Data Analysis with Python

Intro to Pandas DataFrames

DataFrame: most important data structure of pandas

- It is a tabular structure tightly integrated with Series
- Use analysis of G7 countries and Series as reference

G7 Stats							
	Population	GDP	Surface	HDI	Continent		
Canada	35.467	1,785,387.00	9,984,670	0.913	America		
France	63.951	2,833,687.00	640,679	0.888	Europe		
Germany	80.94	3,874,437.00	357,114	0.916	Europe		
Italy	60.665	2,167,744.00	301,336	0.873	Europe		
Japan	127.061	4,602,367.00	377,930	0.891	Asia		
United Kingdom	64.511	2,950,039.00	242,495	0.907	Europe		
United States	318.523	17,348,075.00	9,525,067	0.915	America		

- A DataFrame looks alot like a table
- Creating DataFrames manually can be tedious
- 99% of the time you'll be pulling Data from a Database, a csv file or the web
- You cans till create a DataFrame by specifying the columns and values

```
df = pd.DataFrame({
    'Population': [35.467, 63.951, 80.94, 60.665, 127.061, 64.511, 318.523],
    'GDP': [
        1785387,
        2833687,
        3874437,
        2167744,
        4602367,
        2950039,
        17348075
    1,
    'Surface Area': [
        9984670,
        640679,
        357114,
        301336,
        377930,
        242495,
        9525067
    1,
    'HDI': [
        0.913,
        0.888,
        0.916,
        0.873,
        0.891,
        0.907,
        0.915
    1,
    'Continent': [
        'America',
        'Europe',
        'Europe',
        'Europe',
        'Asia',
        'Europe',
        'America'
}, columns=['Population', 'GDP', 'Surface Area', 'HDI', 'Continent'])
```

- The *columns* attribute is optional; I just wanted to match the table above

In [3]:	df					
		Population	GDP	Surface Area	HDI	Continent
	0	35.467	1785387	9984670	0.913	America
	1	63.951	2833687	640679	0.888	Europe
	2	80.940	3874437	357114	0.916	Europe
	3	60.665	2167744	301336	0.873	Europe
	4	127.061	4602367	377930	0.891	Asia
	5	64.511	2950039	242495	0.907	Europe
	6	318.523	17348075	9525067	0.915	America

DataFrames also have indexes.

- As seen in the table above, pandas has assigned a numeric, auto-incremental index to each row in the DataFrame
- In this case, each row represents a country, so will reassign the index



df

 Notice: the indices within the table changed from basic integer indices to the names of G7 countries

```
In [6]:

df.index

Index(['Canada', 'France', 'Germany', 'Italy', 'Japan', 'UK', 'US'], dtype='object')
```

df.index

```
df.info()
 <class 'pandas.core.frame.DataFrame'>
 Index: 7 entries, Canada to US
 Data columns (total 5 columns):
 # Column Non-Null Count Dtype
 0 Population 7 non-null
                                float64
 1 GDP 7 non-null
                                int64
 2 Surface Area 7 non-null
                                int64
 3 HDI
                 7 non-null
                                float64
     Continent
                 7 non-null
                                object
 dtypes: float64(2), int64(2), object(1)
 memory usage: 280.0+ bytes
```

df.info()

```
In [8]: df.size
```

df.size

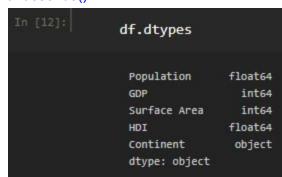


df.shape

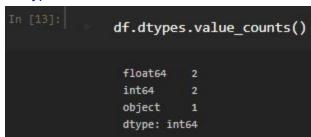
- (rows, cols)



df.describe()



df.dtypes



df.dtypes.value_counts()

Indexing, Selection and Slicing

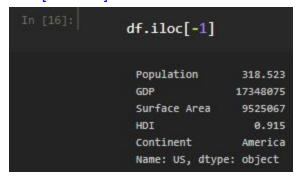
Individual columns in the DataFrame can be selected with regular indexing. Each column is represented as a Series.

