Python Data Visualization for Everyone

Learn how to craft impactful and useful visualizations with matplotlib, seaborn, plotly, bokeh, and streamlit

Bruno Gonçalves

Author, Public Speaker, Trainer, Consultant

Pearson

Lesson 1: Human Perception

1.1 Understanding Color Theory

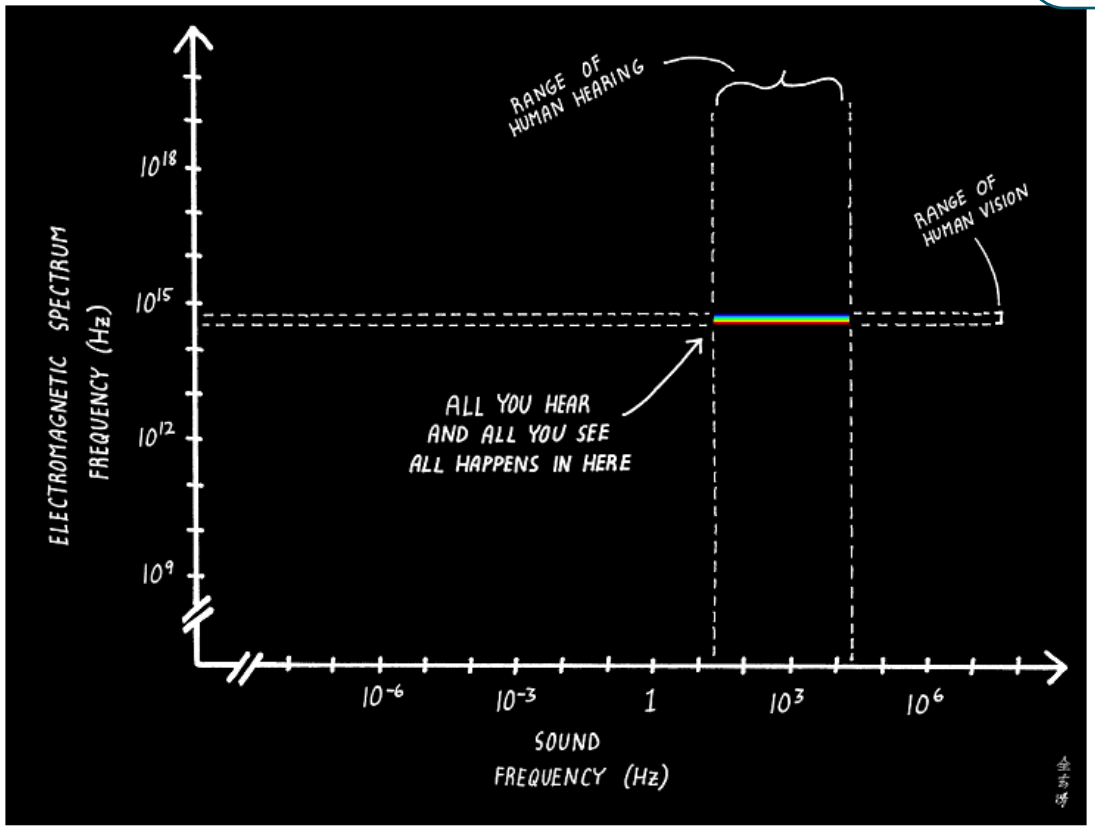
1.2 Overview of Human Vision

1.3 Color Theory

Lesson 1.1:
Understanding Color Theory

1. Human Perception

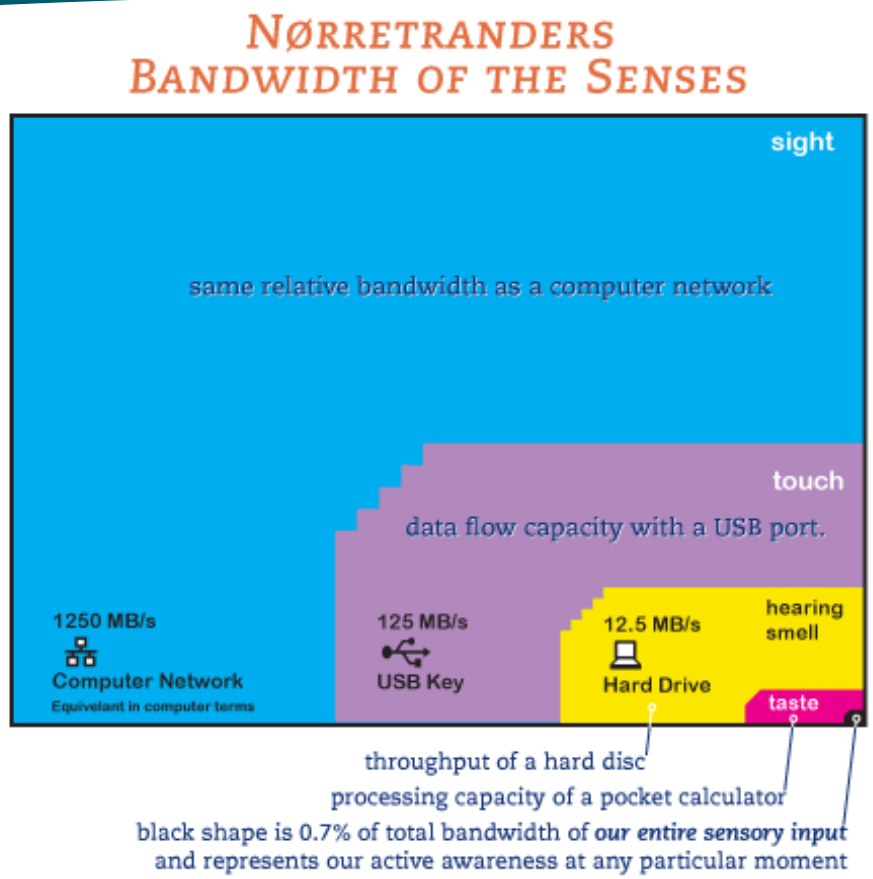
abstrusegoose.com/421



In the grand scheme of things,
we're all pretty much blind and deaf.

