CIS 1100

Jupyter Notebooks

Python
Fall 2024
University of Pennsylvania

From `.py` to `.ipynb`

So far, when writing our Python programs, we:

- place code in a `.py` file
- run the program all the way through by writing `python my_file.py`

Jupyter Notebooks (or just Python Notebooks) provide an alternative scheme for writing & running code.

- place code in different **cells** inside of a `.ipynb` file
- run the program piece by piece by executing individual cells

Notebooks and Cells

Jupyter Notebooks consist of several different cells.

- Cells can contain code or text
- Cells can be run individually
- The output of a cell is displayed underneath the cell
- Any effects of the cell are remembered & shared going forward

Benefits & Drawbacks of Notebooks

Notebooks are extremely useful for data science and experimentation:

- Try small snippets and immediately see the results
- Save table outputs & figure drawings in the notebook to look at later

Notebooks are **not a good way to write all programs**:

- Running cells in different orders means that there's no single predictable output of your program
- Very easy to accidentally overwrite variables
- Get cluttered easily

CIS 1100

Pandas & DataFrames

Python
Fall 2024
University of Pennsylvania

Getting Started

- `pandas` is an extremely popular Python library used for managing tabular data
  - programmatic spreadsheets!
- It has so many tools and features, it's almost like learning an entirely new language
  - my role now: teach you a "starter kit"
  - your role later: learn what you need by building off of what you know.
- This material is adapted from the helpful `pandas` user guide

Importing

```
import pandas as pd # !!!
```

Importing

```
import pandas
```

`pandas` Data Structures: `Series`

- A `Series` is a "one-dimensional labeled array holding data of any type"
  - Basically an indexed list
  - Usually only holds values of a single type at a time

`pandas` Data Structures: `DataFrame`

- A `DataFrame` is a "two-dimensional data structure that holds data like a two-dimension array or a **table with rows and columns.**"
  - Basically a spreadsheet
  - Basically a bunch of `Series` grouped together

Creating DataFrames

Technically, these can be built by combining a bunch of sequences together. Pretty rare to actually "build" the `DataFrame` yourself, though!

- More commonly: `pandas.read_csv()` to load a `DataFrame` from a CSV (comma separated values) file
- Others: `pandas.read_excel()`, `pandas.read_parquet()`, etc.

Creating DataFrames from CSV

A CSV file contains several rows of data.

- Each row is broken up into several columns, usually separated by commas
- A row is an individual entry in a dataset
- A column is an individual dimension that each row shares

```
Territory;Bands;Population;Happiness
Afghanistan;2;37466414;2.404
Albania;7;3088385;5.199
Algeria;16;43576691;5.122
Andorra;2;85645;
Angola;8;33642646;
Argentina;1907;45864941;5.967
Armenia;19;3011609;5.399
...
```

Creating DataFrames from CSV

A column in a CSV is like a `Series` or a column in a `DataFrame`

- sequence of values, (usually) all of the same type
- the first value in the column is the **header**, which gives a name to what the column exists to represent

```
Territory
Afghanistan
Albania
Algeria
Andorra
Angola
Argentina
Armenia
...
```

Creating DataFrames from CSV

A row in a CSV is an individual data point in our larger set

- in this case, contains the name & statistics for a single country
- sometimes a row will have *missing data*

```
Algeria;16;43576691;5.122
Andorra;2;85645;
```

Creating DataFrames from CSV

`pandas.read_csv(filename)` turns a CSV file into a `DataFrame`. There are *a gazillion other options,* though.

Keyword Argument	Usage
`sep`	Specify the separator string between columns
`header`	Specify which row gives the column names, if any
`names`	Provide your own "header" by giving the columns your own names
`usecols`	Choose which columns you want to read from the CSV
`index_col`	Choose which column will be the index
`dtype`	Which data type to use for each column

Creating Our DataFrame

```
df = pandas.read_csv("metal_bands.csv", sep=";")
df
```

	Territory	Bands	Population	Happiness
0	Afghanistan	2.0	37466414.0	2.404
1	Albania	7.0	3088385.0	5.199
2	Algeria	16.0	43576691.0	5.122
3	Andorra	2.0	85645.0	NaN
4	Angola	8.0	33642646.0	NaN
...	...	...	...	...
169	Venezuela	343.0	29069153.0	4.925
170	Vietnam	21.0	102789598.0	5.485
171	Yemen	NaN	NaN	4.197
172	Zambia	NaN	NaN	3.760
173	Zimbabwe	2.0	14829988.0	2.995

174 rows × 4 columns

Viewing `DataFrames`

```
df.head()
```

	Territory	Bands	Population	Happiness
0	Afghanistan	2.0	37466414.0	2.404
1	Albania	7.0	3088385.0	5.199
2	Algeria	16.0	43576691.0	5.122
3	Andorra	2.0	85645.0	NaN
4	Angola	8.0	33642646.0	NaN

`.head()` and `.tail()` return views of the top & bottom rows of a `DataFrame`

Viewing `DataFrames`

`.columns` is a variable that shows the column names for a `DataFrame`

```
df.columns
```

```
Index(['Territory', 'Bands', 'Population', 'Happiness'], dtype='object')
```

```
df.describe()
```

