YData: Introduction to Data Science

Class 07: Pandas Series and DataFrames

Overview

Quick review of Boolean masking

Tuples and dictionaries

pandas

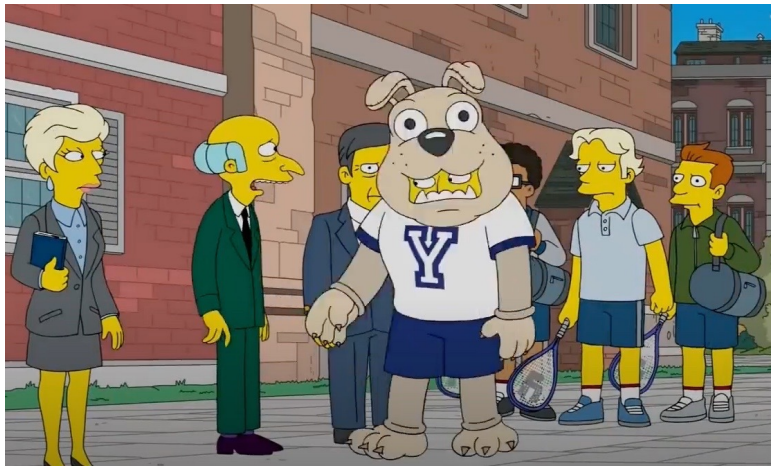
- Series
- DataFrames
- Selecting columns and rows from DataFrames
- Sorting values and adding new columns
- Calculating aggregate statistics for separate groups
- If there is time: joining DataFrames

Announcement: Homework 3

Homework 2 is due on Gradescope on Sunday February 11th at 11pm

- **Be sure to mark each question on Gradescope!**

How was homework 2?



Quick review of Boolean masking

We can also use Boolean arrays to return values in another array

- This is called "Boolean masking" or "Boolean indexing"

```
my_array = np.array([12, 4, 6, 3, 1])
boolean_mask = np.array([False, True, False, True, True])

smaller_array = my_array[boolean_mask]
```

This can be useful for calculating statistics on data that meet particular criteria:

```
np.mean(my_array[my_array < 5])    # what does this do?


boolean_mask = my_array < 5       # breaking it down into steps…
values_less_than_5  = my_array[boolean_mask ]
np.mean(values_less_than_5)
```

Let’s do a warm-up exercise in Jupyter!

Tuples and Dictionaries

Tuples

Tuples are like lists but they are immutable; i.e., once they are created we can’t change the values in a tuple.

We can create a tuple using:

- my_tuple = (10, 20, 30)

Like lists, we can access elements of tuples using square brackets

- my_tuple[1]

We can’t change values in tuples:

- my_tuple[1] = 50 # Error!!!

Tuples

We can assign values in tuples into regular names using “tuple unpacking”

- my_tuple = (10, 20, 30)
- val1, val2, val3 = my_tuple
- val3

Let's explore this in Jupyter!

Dictionaries

Dictionaries allow you to look up ***values*** based on a ***key***

- i.e., you supply a “key” and the dictionary returns the stored value

We can create dictionaries using the syntax:

- my_dict = { 'key1': 5, 'key2': 20}

We can retrieve dictionary values by supplying a key using square brackets []

- my_dict['key2']

Let's explore this in Jupyter!

Series and Tables

Pandas: Series and DataFrames

"pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language."

There are two main data structures in pandas:

- **Series**: represent one-dimensional data
- **DataFrames**: represent data tables
  - i.e., relational data

pandas Series

pandas Series are: One-dimensional ndarray with axis labels

- (including time series)

Example: egg _prices

DATE	
1980-01-01	0.879
1980-02-01	0.774
1980-03-01	0.812

Index

values

pandas Series

We can access elements by Index ***name*** using **.loc**

- egg_prices.loc["1980-01-01"]

We can access elements by Index ***number*** using **.iloc**

- egg_prices.iloc[0]

Since pandas Series are just ndarrays with an Index, all the numpy functions will work on Series

Let's explore this in Jupyter!

