Visualization

Matplotlib++

Dr. Scott Gorlin

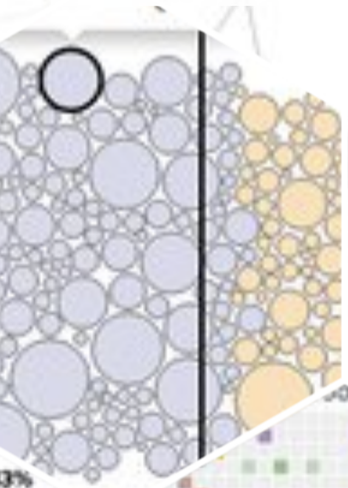
Harvard University

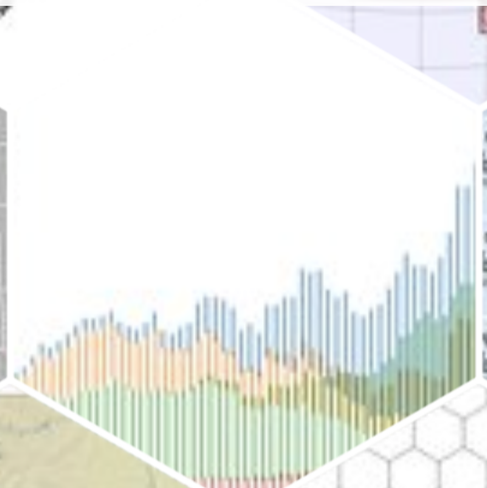
Fall 2018

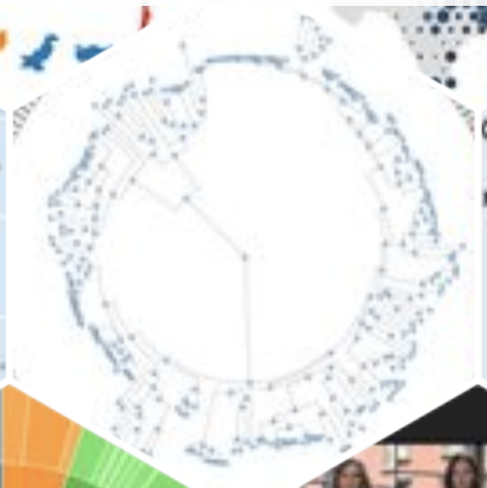
Agenda

- Data Viz
- Declarative Grammars
- Javascript and HTML5
- PyViz - The New Breed
- Colormaps
- Data Shading
- Where We Are
- Readings

Data Viz

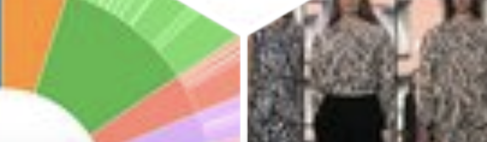











Data Scientists Wear Two Hats

Capturing value

- Top-notch algorithms
- Efficient development
- Realized lift

Demonstrating Value

- Intuitive charts
- Inspiring trust
- Non-technical stakeholders

Visualization: Working Modes

Exploration

- Expert use
- Write any code
- Rerun *ad naseum*

Pandas, MPL+, REPL, Jupyter

Interactive Deploy

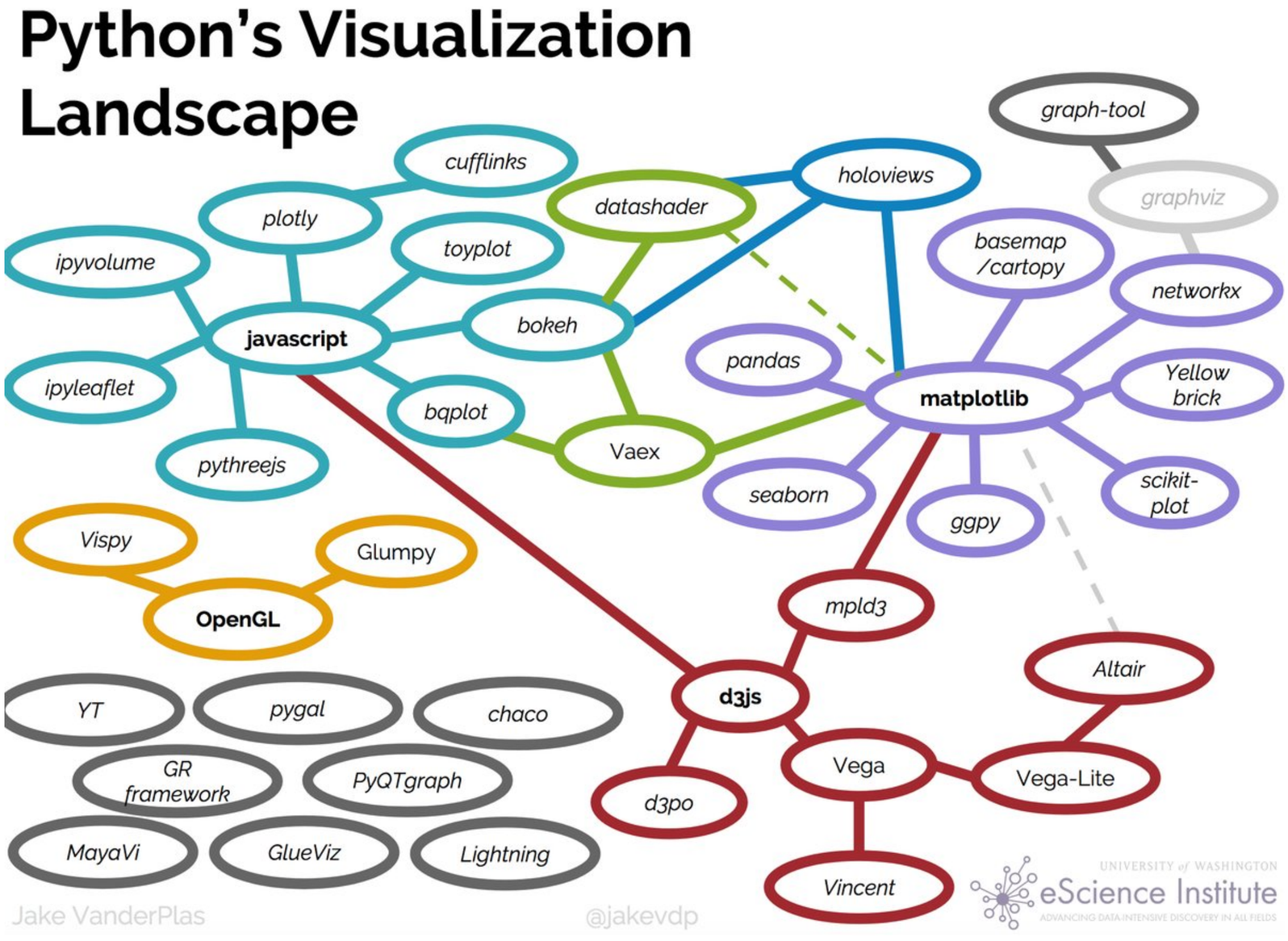
- Point-and-click
- Live data updates
- Web pages

Not handled well by *pure* python

Static Deploys

- Publications
- Slides
- Pictures

MPL+ FTW

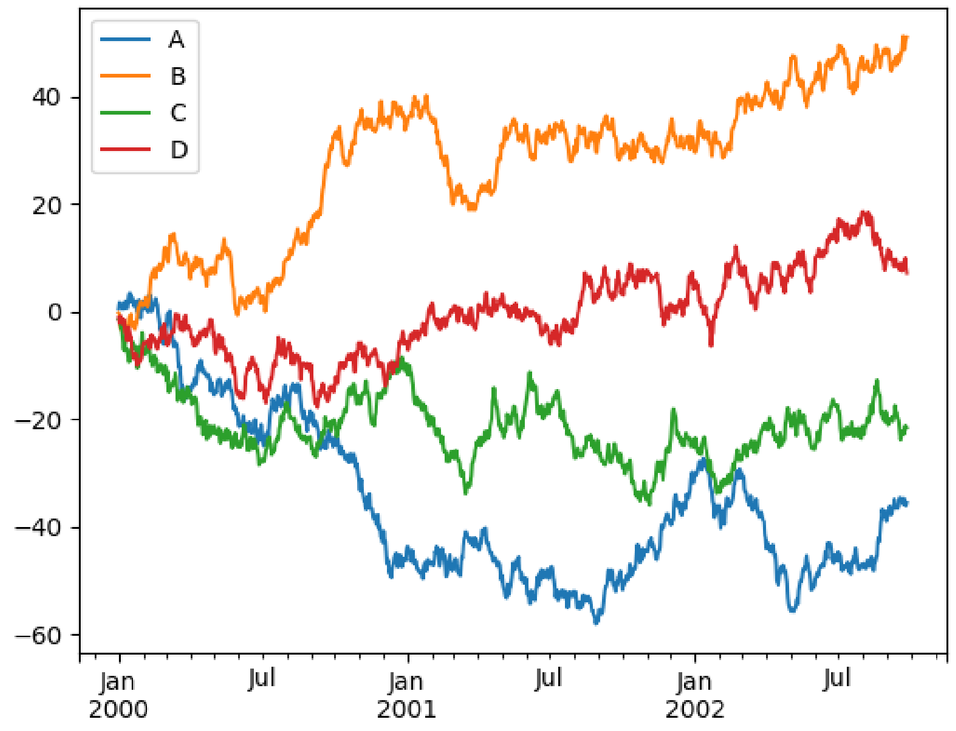


Source, Original

One Lib to Rule Them All

One Lib to Rule Them All

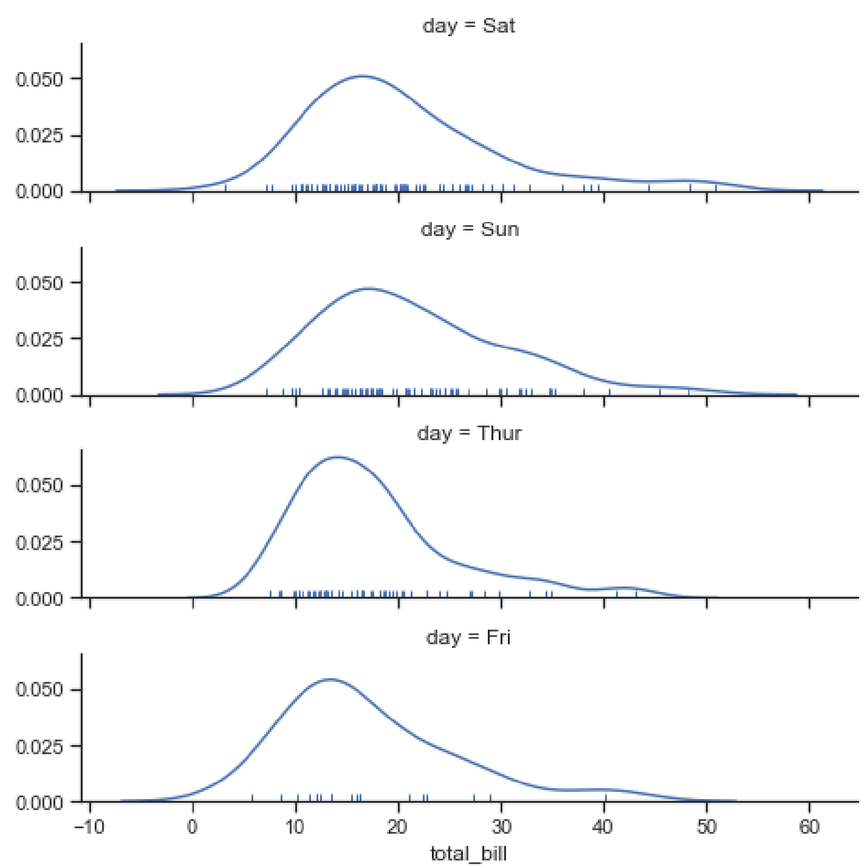
```
df = pd.DataFrame(
    np.random.randn(1000, 4),
    index=pd.date_range(
        '1/1/2000',
        periods=1000),
    columns=list('ABCD')
).cumsum()
df.plot()
```



Pandas offers hierarchical date plots, collation, and reasonable styles

One Lib to Rule Them All

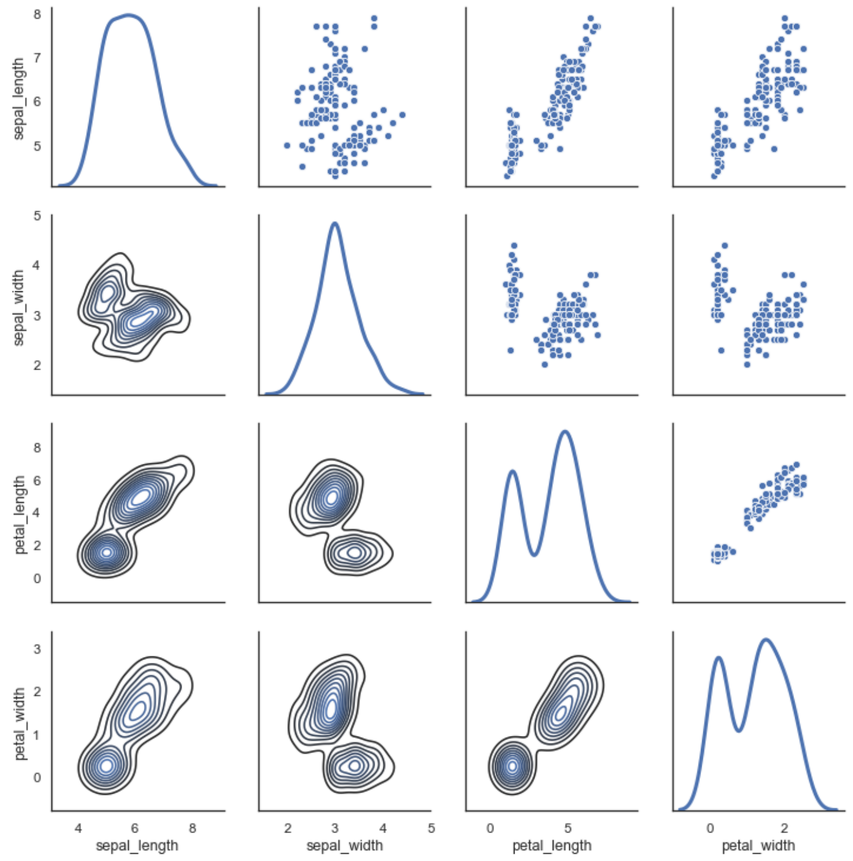
```
import seaborn as sns

ordered_days = [...]
g = sns.FacetGrid(
    tips, row="day",
    row_order=ordered_days)
g.map(sns.distplot, "total_bill",
    hist=False, rug=True);
```



