DATA MANIPULATION AND VISUALIZATION IN PYTHON

UTSC DataFest, April 30 2019

Moshe Gabel

(with many thanks to Sotirios Damouras, Irv Lustig)

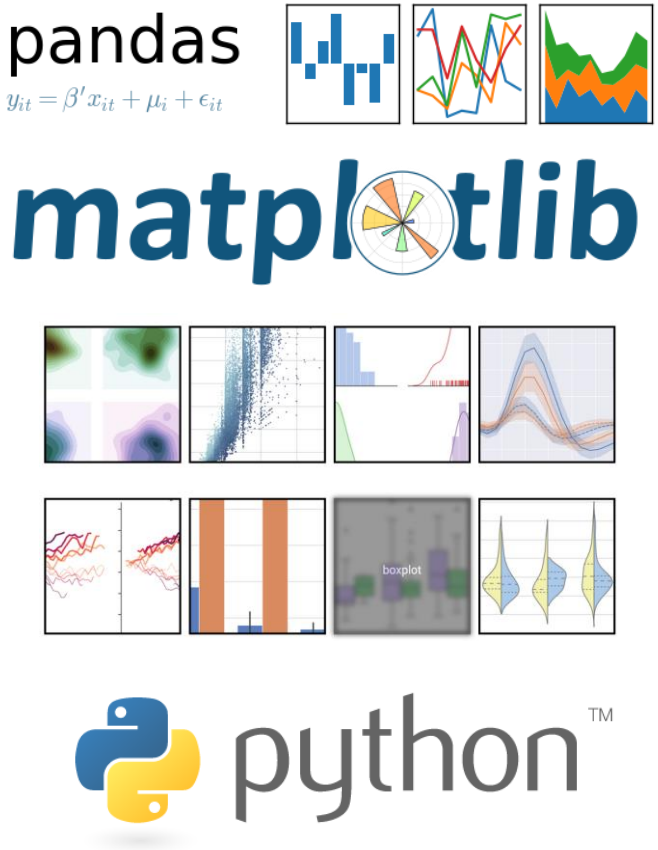
FOLLOW ALONG

- Find slides here: https://tinyurl.com/PyDataVizSlides

- All workshop material: https://tinyurl.com/PyDataViz

WHAT IS THIS ABOUT?

- Learn basic tools of data science in Python
- **pandas** for data manipulation
  - Load datasets.
  - Slice, filter, group, and aggregate data.
- **matplotlib** and **seaborn** for visualization.
- **Python** glues it together.



WHO IS THIS FOR?

- I assume you:
  - Have some ability with Python programming.
  - Are comfortable looking things up.
  - Know how to work with files (just the basics).

- You want to learn basics of:
  - Working with tabular data in Python.
  - Visualizing data.

WHY PYTHON?

- Intuitive, easy to learn and use.
- General purpose.
- Vibrant, mature ecosystem.
- Huge amount of tooling.
- Industrial strength.

- R is a very popular alternative.
- Better stats support.
- Less general.

WHICH LANGUAGE TO USE?

- An objective, scientific comparison!

- Sort an array in **Python**:
  `np.sort(x)`

- Filter values in **Python**:
  `df[df.month == "January"]`

WHICH LANGUAGE TO USE?

- An objective, scientific comparison!

- Sort an array in **R**:
  `sort(x)`

- Filter values in **R**:
  `filter(df, month == "January")`

WHICH LANGUAGE TO USE?

- An objective, scientific comparison!

- Sort an array in **COBOL**:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SORT01.
AUTHOR. SHIBU.T.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 TBL.
 02 WS-TBL OCCURS 10.
 05 WS-FLD PIC 99.
 05 WS-FLD1 PIC X(3).
 05 WS-FLD2 PIC 99.
01 WS-TAB-HLD.
 05 WK-FLD PIC 99.
 05 WK-FLD1 PIC X(3).
 05 WK-FLD2 PIC 99.
```

WHICH LANGUAGE TO USE?

- R or Python? STOP WORRYING ABOUT IT!
- Choose the tool that works for you.
- I prefer Python.

(with thanks to Rob Story: https://medium.com/@oceankidbilly/python-vs-r-vs-cobol-which-is-best-for-data-science-7b2979c6a000)

LOW LEVEL TOOLS

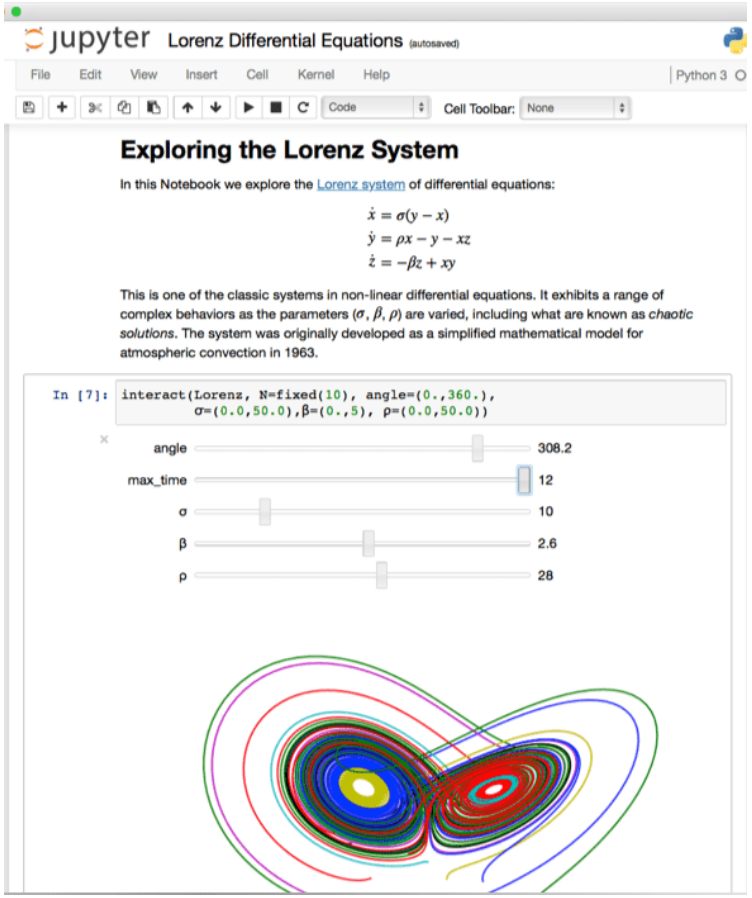
- **numpy** – fast numerical computation for Python
  - matrices, arrays, linear algebra, basic stats, etc.
- **matplotlib** – basic plotting and visualization
  - Supports many basic plotting
  - A bit low-level – focuses on plotting, not data.
- In theory, those are enough…

HIGHER LEVEL TOOLS

- **pandas** – very popular tool for data wrangling in Python
  - Arrange data in tables (DataFrames)
  - Uses numpy internally
- **seaborn** –beautiful plots from pandas tables
  - Uses matplotlib internally
- **Jupyter** – for notebooks. What are those?

NOTEBOOKS

- You are used to coding Python scripts, running them inside an IDE or from command line.
- An alternative is a **notebook**: code, visualization, comments, explanations in a single page!



USING NOTEBOOKS

- We will use a Juptyer Python notebook
  - You don’t have to, but it’s easier to follow along.
- Option 1: run Jupyter locally by installing the needed packages and running it.
  - This is faster
- Option 2: go to https://utoronto.syzygy.ca/ and sign in with UTorID to get your very own notebook without installing anything.

START YOUR ENGINES!

Open Jupyer

Get “worksheet.ipynb” notebook and data from https://tinyurl.com/PyDataViz

PANDAS BASICS

Let’s dig in…

DATAFRAMES

- Our basic data object is a **DataFrame.**
- A DataFrame is like a table: a collection of columns with an **index**.

Index	A	B	C	D
4	3	Slow	0.3	Jan 1
7	5	Fast	0.6	Jan 2
1		Fast	NaN	Jan 2
3	5	Fast	NaN	Mar 3
10	1		3.1	Mar 4

- Each column is a **Series** (like array with index).

CREATING A DATAFRAME

```
import pandas as pd
df = pd.DataFrame(dict(group=["A","B","C"],
                       drug_X=[130, 140, 135],
                       drug_Y=[np.NaN, 150, 135]))
```

We can represent empty numerical data as NaN

	group	drug_X	drug_Y
0	A	130	NaN
1	B	140	150
2	C	135	135

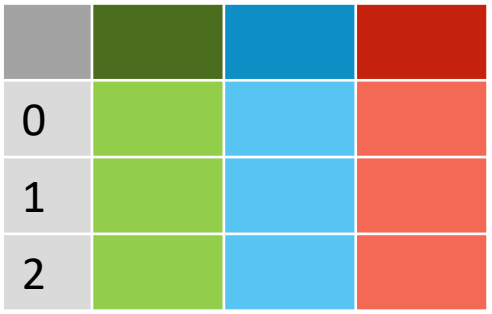
DATAFRAME BASICS

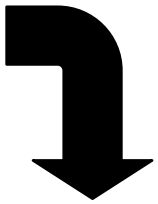
- Shape: `df.shape` → (#rows, #columns)
  - Does not include index, headers
  - `len(df)` is the number of rows
- Each column has a type (like numpy dtype)
- Examine with `df.dtypes`
  - int64, float64, etc – numerical
  - object – string or mixed type
  - Others: bool, datetime64[ns]…

group	object
drug_X	int64
drug_Y	float64

BASIC STATISTICS

- Column statistics: `df.describe()`





Only numerical columns!

	drug_X	drug_Y
count	3.0	2.000000
mean	135.0	142.500000
std	5.0	10.606602
min	130.0	135.000000
25%	132.5	138.750000
50%	135.0	142.500000
75%	137.5	146.250000
max	140.0	150.000000

SELECTING A COLUMN

- A little like dictionary, column name as key:
  `df["drug_X"]`

	group	drug_X	drug_Y
0			
1			
2			

- This returns a **Series**
  - A little like a 1D array of values, with an index.

SLICING DATAFRAMES

- Picking subset of rows, columns….
- By integer position:
  `df.iloc[0:2, 1:]`

	group	drug_X	drug_Y
0			
1			
2			

.iloc matches Python behaviour

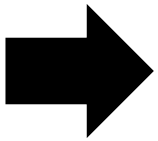
- By index/name
  `df.loc[0:1, "drug_X":]`

.loc index is **INCLUSIVE**

COMPLEX SLICING

- Slices can be non-contiguous slices
  `df[["group","drug_Y"]]`
  - This creates a copy

	group	drug_X	drug_Y
0			
1			
2			



	group	drug_Y
0		
1		
2		

- Also works with rows.

SELECTING ROWS

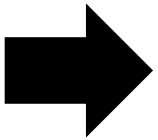
	group	drug_X	drug_Y
0			
1			
2			

- With list of indices:
  `df.loc[[0,2]]`
- …Or a list Booleans:
  `df[[True, False, True]`
- Useful methods:
  - `iloc[]` – by position
  - `sample()` – sample randomly
  - `drop_duplicates()` – remove duplicated rows
  - `nsmallest()` , `nlargest()`
  - `head()` , `tail()`

ADDING COLUMNS

- Add a new column:
  `df["colname"] = [130, 150,200 ]`
- **You can treat columns as vectors**:
  `df["colname"] = df["drug_X"] * 10`

	group	drug_X	drug_Y
0			
1			
2			



	group	drug_X	drug_Y	colname
0				
1				
2				

INPUT/OUTPUT

- Usually you'd use pandas to read existing data.
- Pandas supports **many** I/O schemes:
  - SQL, JSON, HTML tables, Parquet, HDF5, Excel, …
  - Some of these are industrial strength.
- We will use the humble CSV file: `pd.read_csv(filename, ...)`
- There is also a `.to_csv(…)` method

pd.read_csv

- Reading a CSV sounds simple...
- But there are many possible configurations:
  - With header? Is there an index column?
  - What are the datatypes? Should we parse dates?
  - Missing data?
- TIP**: always** examine the source file and the resulting DataFrame and dtypes **carefully**.
  - Be ready to adjust `read_csv()` arguments

DO TASKS IN BLOCK 1

Photo by Petr Kratochvil

WORKING WITH DATA

OUR DATAFRAME

- Quick reminder:

	group	drug_X	drug_Y
0	A	130	NaN
1	B	140	150.0
2	C	135	135.0

AGGREGATING DATA

- Manually by computing over columns:
  `df["drug_X"] + df["drug_Y"]`

drug_Y has missing number, so the sum cannot be computed

	sum
0	NaN
1	290.0
2	270.0

- DataFrames provides aggregation:
  `df[["drug_X","drug_Y"]].sum(1)`

axis=0 → sum over rows
axis=1 → sum over columns

	sum
0	130.0
1	290.0
2	270.0

AGGREGATING DATA

- pandas provides many aggregate operations:
  `sum, count, mean, median, std, min, max`
  - These are fast, and ignore NaNs
- Aggregate over Series:
  ```
  df['drug_X'].mean()
  ```
- User-defined aggregation:
  ```
  df.agg(lambda s: len(s))
  ```
- Aggregate over: sliding windows (`rolling`), groupings (`GroupBy`), cumulative windows (`expanding`), exponentially weighted windows (`ewm`)

COUNTING VALUES

- Number of unique (distinct) values in Series:
  `df['group'].nunique()` → 3
- Get array of unique values:
  `df['group'].unique()` → `['A', 'B', 'C']`
- Count values in a series:
  `df['group'].value_counts()`
  - (more on this later)

A	1
B	1
C	1

THE INDEX

- DataFrames have an **index**
  - Each row has a **label**: `df.loc[label]`
- Index can be unsorted, have duplicates.
- Labels are not just numbers:
  - Strings, times, ranges,…
- Index carries into series: df["drug_X"] is indexed the same as `df`

	Group	drug_X	drug_Y
test A	A	130	NaN
test B	B	140	150
control	C	135	135

	drug_X
test A	130
test B	140
control	135

USING INDEXING

- Change index to column:
  `df.set_index('group')`

Group	drug_X	drug_Y
A	130	NaN
B	140	150
C	135	135

- Reset index to numbers:
  `df.reset_index()`

	group	drug_X	drug_Y
0	A	130	NaN
1	B	140	150
2	C	135	135

- Get the index object:
  `df.index`

ONE USE OF INDEXING

- Count the number of rows with each value: `df["group"].value_counts()`
  - NaNs are not included!

A	1
B	1
C	1

- Result: a series with original values as the index, and the count as the value: `df["group"].value_counts().loc["B“]` → `1`

SORTING ROWS

- You can order rows by index:
  `df.sort_index()`
- Order rows by columns:
  `df.sort_values('drug_X')`

sorted by drug_X

index is no longer sorted: row labels are preserved, row position is not!

	group	drug_X	drug_Y
0	A	130	NaN
2	C	135	135.0
1	B	140	150.0

ITERATING OVER DATAFRAME

- Iterate over column name, series pairs

