foo	**one**	0.151464	0.643566
	three	0.071095	-0.089613
	two	1.719836	0.810063

In [31]:
```
df_stacked = df_means.stack()
df_stacked
```

Out[31]:
```
A    B
bar  one    C    0.848091
            D   -0.769484
     three  C    0.890705
            D   -0.328650
     two    C    0.556773
            D    0.618285
foo  one    C    0.151464
            D    0.643566
     three  C    0.071095
            D   -0.089613
     two    C    1.719836
            D    0.810063
dtype: float64
```

We got a `Series` object with a 3-level `MultiIndex`:

Index Level	Index Values Available
1	`['bar', 'foo']`
2	`['one', 'three', 'two']`
3	`['C', 'D']`

In [32]:
```
df_stacked.index
```

Out[32]:
```
MultiIndex([('bar',   'one', 'C'),
            ('bar',   'one', 'D'),
            ('bar', 'three', 'C'),
            ('bar', 'three', 'D'),
            ('bar',   'two', 'C'),
            ('bar',   'two', 'D'),
            ('foo',   'one', 'C'),
            ('foo',   'one', 'D'),
            ('foo', 'three', 'C'),
            ('foo', 'three', 'D'),
            ('foo',   'two', 'C'),
            ('foo',   'two', 'D')],
           names=['A', 'B', None])
```

Now we can use the `unstack()` function to reverse the result of `stack()` or create a different object. When using the `unstack()` function, we can specify which level we want to "unstack". By default, this function "unstacks" the last level:

In [33]:
```
df_stacked.unstack()  # and we are back to the original DataFrame
```

Out[33]:

A	B	C	D
bar	one	0.848091	-0.769484
	three	0.890705	-0.328650
	two	0.556773	0.618285
foo	one	0.151464	0.643566
	three	0.071095	-0.089613
	two	1.719836	0.810063

We get the same result if we explicitly specify the level we want the unstack operation to be performed on. In this case, level 2:

In [34]:
```
df_stacked.unstack(level=2)
```

Out[34]:

A	B	C	D
bar	one	0.848091	-0.769484
	three	0.890705	-0.328650
	two	0.556773	0.618285
foo	one	0.151464	0.643566
	three	0.071095	-0.089613
	two	1.719836	0.810063

In [35]:

```
# Level 1 "unstack"

df_stacked.unstack(level=1)
```

Out[35]:

A	B	one	three	two
bar	C	0.848091	0.890705	0.556773
	D	-0.769484	-0.328650	0.618285
foo	C	0.151464	0.071095	1.719836
	D	0.643566	-0.089613	0.810063

In [36]:

```
# Level 0 "unstack"

df_stacked.unstack(level=0)
```

Out[36]:

B	A	bar	foo
one	C	0.848091	0.151464
	D	-0.769484	0.643566
three	C	0.890705	0.071095
	D	-0.328650	-0.089613
two	C	0.556773	1.719836
	D	0.618285	0.810063

Case Studies

Case Study 1 - Analysis of NYC 311 Service Requests

This case study is adopted from chapters 2 and 3 of Julia Evans' Pandas Cookbook. The original version you can find on GitHub.

In this analysis, we will look at the **311 Service Requests dataset** published by New York City. The current dataset includes more than 17 million rows of data and contains data from 2010 to the present. The data is updated daily. Here is the link to the homepage of the dataset: https://nycopendata.socrata.com/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9.

We will look into a small subset of this data: all complaints for the month of **April 2018**. File: `311_Service_Requests_APRIL2018.csv`. Make sure that the file is in the same folder as your notebook or update the path in the `read_csv()` function to include the correct path to the file. This dataset has almost 200,000 rows of data and 41 columns / attributes.

The question we will try to answer with the analysis of the 311 NYC data is - **What is the most common complaint type and what part of the city files the highest number of this type of complaint?**

In [37]:
```
complaints = pd.read_csv('311_Service_Requests_APRIL2018.csv')
complaints.head()
```

```
C:\Users\jverc\AppData\Local\Temp\ipykernel_27768\2749112017.py:1: DtypeWarning: Col
umns (8,17,31) have mixed types. Specify dtype option on import or set low_memory=Fa
lse.
  complaints = pd.read_csv('311_Service_Requests_APRIL2018.csv')
```

Out[37]:

	Unique Key	Created Date	Closed Date	Agency	Agency Name	Complaint Type	Descriptor	Lo
0	38837194	04/01/2018 12:00:00 AM	04/10/2018 12:00:00 AM	DOHMH	Department of Health and Mental Hygiene	Standing Water	Swimming Pool - Unmaintained	1-2 D
1	38837043	04/01/2018 12:00:00 AM	03/12/2018 12:00:00 AM	DOHMH	Department of Health and Mental Hygiene	Rodent	Condition Attracting Rodents	3+ B
2	38836824	04/01/2018 12:00:00 AM	04/09/2018 12:00:00 AM	DOHMH	Department of Health and Mental Hygiene	Rodent	Rat Sighting	Comr B
3	38836823	04/01/2018 12:00:00 AM	03/20/2018 12:00:00 AM	DOHMH	Department of Health and Mental Hygiene	Rodent	Rat Sighting	3+ B
4	38836788	04/01/2018 12:00:00 AM	04/04/2018 06:19:08 PM	DOHMH	Department of Health and Mental Hygiene	Unsanitary Animal Pvt Property	Dog	3+ Apa B

