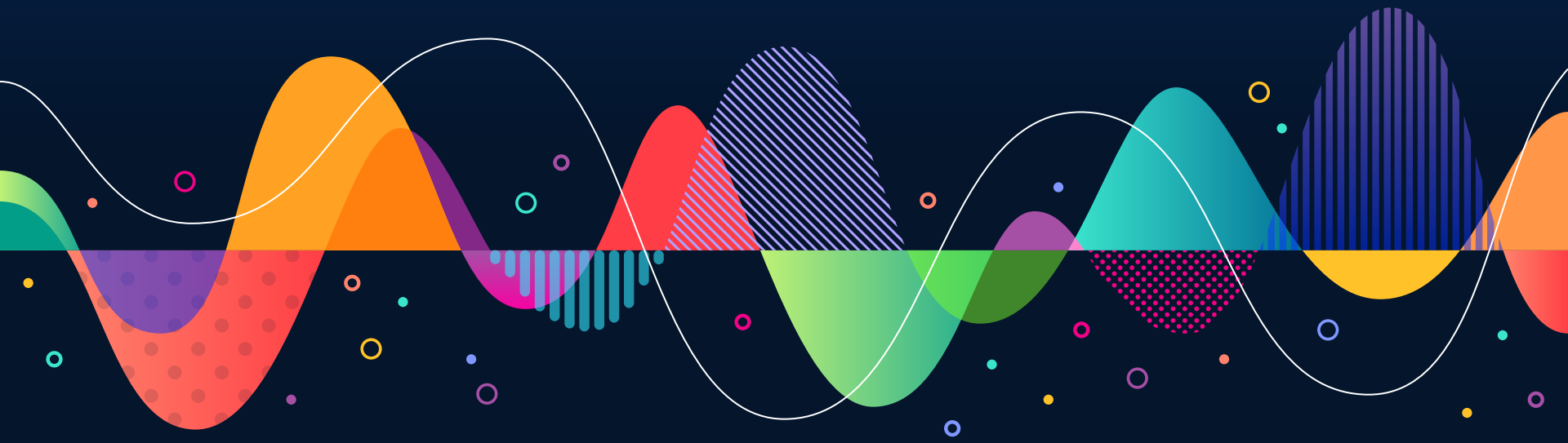
Python Visualization Libraries



USE DIAGRAMS TO EXPLAIN YOUR IDEAS

matplotlib

- The 800-pound gorilla – and like most 800-pound gorillas, this one should probably be avoided unless you genuinely need its power, e.g., to make a custom plot or produce a publication-ready graphic.
- As we'll see, when it comes to statistical visualization, the preferred tack might be: "do as much as you easily can in your convenience layer of choice [i.e., any of the next four libraries], and then use matplotlib for the rest."
- http://matplotlib.org/

pandas

- “Come for the DataFrames; stay for the plotting convenience functions that are arguably more pleasant than the matplotlib code they supplant.”
- https://pandas.pydata.org/pandas-docs/stable/user_guide/visualization.html

Seaborn

- Seaborn has long been my go-to library for statistical visualization; it summarizes itself thusly: “If matplotlib ‘tries to make easy things easy and hard things possible,’ seaborn tries to make a well-defined set of hard things easy too”
- https://stanford.edu/~mwaskom/software/seaborn/

yhat's ggplot

- A Python implementation of the wonderfully declarative ggplot2. It is a Python implementation of the grammar of graphics for R--though there is much greatness in ggplot2, the Python world could stand to benefit from it.
- https://github.com/yhat/ggpy