Pearson

Human Senses



Source: http://chitownmediapsych.blogspot.ie/2010/09/context-will-be-king-working-title.html
[2]David McCandless : Information Is Beautiful

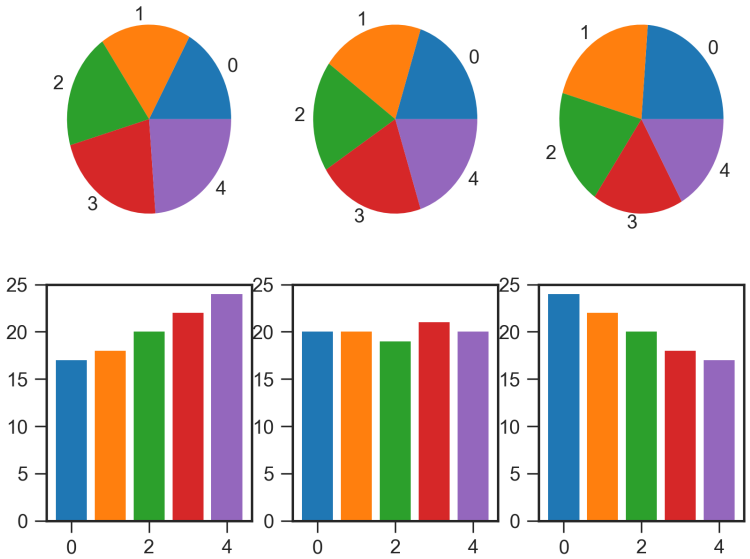
Pearson

Perception

- Some cognitive tasks are significantly easier than others. In order, we are good a distinguishing:
  - Position, length
  - Direction, Angle, Area
  - Volume, Curvature, Shade
  - Color Saturation.

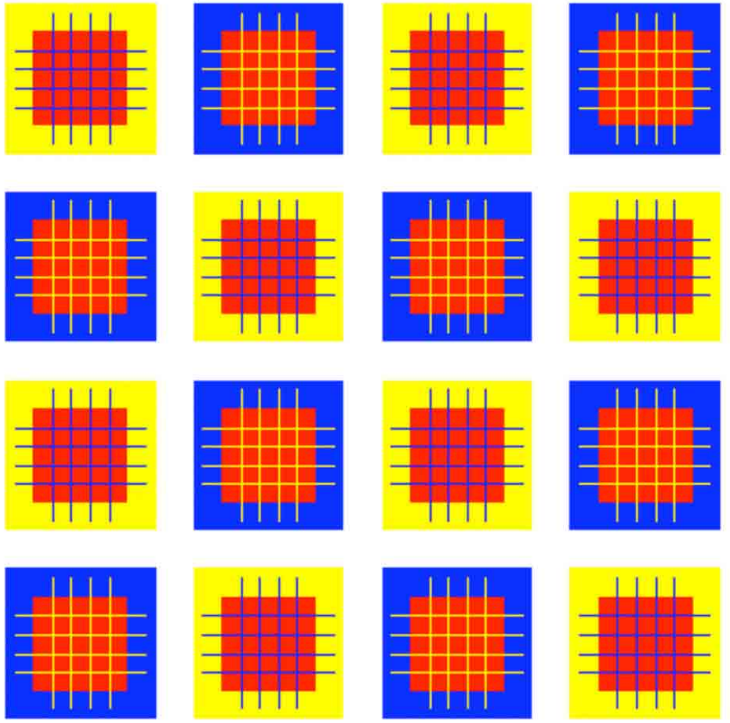
Perception

- Some cognitive tasks are significantly easier than others. In order, we are good a distinguishing:
  - Position, length
  - Direction, Angle, Area
  - Volume, Curvature, Shade
  - Color Saturation.

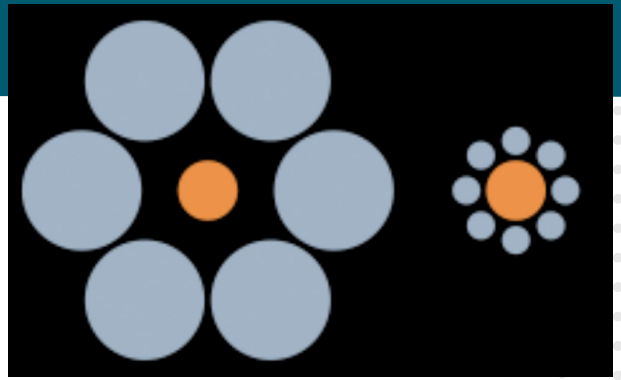


Perception

- Some cognitive tasks are significantly easier than others. In order, we are good a distinguishing:
  - Position, length
  - Direction, Angle, Area
  - Volume, Curvature, Shade
  - Color Saturation.

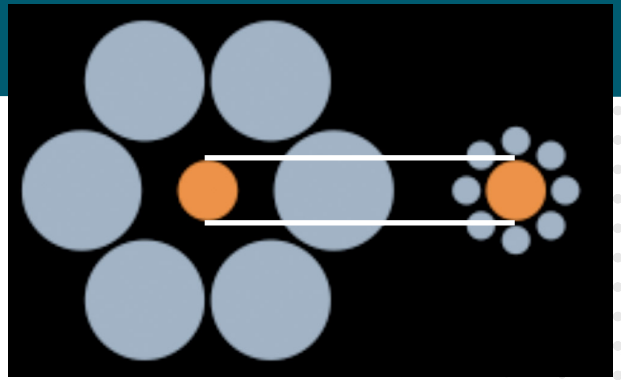


Perception



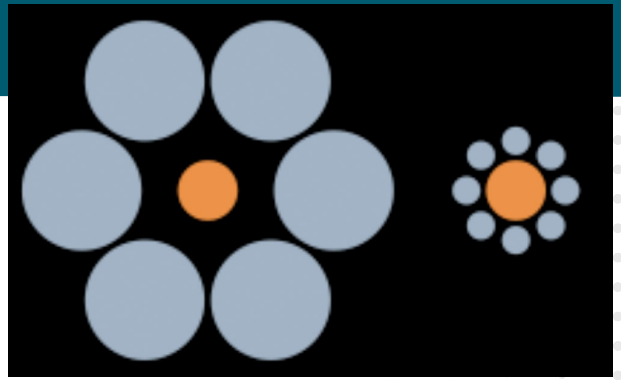
- Context also matters!
  - An object seen in the context of larger objects will appear smaller, while in the content of smaller objects it will appear larger.

Perception



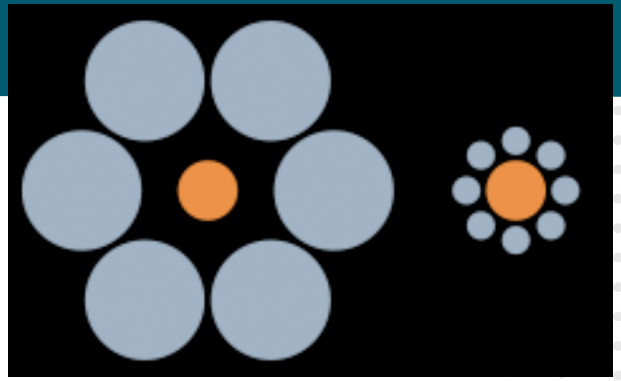
- Context also matters!
  - An object seen in the context of larger objects will appear smaller, while in the content of smaller objects it will appear larger.