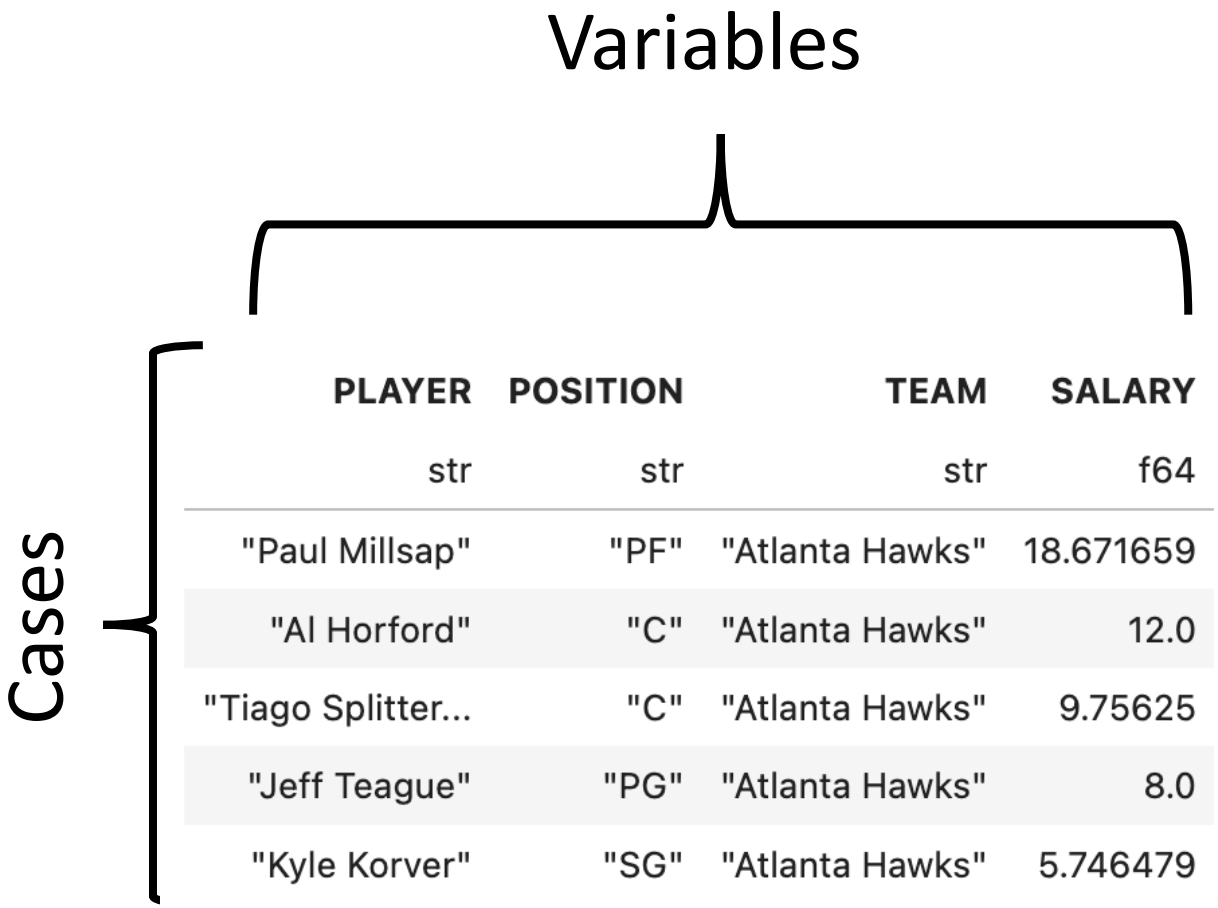
pandas DataFrames

Pandas DataFrame hold Table data

This is one of the most useful formats to extract insights from datasets

Often we read data into a DataFrame using:

- pd.read_csv("file.csv")



PLAYER	POSITION	TEAM	SALARY
str	str	str	f64
"Paul Millsap"	"PF"	"Atlanta Hawks"	18.671659
"Al Horford"	"C"	"Atlanta Hawks"	12.0
"Tiago Splitter...	"C"	"Atlanta Hawks"	9.75625
"Jeff Teague"	"PG"	"Atlanta Hawks"	8.0
"Kyle Korver"	"SG"	"Atlanta Hawks"	5.746479

Let's explore this in Jupyter!

Selecting columns from a DataFrame

We can select a column from a DataFrame using square brackets:

- my_df["my_col"] # returns a Series!

We can select multiple columns from a DataFrame by passing a list into the square brackets

- my_df[["col1", "col2"]]

Let's explore this in Jupyter!

Extracting rows from a DataFrame

We can extract rows from a DataFrame by:

1. The position they appear in the DataFrame
2. The Index values

We use the .iloc[] property to extract values by ***position***

- my_df.iloc[0]

We use the .loc[] property to extract values by ***Index value***

- my_df.loc["index_name"]

Extracting rows from a DataFrame

We can also extract rows through using Boolean masking

For example:

```
bool_mask = my_df[“col_name”] == 7
my_df.loc[bool_mask]
```

Or in one step: `my_df [my_df[“col_name”] == 7]`

Let's explore this in Jupyter!