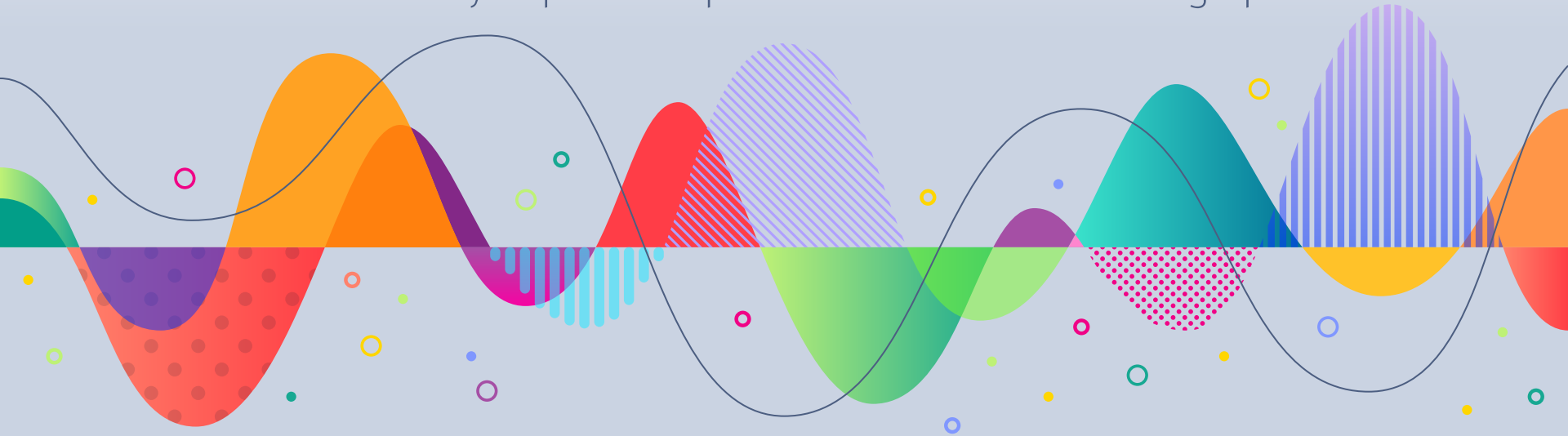
Altair

- The new guy, Altair is a “declarative statistical visualization library” with an exceedingly pleasant API. With Altair, you can spend more time understanding your data and its meaning. Altair’s API is simple, friendly and consistent. This elegant simplicity produces beautiful and effective visualizations with a minimal amount of code.
- https://github.com/ellisonbg/altair

1.

LINES AND DOTS

How would you plot multiple time series on the same graph?



Lines and dots

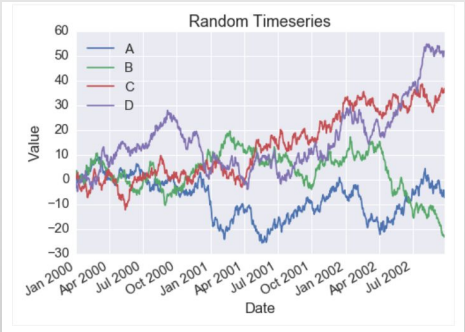
- We'll be dealing with a tidy data set named "**ts**." It consists of three columns: a "**dt**" column (for dates); a "**value**" column (for values); and a "**kind**" column, which has four unique levels: A, B, C, and D. Here's a preview...

	dt	kind	value
0	2000-01-01	A	1.442521
1	2000-01-02	A	1.981290
2	2000-01-03	A	1.586494
3	2000-01-04	A	1.378969
4	2000-01-05	A	-0.277937

Matplotlib, first method

- In the first method, I loop through your trumped-up matrix – or call it a “Data” “Frame” – and subset it to the relevant time series. Next, I invoke my “plot” method and pass in the relevant columns from that subset.

```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1,
                       figsize=(7.5, 5))

for k in ts.kind.unique():
    tmp = ts[ts.kind == k]
    ax.plot(tmp.dt, tmp.value, label=k)

ax.set(xlabel='Date',
       ylabel='Value',
       title='Random Timeseries')

ax.legend(loc=2)
fig.autofmt_xdate()
```



Matplotlib, second method

- Transforming the data into an index with four columns – one for each line I want to plot.

