THE UNIVERSITY OF CHICAGO

MASTERS IN COMPUTATIONAL SOCIAL SCIENCE
THE UNIVERSITY OF CHICAGO

MACS 30111

M7: Pandas

Announcements

- **Switch to accelerated track: need MPCS exam**
  - https://masters.cs.uchicago.edu/student-resources/placement-exams/
  - Register, take the exam
- **Final exam schedule**
  - Tues, Dec 10^{th}, 10-12pm
- **Review**
  - Textbook
  - Team tutorial
  - Short exercise
  - Programming assignment
  - Extra exercise

Agenda / misc

Key commands

- Loading: read_csv()
- Summarizing:
  - data.head()
  - data.tail()
- Selecting:
  - By index: iloc (e.x. `trees.iloc[:,1]`)
  - By name: loc (e.x. `trees.loc[:,"block_id"]`)

https://www.datacamp.com/cheat-sheet/pandas-cheat-sheet-for-data-science-in-python

Motivating example

- Data exploration for the 2015 New York City Street Tree Survey
- Data: https://data.cityofnewyork.us/Environment/2015-Street-Tree-Census-Tree-Data/uvpi-gqnh
  - CSV, 683,789 lines

1. How many different species are planted as street trees in New York?
2. What are the five most common street tree species in New York?
3. What is the most common street tree species in Brooklyn?
4. What percentage of the street trees in Queens are dead or in poor health?
5. How does street tree health differ by borough?

https://github.com/computer-science-with-applications/examples/tree/main/working_with_data/pandas

Coding practice: 4.3.1

Selecting rows and columns

Note: depending on data, might have 'boroname'

```
In [19]: trees.iloc[:5][["health", "borough"]]
Out[19]:
  health   borough
0   Fair    Queens
1   Fair    Queens
2   Good  Brooklyn
3   Good  Brooklyn
4   Good  Brooklyn
```

```
In [20]: trees.loc[[180683, 200540, 204026], ["health",
    ...: "borough"]]
Out[20]:
       health        borough
180683   Good       Brooklyn
200540    NaN  Staten Island
204026   Good  Staten Island
```

Filtering

- Boolean masks
- Logical operations

```
filter = (trees. borough== "Queens") & ((trees.status == "Dead") | (trees.health == "Poor"))
trees[filter]
```

```
0        False
1        False
2        False
3        False
4        False
         ...
683783   False
683784   False
683785   False
683786   False
683787   False
Name: status, Length: 683788, dtype: bool
```

- Remove rows where trees.status is Dead
- Select rows where trees.status is not Dead

```
trees[trees.status != "Dead"]
```

Coding practice: 4.3.4

Common Summary Statistics

DataFrame.count

Count number of non-NA/null observations.

DataFrame.max

Maximum of the values in the object.

DataFrame.min

Minimum of the values in the object.

DataFrame.mean

Mean of the values.

DataFrame.std

Standard deviation of the observations.

DataFrame.select_dtypes

Subset of a DataFrame including/excluding columns based on their dtype.

In : trees.count()
Out :

```
block_id        683788
created_at      683788
tree_dbh        683788
stump_diam      683788
curb_loc        683788
status          683788
health          652172
spc_latin       652169
spc_common      652169
```

Summary statistics for each column: df.describe()

Summary of categorical values: df.value_counts()

Pairwise correlation of columns: df.corr()

In : trees.status.value_counts()
Out:

```
status
Alive    652173
Stump     17654
Dead      13961
Name: count, dtype: int64
```

Reference: https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.describe.html

TASK: develop a research question

- In your groups, develop a research question you can answer with the **trees dataset**.
- Your question needs to use **ALL of** the following:
  - **Filtering**
  - **Applying a Boolean criterion**
  - **Label-based indexing**
  - **Calculating a numerical value (e.g. mean/min/max/std/corr)**
  - **Exporting the text file (documentation!)**