Sorting rows from a DataFrame

We can sort values in a DataFrame using .sort_values("col_name")

- my_df.sort_values("col_name")

We can sort from highest to lowest by setting the argument ascending = False

- my_df.sort_values("col_name", ascending = False)

Let's explore this in Jupyter!

Adding new columns and renaming columns

We can add a column to a data frame using square backets. For example:

- my_df[“new_col”] = values_array
- my_df["new col"] = my_df["col1"] + my_df["col2"]

We can rename columns by passing a dictionary to the .rename() method.

- rename_dictionary = {"old_col_name": "new_col_name"}
- my_df.rename(columns = rename_dictionary)

Let's explore this in Jupyter!

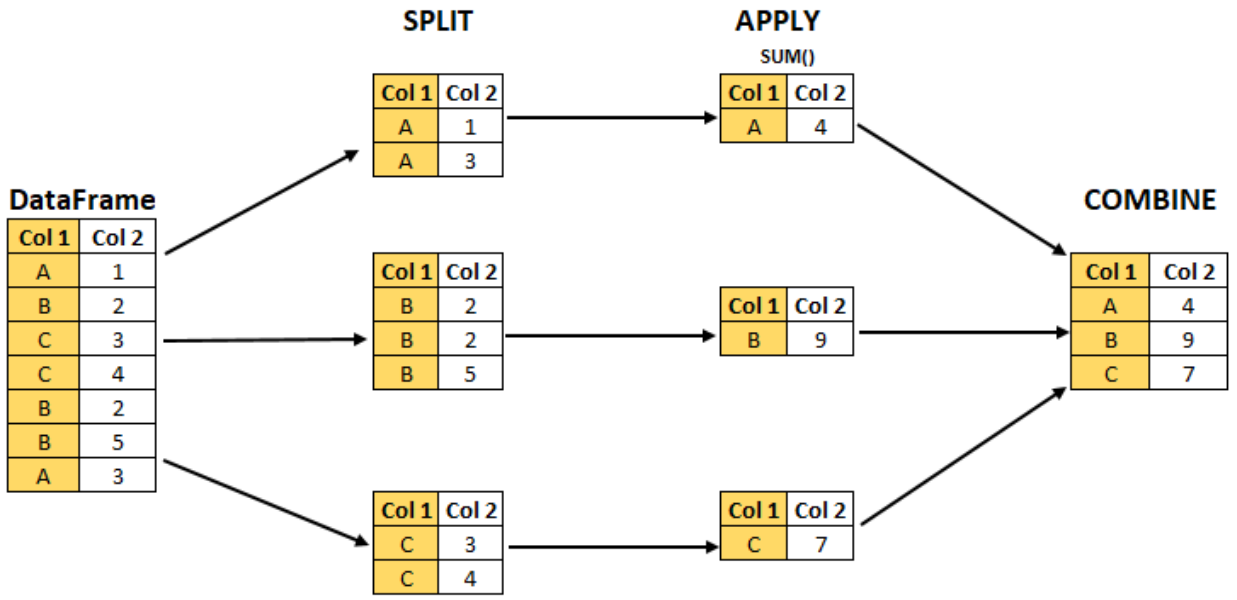
Creating aggregate statistics by group

We can get statistics separately by group using the .groupby() and .agg() methods

- E.g. dow.groupby("Year").agg("max")

This implements:

“Split-apply-combine”



Creating aggregate statistics by group

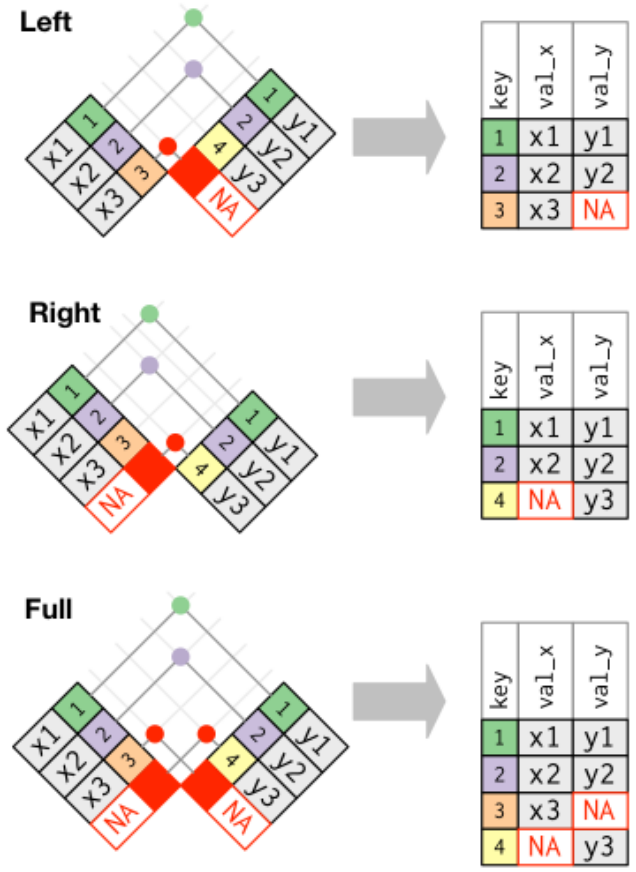
There are several ways to get multiple statistics by group