5 rows × 41 columns

NOTE: You will probably see an error "DtypeWarning: Columns (8,17,31) have mixed types". This means that pandas has encountered a problem while reading the data in the columns with index 8, 17 and 31. Most probably, these columns contain a mix of numbers and strings. We can check what these columns are and decide if we want to ignore the warning for now or if we will need to fix the data.

The list of all columns:

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

Out[38]:
```
Index(['Unique Key', 'Created Date', 'Closed Date', 'Agency', 'Agency Name',
       'Complaint Type', 'Descriptor', 'Location Type', 'Incident Zip',
       'Incident Address', 'Street Name', 'Cross Street 1', 'Cross Street 2',
       'Intersection Street 1', 'Intersection Street 2', 'Address Type',
       'City', 'Landmark', 'Facility Type', 'Status', 'Due Date',
       'Resolution Description', 'Resolution Action Updated Date',
       'Community Board', 'BBL', 'Borough', 'X Coordinate (State Plane)',
       'Y Coordinate (State Plane)', 'Open Data Channel Type',
       'Park Facility Name', 'Park Borough', 'Vehicle Type',
       'Taxi Company Borough', 'Taxi Pick Up Location', 'Bridge Highway Name',
       'Bridge Highway Direction', 'Road Ramp', 'Bridge Highway Segment',
       'Latitude', 'Longitude', 'Location'],
      dtype='object')
```

In [39]:
```
# and the columns in question are:

complaints.columns[[8,17,31]]
```

Out[39]:
```
Index(['Incident Zip', 'Landmark', 'Vehicle Type'], dtype='object')
```

For the purpose of our analysis, we can ignore these columns for now.

However, just to make sure that we are correct in our understanding of the warning, we can look into the column 8 data, `'Incident Zip'`:

In [55]:
```
# Unique values in this column:

complaints['Incident Zip'].unique()
```