Seaborn offers better aesthetics, a few new plot types, and ***faceting***

One Lib to Rule Them All

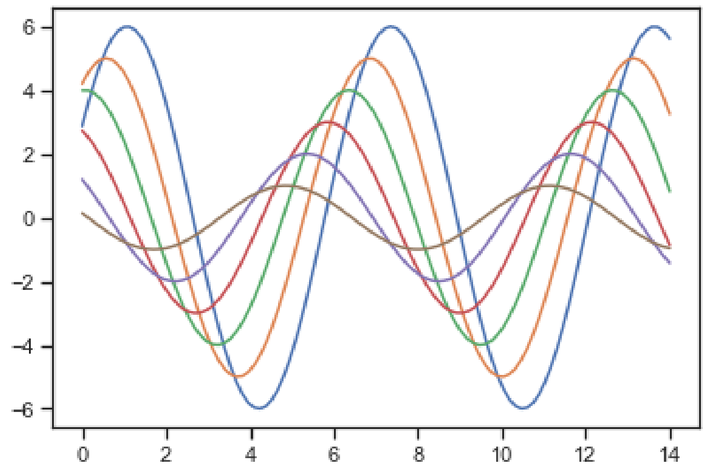
```
df = sns.load_dataset("iris")

g = sns.PairGrid(df, diag_sharey=False)
g.map_lower(sns.kdeplot)
g.map_upper(sns.scatterplot)
g.map_diag(sns.kdeplot, lw=3)
```



… density plots …

One Lib to Rule Them All

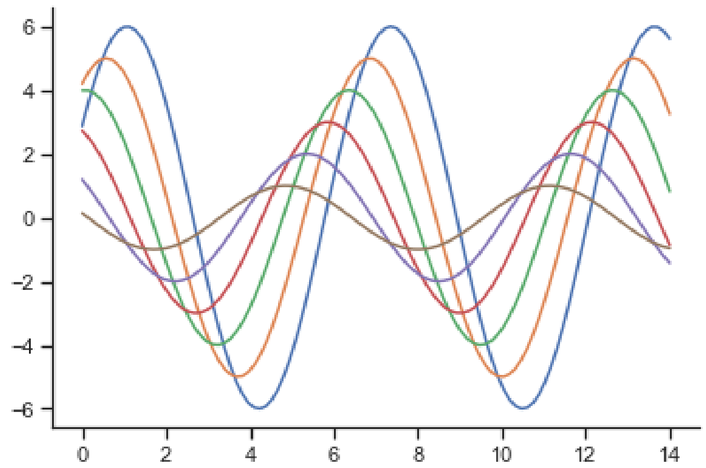
```python
def sinplot(flip=1):
    x = np.linspace(0, 14, 100)
    for i in range(1, 7):
        plt.plot(
            x,
            np.sin(x + i * .5) * (7 - i) * flip
        )

sinplot()
# sns.despine()
```



… plus an easy win…

One Lib to Rule Them All

```
def sinplot(flip=1):
    x = np.linspace(0, 14, 100)
    for i in range(1, 7):
        plt.plot(
            x,
            np.sin(x + i * .5) * (7 - i) * flip
        )

sinplot()
sns.despine()
```



… to improve ink ratio!

MPL Backend

MPL has become an ***imperative backend*** for many python viz libraries

Imperative

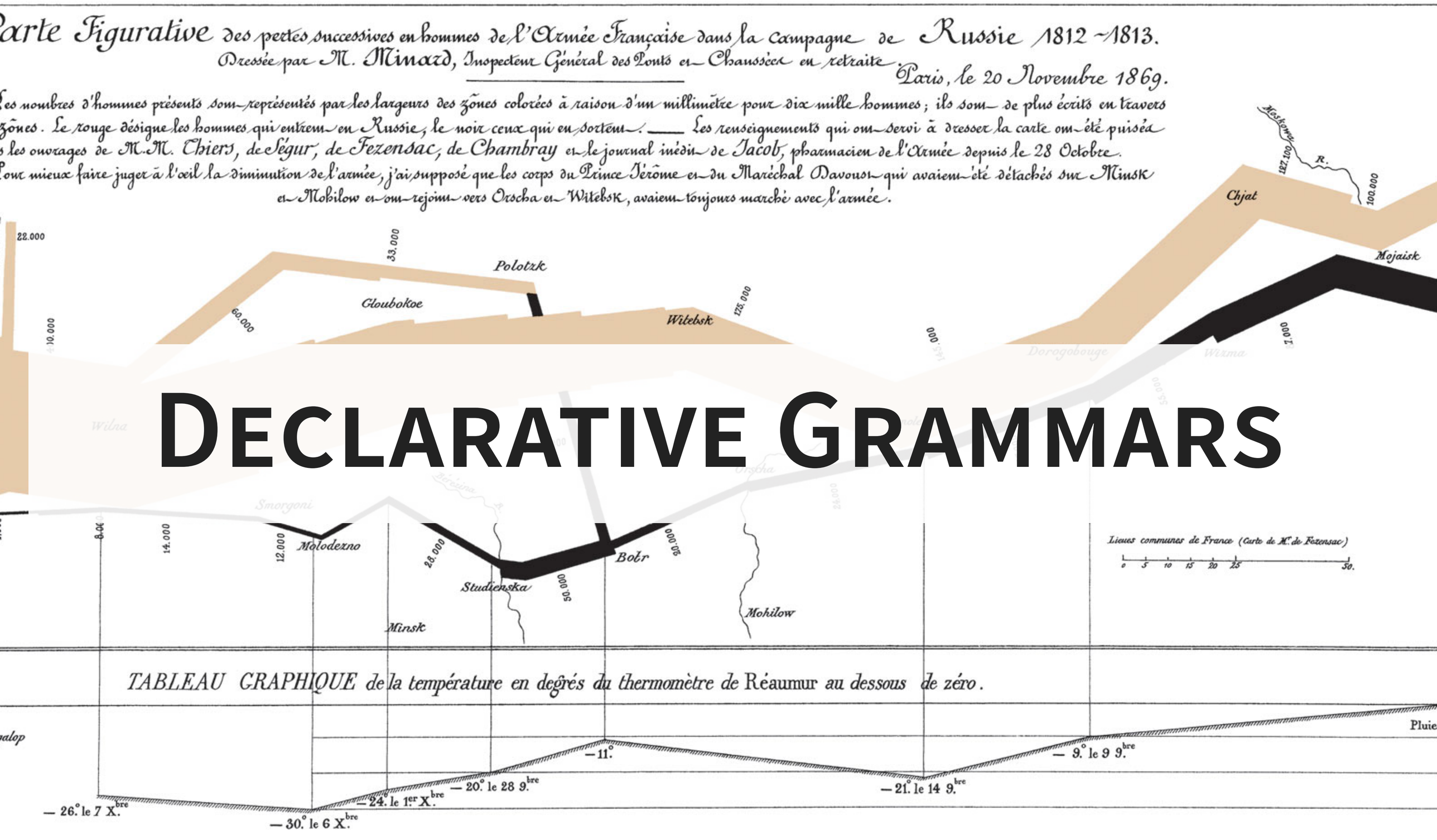
- ***How*** to graph
- Ultimate control
- Great for polish, publish

Declarative

- ***What*** the graph should say
- Faster to implement
- Easier to read (code)

But, small issues in viz are
disproportionately bad

Declarative Grammars



Grammar of Graphics

A grammar of graphics is a tool that enables us to concisely describe the components of a graphic. Such a grammar allows us to move beyond named graphics (e.g., the "scatterplot") and gain insight into the deep structure that underlies statistical graphics

— Wickham, A Layered Grammar of Graphics

Grammar of Graphics

To be precise, the layered grammar defines the components of a plot as:

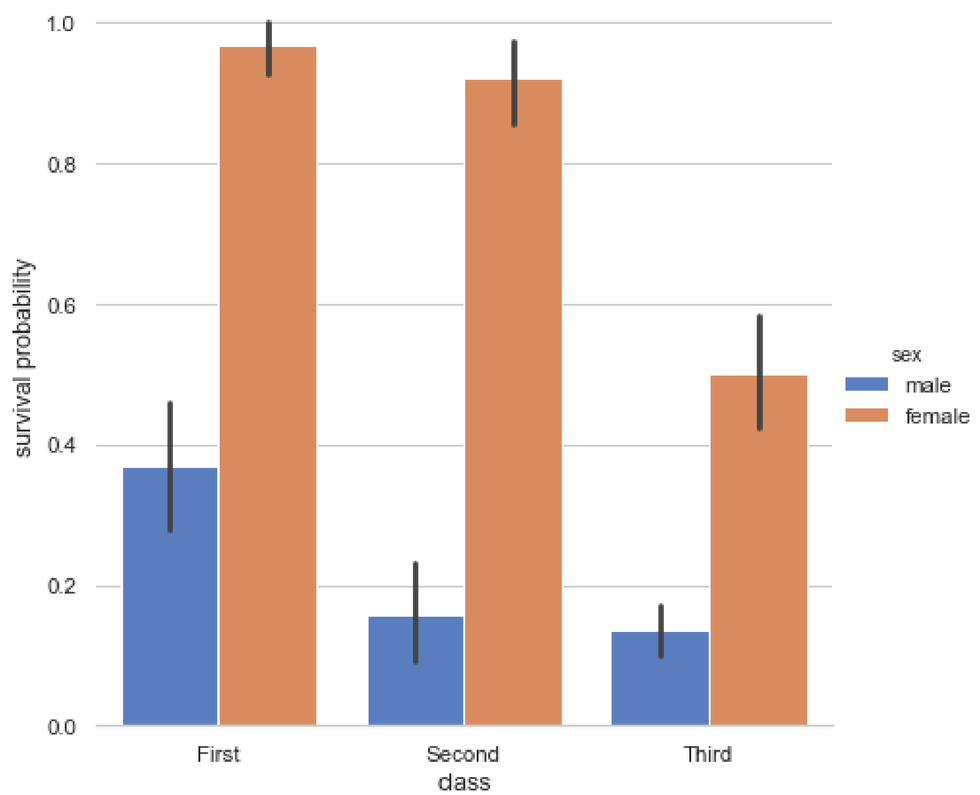
- *a default dataset and set of mappings from variables to aesthetics,*
- *[layers of] 1 geometric object, 1 statistical transformation, 1 position adjustment, and optionally, 1 dataset and set of aesthetic mappings,*
- *one scale for each aesthetic mapping used,*
- *a coordinate system,*
- *the facet specification.*

— Wickham, A Layered Grammar of Graphics

Seaborn - Pidgin (?) of Graphics

```
# Load the example Titanic dataset
titanic = sns.load_dataset("titanic")

# Draw a nested P(survive | class, sex)
g = sns.catplot(
    x="class", y="survived",
    hue="sex", data=titanic,
    kind="bar")
```

We pass a ***DataFrame*** and configured relationship between data (columns) and the graphics (x, y, etc).

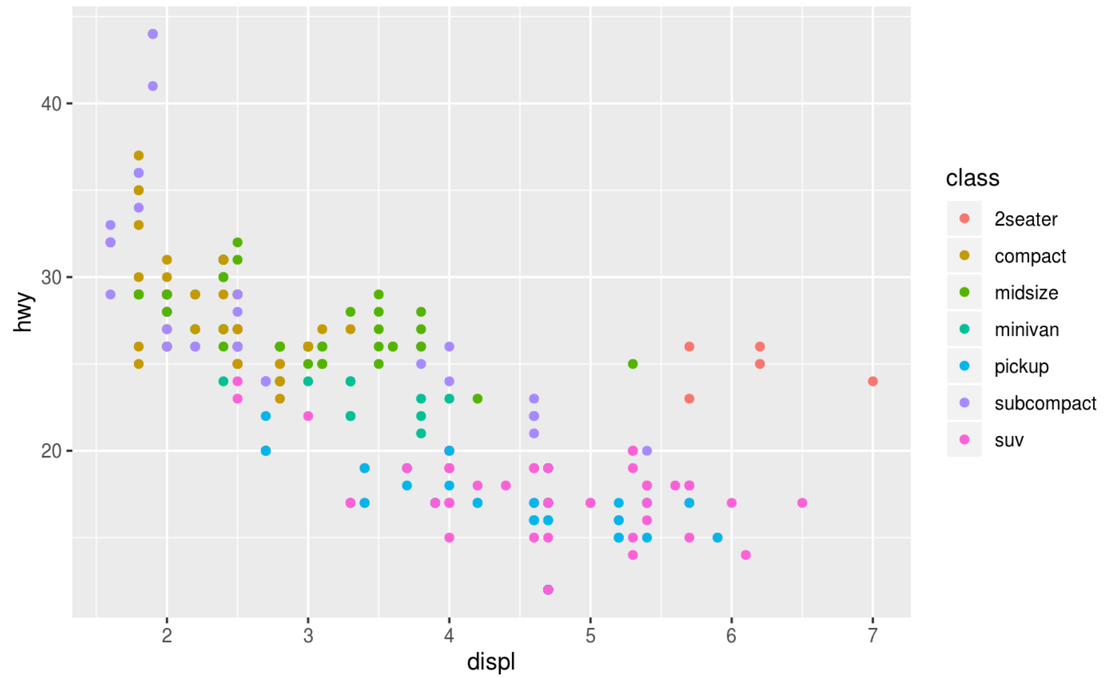


GGPLOT2

The gold standard of grammars

R code:

```
library(ggplot2)

ggplot(
    mpg,
    aes(displ, hwy,
        colour = class
)) + geom_point()
```

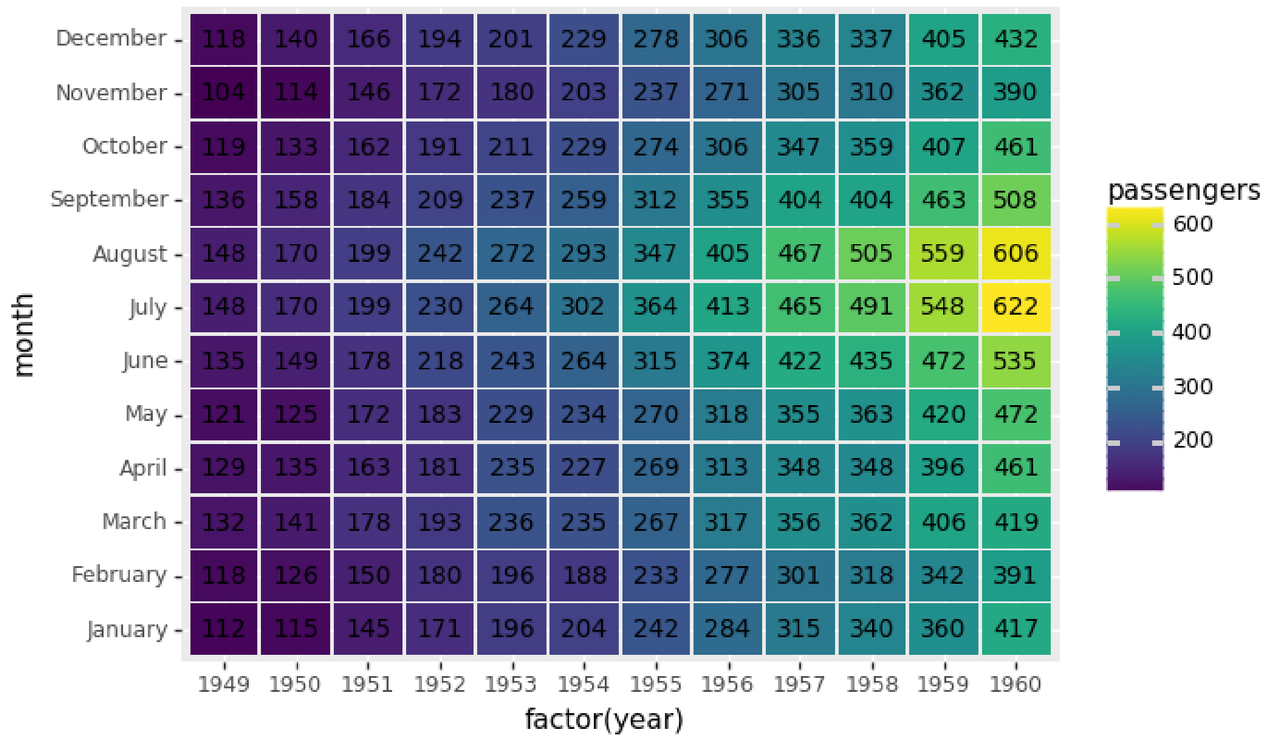


But, sadly, the python port appears dead :(

PLOTNINE

Another promising port, abeit very new

```
(ggplot(
    flights, aes(
        'factor(year)', 'month',
        fill='passengers'))
 + geom_tile(aes(
    width=.95, height=.95))
 + geom_text(aes(
    label='passengers'), size=10)
)
```



JAVASCRIPT AND HTML5

Why?