```
for col_name, series in df.items():
    print (col_name, series.tolist())
```

  - Like dictionary.

- Iterate over rows as tuples:

```
for index, group, x, y in df.itertuples():
    print(group, x, y)
```

MULTI-INDEX

- You can have sub indexes (**levels**)
- `df["intervention"]`
- `df[("intervention","drug_Y")]`
- Beyond the scope of this workshop.

		group	intervention	
			drug_X	drug_Y
Tests	test A	A	130	NaN
	test B	B	140	150
control		C	135	135

FILTERING DATA

- Filtering is just indexing with Boolean array
  `df.loc[ df["drug_X"]>132 ]`
  - Other algebraic operators:
    `<` `>` `==` `!=` `<=` `>=` etc.

	group	drug_X	drug_Y
1	B	140	150
2	C	135	135

- Complex expressions:
  `df.loc[(df["drug_X"]>132) & (df["group"]!="C")]`
  - Other logic operators:
    `&` `|` `~` `^` etc.

	group	drug_X	drug_Y
1	B	140	150

- Use query expression engine:
  `df.query("drug_X > 132 and group != 'C'")`

USEFUL METHODS FOR SELECTION

- Return Booleans for later indexing:
  - `.isin(seq)` – is value is one of `seq`?
  - `.isna() / .notna()` – find missing values.
  - `.duplicated()` – value is duplicated.
  - `.between(…)` – values between some range.
- Others
  - `.idxmin() / .idxmin()` – row index (not position!) of minimum / maximum value
  - …

SHORTCUTS

- Since filtering is just indexing, combine:
  `df.loc[(df['drug_X']>132), ['group','drug_Y']]`

- For filtering can use [ ] as shortcut
  `df[df["drug_X"]>132]`

	group	drug_Y
1	B	150
2	C	135

- Careful! **[ ] is not the same as `.loc`**:
  `df[(df['drug_X']>132), ['group','drug_Y']]`
  → results in error
  - Pandas has to guess what you mean

PUTTING IT TOGETHER

- We want the mean blood pressure for groups A and C.

	group	drug_X	drug_Y
0	A	130	NaN
1	B	140	150
2	C	135	135

```
df[df['group'].isin(['A','C'])][['drug_X','drug_Y']].mean()
```

Select rows from group A and C

Select columns

Compute the mean

drug_X	132.5
drug_Y	135.0

USEFUL METHODS

- `head() / tail()` – get first/last few rows
- `diff()` – compute differences between rows
- `rank()`
- `nsmallest() / nlargest()`
- `str.method()` … – call string methods!
- Many more…
- pandas user guide is your friend
  https://pandas.pydata.org/pandas-docs/stable/user_guide/index.html

DO TASKS IN BLOCK 2

Photo by Larisa Koshkina

BASIC VISUALIZATION

- Let’s say we have some data

	group	drug_X	drug_Y
0	A	130	NaN
1	B	140	150
2	C	135	135
3	A	132	142
4	A	126	144
5	B	143	151
6	C	133	132

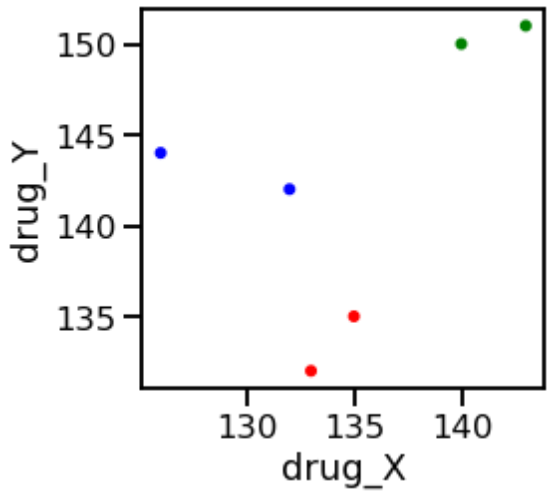
WHY USE VISUALIZATION?

- Extract meaning from lots of data.
  - Huge tables of data are hard to read.
- Sometimes to make a point, tell a story.
- In science, clarity is key.
  - What does the data show?
  - What do I want the reader to see?
  - The point is not “look at this cool figure”
- There are **books** on this.

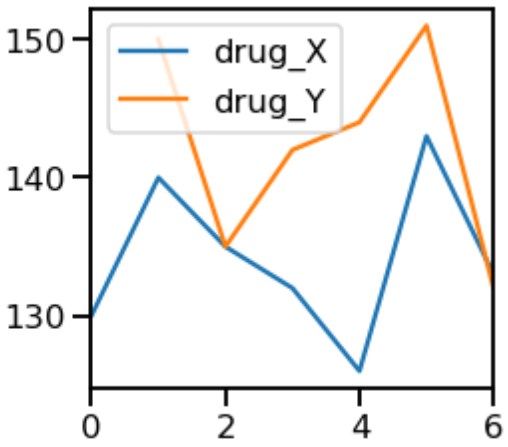
TYPES OF FIGURES

Show relationship between variables X and Y:

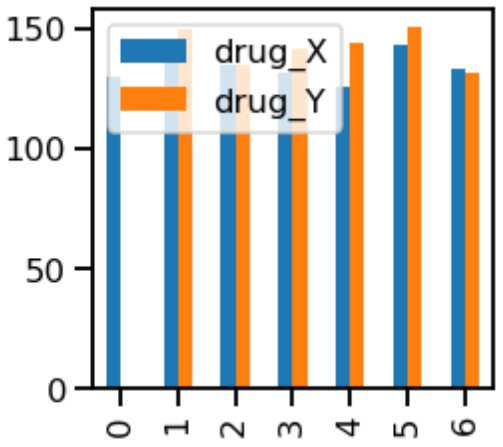
scatter plot



line plot



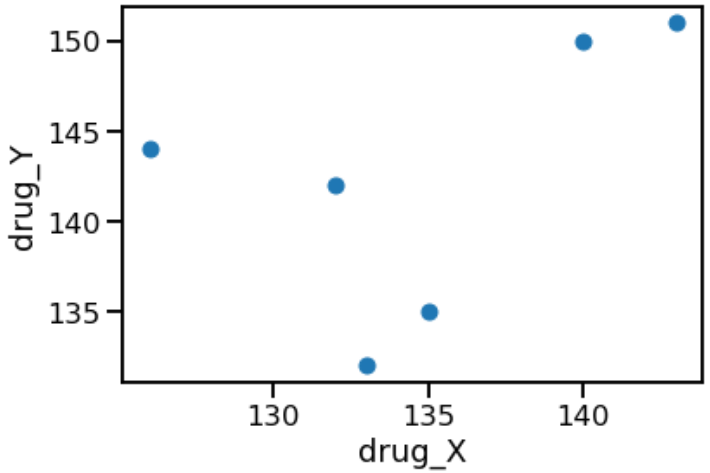
bar plot



PLOTTING IN MATPLOTLIB

- matplotlib is a lower level plotting library
  `import matplotlib.pyplot as plt`
- It doesn't really know about DataFrames
- Just tell it what to do.

```
plt.plot(df['drug_X'],
         df['drug_Y'],
         'o')
plt.xlabel('drug_X')
plt.ylabel('drug_Y')
```



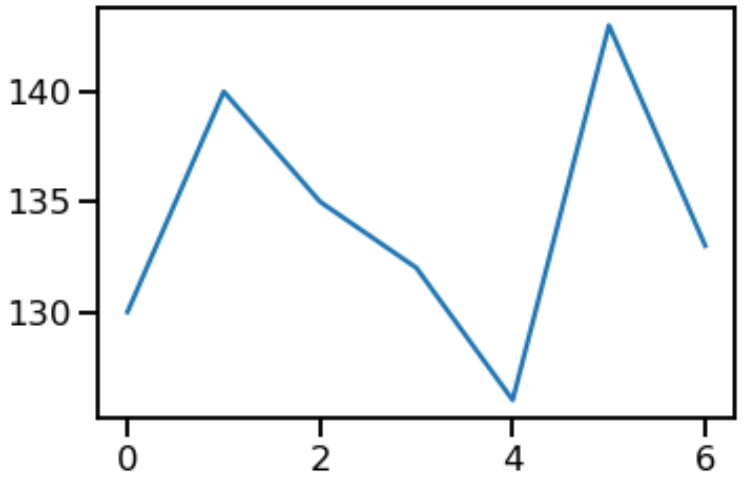
MATPLOTLIB PLOT

- `plt.plot` is the workhorse of matplotlib
- You then add things like labels, legends, and adjust parameters.
- There are many was to call it.
- We will only cover a couple.

MATPLOTLIB LINE PLOT

- Want a line plot? Set format to have lines

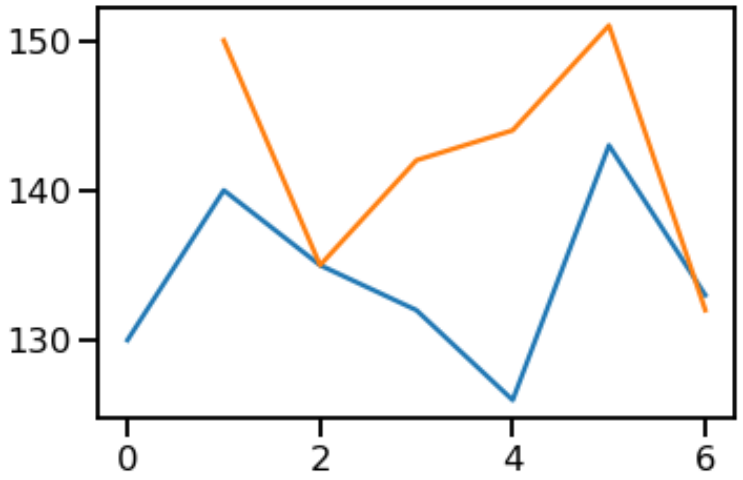
```
plt.plot(df.index, df["drug_X"], "-")
```



MATPLOTLIB LINE PLOT

- Want a line plot? Set format to have lines
  `plt.plot(df.index, df["drug_Y"], "-")`

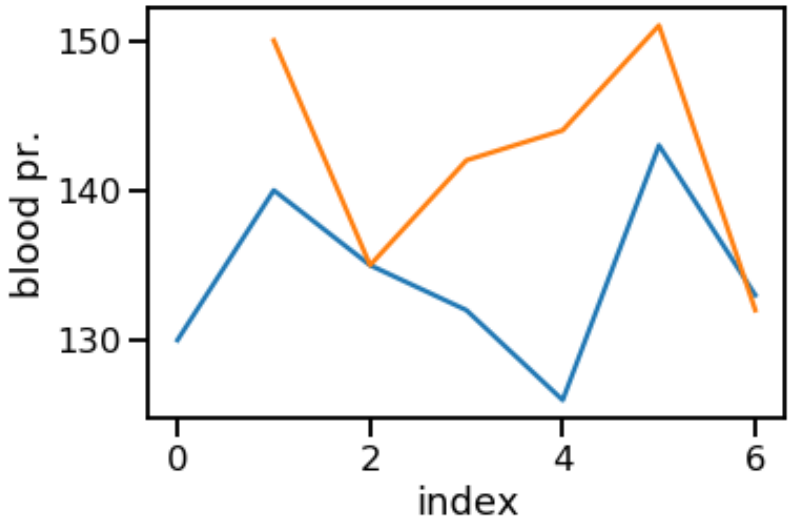
matplotlib will create and adjust axes for you!



MATPLOTLIB LINE PLOT

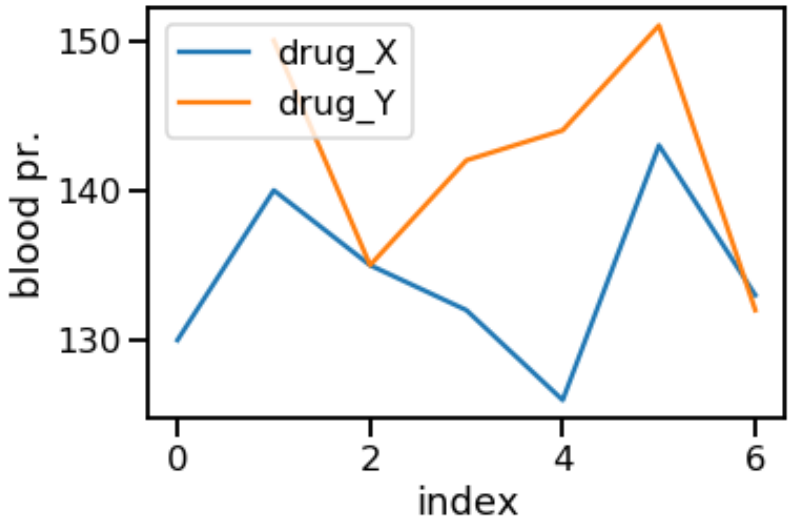
- axis labels:

```
plt.xlabel('index')
plt.xlabel('blood pr.')
```



MATPLOTLIB LINE PLOT

- Add a legend:
  `plt.legend(['drug_X', 'drug_Y'])`

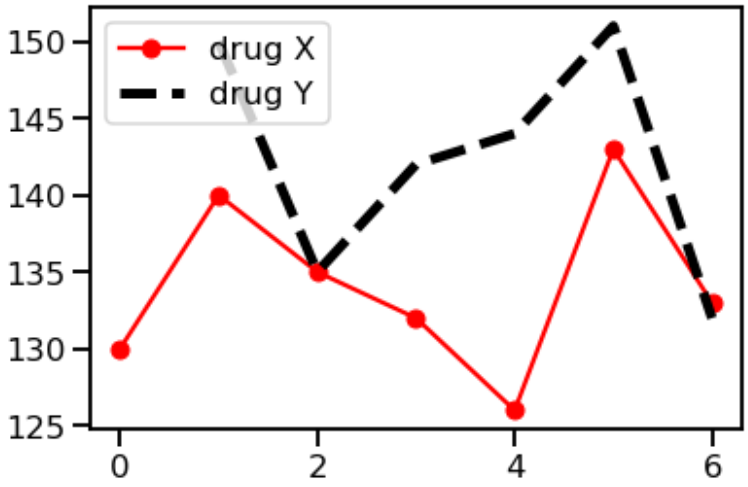


CONTROL LINE STYLE, COLOR, FORMAT

- matplotlib is highly configurable:

```
plt.plot(df.index, df['drug_X'], 'o-', color='red',
         label='drug X')
plt.plot(df.index, df['drug_Y'], 'k--', linewidth=5,
         label='drug Y')
plt.legend(loc='upper left')
```

- Control `alpha`, `markersize`, and more.



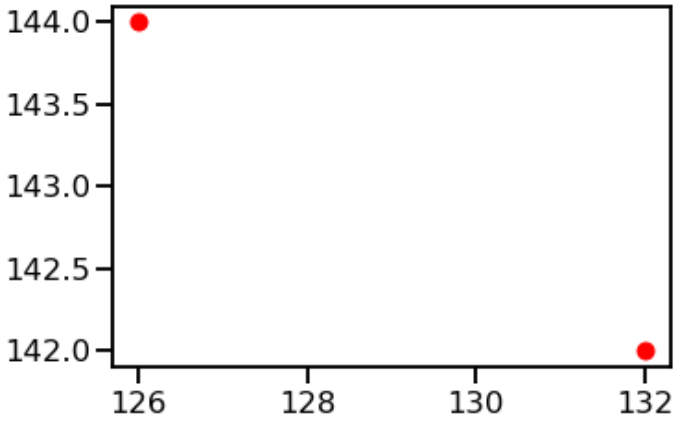
MATPLOTLIB SCATTER PLOT

- How about a scatterplot, with colors?
- For every group we want a different color and marker.

MATPLOTLIB SCATTER PLOT

- Get and draw group A in red with circle marker:

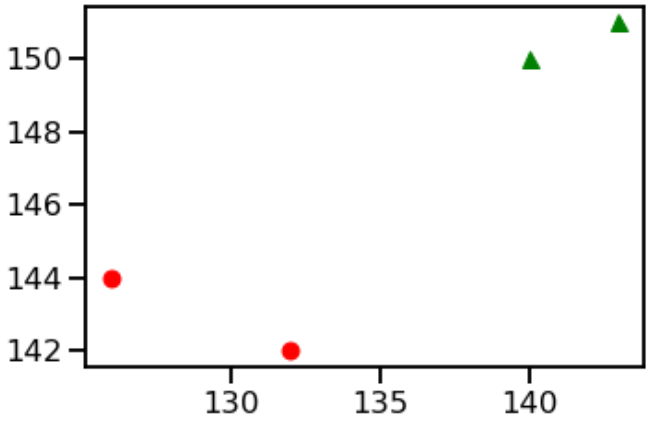
```
a = df[df['group'] == 'A']
plt.plot(a['drug_X'], a['drug_Y'], 'ro')
```



MATPLOTLIB SCATTER PLOT

- Add group B in green with triangles:

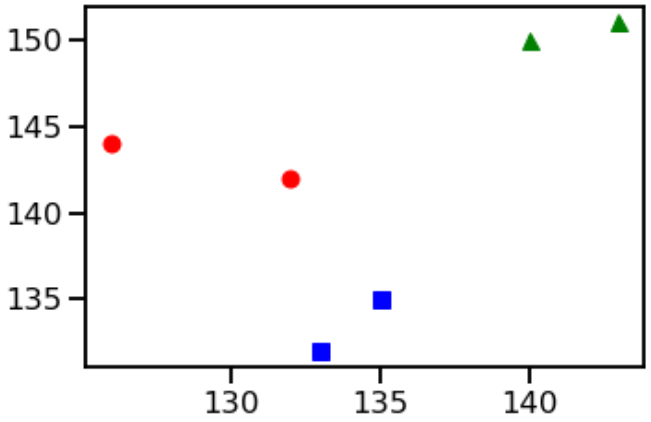
```
b = df[df['group'] == 'B']
plt.plot(b['drug_X'], b['drug_Y'], 'g^')
```



MATPLOTLIB SCATTER PLOT

- Add group C in blue with squares:

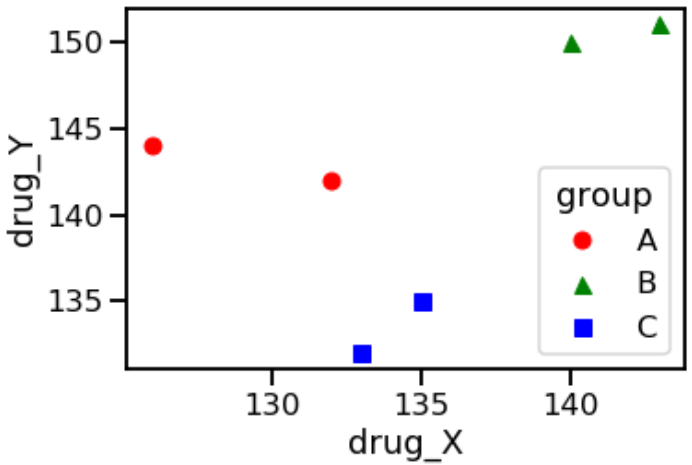
```
c = df[df['group'] == 'C']
plt.plot(c['drug_X'], c['drug_Y'], 'bs')
```



MATPLOTLIB SCATTER PLOT

- Add labels and legend:

```
plt.legend(['A', 'B', 'C'],
           title='group')
plt.xlabel('drug_X')
plt.ylabel('drug_Y')
```



MATPLOTLIB CUSTOMIZATION

- Get or set X and Y axis edges (limits):
  `plt.xlim(xmin, xmax), plt.ylim(…)`
- Control ticks and tickmarks
  `plt.xticks(…), plt.tickparams(…)`
- Annotate figure with text and arrows
  `plt.annotate(…)`

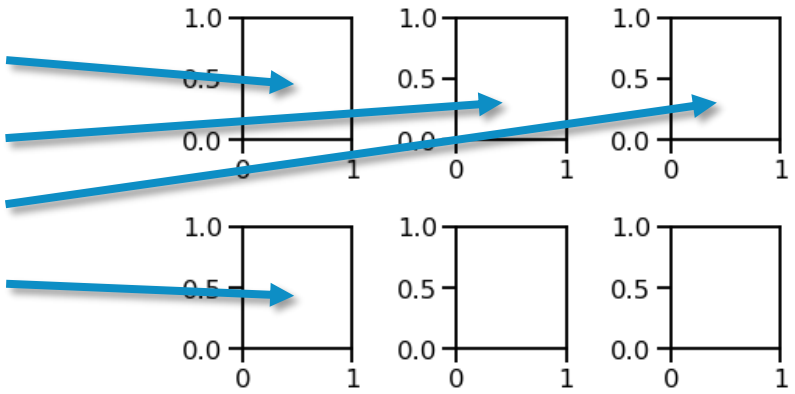
SUBPLOTS

- A matplotlib **`axes`** object is one plot
  - Axes, data visualization, legend, labels, ticks…
- A **`figure`** contains one or more **`axes`**
- Create multiple figures:

```
fig, axs = plt.subplots(2, 3, …)
```

Or…

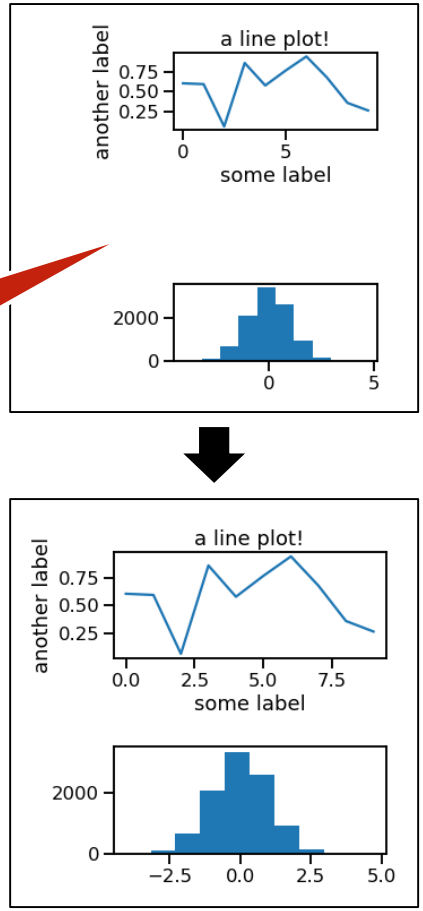
```
plt.subplot(2,3,1)
plt.subplot(2,3,2)
plt.subplot(2,3,3)
plt.subplot(2,3,4)
```



FIGURES AND SUBPLOTS

- Control layout:
  `plt.subplots_adjust(…)`
- Or…
  `plt.tight_layout()`

wasted space

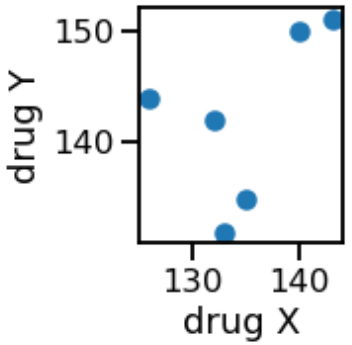


- Save the current figure
  `plt.savefig("figure1.png")`
  - Can also save .pdf, .eps, etc.

OBJECT ORIENTED API

- You can work with the “MATLAB-like” API:
  - `plt.command()` works on current figure/axes.
- It is a thin shell over an object oriented API:

```
fig = plt.figure(figsize=(3,3))
ax = fig.add_subplot(1,1,1)
ax.scatter(df['drug_X'],
           df['drug_Y'])
ax.set_xlabel('drug X')
ax.set_ylabel('drug Y')
```



LOW VS HIGH LEVEL

- Lots of work.
- matplotlib does not understand the data…
- But pandas does!
- It can do this for you.

