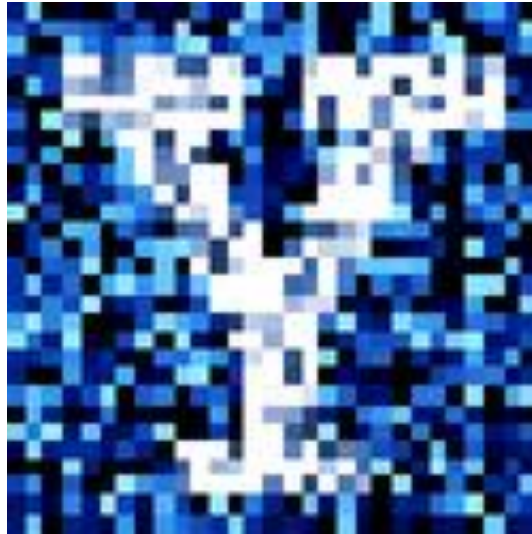


YData: Introduction to Data Science



Class 08: 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 4

Homework 4 is due on Gradescope on **Sunday September 28th at 11pm**

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

How was homework 3?

My office hours on Wednesday are cancelled but you can attend my S&DS
1000 office hours from 2pm-2:45pm

Quick review of Boolean indexing

We can use Boolean arrays to return values in another array

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

```
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!

Quick review of higher dimensional
arrays and Image processing

Higher dimensional arrays

We can make higher dimensional arrays

- (matrices and tensors)

