Week 2 Lab (Intro to pandas)

COSC 3337 Dr. Rizk

Intro to pandas

Pandas is a high-level data manipulation tool developed by Wes McKinney. It is built on the Numpy package and offers data structures and operations for manipulating numerical tables and time series. Pandas allows us to import data from various file formats such as comma-separated values, JSON, SQL, Microsoft Excel, etc. Throughout the course, we'll be taking advantage of pandas' various data manipulation operations such as merging, reshaping, selecting, as well as data cleaning, and data wrangling features. Just like Numpy, pandas is highly optimized for performance, with critical code paths written in C/C++.

Let's begin by importing pandas and learning about the Series data type. If for some reason you don't have pandas installed, you will first have to go to your terminal (or Anaconda Prompt if on Windows) and enter the following:

```
conda install pandas
```

Make sure you've already installed Anaconda

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

Series

A Series is a one-dimensional labeled array. What this means is that we can now access (index) elements in this array using some assigned labels. We create a Series using ***pd.Series(data, index)***, where data is our array and index is the corresponding labels. Let's see an example below:

In [2]:
```python
my_series = pd.Series(data=[1, 2, 3], index=['A', 'B', 'C'])
print(my_series)
print(type(my_series))
```

```
A    1
B    2
C    3
dtype: int64
<class 'pandas.core.series.Series'>
```

Since ***my_series*** is of type Series, we can now access the first element in the array in two ways. The usual way we access array elements: ***my_series[0]***, and an additional way using the index labels we assigned: ***my_series['A']***. Try accessing all of the array elements using both methods. See the example below.

In [3]:
```python
print(f"Accessing first element using my_series[0]: {my_series[0]}")
print(f"Accessing first element using my_series['A']: {my_series['A']}")
```

```
Accessing first element using my_series[0]: 1
Accessing first element using my_series['A']: 1
```

Note that ***data*** can be passed as:

- A Python list (like we saw above)
- A Numpy array
- A Python dictionary

Here's an example of how we would pass a dictionary to ***data***. Since dictionarys already come with key value pairs, there's no need for us to pass index labels.

In [4]:
```python
my_series = pd.Series(data={'A': 1, 'B': 2, 'C': 3})
print(my_series)
print(type(my_series))
```

```
A    1
B    2
C    3
dtype: int64
<class 'pandas.core.series.Series'>
```

And here's a Numpy array example:

In [5]:
```python
my_series = pd.Series(data=np.array([1, 2, 3]), index=['A', 'B', 'C'])
print(my_series)
print(type(my_series))
```

```
A    1
B    2
C    3
dtype: int64
<class 'pandas.core.series.Series'>
```

Note: If index labels are not specified for a Series, they will default to [0, n) where n is the number of data values we have. For example:

In [6]:
```python
my_series = pd.Series(pd.Series(data=[10, 20, 30]))
print(my_series)
print(type(my_series))
```

```
0    10
1    20
2    30
dtype: int64
<class 'pandas.core.series.Series'>
```

What's really cool about Series is that we can perform operations on them, which will be done based off of the index. For example, let's say that I have two Series: ***week_one*** and ***week_two***, both representing how much money I owe my employees for each week.

In [7]:
```python
week_one = pd.Series(data=[100, 50, 300], index=['Bob', 'Sally', 'Jess'])
week_one
```

Out[7]:
```
Bob      100
Sally     50
Jess     300
dtype: int64
```

In [8]:
```python
week_two = pd.Series(data=[500, 30, 20], index=['Bob', 'Sally', 'Jess'])
week_two
```

Out[8]:
```
Bob      500
Sally     30
Jess      20
dtype: int64
```

We can then sum these two Series together to get a new Series ***total_due*** representing the total amount that we owe each person for the two weeks.

In [9]:
```python
total_due = week_one + week_two
total_due
```

Out[9]:
```
Bob      600
Sally     80
Jess     320
dtype: int64
```

Notice how the operator (+ in this case) was applied along the corresponding labels.

Awesome! Let's now move on to talk about pandas DataFrames, which builds off of Series and is what we'll be using most throughout the course.

DataFrames

Formally, a DataFrame is a ***2-dimensional*** labeled data structure with columns of potentially different types. It's probably easiest to think of DataFrames as many Series placed next to each other to share a common index label, but you can also think of them as spreadsheets, SQL tables, or a dictionary of Series objects.

