The Fundamentals of Pandas

Pandas is a data analysis library for python that enables powerful and easy ingress, manipulation, and storage of data. This notebook will cover some of the basics of using the Pandas library, for more extensive information, please visit the offical documentation here.

The notebook is broken into 5 sections for mastering the basics of Pandas:

Table of Contents

0) Pandas Setup

To install Pandas, simply open a terminal and run `pip install pandas` , or run the following cell to accomplish the same:

```
!pip install pandas
```

```
Requirement already satisfied: pandas in c:\python39\lib\site-packages (1.3.0)
Requirement already satisfied: numpy>=1.17.3 in c:\python39\lib\site-packages (from pandas) (1.20.1)
Requirement already satisfied: python-dateutil>=2.7.3 in c:\python39\lib\site-packages (from pandas) (2.8.1)
Requirement already satisfied: pytz>=2017.3 in c:\python39\lib\site-packages (from pandas) (2021.1)
Requirement already satisfied: six>=1.5 in c:\python39\lib\site-packages (from python-dateutil>=2.7.3->pandas) (1.10.0)


WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: You are using pip version 21.1.2; however, version 21.2.1 is available.
You should consider upgrading via the 'c:\python39\python.exe -m pip install --upgrade pip' command.
```

Next, we can import pandas for use in our script or notebook.

```
# Importing Pandas as a dependency. We alias the library to "pd" using the "as" operator to make it shorter to write in our code.
import pandas as pd
```

1) Pandas Data Structures

Pandas features two major data structures: Series and DataFrames.

Series

Series objects are indexed, one-dimensional arrays that behave similar to native python lists or dictionaries, as well as featuring several methods native to pandas.

```
# A native python list
my_data = ["a","b","c","d","e","f","h","i","j"]

# Conversion to Pandas Series object
my_series = pd.Series(my_data)

# OPTIONAL: Naming our series
my_series.name = "My Letters"

# Printing out our Series
my_series
```

```
0    a
1    b
2    c
3    d
4    e
5    f
6    h
7    i
8    j
Name: My Letters, dtype: object
```

On the Left, the series index is visible, starting from 0. To the right is the data we created in our list. At the bottom, we can see the optional name we added to the series, as well as the datatype of the data within the series.

While this may not seem terribly impressive compared to a normal python list, this allows us to then use special pandas methods with our data. Below, the `decribe()` method is used to easily return statistical analysis on a set of data.

```
my_data_2 = [24,52,62,22,24,22,22,28,32]
my_series_2 = pd.Series(my_data_2)
my_series_2
```

```
0    24
1    52
2    62
3    22
4    24
5    22
6    22
7    28
8    32
dtype: int64
```

```
my_series_2.describe()
```

```
count     9.000000
mean     32.000000
std      14.764823
min      22.000000
25%      22.000000
50%      24.000000
75%      32.000000
max      62.000000
dtype: float64
```

Note that the datatype for the "describe" result is different -that is because this is Series of it's own!

```
type(my_series_2.describe())
```

```
pandas.core.series.Series
```

You will also notice that the index for a Series does not always have to be numerical like a python list, but can can also be string based like a dictionary. Positions in the Series can thus accessed using the index like a list or dictionary.

```
print(f"Accessing an item like a list: {my_series_2[0]}.")
print(f"Accessing an item like a dict: {my_series_2.describe()['count']}.")
```

```
Accessing an item like a list: 24.
Accessing an item like a dict: 9.0.
```

DataFrames

The second major Pandas datatype is the DataFrame. A DataFrame is a tabular (table-like), 2-dimensional(i.e., rows and columns) object that is in many ways the central part of the pandas library. DataFrames are both indexed, like Series, and labeled. The index corresponds to the rows of the DataFrame while the labels correspond to the columns.

