Python II

Yale SCHOOL OF PUBLIC HEALTH

Big Data Summer Immersion

Overview

Continuation of python basics:

- Comparisons
- Booleans

Some Important Libraries/Packages:

- Matplotlib.
- NumPy.
- Pandas.
- Functions and For Loops (if time permits).

Booleans and comparisons

Comparisons

Review

We can use mathematical operators to compare numbers and strings

- Results return Boolean values True and False

Comparison	Operator	True example	False Example
Less than	<	2 < 3	2 < 2
Greater than	>	3 > 2	3 > 3
Less than or equal	<=	2 <= 2	3 <= 2
Greater or equal	>=	3 >= 3	2 >= 3
Equal	==	3 == 3	3 == 2
Not equal	!=	3 != 2	2 != 2

True is equal to 1

False is equal to 0

True + True + False
is equal to…

2

We can compare strings alphabetically

- 'a' < 'b'

Let's explore this in Jupyter!

Plotting data

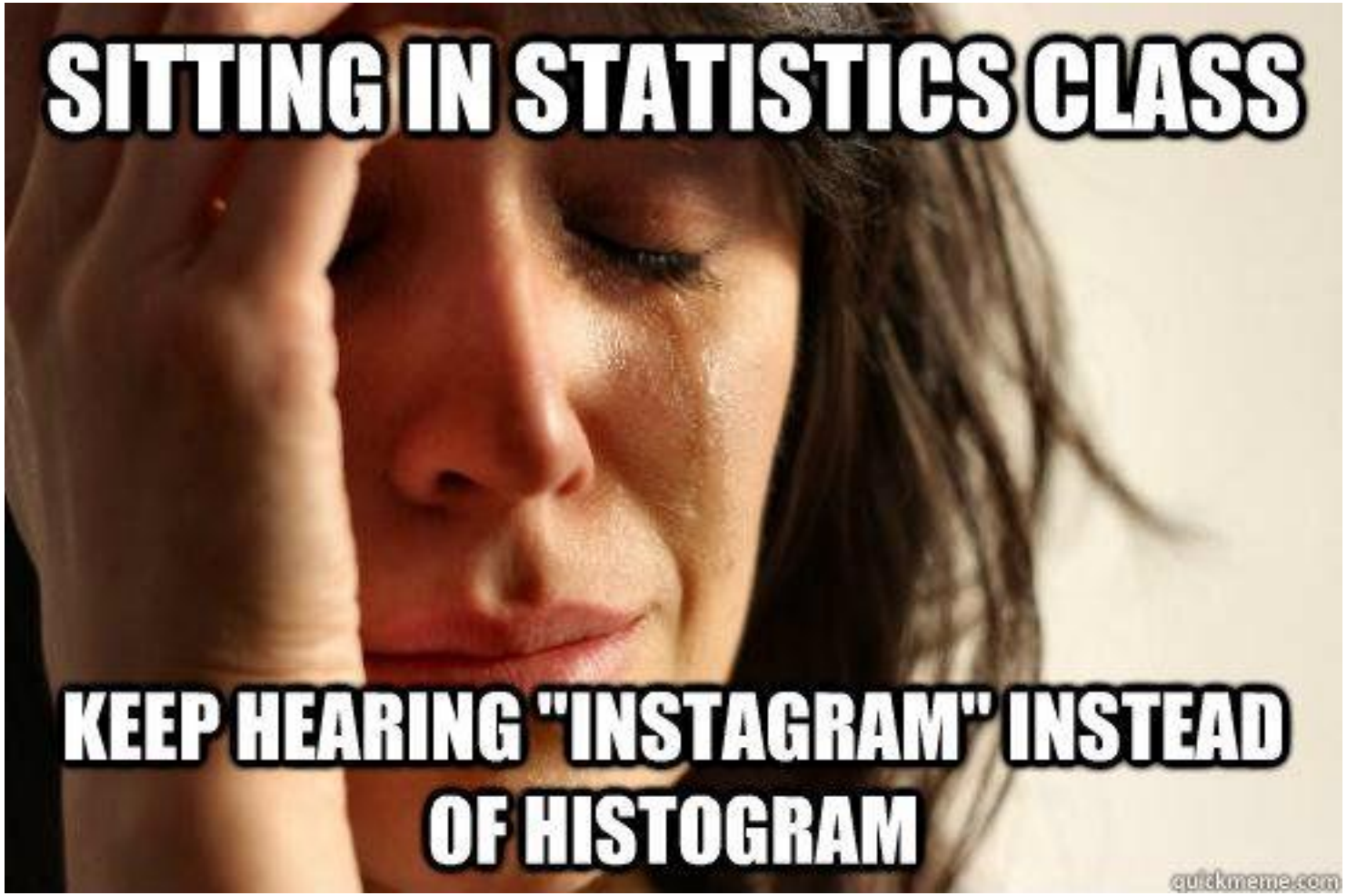
To create basic visualizations in Python we can use the matplotlib library

```
import matplotlib.pyplot as plt
```