We create a DataFrame using ***pd.DataFrame(data, index, columns)***. ***data*** and ***index*** are pretty familiar to us now that we've seen Series, but what is this new ***columns*** parameter? Well, if you think of DataFrames as many Series placed next to each other to share a common index label, we need some way of accessing each individual Series. This is where ***columns*** comes in. You can think of it as additional lables for each individual Series/column. Let's see an example below.

In [10]:
```python
my_df = pd.DataFrame(data=np.arange(0,20).reshape(4,5), index=['A', 'B', 'C', 'D'],
                     columns=['col1', 'col2', 'col3', 'col4', 'col5'])
print(my_df)
print(type(my_df))
```

```
   col1  col2  col3  col4  col5
A     0     1     2     3     4
B     5     6     7     8     9
C    10    11    12    13    14
D    15    16    17    18    19
<class 'pandas.core.frame.DataFrame'>
```

Notice the type of what's returned when we access 'col2'. A Series 😮, which shouldn't come as too much of a surprise since we mentioned that DataFrames can be thought of as these individual Series placed next to each other to share a common index label.

In [11]:
```python
print(my_df['col2'])
print(type(my_df['col2']))
```

```
A     1
B     6
C    11
D    16
Name: col2, dtype: int64
<class 'pandas.core.series.Series'>
```

Note: As mentioned in the Series section, DataFrame index labels will also default to [0, n) if not specified. For example:

In [12]:
```python
my_df = pd.DataFrame(data=np.arange(0,20).reshape(4,5), columns=['col1', 'col2', 'col3', 'col4', 'col5'])
my_df
```

Out[12]:

	col1	col2	col3	col4	col5
0	0	1	2	3	4
1	5	6	7	8	9
2	10	11	12	13	14
3	15	16	17	18	19

Also, DataFrames will display a lot nicer in jupyter notebooks if you don't call print() on them. 😊

Let's now see how we can access data from these DataFrames.

DataFrames: Selection

In [13]:
```python
my_df = pd.DataFrame(data=np.arange(0,20).reshape(4,5), index=['A', 'B', 'C', 'D'],
                     columns=['col1', 'col2', 'col3', 'col4', 'col5'])
my_df
```

Out[13]:

	col1	col2	col3	col4	col5
A	0	1	2	3	4
B	5	6	7	8	9
C	10	11	12	13	14
D	15	16	17	18	19

We can access individual columns using the column names/labels we specified.

In [14]:
```python
my_df['col2']
```

Out[14]:
```
A     1
B     6
C    11
D    16
Name: col2, dtype: int64
```

We can access multiple columns by specifying a list of the column names that we'd like to retrieve.

In [15]:
```python
my_df[['col2', 'col3']]
```

Out[15]:

	col2	col3
A	1	2
B	6	7
C	11	12
D	16	17

What if we'd like to access row information? We can specify the index location using ***.iloc***, or the index name/label using ***.loc***.

In [16]:
```python
my_df.iloc[0]
```

Out[16]:
```
col1    0
col2    1
col3    2
col4    3
col5    4
Name: A, dtype: int64
```

In [17]:
```python
my_df.loc['A']
```

Out[17]:
```
col1    0
col2    1
col3    2
col4    3
col5    4
Name: A, dtype: int64
```

We can grab a section using ***[start_index : stop_index]***. stop_index is not inclusive when using index locations. This should look familiar to how we accessed elements in Numpy arrays.

In [18]:
```python
my_df.iloc[0:3]
```

Out[18]:

	col1	col2	col3	col4	col5
A	0	1	2	3	4
B	5	6	7	8	9
C	10	11	12	13	14

In [19]:
```python
my_df.loc['A':'C']
```

Out[19]:

	col1	col2	col3	col4	col5
A	0	1	2	3	4
B	5	6	7	8	9
C	10	11	12	13	14

For a section of both rows and columns, we must specify both the row sections of interest and column sections of interest. This should also look familiar to how we accessed row and column sections of interest from 2d Numpy arrays, except we now have to use ***.iloc*** or ***.loc*** depending on how we'd like to specify the rows (by index position or index label/name).