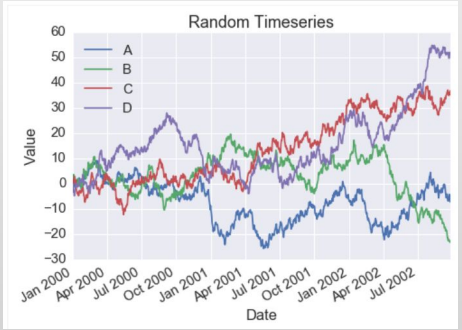
```
# the notion of a tidy dataframe matters not here
dfp = ts.pivot(index='dt', columns='kind', values='value')
dfp.head()
```

kind	A	B	C	D
dt				
2000-01-01	1.442521	1.808741	0.437415	0.096980
2000-01-02	1.981290	2.277020	0.706127	-1.523108
2000-01-03	1.586494	3.474392	1.358063	-3.100735
2000-01-04	1.378969	2.906132	0.262223	-2.660599
2000-01-05	-0.277937	3.489553	0.796743	-3.417402

Matplotlib, second method

- In the first method, I loop through your trumped-up matrix – or call it a “Data” “Frame” – and subset it to the relevant time series. Next, I invoke my “plot” method and pass in the relevant columns from that subset.

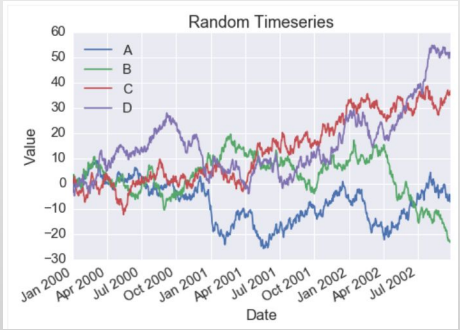
```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1,
                       figsize=(7.5, 5))

ax.plot(dfp)

ax.set(xlabel='Date',
       ylabel='Value',
       title='Random Timeseries')

ax.legend(dfp.columns, loc=2)
fig.autofmt_xdate()
```



pandas

- We are using the same dfp (the one we got from the pivot command) and do the same thing as before.

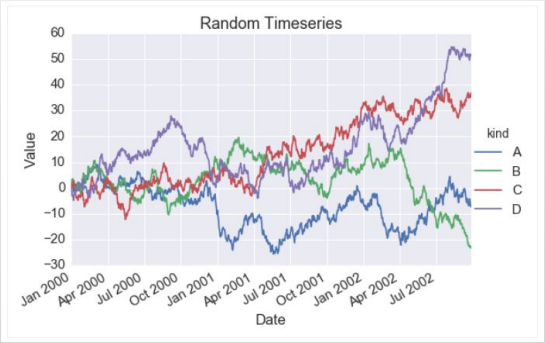
```
# PANDAS
fig, ax = plt.subplots(1, 1,
                       figsize=(7.5, 5))

dfp.plot(ax=ax)

ax.set(xlabel='Date',
       ylabel='Value',
       title='Random Timeseries')

ax.legend(loc=2)
fig.autofmt_xdate()
```



Seaborn

- Seems like an awful lot of data manipulation for a silly line graph. I mean, for loops and pivoting? Luckily, we have this thing called a FacetGrid.

```
# SEABORN
g = sns.FacetGrid(ts, hue='kind', size=5, aspect=1.5)
g.map(plt.plot, 'dt', 'value').add_legend()
g.ax.set(xlabel='Date',
         ylabel='Value',
         title='Random Timeseries')
g.fig.autofmt_xdate()
```

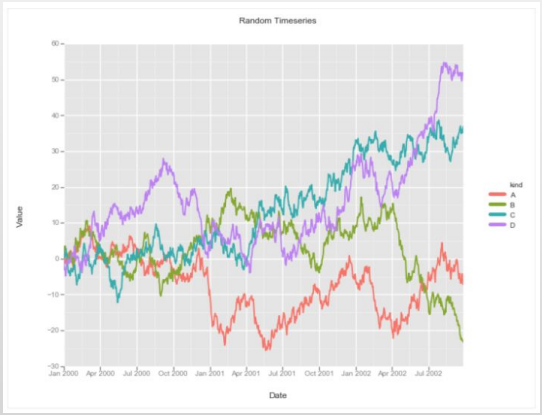


- See? You hand FacetGrid your un-manipulated tidy data. At that point, passing in “kind” to the “hue” parameter means you’ll plot four different lines – one for each level in the “kind” field.

ggplot

- something similar...