```
a = df[df['group'] == 'A']
plt.plot(a['drug_X'],
         a['drug_Y'], 'ro')
b = df[df['group'] == 'B']
plt.plot(b['drug_X'],
         b['drug_Y'], 'go')
c = df[df['group'] == 'C']
plt.plot(c['drug_X'],
         c['drug_Y'], 'bo')
plt.xlabel('drug_X')
plt.ylabel('drug_Y')
plt.legend(['A', 'B', 'C'],
           title='group')
```

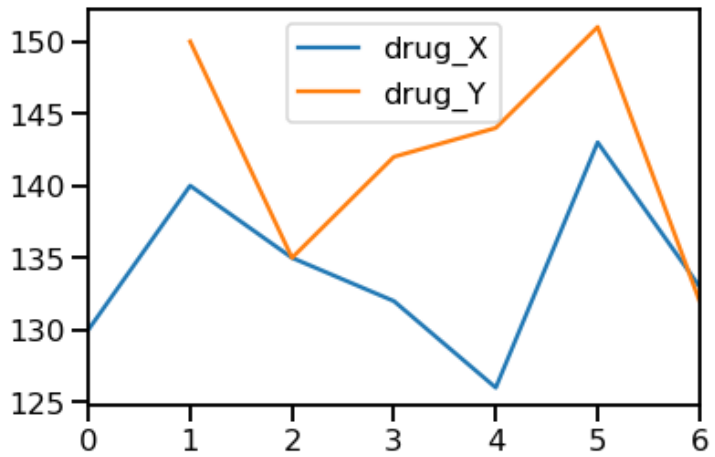
LINE CHART IN PANDAS

matplotlib

```
plt.plot(df.index, df['drug_X'], '-')
plt.plot(df.index, df['drug_Y'], '-')
plt.legend(['drug_X', 'drug_Y'])
```

pandas

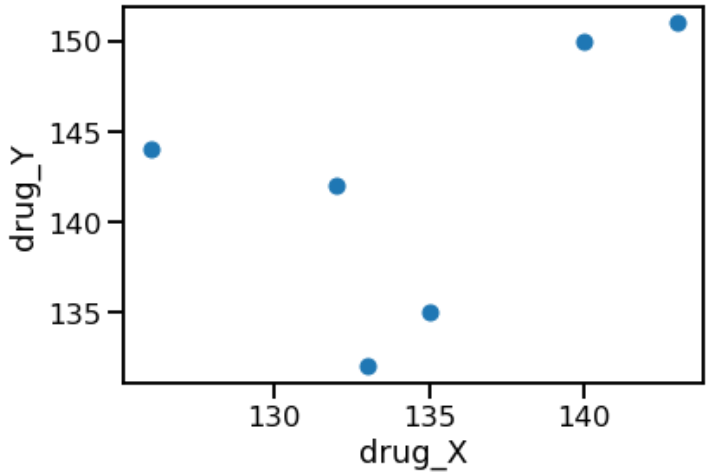
```
df.plot.line()
```



SCATTER IN PANDAS

matplotlib

```
plt.plot(df['drug_X'], df['drug_Y'], 'o')
plt.xlabel('drug_X')
plt.ylabel('drug_Y')
```



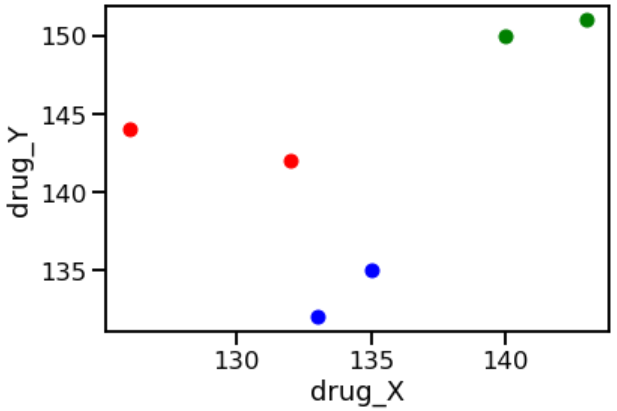
pandas

```
df.plot.scatter('drug_X', 'drug_Y')
```

SCATTER WITH COLORS

matplotlib

```
a = df[df['group'] == 'A']
plt.plot(a['drug_X'], a['drug_Y'], 'ro')
b = df[df['group'] == 'B']
plt.plot(b['drug_X'], b['drug_Y'], 'go')
c = df[df['group'] == 'C']
plt.plot(c['drug_X'], c['drug_Y'], 'bo')
plt.xlabel('drug_X')
plt.ylabel('drug_Y')
```



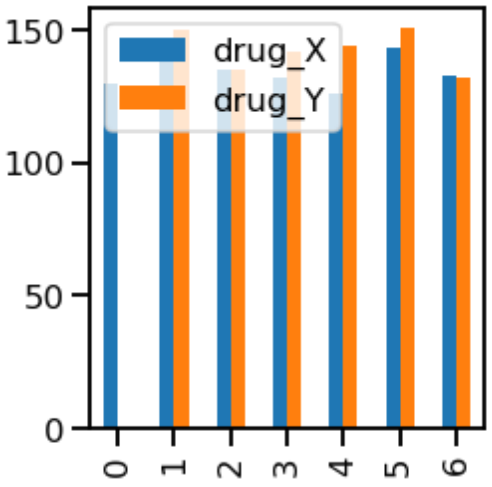
pandas