Using functions

What does this do/mean? Why and how would we do this?

```
for col_name in ["boroname", "health", "spc_common", "status"]:
    trees[col_name] = trees[col_name].astype("category")
```

Using functions

- What does this do/mean? Why and how would we do this?

```
def tree_health_by_boro(trees):
        combined_status = trees.status.where(trees.status != "Alive", trees.health)
        num_per_boro = trees.groupby("borough").size()
        combined_per_boro = trees.groupby(["borough",combined_status]).size()
        pct_per_boro = combined_per_boro/num_per_boro*100.0
        pct_per_boro_df = pct_per_boro.unstack()

        return pct_per_boro_df[["Good", "Fair", "Poor", "Dead", "Stump"]]

tree_health_by_boro(trees)
```

New variables: creating new from old

- Start with continuous-y variables and can reshape – maybe you want or need a categorical, maybe you are looking to reshape
- **Categorical**
- **Cut (and qcut)**
- **Where (np.where)**
- Apply
- Map

- **Bonus: lambda functions**

Pd.cut

- This.is one option to make more variables – I like it because you can slice the variable in different ways. Basically, it's for when you have a continuous-y variable and you want to do some calculating or plotting by groups.

Cutting:

- **Cut the range into equal slices:**

```
trees_life["lifespan_cat_split"] = pd.cut(trees_life['Average_lifespan'],
bins=4,
labels=['Short', 'Medium', 'Long', 'ExtraLong'])
```

- **Cut the distribution into equal slices:**

```
trees_life["lifespan_cat_eq"] = pd.qcut(trees_life['Average_lifespan'],
q=4,
labels=['Short', 'Medium', 'Long', 'ExtraLong'])
```

- **DIY where you set the cut points:**

```
trees_life["lifespan_cat"] = pd.cut(trees_life['Average_lifespan'],
bins=[0, 90, 200, 300, 600], right = True,
labels=['Short', 'Medium', 'Long', 'ExtraLong'])
```

Pivot example

```
filter_col = [col for col in trees_life if col.startswith('lifespan_cat_eq’)]

pd.melt(trees_life,value_vars=filter_col,
id_vars= ["Common Name", "Scientific Name"],
var_name='lifespan_cat’)

pd.wide_to_long(trees_life, stubnames="lifespan_cat_eq",
i="Common Name",
j="lifespan_cat", suffix='\\w+')

trees_life.pivot(index='Common Name', columns='lifespan_cat',
values='Average_lifespan’)
```

*(**bonus**) pd.get_dummies(trees_life, columns=['lifespan_cat_eq'], dtype=int)*

Combining DataFrames

- **concat( )**
  - perform concatenation along an axis
  - while performing set logic of the indexes on other axes
  - make a full copy of the data

- **merge( )**
  - Standard database join operations between DataFrame or Series objects

- **join( )**
  - join on index
  - combine columns of two differently indexed DataFrames into a single one

Reference: https://pandas.pydata.org/docs/user_guide/merging.html

Toy examples

- Consider the following three dataframes:
- df1 = pd.DataFrame([['a', 1], ['b', 2]], columns=['letter', 'number'])
- df2 = pd.DataFrame([['a', 1], ['c', 4]], columns=['letter', 'number'])
- df3 = pd.DataFrame([['b', 1], ['d', 2]], columns=['entry', 'number’])

```
pd.concat([df1,df2])
pd.concat([df1,df3])

pd.merge(df1,df3, on = "number”)
df3.set_index("number").join(df1.set_index("number"))

pd.merge(df1,df2, on = "number") ## Play around with this and ‘how’ options
```

Join vs merge

- Merge is the ‘big picture’ for things
  - Can have more freedom with indices / how you merge
- ‘join’ is like a subset of merge – think of it like a merge based on the index. Can be faster than merge.