Out[55]:
```
array([10312.0, 11217.0, 11224.0, 10025.0, 10454.0, 11204.0, 11226.0,
       11225.0, 10029.0, 11432.0, 10014.0, 11422.0, 10467.0, 11416.0,
       10463.0, 10016.0, 10026.0, 11356.0, 10301.0, 11235.0, 10037.0,
       11385.0, 11222.0, 11207.0, 10040.0, 10306.0, 10024.0, 11236.0,
       11205.0, 10022.0, 10468.0, 10023.0, 11238.0, 11435.0, 11213.0,
       10473.0, 11372.0, 10010.0, 10469.0, 11220.0, 11203.0, 11419.0,
       10002.0, 10452.0, 11234.0, 10453.0, 11001.0, 10032.0, 11378.0,
       11216.0, 11362.0, 11368.0, 10460.0, 10310.0, 11211.0, 11355.0,
       10459.0, 10457.0, 10465.0, 10455.0, 11214.0, 10456.0, 11223.0,
       10001.0, 10012.0, 10003.0, 10031.0, 11237.0, 11426.0, 11102.0,
       10011.0, 10305.0, 10018.0, 10017.0, 10314.0, 11232.0, 11210.0,
       10039.0, 10033.0, 11433.0, 11417.0, 10466.0, 11106.0, 10309.0,
       11358.0, 11360.0, 11209.0, nan, 10308.0, 10475.0, 11354.0, 10451.0,
       11208.0, 11105.0, 11413.0, 11229.0, 10458.0, 10034.0, 11219.0,
       10028.0, 11103.0, 10128.0, 11429.0, 11109.0, 11233.0, 11374.0,
       11375.0, 11379.0, 11423.0, 10027.0, 11414.0, 11692.0, 10470.0,
       11221.0, 10474.0, 10471.0, 11101.0, 11377.0, 10035.0, 10019.0,
       10030.0, 10013.0, 10009.0, 11215.0, 10472.0, 11693.0, 10462.0,
       11206.0, 11370.0, 11434.0, 11104.0, 11365.0, 11201.0, 11212.0,
       10304.0, 11373.0, 11004.0, 10036.0, 10065.0, 11357.0, 11691.0,
       11230.0, 11420.0, 10461.0, 11249.0, 11421.0, 10307.0, 11369.0,
       10005.0, 10038.0, 11228.0, 11364.0, 10464.0, 11231.0, 11361.0,
       11418.0, 10303.0, 10302.0, 11436.0, 10111.0, 10021.0, 11367.0,
       11411.0, 11694.0, 11415.0, 11363.0, 11218.0, 11366.0, 11427.0,
       10007.0, 10282.0, 11412.0, 11428.0, 10075.0, 11040.0, 0.0, 10020.0,
       10168.0, 10004.0, 11430.0, 7114.0, 11239.0, 10006.0, 10119.0,
       10279.0, 60179.0, 10120.0, 7020.0, 10178.0, 10165.0, 10280.0,
       10069.0, 10504.0, 10048.0, 10281.0, 83.0, 10271.0, 11697.0,
       10000.0, 10044.0, 10704.0, '11203', '10463', '10040', '10468',
       '11229', '10026', '11230', '11418', '10019', '10009', '11219',
       '11412', '10065', '11413', '10002', '11426', '10022', '10301',
       '10027', '10025', '11204', '10036', '10464', '10011', '11385',
       '11691', '10469', '10304', '11375', '10013', '11420', '10033',
       '11207', '11216', '10308', '11220', '10460', '11435', '11422',
       '11103', '11377', '11106', '10306', '10023', '11233', '11206',
       '11228', '11237', '10473', '11694', '10467', '11373', '10310',
       '11208', '11212', '11434', '11372', '10031', '10032', '11364',
       '10451', '11101', '10037', '10005', '11238', '10111', '10453',
       '10312', '11226', '10029', '10456', '11436', '11368', '10458',
       '10034', '10457', '11355', '11218', '10462', '10039', '10466',
       '11234', '11432', '11379', '11370', '11419', '11214', '10474',
       '11211', '11225', '11423', '11217', '11223', '10471', '10314',
       '10010', '11209', '10075', '11201', '11213', '10309', '11236',
       '11102', '10020', '11210', '11411', '10038', '11374', '11221',
       '11356', '10001', '11004', '11215', '10454', '11429', '11358',
       '10459', '11361', '11417', '10024', '10014', '11378', '11231',
       '10128', '11363', '11428', '11224', '11433', '11360', '11232',
       '10452', '11222', '11104', '10302', '11357', '10470', '11415',
       '10280', '11235', '11369', '11362', '11414', '11001', '11105',
       '11205', '10455', '10021', '11421', '11416', '10030', '11354',
       '10303', '10028', '10305', '10307', '10465', '11249', '10035',
       '10017', '10003', '10461', '10018', '10012', '11692', '11365',
       '11366', '11239', '10016', '11367', '11430', '10282', '10472',
       '10007', '11693', '10006', '10475', '10158', '11040', '11427',
       '10044', '07114', '112516', '10271', '11710', '11557', '10168',
       '10004', '10119', '10103', '10121', '10069', '10041', '*', '11735',
```

```
'10112', '11251', '11580', '11109', '11371', '98057', 74020.0,
10598.0, 10103.0, 14221.0, 14202.0, 10278.0, 10122.0, 10167.0,
7656.0, 10121.0, 10154.0, 10112.0, 23502.0, 11005.0, 55164.0,
11695.0, 10583.0, 10516.0, 10118.0, 10116.0, 10176.0, '10123',
'07607', '10118', '10278', '11202', '10000', '55164-0437', '0740',
'11709', '10105', '11005', '06901', '10169', '10155', '07013',
'07621', '10115', '90091', '10606', '00083', 11559.0, 11371.0,
7030.0, 10041.0, 11242.0, 7042.0, 6901.0, '10603', 'VARIES',
'11561', '10162', '00565', '10281', '10528', '07094', '07666',
'10045', '10173', 11247.0, 11548.0, 11727.0, 10107.0, 11757.0,
'11697', '11576-1502', '10279', '11553', '07514', '10176', '11548',
'10803', '210002', 7010.0, 10162.0, 11598.0, 1423.0, 10707.0,
10173.0, 10801.0, 10170.0, 10155.0, 10172.0, 11251.0, 98036.0,
10601.0, 11797.0, 7093.0, 14266.0, 11042.0, 11590.0, 94524.0,
11580.0, 12222.0, 7503.0, 7302.0, 98057.0, 10174.0, 10705.0],
dtype=object)
```

First of all, the `dtype` of this column is `object`, which tells us that pandas treats the data as strings. We see that there are indeed strings among the values, for example: `'VARIES'`, `'11576-1502'`, `'*'` and numbers that are formatted as strings `'11373'`. Some of the ZIP codes are integers and some are floating point numbers. When the column in the DataFrame contains the data of different types, pandas chooses the type that accomodates all data types, which in most cases is `object`. See the documentation for more details: pandas.DataFrame.dtypes. If we wanted to use this column for our analysis, we would need to clean this data and make sure that all values are of the same data type.

For the purpose of our exercise, however, we can ignore this column, because to identify the area of the city we will use the `Borough` attribute which is a descriptive name of the city area.