```
colors = df['group'].map(dict(A='r', B='g', C='b'))
df.plot.scatter('drug_X', 'drug_Y', c=colors)
```

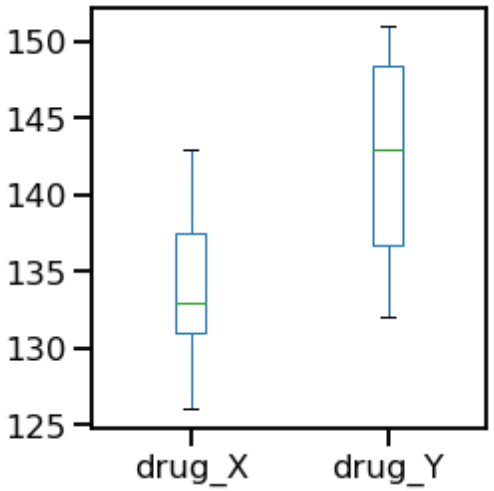
MORE PLOTTING

- Additional pandas plotting functions
  - Built on top of matplotlib !

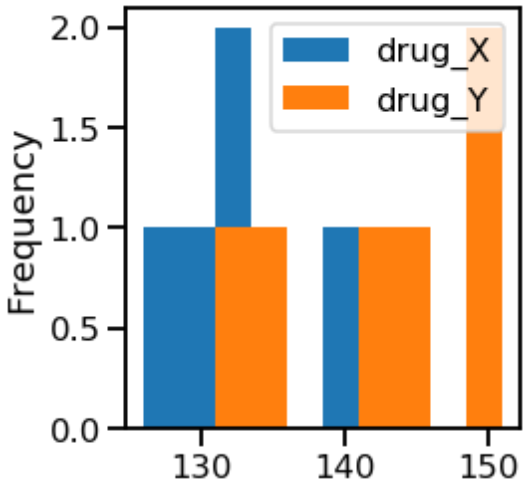
`df.plot.bar()`



`df.plot.box()`



`df.plot.hist()`



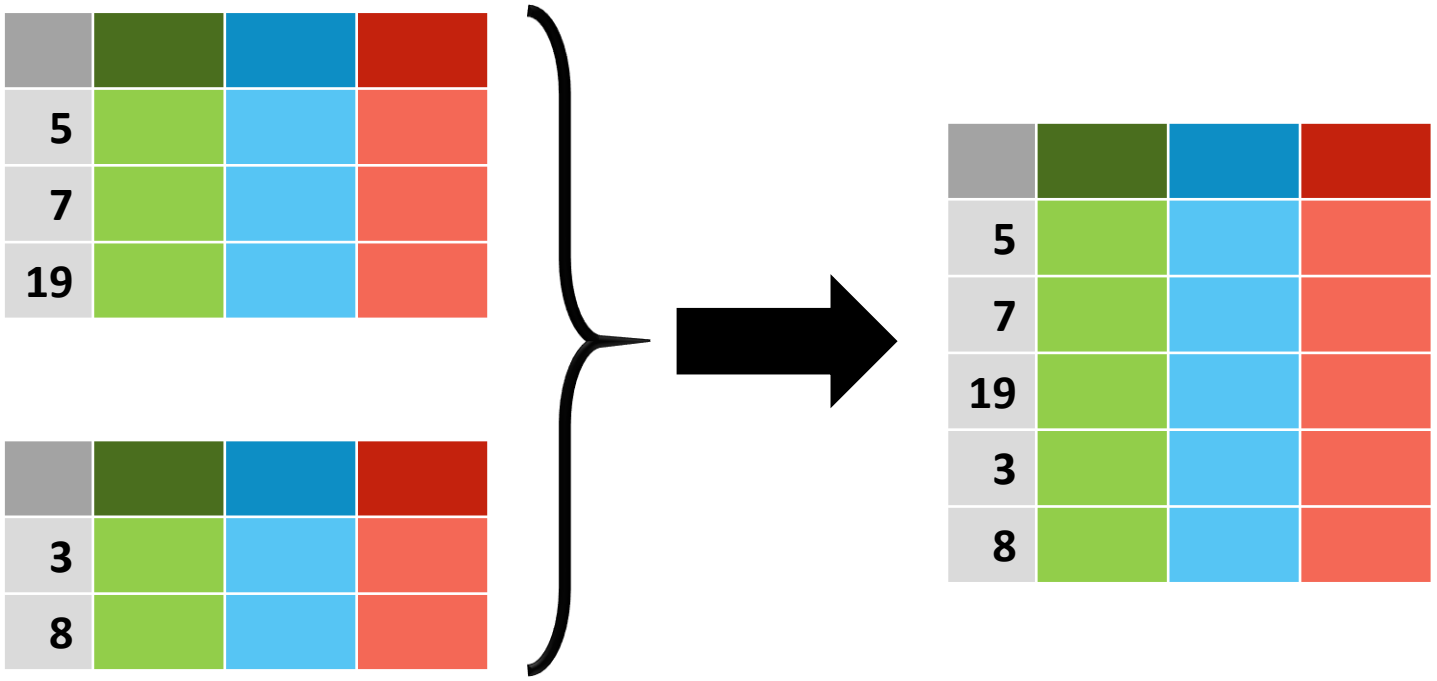
DO TASKS IN BLOCK 3

Photo by George Hodan

RESHAPING DATA

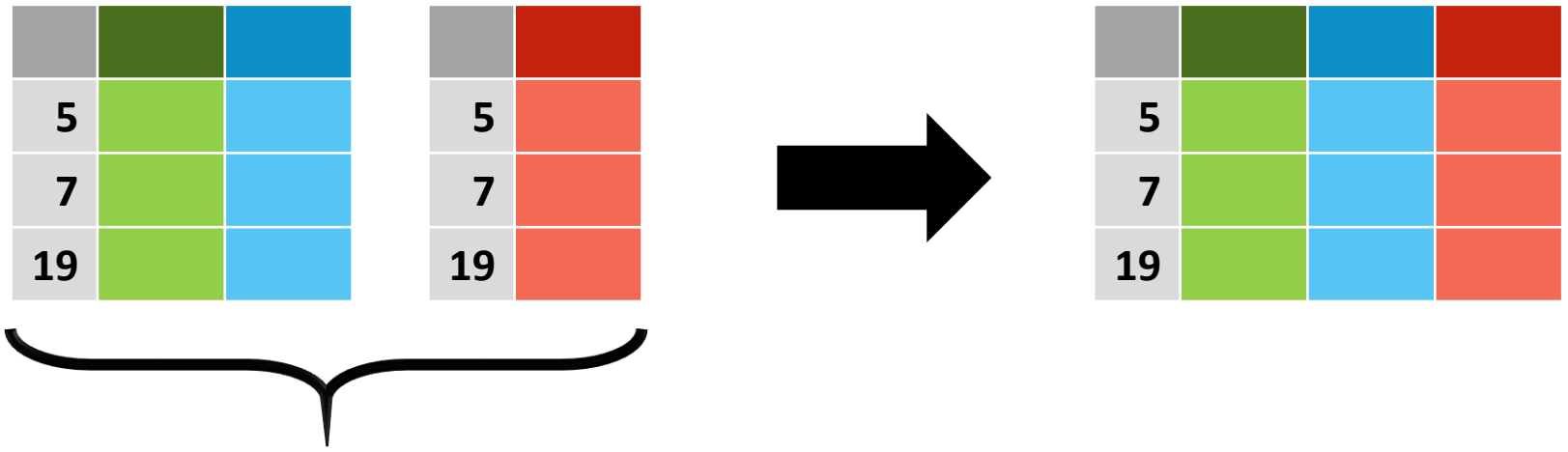
CONCATENATING

- Concatenate rows: `pd.concat([df1,df2])`



CONCATENATING

- Concatenate columns:
  `pd.concat([df1,df2], axis=1)`



COMBINING DATA

- Say we have two DataFrames.

df1

X	Y
A	1
B	2
C	3

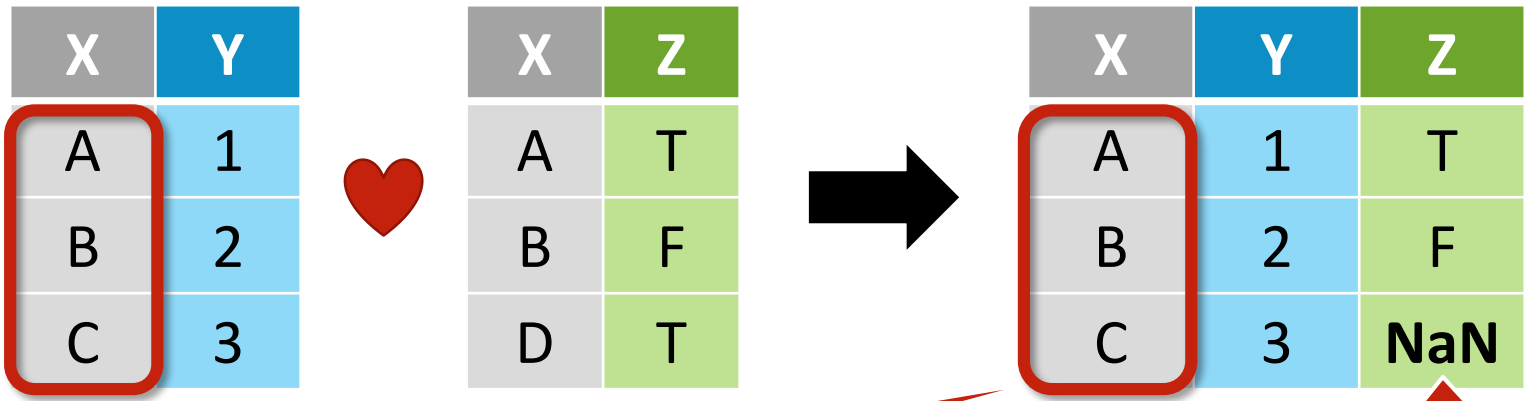
X	Z
A	T
B	F
D	T

df2

- We want to merge them based on common values and/or columns

LEFT JOIN

- Join matching rows that appear on **df1**:
  `pd.merge(df1, df2, on="X", how="left")`

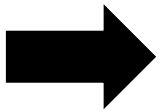


RIGHT JOIN

- Join matching rows that appear on **df2**:
  `pd.merge(df1, df2, on="X", how="right")`

X	Y
A	1
B	2
C	3

X	Z
A	T
B	F
D	T

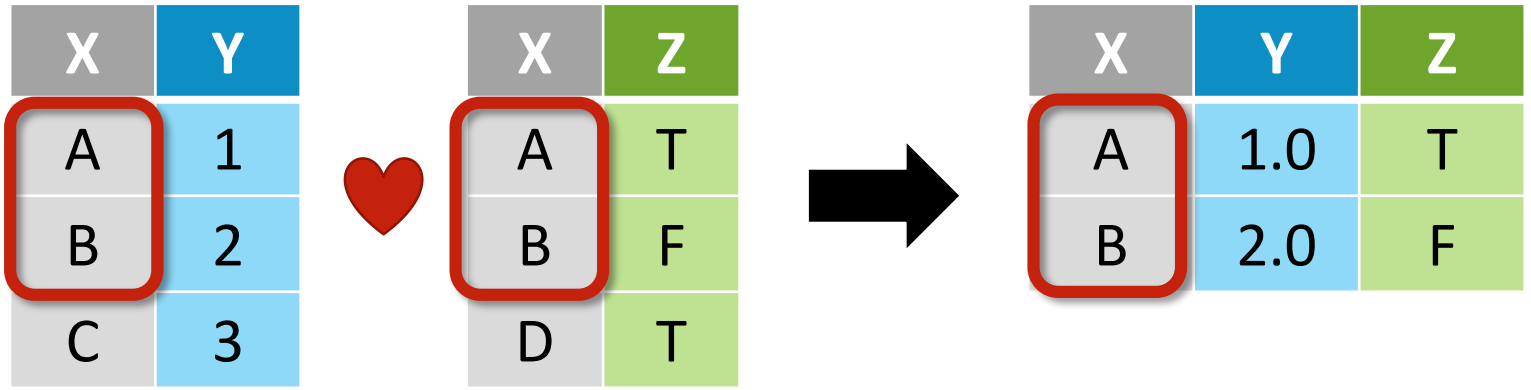


X	Y	Z
A	1.0	T
B	2.0	F
D	**NaN**	T

1. pandas coverted column Y to float to allow NaNs.
2. df1 has no matching row for X == “D”.

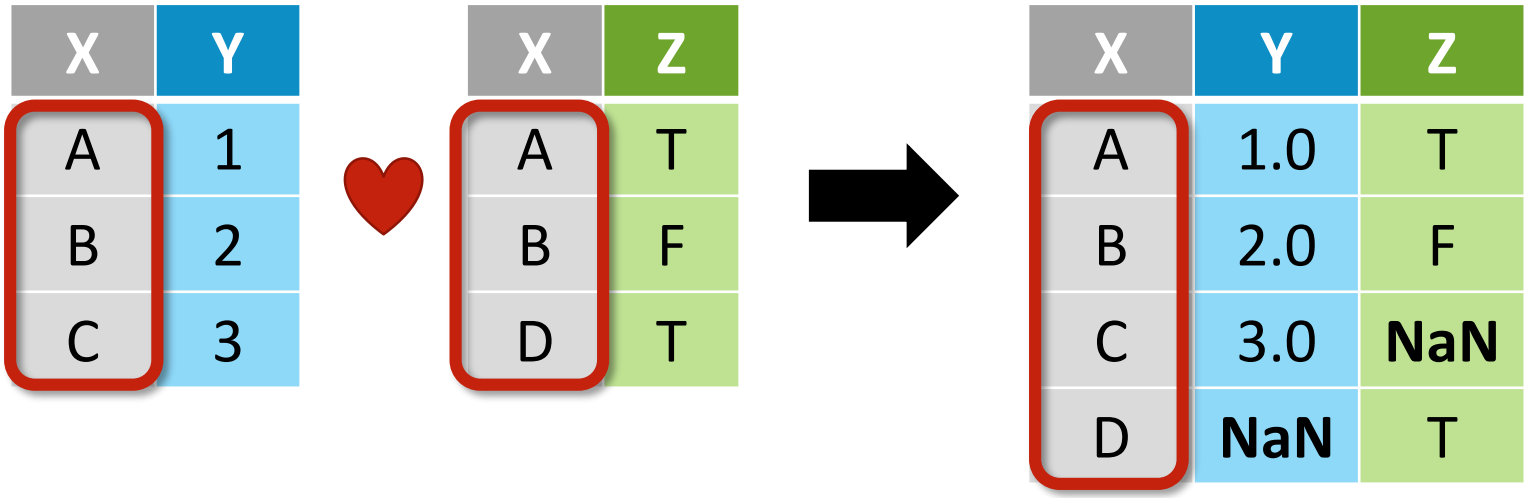
INNER JOIN

- Join matching rows that appear on **both**:
  `pd.merge(df1, df2, on="X", how="inner")`



OUTER JOIN

- Join matching rows that appear on **either**:
  `pd.merge(df1, df2, on="X", how="outer")`



X	Y
A	1
B	2
C	3

X	Z
A	T
B	F
D	T

X	Y	Z
A	1.0	T
B	2.0	F
C	3.0	**NaN**
D	**NaN**	T

MORE COMBINING

- Filtering “join”, for example:
  all rows in df1 that do **not** have a match in df2
  `df1[~df1['X'].isin(df2['X'])]`

X	Y
C	3

- Merge on several columns together
  `pd.merge(df1, df2, on=[…], …)`
- Merge on all columns:
  `pd.merge(df1, df2, …)`
- Merge on index

SET OPERATIONS

- Set operation on whole rows by merging on all columns

A

X	Y
A	1
B	2

B

X	Y
B	2
C	3

Intersection:

```
pd.merge(A, B)
```

X	Y
B	2

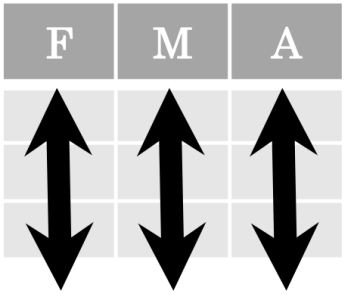
Union:

```
pd.merge(A, B, how="outer")
```

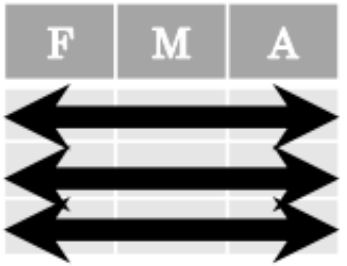
X	Y
A	1
B	2
C	3

TIDY DATA

Column for each **variable**



Row for each **observation**



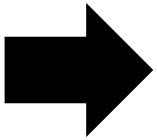
- Complements pandas columnar storage.
- Preserves observation when adding columns.
- Helps with visualization.

M * A

TIDY DATA

wide format

	group	drug_X	drug_Y
0	A	130	NaN
1	B	135	150
2	C	140	135



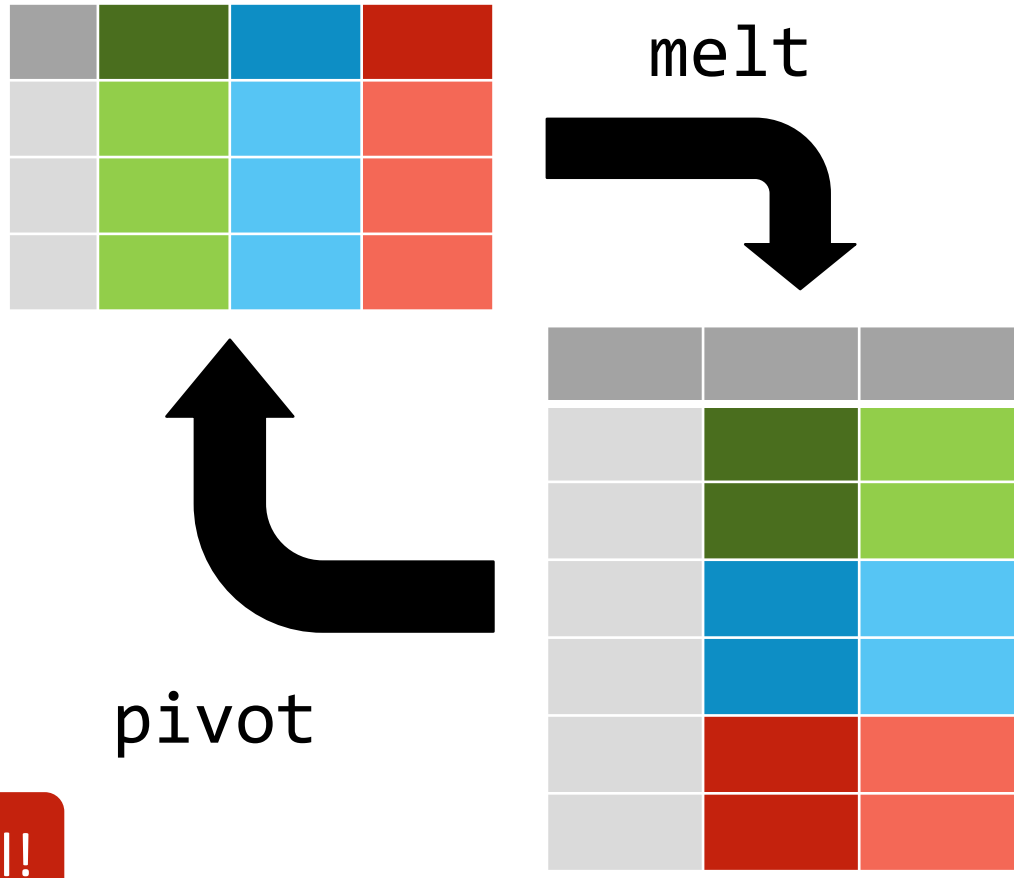
obs.	group	drug	blood pressure
0	A	X	130.0
1	B	X	135.0
2	C	X	140.0
3	A	Y	NaN
4	B	Y	150.0
5	C	Y	135.0

long (or tidy) format

RESHAPING DATA

- Gather data to columns (tidy) `df.melt(…)`
- Spread rows into columns `df.pivot(…)`

Pivoting is very powerful!



CHAINING

- Many DataFrame methods return the resulting data frame.
- Can therefore chain methods together!