In [14]:	# Indexi df	# Indexing, Selection and Slicing df						
		Population	GDP	Surface Area	HDI	Continent		
	Canada	35.467	1785387	9984670	0.913	America		
	France	63.951	2833687	640679	0.888	Europe		
	Germany	80.940	3874437	357114	0.916	Europe		
	Italy	60.665	2167744	301336	0.873	Europe		
	Japan	127.061	4602367	377930	0.891	Asia		
	UK	64.511	2950039	242495	0.907	Europe		
	US	318.523	17348075	9525067	0.915	America		

df



df.loc['Canada']



df.iloc[-1]

- Note the difference between *In[15]* and *In[16]* df.loc['...'] vs df.iloc[index/int]

```
df['Population']
             35.467
 Canada
 France
             63.951
             80.940
 Germany
 Italy
             60.665
 Japan
            127.061
 UK
             64.511
 US
            318.523
 Name: Population, dtype: float64
```

df['Population']

- Note: The index of the returned Series is the same as the DataFrame one. And its name is the name of the column.

If you are working on a notebook and want to see a more **DataFrame-like format** you can use the *to frame method*



df['Population'].to_frame()



df[['Population', 'GDP']]

- This shows that multiple columns can also be selected similarly to numpy and Series
- In this case, the result is a DataFrame



df[1:3]

Slicing works differently. It acts at 'row level' and can be counter intuitive

- Note: the upper limit is EXCLUDED



df.loc['ltaly']

Row level selection works better with *loc* and *iloc* which are recommended over regular "direct slicing" (df[:]).

- loc selects rows matching the given index



df.loc['France': 'Italy']

- Notice the difference between In[20] and In[22]
 - In[22] includes the upper limit while In[20] doesn't

```
df.loc['France': 'Italy', 'Population']

France 63.951
Germany 80.940
Italy 60.665
Name: Population, dtype: float64
```

df.loc['France': 'Italy', 'Population']

- As a **second argument**, you can pass the **column(s)** you'd like to extract



df.loc['France': 'Italy', 'Population'].to_frame()

- Notice the difference between In[24] and In[25]



df.loc['France': 'Italy', ['Population', 'GDP']]

Generally, if there are multiple columns indexed, then the output will be a dataFrame

Even without the .to_frame()

In [29]:	df					
		Population	GDP	Surface Area	HDI	Continent
	Canada	35.467	1785387	9984670	0.913	America
	France	63.951	2833687	640679	0.888	Europe
	Germany	80.940	3874437	357114	0.916	Europe
	Italy	60.665	2167744	301336	0.873	Europe
	Japan	127.061	4602367	377930	0.891	Asia
	UK	64.511	2950039	242495	0.907	Europe
	US	318.523	17348075	9525067	0.915	America



df.iloc[0]



df.iloc[-1]



df.iloc[[0, 1, -1]]



df.iloc[1:3]

- Note: the upper limit index is excluded in the output



df.iloc[1:3, 3]



df.iloc[1:3, 3].to_frame()

- Compare the output between In[35] and In[36]



df.iloc[1:3, [0, 3]]



df.iloc[1:3, 1:3]

- Note: the upper limit for both the row index slice and column index slice are excluded

NOTE: Always use *iloc* and *loc* to reduce ambiguity, especially for DataFrames with numeric indices

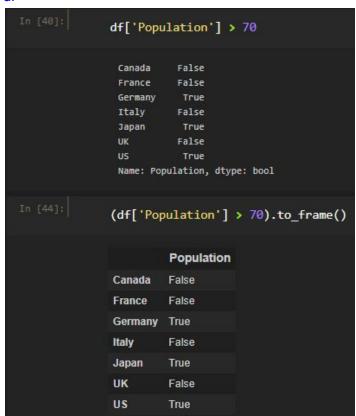
Conditional Selection (boolean arrays)

Recall conditional selection applied to Series. The methods will work the same way for DataFrames.