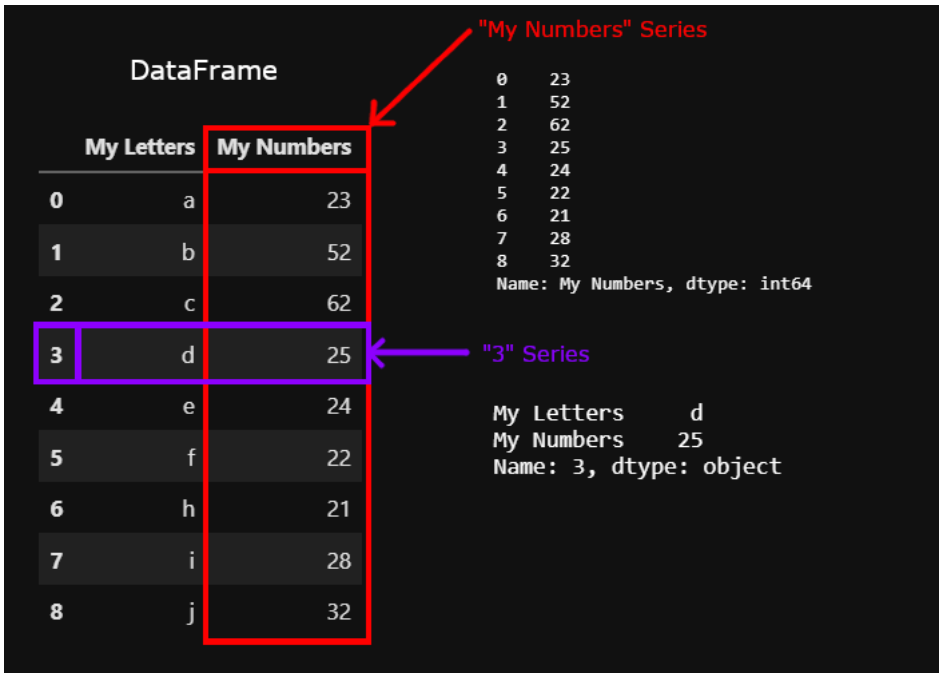
```
# Iniitalizing a DataFrame using the first Series we created.
df = pd.DataFrame(my_series)

# Adding the second Series to the DataFrame
df["My Numbers"] = my_series_2

# Viewing the DataFrame
df
```

	My Letters	My Numbers
0	a	24
1	b	52
2	c	62
3	d	22
4	e	24
5	f	22
6	h	22
7	i	28
8	j	32

The DataFrame features the same index as the Series used to create it's columns. Within the DataFrame, every row and column that makes it up is in fact its own Pandas Series. In other words, a DataFrame really is a matrix of intersecting Series!




```
df["My Numbers"].count()
```

```
9
```

```
df["My Numbers"].value_counts()
```

```
22    3
24    2
62    1
28    1
52    1
32    1
Name: My Numbers, dtype: int64
```

2) Creating DataFrames

There are numerous ways to create data frames conveniently built into Pandas depending on the structure of our target data. The following are just a few of the most common:

A list of dictionaries

This method is ideal for creating dictionaries from data generated within a loop, such as iterating over data from an API.

```
# Create a series of dictionaries
my_dict_1 = {"Letters": "a", "Num_1": 23, "Num_2": 2}
my_dict_2 = {"Letters": "b", "Num_1": 26, "Num_2": 3}
my_dict_3 = {"Letters": "c", "Num_1": 32, "Num_2": 2}
my_dict_4 = {"Letters": "d", "Num_1": 21, "Num_2": 4}

# Add all of these dictionaries to a list
my_list = [my_dict_1, my_dict_2, my_dict_3, my_dict_4]

# Then convert that into a DataFrame
df = pd.DataFrame(my_list)
df
```

	Letters	Num_1	Num_2
0	a	23	2
1	b	26	3

	Letters	Num_1	Num_2
2	c	32	2
3	d	21	4

A dictionary of lists

A dictionary of lists is a quick way to hand-write small data into a DataFrame.

```
# Create a series of lists
column_a = ["a","b","c","d"]
column_b = [23,26,32,21]
column_c = [2,3,2,4]

# Insert them into a dictionary
my_dict = {"Letters": column_a, "Num_1": column_b, "Num_2": column_c}

# Then convert that into a DataFrame
df = pd.DataFrame(my_dict)
df
```

	Letters	Num_1	Num_2
0	a	23	2
1	b	26	3
2	c	32	2
3	d	21	4

Reading from a SQL database

Data can be read directly from SQL databases using Pandas. For this example, we will use sqlalchemy to quickly build a SQL database from a SQLite file.

```
# Importing additional dependencies
!pip install --user sqlalchemy
from sqlalchemy import create_engine

# Path to SQLite file
database_path = "data_sources/Census_Data.sqlite"

# Creating the SQL database
engine = create_engine(f"sqlite:///{database_path}")

# Establisting a connection to our database
conn = engine.connect()

# Using pandas to read data out of SQL
census_data = pd.read_sql("SELECT * FROM Census_Data", conn)

# Because this DataFrame is so large, we will use the head() method to print out the top 5 entries.
census_data.head()
```

```
Requirement already satisfied: sqlalchemy in c:\users\farad\appdata\roaming\python\python39\site-packages (1.4.21)
Requirement already satisfied: greenlet!=0.4.17 in c:\python39\lib\site-packages (from sqlalchemy) (1.1.0)
```

```
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: Ignoring invalid distribution -ip (c:\python39\lib\site-packages)
WARNING: You are using pip version 21.1.2; however, version 21.2.1 is available.
You should consider upgrading via the 'c:\python39\python.exe -m pip install --upgrade pip' command.
```