```
df.set_index('group')
  .melt()
  .query('value < 150')
  .mean()
```

GROUPING

- We want a table of mean blood pressure for every type of drug.
- Start with tidy data
- We need to **group** by value of “drug” columns

obs.	group	drug	blood pressure
0	A	X	130.0
1	B	X	135.0
2	C	X	140.0
3	A	Y	NaN
4	B	Y	150.0
5	C	Y	135.0

GROUPING

Can group by multiple columns:
`["drug", "v1"]`

	v1	drug	v2
1			
5			
7			
3			
2			
9			
4			

- Group rows by column value:
  `grp = long.groupby("drug")`
- Can now iterate over groups…

	v1	drug	v2
1			
7			

	v1	drug	v2
3			
2			

	v1	drug	v2
5			
9			
4			

GROUPING

	v1	drug	v2
1			
5			
7			
3			
2			
9			
4			

- Can simply aggregate: `long.groupby("drug").mean()`

- Can use any aggregation function: `median`, `agg`, etc.

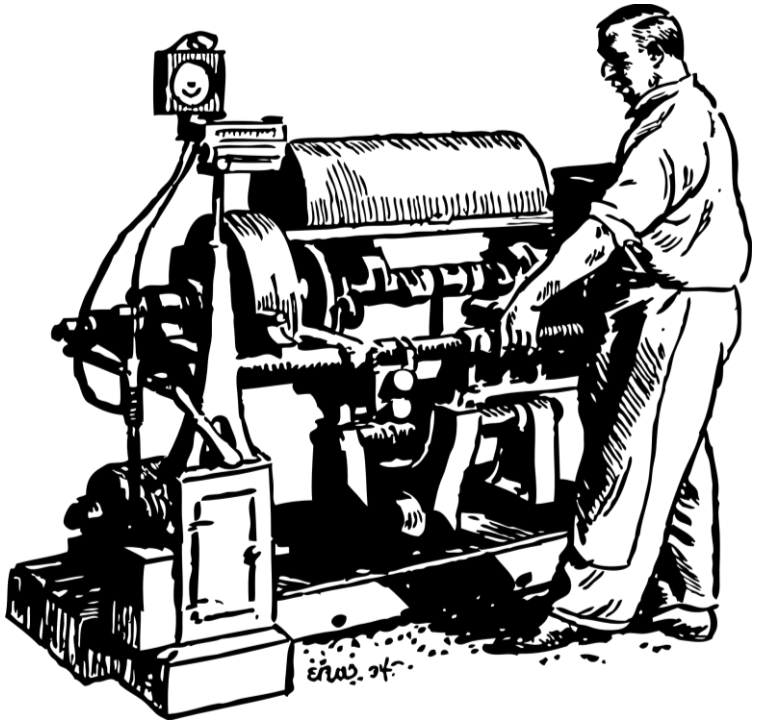
GROUPING

	v1	drug	v2
1			
5			
7			
3			
2			
9			
4			

- Can also select a column first:

```
long.groupby("drug")['v1']
    .mean()
```

drug	v1



DO TASKS IN BLOCK 4

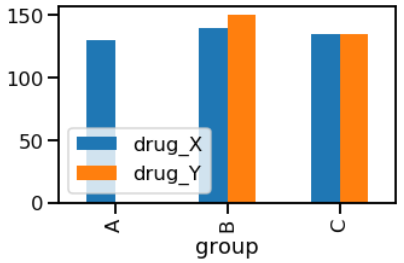
ADVANCED PLOTTING

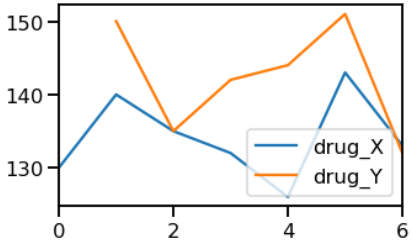
TYPES OF VARIABLES

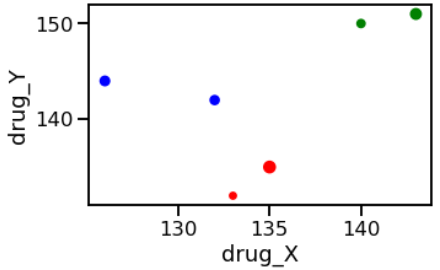
- **Numeric**: a number
  - Can be integers or continuous
  - Examples: weight, year, speed, memory, time
- **Categorical**:
  - Possible values with no ordering.
  - Can also be numeric with few values.
  - Examples: day of week, type of drug, colors, strings
- There are more…

WHEN TO USE WHICH?

- Bar chart
  - X is categorical or has few values.
  - Add color for categorical variable.
- Line plot
  - X is continuous, has many values, is time.
  - To emphasize Y as function of X.
  - Additional lines for for categorical variable.
- Scatter plot
  - Relationship between X and Y.
  - Add color for categorical variable.
  - Change point size for numeric variable.

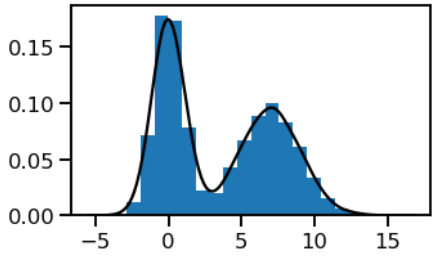


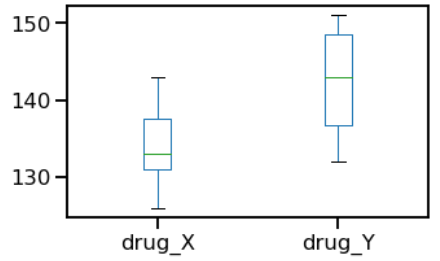


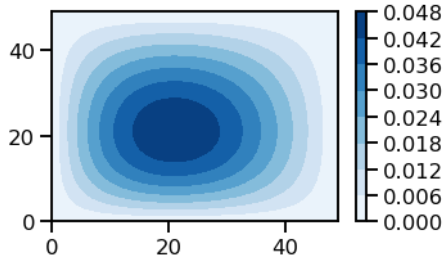


WHEN TO USE WHICH?

- Histogram
  - Distribution of numeric values.
  - Can complement with density plot.
  - Or colors for categorical variable.
- Box plot
  - Compare distributions.
  - Usually between different values of categorical variable.
- Contour
  - Third variable as function of X and Y.







SEABORN

- Say we have some tidy data.
- From experiments or whatever.
- Let's explore it.

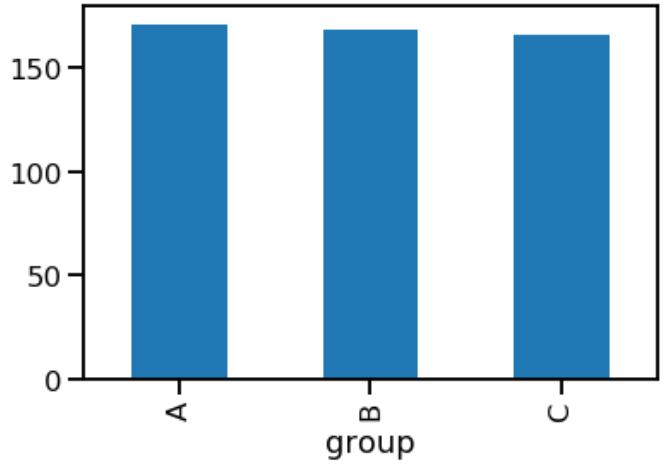
	group	sex	age	height	drug	bl. pressure
0	A	M	31	180	X	130.0
1	B	F	40	156	X	140.0
2	C	M	82	173	X	135.0
3	A	F	50	164	X	132.0
4	A	F	55	170	X	126.0
5	B	M	70	182	X	143.0
6	C	F	28	159	X	133.0
7	A	M	31	180	Y	NaN
8	B	F	40	156	Y	150.0
9	C	M	82	173	Y	135.0
10	A	F	50	164	Y	142.0
11	A	F	55	170	Y	144.0
12	B	M	70	182	Y	151.0
13	C	F	28	159	Y	132.0

FACETS OF DATA

- Is there a connection between **group** and **height**?

```
grp = df.groupby('group')
grp['height'].mean().plot.bar()
```

- Maybe a small one?
- What if we split by **gender?**
- Stop and think!



PLOTTING AGGREGATES

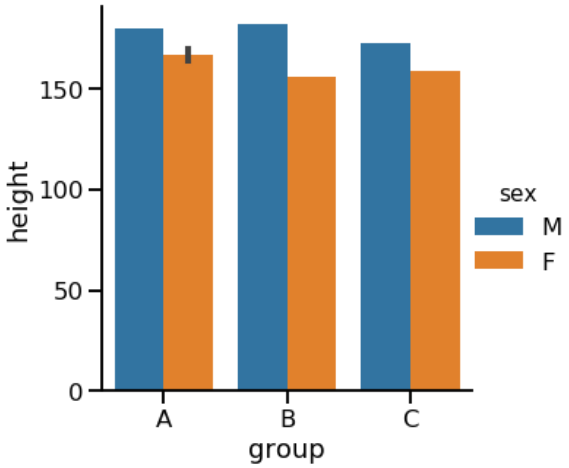
- We keep wanting to
  - Group or aggregate by some columns
  - Plot some variable on X or Y
  - Show others by color, etc.
- This will get unwieldy, quick.
- What we want is a way to just specify columns and have someone else do the grouping and plotting.

ENTER SEABORN

Seaborn helps explore **facets** of your data:

1. Put your data in tidy format.
2. Specify plot type, columns, axes.
3. Seaborn does the rest!

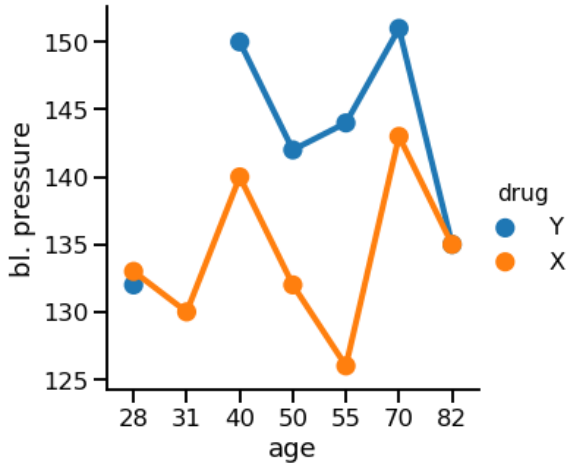
```
sns.catplot(data=df,
            kind='bar',
            x='group',
            y='height',
            hue='sex')
```



ENTER SEABORN

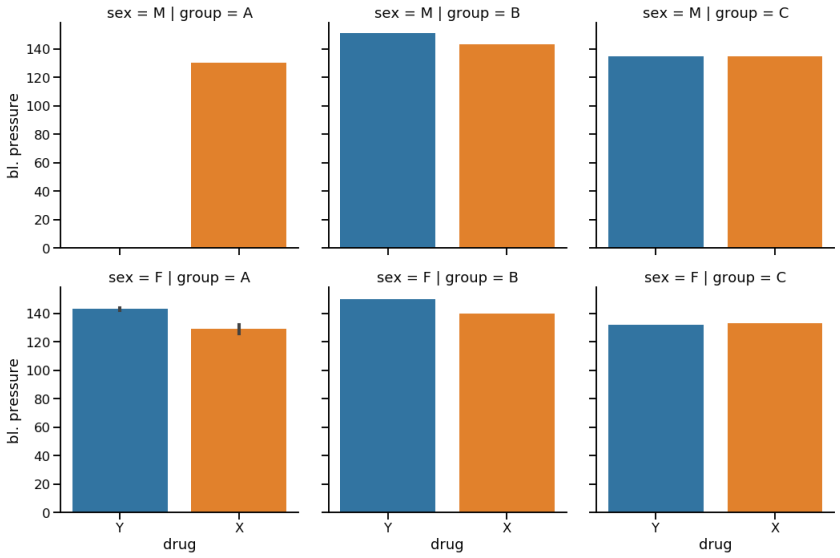
Does drug effect change by age?