Merge: syntax

- **DataFrame.merge(*right*, *how*=*'inner'*, *on*=*None*, *left_on*=*None*, *right_on*=*None*, *left_index*=*False*, *right_index*=*False*, *sort*=*False*, *suffixes*=*('_x', '_y')*, *copy*=*None*, *indicator*=*False*, *validate*=*None*)**

Parameters:

- **how***{'left', 'right', 'outer', 'inner', 'cross'}, default 'inner'*Type of merge to be performed.
  - left: use only keys from left frame, similar to a SQL left outer join; preserve key order.
  - right: use only keys from right frame, similar to a SQL right outer join; preserve key order.
  - outer: use union of keys from both frames, similar to a SQL full outer join; sort keys lexicographically.
  - inner: use intersection of keys from both frames, similar to a SQL inner join; preserve the order of the left keys.
  - cross: creates the cartesian product from both frames, preserves the order of the left keys.

•**On** *label or list*Column or index level names to join on. These must be found in both DataFrames. If *on* is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.merge.html#pandas.DataFrame.merge

Syntax: is it df.merge or pd.merge?

Similar across circumstances but just depends on how you want to call it…when you look at the documentation, you'll see this trend consistently.

If you call pd.function, you will need to specify the data frame(s).

If you df.function, you no longer need to specify the data frame.

Merge hints

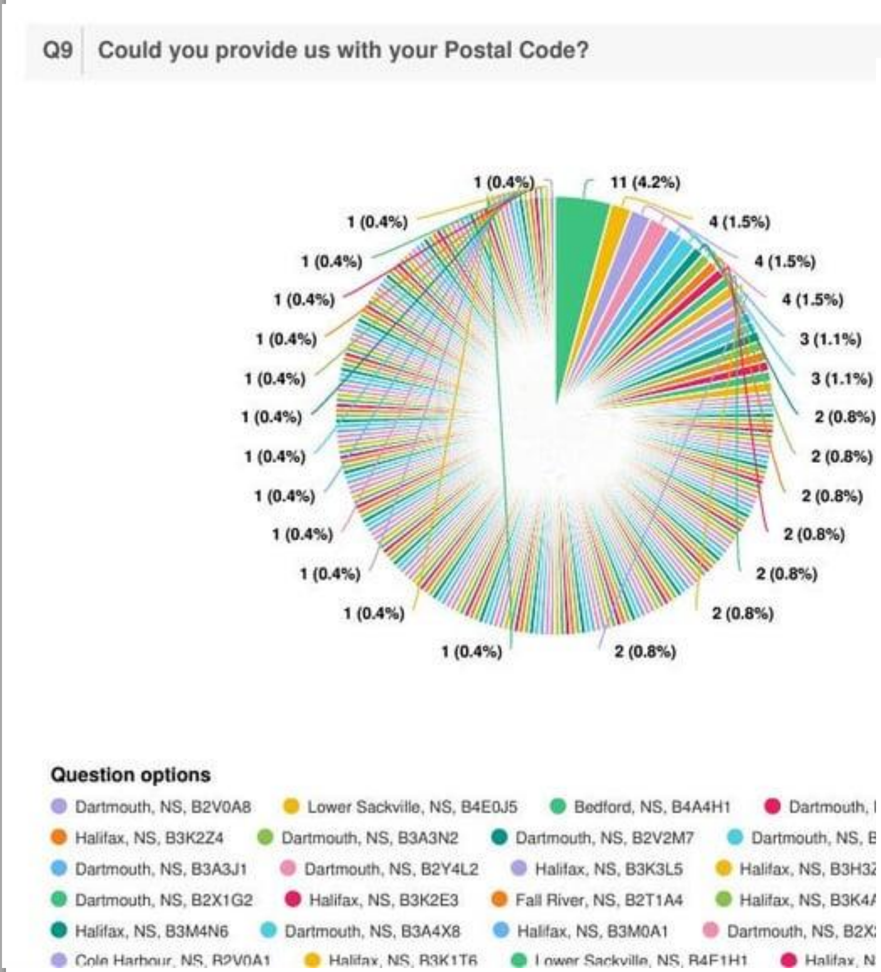
TASK: ADVENTURE TIME (cont'd)