Interactivity!

- Cross filtering
- Tooltips
- Dimension switching
- Styling (eg, alpha, cmap, for exploration)

Deployability!

- Customer facing
- Scalability
  - Client side rendering
  - Server side data processing
- Live, deployed

HTML Deploys

Wait! This is a python course!

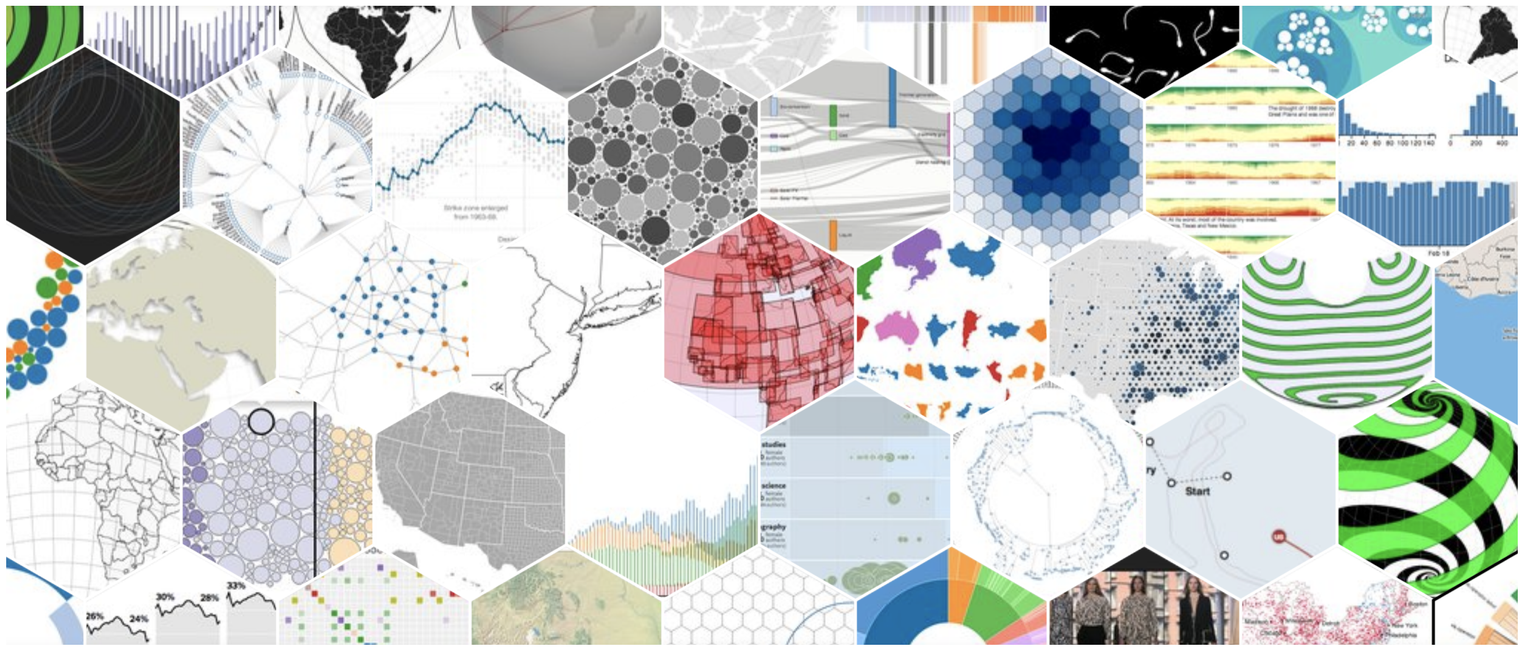
Just like you must know SQL, scala, git, C… a small amount of JS, HTML, and CSS will greatly amplify your productivity and impact

HTML Deploys

```
<html>
  <head>
    <script src="https://vega.github.io/vega/vega.min.js"></script>
  </head>
  <body>
      <div id="vega-chart" class="vega-lite"></div>
      <script type="text/javascript">
        vegaEmbed("#vega-chart", {...});
      </script>
  </body>
</html>
```

D3

Data Driven Documents



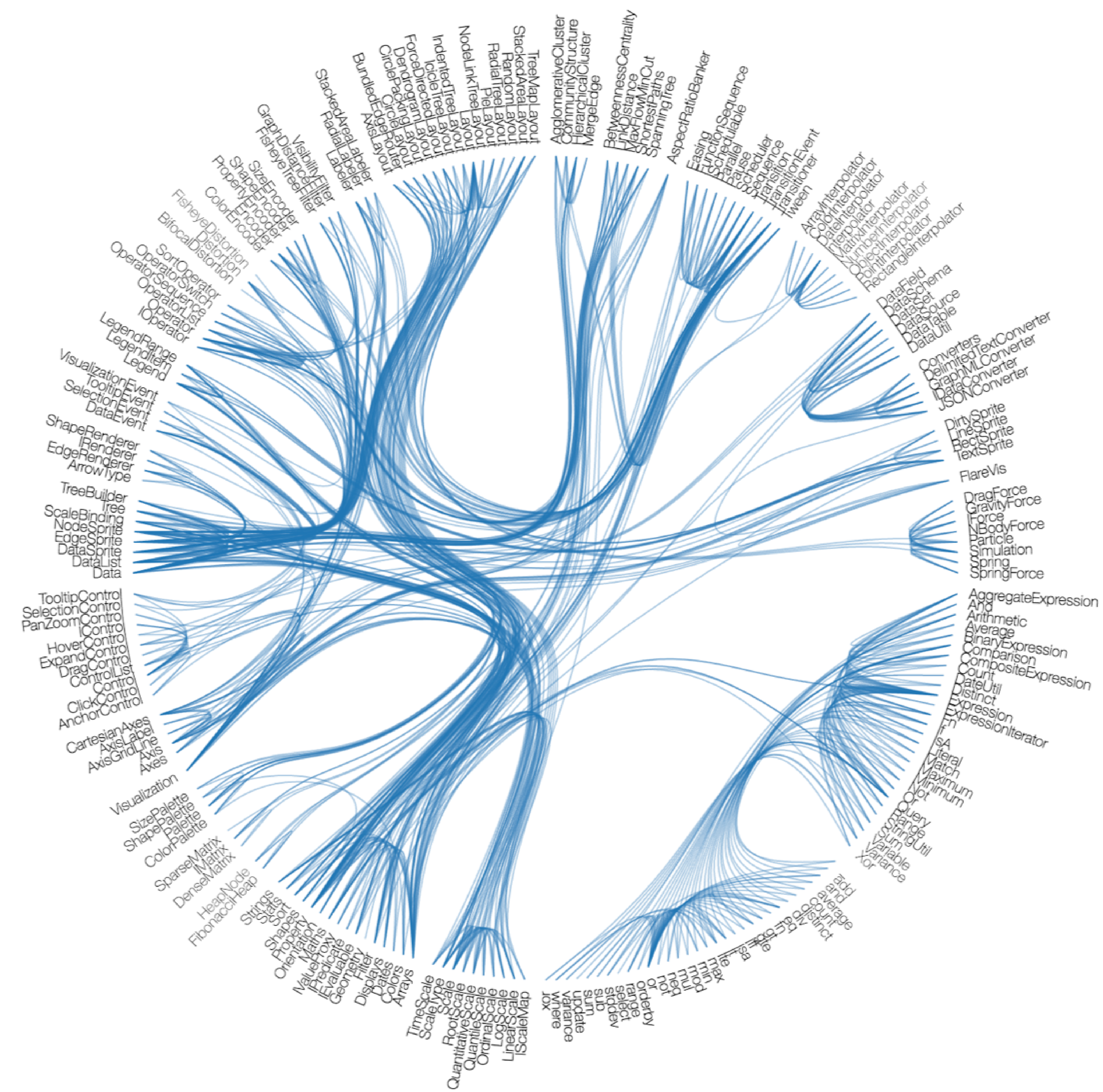
D3 allows you to bind arbitrary data to a Document Object Model (DOM), and then apply data-driven transformations to the document.

D3

D3 is the gold standard backend for charting and visualization

It excels for high-touch, custom, interactive visualization

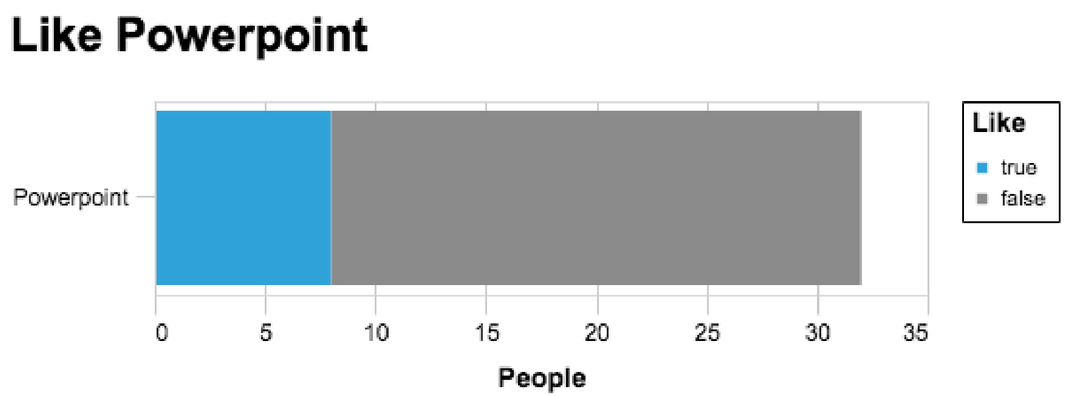
However, it is hard to perfect for generic or repeatable plots, live data, etc



Vega-Lite

```
{
  "$schema": "v2.json",
  "data": data,
  "title": "Like Powerpoint",
  "mark": "bar",
  "encoding": encoding
}
```

```
encoding = {
  "x": {
    "field": "People", "aggregate": "sum",
    "type": "quantitative"},
  "y": {"field": "q", "type": "nominal"},
  "color": {
    "field": "Like", "type": "nominal",
    "legend":{
      "orient":"top-right",
      "direction":"vertical",
      "values":[true, false]},
    "scale": {
      "domain": [false, true],
      "range": ["#8b8b8b", "#30a2da"]
    }
  }
}
```



Vega and Vega-Lite expose a declarative grammar, configured via json, on top of D3

```
data = {
    "values": [
        {"q": "Powerpoint", "Like": false, "People": 24},
        {"q": "Powerpoint", "Like": true, "People": 8}
    ]
}
```

But Wait! There's More!

You don't need to deploy your own site to use D3!

mpld3

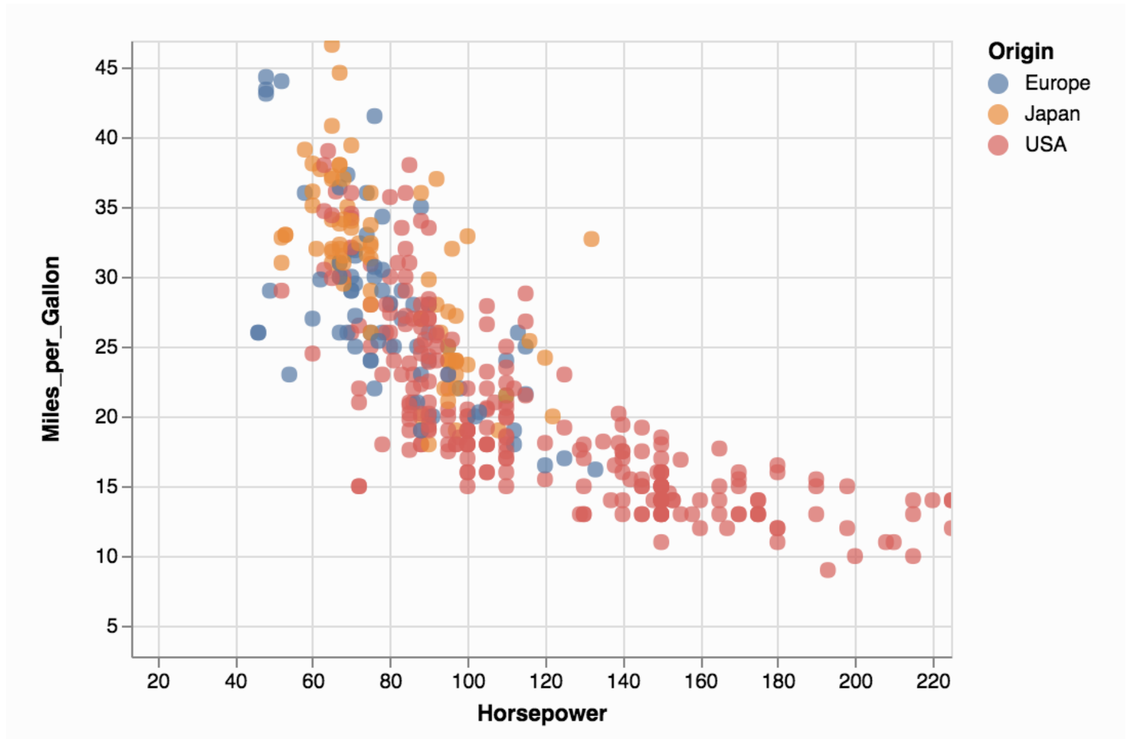
```
import matplotlib.pyplot as plt
import mpld3
plt.plot(range(5))
mpld3.show()
```

But Wait! There's More!

You don't need to deploy your own site to use D3!