```
sns.catplot(data=df, kind='point',
            x='age', y='bl. pressure',
            hue='drug')
```



FACETS AS SUBPLOTS

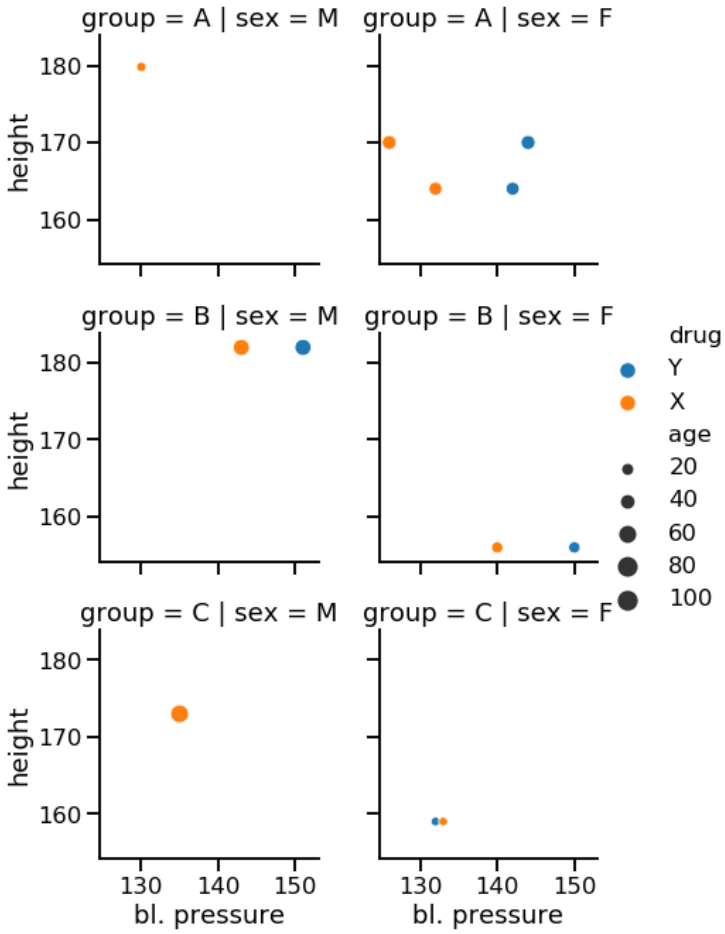
- Use subplots to explore additional facets for categorical variables




```
sns.catplot(data=df, kind='bar',
                x='drug', y='bl. pressure',
                row='sex', col='group')
```

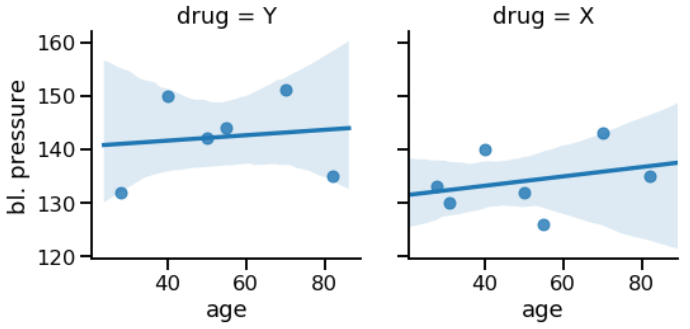
GO NUTS!

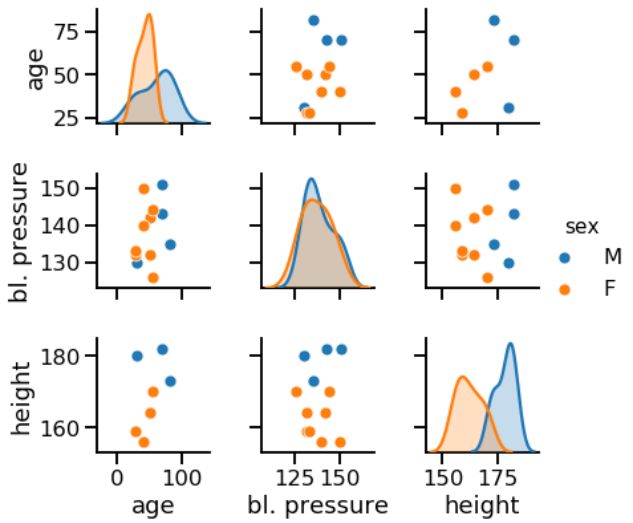
- blood pressure vs height
  - Scatter plot (`relplot` in seaborn)
  - **X** = blood pressure
  - **Y** = height
  - **Color** = drug
  - **Size** = age
  - **Subplot rows** = group
  - **Subplot columns** = sex



HIGHER LEVEL PLOTS

- Seaborn has lots of plotting functions.
- Common interface!
- Linear regression `sns.lmplot()`
- Matrix of variable pairs `sns.pairplot()`

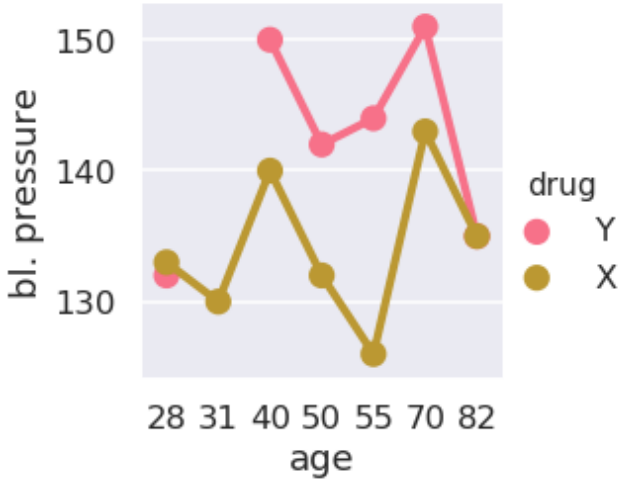




PLOT CUSTOMIZATION

- Use Seaborn highlevel commands to set plot style, colors, and scale:

```
sns.set(style='darkgrid',
        palette='husl')
```



- Can use matplotlib commands: `plt.ylables`, `plt.xlim`, etc.

WORKFLOW

1. Prepare tidy pandas DataFrame.
2. **Think**: what are you trying to say / show?
3. Control figure aesthetics (style, colors, size).
4. Use Seaborn to plot.
5. Customize plot using Seaborn or matplotlib.
6. Save plot.

Do not skip this step!

DIFFERENT DATA

- Seaborn comes with example datasets.
- Lets try the tips dataset

```
tips = sns.load_dataset('tips')
```

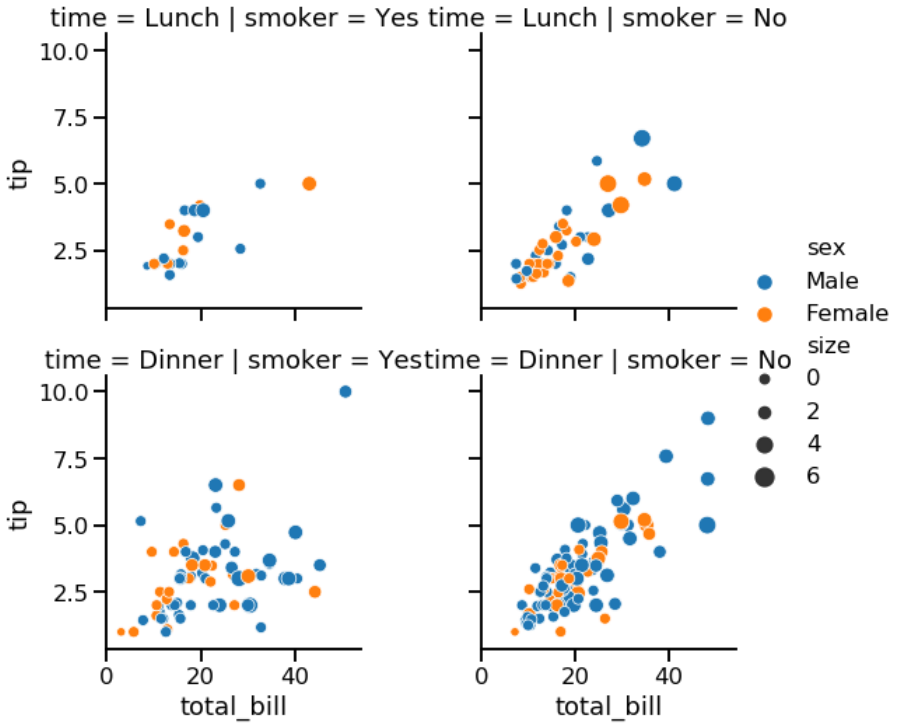
	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
…	…	…	…	…	…	…	…
223	15.98	3.00	Female	No	Fri	Lunch	2
…	…	…	…	…	…	…	…

TIPS DATA

Visualize the data:

```
sns.relplot(data=tips,
      x='total_bill',
      y='tip',
      hue='sex',
      size='size',
      row='time',
      col='smoker')
```

Not very helpful :(

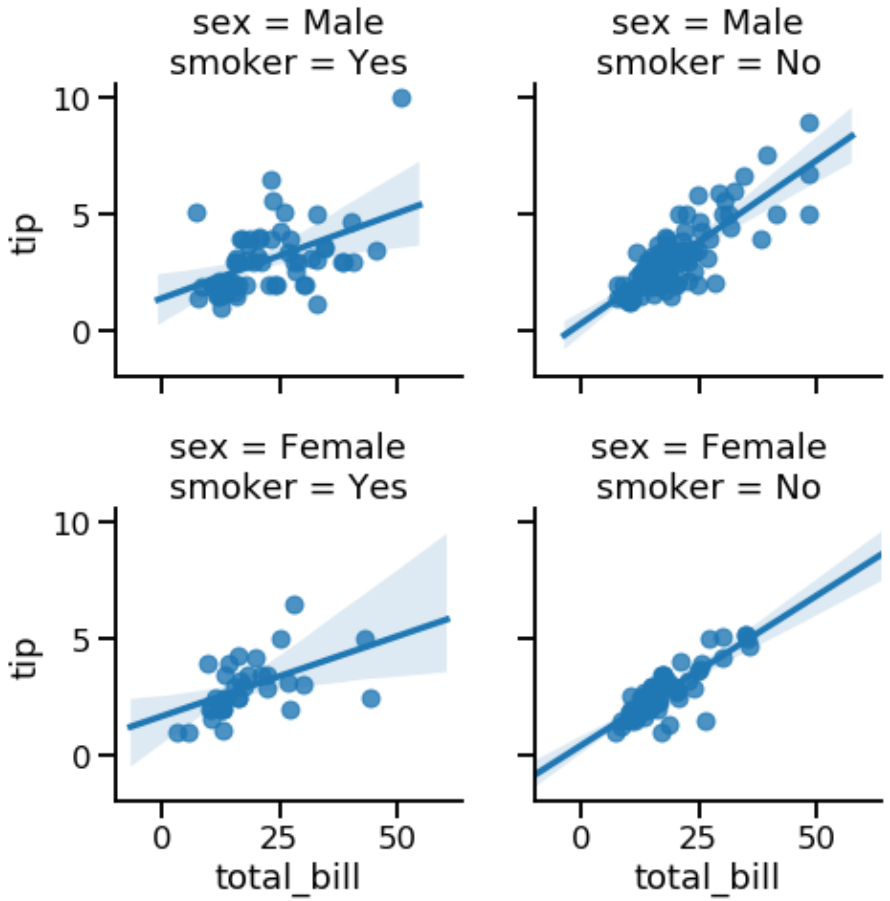


TIPS DATA

What affects the tip?

- Smoking?
- Sex?

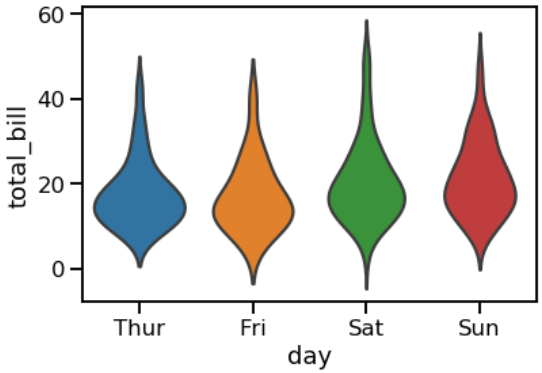
```
sns.lmplot(data=tips,
     x='total_bill‘,
     y='tip',
     row='sex',
     col='smoker')
```



MORE PLOTS

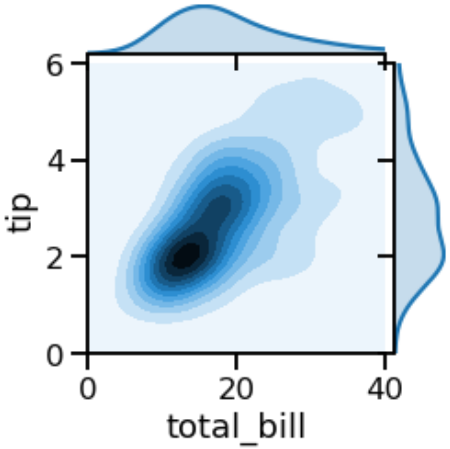
- Violin plot shows distribution