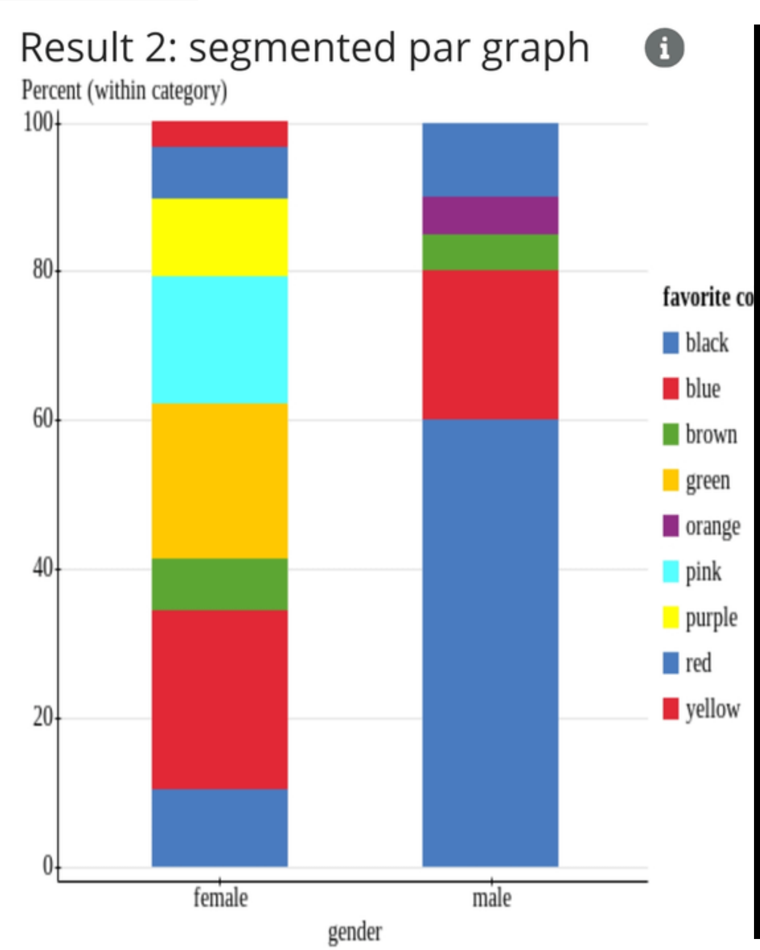
1. Get to know your second dataset
2. Merge the dataframes
3. Think about a question you could ask and give it a go

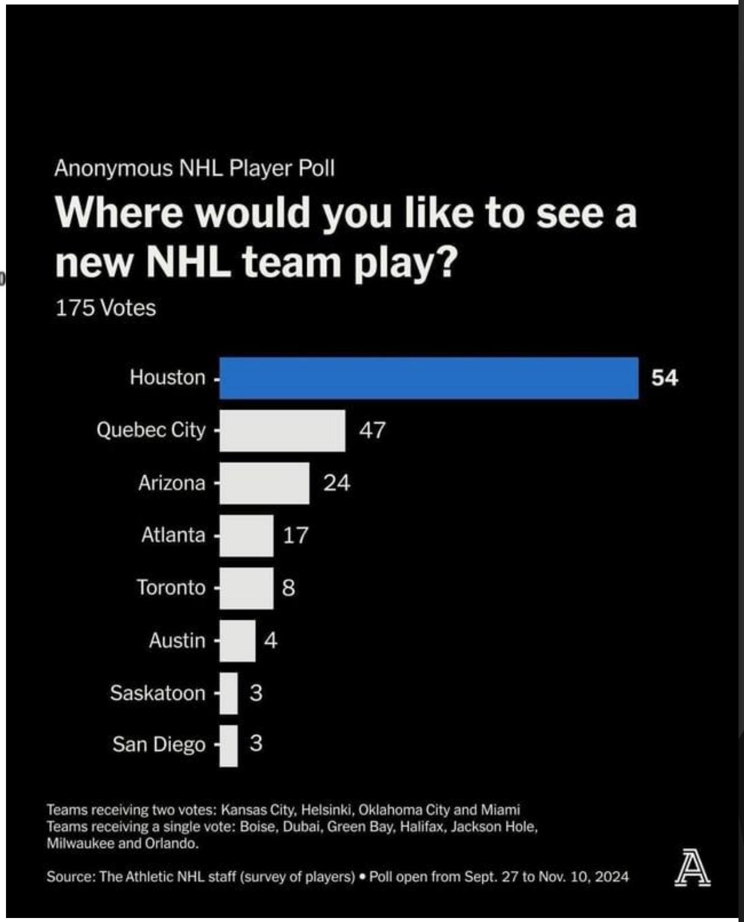
https://github.com/jmclip/macs30111/tree/main/class_examples/pandas_trees