Altair (Vega-Lite)

```
import altair as alt
from vega_datasets import data

cars = data.cars()

alt.Chart(cars).mark_circle(size=60).encode(
    x='Horsepower',
    y='Miles_per_Gallon',
    color='Origin',
).interactive()
```



Hybrids

These hybrid libraries are easy to use directly for interactivity

… or, use them in a Django view, rendering html but allowing the web browser to do the heavy rendering client side in JS, eg `mpld3.fig_to_html()`

MPL etc can actually directly render charts in a Django view, but it puts a lot of load on the server, and doesn't give you interactivity

PyViz - The New Breed

PyViz

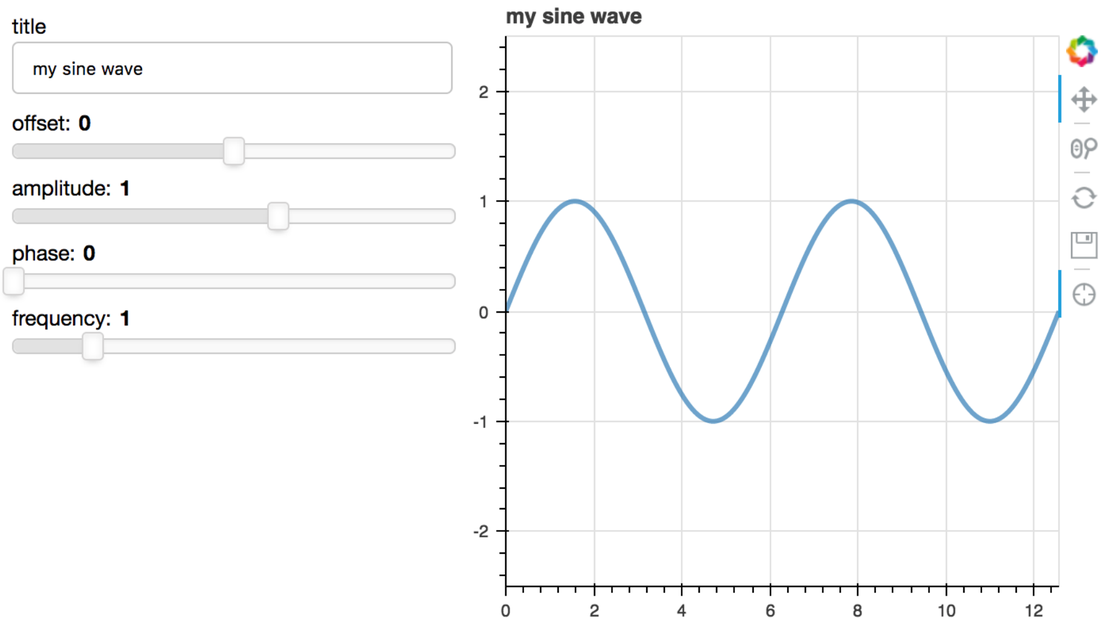
A coordinated effort for interactive plotting in browsers

Built on Bokeh

Easy charting/webserver

```
# myapp.py
N = 200
x = np.linspace(0, 4*np.pi, N)
y = np.sin(x)
source = ColumnDataSource(data=dict(x=x, y=y))

# Set up plot
plot = figure(...)
plot.line('x', 'y', source=source)

# Set up widgets
amplitude = Slider(title="amplitude")
```

```
def update_data(attrname, old, new):
    a = amplitude.value
    y = a*np.sin(k*x + w) + b
    source.data = dict(x=x, y=y)

# Set up layouts and add to document
inputs = widgetbox(amplitude)

curdoc().add_root(row(inputs, plot))
```

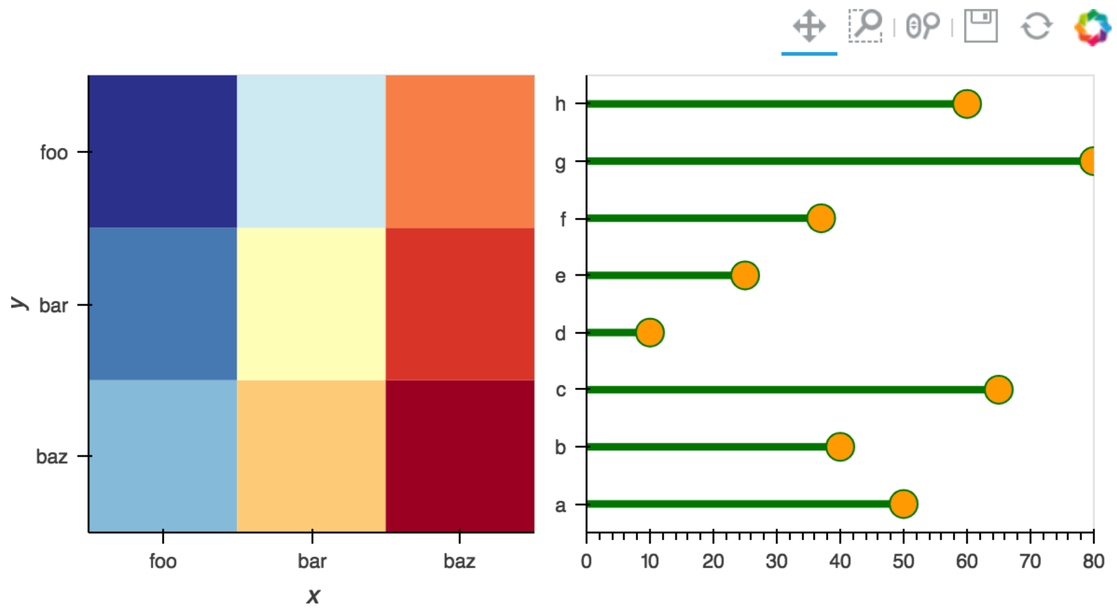



```
bokeh serve --show myapp.py
```

HoloViews, hvPlot, GeoViews

Higher-level, grammatical(ish) plots and widgets

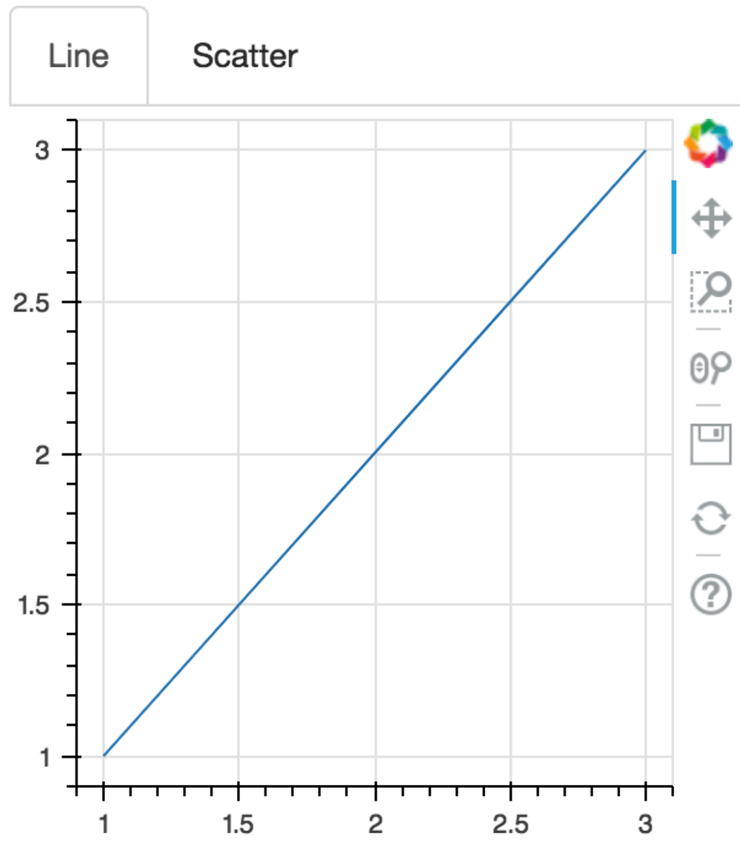
```
factors = ["a", "b", "c", ...]
x =  [50, 40, 65, ...]
scatter = hv.Scatter((factors, x))
spikes = hv.Spikes(scatter)

x = ["foo", "foo", "foo", ...]
y = ["foo", "bar", "baz", ...]
z = [0, 1, 2, 3, 4, 5, 6, 7, 8]
heatmap = hv.HeatMap((x, y, z))

heatmap + spikes() * scatter()
```



Panel for Apps

Layouts, dashboards, etc

```
from bokeh.plotting import figure
from panel.layout import Tabs

p1 = figure()
p2 = figure()

p1.line([1, 2, 3], [1, 2, 3])
p2.circle([1, 2, 3], [1, 2, 3])

tabs = Tabs(
    ('Line', p1), ('Scatter', p2),
)
tabs
```



. . . AND, OF COURSE, PLOTLY/DASH

Plotly is an increasingly popular graphing and dashboard solution

It is a hosted service; the python library is an api client!

Ensure you are not sending sensitive data to an outside service! You must set up offline mode for most enterprise work!

Colormaps

Color for Science

Color is the thing your customer thinks they know the most about.

They do not. Guard your colors with your (scientific) life!