```
my_matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

```
my_matrix
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

We can slice higher dimensional array

- `my_matrix[0:2, 0:2]`

We can apply operations to rows, columns, etc.

- `np.sum(my_matrix, axis = 0)` # sum the values down rows

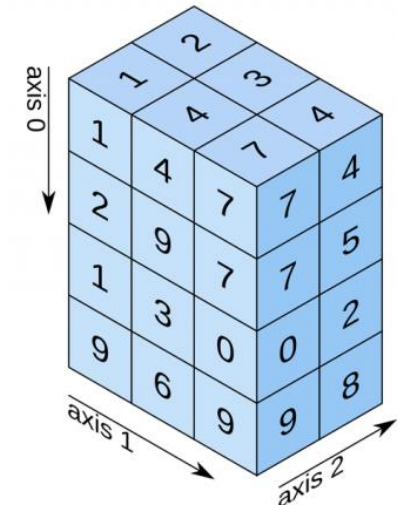
2D array



axis 1

shape: (2, 3)

3D array



shape: (4, 3, 2)

Image processing

We can use higher dimensional numpy arrays to store and manipulate images

Digital images are made up of pixels

Each pixel consists of a red (R), green (G), and Blue (B) color channel

- i.e., we have an “RGB image”
- See the [RGB color picker](#)

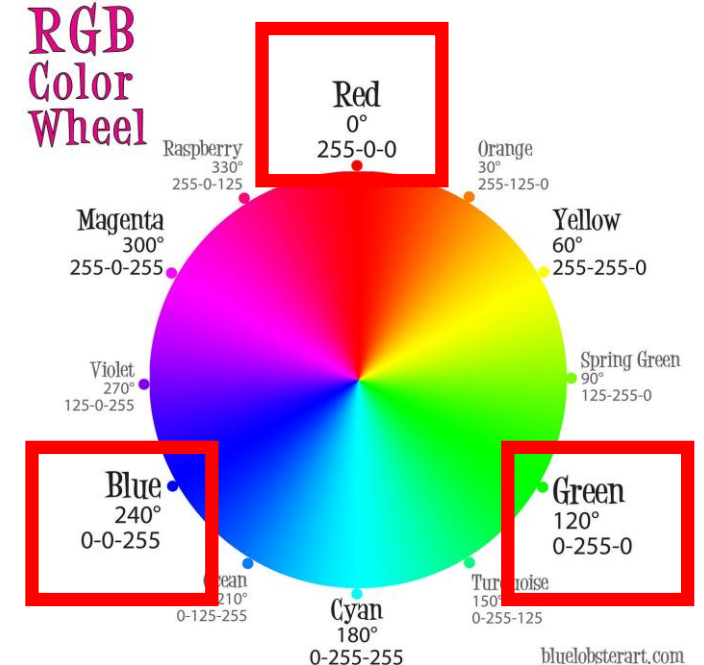
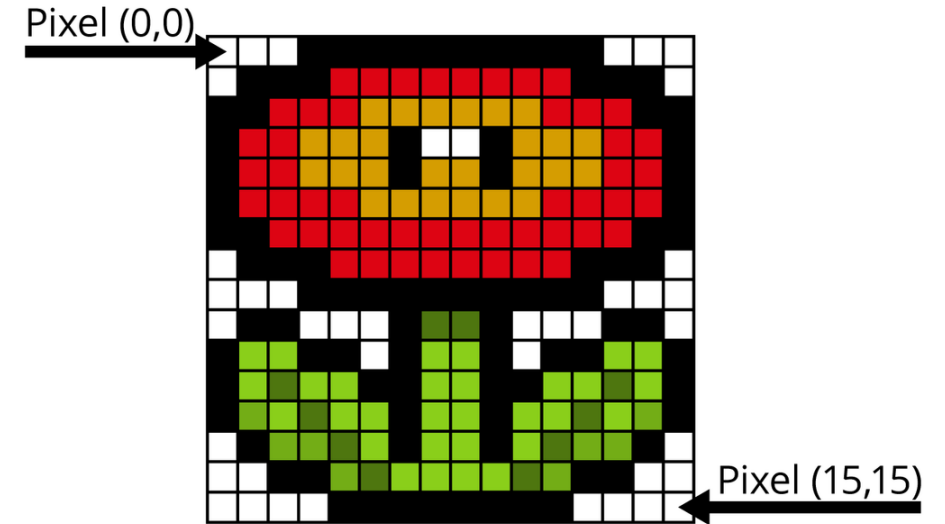
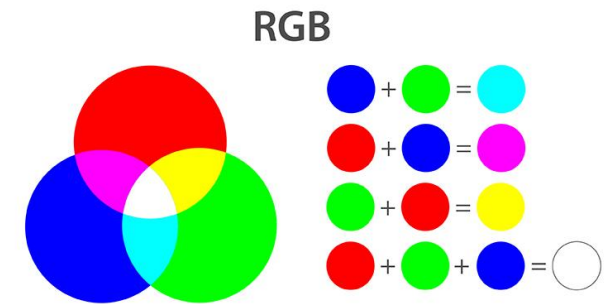
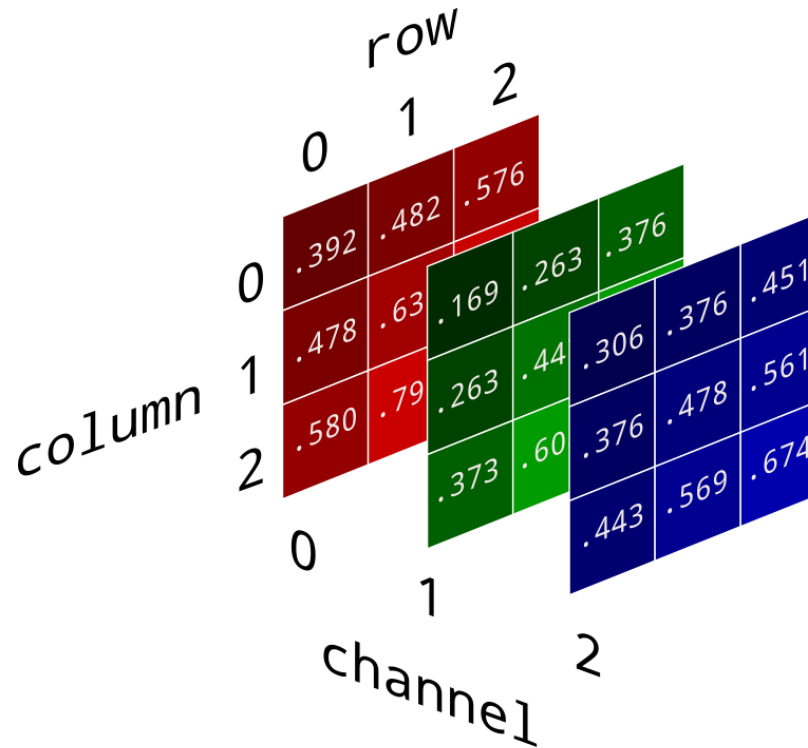


Image processing

We can use 3-dimensional numerical arrays to store digital RGB images

We can use masking and other array operations to process images



Let's quickly explore this 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 a package that is useful for processing time series and tables of data

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")
```