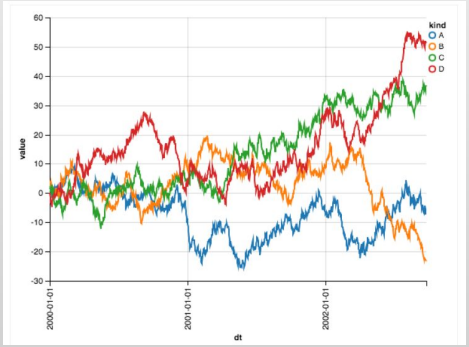
```
# GGPLOT
fig, ax = plt.subplots(1, 1, figsize=(7.5, 5))

g = ggplot(ts, aes(x='dt', y='value', color='kind')) + \
        geom_line(size=2.0) + \
        xlab('Date') + \
        ylab('Value') + \
        ggtitle('Random Timeseries')
g
```



Altair

- You give my Chart class some data and tell it what kind of visualization you want: in this case, it's "mark_line". Next, you specify your aesthetic mappings: our x-axis needs to be "date"; our y-axis needs to be "value"; and we want to split by kind, so we pass "kind" to "color."

```
# ALTAIR
c = Chart(ts).mark_line().encode(
    x='dt',
    y='value',
    color='kind'
)
c
```



Comments and thoughts

- In matplotlib and pandas, you must either make multiple calls to the "plot" function (e.g., once-per-for loop), or you must manipulate your data to make it optimally fit the plot function (e.g., pivoting).
- Conversely, ggplot and Altair implement similar and declarative "grammar of graphics"-approved ways to handle our simple case:
  - you give their "main" function– "ggplot" in ggplot and "Chart" in Altair" – a tidy data set.
  - Next, you define a set of aesthetic mappings – x, y, and color – that explain how the data will map to our *geoms* (i.e., the visual marks that do the hard work of conveying information to the reader).
  - Once you actually invoke said *geom* ("*geom_line*" in ggplot and "*mark_line*" in Altair), the data and aesthetic mappings are transformed into visual ticks that a human can understand – and thus, an angel gets its wings.
-

Comments and thoughts

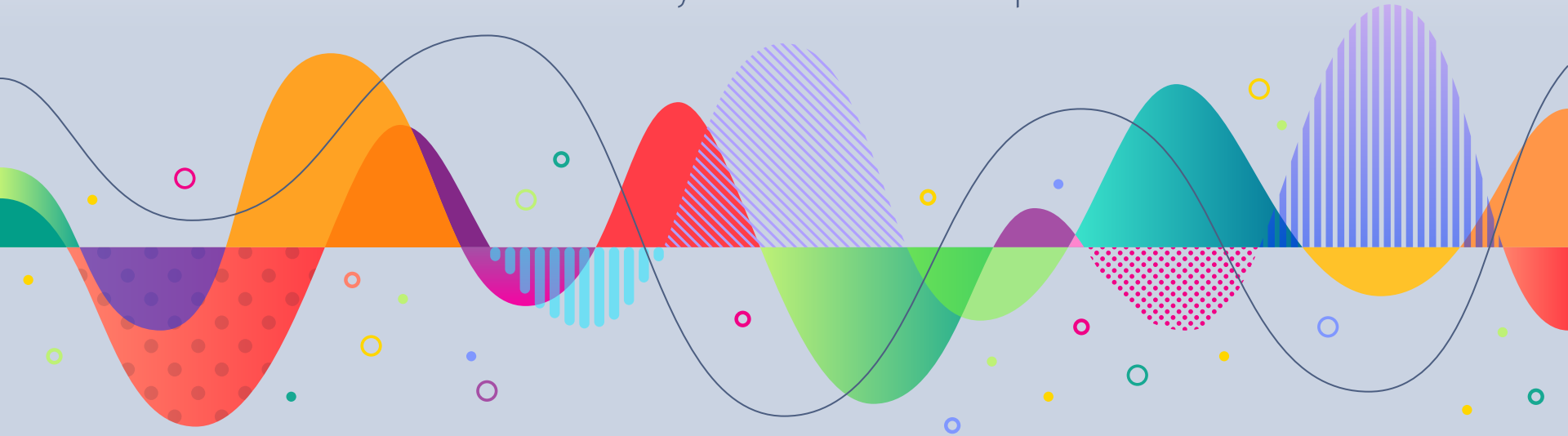
- Intellectually, you should also view Seaborn's FacetGrid through the same lens. FacetGrid needs a hue argument upfront – alongside your data – but wants the x and y arguments later.