Kinds of Color

Qualitative

- Categorical
- Distinct

Sequential

- Monotonic change
- Magnitude

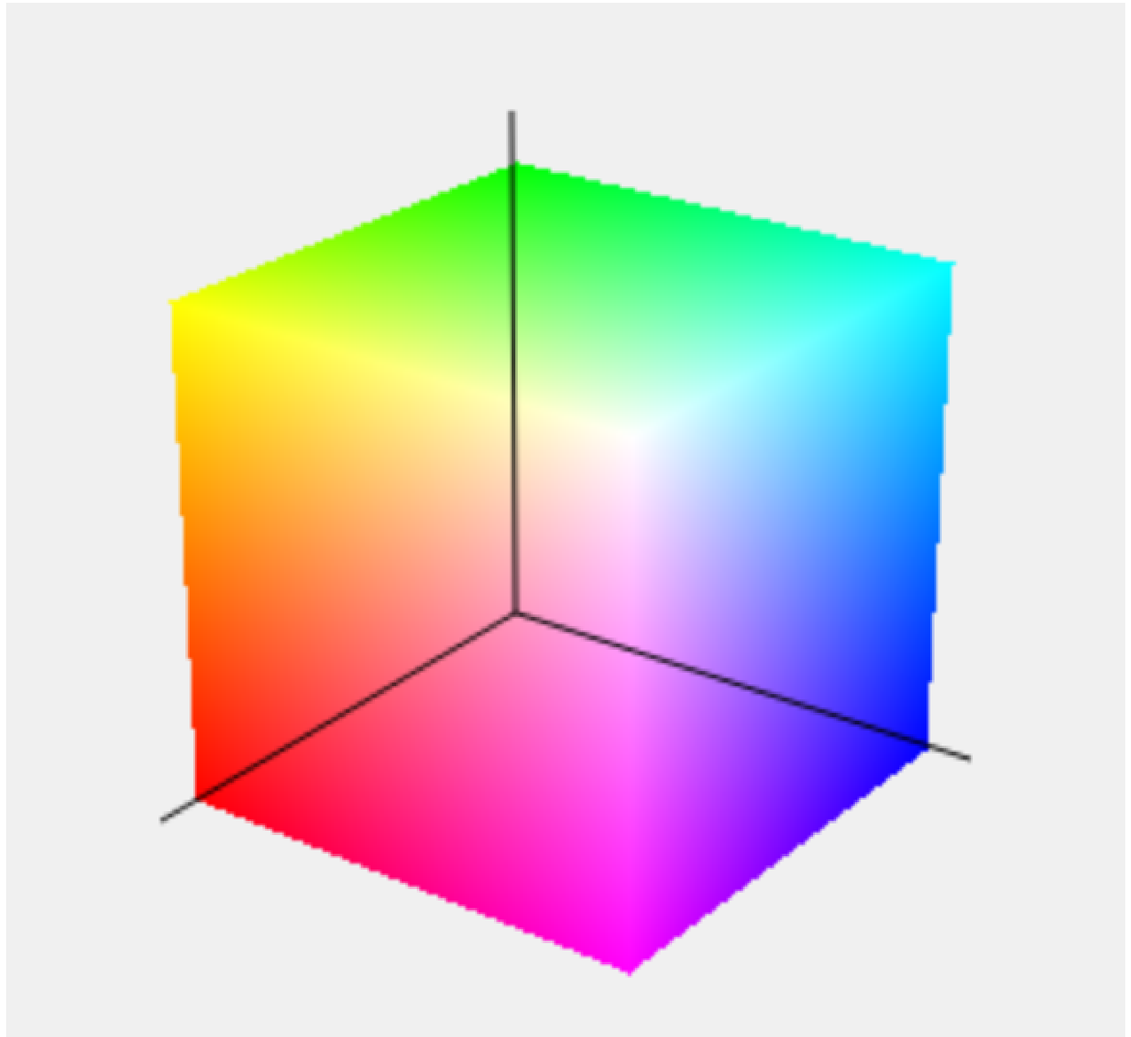
Divergent

- Good <-> Bad
- Depth <- 0 -> Altitude

Perceptual differences in colors should be ***semantically*** meaningful

Colorspace

Additive color is represented with the triplet (R, G, B) in terms of pixels

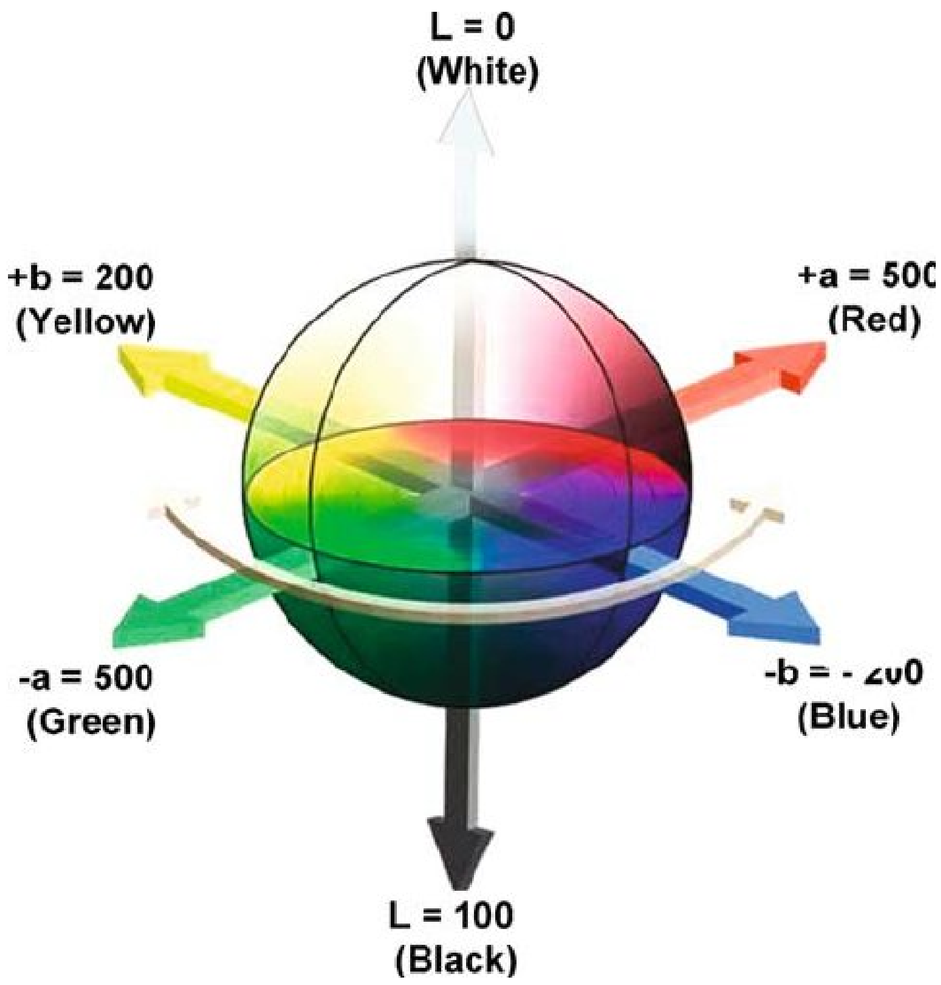


Colorspace

… which can be mapped into a perceptually-uniform Euclidean space

L^*ab is one such space

Source

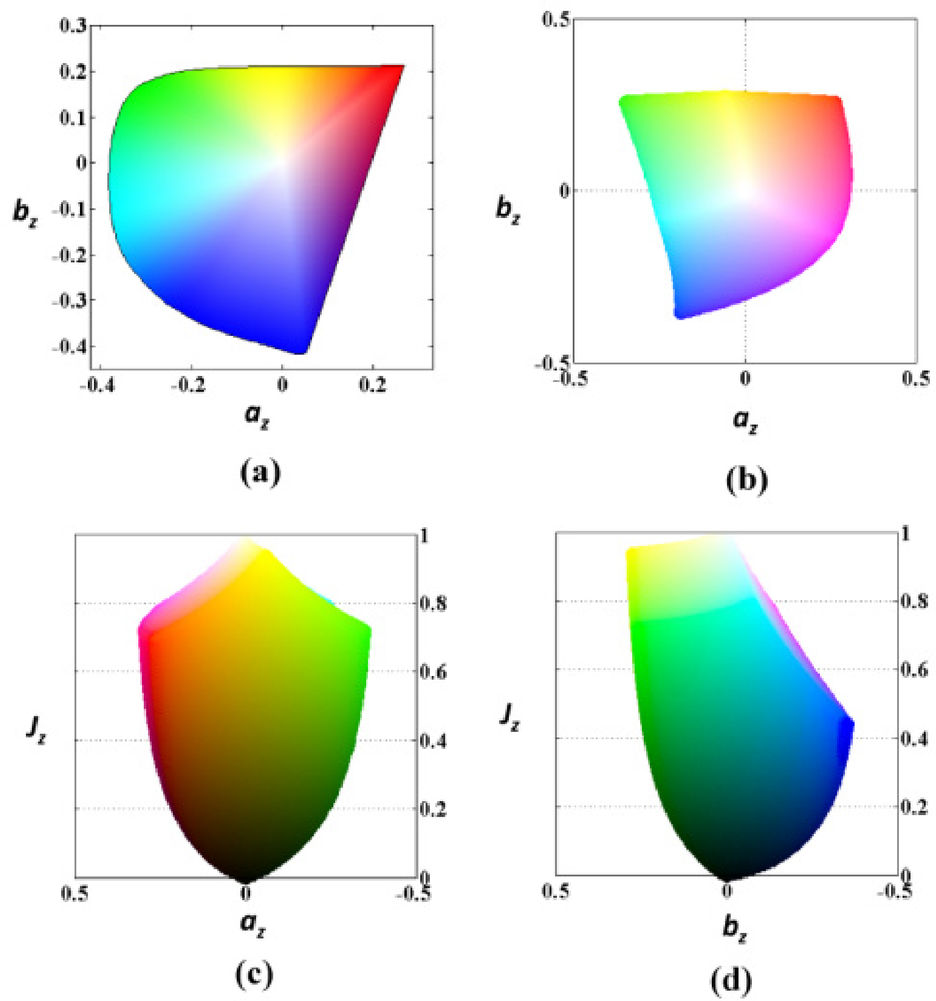


Colorspace

… which can be mapped into a perceptually-uniform Euclidean space

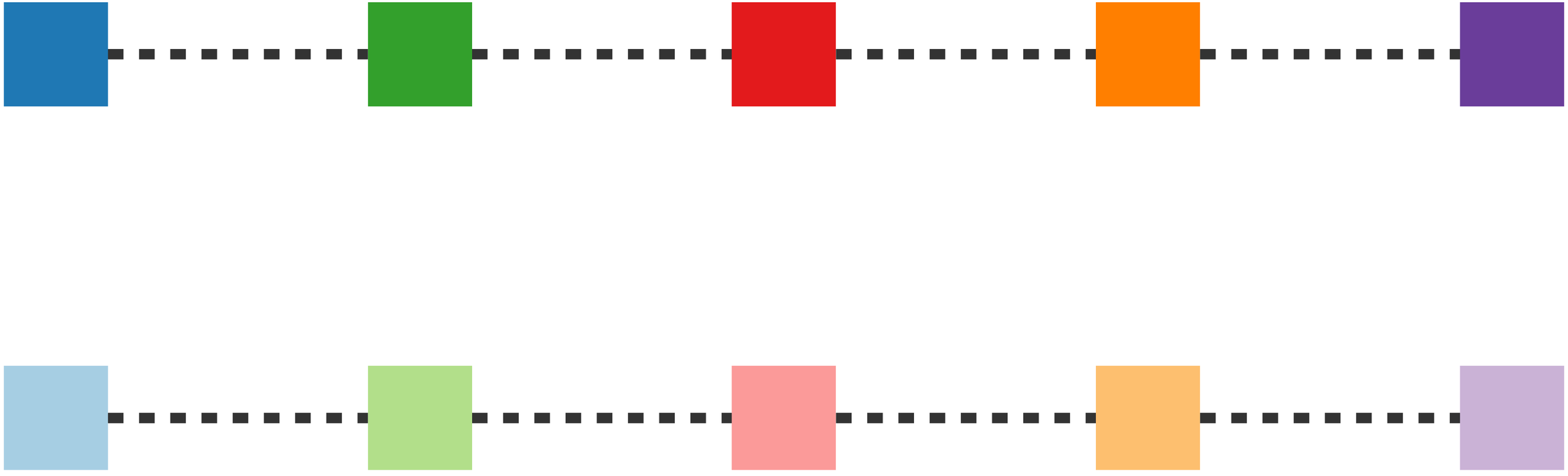
L^*ab is one such space

Source



Qualitative Color

This is easy. Pick a lightness/saturation, circle around hues uniformly



Qualitative Color

This is easy. Pick a lightness/saturation, circle around hues uniformly

```
sns.palplot(sns.color_palette("hls", 8))
```

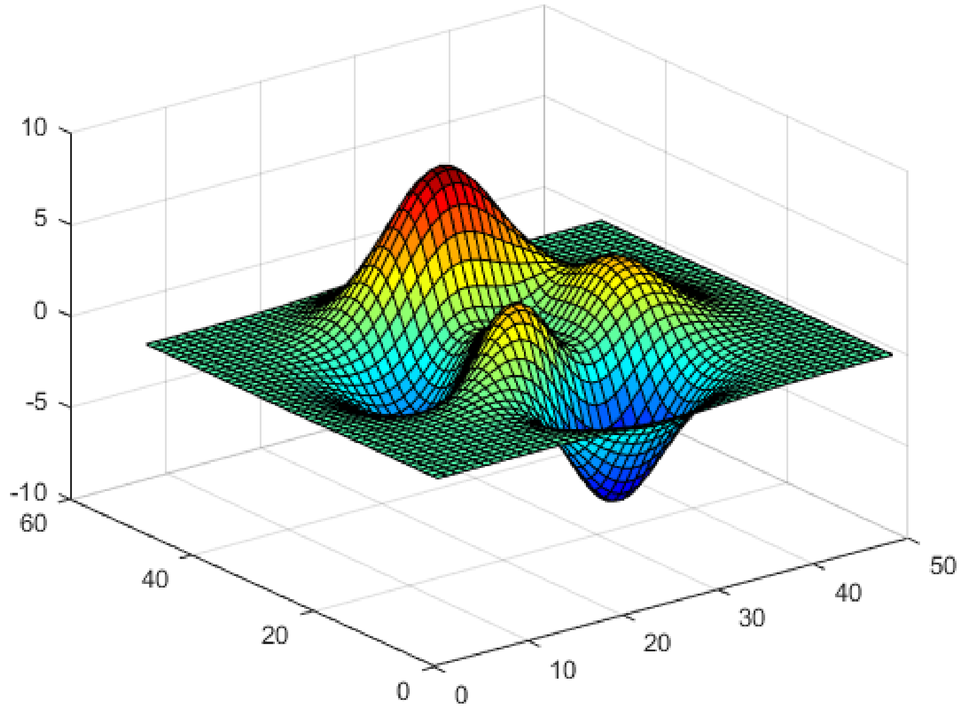
See also Color Brewer

Sequential Color

Pick a ‘high’ and ‘low’ color, plus a few bands

Matlab:

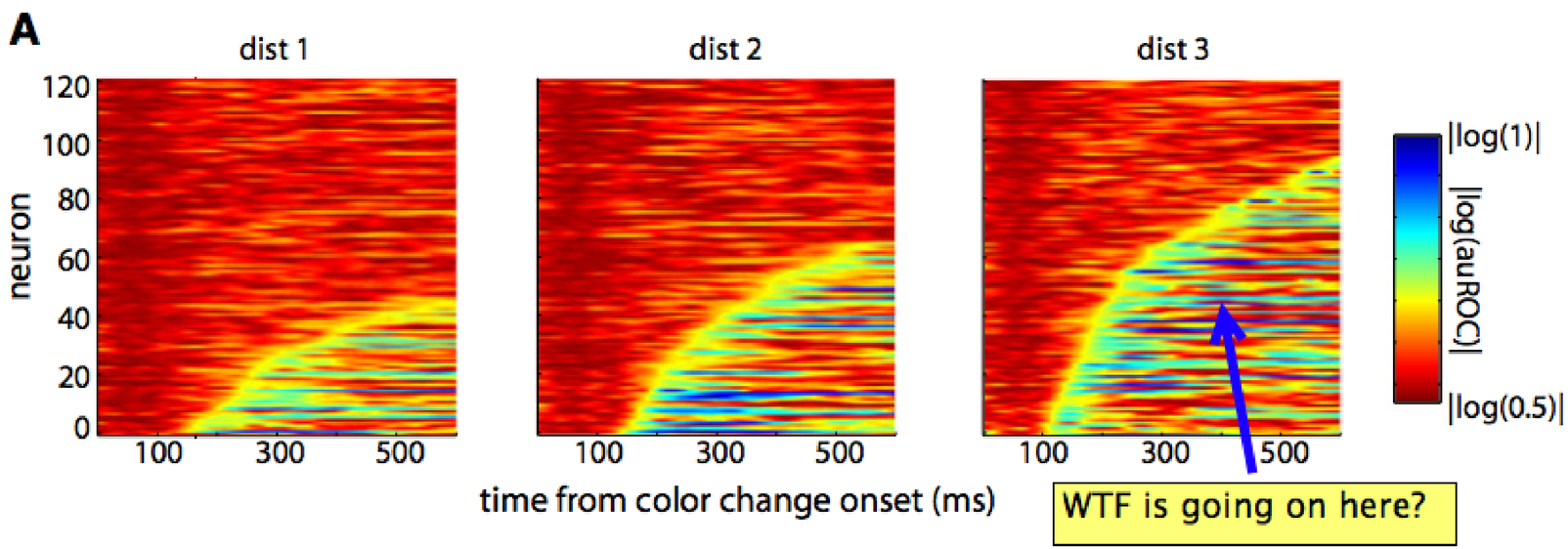
```
surf(peaks);
colormap('jet');
```

Sequential Color

NO!

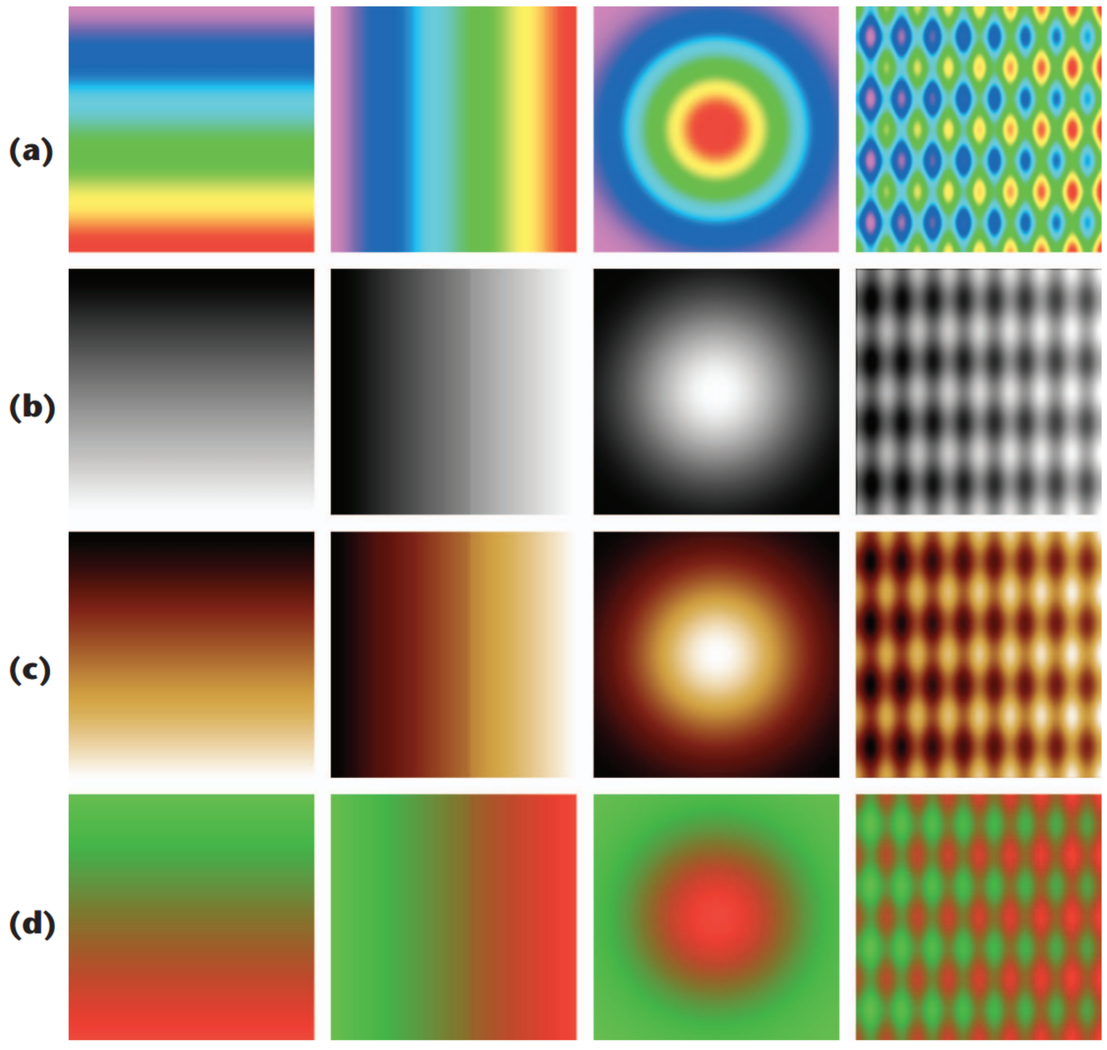
You ***must*** represent linear data change with linear perceptual change