We can then create plots using functions such as:

- plt.plot()
- plt.bar()
- plt.hist()
- etc.


```
plt.ylabel("y label")

plt.xlabel("x label")

plt.title("my title")
```



Let's explore this in Jupyter!

Array computations

Arrays

Often, we are processing data that is all of the same type

- For example, we might want to do processing on a data set of numbers
  - e.g., if we were analyzing salary data

When we have data that is all of the same type, there are more efficient ways to process data than using a list

- i.e., methods that are faster and take up less memory

In Python, the *NumPy package* offers ways to store and process data that is all of the same type using a data structure called a ***ndarray***

There are also functions that operate on ndarrays that can do computations very efficiently.

ndarrays

We can import the NumPy package using: `import numpy as np`

We can then create an array by passing a list to the *np.array()* function

- `my_array = np.array([1, 2, 3])`

We can get elements of an array using similar syntax as using a list

- `my_array[1]` `# what does this return`

ndarrays have properties that tell us the type and size

- `my_array.dtype` `# get the type of elements stored in the array`
- `my_array.shape` `# get the dimension of the array`
- `my_array.astype('str')` `# convert the numbers to strings`
- `sequential_nums = np.arange(1, 10)` `# creates numbers 1 to 9`

NumPy functions on numerical arrays

The NumPy package has a number of functions that operate very efficiently on numerical ndarrays

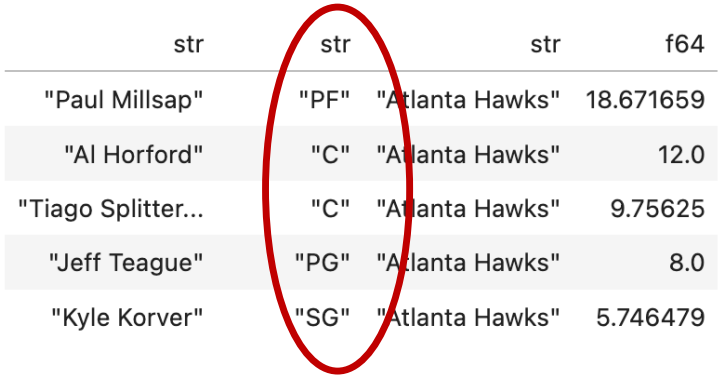
- np.sum()
- np.max(), np.min()
- np.mean(), np.median()
- np.diff() # takes the difference between elements
- np.cumsum() # cumulative sum

There are also "broadcast" functions that operate on all elements in an array

- my_array = np.array([12, 4, 6, 3, 4, 3, 7, 4])
- my_array * 2
- my_array2 = np.array([10, 9, 2, 8, 9, 3, 8, 5])
- my_array - my_array2

Boolean arrays

Categorical Variable



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

It is often to compare all values in an ndarray to a particular value

- my_array = np.array([12, 4, 6, 3, 4, 3, 7, 4])
- my_array < 5 # any guesses what this will return
  - array([False, True, False, True, True, True, False, True])

This can be useful for calculating proportions

- True == 1 and False == 0
- Taking the sum of a Boolean array gives the total number of True values
- The number of True 's divided by the length is the proportion
  - Or we can use the np.mean() function

$$\text{Proportion centers} = \frac{\text{number of centers}}{\text{total number}}$$

Let's explore this in Jupyter!