- At that point, your mapping isn't an aesthetic one, but a functional one: for each "hue" in your data set, you're simply calling matplotlib's plot function using "dt" and "value" as its x and y arguments. The for loop is simply hidden from you.

2.

LINES AND DOTS

How would you make a scatter plot?



Data Aside

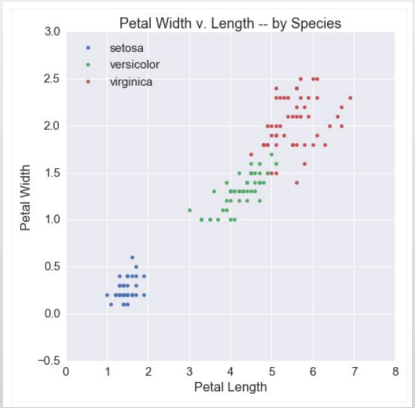
- We'll be dealing with the famous "iris" data set [though we refer to it as "df" in our code. It consists of four numeric columns corresponding to various measurements, and a categorical column corresponding to one of three species of iris.

	petalLength	petalWidth	sepalLength	sepalWidth	species
0	1.4	0.2	5.1	3.5	setosa
1	1.4	0.2	4.9	3.0	setosa
2	1.3	0.2	4.7	3.2	setosa
3	1.5	0.2	4.6	3.1	setosa
4	1.4	0.2	5.0	3.6	setosa

matplotlib

- Just remember to set the color argument explicitly or else the dots will all be blue…

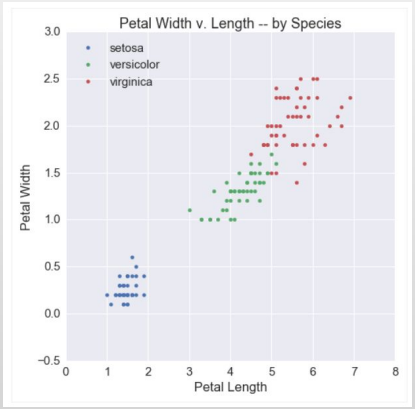
```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1, figsize=(7.5, 7.5))

for i, s in enumerate(df.species.unique()):
    tmp = df[df.species == s]
    ax.scatter(tmp.petalLength, tmp.petalWidth,
               label=s, color=cp[i])

ax.set(xlabel='Petal Length',
       ylabel='Petal Width',
       title='Petal Width v. Length -- by Species')

ax.legend(loc=2)
```



Matplotlib, method 1

- Just remember to set the color argument explicitly or else the dots will all be blue…

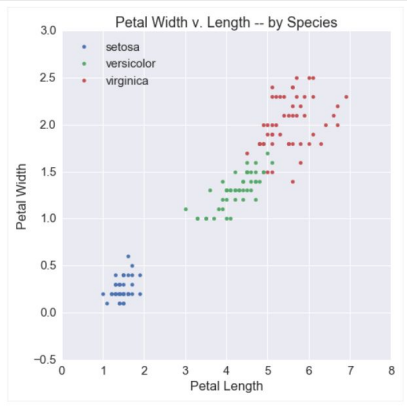
```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1, figsize=(7.5, 7.5))

for i, s in enumerate(df.species.unique()):
    tmp = df[df.species == s]
    ax.scatter(tmp.petalLength, tmp.petalWidth,
               label=s, color=cp[i])

ax.set(xlabel='Petal Length',
       ylabel='Petal Width',
       title='Petal Width v. Length -- by Species')

ax.legend(loc=2)
```



Matplotlib, method 2

- Here, I define a function named “scatter.” It will take groups from a pandas groupby object and plot petal length on the x-axis and petal width on the y-axis. Once per group! Powerful!