Perception

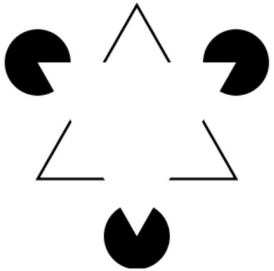


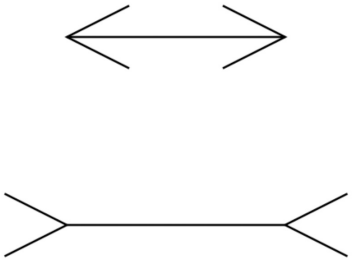
- Context also matters!
  - An object seen in the context of larger objects will appear smaller, while in the content of smaller objects it will appear larger.
  - And we “fill in the gaps”

Perception

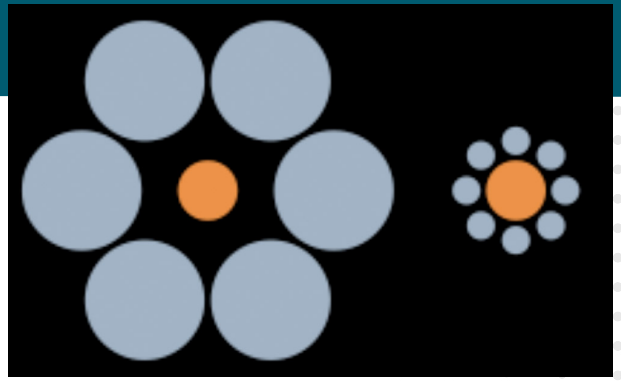


- Context also matters!
  - An object seen in the context of larger objects will appear smaller, while in the content of smaller objects it will appear larger.
  - And we “fill in the gaps”

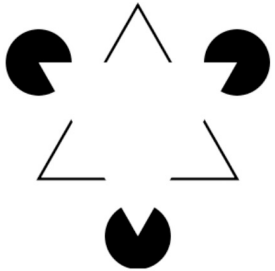


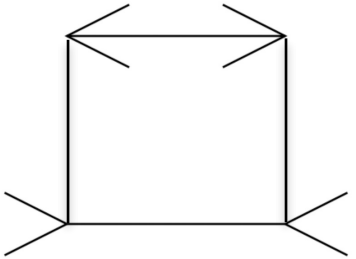


Perception



- Context also matters!
  - An object seen in the context of larger objects will appear smaller, while in the content of smaller objects it will appear larger.
  - And we “fill in the gaps”

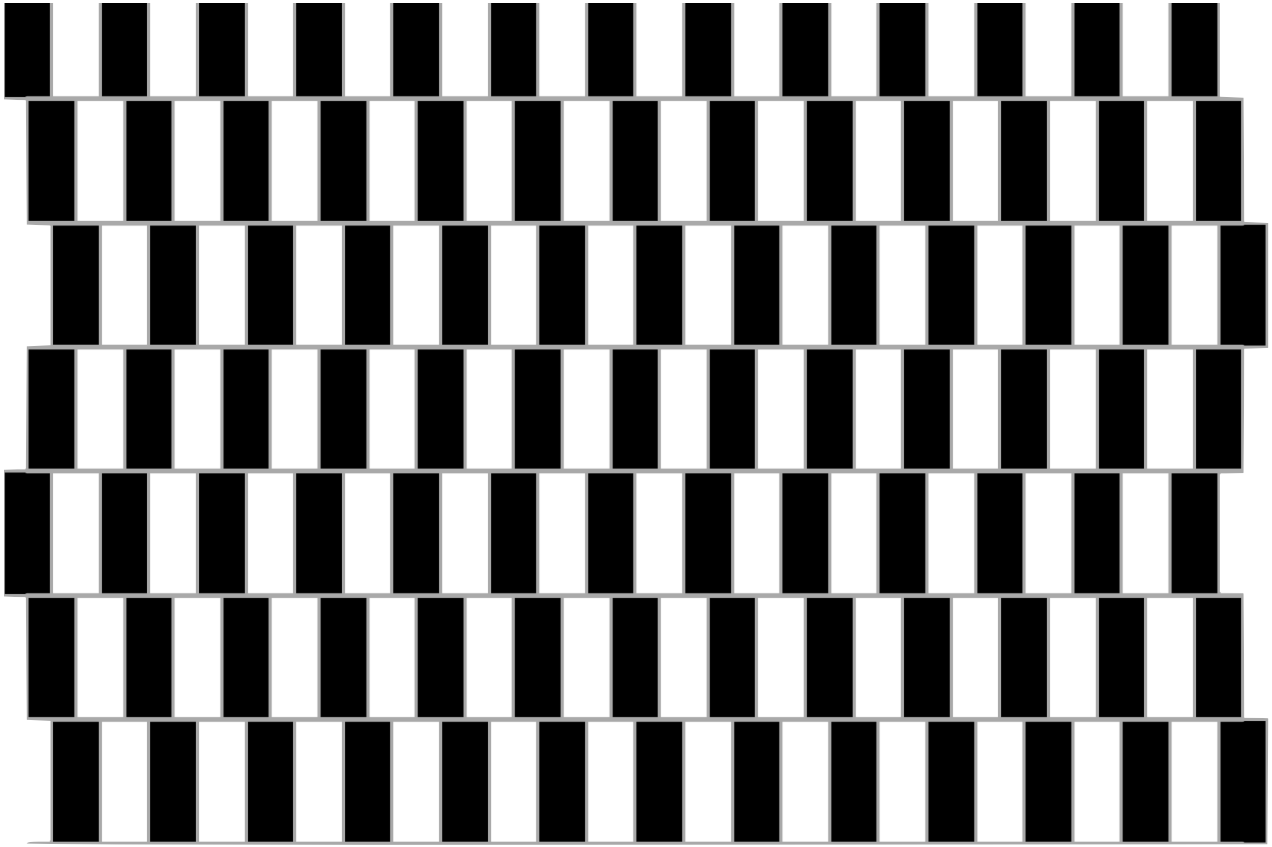




Perception

- Some cognitive tasks are significantly easier than others. In order, we are good at distinguishing:
- Position, length
- Direction, Angle, Area
- Volume, Curvature, Shade
- Color Saturation.