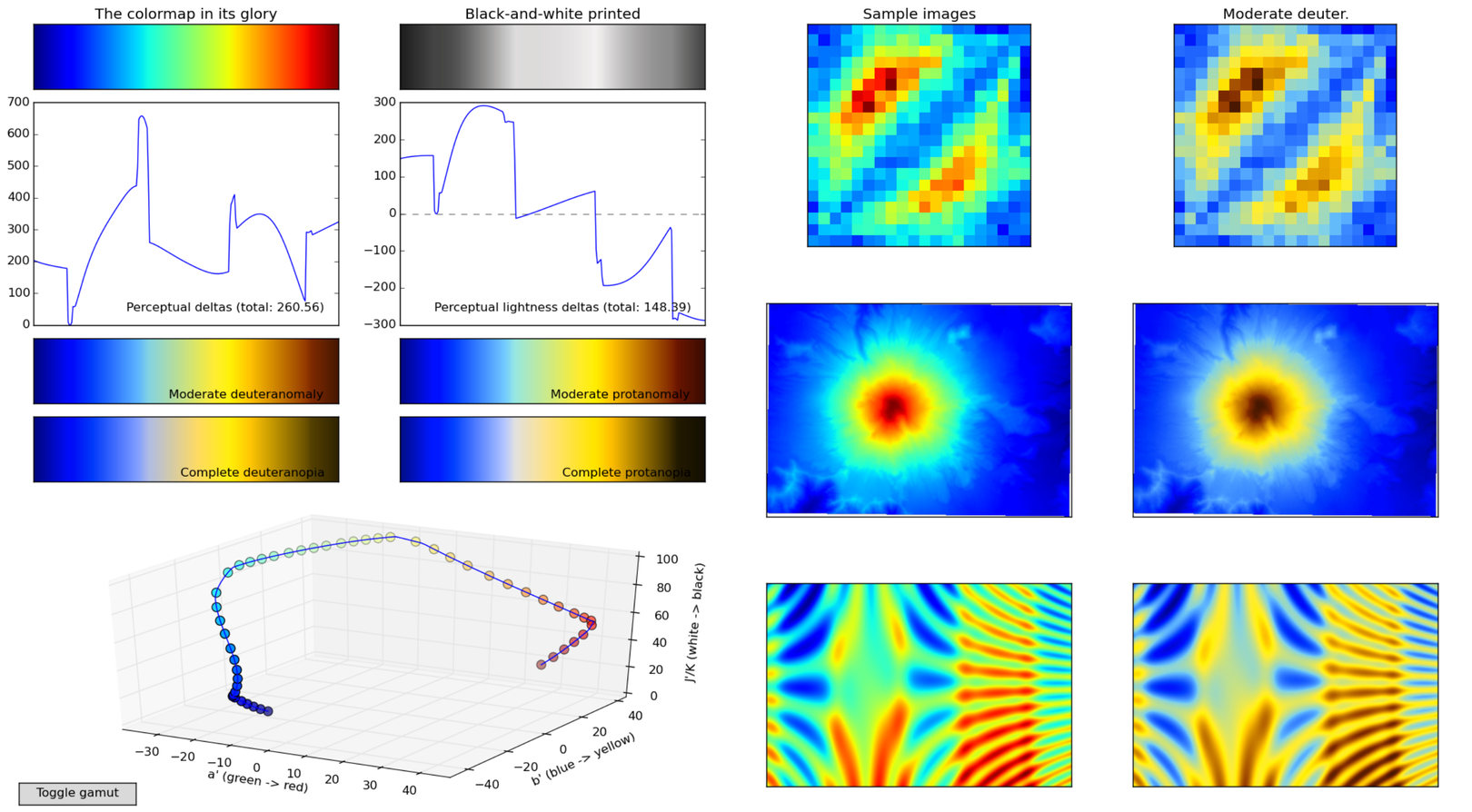
Sequential Color

Perceptual color bands result in misleading artifacts (a), while sharp gradients in the data (b, c second column) are obscured

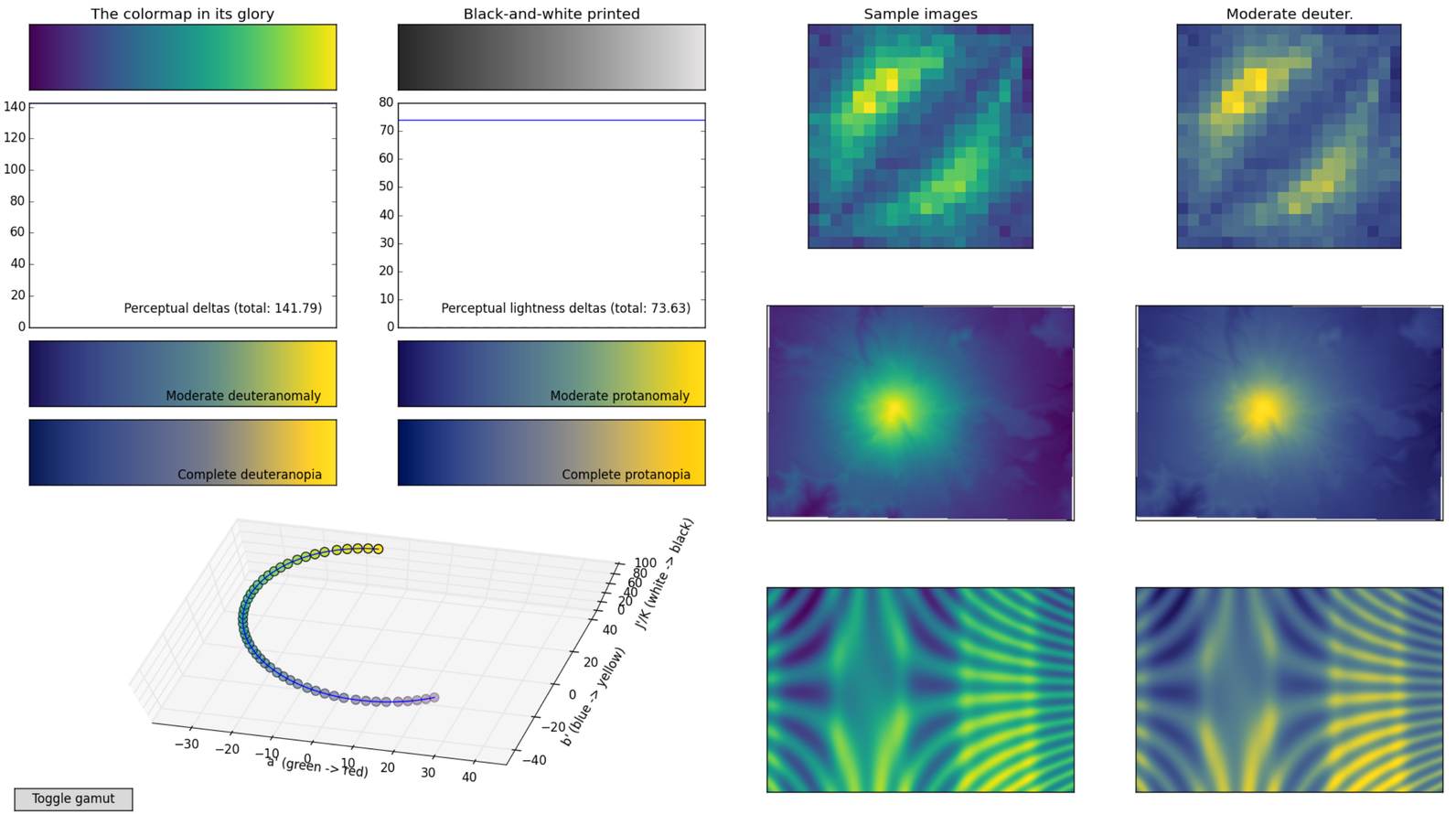
Rainbow Color Map (Still) Considered Harmful



Perceptually Uniform Maps

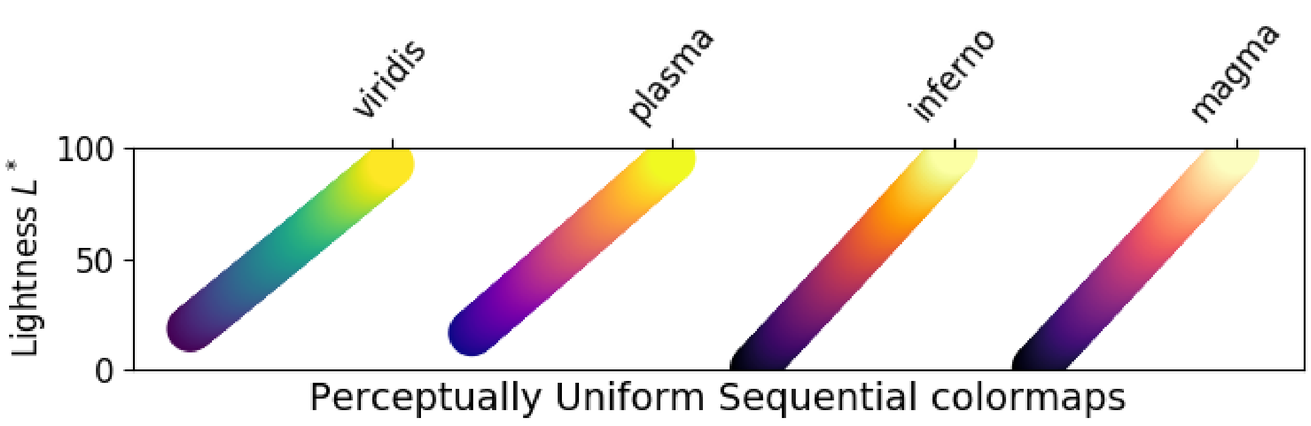


Perceptually Uniform Maps



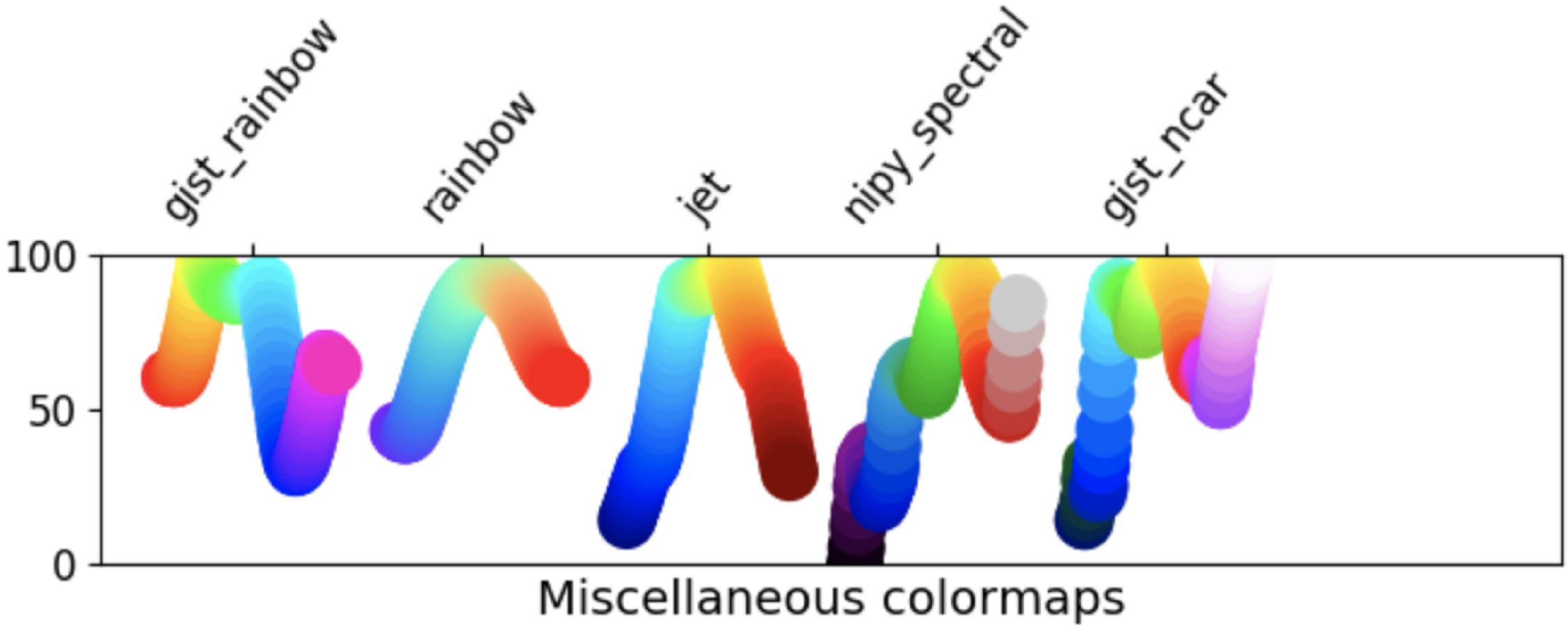
Perceptually Uniform Maps

Matplotlib's new(ish) color maps provide linear increases in L and rotations through ab for linear data changes