Boolean masking

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

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

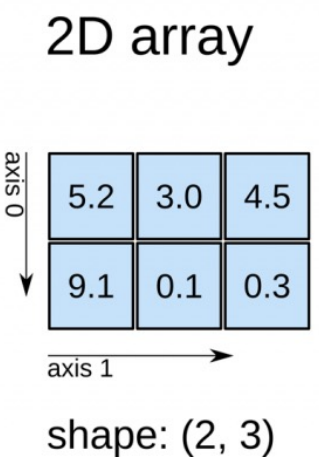
```
my_array = np.array([12, 4, 6, 3])
boolean_mask = np.array([False, True, False, True])

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?`

Higer dimensional arrays

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
```

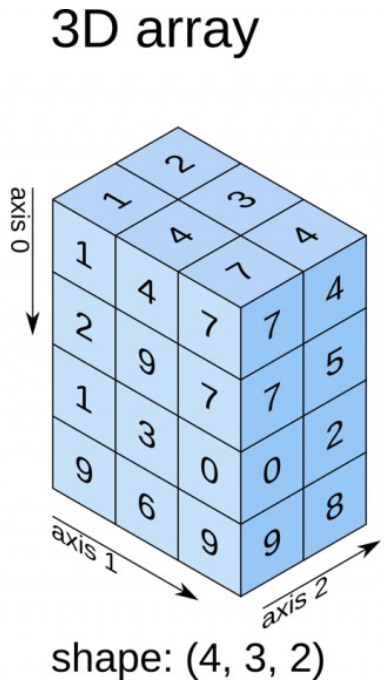
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

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



Let's explore this in Jupyter!

Image processing

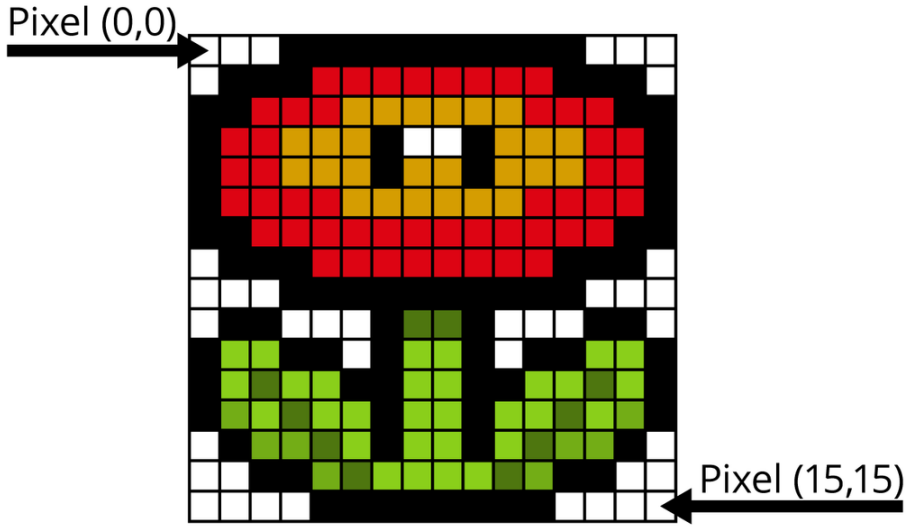
Image processing

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”