In [20]:
```python
my_df.iloc[1:4, 0:3]
```

Out[20]:

	col1	col2	col3
B	5	6	7
C	10	11	12
D	15	16	17

In [21]:
```python
my_df.loc['B':'D', 'col1':'col3']
```

Out[21]:

	col1	col2	col3
B	5	6	7
C	10	11	12
D	15	16	17

Recall how we were able to select elements from a Numpy array based off of some condition. The same can be done with DataFrames since our data is essentially just a Numpy array. Let's see a quick example.

In [22]:
```python
my_df = pd.DataFrame(data=np.arange(0,20).reshape(4,5), index=['A', 'B', 'C', 'D'],
                     columns=['col1', 'col2', 'col3', 'col4', 'col5'])
my_df
```

Out[22]:

	col1	col2	col3	col4	col5
A	0	1	2	3	4
B	5	6	7	8	9
C	10	11	12	13	14
D	15	16	17	18	19

Using comparison operators with our DataFrame, we get back a DataFrame with True/False values indicating whether the value at that position satisfied the condition.

In [23]:
```python
my_df % 2 == 0
```

Out[23]:

	col1	col2	col3	col4	col5
A	True	False	True	False	True
B	False	True	False	True	False
C	True	False	True	False	True
D	False	True	False	True	False

So as we saw with Numpy arrays, we can specify this condition as what we would like to select. We'll then get back only the values that met the condition. Those that did not meet the condition will get replaced with NaN representing a missing or empty value.

In [24]:
```python
my_df[my_df % 2 == 0]
```

Out[24]:

	col1	col2	col3	col4	col5
A	0.0	NaN	2.0	NaN	4.0
B	NaN	6.0	NaN	8.0	NaN
C	10.0	NaN	12.0	NaN	14.0
D	NaN	16.0	NaN	18.0	NaN

We'll later learn how to properly take care of these NaN values, but we can fill them all with a certain value using ***fillna(value)***. For example:

In [25]:
```python
# filling all NaN values with 0
my_df[my_df % 2 == 0].fillna(value=0)
```

Out[25]:

	col1	col2	col3	col4	col5
A	0.0	0.0	2.0	0.0	4.0
B	0.0	6.0	0.0	8.0	0.0
C	10.0	0.0	12.0	0.0	14.0
D	0.0	16.0	0.0	18.0	0.0

In [26]:
```python
# filling all NaN values with whatever the mean of my_df's original col2 is: (1+6+11+16)/4 = 8.5
my_df[my_df % 2 == 0].fillna(value=my_df['col2'].mean())
```

Out[26]:

	col1	col2	col3	col4	col5
A	0.0	8.5	2.0	8.5	4.0
B	8.5	6.0	8.5	8.0	8.5
C	10.0	8.5	12.0	8.5	14.0
D	8.5	16.0	8.5	18.0	8.5

DataFrames: Adding and Dropping Columns

In [27]:
```python
my_df
```

Out[27]:

	col1	col2	col3	col4	col5
A	0	1	2	3	4
B	5	6	7	8	9
C	10	11	12	13	14
D	15	16	17	18	19

We can create new columns in a DataFrame by either passing in the new data we would like to store there, or from existing columns/features in our DataFrame. Let's see an example of both cases below.

In [28]:
```python
my_df['newCol'] = [10, 20, 30, 40]
my_df
```

Out[28]:

	col1	col2	col3	col4	col5	newCol
A	0	1	2	3	4	10
B	5	6	7	8	9	20
C	10	11	12	13	14	30
D	15	16	17	18	19	40

In [29]:
```python
my_df['col1+col2'] = my_df['col1'] + my_df['col2']
my_df
```

Out[29]:

	col1	col2	col3	col4	col5	newCol	col1+col2
A	0	1	2	3	4	10	1
B	5	6	7	8	9	20	11
C	10	11	12	13	14	30	21
D	15	16	17	18	19	40	31

What if we want to drop/remove certain column(s)? We can acomplish this using ***drop(columns)***.

In [30]:
```python
my_df.drop(columns=['newCol'])
```

Out[30]:

	col1	col2	col3	col4	col5	col1+col2
A	0	1	2	3	4	1
B	5	6	7	8	9	11
C	10	11	12	13	14	21
D	15	16	17	18	19	31

Note that these changes are not done inplace. This means that these changes are not permanent. We can see this if we print my_df again.