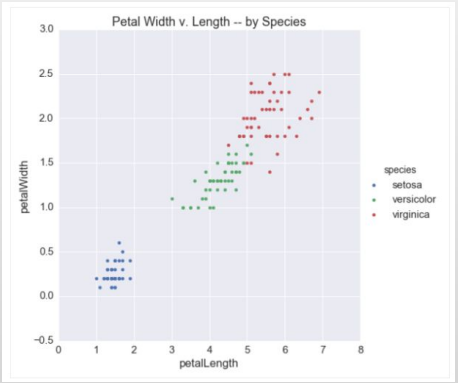
```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1, figsize=(7.5, 7.5))

def scatter(group):
    plt.plot(group['petalLength'],
             group['petalWidth'],
             'o', label=group.name)

df.groupby('species').apply(scatter)

ax.set(xlabel='Petal Length',
       ylabel='Petal Width',
       title='Petal Width v. Length -- by Species')

ax.legend(loc=2)
```



seaborn

- Anyway, in my mind, this problem is the same as the last one. Build another FacetGrid but borrow plt.scatter rather than plt.plot.

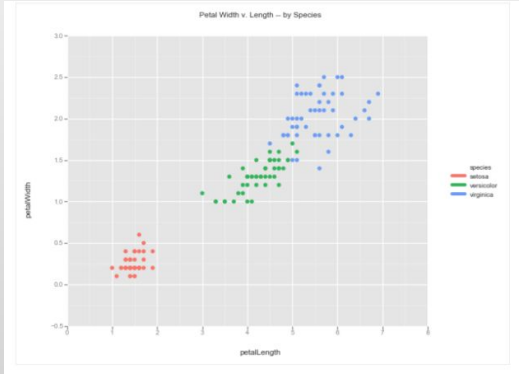
```
# SEABORN
g = sns.FacetGrid(df, hue='species', size=7.5)
g.map(plt.scatter, 'petalLength', 'petalWidth').add_legend()
g.ax.set_title('Petal Width v. Length -- by Species')
```



ggplot

- You just gotta swap out geom_line for geom_point!

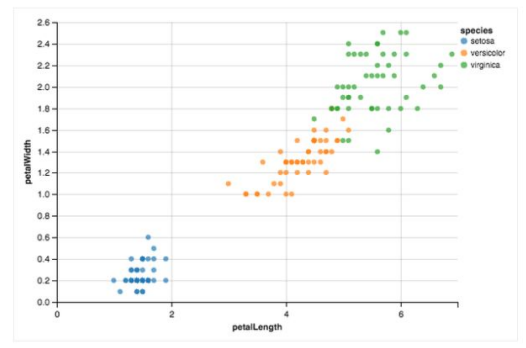
```
# GGPLOT
g = ggplot(df, aes(x='petalLength',
                   y='petalWidth',
                   color='species')) + \
        geom_point(size=40.0) + \
        ggtitle('Petal Width v. Length -- by Species')
g
```



ggplot

- just swap our mark_line for mark_point

```
# ALTAIR
c = Chart(df).mark_point(filled=True).encode(
    x='petalLength',
    y='petalWidth',
    color='species'
)
c
```