	Bands	Population	Happiness
count	145.000000	1.450000e+02	146.000000
mean	523.862069	4.681290e+07	5.553575
std	1673.237176	1.650167e+08	1.086843
min	1.000000	5.321000e+03	2.404000
25%	7.000000	2.711566e+06	4.888750
50%	38.000000	8.884864e+06	5.568500
75%	285.000000	3.364265e+07	6.305000
max	17557.000000	1.397898e+09	7.821000

Viewing `DataFrames`

`.describe()` produces summary statistics for each of the columns in your `DataFrame`

CIS 1100

Missing Values

Python
Fall 2024
University of Pennsylvania

Data is Messy

Every real-world dataset has imperfections in it:

- bias in collecting data (hard for us to fix)
- typos in data entry (might require manual fixes)
- missing or nonsensical values (require systematic fixes)

	Territory	Bands	Population	Happiness
0	Afghanistan	2.0	37466414.0	2.404
1	Albania	7.0	3088385.0	5.199
2	Algeria	16.0	43576691.0	5.122
3	Andorra	2.0	85645.0	NaN
4	Angola	8.0	33642646.0	NaN
...	...	...	...	...
169	Venezuela	343.0	29069153.0	4.925
170	Vietnam	21.0	102789598.0	5.485
171	Yemen	NaN	NaN	4.197
172	Zambia	NaN	NaN	3.760
173	Zimbabwe	2.0	14829988.0	2.995

174 rows × 4 columns

Missing or Nonsensical Values

There are some `NaN` values in rows 3, 4, 171, and 172...

- `NaN` ➡️ not a number
- Value may have been missing in original CSV or typed in an inscrutable way.

Sniffing out `NaN`

```
df.isna().sum()
```

- `.isna()` produces a new `DataFrame` with the same shape, but replacing all `NaN` with `True` and all other values with `False`.
- `.sum()` counts up all of the instances of `True` in a column

```
Territory      0
Bands         29
Population    29
Happiness     28
dtype: int64
```

i.e. 0 rows with `NaN` for `Territory`, 29 rows with `NaN` for `Bands`, etc.

How to Solve `NaN`

There's no individual right answer for how to deal with missing data.

- `.dropna()` produces a new `DataFrame` by
  dropping all rows with at least one missing value.
- `.fillna(value=0)` produces a new `DataFrame` by replacing all values of `NaN` with `0`

On the one hand, losing all of the rows with any missing data can be quite wasteful.

But on the other hand, it's not clear how to interpret missing data. Do your best. 🤷

CIS 1100

Selection

Python
Fall 2024
University of Pennsylvania

Choosing Columns

Three major ways:

1. When the column name is a valid Python identifier, you can use the `.` syntax to select a single column as a `Series`

```
df.Bands
```

```
0         2.0
1         7.0
2        16.0
3         2.0
4         8.0
        ...
169     343.0
170      21.0
171       NaN
172       NaN
173       2.0
Name: Bands, Length: 174, dtype: float64
```

Choosing Columns

Three major ways:

1. When the column name is a valid Python identifier, you can use the `.` syntax to select a single column as a `Series`
2. No matter the column name, you can use indexing `[]` syntax to select a single column as a `Series`

```
df["Bands"]
```

```
0        2.0
1        7.0
2       16.0
3        2.0
4        8.0
       ...
169    343.0
170     21.0
171      NaN
172      NaN
173      2.0
Name: Bands, Length: 174, dtype: float64
```

Choosing Columns

Three major ways:

1. When the column name is a valid Python identifier, you can use the `.` syntax to select a single column as a `Series`
2. No matter the column name, you can use indexing `[]` syntax to select a single column as a `Series`
3. You can index with a list of column names to select several columns as a `DataFrame`

```
df[["Territory", "Bands"]]
```

	Territory	Bands
0	Afghanistan	2.0
1	Albania	7.0
2	Algeria	16.0
3	Andorra	2.0
4	Angola	8.0
...	...	...
169	Venezuela	343.0
170	Vietnam	21.0
171	Yemen	NaN
172	Zambia	NaN
173	Zimbabwe	2.0

174 rows × 2 columns

Choosing Rows

	Territory	Bands	Population	Happiness
10	Bahrain	6.0	1526929.0	6.647
11	Bangladesh	65.0	164098818.0	5.155
12	Barbados	3.0	301865.0	NaN
13	Belarus	293.0	9441842.0	5.821
14	Belgium	666.0	11778842.0	6.805

You can select a *slice* of a `DataFrame` by indexing using range syntax.

```
df[10:20]
```

Choosing Rows

You can select an individual row from a `DataFrame` as a `Series` by using `.iloc[index]`

```
df.iloc[13]
```

```
Territory       Belarus
Bands             293.0
Population    9441842.0
Happiness         5.821
Name: 13, dtype: object
```