Perceptually Uniform Maps

… and guidance on why other maps fall short



Never jet

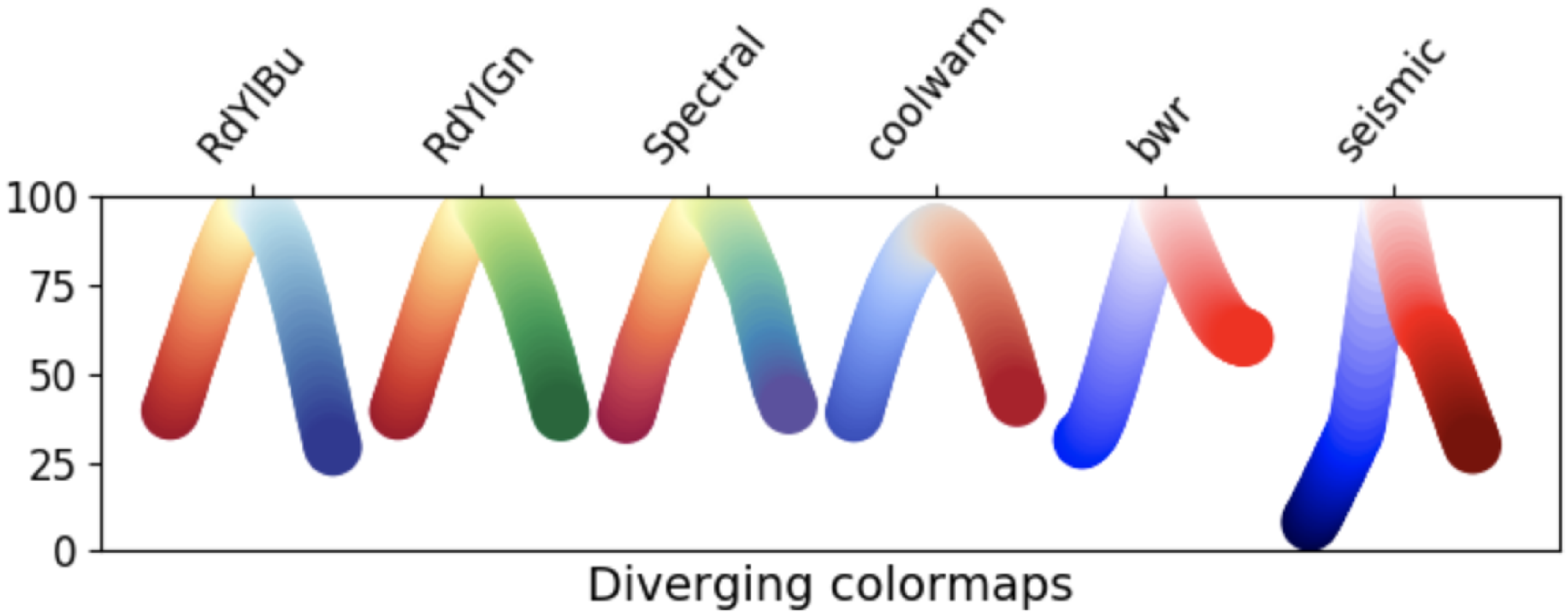
TIL that seaborn refuses to use the jet color scheme in a most wonderful fashion

```
In [1]: import seaborn
        seaborn.set(palette='jet')
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-1-88c614984290> in <module>()
      1 import seaborn
----> 2 seaborn.set(palette='jet')

~/.local/lib/python3.5/site-packages/seaborn/rcmod.py in set(context, style
, palette, font, font_scale, color_codes, rc)
    107     set_context(context, font_scale)
    108     set_style(style, rc={"font.family": font})
--> 109     set_palette(palette, color_codes=color_codes)
    110     if rc is not None:
    111         mpl.rcParams.update(rc)

~/.local/lib/python3.5/site-packages/seaborn/rcmod.py in set_palette(palett
e, n_colors, desat, color_codes)
    501
    502     """
--> 503     colors = palettes.color_palette(palette, n_colors, desat)
    504     if mpl_ge_150:
    505         from cycler import cycler

~/.local/lib/python3.5/site-packages/seaborn/palettes.py in color_palette(p
alette, n_colors, desat)
    170             palette = husl_palette(n_colors)
    171         elif palette.lower() == "jet":
--> 172             raise ValueError("No.")
    173         elif palette in SEABORN_PALETTES:
    174             palette = SEABORN_PALETTES[palette]

ValueError: No.
```

Divergent

Easy: perceptually uniform lightness “^”, with different hues

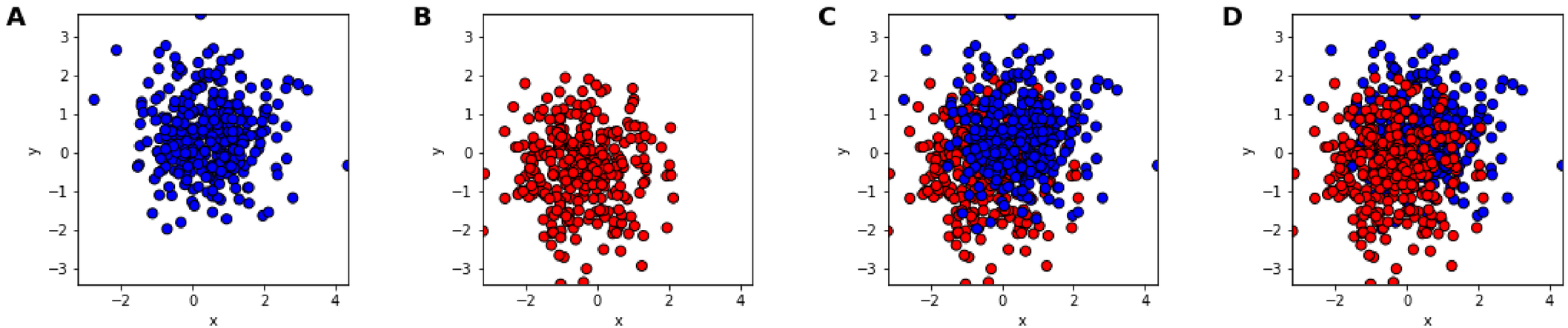


Divergent

Data Shading

Big Data Pitfalls

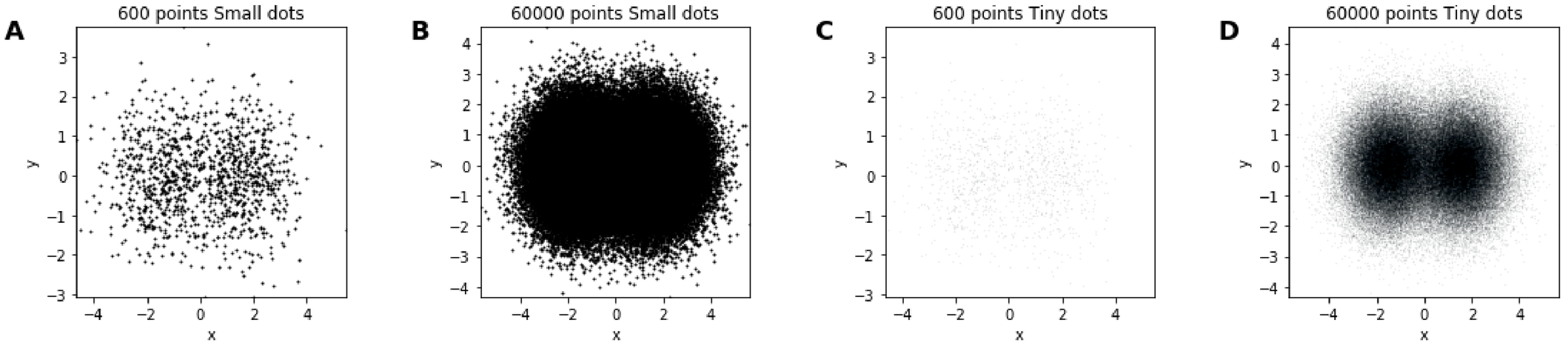
As observations increase, we can no longer directly plot them!



A single marker is substantially larger than what it represents

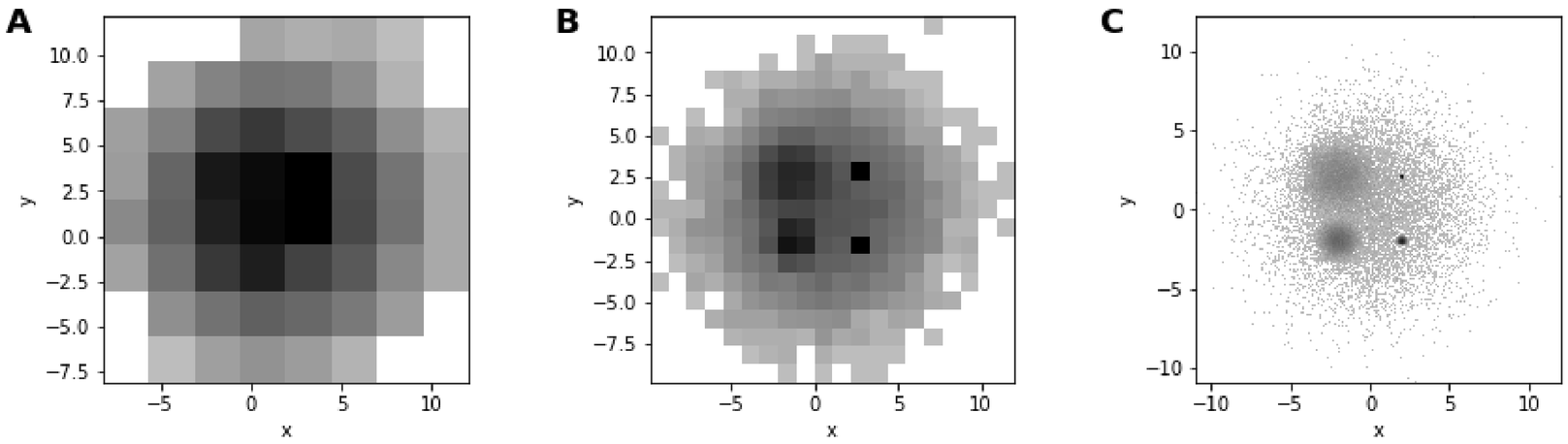
Big Data Pitfalls

We can’t even choose a good marker size up front



Big Data Pitfalls

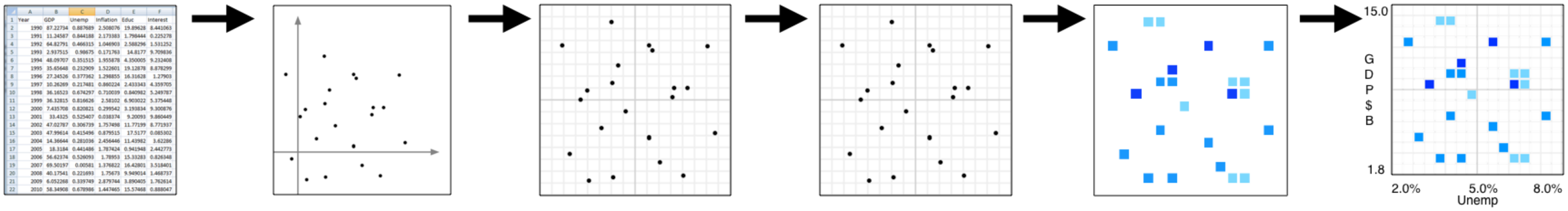
Heatmaps have similar issues wrt resolution



Visualization As Map Reduce

Data Shading is the process of turning data directly into an image. You can think of this like a Map Reduce ETL per pixel

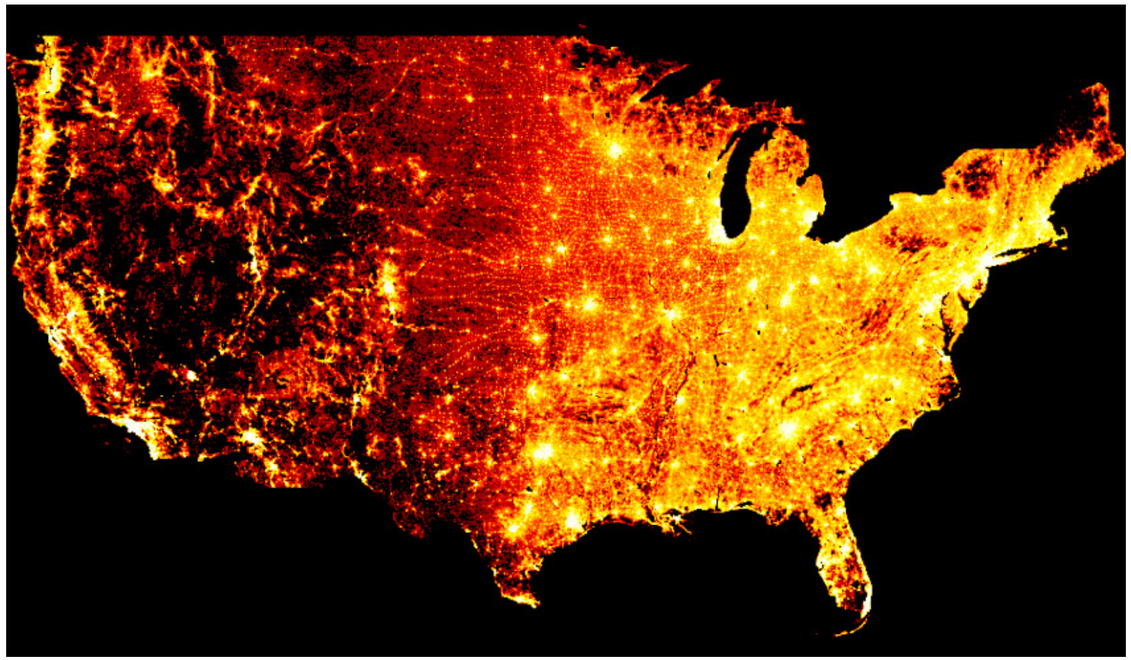
Projection *Aggregation* *Transformation* *Colormapping* *Embedding*



Data Scene Aggregate(s) Image Plot

Datashader

```
import datashader as ds
from datashader import transfer_functions as tf
import dask.dataframe as dd

# lat, lon for every person in census!
df = dd.read_parquet('...')

cvs = ds.Canvas(...)
agg = cvs.points(df, 'lon', 'lat')
tf.shade(agg, how='eq_hist')
```



Where We Are

```cython
def primes(int nb_primes):
    cdef int n, i, len_p
    cdef int p[1000]
    if nb_primes > 1000:
        nb_primes = 1000

    len_p = 0  # The current number of elements in p.
    n = 2
    while len_p < nb_primes:
        # Is n prime?
        for i in p[:len_p]:
            if n % i == 0:
                break
        else:
            p[len_p] = n
            len_p += 1
        n += 1

    # Let's return the result in a python list:
    result_as_list  = [prime for prime in p[:len_p]]
    return result_as_list
```

“Python”

- We have repeatable and appropriately specified virtual environments
- We can choose between minimal reqs and frozen dependencies
- We understand the role of `pipenv` for library and app development