In [31]:
```python
my_df
```

Out[31]:

	col1	col2	col3	col4	col5	newCol	col1+col2
A	0	1	2	3	4	10	1
B	5	6	7	8	9	20	11
C	10	11	12	13	14	30	21
D	15	16	17	18	19	40	31

To make these changes permanent, we can supply an additional parameter ***inplace=True***.

In [32]:
```python
my_df.drop(columns=['newCol', 'col1+col2'], inplace=True)
```

Now if we print my_df again we can see that the changes were saved. Keep this in mind when you want to modify the original DataFrame.

In [33]:
```python
my_df
```

Out[33]:

	col1	col2	col3	col4	col5
A	0	1	2	3	4
B	5	6	7	8	9
C	10	11	12	13	14
D	15	16	17	18	19

DataFrames: Groupby and Common Operations

In [34]:
```python
my_df = pd.DataFrame({'Type': ['Falcon', 'Falcon', 'Parrot', 'Parrot', 'Cat', 'Cat', 'Cat'],
                      'Max Speed': [380., 370., 24., 26., 50., 50., 150.]})
my_df
```

Out[34]:

	Type	Max Speed
0	Falcon	380.0
1	Falcon	370.0
2	Parrot	24.0
3	Parrot	26.0
4	Cat	50.0
5	Cat	50.0
6	Cat	150.0

We'll use this small DataFrame to demonstrate some common operations and useful functions that we can perform on DataFrames. ***my_df*** has 7 observations (0-6) of animals. For each animal, we recorded the type of animal that they are, and their max speed.

Something we'll often like to know is how many unique values are in a certain column. We can achieve this by calling ***unique()*** on our column of interest.

In [35]:
```python
print(f"unique types: {my_df['Type'].unique()}")
print(f"unique max speeds: {my_df['Max Speed'].unique()}")
```

```
unique types: ['Falcon' 'Parrot' 'Cat']
unique max speeds: [380. 370.  24.  26.  50. 150.]
```

What if we'd like to also know how many of each unique type there are? This can be done by instead calling ***value_counts()***. This will not only tell us how many unique values there are in that column (cat, parrot, and falcon in this case), but also how many of that type we have (3 cats, 2 parrots, and 2 falcons in this DataFrame).

In [36]:
```python
my_df['Type'].value_counts()
```

Out[36]:
```
Cat       3
Parrot    2
Falcon    2
Name: Type, dtype: int64
```

We can also do things like get the sum, mean, min, or max of a column:

In [37]:
```python
print(f"sum of Max Speed col: {my_df['Max Speed'].sum()}")
print(f"mean of Max Speed col: {my_df['Max Speed'].mean()}")
print(f"min from Max Speed col: {my_df['Max Speed'].min()}")
print(f"max from Max Speed col: {my_df['Max Speed'].max()}")
```

```
sum of Max Speed col: 1050.0
mean of Max Speed col: 150.0
min from Max Speed col: 24.0
max from Max Speed col: 380.0
```

What if we wanted to know the mean Max Speed for each group of animals? This is where a function called ***groupby(by)*** will come in handy. This will group all common types together and then allow us to apply a function like mean to each group. For example:

In [38]:
```python
# This grouped all cats together, all falcons together, all parrots together and then applied the mean funct
# to each groups columns (only Max Speed in this case). So we can see that the mean Max Speed for all cats i
# DataFrame is 83.333333.
my_df.groupby(by='Type').mean()
```

Out[38]:

	Max Speed
Type	
Cat	83.333333
Falcon	375.000000
Parrot	25.000000

Note that here we grouped by the 'Type' column, but we could specify a different column if we wanted to. Just make sure that you group by something that makes sense.

Extra Practice

Great! now let's use what we've learned to explore a real dataset using pandas.

We'll be looking at a pokemon dataset, which contains the following attributes:

- #: ID for each pokemon
- Name: Name of each pokemon
- Type 1: Each pokemon has a type, this determines weakness/resistance to attacks
- Type 2: Some pokemon are dual type and have 2
- Total: sum of all stats that come after this, a general guide to how strong a pokemon is
- HP: hit points, or health, defines how much damage a pokemon can withstand before fainting
- Attack: the base modifier for normal attacks (eg. Scratch, Punch)
- Defense: the base damage resistance against normal attacks
- SP Atk: special attack, the base modifier for special attacks (e.g. fire blast, bubble beam)
- SP Def: the base damage resistance against special attacks
- Speed: determines which pokemon attacks first each round

We'll typically be reading in an existing dataset from our computer, which pandas will then convert into a beautiful DataFrame for us rather than having to create the whole thing from scratch like we did in this lab. To do this, we'll use ***pd.read_csv(filepath_or_buffer)***. If the dataset is located in the same directory as this jupyter notebook, we can simply provide the name of the dataset file into the ***filepath_or_buffer*** parameter. Otherwise we'll have to specify the path to this file.