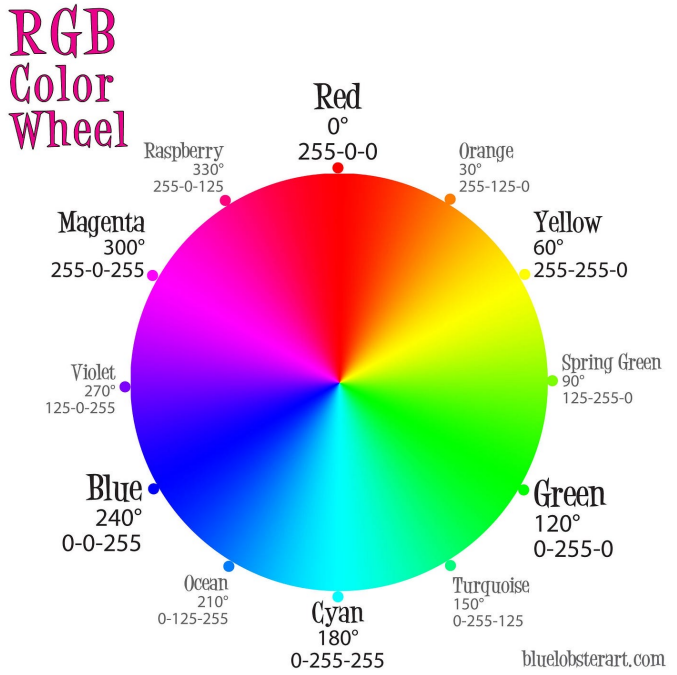
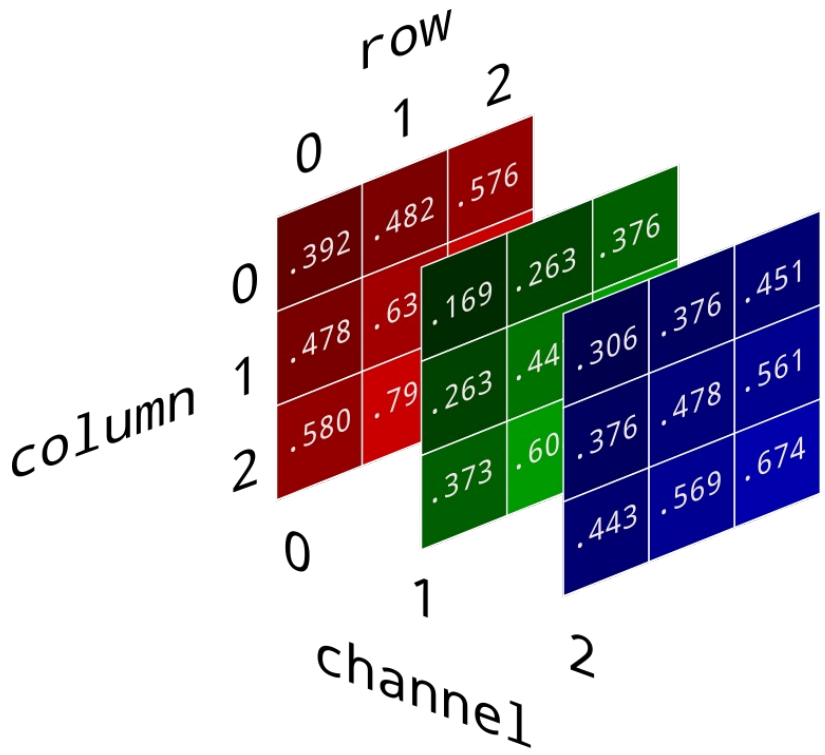
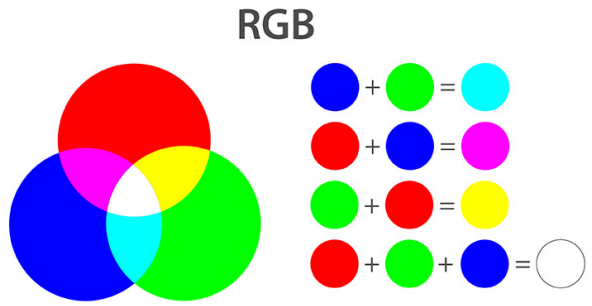