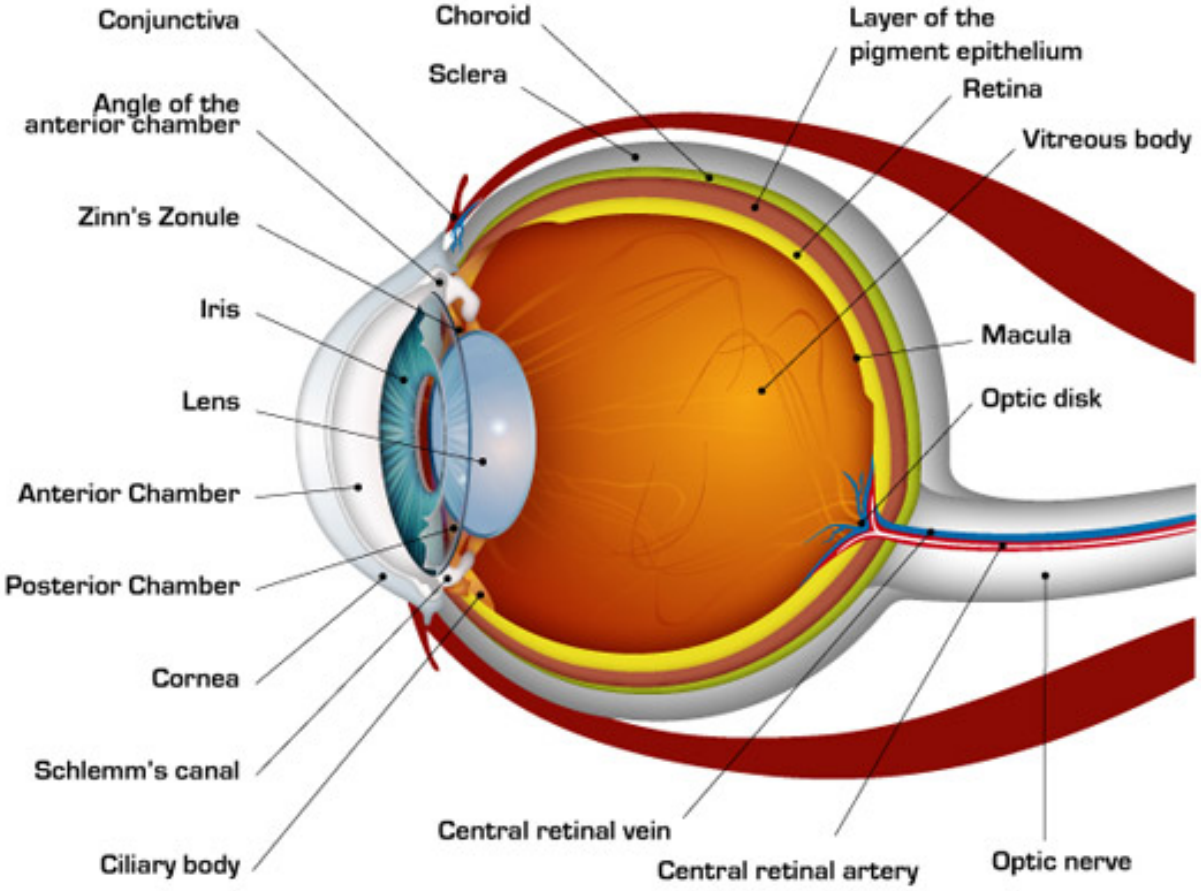
Perception Biases



Lesson 1.2:
Overview of Human Vision

Human Vision

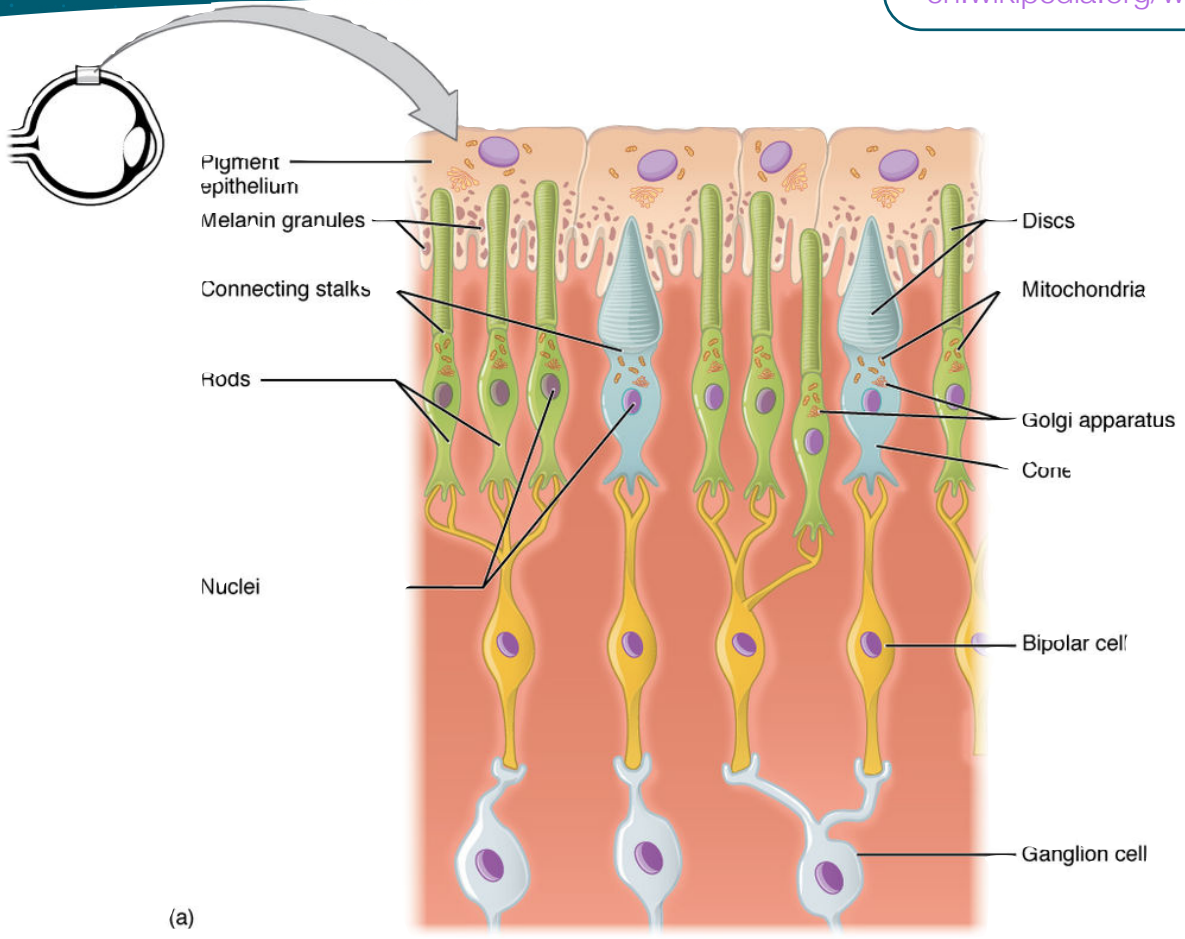
en.wikipedia.org/wiki/Photoreceptor_cell