Variables

Cases

title	clean_test	binary
21 & Over	notalk	FAIL
Dredd 3D	ok	PASS
12 Years a Slave	notalk	FAIL
2 Guns	notalk	FAIL
42	men	FAIL

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 brackets. 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 )
```

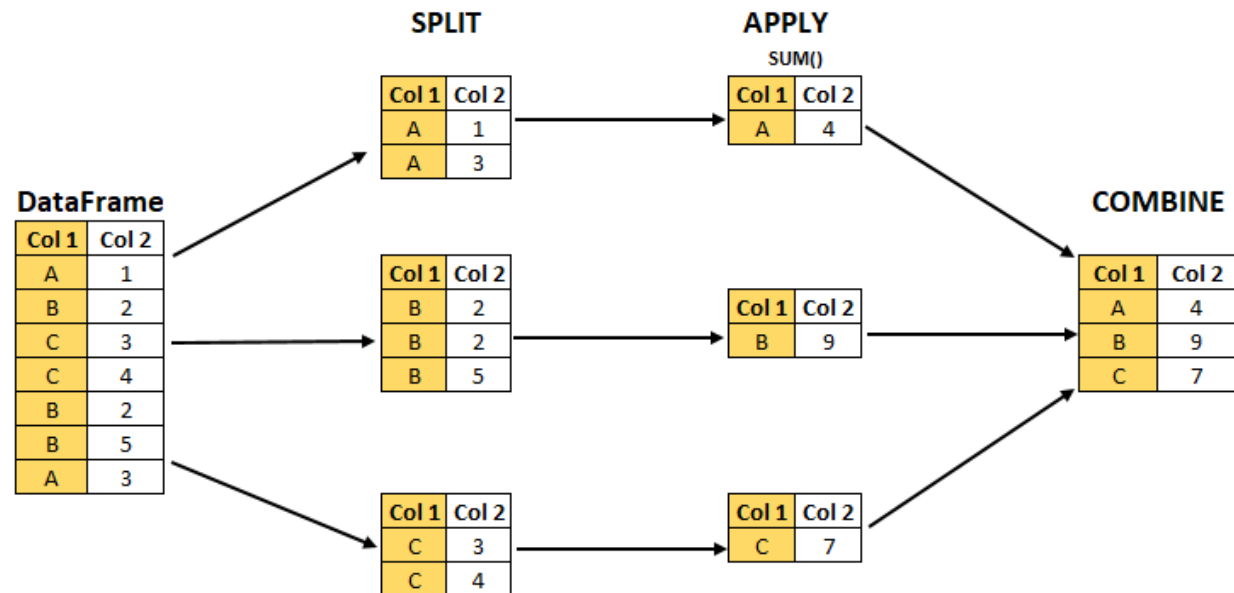
Let's explore this in Jupyter!

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

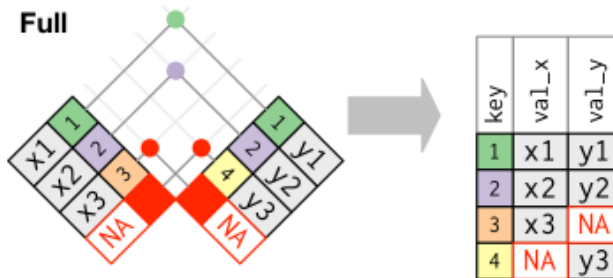
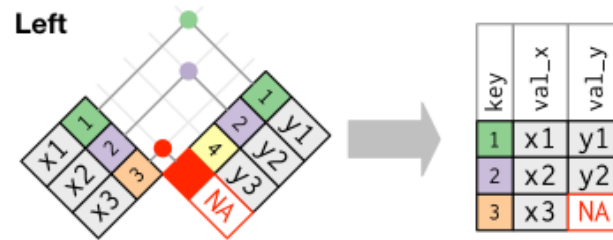
Perhaps the most useful way is to use the syntax:

```
my_df.groupby("group_col_name").agg(  
    new_col1 = ('col_name1', 'statistic_name1'),  
    new_col2 = ('col_name2', 'statistic_name2'),  
    new_col3 = ('col_name3', 'statistic_name3')  
)
```

```
nba_salaries.groupby("TEAM").agg(  
    max_salary = ("SALARY", "max"),  
    min_salary = ("SALARY", "min"),  
    first_player = ("PLAYER", "min")  
)
```

Let's explore this in Jupyter!

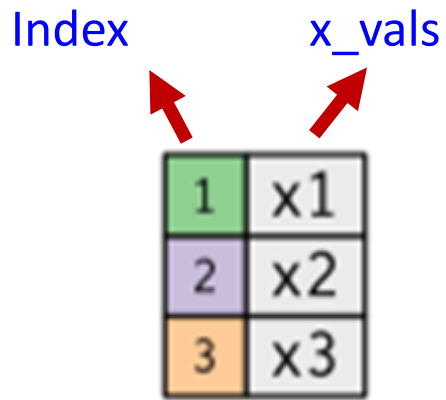
Joining data frames



Left and right tables

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

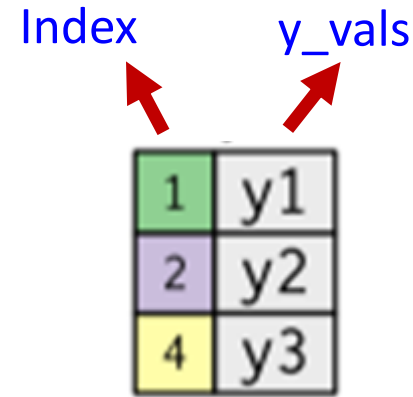
- **x_df** have one column called **x_vals**
- **y_df** has one column called **y_vals**



A diagram of a DataFrame with two columns. The first column is labeled 'Index' with a red arrow pointing to the first column. The second column is labeled 'x_vals' with a red arrow pointing to the second column. The data is as follows:

1	x1
2	x2
3	x3

DataFrame: x_df



A diagram of a DataFrame with two columns. The first column is labeled 'Index' with a red arrow pointing to the first column. The second column is labeled 'y_vals' with a red arrow pointing to the second column. The data is as follows:

1	y1
2	y2
4	y3

DataFrame: y_df

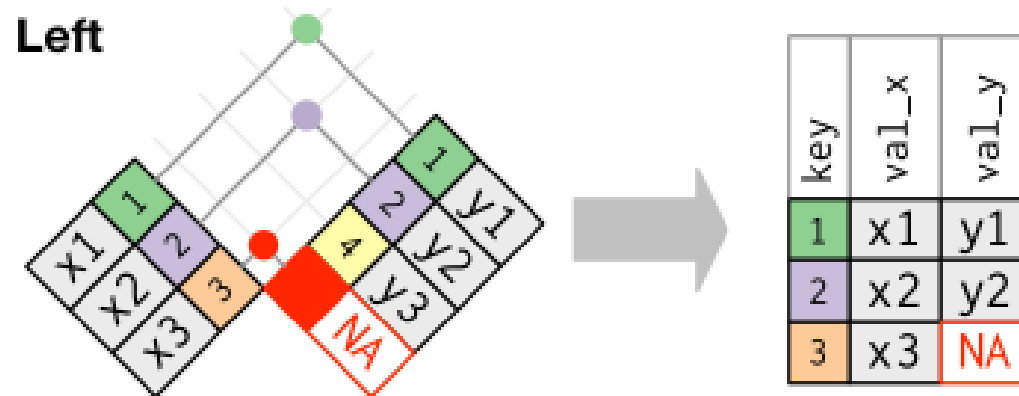
We can join these two DataFrames into a single DataFrame by aligning rows with the same Index value using the general syntax: **x_df.join(y_df)**

- i.e., the new joined data frame will have two columns: **x_vals**, and **y_vals**

Left joins

Left joins keep all rows in the left table.

Data from right table is added when there is a matching Index value, otherwise NA is added

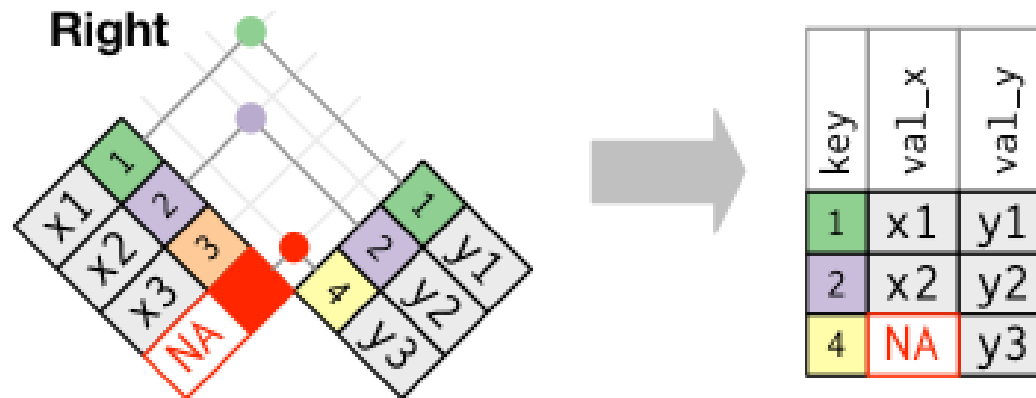