Getting to know graphs

…but first…







https://www.reddit.com/r/dataisugly/

Starting with a summary:

```
def tree_health_by_boro(trees):
combined_status = trees.status.where(trees.status != "Alive", trees.health)
num_per_boro = trees.groupby("borough").size()
combined_per_boro = trees.groupby(["borough",combined_status]).size()
pct_per_boro = combined_per_boro/num_per_boro*100.0
pct_per_boro_df = pct_per_boro.unstack()

return pct_per_boro_df[["Good", "Fair", "Poor", "Dead", "Stump"]]

tree_health_by_boro(trees)
```

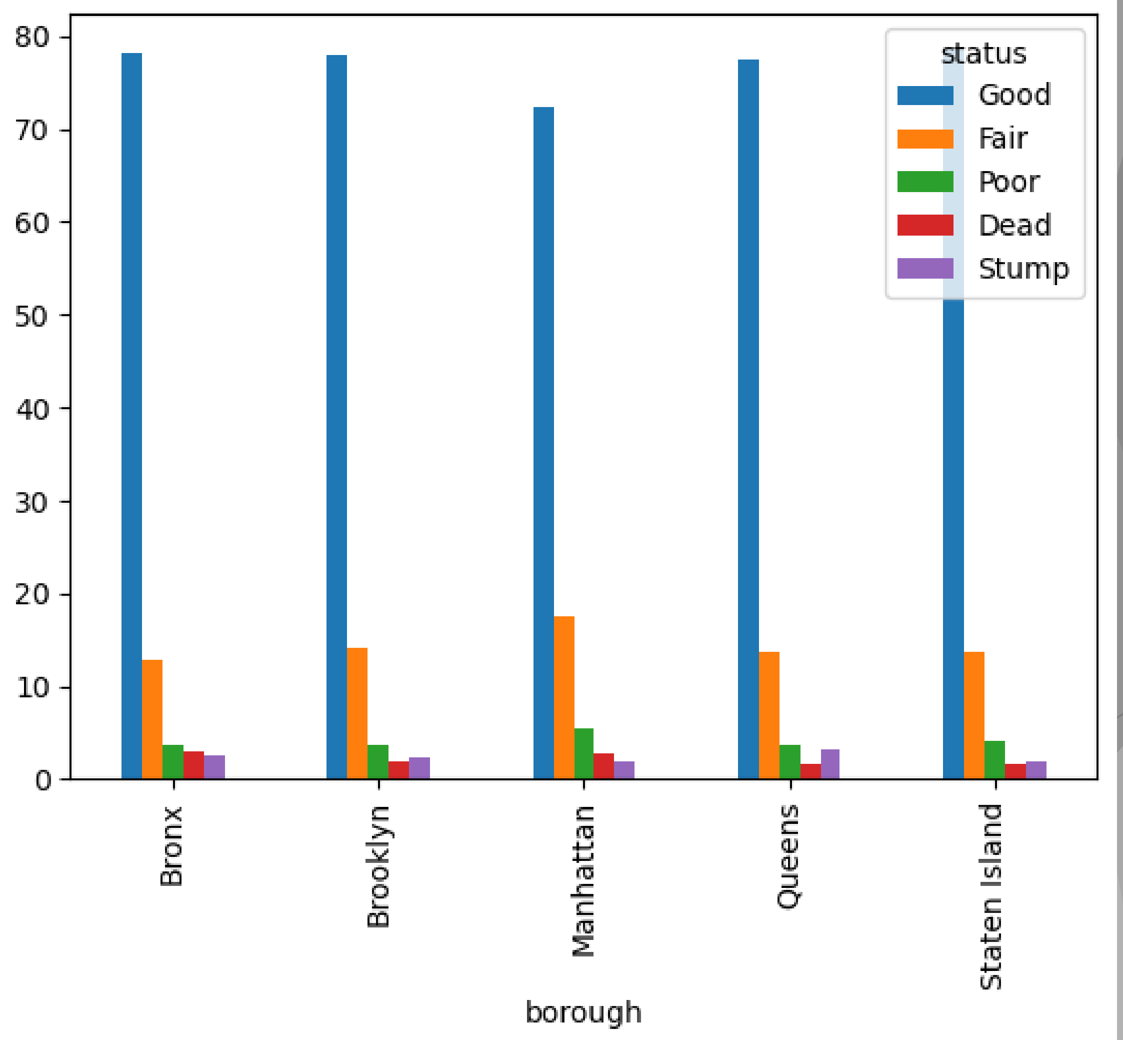
status	Good	Fair	Poor	Dead	Stump
borough					
Bronx	78.169783	12.777719	3.632501	2.969379	2.450618
Brooklyn	77.956829	14.142126	3.643122	1.872042	2.385881
Manhattan	72.387387	17.516775	5.516409	2.754383	1.825046
Queens	77.432539	13.789209	3.758516	1.772094	3.247642
Staten Island	78.494654	13.801060	4.024003	1.775575	1.903758