Pearson

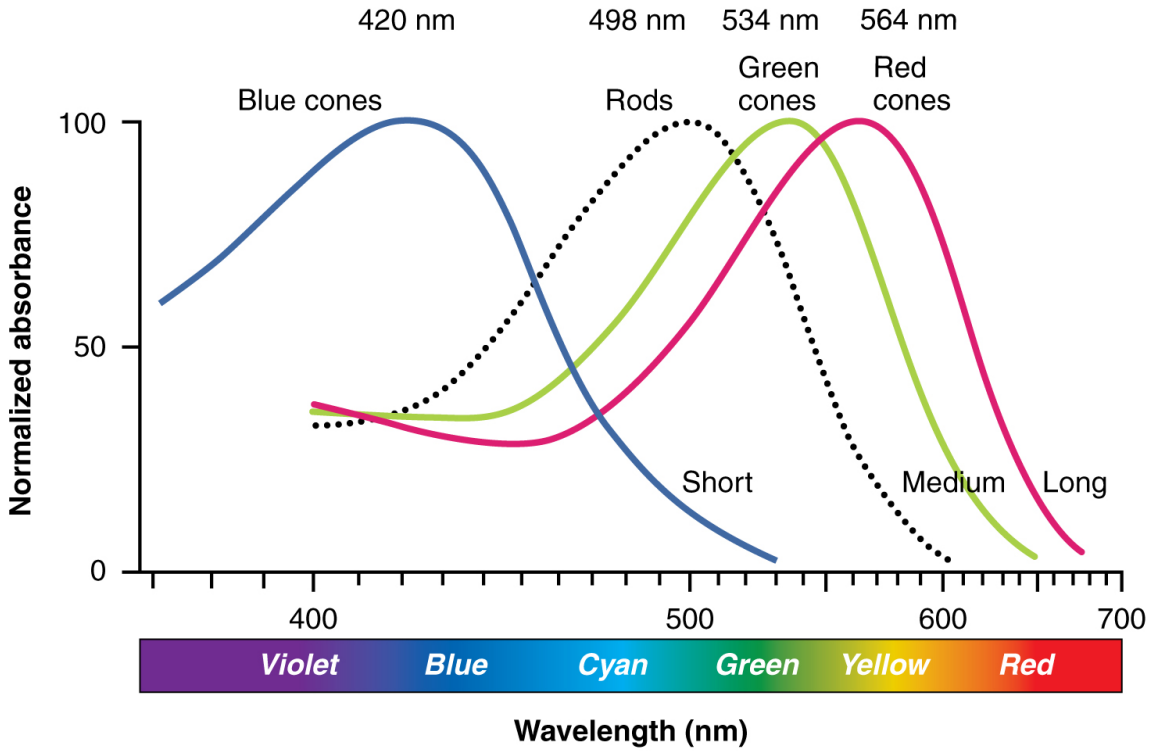
Human Vision

en.wikipedia.org/wiki/Photoreceptor_cell



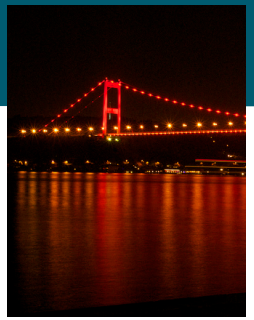
Human Vision

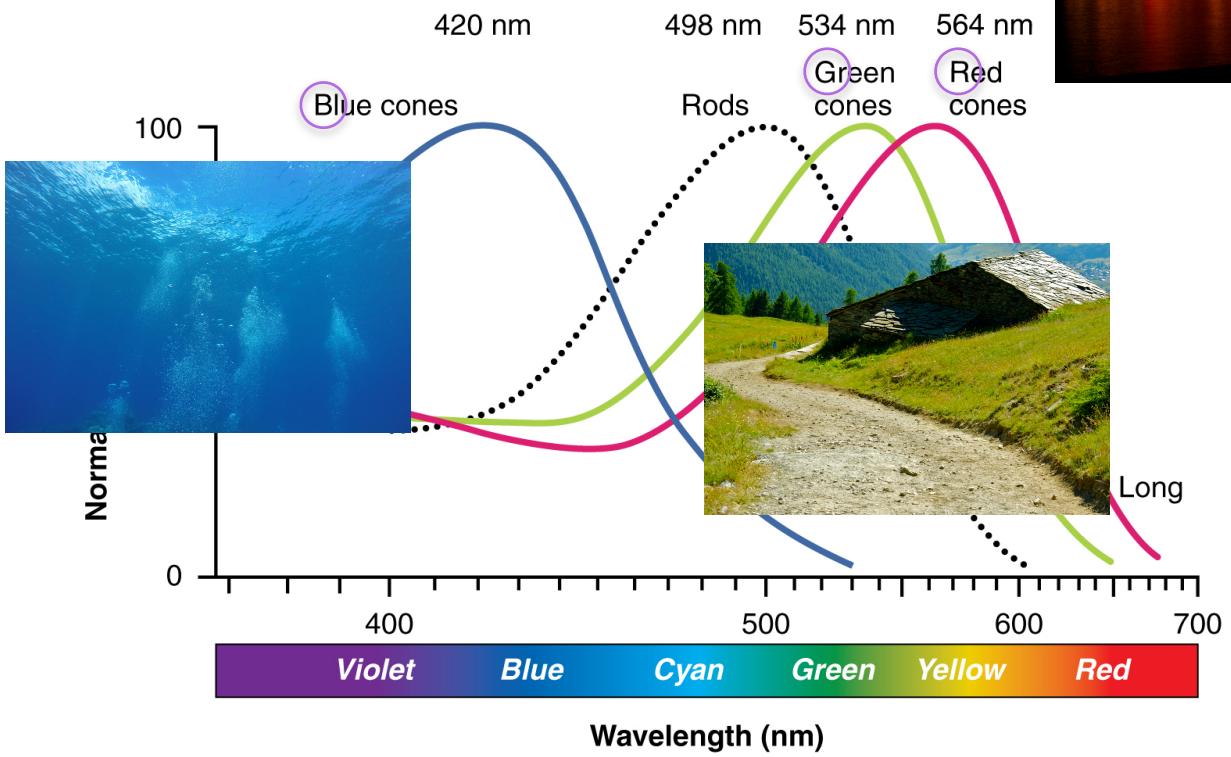
en.wikipedia.org/wiki/Photoreceptor_cell