	CityState	city	state	Population	White Population	Black Population	Native American Population	Asian Population	Hispanic Population	Education None
0	HOUSTON, TX	HOUSTON	TX	3061887	1775897	684416	11586	230549	1368287	54180
1	CHICAGO, IL	CHICAGO	IL	2702091	1318869	843633	7554	161478	785374	32800
2	BROOKLYN, NY	BROOKLYN	NY	2595259	1126111	870465	8744	297890	509243	48934
3	LOS ANGELES, CA	LOS ANGELES	CA	2426413	1068202	324842	15949	273829	1292382	62684
4	MIAMI, FL	MIAMI	FL	1820704	1361009	363514	2250	33144	1162711	27137

5 rows × 31 columns

Do not forget to shutdown the database when we are done with it!

```
engine.dispose()
```

Web Scraping a table

You can scrape table elements directly from HTML using Pandas.

```
# Defining the URL to scrape from
url = "https://en.wikipedia.org/wiki/List_of_the_highest_major_summits_of_North_America"

# Converting all table elements from the page into DataFrames. This method returns a list of DataFrames from the URL.
mountains_table_list = pd.read_html(url)

# Parsing through the list to find the table we want
mountains_table_list
```

```
[     Rank                            Mountain peak                Region  \
 0       1                Denali[a](Mount McKinley)                Alaska
 1       2                           Mount Logan[b]                 Yukon
 2       3          Pico de Orizaba[c](Citlaltépetl)        Puebla Veracruz
 3       4                      Mount Saint Elias[d]           Alaska Yukon
 4       5                        Popocatépetl[e][f]  México Morelos Puebla
 ..    ...                                       ...                   ...
 396   396                          Sierra Fría[my]         Aguascalientes
 397   398                     Hayford Peak[147][mz]                 Nevada
 398   399  Ulysses Mountain[na][nb](Mount Ulysses)       British Columbia
 399   400                       Eagle Peak[148][nc]             California
 400   401                    Sacajawea Peak[nd][ne]                 Oregon

               Mountain range  Elevation Prominence Isolation  \
 0                Alaska Range  20,310 ft  20,146 ft       NaN
 1      Saint Elias Mountains  19,551 ft  17,215 ft    387 mi
 2    Cordillera Neovolcanica  18,491 ft  16,148 ft       NaN
 3      Saint Elias Mountains  18,009 ft  11,250 ft   25.6 mi
 4    Cordillera Neovolcanica  17,749 ft   9,974 ft   88.8 mi
 ..                       ...        ...        ...       ...
 396  Sierra Madre Occidental   9,941 ft   1,640 ft  145.6 mi
 397              Sheep Range   9,924 ft   5,412 ft   33.8 mi
 398            Muskwa Ranges   9,921 ft   7,526 ft    271 mi
 399         Warner Mountains   9,895 ft   4,362 ft   87.4 mi
 400        Wallowa Mountains   9,843 ft   6,393 ft  125.5 mi

                                           Location
 0    .mw-parser-output .geo-default,.mw-parser-outp...
 1      60°34′02″N 140°24′20″W / 60.5671°N 140.4055°W
 2        19°01′50″N 97°16′11″W / 19.0305°N 97.2698°W
 3      60°17′34″N 140°55′51″W / 60.2927°N 140.9307°W
 4        19°01′21″N 98°37′40″W / 19.0225°N 98.6278°W
 ..                                              ...
 396    22°16′26″N 102°36′26″W / 22.2739°N 102.6073°W
 397    36°39′28″N 115°12′03″W / 36.6577°N 115.2008°W
 398    57°20′47″N 124°05′34″W / 57.3464°N 124.0928°W
 399    41°17′01″N 120°12′03″W / 41.2835°N 120.2007°W
 400    45°14′42″N 117°17′34″W / 45.2450°N 117.2929°W

 [401 rows x 8 columns],
   .mw-parser-output .navbar{display:inline;font-size:88%;font-weight:normal}.mw-parser-output .navbar-collapse{float:left;text-align:left}.mw-pa
 0                                  Sovereign states
 1                Dependencies andother territories

   .mw-parser-output .navbar{display:inline;font-size:88%;font-weight:normal}.mw-parser-output .navbar-collapse{float:left;text-align:left}.mw-pa
 0  Antigua and Barbuda Bahamas Barbados Belize Ca...
 1  Anguilla Aruba Bermuda Bonaire British Virgin ...
   vteThe 124 highest major summits of greater North America  \
 0  .mw-parser-output .div-col{margin-top:0.3em;co...

   vteThe 124 highest major summits of greater North America.1
 0  .mw-parser-output .div-col{margin-top:0.3em;co...           ,
   vteThe 100 most prominent summits of greater North America  \
 0  Denali Mount Logan Pico de Orizaba Mount Raini...

   vteThe 100 most prominent summits of greater North America.1
 0  Denali Mount Logan Pico de Orizaba Mount Raini...            ,
   vteThe 107 most isolated major summits of greater North America  \
 0  Denali Gunnbjørn Fjeld Pico de Orizaba Mount W...

   vteThe 107 most isolated major summits of greater North America.1
 0  Denali Gunnbjørn Fjeld Pico de Orizaba Mount W...                ,
   vteMountain peaks of North America  \
 0                  Sovereign states
 1  Dependencies andother territories

                vteMountain peaks of North America.1
 0  Antigua and Barbuda Bahamas Barbados Belize Ca...
 1  Anguilla Aruba Bermuda Bonaire British Virgin ...  ]
```

```
# It looks like the table we want is the second entry (Or now the first, as of the latest update) in the list of tables, so we will save it and pr
mountains_df = mountains_table_list[0]
mountains_df.head()
```