output

Plotting with Pandas

Using the plot method:

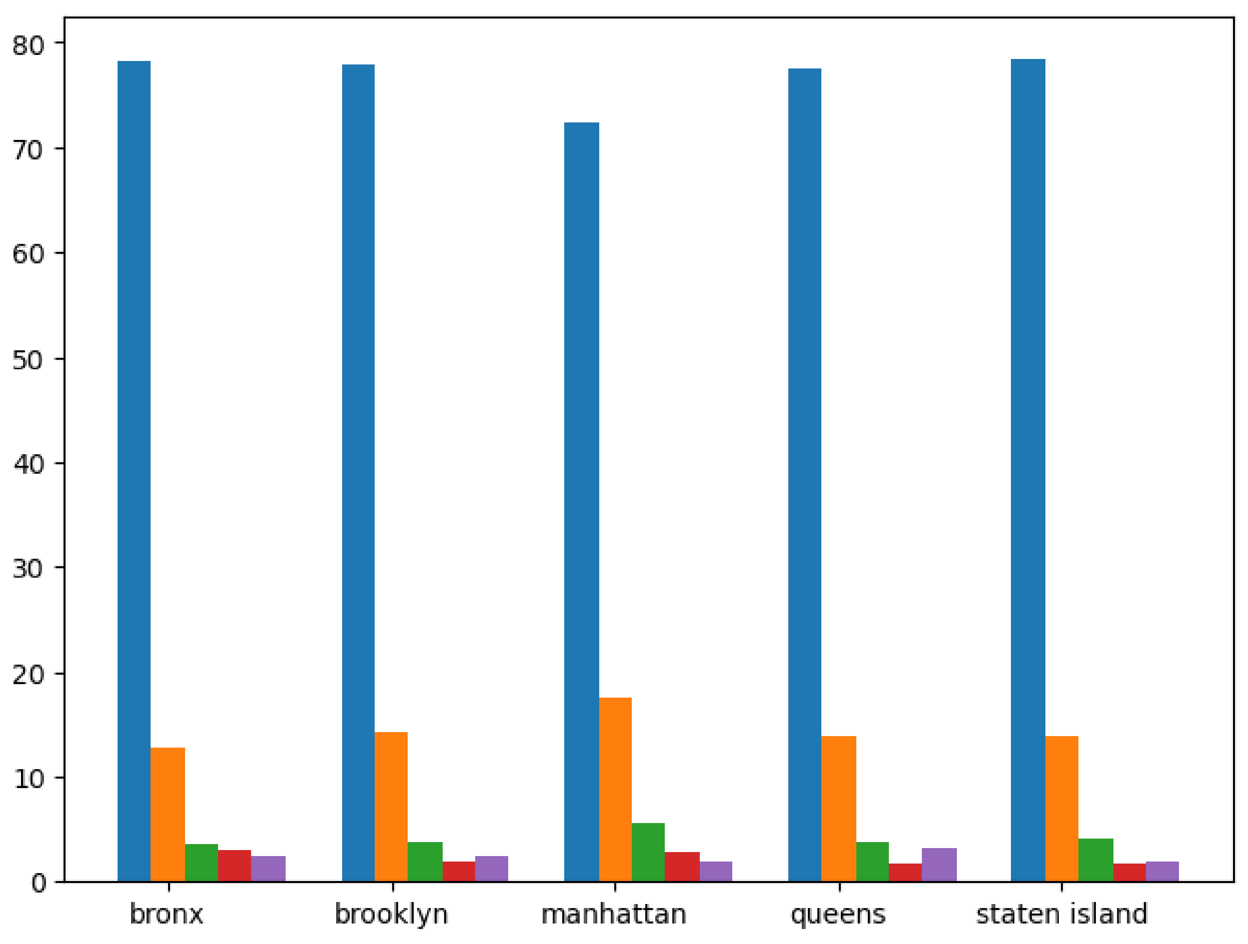
- tree_health_by_boro(trees).plot.bar()