Pearson

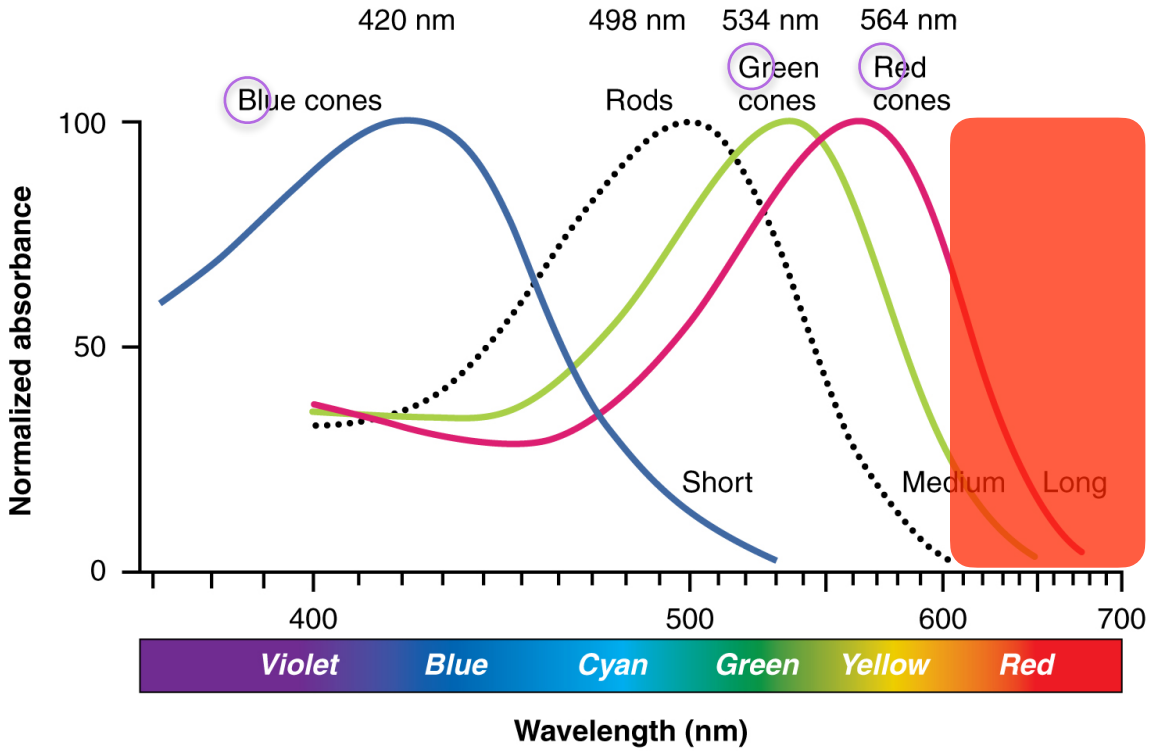
Human Vision





Pearson

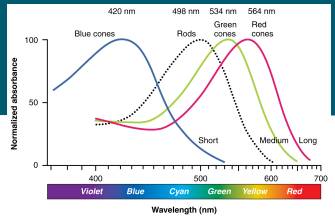
Human Vision



Night vision mode

Pearson

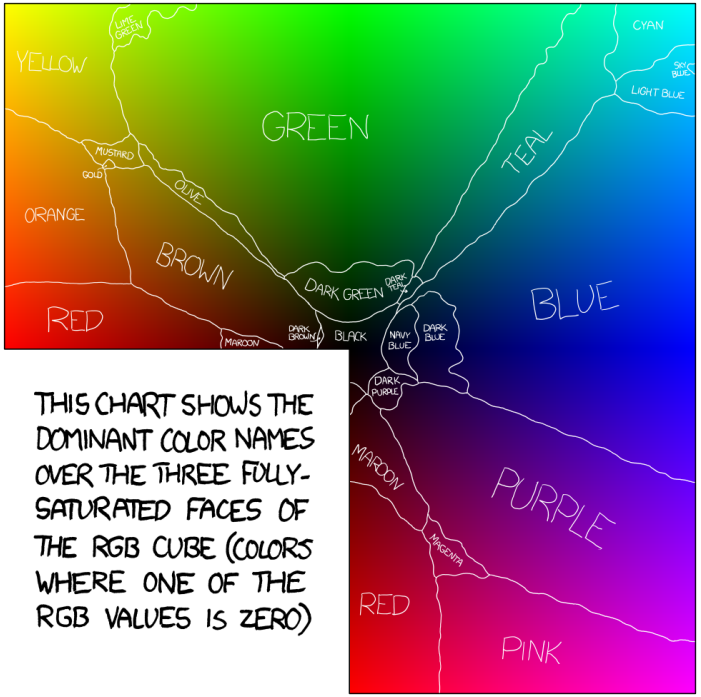
Colors galore!



Pearson

Color Perception

blog.xkcd.com/2010/05/03/color-survey-results/

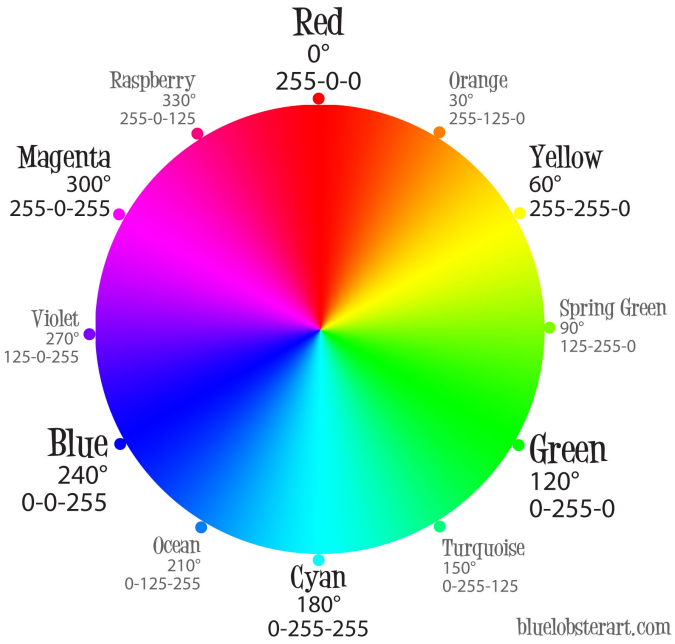


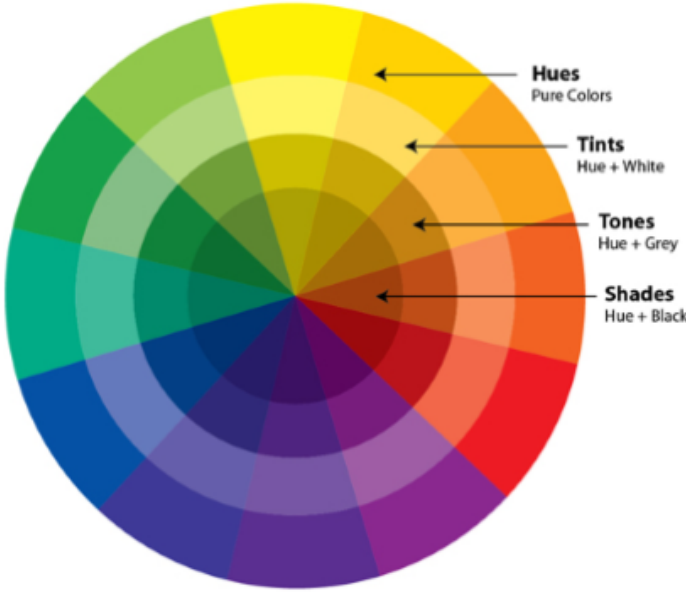
THIS CHART SHOWS THE DOMINANT COLOR NAMES OVER THE THREE FULLY-SATURATED FACES OF THE RGB CUBE (COLORS WHERE ONE OF THE RGB VALUES IS ZERO)