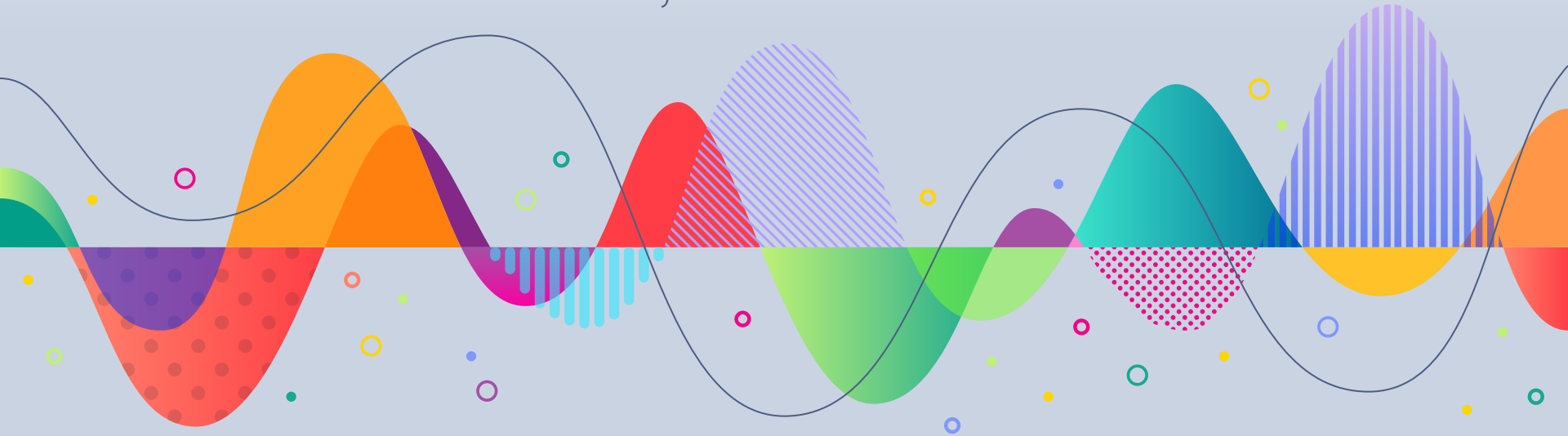
Comments and thoughts

- Here, the potential complications that emerge from building up the API from your data become clearer. While the pandas pivoting trick was extremely convenient for time series, it doesn't translate so well to this case.
- To be fair, the "group by" method is somewhat generalizable, and the "for loop" method is very generalizable; however, they require more custom logic, and custom logic requires custom work: *you would need to reinvent a wheel that Seaborn has kindly provided for you*.
- Conversely, *Seaborn, ggplot, and Altair all realize that scatter plots are in many ways line plots without the assumptions* (however innocuous those assumptions may be). As such, our code from Scene 1 can largely be reused, but with a new geom (geom_point/mark_point in the case of ggplot/Altair) or a new method (plt.scatter in the case of Seaborn). At this junction, none of these options seems to emerge as particularly more convenient than the other.

2.

DISTRIBUTIONS AND BARS

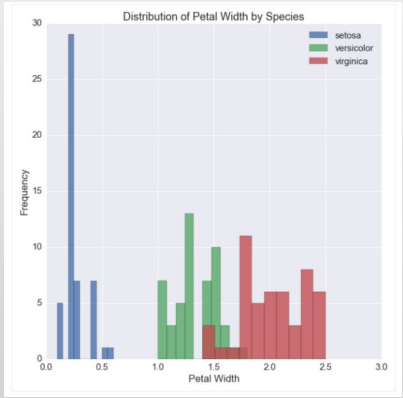
How would you visualize distributions?



matplotlib

- if we wanted a histogram - I have a method for that, which you can produce using either the for loop or group by methods from before.

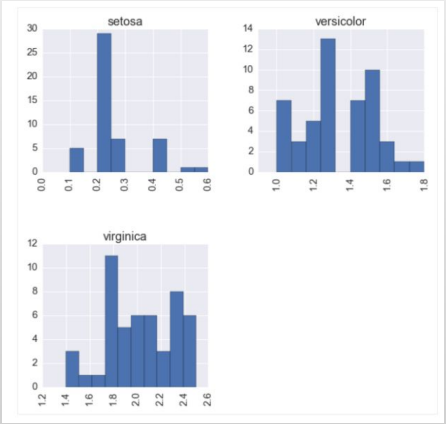
```
# MATPLOTLIB
fig, ax = plt.subplots(1, 1, figsize=(10, 10))

for i, s in enumerate(df.species.unique()):
    tmp = df[df.species == s]
    ax.hist(tmp.petalWidth, label=s, alpha=.8)

ax.set(xlabel='Petal Width',
       ylabel='Frequency',
       title='Distribution of Petal Width by Species')

ax.legend(loc=1)
```