CIS 1100

Finer Points about the `DataFrame`

Python
Fall 2024
University of Pennsylvania

	Territory	Bands	Population	Happiness
0	Afghanistan	2.0	37466414.0	2.404
1	Albania	7.0	3088385.0	5.199
2	Algeria	16.0	43576691.0	5.122
3	Andorra	2.0	85645.0	NaN
4	Angola	8.0	33642646.0	NaN
...	...	...	...	...
169	Venezuela	343.0	29069153.0	4.925
170	Vietnam	21.0	102789598.0	5.485
171	Yemen	NaN	NaN	4.197
172	Zambia	NaN	NaN	3.760
173	Zimbabwe	2.0	14829988.0	2.995

174 rows × 4 columns

The Index

Like any sequence, a `Series` and `DataFrame` will always have an **index**

- `Index` is actually a special type in Pandas
- By default, a `DataFrame` read from a CSV is indexed by row number
- Unlike lists, Pandas data doesn't have to be indexed by numbers

Choosing the Index

- Choose the index by: `df = df.set_index(keys)`
  - `keys` can be a single column name (as `str`) or a list of columns
- `Territory` is no longer a column and so cannot be easily modified 🤦

```
df2 = df.copy()
df2 = df2.set_index("Territory")
df2.head()
```

Territory	Bands	Population	Happiness	Bands Per Capita
Afghanistan	2.0	37.466414	2.404	0.053381
Albania	7.0	3.088385	5.199	2.266557
Algeria	16.0	43.576691	5.122	0.367169
Andorra	2.0	0.085645	NaN	23.352210
Angola	8.0	33.642646	NaN	0.237793

Queries over the Index

```
df2["Belarus":"Botswana"]
```

Territory	Bands	Population	Happiness	Bands Per Capita
Belarus	293.0	9.441842	5.821	31.032080
Belgium	666.0	11.778842	6.805	56.542061
Belize	1.0	0.405633	NaN	2.465283
Benin	NaN	NaN	4.623	NaN
Bolivia	243.0	11.758869	5.600	20.665253
Bosnia and Herzegovina	86.0	3.824782	5.768	22.484942
Botswana	7.0	2.350667	3.471	2.977878

If our index is now the `Territory`, we can select ranges of territory names using `[start:stop]`

Operations Create New `DataFrames`

```
df.rename(columns={"Territory" : "Name"})
```

	Name	Bands	Population	Happiness	Bands Per Capita
0	Afghanistan	2.0	37.466414	2.404	0.053381
1	Albania	7.0	3.088385	5.199	2.266557
2	Algeria	16.0	43.576691	5.122	0.367169
3	Andorra	2.0	0.085645	NaN	23.352210
4	Angola	8.0	33.642646	NaN	0.237793
...	...	...	...	...	...

Suppose we want to rename a column.

`.rename(columns=name_mapping)`

comes in handy!

Operations Create New `DataFrames`

OK. Then, let's just remind ourselves what the `DataFrame` looks like with `df.columns`...

```
Index(['Territory', 'Bands', 'Population', 'Happiness', 'Bands Per Capita'], dtype='object')
```

Where did our `Name` column go? Why is `Territory` still there? 😠😠😠😠

Operations Create New `DataFrames`

Operations that seem to modify a `DataFrame` actually create **new** tables.

- The `DataFrame` that a function was called on is totally unchanged
- If we want the modification to apply to the `DataFrame` we're working with, we have to save it back into the variable:

```
df = df.rename(columns={"Territory" : "Name"})
```

- If you just want to see what an operation does without modifying the `DataFrame`, then you can perform the operation without saving it back:

```
df.rename(columns={"Territory" : "Name"}) # not "permanent"
```

CIS 1100

Filtering

Python
Fall 2024
University of Pennsylvania

Boolean Indexing

```
df["Bands"] > 50
```

```
0      False
1      False
2      False
3      False
4      False
       ...
169     True
170    False
171    False
172    False
173    False
Name: Bands, Length: 174, dtype: bool
```

Pandas allows you to write boolean expressions over a `Series`/`DataFrame` that we typically do on individual values:

- (assuming `x` is a number) `x > 50` is an expression that produces either `True` or `False`
- `df["Bands"]` is a `Series` of numbers, and `df["Bands" > 50]` is a `Series` of booleans.

```
df[df["Bands"] > 50]
```

	Territory	Bands	Population	Happiness
5	Argentina	1907.0	45864941.0	5.967
7	Australia	1545.0	25809973.0	7.162
8	Austria	664.0	8884864.0	7.163
11	Bangladesh	65.0	164098818.0	5.155
13	Belarus	293.0	9441842.0	5.821
...	...	...	...	...
163	Ukraine	715.0	43745640.0	5.084
165	United Kingdom	3244.0	66052076.0	6.943
166	United States	17557.0	334998398.0	6.977
167	Uruguay	121.0	3398239.0	6.474
169	Venezuela	343.0	29069153.0	4.925

70 rows × 4 columns

Boolean Indexing & Filtering

- Alone, a `Series` of booleans may not be so interesting.
- Used as an index into another `DataFrame`, this `Series` allows us to **select only those rows that meet a condition.**

"Show me the rows of `df` where the number of bands is greater than 50!"

More Complex Filtering

Same as logical `and` / `or`, we can combine filtering expressions in Pandas.