The first part of the question we are trying to answer: **What is the most common complaint type?** The complaint type is stored in the `'Complaint Type'` attribute. Let's look at the top 10 rows of data for this column:

In [56]:
```
complaints['Complaint Type'][:10]
```

Out[56]:
```
0                   Standing Water
1                           Rodent
2                           Rodent
3                           Rodent
4    Unsanitary Animal Pvt Property
5                   Standing Water
6                           Rodent
7                           Rodent
8                           Rodent
9                           Rodent
Name: Complaint Type, dtype: object
```

Combining this data with the area of the city, or `Borough` attribute:

In [57]:
```
complaints[['Complaint Type', 'Borough']][:10]
```

Out[57]:

	Complaint Type	Borough
0	Standing Water	STATEN ISLAND
1	Rodent	BROOKLYN
2	Rodent	BROOKLYN
3	Rodent	MANHATTAN
4	Unsanitary Animal Pvt Property	BRONX
5	Standing Water	BROOKLYN
6	Rodent	BROOKLYN
7	Rodent	BROOKLYN
8	Rodent	BROOKLYN
9	Rodent	MANHATTAN

We see that most of the complaints in the first 10 rows of data are of type `Rodent`, but is this the most common complaint type in the entire dataset? To answer this question, we will use the `value_counts()` method:

In [58]:
```
complaints['Complaint Type'].value_counts()
```

```
Out[58]: Noise - Residential                     17665
         HEAT/HOT WATER                          14822
         Request Large Bulky Item Collection     13817
         Illegal Parking                         11875
         Blocked Driveway                        10448
         Street Condition                        10020
         Street Light Condition                   6217
         Noise                                    6032
         UNSANITARY CONDITION                     5588
         Water System                             4407
         PAINT/PLASTER                            4178
         Noise - Street/Sidewalk                  4110
         Noise - Commercial                       3946
         PLUMBING                                 3553
         Sewer                                    3347
         Dirty Conditions                         3089
         WATER LEAK                               3075
         Derelict Vehicle                         3063
         Missed Collection (All Materials)        3061
         Sanitation Condition                     3010
         Traffic Signal Condition                 2998
         Derelict Vehicles                        2952
         Noise - Vehicle                          2581
         DOOR/WINDOW                              2568
         Rodent                                   2554
         General Construction/Plumbing            2491
         Sidewalk Condition                       2440
         Building/Use                             2352
         FLOORING/STAIRS                          1971
         Damaged Tree                             1922
                                                 ...
         Window Guard                                5
         Highway Sign - Damaged                      5
         Unsanitary Animal Facility                  5
         Plant                                       5
         Illegal Fireworks                           4
         Cooling Tower                               4
         Home Delivered Meal - Missed Delivery       4
         Ferry Permit                                4
         Public Toilet                               4
         Bereavement Support Group                   4
         Poison Ivy                                  3
         Mosquitoes                                  3
         Municipal Parking Facility                  3
         Advocate-Co-opCondo Abatement               3
         Lifeguard                                   3
         Parking Card                                2
         Legal Services Provider Complaint           2
         Research Questions                          2
         Transportation Provider Complaint           2
         Case Management Agency Complaint            2
         Highway Sign - Dangling                     2
         Advocate-Commercial Exemptions              1
         Tunnel Condition                            1
         Calorie Labeling                            1
         Home Care Provider Complaint                1
```

```
Elevator                                     1
X-Ray Machine/Equipment                      1
SNW                                          1
Advocate - Lien                              1
Bottled Water                                1
Name: Complaint Type, Length: 197, dtype: int64
```

Apparently, the most complaints are about Residential Noise, and the `Rodent` complaint type does not even make the top 10. What are the top 10 complaints?

In [59]:
```
complaints['Complaint Type'].value_counts()[:10]
```

Out[59]:
```
Noise - Residential                     17665
HEAT/HOT WATER                          14822
Request Large Bulky Item Collection     13817
Illegal Parking                         11875
Blocked Driveway                        10448
Street Condition                        10020
Street Light Condition                   6217
Noise                                    6032
UNSANITARY CONDITION                     5588
Water System                             4407
Name: Complaint Type, dtype: int64
```

`Noise - Residential` is at the top of the list. We can also see that there are multiple types of noise-related complaints, such as `Noise - Street/Sidewalk`, simply `Noise`, `Noise - Commercial` and others. We can select all types of noise:

In [40]:
```
# add case = False here
is_noise = complaints['Complaint Type'].str.contains('Noise')
noise_complaints = complaints[is_noise]
noise_complaints.head(5)
```

Out[40]:

	Unique Key	Created Date	Closed Date	Agency	Agency Name	Complaint Type	Descriptor	I
59	38829758	04/01/2018 12:00:05 AM	04/01/2018 06:40:07 AM	NYPD	New York City Police Department	Noise - Commercial	Loud Music/Party	Club/I
60	38826829	04/01/2018 12:00:08 AM	04/01/2018 03:16:18 AM	NYPD	New York City Police Department	Noise - Residential	Loud Music/Party	B
61	38827632	04/01/2018 12:00:28 AM	04/01/2018 04:29:11 AM	NYPD	New York City Police Department	Noise - Residential	Loud Music/Party	B
62	38829976	04/01/2018 12:00:34 AM	04/01/2018 12:26:35 AM	NYPD	New York City Police Department	Noise - Residential	Loud Music/Party	B
63	38831504	04/01/2018 12:00:36 AM	04/01/2018 06:30:04 AM	NYPD	New York City Police Department	Noise - Residential	Loud Music/Party	B

5 rows × 41 columns

This is the subset of the original DataFrame which only contains the data where the Complaint Type has "Noise" in the string.

Now we are ready to answer the second part of the question - **What area of the city has the most noise complaints?** For the area of the city we will use the `Borough` attribute.

In [41]:
```
noise_complaints['Borough'].value_counts()
```

Out[41]:
```
Borough
MANHATTAN        11911
BROOKLYN          9381
BRONX             6608
QUEENS            5751
STATEN ISLAND      854
Unspecified        172
Name: count, dtype: int64
```

And it's **Manhattan** closely followed by Brooklyn.

We can also calculate the percentage of noise complaints from the total number of complaints for each borough:

In [42]:
```
noise_complaint_counts = noise_complaints['Borough'].value_counts()

# Total number of complaints for each borough
complaint_counts = complaints['Borough'].value_counts()
```

In [43]:
```
noise_complaint_per = noise_complaint_counts / complaint_counts * 100

# sorting the result
noise_complaint_per.sort_values(ascending=False)
```

Out[43]:
```
Borough
MANHATTAN        31.077308
BRONX            18.925421
BROOKLYN         15.203228
QUEENS           11.672654
STATEN ISLAND     7.276135
Unspecified       4.436420
Name: count, dtype: float64
```

Case Study 2 - Analysis of the Toronto Bikeshare Data

In this case study we will look at the Bike Share Toronto Ridership Data. This is an open data set and can be found on Toronto Open Data catalogue page. For this exercise, we downloaded Bikeshare Ridership data for Q4 of 2016 - https://www.toronto.ca/ext/open_data/catalog/data_set_files/2016_Bike_Share_Toronto_Ridershi

The question we are trying to answer - **What day of the week do Torontonians bike the most?**

First, we will read the data file. Please **NOTE** that the file contains more than 200,000 rows of data and depending on your system capacity, it might take a couple of minutes for the file to load. While the file is being read, you will see a star appear `In [*]:` to the left of the next cell.

In [44]:
```
TObike = pd.read_excel('2016_Bike_Share_Toronto_Ridership_Q4.xlsx')
TObike.head(5)
```

Out[44]:

	trip_id	trip_start_time	trip_stop_time	trip_duration_seconds	from_station_name	to_sta
0	462305	2016-01-10 00:00:00	2016-01-10 00:07:00	394	Queens Quay W / Dan Leckie Way	Fort (
1	462306	2016-01-10 00:00:00	2016-01-10 00:09:00	533	Sherbourne St / Wellesley St	I
2	462307	2016-01-10 00:00:00	2016-01-10 00:07:00	383	Queens Quay W / Dan Leckie Way	Fort (
3	462308	2016-01-10 00:01:00	2016-01-10 00:27:00	1557	Cherry St / Distillery Ln	Fort (
4	462309	2016-01-10 00:01:00	2016-01-10 00:27:00	1547	Cherry St / Distillery Ln	Fort (

In [46]:

```
TObike.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 217569 entries, 0 to 217568
Data columns (total 7 columns):
 #   Column                 Non-Null Count   Dtype
---  ------                 --------------   -----
 0   trip_id                217569 non-null  int64
 1   trip_start_time        217569 non-null  object
 2   trip_stop_time         217569 non-null  object
 3   trip_duration_seconds  217569 non-null  int64
 4   from_station_name      217567 non-null  object
 5   to_station_name        217567 non-null  object
 6   user_type              217569 non-null  object
dtypes: int64(2), object(5)
memory usage: 11.6+ MB
```

The DataFrame has only 7 columns of data. For our exercise, we need to look at the dates when the trips were made. Let's look only at the start date/time of the trips, attribute `trip_start_time`. Since we have the `trip_duration_seconds` column, we don't really need `trip_stop_time`.

We can notice that the `trip_start_time` is a `datetime64[ns]` data type which is a special data type that Python and pandas use to store date and time data. We will discuss this data type and specifics of working with the DateTime objects in the module **Time Series**.