```
sns.violinplot(data=tips,
   x='day', y='total_bill',
   inner=None)
```



- Joint distribution:

```
sns.jointplot(data=tips,
    x='total_bill',
    y='tip', kind='kde',
    xlim=(0,40),
    ylim=(0,6))
```



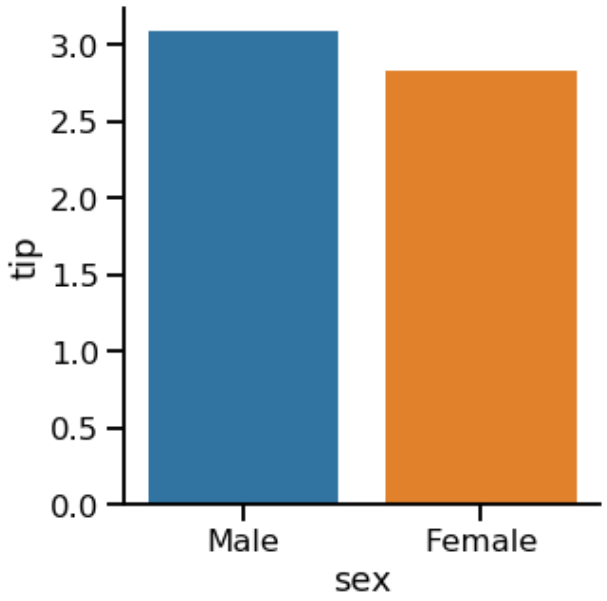
DO TASKS IN BLOCK 5

FINAL COMMENTS

LIES, DAMN LIES, AND VISUALIZATIONS

- Do women tip less than men?

```
sns.catplot(data=tips,
  x='sex',
  y='tip',
  kind='bar', ci=0)
```



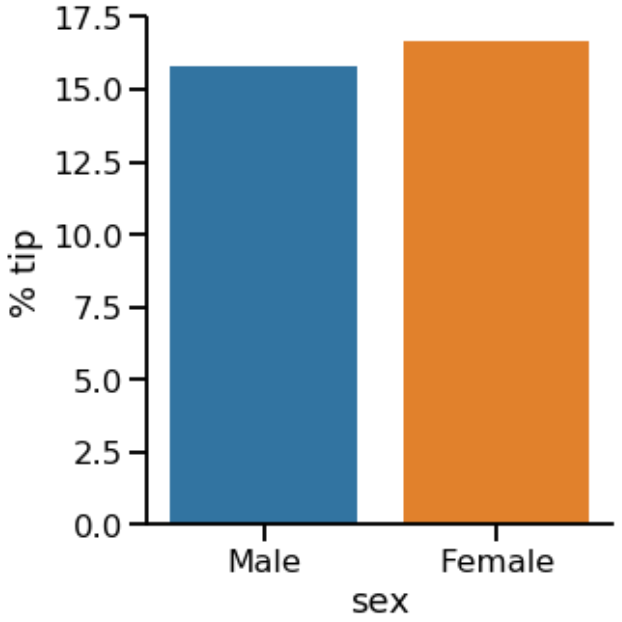
- Yes?
  - 30 cents less on average.

LIES, DAMN LIES, AND VISUALIZATIONS

- Shouldn’t we normalize by the total bill?

```
tips['% tip'] = 100*tips['tip']/tips['total_bill']
sns.catplot(data=tips,
   x='sex',
   y='% tip',
   kind='bar', ci=0)
```

- So women give higher tips?

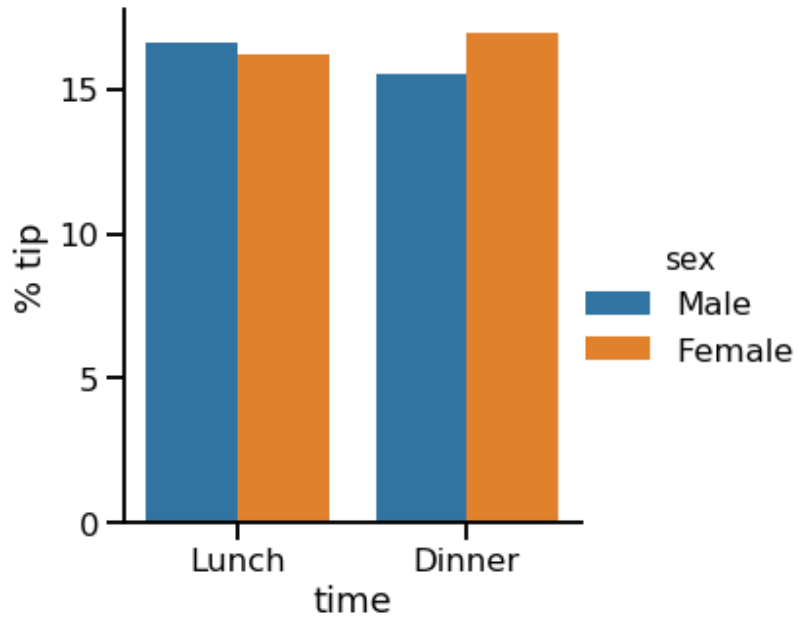


PERHAPS IT DIFFERS BY TIME?

- Maybe it changers over time?

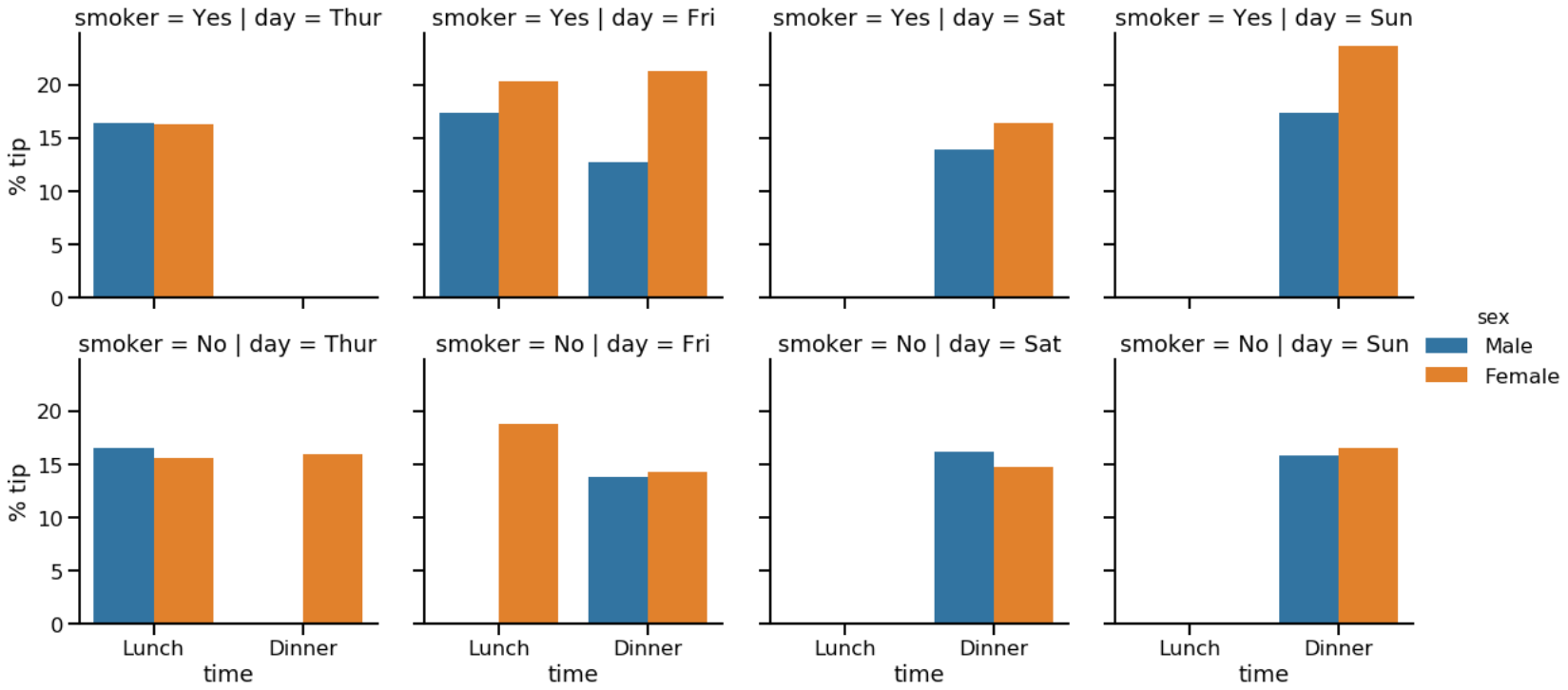
```
sns.catplot(data=tips,
            x='time',
            hue='sex',
            y='% tip',
            kind='bar',
            ci = 0)
```

- Women give more at evening, men give less?



THE LIMITS OF VISUALIZATION

- What about day of week? Smoking?

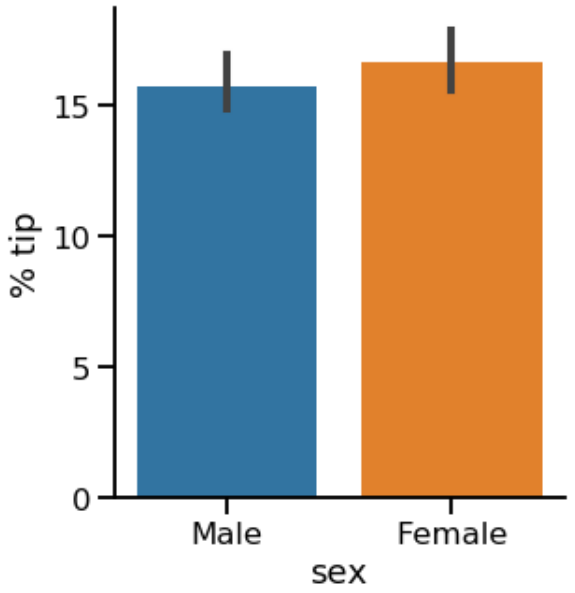



STOP

TAKEAWAY

- **Visualization is not statistical analysis!**

- Also: we should have added **error bars** or **confidence intervals**
  - Seaborn actually does it automatically for us ☺



GENERAL ADVICE FOR VISUALIZATION

1. Think about what you want to show.
2. Say “yes!” to error bars confidence intervals.
3. Bar charts > pie charts (usually).
4. Be thoughtful: **aesthetics**, **colors**, style.
5. Be judicious: don’t overload the reader.
6. Visualization != analysis.
7. Experiment and iterate.

WHAT'S NEXT

- You know **how** to organize, manipulate, and visualize data.
  - Basics of data wrangling in Python.
  - There is a lot we haven't seen
- … but not **what** to do, and **when**.
- Keep learning:
  - Start using this in your work, experiment.
  - Learn basic statistical analysis.

RESOURCES

- Anaconda – Python/R distribution for data science: https://www.anaconda.com/distribution/
- Python for Data Analysis by Wes McKinney
  - Written by creator of pandas
- Pandas user guide: http://pandas.pydata.org/pandas-docs/stable/user_guide/index.html
- Seaborn tutorial: https://seaborn.pydata.org/tutorial.html
- Google and Stack Overflow