- This makes sense since a DataFrame is a collection of Series

In [39]:	df					
		Population	GDP	Surface Area	HDI	Continent
	Canada	35. 4 67	1785387	9984670	0.913	America
	France	63.951	2833687	640679	0.888	Europe
	Germany	80.940	3874437	357114	0.916	Europe
	Italy	60.665	2167744	301336	0.873	Europe
	Japan	127.061	4602367	377930	0.891	Asia
	UK	64.511	2950039	242495	0.907	Europe
	US	318.523	17348075	9525067	0.915	America

df



df['Population'] > 70



df.loc[df['Population'] > 70]



df.loc[df['Population'] > 70, 'Population']

The boolean matching is done at index level, so you can filter by any row, as long as it contains the right indexes.

Column selection still works as expected



df.loc[df['Population'] > 70, ['Population', 'GDP']]

Dropping Stuff

Opposed to the concept of selection, we have "dropping". Instead of pointing out which values you'd like to *select* you could specify which ones you'd like to drop.

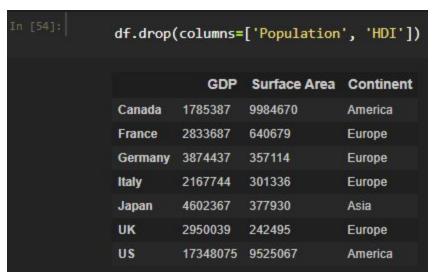


df.drop('Canada')

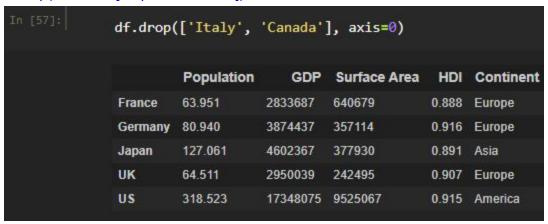


df.drop(['Canada', 'Japan'])

- Compare *In[50]* and *In[53]*
 - Notice when you index **multiple rows**, the rows must be within [...]
 - When there's **one** row, (...) is fine



df.drop(columns=['Population', 'HDI'])



df.drop(['Italy', 'Canada'], axis=0)

drops rows



df.drop(['Population', 'HDI'], axis=1)

- Drops columns



df.drop(['Population', 'HDI'], axis='columns')

- This is an equivalent code/ prompt as above (receive same DataFrame output)



df.drop(['Canada', 'Germany'], axis='rows')

All these *drop* methods return a **new DataFrame**. If you want to modify it "in place", can use the *inpace* attribute.

- Example below

Operations

In [62]:	df[['Population', 'GDP']]					
		Population	GDP			
	Canada	35.467	1785387			
	France	63.951	2833687			
	Germany	80.940	3874437			
	Italy	60.665	2167744			
	Japan	127.061	4602367			
	UK	64.511	2950039			
	US	318.523	17348075			

df[['Population', 'GDP']]



df[['Population', 'GDP']] / 100

 $\label{eq:crisis} \textit{crisis} = \textit{pd}. \\ \textit{Series}([-1_000_000, -0.3], \\ \textit{index=['GDP', 'HDI']}) \\ \textit{crisis}$



df[['GDP', 'HDI']]



df[['GDP', 'HDI']] + crisis

- Notice: calculation has been applied to each entry within this DataFrame

Modifying DataFrames

It's simple and intuitive. You can add columns, or replace values for columns without issues.

https://www.youtube.com/watch?v=r-uOLxNrNk8

 $\underline{https://github.com/ine-rmotr-curriculum/freecodecamp-intro-to-pandas/blob/master/3\%20-\%20Pandas\%20-\%20DataFrames.ipynb}$

 $\underline{https://docs.google.com/spreadsheets/d/1IlorV2-Oh9Da1JAZ7weVw86PQrQydSMp-ydVMH135i}\\ \underline{I/edit\#gid=0}$