	Rank	Mountain peak	Region	Mountain range	Elevation	Prominence	Isolation	Location
0	1	Denali[a](Mount McKinley)	Alaska	Alaska Range	20,310 ft	20,146 ft	NaN	.mw-parser-output .geo-default,.mw-parser-outp...
1	2	Mount Logan[b]	Yukon	Saint Elias Mountains	19,551 ft	17,215 ft	387 mi	60°34′02″N 140°24′20″W / 60.5671°N 140.4055°W
2	3	Pico de Orizaba[c] (Citlaltépetl)	Puebla Veracruz	Cordillera Neovolcanica	18,491 ft	16,148 ft	NaN	19°01′50″N 97°16′11″W / 19.0305°N 97.2698°W
3	4	Mount Saint Elias[d]	Alaska Yukon	Saint Elias Mountains	18,009 ft	11,250 ft	25.6 mi	60°17′34″N 140°55′51″W / 60.2927°N 140.9307°W
4	5	Popocatépetl[e][f]	México Morelos Puebla	Cordillera Neovolcanica	17,749 ft	9,974 ft	88.8 mi	19°01′21″N 98°37′40″W / 19.0225°N 98.6278°W

Reading from a CSV

One of the most common ways to ingress data using Pandas, the humble CSV.

```
# Defining the CSV path
path = "data_sources/Census_Data.csv"

# Creating a DataFrame from the CSV
census_data = pd.read_csv(path)
census_data.head()
```

	CityState	city	state	Population	White Population	Black Population	Native American Population	Asian Population	Hispanic Population	Education None
0	HOUSTON, TX	HOUSTON	TX	3061887	1775897	684416	11586	230549	1368287	54180
1	CHICAGO, IL	CHICAGO	IL	2702091	1318869	843633	7554	161478	785374	32800
2	BROOKLYN, NY	BROOKLYN	NY	2595259	1126111	870465	8744	297890	509243	48934
3	LOS ANGELES, CA	LOS ANGELES	CA	2426413	1068202	324842	15949	273829	1292382	62684
4	MIAMI, FL	MIAMI	FL	1820704	1361009	363514	2250	33144	1162711	27137

5 rows × 31 columns

3) Reading data from a DataFrame

Now that we have our data in a DataFrame format, we need to be able to use it. The first thing we will want to learn to that end is how to read data back out!

Keys/Indexing

We can parse a DataFrame similar to how we might a dictionary, selecting the column by using its label as a key.

```
df['Letters']
```

```
0    a
1    b
2    c
3    d
Name: Letters, dtype: object
```

We can further drill down using the index.

```
df['Letters'][3]
```

```
'd'
```

Using iloc[ ]

Another option is to navigate the DataFrame entirely by numbers using `iloc[ ]`. We can retrieve a whole column:

```
df.iloc[:, 0] # Note the format here, [rows, columns]
```

```
0    a
1    b
2    c
3    d
Name: Letters, dtype: object
```

Or a single cell:

```
df.iloc[3, 0]
```

```
'd'
```

Using loc[ ]

The above techniques do not always work well because of the way indexes can be set to non-numerical data in DataFrames and can generally appear cluttered or as a mass of incomprehensible numbers. To get around this issue when we find ourselves in such situations, we can use the `loc[ ]` attribute.

```
df = df.set_index('Letters')
df
```

	Num_1	Num_2
Letters		
a	23	2
b	26	3
c	32	2
d	21	4

Because there is no numerical index for us to gauge what item we want with, instead we will use loc

```
df.loc["d"]
```

```
Num_1    21
Num_2     4
Name: d, dtype: int64
```

```
df.loc["d","Num_1"]
```

```
21
```

It is a good rule of thumb to keep in mind that `loc[ ]` is for selecting data by using words for the rows or columns, while `iloc[ ]` is for selecting data using numerical position.