Perhaps the most useful way is to use the syntax:

```
my_df.groupby("group_col_name").agg(
      new_col1 = ('col_name', 'statistic_name1’),
      new_col2 = ('col_name', 'statistic_name2’),
      new_col3 = ('col_name', 'statistic_name3')
)
```

Let's explore this in Jupyter!

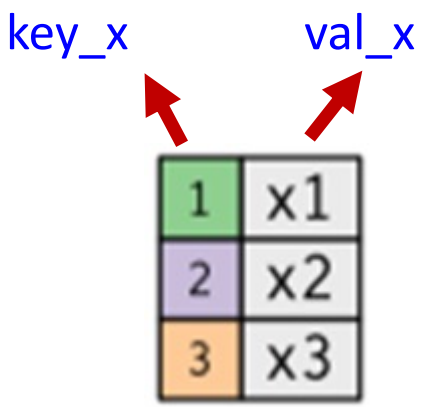
Joining data frames



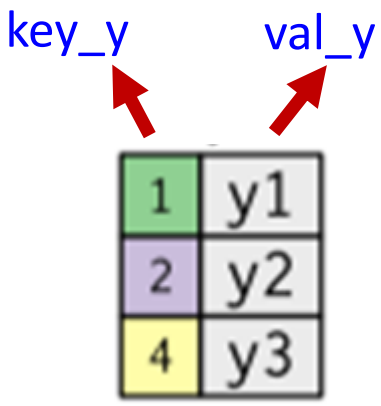
Left and right tables

Suppose we have two DataFrames (or Series) called **x_df** and **y_df**

- x_df have two columns called key_x, and val_x
- y_df has two columns called key_y and val_y



DataFame: x_df



DataFrame y_df

Joins have the general form:

```
x_df.merge(y_df, left_on = "key_x",  right_on = "key_y")
```

Left joins

Left joins keep all rows in the left table.

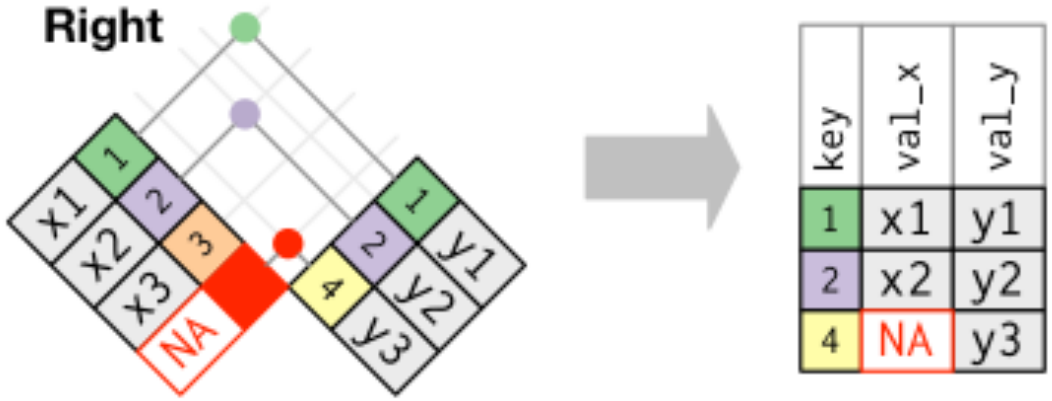
Data from right table is added when there is a matching key, otherwise NA as added.


```
x_df.merge(y_df, how = "left", left_on = "key_x",  right_on = "key_y")
```

Right joins

Right joins keep all rows in the right table.