- Need to use `&` for "and", `|` for "or"
- Annoyingly, terms usually need to be wrapped in parentheses

```
# "Many bands AND lower happiness"
df[(df["Bands"] > 50) & (df["Happiness"] < 4.0)]

# "Few bands OR low population"
df[(df["Bands"] < 10) | (df["Population"] < 5000000)]
```

```
df[df["Bands"] > 50]
```

	Territory	Bands	Population	Happiness
5	Argentina	1907.0	45864941.0	5.967
7	Australia	1545.0	25809973.0	7.162
8	Austria	664.0	8884864.0	7.163
11	Bangladesh	65.0	164098818.0	5.155
13	Belarus	293.0	9441842.0	5.821
...	...	...	...	...
163	Ukraine	715.0	43745640.0	5.084
165	United Kingdom	3244.0	66052076.0	6.943
166	United States	17557.0	334998398.0	6.977
167	Uruguay	121.0	3398239.0	6.474
169	Venezuela	343.0	29069153.0	4.925

70 rows × 4 columns

Boolean Indexing & Filtering

"Show me the rows of `df` where the number of bands is greater than 50 and the happiness is less than `4.0`."

```
df[(df["Bands"] > 50) & (df["Happiness"] < 4.0)]
```

CIS 1100

Setting

Python
Fall 2024
University of Pennsylvania

Systematically Modifying a Column

Suppose "true" population numbers are too unwieldy—
would like to work with population as number of millions

- A little easier to filter:
  - `df["Population"] < 4` instead of `df["Population"] < 4000000`
- Can set replace a column's values by setting that column equal to a new `Series` **with a compatible index.**

Scaling Population

```
df["Population"] = df["Population"] / 1000000
df
```

	Territory	Bands	Population	Happiness
0	Afghanistan	2.0	37.466414	2.404
1	Albania	7.0	3.088385	5.199
2	Algeria	16.0	43.576691	5.122
3	Andorra	2.0	0.085645	NaN
4	Angola	8.0	33.642646	NaN
...	...	...	...	...
169	Venezuela	343.0	29.069153	4.925
170	Vietnam	21.0	102.789598	5.485
171	Yemen	NaN	NaN	4.197
172	Zambia	NaN	NaN	3.760
173	Zimbabwe	2.0	14.829988	2.995

`df["Population"] = df["Population"] / 1000000`

- RHS: create a new `Series` with all of the values of `df["Population"]` but scaled down by a factor of a million
- LHS: replace the `df["Population"]` column with the RHS
- **This modifies the `DataFrame`!**

	Territory	Bands Per Capita
0	Afghanistan	0.053381
1	Albania	2.266557
2	Algeria	0.367169
3	Andorra	23.352210
4	Angola	0.237793
...	...	...
169	Venezuela	11.799449
170	Vietnam	0.204301
171	Yemen	NaN
172	Zambia	NaN
173	Zimbabwe	0.134862

Creating a New Column

Suppose we want a notion of "bands per capita", or number of heavy metal bands per one million citizens.

```
df["Bands Per Capita"] = df["Bands"] / df["Population"]
df[["Territory", "Bands Per Capita"]]
```

Setting is Permanent!

- If you overwrite a column, that data is lost unless you "reload" from the source data
- If you add a new column, it's stuck in the data frame unless you remove it manually
  - Remove Columns: `df = df.drop(columns=["Bands Per Capita"])`

CIS 1100

Strings & Dealing with Types

Python
Fall 2024
University of Pennsylvania

Syntax for String Operations

When performing operations on string-valued columns, you need to insert `str` into the function call.

Operation	Call
Getting the length of country names	`df["Territory"].str.len()`
Converting country names to all lowercase	`df["Territory"].str.lower()`
Removing any whitespace around a country's name	`df["Territory"].str.strip()`
Replacing spaces within names with `_`	`df["Territory"].str.replace(" ", "_")`

All of these still need to be "saved back" to be permanent.

Concatenating Strings

Suppose I have a `DataFrame` with first names and last names. I might want to use/store full names as well!

```
roster = pandas.DataFrame([{"first" : "Harry", "last" : "Smith"},
                           {"first" : "Henry", "last" : "Gifford"},
                           {"first" : "Travis", "last" : "McGaha"}])
roster
```

	first	last
0	Harry	Smith
1	Henry	Gifford
2	Travis	McGaha

Concatenating Strings

`this_col.str.cat(other_col)` allows you to join the contents of one string `Series` with another:

```
roster["full"] = roster["first"].str.cat(roster["last"])
roster
```

	first	last	full
0	Harry	Smith	HarrySmith
1	Henry	Gifford	HenryGifford
2	Travis	McGaha	TravisMcGaha

... we probably need a space.

Concatenating Strings

`this_col.str.cat(other_col, sep=" ")` allows you to join the contents of one string `Series` with another, adding a space in between:

```
roster["full"] = roster["first"].str.cat(roster["last"], sep=" ")
roster
```

	first	last	full
0	Harry	Smith	Harry Smith
1	Henry	Gifford	Henry Gifford
2	Travis	McGaha	Travis McGaha

Checking for Patterns

`str.contains(pattern)` allows you to check if a column contains a certain pattern.