Image processing

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

We can use masking and other array operations to process images





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

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

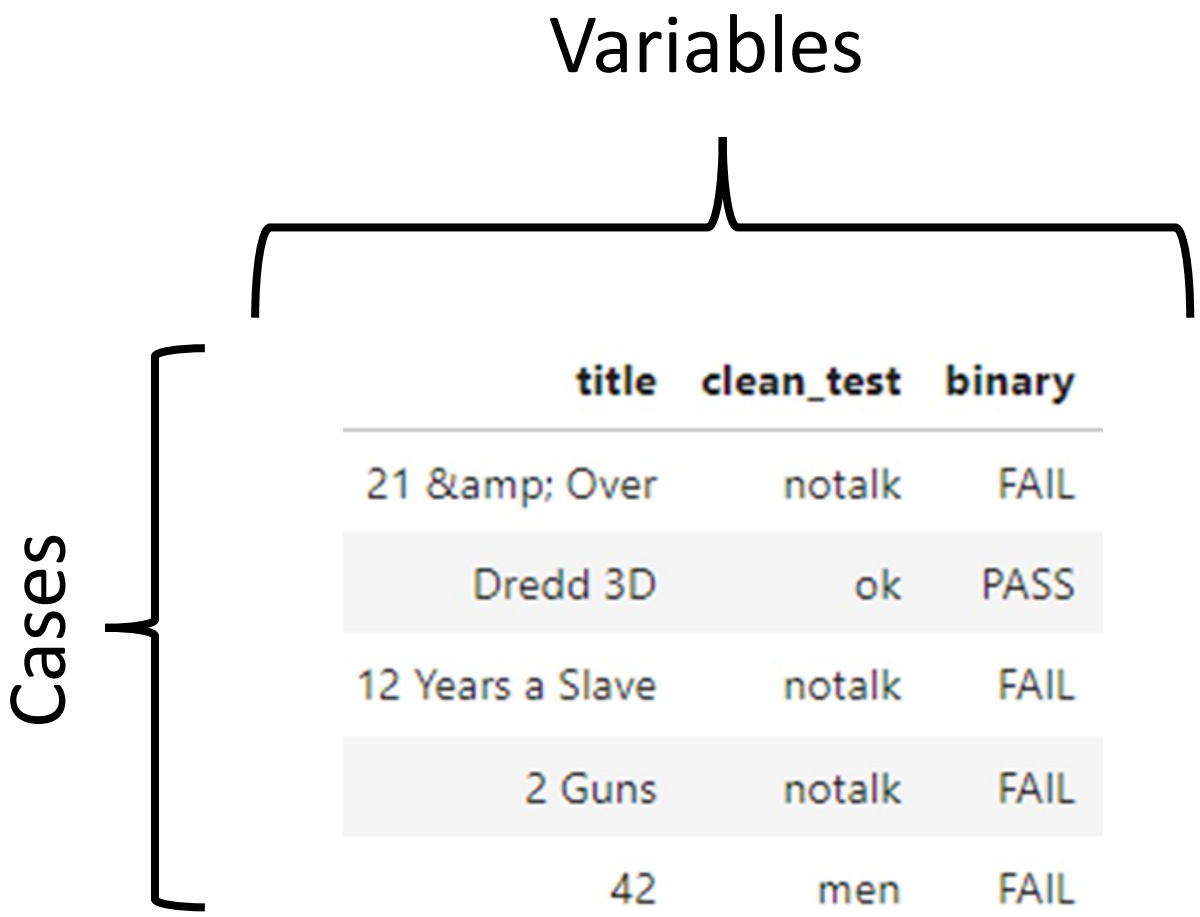
pandas DataFrames

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")
```



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

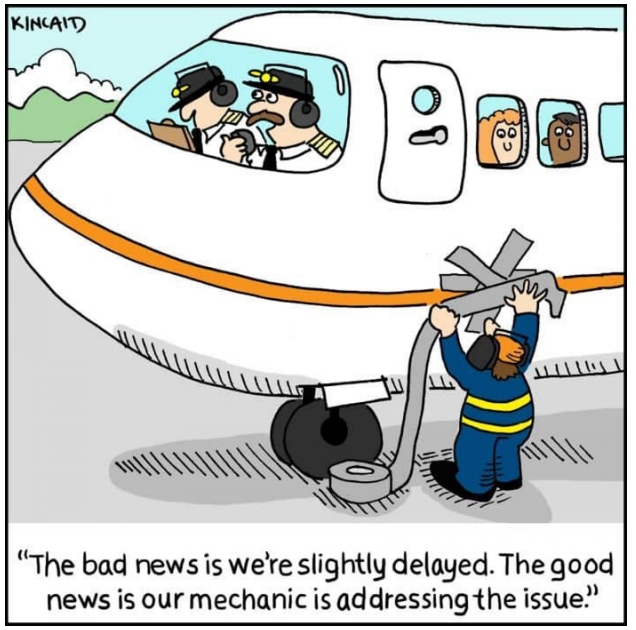
Extracting rows from a DataFrame