```
x_df.join(y_df, how = "left")
```

Right joins

Right joins keep all rows in the right table.

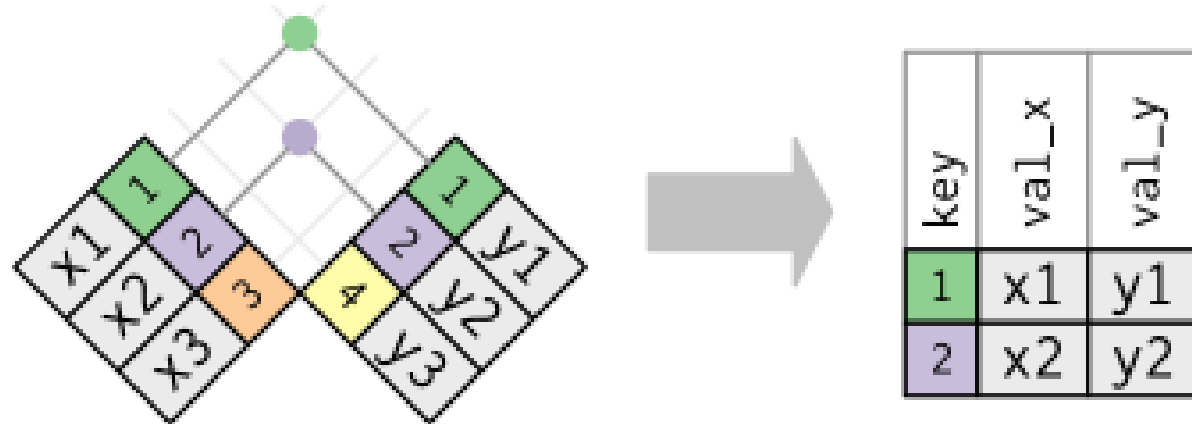
Data from left table added when there is a matching Index value otherwise NA as added