Data from left table added when there is a matching key, otherwise NA as added.




```
x_df.merge(y_df, how = "right", left_on = "key_x",  right_on = "key_y")
```

Inner joins

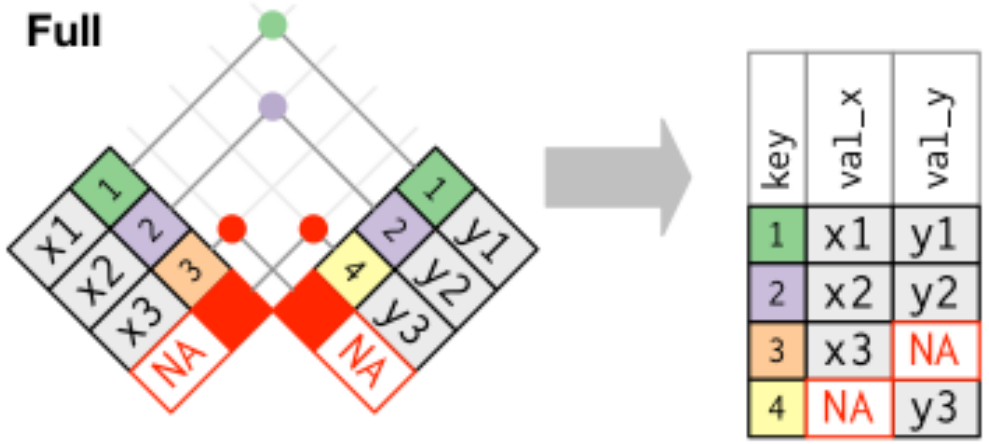
Inner joins only keep rows in which there are matches between the keys in both tables.


```
x_df.merge(y_df, how = "inner", left_on = "key_x",  right_on = "key_y")
```

Full (outer) joins

Full joins keep all rows in both table.

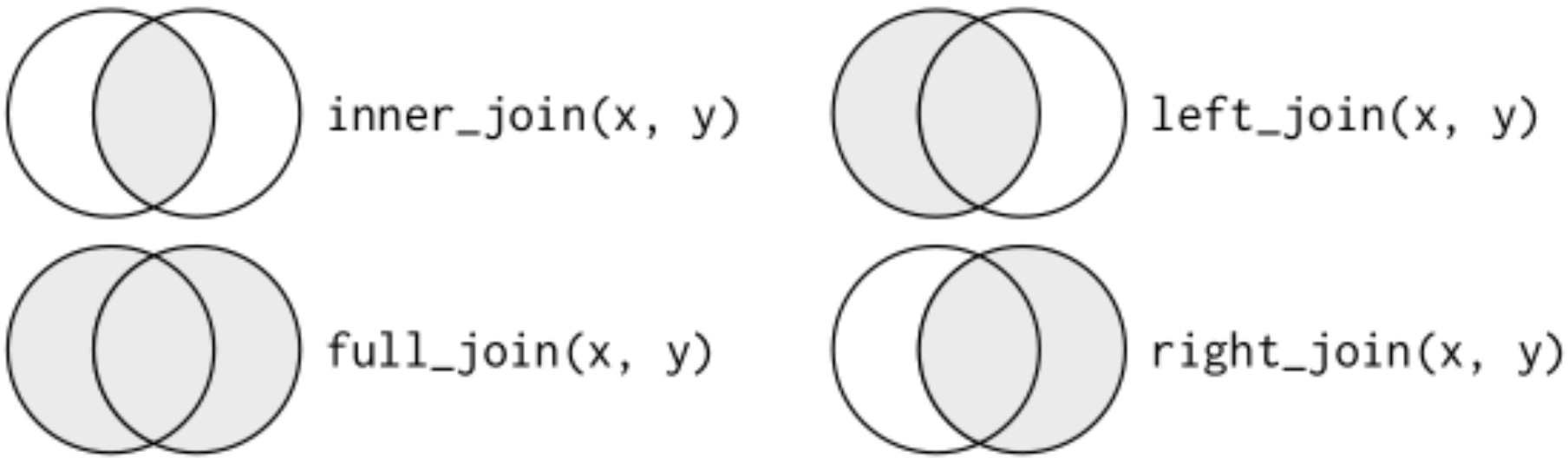
NAs are added where there are no matches.



key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

```
x_df.merge(y_df, how = "outer", left_on = "key_x",  right_on = "key_y")
```

Summary



Joining on Index values

If two DataFrames **have the same Index values** then we can join them using the .join() method instead of the .merge() method

The .join() method is very similar to .merge() except we don't need to specify left_on and right_on arguments since the DataFrames are being joined by their Indexes

An example of a left join would be:

- x_df.join(y_df, how = "left") # assuming x_df and y_df has the same Index values

Let's explore this in Jupyter!

Questions?