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 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!

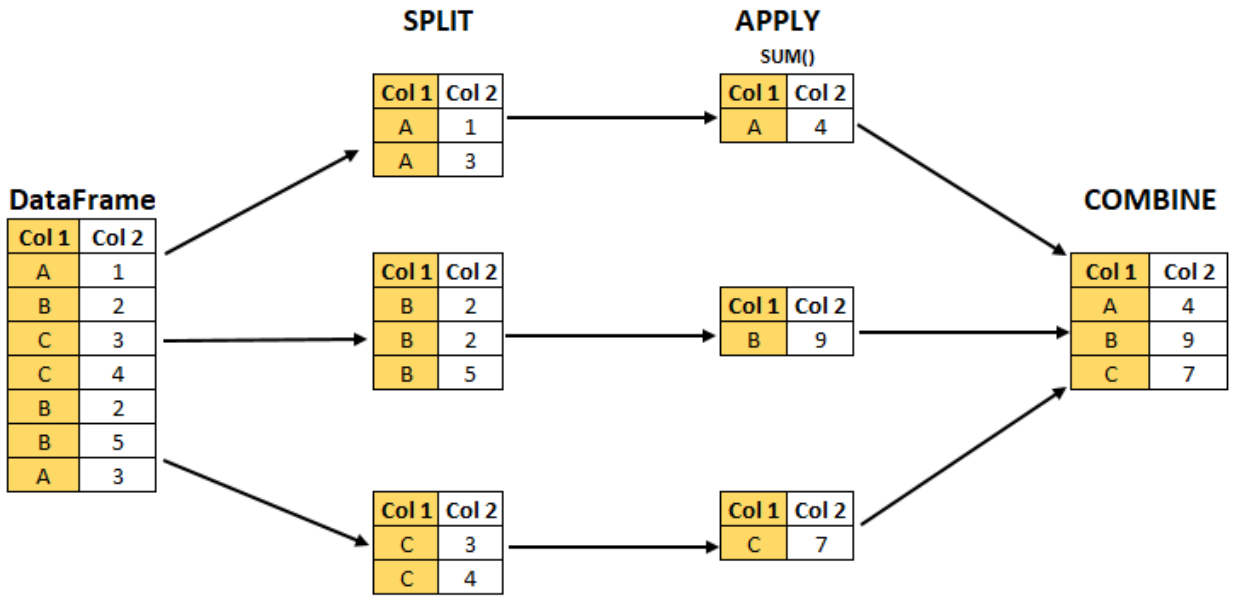
Creating aggregate statistics by group

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_name1', 'statistic_name1'),
      new_col2 = ('col_name2', 'statistic_name2'),
      new_col3 = ('col_name3', 'statistic_name3')
)
```

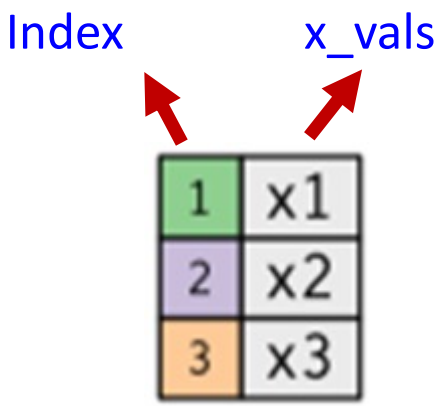
```
nba_salaries.groupby("TEAM").agg(
    max_salary = ("SALARY", "max"),
    min_salary = ("SALARY", "min"),
    first_player = ("PLAYER", "min")
)
```

Let's explore this in Jupyter!