Pearson

Lesson 1.3:
Color Schemes

Color Wheel

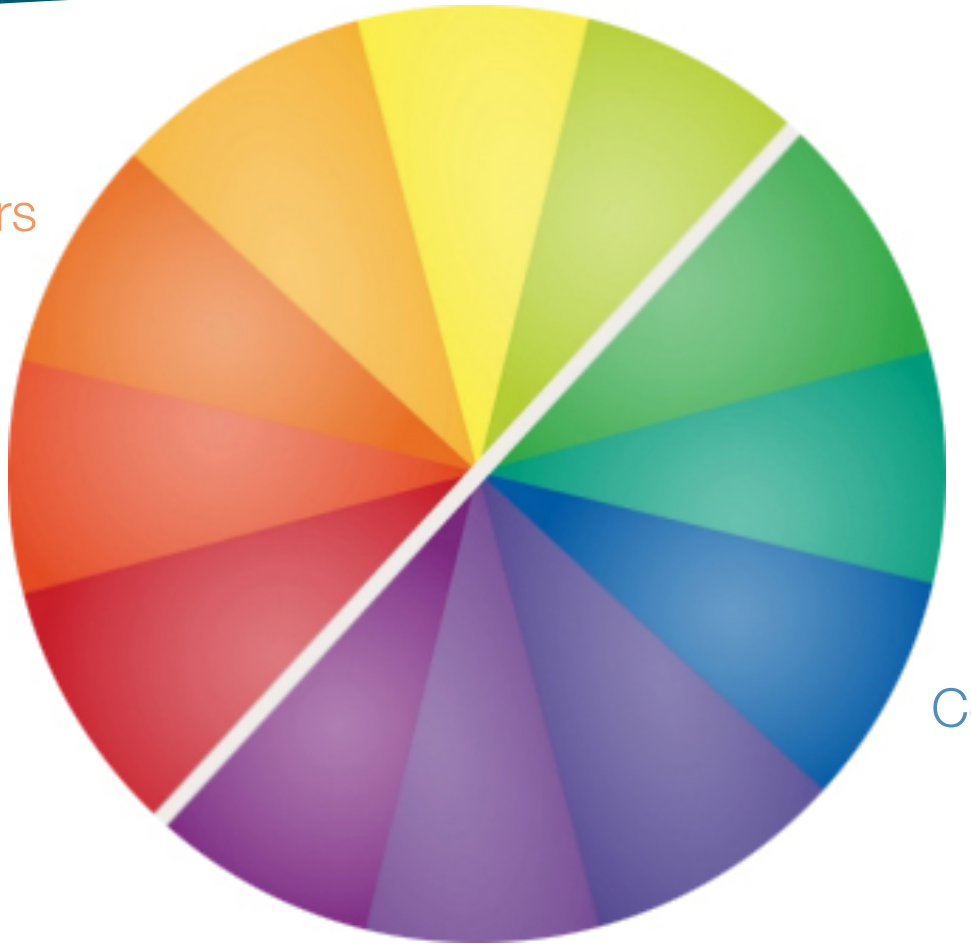




Pearson

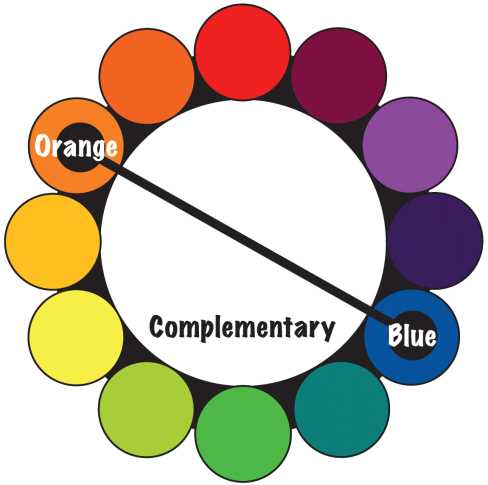
Color Schemes

Warm Colors



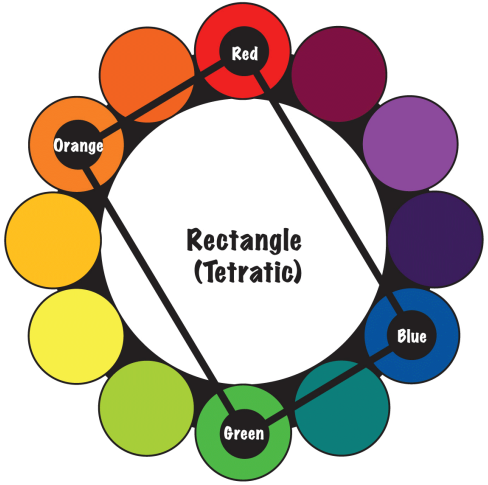
Cold Colors

Color Schemes



Complementary color scheme
Colors that are opposite each other on the color wheel are considered to be complementary colors

(example: Orange and Blue).



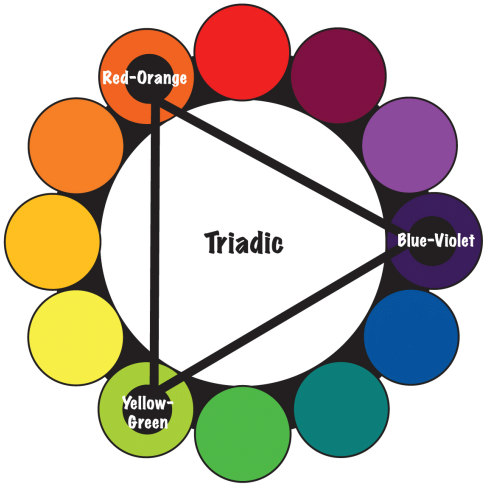
Rectangle (tetradic) color scheme
The rectangle or tetradic color scheme uses four colors arranged into two complementary pairs.