Conditional views

What if want to view data that only matches certain criteria? For this sort of data parsing, we will want to use a conditional view.

```
# To start, let's clean up the mountains_df we created earlier.
# You can ignore this cell unless you just want to see an example of lambda functions in action.


# ~~~~~~~~~~~~~~~~ IGNORE THIS CELL ~~~~~~~~~~~~~~~~ #


# Set the index for the mountains DataFrame to the rank column
mountains_df = mountains_df.set_index('Rank')

# Use lambda functions to convert the Prominence, Elevation, and Isolation to numerical datatypes
def convert_ht(x):
    height = x.replace(",","").replace("\xa0ft","")
    return int(height)

def convert_mi(x):
    if isinstance(x, str):
        x = float(x.replace(",","").replace("\xa0mi",""))
    return x

def remove_bad_space(x):
    return x.replace("\xa0"," ")

mountains_df['Region'] = mountains_df['Region'].apply(lambda x:remove_bad_space(x))
mountains_df['Elevation'] = mountains_df['Elevation'].apply(lambda x:convert_ht(x))
mountains_df['Prominence'] = mountains_df['Prominence'].apply(lambda x:convert_ht(x))
mountains_df['Isolation'] = mountains_df['Isolation'].apply(lambda x:convert_mi(x))
mountains_df.head() # Much cleaner looking!
```

	Mountain peak	Region	Mountain range	Elevation	Prominence	Isolation	Location
Rank							
1	Denali[a](Mount McKinley)	Alaska	Alaska Range	20310	20146	NaN	.mw-parser-output .geo-default,.mw-parser-outp...
2	Mount Logan[b]	Yukon	Saint Elias Mountains	19551	17215	387.0	60°34′02″N 140°24′20″W / 60.5671°N 140.4055°W
3	Pico de Orizaba[c] (Citlaltépetl)	Puebla Veracruz	Cordillera Neovolcanica	18491	16148	NaN	19°01′50″N 97°16′11″W / 19.0305°N 97.2698°W
4	Mount Saint Elias[d]	Alaska Yukon	Saint Elias Mountains	18009	11250	25.6	60°17′34″N 140°55′51″W / 60.2927°N 140.9307°W
5	Popocatépetl[e][f]	México Morelos Puebla	Cordillera Neovolcanica	17749	9974	88.8	19°01′21″N 98°37′40″W / 19.0225°N 98.6278°W

Let's get started with conditional views with an example using only one condition. We will try to view all the mountains where the region is Alaska. The first step will be to create a boolean Series, a Series object that is "True" for each row where the region is Alaska and "False" for anything else.

```
# Create a Boolean Series
mountains_df['Region'] == "Alaska"
```

```
Rank
1       True
2      False
3      False
4      False
5      False
       ...
396    False
398    False
399    False
400    False
401    False
Name: Region, Length: 401, dtype: bool
```

Next, we can use this boolean Series as a key. Note that the entire code for the boolean Series is placed between the brackets of `mountains_df[ ]`.

```
# Use Boolean Series as a key to output data
mountains_df[mountains_df['Region'] == "Alaska"].head()
```

	Mountain peak	Region	Mountain range	Elevation	Prominence	Isolation	Location
Rank							
1	Denali[a](Mount McKinley)	Alaska	Alaska Range	20310	20146	NaN	.mw-parser-output .geo-default,.mw-parser-outp...
6	Mount Foraker[g]	Alaska	Alaska Range	17400	7250	14.27	62°57′37″N 151°23′59″W / 62.9604°N 151.3998°W
10	Mount Bona[k]	Alaska	Saint Elias Mountains	16550	6900	49.70	61°23′08″N 141°44′58″W / 61.3856°N 141.7495°W
12	Mount Blackburn[7][m]	Alaska	Wrangell Mountains	16390	11640	60.70	61°43′50″N 143°24′11″W / 61.7305°N 143.4031°W
13	Mount Sanford	Alaska	Wrangell Mountains	16237	7687	40.30	62°12′48″N 144°07′45″W / 62.2132°N 144.1292°W

We can also parse a DataFrame using multiple conditions. Each condition must be wrapped in parantheses `( )`, then separated with the appropriate operator. Unlike regular python, here we use:

`&` for `and`

`|` for `or`

```
mountains_df[(mountains_df['Region'] == "Alaska") & (mountains_df['Elevation'] > 15000)]
```

	Mountain peak	Region	Mountain range	Elevation	Prominence	Isolation	Location
Rank							
1	Denali[a](Mount McKinley)	Alaska	Alaska Range	20310	20146	NaN	.mw-parser-output .geo-default,.mw-parser-outp...
6	Mount Foraker[g]	Alaska	Alaska Range	17400	7250	14.27	62°57′37″N 151°23′59″W / 62.9604°N 151.3998°W
10	Mount Bona[k]	Alaska	Saint Elias Mountains	16550	6900	49.70	61°23′08″N 141°44′58″W / 61.3856°N 141.7495°W
12	Mount Blackburn[7][m]	Alaska	Wrangell Mountains	16390	11640	60.70	61°43′50″N 143°24′11″W / 61.7305°N 143.4031°W

	Mountain peak	Region	Mountain range	Elevation	Prominence	Isolation	Location
Rank							
13	Mount Sanford	Alaska	Wrangell Mountains	16237	7687	40.30	62°12′48″N 144°07′45″W / 62.2132°N 144.1292°W

4) Data Manipulation in Pandas

Now that we know how to view our data, we can begin manipulating it.