CODE

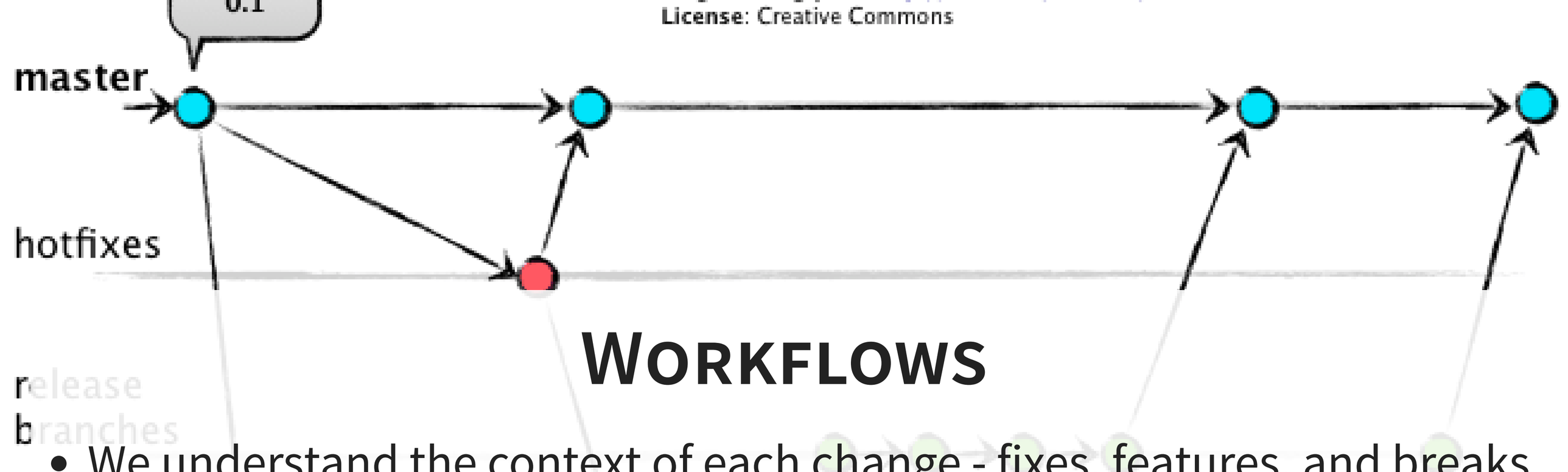
DEPLOY

Testing

- We know the importance of unit testing, how to test, and what to test
- We can measure coverage, including code branches
- We know the environment matters, and test libraries where it counts
- We can mock out code, even faking integration with other systems

TEST

MONITOR



Workflows

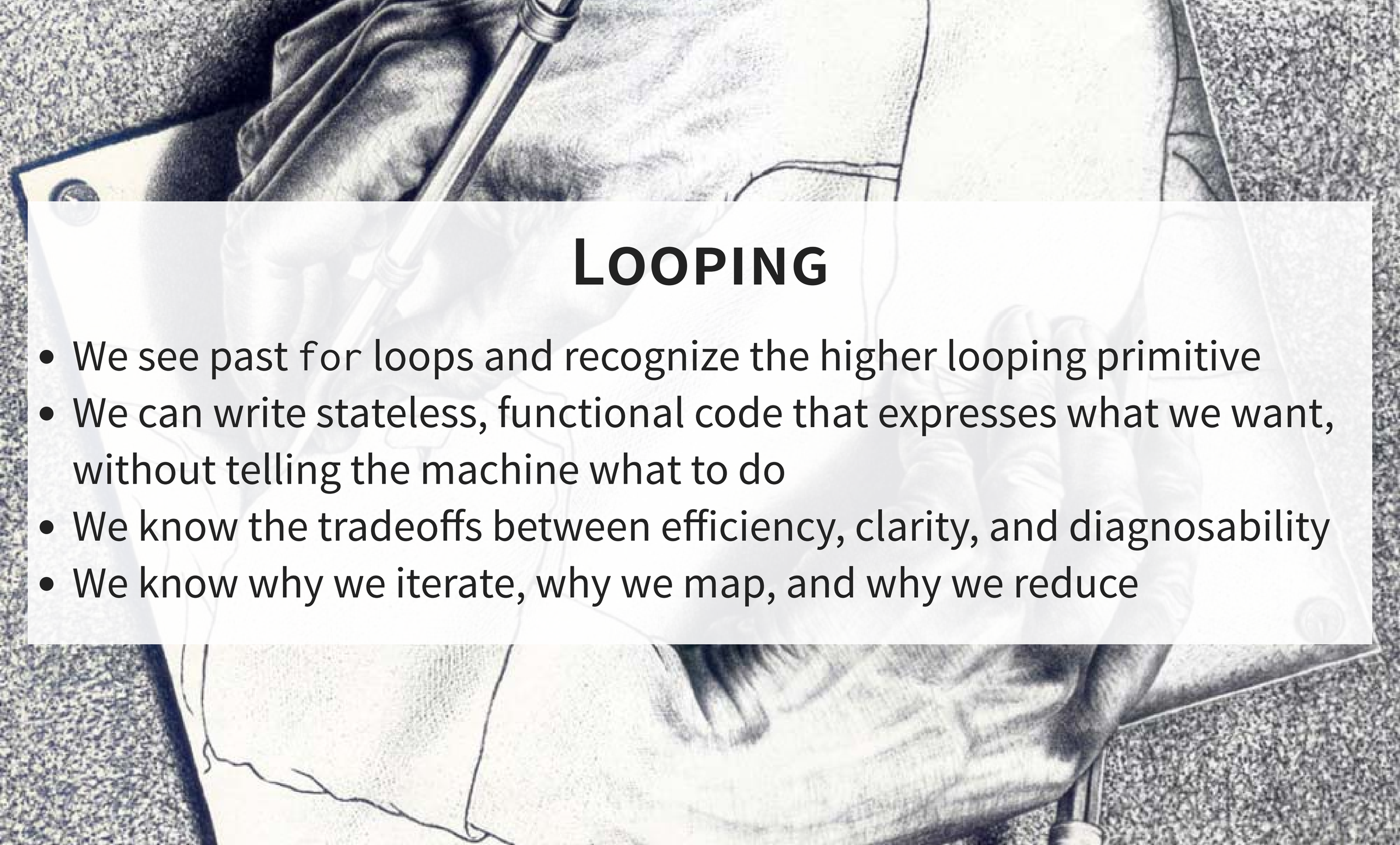
- We understand the context of each change - fixes, features, and breaks
- We track history linearly and meaningfully using semantic versions
- We bootstrap our work and codify best practices with templates

Higher Levels

- We recognize meta-patterns in code at levels higher than a function
- We wrap, register, and alter functions using decorators
- We provide reusable context to code using context managers

Deployment

- We know what is part of our code, and what is not
- We configure our deployments with environment variables, ensuring our code is useable anywhere
- We handle data and secrets with discretion and privacy

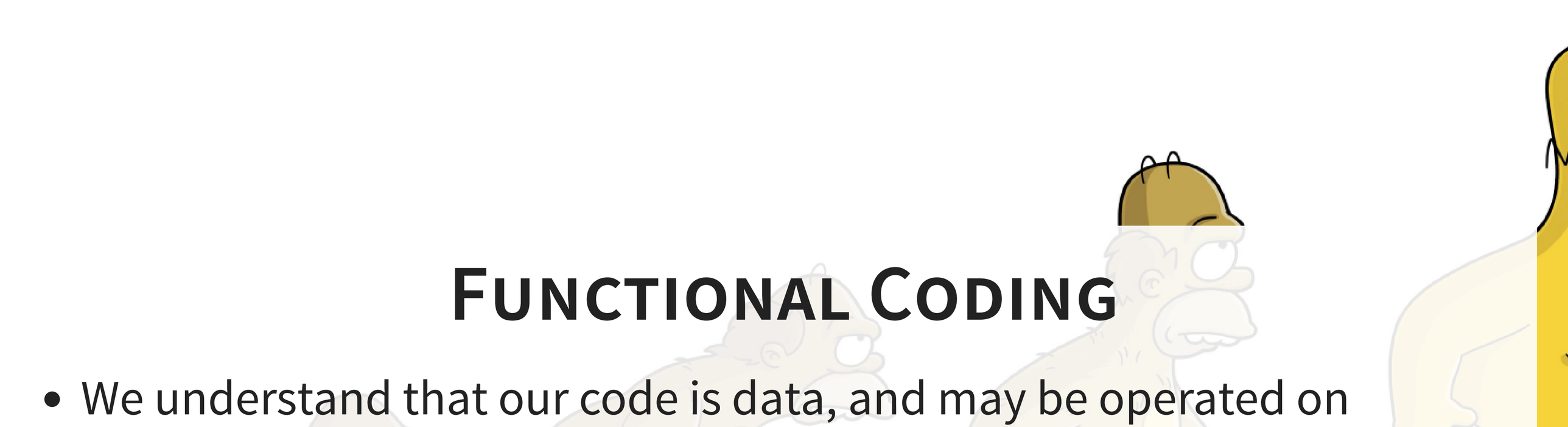


Looping

- We see past `for` loops and recognize the higher looping primitive
- We can write stateless, functional code that expresses what we want, without telling the machine what to do
- We know the tradeoffs between efficiency, clarity, and diagnosability
- We know why we iterate, why we map, and why we reduce

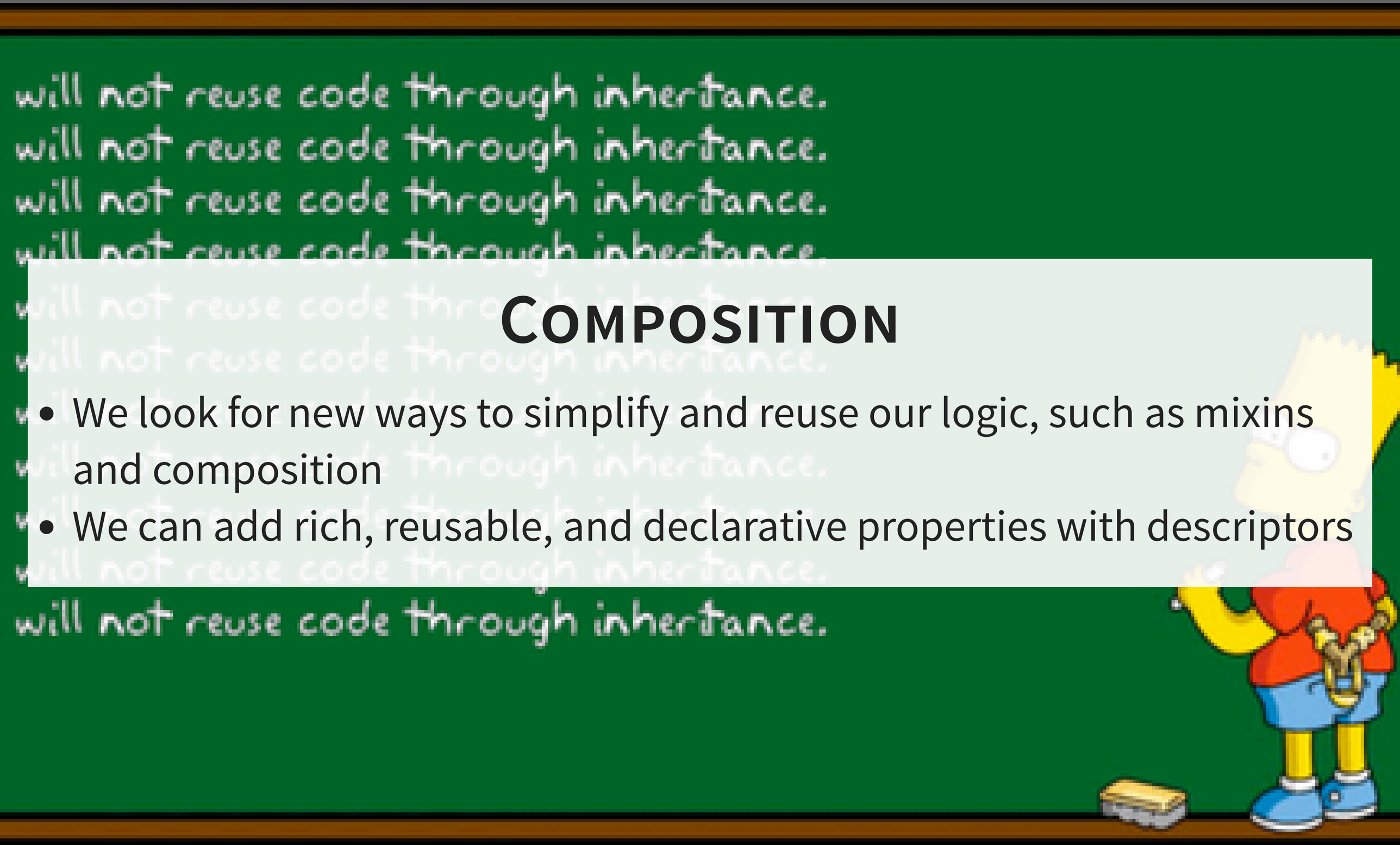
Functional Coding

- We understand that our code is data, and may be operated on
- We know when state is valuable, and when it fails us
- We can encapsulate logic in functions that we pass as arguments to higher frameworks
- We strive to be declarative in all that we do



Composition

- We look for new ways to simplify and reuse our logic, such as mixins and composition
- We can add rich, reusable, and declarative properties with descriptors



Graphical Programs

- We recognize our programs, literally, as directed graphs
- We can directly visualize, debug, and optimize a graph
- We can construct graphs to represent our ideas and programs

Graphical Programs

- We can scaffold our application at the highest level using dataflows
- We can solve new and extremely important problems using the graph, such as seen with Salted

Data Scaling

- We scale beyond memory with dask, and know how to represent data in columns and on disk
- We recognize the interactions between algorithms and storage, including predicate pushdowns, columnar operations, fancy partitions, and indexing

The Web

- We know when data is immutable, when it is live, and when we can compromise
- We can build a database for our live data, whether flat, star, or snowflake
- We recognize the all-encompassing power of Django, yet can use a best-of-breed component like SqlAlchemy when necessary

DB's

- We recognize the difference between and ORM and an analytical workflow, and choose different tools and approaches appropriately
- We can build and query all tables, flat, star, or snowflake
- We intelligently design our primary keys, or ignore them deliberately
- Never again shall we write raw SQL
- We can ***test*** a query
- We ***version*** our DB schema through migrations
- We can reuse table designs with class inheritance

API’s

- We don’t write API clients - we swagger!

Meta

- At the highest level, we use meta classes to augment the very core of how we write code
- We make new methods and factories to churn out complex objects repeatably

Optimization

- We know when to go fast… and when not to
- We can compile python to C with cython, acclerate it JIT with numba, and now when each prevails
- We can leverage the generality of numpy with custom vectorized functions (ufuncs)
- We exploit memory views rather than duplicating data
- We can trade memory for speed and avoid work via memoization

Data Structures

- We can trade accuracy for speed and memory via a sketch
- We understand the utility of a hash, and even how to choose a new hash function

Visualization

- We know our job depends not just on the quality of our math, but by the perception of the results
- We can rapidly present work using declarativce grammars
- We know The Web is the best technology for interactive data, and we embrace those technologies full heartedly
- … yet, we know when it's best to use Python as a glue
- We plot data ***at scale*** via data shading
- … and we never use `jet`!

END OF LIST!

CONGRATS!

Readings

- Python Data Visualization 2018
- **Slides/Video** JakeVDP Python Viz Landscape
- Matplotlib colormaps:
  - MPL Colormaps
  - **Video** A Better Default Colormap for MPL
  - Choosing colormaps
- Ten simple rules for better figures
- Plotting pitfalls (big data)
- Poke around:
  - PyViz
  - D3js
  - Color Brewer
- Optional: rants on `jet`
  - How bad is your colormap?
  - The Rainbow is Dead - Long Live the Rainbow!
  - Rainbow Color Map (Still) Considered Harmful

Bonus: Best of Python

- Hadley Wickham (2010) A Layered Grammar of Graphics, Journal of Computational and Graphical Statistics, 19:1, 3-28, DOI: 10.1198/jcgs.2009.07098