NOTE: You might see in the output of the `info()` method above that the attribute `trip_start_time` is of type `object` instead of `datetime64[ns]`. If this is the case, this means that pandas was not able to infer the data type of `trip_start_time` correctly while reading the file. In this case, we would usually spend some time investigating why the data was not read as expected. There could be multiple reasons; it might be that one or more values in the column are stored and/or interpreted as a string, not as a number. After you investigate the reason, you can use pandas `to_datetime()` method to convert the column `trip_start_time` to `datetime64[ns]`:

```
In [47]: TObike.trip_start_time = pd.to_datetime(TObike.trip_start_time)
```

```
In [48]: TObike.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 217569 entries, 0 to 217568
Data columns (total 7 columns):
 #   Column                 Non-Null Count   Dtype
---  ------                 --------------   -----
 0   trip_id                217569 non-null  int64
 1   trip_start_time        217569 non-null  datetime64[ns]
 2   trip_stop_time         217569 non-null  object
 3   trip_duration_seconds  217569 non-null  int64
 4   from_station_name      217567 non-null  object
 5   to_station_name        217567 non-null  object
 6   user_type              217569 non-null  object
dtypes: datetime64[ns](1), int64(2), object(4)
memory usage: 11.6+ MB
```

We can see that the datetime stamp in the DataFrame `TObike` has the date and time information, but we really need only the day of the week to answer the question of what day of the week is the busiest for Bikeshare Toronto. When the data is formatted as a DateTime object, we can use either the `dayofweek` method which will give us the numeric value of the day of the week or `weekday_name` for the full day of week name. We will add the day of the week name as a new column:

```
In [49]: TObike['weekday'] = TObike.trip_start_time.dt.day_name()
         TObike.head()
```

Out[49]:

	trip_id	trip_start_time	trip_stop_time	trip_duration_seconds	from_station_name	to_sta
0	462305	2016-01-10 00:00:00	2016-01-10 00:07:00	394	Queens Quay W / Dan Leckie Way	Fort (
1	462306	2016-01-10 00:00:00	2016-01-10 00:09:00	533	Sherbourne St / Wellesley St	I
2	462307	2016-01-10 00:00:00	2016-01-10 00:07:00	383	Queens Quay W / Dan Leckie Way	Fort (
3	462308	2016-01-10 00:01:00	2016-01-10 00:27:00	1557	Cherry St / Distillery Ln	Fort (
4	462309	2016-01-10 00:01:00	2016-01-10 00:27:00	1547	Cherry St / Distillery Ln	Fort (

The answer to the question can be interpreted either as "What is the day of the week with the longest duration of all trips during that day?" or "What is the day of the week with the most bike rides?" We will answer both questions.

Here is the answer to the question **"What is the day of the week with the longest duration of all trips during that day?"**:

In [69]:
```
TObike['trip_duration_seconds'].groupby(TObike['weekday']).aggregate(sum).sort_valu
```

Out[69]:
```
weekday
Sunday       25490178
Friday       24531608
Monday       24355459
Wednesday    24092014
Tuesday      23752117
Thursday     22613646
Saturday     21499246
Name: trip_duration_seconds, dtype: int64
```

And here is the answer to the question **"What is the day of the week with the most bike rides?"**

In [70]:
```
TObike['weekday'].value_counts()
```

Out[70]:
```
Friday       34101
Tuesday      33763
Monday       32913
Thursday     32069
Sunday       30750
Wednesday    29759
Saturday     24214
Name: weekday, dtype: int64
```

EXERCISE 4: Top 10 chocolate bars

For this short exercise, we will use the Chocolate Bar Rating dataset which we worked with in Part 2. We will read the data and rename data columns based on our prior knowledge of data:

In [50]:
```
chocolate = pd.read_csv('flavors_of_cacao.csv', na_values='\xa0')
chocolate.columns = ['company', 'bar_name', 'REF', 'review_date',
                     'cocoa_per', 'company_location', 'rating','bean_type', 'bean_o
chocolate.head(5)
```

Out[50]:

	company	bar_name	REF	review_date	cocoa_per	company_location	rating	bean_typ
0	A. Morin	Agua Grande	1876	2016	63%	France	3.75	Nal
1	A. Morin	Kpime	1676	2015	70%	France	2.75	Nal
2	A. Morin	Atsane	1676	2015	70%	France	3.00	Nal
3	A. Morin	Akata	1680	2015	70%	France	3.50	Nal
4	A. Morin	Quilla	1704	2015	70%	France	3.50	Nal

Task 1: Find the record with the highest chocolate rating. What company produces the chocolate bar and what country do the beans originate from?