Drop

To remove unnecessary data elements, we can use the `drop()` method.

```
# Removing the Location column
mountains_df.drop(columns="Location",inplace=True)
#mountains_df_1 = mountains_df.drop(columns="Location")
```

DropNA

We can remove data elements from our DataFrame that contain empty cells using the `dropna()` method.

```
# We can use .info() to see what columns have null values
mountains_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 401 entries, 1 to 401
Data columns (total 6 columns):
Mountain peak     401 non-null object
Region            401 non-null object
Mountain range    401 non-null object
Elevation         401 non-null int64
Prominence        401 non-null int64
Isolation         395 non-null float64
dtypes: float64(1), int64(2), object(3)
memory usage: 21.9+ KB
```

```
# Using .drop() to remove rows with empty cells.
mountains_df.dropna(how="any").head()
```

	Mountain peak	Region	Mountain range	Elevation	Prominence	Isolation
Rank						
2	Mount Logan[b]	Yukon	Saint Elias Mountains	19551	17215	387.00
4	Mount Saint Elias[d]	Alaska Yukon	Saint Elias Mountains	18009	11250	25.60
5	Popocatépetl[e][f]	México Morelos Puebla	Cordillera Neovolcanica	17749	9974	88.80
6	Mount Foraker[g]	Alaska	Alaska Range	17400	7250	14.27
7	Mount Lucania[h][i]	Yukon	Saint Elias Mountains	17257	10105	26.70

The `how="any"` argument means that we will drop a row if **ANY** of the cells in that row are empty. Alternatively, we can specify `how="all"` to only drop rows where **ALL** the cells in that row are empty. This second version can be very useful for cleaning up poorly sourced or formatted data, such as CSVs with unnecessary empty rows.

Duplicated

The `duplicated( )` method will display duplicate data from within our DataFrame.

```
# Creating a DataFrame with duplicate data.
duplicate_df = pd.DataFrame({
    "a": [14,14,23,45,67,32],
    "b": [22,22,23,39,55,22],
    "c": ["w","w","w","x","y","z"]
})
duplicate_df
```

	a	b	c
0	14	22	w
1	14	22	w
2	23	23	w
3	45	39	x
4	67	55	y
5	32	22	z

```
duplicate_df[duplicate_df.duplicated(keep=False)]
```

	a	b	c
0	14	22	w
1	14	22	w

Drop Duplicates

The `drop_duplicates( )` method is an efficient way to remove an duplicate data from your DataFrame. The "keep" parameter features three possible values: first, last, and False.

```
# Using the drop_duplicate method. The default value for the "keep" parameter is first.
duplicate_df.drop_duplicates(subset="b")
```

	a	b	c
0	14	22	w
2	23	23	w
3	45	39	x
4	67	55	y

```
# Specifiying the keep="first" parameter. This keeps the first instance of duplicated data.
duplicate_df.drop_duplicates(keep="first")
```

	a	b	c
0	14	22	w
2	23	23	w
3	45	39	x
4	67	55	y
5	32	22	z

```
# Specifiying the keep="last" parameter. This keeps the last instance of duplicated data.
duplicate_df.drop_duplicates(keep="last")
```

	a	b	c
1	14	22	w
2	23	23	w
3	45	39	x
4	67	55	y
5	32	22	z

```
# Specifiying the keep=False parameter. This will drop all duplicated data, including first and last instances.
duplicate_df.drop_duplicates(keep=False)
```

	a	b	c
2	23	23	w
3	45	39	x
4	67	55	y
5	32	22	z

At and lat

The `at[ ]` and `iat[ ]` are similiar to `loc[ ]` and `iloc[ ]` , but instead of viewing the data, they allow us to manipulate, or change, it directly.

```
df
```

	Num_1	Num_2
Letters		
a	23	2
b	26	3
c	32	2
d	21	4

```
df.at["a","Num_2"] = 5
df
```

	Num_1	Num_2
Letters		
a	23	5
b	26	3
c	32	2
d	21	4

```
df.iat[0,1] = 2
df
```

	Num_1	Num_2
Letters		
a	23	2

	Num_1	Num_2
Letters		
b	26	3
c	32	2
d	21	4

Append

Append is a method for combining two DataFrames to create a stack.

```
df
```

	Num_1	Num_2
Letters		
a	23	2
b	26	3
c	32	2
d	21	4

```
my_dict = {"Letters": ["e","f","g"], "Num_1": [20,23,24], "Num_2": [2,1,3]}
df2 = pd.DataFrame(my_dict).set_index("Letters")
df2
```