In [39]:
```python
pokemon_df = pd.read_csv(filepath_or_buffer='Pokemon.csv')
```

After reading, we can get a preview of this dataset using ***head()***. This will show us the first 5 values in the dataset by default, but you can specify more or less inside the parenthesis.

In [40]:
```python
pokemon_df.head()
```

Out[40]:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
0	1	Bulbasaur	Grass	Poison	318	45	49	49	65	65	45	1	False
1	2	Ivysaur	Grass	Poison	405	60	62	63	80	80	60	1	False
2	3	Venusaur	Grass	Poison	525	80	82	83	100	100	80	1	False
3	3	VenusaurMega Venusaur	Grass	Poison	625	80	100	123	122	120	80	1	False
4	4	Charmander	Fire	NaN	309	39	52	43	60	50	65	1	False

We can then print a concise summary of our pokemon DataFrame using ***info()***. The info below lets us know that we have 800 entries, and how many non-null values are in each feature/column. Since the Type 2 column only contains 414 non-null values, there are 386 missing values in this column. We'll want to take care of this since we don't like to have missing values.

In [41]:
```python
pokemon_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800 entries, 0 to 799
Data columns (total 13 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   #           800 non-null    int64
 1   Name        800 non-null    object
 2   Type 1      800 non-null    object
 3   Type 2      414 non-null    object
 4   Total       800 non-null    int64
 5   HP          800 non-null    int64
 6   Attack      800 non-null    int64
 7   Defense     800 non-null    int64
 8   Sp. Atk     800 non-null    int64
 9   Sp. Def     800 non-null    int64
 10  Speed       800 non-null    int64
 11  Generation  800 non-null    int64
 12  Legendary   800 non-null    bool
dtypes: bool(1), int64(9), object(3)
memory usage: 75.9+ KB
```

As you'll learn in lecture, there are many ways to handle missing data. I'll list some options down below for both categorical and quantitative variables.

Some options for ***categorical*** variables include:

- Remove observations with missing values if we are dealing with a large dataset and the number of records containing missing values are few.
- Remove the variable/column if it is not significant.
- Develop a model to predict missing values. KNN for example.
- Replace missing values with the most frequent in that column.

Some options for ***quantitative*** variables include:

- Remove the variable/column if it is not significant.
- Impute missing values with something like the mean/average value in that column.
- Develop a model to predict missing values.

Note: There is not one single method that will work for every case. Determining how you'll handle missing data will vary between datasets.

Here we will keep it simple and fill missing values with their corresponding Type 1 value, but we'll see some of the other methods in later labs.

In [42]:
```python
pokemon_df['Type 2'].fillna(pokemon_df['Type 1'], inplace=True)
```

After filling missing Type 2 values, we can call ***info()*** again and see that there are no longer any missing values. This dataset contains a total of 800 entries, and there are 800 non-null values in every column.

In [43]:
```python
pokemon_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 800 entries, 0 to 799
Data columns (total 13 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   #           800 non-null    int64
 1   Name        800 non-null    object
 2   Type 1      800 non-null    object
 3   Type 2      800 non-null    object
 4   Total       800 non-null    int64
 5   HP          800 non-null    int64
 6   Attack      800 non-null    int64
 7   Defense     800 non-null    int64
 8   Sp. Atk     800 non-null    int64
 9   Sp. Def     800 non-null    int64
 10  Speed       800 non-null    int64
 11  Generation  800 non-null    int64
 12  Legendary   800 non-null    bool
dtypes: bool(1), int64(9), object(3)
memory usage: 75.9+ KB
```

Recall how we were able to select observations from a DataFrame based off of some conditional. Let's use this to see how many legendary pokemon are in this dataset.