Reference:
https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.plot.html

Plotting with matplotlib

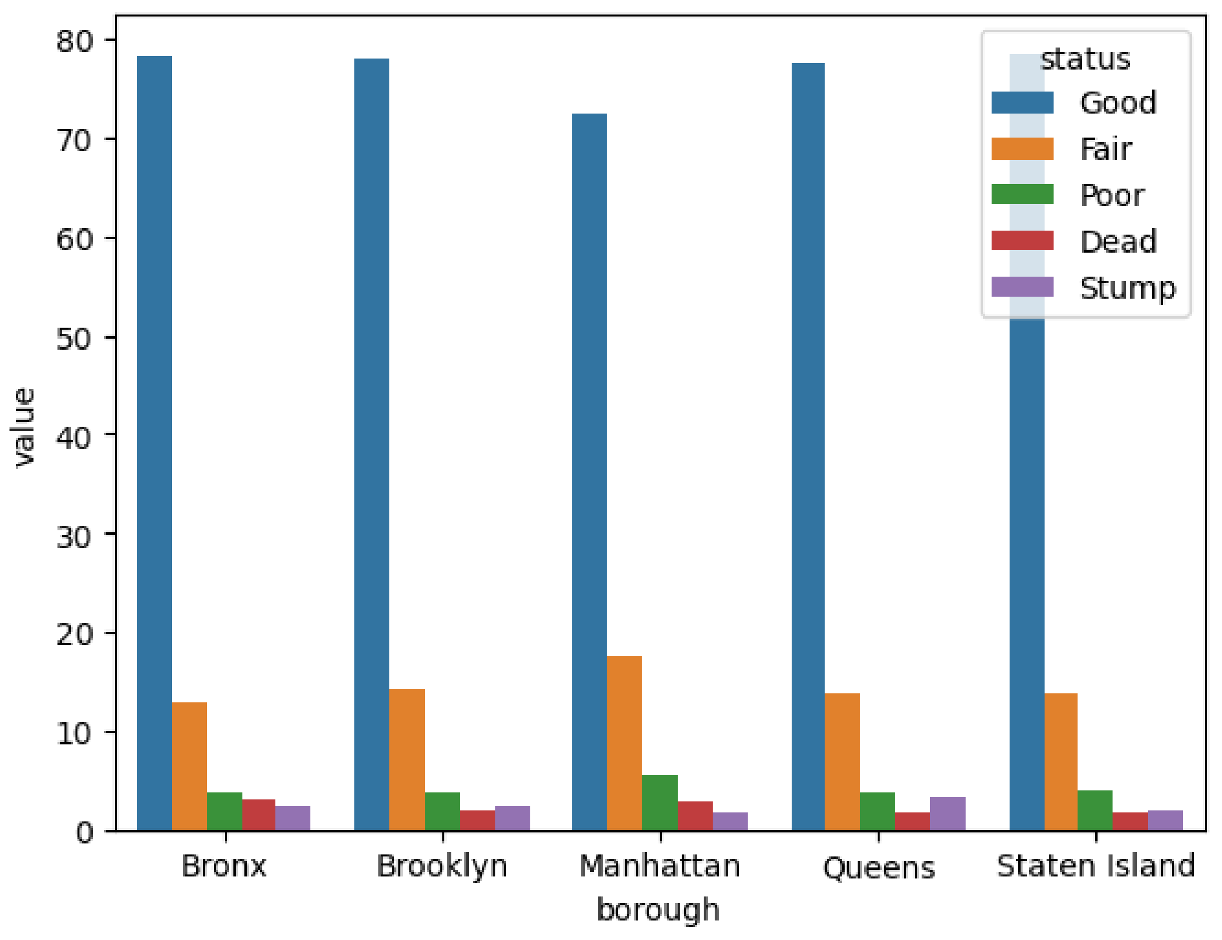
```
import numpy as np
df = pd.DataFrame(tree_health_by_boro(trees))
df.index

#df["Good"]
plt.bar( df.index, df["Good"])
plt.bar(df.index, df["Fair"])
plt.bar(df.index, df["Poor"])
plt.bar(df.index, df["Dead"])
plt.bar(df.index, df["Stump"])


x = np.arange(len(df.index)) # the label locations
width = 0.15 # the width of the bars
multiplier = 0

fig, ax = plt.subplots(layout='constrained')

for attribute, measurement in df.items():
offset = width * multiplier
rects = ax.bar(x + offset, measurement, width, label=attribute)
multiplier += 1
ax.set_xticks(x + width, ["bronx", "brooklyn", "manhattan", "queens", "staten island"] )
plt.show()
```



https://matplotlib.org/cheatsheets/

Plotting with Seaborn

```
import seaborn as sns
df2 = df.reset_index()
df2 = df2.melt(id_vars = "borough")
sns.barplot(df2, x = "borough", y = "value" , hue
= "status")
```



https://www.datacamp.com/cheat-sheet/python-seaborn-cheat-sheet

Which to use?

- Probably not matplotlib…
- Usually, quick and dirty, proof-of-concept: basic **plot** methods but nice and pretty: **seaborn**

- PRACTICE PRACTICE PRACTICE

Misc: other POLARS

Seems cool but pandas is probably better for what you need right now

https://pola.rs/

RECAP

- Pandas is going to be HUGE (ditto NumPy!)
- Think about what you need
- Sometimes it is MUCH faster to puzzle through and sketch before trying to do something – often there is a simpler path
- Think about what you are trying to do and what it will look like.
  - Do you need a new column?
  - Are you summarizing data?
  - Is this a ‘permanent’ or ‘temporary’ alteration?
- Graphing: google search **IMAGES**