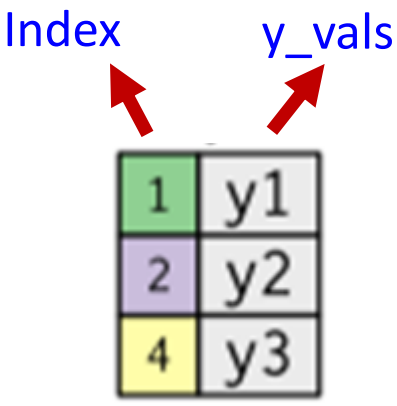
Left and right tables

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



DataFame: x_df



DataFrame: y_df

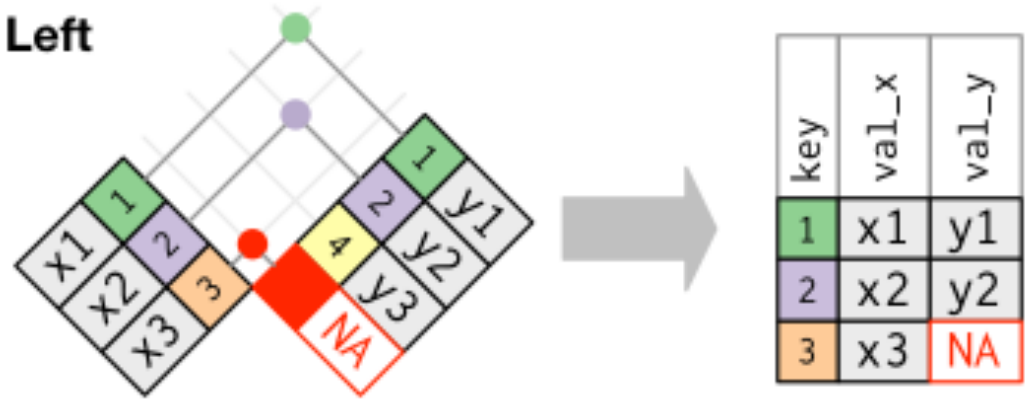
We can join these two DataFrames into a single DataFrame by aligning rows with the same Index value using the general syntax: x_df.join(y_df)

- 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 as added

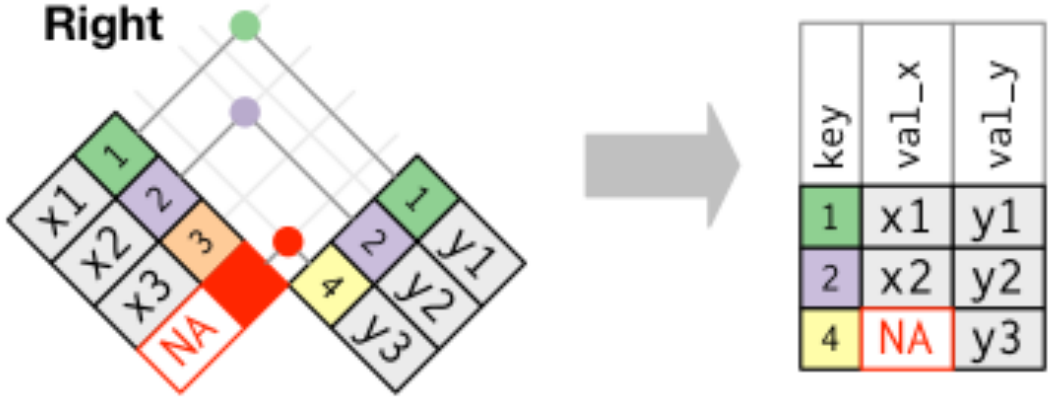


x_df.join(y_df, how = "left”)