In [54]:
```
chocolate['rating'].value_counts(ascending=False)
```

Out[54]:
```
rating
3.50    392
3.00    341
3.25    303
2.75    259
3.75    210
2.50    127
4.00     98
2.00     32
2.25     14
1.50     10
1.00      4
1.75      3
5.00      2
Name: count, dtype: int64
```

In [56]:
```
chocolate.loc[chocolate.rating.idxmax()]
```

Out[56]:
```
company                 Amedei
bar_name                 Chuao
REF                        111
review_date               2007
cocoa_per                  70%
company_location         Italy
rating                     5.0
bean_type           Trinitario
bean_origin          Venezuela
Name: 78, dtype: object
```

In [58]:
```
chocolate.loc[chocolate['rating']==5]
```

Out[58]:

	company	bar_name	REF	review_date	cocoa_per	company_location	rating	bean_typ
78	Amedei	Chuao	111	2007	70%	Italy	5.0	Trinitari
86	Amedei	Toscano Black	40	2006	70%	Italy	5.0	Blen

Task 2: Beans from what country are the most frequently used in the chocolate bars? We are looking for a top 10 countries of origin.

In [60]:
```
chocolate.bean_origin.value_counts().head(10)
```

Out[60]:
```
bean_origin
Venezuela             214
Ecuador               193
Peru                  165
Madagascar            145
Dominican Republic    141
Nicaragua              60
Brazil                 58
Bolivia                57
Belize                 49
Papua New Guinea       42
Name: count, dtype: int64
```

In [ ]:

In [ ]:

Task 3: Find the region of origin (column `bean_origin`) for the chocolate bars with the highest rating. Display the top 10 results, sorted by the rating and cocoa percent (column `cocoa_per`).

In [ ]:
```
chocolate.groupby(
```

In [ ]:

In [ ]:

Task 4:

- What countries are the top 10 chocolate producers, based on the variety of chocolate bars produced?
- And what countries produce the highest rated chocolate bars (top 10)?

In [ ]:

In [ ]:

In [ ]:

In [ ]:

Solutions

Task 1: For this task, all we need to do is to use the `idxmax()` function to find a record with the highest value within the column `rating`:

In [72]:
```
chocolate.loc[chocolate.rating.idxmax()]
```

Out[72]:
```
company                  Amedei
bar_name                 Chuao
REF                        111
review_date               2007
cocoa_per                  70%
company_location         Italy
rating                       5
bean_type           Trinitario
bean_origin          Venezuela
Name: 78, dtype: object
```

As you can see, the highest rating of `5` was given to the chocolate bar *Chuao* of the company *Amedei* with *70%* cacao made from a bean from *Venezuela*. However, we can see that there are actually two chocolate bars, both from the company *Amedei*, that have the highest rating of `5`:

In [73]:
```
chocolate.loc[chocolate['rating'] == 5]
```

Out[73]:

	company	bar_name	REF	review_date	cocoa_per	company_location	rating	bean_typ
78	Amedei	Chuao	111	2007	70%	Italy	5.0	Trinitari
86	Amedei	Toscano Black	40	2006	70%	Italy	5.0	Blen

We can retrieve both records using the `max()` function:

In [74]:
```
chocolate[chocolate.rating == chocolate.rating.max()]
```

Out[74]:

	company	bar_name	REF	review_date	cocoa_per	company_location	rating	bean_typ
78	Amedei	Chuao	111	2007	70%	Italy	5.0	Trinitari
86	Amedei	Toscano Black	40	2006	70%	Italy	5.0	Blen

In [75]:
```
# By adding `index` method, we get only the index for both records:
```

```
chocolate[chocolate.rating == chocolate.rating.max()].index
```

Out[75]:
```
Int64Index([78, 86], dtype='int64')
```

Task 2. Here, we are simply looking for the top 10 countries of origin which have the most rows of data in the dataset.

In [76]:
```
chocolate.bean_origin.value_counts().head(10)
```

Out[76]:
```
Venezuela             214
Ecuador               193
Peru                  165
Madagascar            145
Dominican Republic    141
Nicaragua              60
Brazil                 58
Bolivia                57
Belize                 49
Papua New Guinea       42
Name: bean_origin, dtype: int64
```

In [77]:
```
# We can also check if `bean_origin` has any null values:

chocolate.bean_origin.isnull().value_counts()
```

Out[77]:
```
False    1721
True       74
Name: bean_origin, dtype: int64
```

Yes, there are 74 missing values. We will learn techniques for dealing with missing data in the next module.