pandas

- You only need two things: (A) The column name by which you'd like to stratify; and (B) The column name for which you'd like distributions. These go to the "by" and "column" parameters, respectively, resulting in instant plots!

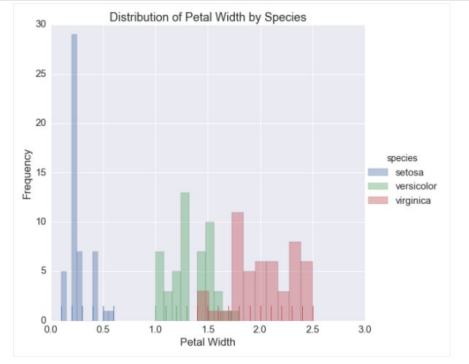
```
# PANDAS
fig, ax = plt.subplots(1, 1, figsize=(10, 10))

df.hist(column='petalWidth', by='species', grid=None, ax=ax)
```



seaborn

- Here we have a special distribution method named “distplot” that goes beyond histograms (*looks at pandas haughtily*). You can use it for histograms, KDEs, and rugplots – even plotting them simultaneously. For example, by combining this method with FacetGrid, I can produce a histo-rugplot for every species of iris:

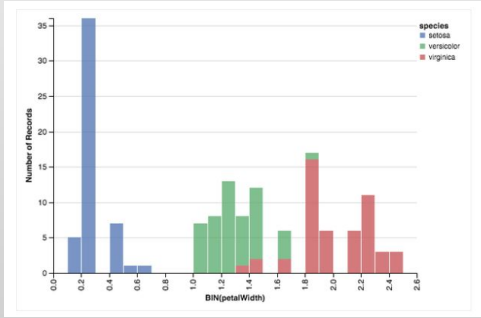
```
# SEABORN
g = sns.FacetGrid(df, hue='species', size=7.5)

g.map(sns.distplot, 'petalWidth', bins=10,
      kde=False, rug=True).add_legend()

g.set(xlabel='Petal Width',
      ylabel='Frequency',
      title='Distribution of Petal Width by Species')
```



altair

- The code may look weird at first glance, but don't be alarmed. All we're saying here is: "Hey, histograms are effectively bar charts." Their x-axes correspond to bins, which we can define with my Bin class; meanwhile, their y-axes correspond to the number of items in the data set which fall into those bins, which we can explain using a SQL-esque "count(*)" as our argument for y.

```
# ALTAIR
c = Chart(df).mark_bar(opacity=.75).encode(
    x=X('petalWidth', bin=Bin(maxbins=30)),
    y='count(*)',
    color=Color('species', scale=Scale(range=cp.as_hex()))
)
c
```



Comments and thoughts

- Most people actually find pandas' convenience functions very convenient; however, I'll admit that there's some cognitive overhead in remembering that pandas has implemented a "by" parameter for boxplots and histograms but not for lines.
- Users actually came to Seaborn from matplotlib/pandas for its rich set of "proprietary" visualization functions (e.g., distplot, violin plots, regression plots, etc.). You will also learn to love FacetGrid which is Seaborn's killer app. I need to note: Seaborn implements a number of awesome visualizations that lesser libraries ignore.
- On the ggplot now, using mostly the same code (and more importantly, mostly the same thought process), we create a wildly different graph. We do this not by calling an entirely separate function, but by changing how our aesthetic mappings get presented to the viewer, i.e., by swapping out one geom for another.
- Similarly, Altair's API remains remarkably consistent. Even for what feels like a different operation, Altair's API is simple, elegant, and expressive.

THANKS!

Any questions?