- `pattern` can be a string specifying a pattern literally or a *regular expression* 👀
- returns a boolean series, so can be used for Boolean Indexing (filtering)

```
df[df["Territory"].str.contains("New")]
```

	Territory	Bands	Population	Happiness	Bands Per Capita
113	New Caledonia	5.0	0.293608	NaN	17.029509
114	New Zealand	288.0	4.991442	7.2	57.698757

Splitting Strings

```
df["split_names"] = df["Territory"].str.split(" ")
df
```

	Territory	Bands	Population	Happiness	Bands Per Capita	split_names
0	Afghanistan	2.0	37.466414	2.404	0.053381	[Afghanistan]
1	Albania	7.0	3.088385	5.199	2.266557	[Albania]
2	Algeria	16.0	43.576691	5.122	0.367169	[Algeria]
3	Andorra	2.0	0.085645	NaN	23.352210	[Andorra]
4	Angola	8.0	33.642646	NaN	0.237793	[Angola]
...	...	...	...	...	...	...
169	Venezuela	343.0	29.069153	4.925	11.799449	[Venezuela]
170	Vietnam	21.0	102.789598	5.485	0.204301	[Vietnam]
171	Yemen	NaN	NaN	4.197	NaN	[Yemen]
172	Zambia	NaN	NaN	3.760	NaN	[Zambia]
173	Zimbabwe	2.0	14.829988	2.995	0.134862	[Zimbabwe]

`str.split(sep)` splits a column of strings into lists of strings

- This is a little weird: the value stored in a cell of the `DataFrame` is now a list...

Splitting Strings

```
df["split_names"].str.get(0)
```

```
0      Afghanistan
1          Albania
2          Algeria
3          Andorra
4           Angola
           ...
169       Venezuela
170         Vietnam
171           Yemen
172          Zambia
173        Zimbabwe
Name: split_names, Length: 174, dtype: object
```

`str.split(sep)` splits a column of strings into lists of strings

- To get the value at a given index of a list in a `DataFrame`, use `.str.get()`

Splitting Strings

```
df["split_names"].str.len()
```

```
0      1
1      1
2      1
3      1
4      1
      ..
169    1
170    1
171    1
172    1
173    1
Name: split_names, Length: 174, dtype: int64
```

`str.split(sep)` splits a column of strings into lists of strings

- To get the length a list in a `DataFrame`, use `.str.len()`

Splitting Strings

Can use string splitting & length checking to ask more complex questions:

"Show me the countries that have more than one-word names."

```
df[df["split_names"].str.len() > 1]
```

	Territory	Bands	Population	Happiness	Bands Per Capita	split_names
18	Bosnia and Herzegovina	86.0	3.824782	5.768	22.484942	[Bosnia, and, Herzegovina]
23	Burkina Faso	NaN	NaN	4.670	NaN	[Burkina, Faso]
33	Costa Rica	235.0	5.151140	6.582	45.620969	[Costa, Rica]
37	Czech Republic	964.0	10.702596	6.920	90.071605	[Czech, Republic]
39	Dominican Republic	31.0	10.597348	5.737	2.925260	[Dominican, Republic]
40	East Timor	1.0	1.413958	NaN	0.707235	[East, Timor]
43	El Salvador	96.0	6.528135	6.120	14.705578	[El, Salvador]

CIS 1100

Plotting

Python
Fall 2024
University of Pennsylvania

Plotting with `.plot()`

`DataFrame.plot()` is an extremely powerful function. On its own, it can do:

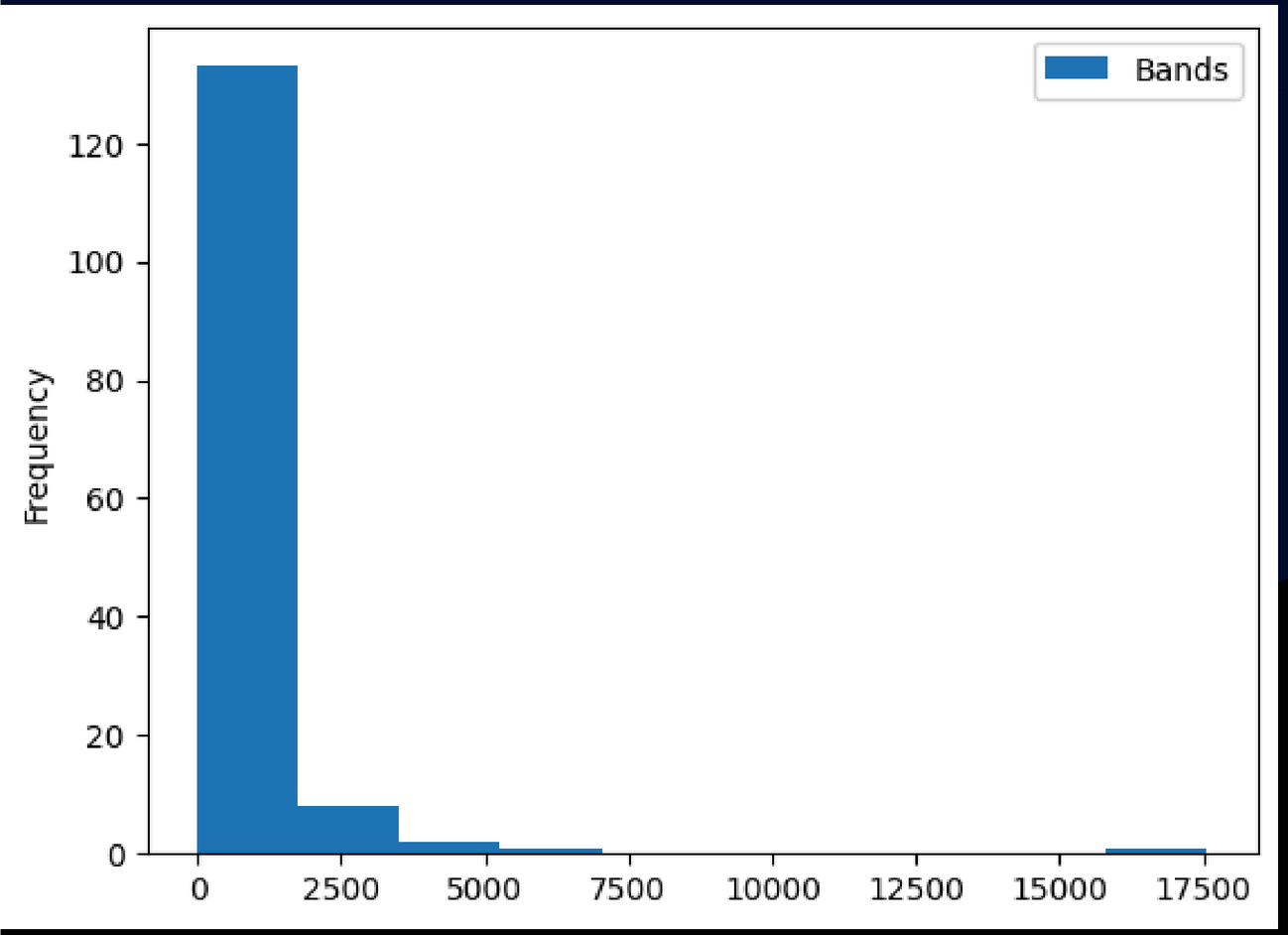
- line plots
- bar plots
- horizontal bar plots
- histograms
- box plots
- Kernel Density Estimation plots
- area plots
- pie plots
- scatter plots
- hexbin plots

Plotting with `.plot()`

`DataFrame.plot()` is an extremely powerful function.

- In later lectures, we'll talk about all the finer points of data visualization
- For now, just READ THE DOCUMENATION!