Task 3: Before we can start answering the question for this task, we need to fix the cocoa percentage column. Currently, the data in this column is formatted as `string`. We can validate it:

In [78]:
```
chocolate[['cocoa_per']].info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1795 entries, 0 to 1794
Data columns (total 1 columns):
cocoa_per    1795 non-null object
dtypes: object(1)
memory usage: 14.1+ KB
```

If we want to be able to calculate the mean and sort the results, we need to convert the data into numeric format. Here is a simple and quick way of achieving this:

In [79]:
```
chocolate['cocoa_per'] = chocolate['cocoa_per'].apply(lambda x: x.split('%')[0]).as
chocolate.head()
```

Out[79]:

	company	bar_name	REF	review_date	cocoa_per	company_location	rating	bean_typ
0	A. Morin	Agua Grande	1876	2016	63.0	France	3.75	NaI
1	A. Morin	Kpime	1676	2015	70.0	France	2.75	NaI
2	A. Morin	Atsane	1676	2015	70.0	France	3.00	NaI
3	A. Morin	Akata	1680	2015	70.0	France	3.50	NaI
4	A. Morin	Quilla	1704	2015	70.0	France	3.50	NaI

Now, we can do the following:

1. Group the data by the `bean_origin` column
2. Calculate the mean for rating and cocoa percentage
3. Sort the result
4. Display only first 10 rows

In [80]:

```
chocolate.groupby('bean_origin')[['rating','cocoa_per']]\
         .aggregate('mean')\
         .sort_values(by = ['rating','cocoa_per'],ascending=False)\
         .head(10)
```

Out[80]:

bean_origin	rating	cocoa_per
Guat., D.R., Peru, Mad., PNG	4.00	88.0
Dom. Rep., Madagascar	4.00	70.0
Gre., PNG, Haw., Haiti, Mad	4.00	70.0
Ven, Bolivia, D.R.	4.00	70.0
Venezuela, Java	4.00	70.0
Peru, Dom. Rep	4.00	67.0
Peru, Belize	3.75	75.0
Ven.,Ecu.,Peru,Nic.	3.75	75.0
DR, Ecuador, Peru	3.75	70.0
Dominican Rep., Bali	3.75	70.0

The result is very interesting. It shows us that the highest rated chocolate bars are made of a blend of beans from different countries. The result also shows that the highest rated bars have a fairly high cocoa percentage.

Task 4:

- What countries are the top 10 chocolate producers, based on the variety of chocolate bars produced?
- And what countries produce the highest rated chocolate bars (top 10)?

Task 4 Solution: The first part of this task is a simple aggregation of the number of rows in the dataset, based on the `company_location` column:

In [81]:
```
chocolate.company_location.value_counts().head(10)
```

Out[81]:
```
U.S.A.         764
France         156
Canada         125
U.K.            96
Italy           63
Ecuador         54
Australia       49
Belgium         40
Switzerland     38
Germany         35
Name: company_location, dtype: int64
```

As we can see, the largest producer of chocolate bars is the US followed by France, Canada, the UK and Italy, to name the top five countries.

To answer the second question, we need to group the data by `company_location` and calculate the max of rating (we also rounded the value):

In [82]:
```
chocolate.groupby('company_location')['rating'].max().round(2).sort_values(ascendin
```

Out[82]:
```
company_location
Italy         5.0
Brazil        4.0
Guatemala     4.0
Germany       4.0
France        4.0
Ecuador       4.0
Colombia      4.0
Sao Tome      4.0
Madagascar    4.0
Canada        4.0
Name: rating, dtype: float64
```

The highest rated chocolate bars are produced in Italy, Brazil and Guatemala, to name the top 3 locations. However, as you can see, there are multiple countries that achieved a 4.0 rating.

End of Module

You have reached the end of this module.

If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

When you are comfortable with the content, and have practiced to your satisfaction, you may proceed to any related assignments, and to the next module.

References

Wickham, Hadley (2011). The Split-Apply-Combine Strategy for Data Analysis. Journal of Statistical Software, 40(1).

McKinney, W. (2017). 10.1 GroupBy Mechanics. Python for Data Analysis (p 290). O'Reilly: Boston

Pandas 0.23.4 documentation (2018). Group By: split-apply-combine. Retrieved from https://pandas.pydata.org/pandas-docs/stable/groupby.html.

Wikipedia (2018). Iris flower data set. Retrieved from https://en.wikipedia.org/wiki/Iris_flower_data_set.

UCI Machine Learning Repository (1988). Iris Data Set. Retrieved from https://archive.ics.uci.edu/ml/datasets/Iris.

Pandas 0.23.4 documentation (2018). Reshaping and Pivot Tables. Retrieved from https://pandas.pydata.org/pandas-docs/stable/reshaping.html.

Evans, Julia (2017). Pandas cookbook. Retrieved from https://github.com/jvns/pandas-cookbook.

NYC Open Data (2018). 311 Service Requests from 2010 to Present. Retrieved from https://nycopendata.socrata.com/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9

Pandas 0.23.4 documentation. API Reference. (2018). pandas.DataFrame.dtypes. Retrieved from https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.dtypes.html

Toronto Open Data Catalogue. (2016). Bike Share Toronto Ridership Data. Retrieved from https://www.toronto.ca/city-government/data-research-maps/open-data/open-data-catalogue/#343faeaa-c920-57d6-6a75-969181b6cbde

Pandas 0.23.4 documentation. API Reference. (2018). pandas.to_datetime. Retrieved from https://pandas.pydata.org/pandas-docs/stable/generated/pandas.to_datetime.html