	Num_1	Num_2
Letters		
e	20	2
f	23	1
g	24	3

```
df3 = df.append(df2)
df3
```

	Num_1	Num_2
Letters		
a	23	2
b	26	3
c	32	2
d	21	4
e	20	2
f	23	1
g	24	3

Join

Join also combines DataFrames, but merges them along the lateral dimension.

```
my_dict = {"Letters": ["a","b","c"], "Num_1": [14,13,14], "Num_2": [7,10,13]}
df = pd.DataFrame(my_dict).set_index("Letters")
df
```

	Num_1	Num_2
Letters		
a	14	7
b	13	10
c	14	13

```
my_dict = {"Letters": ["a","b","c"], "Num_3": [20,23,24], "Num_4": [2,1,3]}
df2 = pd.DataFrame(my_dict).set_index("Letters")
df2
```

	Num_3	Num_4
Letters		
a	20	2
b	23	1
c	24	3

```
df3_join = df.join(df2)
df3_join
```

	Num_1	Num_2	Num_3	Num_4
Letters				
a	14	7	20	2
b	13	10	23	1
c	14	13	24	3

```
df3["Special Column just for Ranjani"] = [20,23,1,1,24,1,1]
df3
```

	Num_1	Num_2	Special Column just for Ranjani
Letters			
a	23	2	20
b	26	3	23
c	32	2	1
d	21	4	1
e	20	2	24
f	23	1	1
g	24	3	1

```
df2
```

	Num_3	Num_4

Letters	Num_3	Num_4

Letters		
a	20	2
b	23	1
c	24	3

```
pd.merge(df3, df2, left_on="Special Column just for Ranjani", right_on="Num_3", how="right")
```

	Num_1	Num_2	Special Column just for Ranjani	Num_3	Num_4
0	23	2	20	20	2
1	26	3	23	23	1
2	20	2	24	24	3

GroupBy

```
mountains_df
```

	Mountain peak	Region	Mountain range	Elevation	Prominence	Isolation
Rank						
1	Denali[a](Mount McKinley)	Alaska	Alaska Range	20310	20146	NaN
2	Mount Logan[b]	Yukon	Saint Elias Mountains	19551	17215	387.0
3	Pico de Orizaba[c](Citlaltépetl)	Puebla Veracruz	Cordillera Neovolcanica	18491	16148	NaN
4	Mount Saint Elias[d]	Alaska Yukon	Saint Elias Mountains	18009	11250	25.6
5	Popocatépetl[e][f]	México Morelos Puebla	Cordillera Neovolcanica	17749	9974	88.8
...	...	...	...	...	...	...
396	Sierra Fría[my]	Aguascalientes	Sierra Madre Occidental	9941	1640	145.6
398	Hayford Peak[147][mz]	Nevada	Sheep Range	9924	5412	33.8
399	Ulysses Mountain[na][nb](Mount Ulysses)	British Columbia	Muskwa Ranges	9921	7526	271.0
400	Eagle Peak[148][nc]	California	Warner Mountains	9895	4362	87.4
401	Sacajawea Peak[nd][ne]	Oregon	Wallowa Mountains	9843	6393	125.5

401 rows × 6 columns

```
mgbdf = mountains_df.groupby("Region").agg(["mean","min","max","std"])
mgbdf.head(10)
```

	Elevation				Prominence				Isolation	
	mean	**min**	**max**	**std**	**mean**	**min**	**max**	**std**	**mean**	**min**
Region										
Aguascalientes	9941.000000	9941	9941	NaN	1640.000000	1640	1640	NaN	145.600000	145.60
Alaska	12823.886792	10016	20310	1898.682850	4923.792453	1650	20146	3419.844792	21.714615	2.26

	Elevation				Prominence				Isolation	
	mean	min	max	std	mean	min	max	std	mean	min
Region										
Alaska British Columbia	12742.666667	10016	15325	2657.441313	6837.000000	2979	12995	5389.579204	51.953333	5.46
Alaska Yukon	15058.000000	13760	18009	1709.690177	6810.600000	1950	11250	3489.060877	15.620000	2.25
Alberta	11471.555556	10879	12247	475.059236	4864.111111	2438	6670	1423.999600	15.918889	4.26
Alberta British Columbia	11800.333333	11263	12274	508.498115	6516.000000	4938	7779	1446.457397	72.300000	30.60
Arizona	11590.000000	10724	12637	969.258995	5702.333333	4728	6340	857.113956	160.800000	82.40
Baja California	10154.000000	10154	10154	NaN	6972.000000	6972	6972	NaN	208.000000	208.00
British Columbia	11052.428571	9921	13186	874.006078	6374.821429	1716	10791	2145.744213	60.371786	1.78
California	12596.794118	9895	14505	1417.333749	3806.529412	1676	10080	2422.423270	34.232121	3.09

```
mountains_df.groupby("Elevation").mean()
```