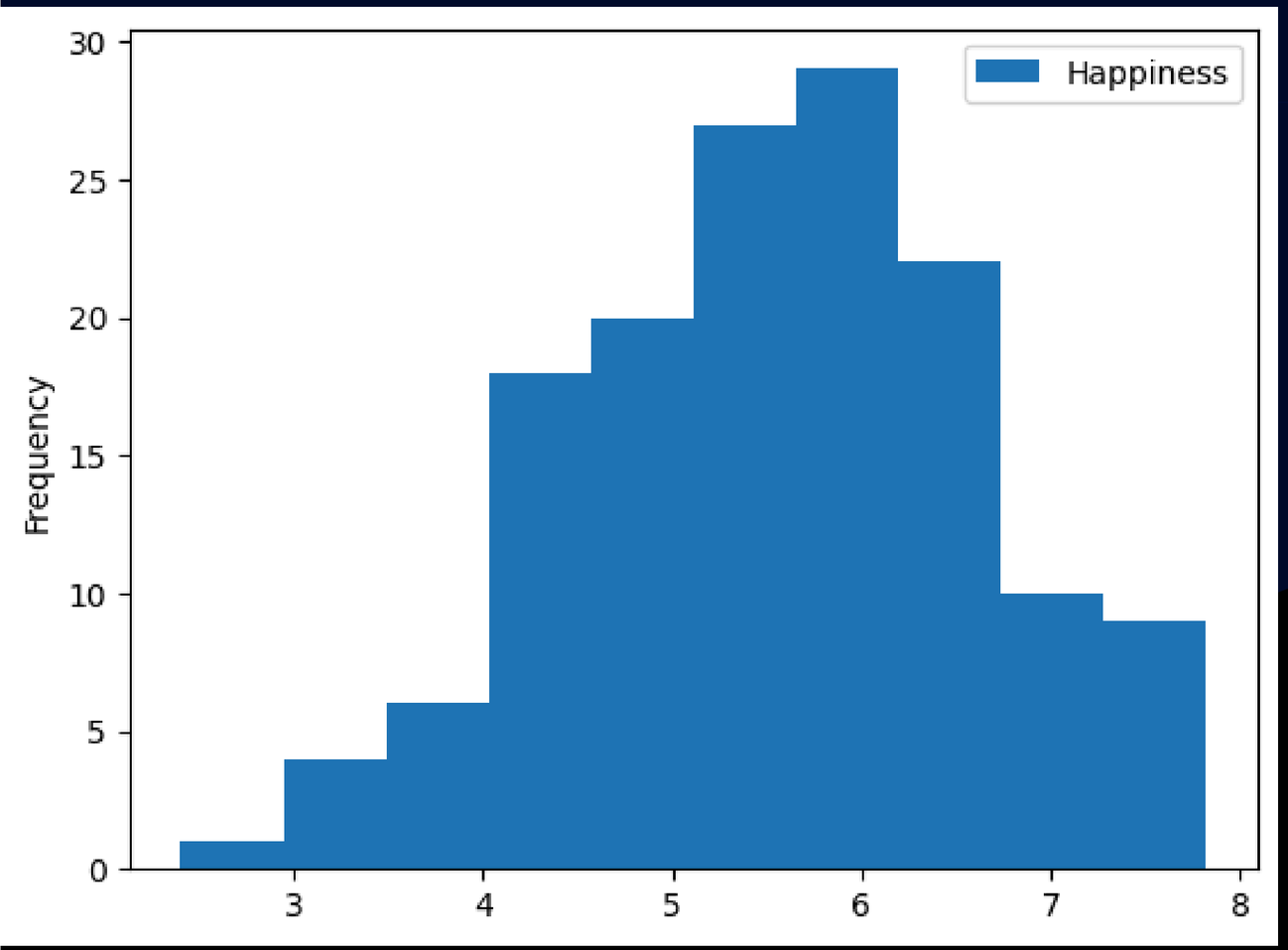
Histograms



What's the overall distribution of a given column in my DataFrame?

```
df.plot(kind="hist", y="Bands")
```

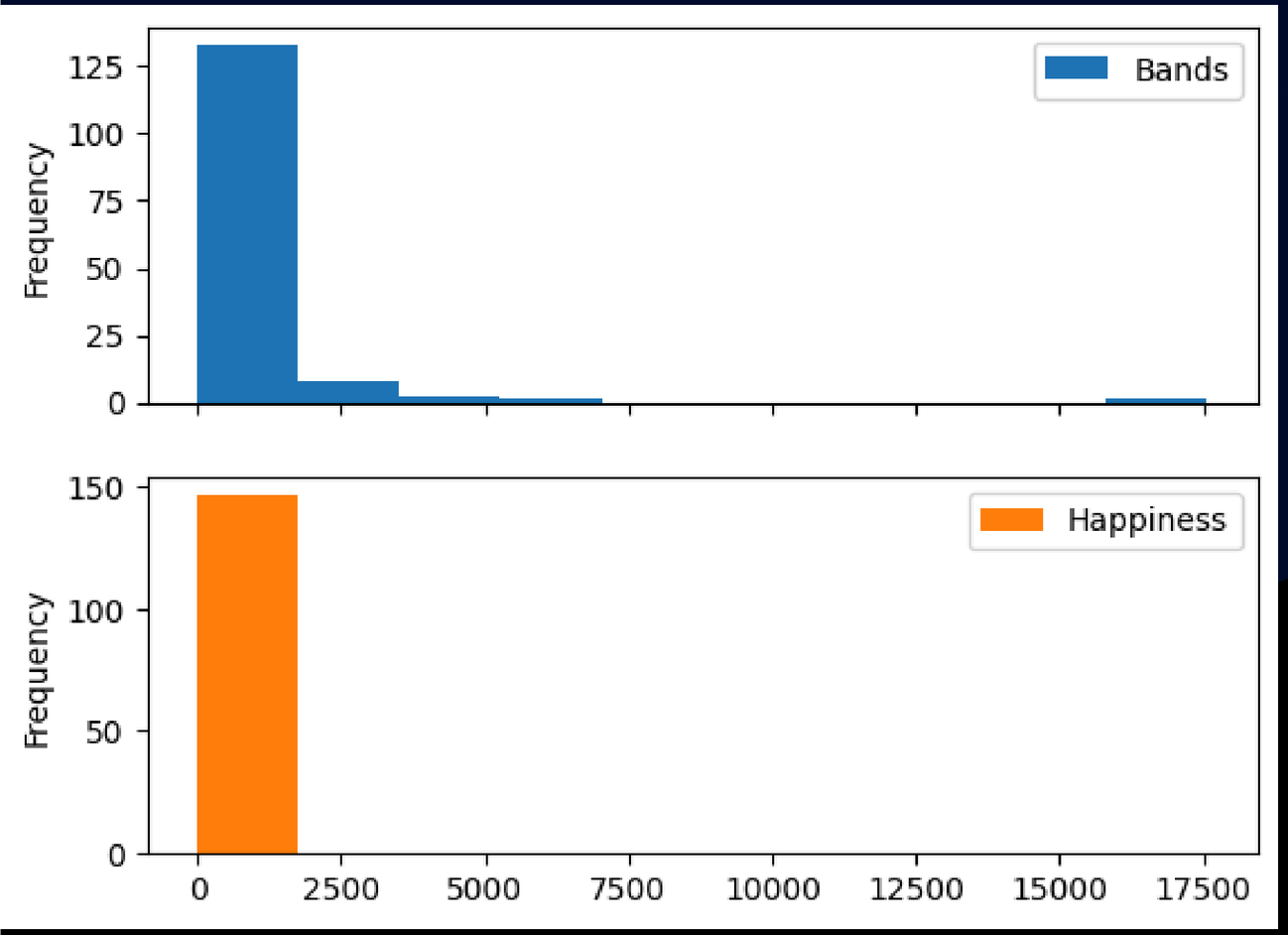
Histograms



Change `y` to pick a different column to map out:

```
df.plot(kind="hist", y="Happiness")
```