In [44]:
```python
pokemon_df[pokemon_df['Legendary'] == True]
```

Out[44]:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
156	144	Articuno	Ice	Flying	580	90	85	100	95	125	85	1	True
157	145	Zapdos	Electric	Flying	580	90	90	85	125	90	100	1	True
158	146	Moltres	Fire	Flying	580	90	100	90	125	85	90	1	True
162	150	Mewtwo	Psychic	Psychic	680	106	110	90	154	90	130	1	True
163	150	MewtwoMega Mewtwo X	Psychic	Fighting	780	106	190	100	154	100	130	1	True
...	...	...	...	...	...	...	...	...	...	...	...	...	...
795	719	Diancie	Rock	Fairy	600	50	100	150	100	150	50	6	True
796	719	DiancieMega Diancie	Rock	Fairy	700	50	160	110	160	110	110	6	True
797	720	HoopaHoopa Confined	Psychic	Ghost	600	80	110	60	150	130	70	6	True
798	720	HoopaHoopa Unbound	Psychic	Dark	680	80	160	60	170	130	80	6	True
799	721	Volcanion	Fire	Water	600	80	110	120	130	90	70	6	True

65 rows × 13 columns

65 rows x 13 columns tells us that there are 65 legendary pokemon in this dataset. Note that you could also retrieve this information as a tuple by calling ***shape*** off of this DataFrame. For example:

In [45]:
```python
pokemon_df[pokemon_df['Legendary'] == True].shape
```

Out[45]:
```
(65, 13)
```

Try to see if you can figure out how many pokemon of type fire there are.

In [46]:
```python
pokemon_df[pokemon_df['Type 1'] == 'Fire'].shape
```

Out[46]:
```
(52, 13)
```

How about if we'd like to find the pokemon with max HP? Well, there are multiple ways to do this. One way is by using ***idxmax()***, which will tell us the index location of the max value in the column of interest, and we can then use this information to index the DataFrame.

In [47]:
```python
pokemon_df['HP'].idxmax()
```

Out[47]:
```
261
```

In [48]:
```python
pokemon_df.iloc[pokemon_df['HP'].idxmax()]
```

Out[48]:
```
#                 242
Name          Blissey
Type 1         Normal
Type 2         Normal
Total             540
HP                255
Attack             10
Defense            10
Sp. Atk            75
Sp. Def           135
Speed              55
Generation          2
Legendary       False
Name: 261, dtype: object
```

An alternative method is to use ***sort_values(by)*** to sort the DataFrame by 'HP' and using ***head(1)*** to only display to top observation.

In [49]:
```python
pokemon_df.sort_values(by='HP',ascending=False).head(1)
```

Out[49]:

	#	Name	Type 1	Type 2	Total	HP	Attack	Defense	Sp. Atk	Sp. Def	Speed	Generation	Legendary
261	242	Blissey	Normal	Normal	540	255	10	10	75	135	55	2	False

As you can see, there will often be more than one way to do things in this course. 🙂

Now see if you can figure out how many of each unique Type 1 pokemon there are.

In [50]:
```python
pokemon_df['Type 1'].value_counts()
```

Out[50]:
```
Water       112
Normal       98
Grass        70
Bug          69
Psychic      57
Fire         52
Electric     44
Rock         44
Ground       32
Dragon       32
Ghost        32
Dark         31
Poison       28
Fighting     27
Steel        27
Ice          24
Fairy        17
Flying        4
Name: Type 1, dtype: int64
```

Last challenge. See if you can find out what the mean HP is for each of the Type 1 groups above. Hint: refer back to the groupby section of this lab.

In [51]:
```python
pokemon_df.groupby(by='Type 1').mean()['HP']
```

Out[51]:
```
Type 1
Bug         56.884058
Dark        66.806452
Dragon      83.312500
Electric    59.795455
Fairy       74.117647
Fighting    69.851852
Fire        69.903846
Flying      70.750000
Ghost       64.437500
Grass       67.271429
Ground      73.781250
Ice         72.000000
Normal      77.275510
```

```
Psychic     70.631579
Rock        65.363636
Steel       65.222222
Water       72.062500
Name: HP, dtype: float64
```

Congrats! You now know enough about Pandas to begin exploring your own datasets. 😎 Try finding a dataset of interest on kaggle.com and see what interesting things you can discover using what you've learned. In the Matplotlib lab we'll learn how to graph data and explore this dataset even further.