```
x_df.join(y_df, how = "right")
```

Inner joins

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

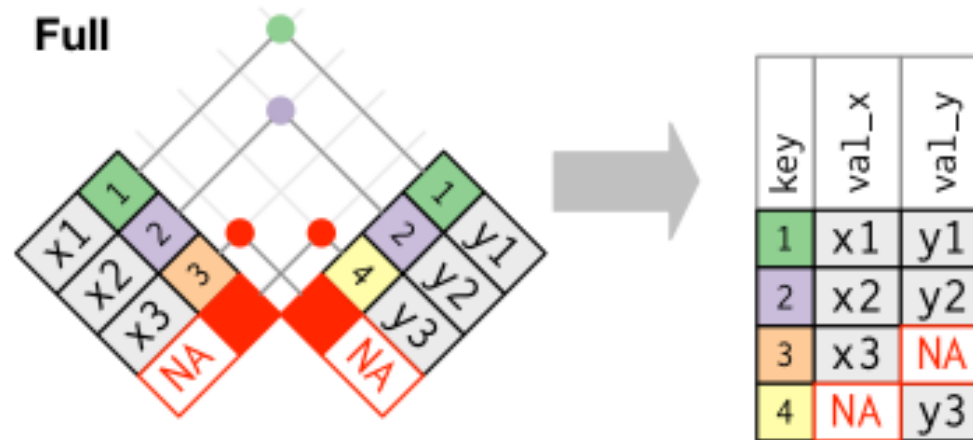


```
x_df.join(y_df, how = "inner")
```

Full (outer) joins

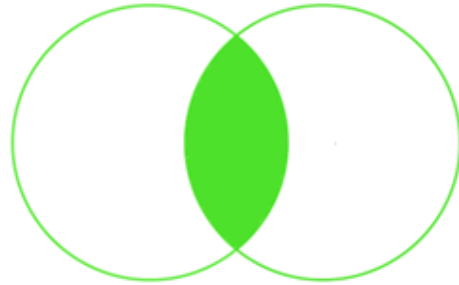
Full joins keep all rows in both table

NAs are added where there are no matches

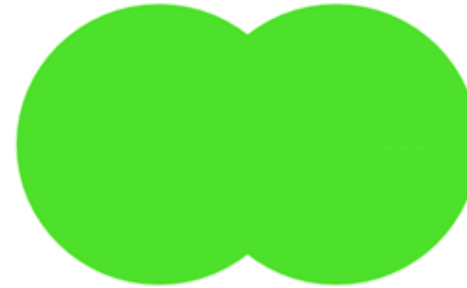


```
x_df.join(y_df, how = "outer")
```

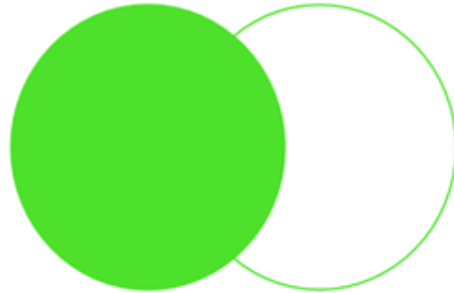
Summary



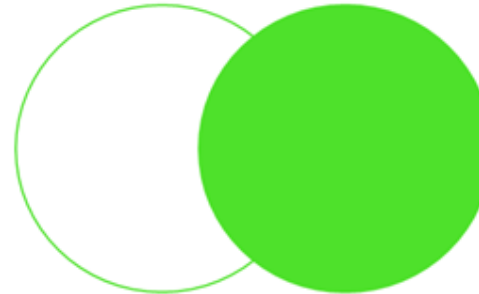
INNER JOIN



FULL OUTER JOIN



LEFT JOIN



RIGHT JOIN

“Merging” data frames

We can also join DataFrames based on values in *columns* rather than based on the DataFrames Index values

To do this we can use the merge method which has the form:

- `x_df.merge(y_df, how = "left", left_on = "x_col", right_on = "y_col")`

All the same types of joins still work

- i.e., we can do: left, right, inner and outer joins

Let's explore this in Jupyter!

Questions?