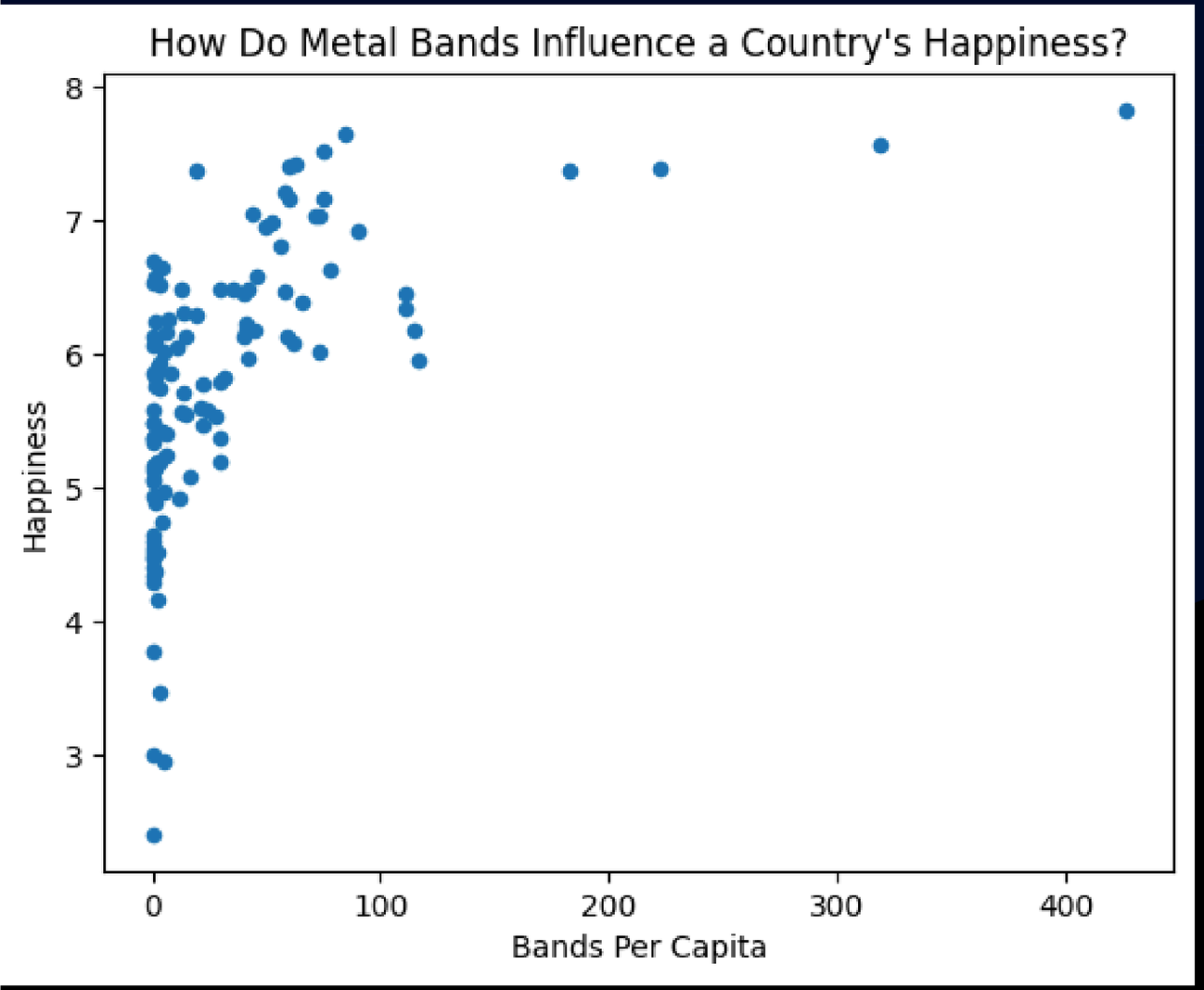
Histograms



Create a list of columns for `y` and use `subplots=True` to generate multiple plots at once. **Needs refinement...** 😄

```
df.plot(kind="hist", y=["Bands", "Happiness"],
        subplots=True)
```

Scatter Plots



Not an excellent plot, but...

```
df.plot(kind="scatter", x="Bands Per Capita", y="Happiness",
        title="How Do Metal Bands Influence a Country's Happiness?")
```

- `x`: the column to show on the x axis
- `y`: the column to show on the y axis
- `title`: add a title to the figure

Plotting

- Best learned by trying things with different data sets
- You can read about all of the kinds of plots that come with Pandas and their options by reading the documentation
- Will talk in later modules about best practices for picking & designing plots
- Will work in class to make different kinds of plots with different datasets.