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



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

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

Inner joins

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

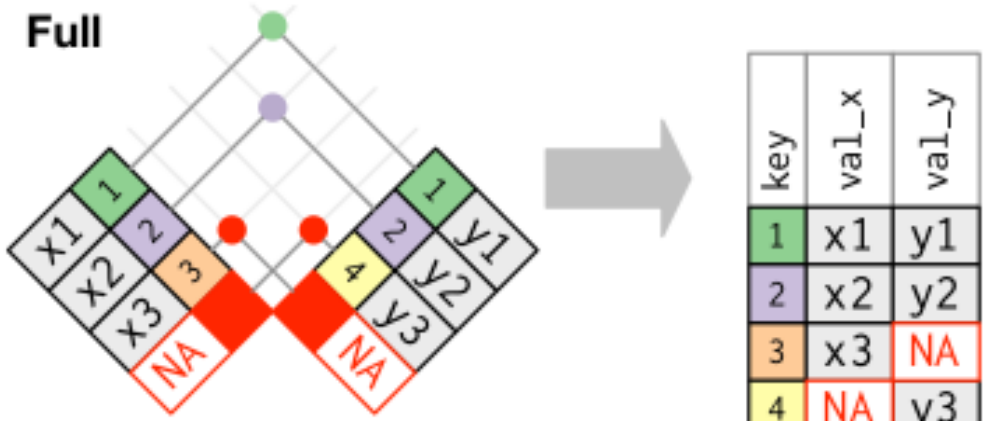
key	val_x	val_y
1	x1	y1
2	x2	y2

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

Full (outer) joins

Full joins keep all rows in both table

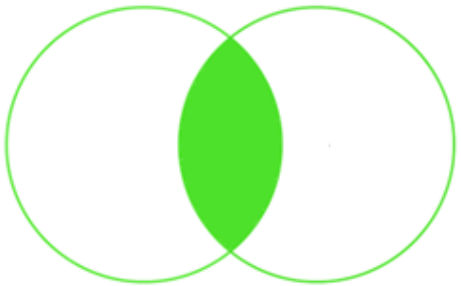
NAs are added where there are no matches



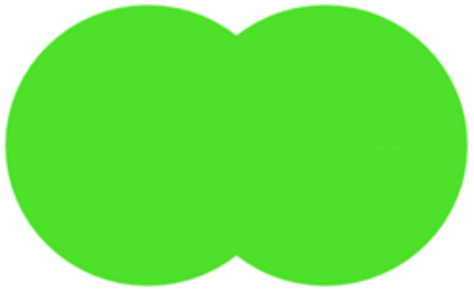
key	val_x	val_y
1	x1	y1
2	x2	y2
3	x3	NA
4	NA	y3

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

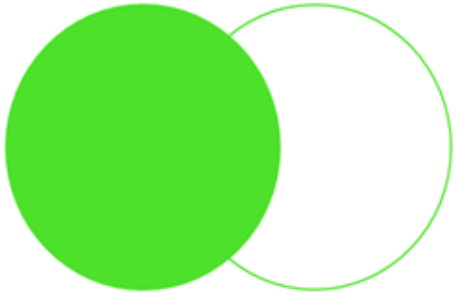
Summary



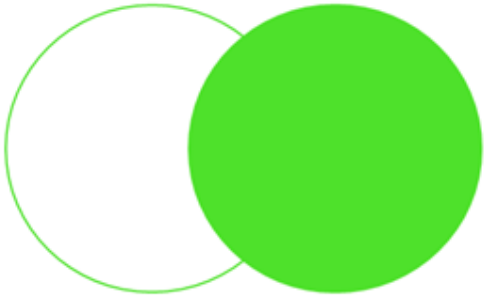
INNER JOIN



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!

Python Learning Resources

- Computational and Inferential Thinking
- Data Science for Everyone
- Elements of Data Science
- Python for Data Analysis

Acknowledgements

- Thanks to Stats Department @ Yale, Ethan Meyers, and UC Berkeley, Data 8 course.

- If you have questions related to Python, please reach out: shivam.sharma@yale.edu

- Thank You !!