	Prominence	Isolation
Elevation		
9843	6393.0	125.50
9895	4362.0	87.40
9921	7526.0	271.00
9924	5412.0	33.80
9941	3945.0	143.15
...	...	...
17749	9974.0	88.80
18009	11250.0	25.60
18491	16148.0	NaN
19551	17215.0	387.00
20310	20146.0	NaN

377 rows × 2 columns

```
mgbdf["Elevation"].head()
```

	mean	min	max	std
Region				
Aguascalientes	9941.000000	9941	9941	NaN
Alaska	12823.886792	10016	20310	1898.682850
Alaska British Columbia	12742.666667	10016	15325	2657.441313
Alaska Yukon	15058.000000	13760	18009	1709.690177

	mean	min	max	std
Region				
Alberta	11471.555556	10879	12247	475.059236

```
mgbdf["Elevation"].loc['Alaska British Columbia', "std"]
```

```
2657.441313243499
```

Binning Data

Binning data is another important technique that is often used in conjuntion with with groupby. Binning involves cutting numerical data into pre-defined ranges for discrete analysis. Other terms for binning are dicretization, bucketing, or quantization. Binning is very useful for processing continuous data, meaning data with numbers that can all unique and do not fit into categories.

```
# Cutting the DataFrame into bins
bins = [0,14000,16000,18000,20000,22000]
labels = ["0-14000","14000-1600","1600-18000","18000-20000","20000-22000"] # Labels should always be one less than the number of bins

mountains_df['Elevation Range'] = pd.cut(mountains_df['Elevation'], bins, labels=labels)
mountains_df
```

	Mountain peak	Region	Mountain range	Elevation	Prominence	Isolation	Elevation Range
Rank							
1	Denali[a](Mount McKinley)	Alaska	Alaska Range	20310	20146	NaN	20000-22000
2	Mount Logan[b]	Yukon	Saint Elias Mountains	19551	17215	387.0	18000-20000
3	Pico de Orizaba[c](Citlaltépetl)	Puebla Veracruz	Cordillera Neovolcanica	18491	16148	NaN	18000-20000
4	Mount Saint Elias[d]	Alaska Yukon	Saint Elias Mountains	18009	11250	25.6	18000-20000
5	Popocatépetl[e][f]	México Morelos Puebla	Cordillera Neovolcanica	17749	9974	88.8	1600-18000
...	...	...	...	...	...	...	...
396	Sierra Fría[my]	Aguascalientes	Sierra Madre Occidental	9941	1640	145.6	0-14000
398	Hayford Peak[147][mz]	Nevada	Sheep Range	9924	5412	33.8	0-14000
399	Ulysses Mountain[na][nb](Mount Ulysses)	British Columbia	Muskwa Ranges	9921	7526	271.0	0-14000
400	Eagle Peak[148][nc]	California	Warner Mountains	9895	4362	87.4	0-14000
401	Sacajawea Peak[nd][ne]	Oregon	Wallowa Mountains	9843	6393	125.5	0-14000

401 rows × 7 columns

```
# Using groupby to get the average statistics in each range
mountains_df.groupby('Elevation Range').mean()
```

	Elevation	Prominence	Isolation
Elevation Range			

	Elevation	Prominence	Isolation
Elevation Range			
0-14000	11952.266862	4033.539589	44.144265
14000-1600	14446.000000	4631.425532	38.856818
1600-18000	16909.333333	7187.444444	33.347778
18000-20000	18683.666667	14871.000000	206.300000
20000-22000	20310.000000	20146.000000	NaN

5) Views vs. Copies

```
x = 3
items = [3, x]
items
x = 29
items[1] = 29
```

```
df
```

	Num_1	Num_2	Special Column just for Ranjani
Letters			
a	14	7	20
b	13	10	23
c	14	13	24

```
view = df['Num_1']
view
```

```
Letters
a    365
b     13
c     14
Name: Num_1, dtype: int64
```

```
view.iat[0] = 365
df
```

	Num_1	Num_2	Special Column just for Ranjani
Letters			
a	365	7	20
b	13	10	23
c	14	13	24

```
copy_1 = df['Num_1'].copy()
copy_1
```

```
Letters
a    365
b     13
c     14
Name: Num_1, dtype: int64
```

```
copy_1.iat[0] = 0
copy_1
```

```
Letters
a     0
b    13
c    14
Name: Num_1, dtype: int64
```

```
df1 = df.loc[["a","b"]]
df1.at["a","Num_1"] = 4
df1
```

	Num_1	Num_2	Special Column just for Ranjani
Letters			
a	4	7	20
b	13	10	23

```
df
```

	Num_1	Num_2	Special Column just for Ranjani
Letters			
a	365	7	20
b	13	10	23
c	14	13	24

Future inclusions:

- set_index
- reset_index
- add a new column
- rename a column

Thank you for reading!

-Seth Pruitt