Welcome to DSCI 101

Introduction to Data Science

Week 3 Recap

- Introduction to Python programming
  - Python
  - Jupyter notebook
- Basic data type in Python
  - Numbers and String
  - Numpy arrays
  - List, Dictionary and Functions
- RDBMS
  - SQL

Week 4 Preview

- Pandas data structures
  - Dataframe
  - Series
- Pandas commands
  - import
  - basic utility operations
  - index, mutate, and sort…

Intro to Pandas

- A module in Python
  - remember what a module is?

- Learning goals
  - key data structures in Pandas: pd.dataframe, series, indices
  - how to create these structures
  - how to index into these structures
  - other basic operations

- Lots of demos alone the way!
  - slides -> demo -> lab

Pandas data structures

- Dataframe: tabular data
- Series: column of tabular data
- Index: a sequence of row labels

can think of a dataframe as a collection of series that share the same index

Data Frame

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

Series

```
0      Obama
1     McCain
2      Obama
3     Romney
4    Clinton
5      Trump
Name: Candidate, dtype: object
```

Index

Indices (a.k.a row labels)

- Default choice is numeric: 0, 1, 2, … number of rows-1
  - np.array([0,1,2,…n-1)) or np.arrange(n)
  - but it can be any of the columns
  - it does not have to be all unique values (remember primary key?)

	Candidate	Party	%	Year	Result
0	Obama	Democratic	52.9	2008	win
1	McCain	Republican	45.7	2008	loss
2	Obama	Democratic	51.1	2012	win
3	Romney	Republican	47.2	2012	loss
4	Clinton	Democratic	48.2	2016	loss
5	Trump	Republican	46.1	2016	win

	Candidate	Party	%	Result
Year				
2008	Obama	Democratic	52.9	win
2008	McCain	Republican	45.7	loss
2012	Obama	Democratic	51.1	win
2012	Romney	Republican	47.2	loss
2016	Clinton	Democratic	48.2	loss
2016	Trump	Republican	46.1	win

Create a dataframe

- Two ways to create a dataframe in Pandas

- By columns:
  - df = pd.DataFrame({“column1”: [x_1, … x_n], “column2”: [y_1, … y_n], …},
    index = [0, 1, …n-1])
  - what’s inside { } is a Python dictionary!
- By rows:
  - df = pd.DataFrame([[row_1], …[row_n]],
    index = [0, 1, … n-1],
    columns = [“column1”, “column2”, …])

Read in files

- ***pd.read_csv***
- ***pd.read_html***
- ***pd.read_excel***

- More: ***pd.read_everything***

- Many arguments can be used, but mostly optional
  - ?***pd.read_csv***

Some basic operations

- ***df.columns***
  - return **all column names** of the table
- ***df.shape***
  - return row number and column number (dimension of table)
  - len(df) only return row number, so it df.shape[0]
- ***df.head(#), df.tail(#)***
  - return the **first / last # rows** of the table, default is 5
- ***df.column_name***
  - return one column as a series

Indexing columns

- It is very common to extract one or more columns from a dataframe
  - known as “indexing by column”
- This can be done by multiple ways, the easiest is to use [ ] operator
  - we will learn more ways of indexing
- Dataframe[here you put in column names you want, **as a list**]
  - ***df[[column_name_1, column_name_2, …]]***
- If you only want one column, the following are NOT the same
  - ***df[column_name]*** returns a series, same as ***df.column_name***
  - ***df[[column_name]]*** returns a dataframe

Indexing rows with slicing

- You can also index by row indices slicing
  - ***df[0: 3]*** yields row with indices 0, 1, 2
  - 0:3 means from 0 to 3, but not including 3
  - what if you want only one row, say first? df[0] deos NOT work!
  - need ***df[0:1]***
  - or you can do ***df.head(1)***

- As we learn more Pandas command, there are always multiple ways to do the same thing!

Indexing rows using Boolean

- Boolean: True / False

- What does the following return:
  - ***df[[False, True, False]]***
  - ***df[1:2]***

- Why on earth I want to use Boolean???
  - filtering!!!
  - more on this later

More column operations

- Add a column to the dataframe
  - ***df["new_column_name"] = [new column content]***
- Modify an existing column
  - ***df["existing_column_name"] = [new column content]***
- Rename a column
  - ***df.rename(columns = {"old_column_name" : "new_column_name"})***
- Drop a column
  - ***df.drop(columns = ["column_x", "column_y"])***

Df.loc and df.iloc

- How do I index column and row at the same time???

- df.loc and df.iloc are alternative ways to index into a Dataframe
  - take a lot to getting used to
  - some weird usages can be quite complex
  - we only cover the common ones

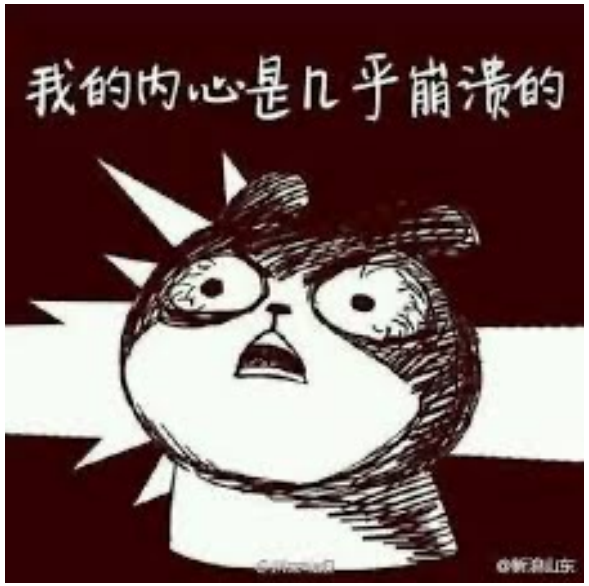
- Python documentations:
  - ***df.loc***
  - ***df.iloc***
  - ***general on indexing and selecting***

Df.loc

- The easiest way of using df.loc
  - ***df.loc[[a list of row indices], [a list of column names]]***
  - if you don't put row indices, default is selecting all rows

- df.loc can also use with slicing
  - ***df.loc[0:10, "column_1":"column_5"]***
  - no need to put them in a list
  - however ***df.loc*** slicing will include both ends……

- So ***df[0:5]*** and ***df.loc[0:5]*** are different!!!



[] and df.loc with Boolean arrays

- ***df[df["column1"] == value1]***
- ***df.loc[df["column1"] == value1]***
- These two return the same results! YaY
- Think about excel spread sheet, filtering column1 on value1
- But the filtering condition can get as complicated as you want…

Df.iloc: i for integer

- Integer-based indexing for selection by position

- df.loc looks for labels
  - column labels are column names
  - row labels are row indices

- df.iloc looks for positions, thus integer
  - Example: **if index is not numerical**
  - ***df.loc[0:2]*** return empty table
  - ***df.iloc[0:2]*** return first 3 rows

Df.loc or df.iLoc, this is a question…

- I would use loc 99% of the time….
  - easier to read code and harder to make mistakes
  - less vulnerable to changes of the ordering of rows/columns in raw data

- However, iloc can be more convenient sometimes… example

- Use iloc judiciously!

Sort by column or index

- Sort a specific column:
  - ***df.sort_values("column_name")*** in ascending order by default
  - ***df.sort_values("column_name", ascending=True)*** in descending order

- Sort the index:
  - ***df.sort_index*** will return a series