(example: Orange, Red, Blue and Green)

Analogous color scheme
Analogous color schemes use colors that are next to each other on the color wheel.

(example: Green, Blue-Green and Blue)

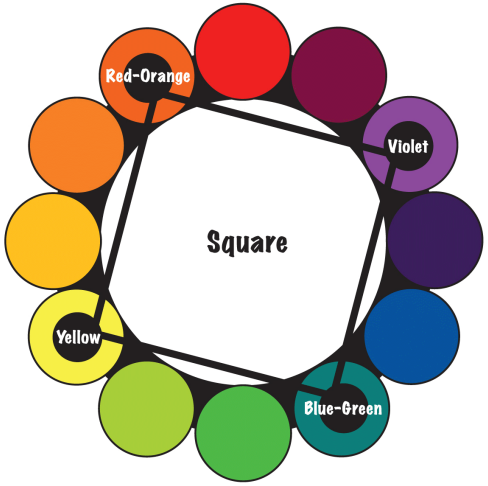
Color Schemes



Triadic color scheme

A triadic color scheme uses colors that are evenly spaced around the color wheel.

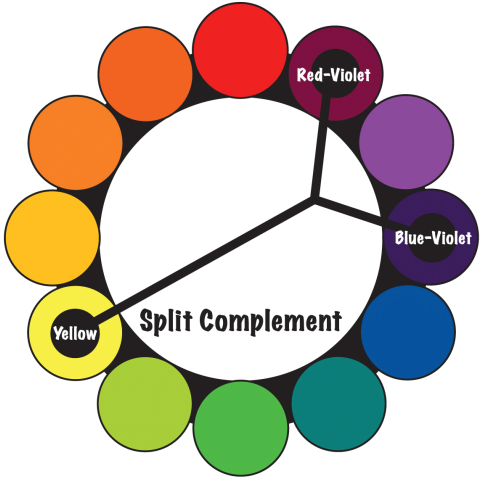
(example: Yellow-Green, Red-Orange and Blue-Violet)



Square color scheme

The square color scheme is similar to the rectangle, but with all four colors spaced evenly around the color circle.

(example: Yellow, Red-Orange, Violet and Blue-Green)

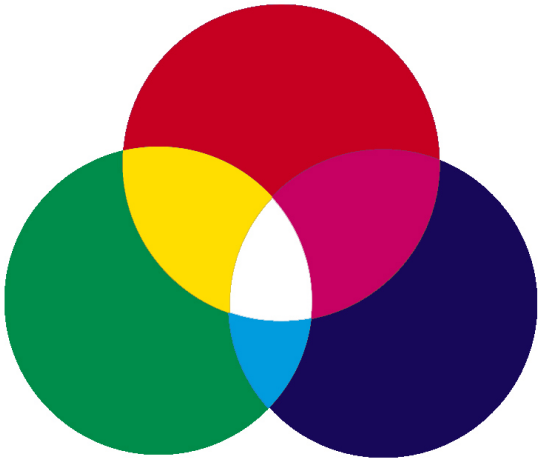


Split-Complementary color scheme

The split-complementary color scheme is a variation of the complementary color scheme. In addition to the base color, it uses the two colors adjacent to its complement.

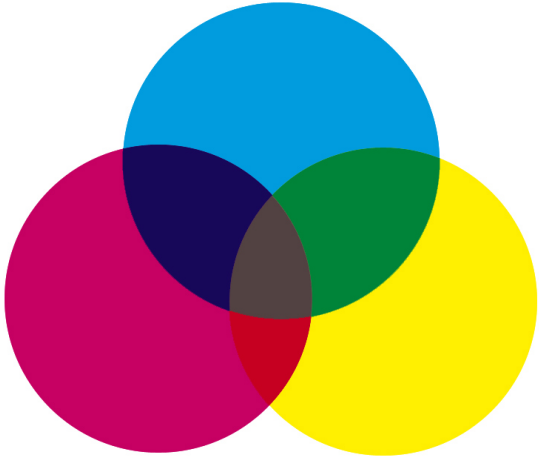
(example: Yellow, Red-Violet and Blue-Violet)

Color Systems



Additive Color (RGB)

Light

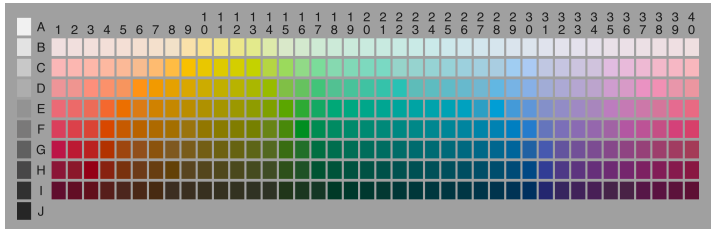


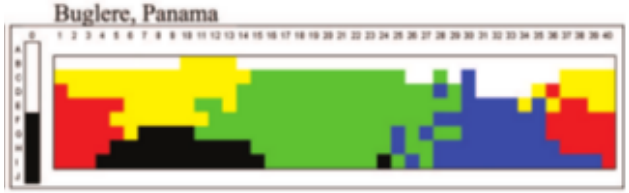
Subtractive color (CMYK)

Ink

Colors and Culture

www1.icsi.berkeley.edu/wcs/





Pearson

Color Blindness

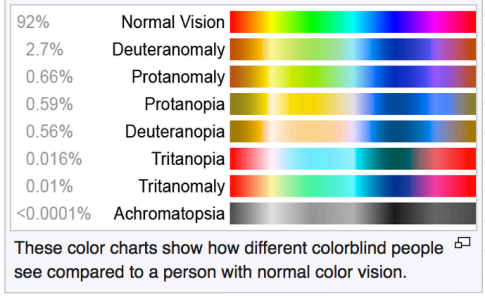
Color Blindness

Pearson

Color Blindness

https://en.wikipedia.org/wiki/Color_blindness

	Cone system	Red		Green		Blue	
	N=normal **A**=anomalous	**N**	**A**	**N**	**A**	**N**	**A**
1	Normal vision						
2	Protanomaly						
3	Protanopia						
4	Deuteranomaly						
5	Deuteranopia						
6	Tritanomaly						
7	Tritanopia						
8	Achromatopsia						
9	Tetrachromat						
10							

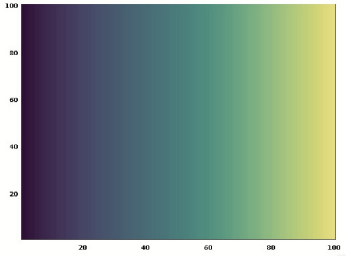


These color charts show how different colorblind people see compared to a person with normal color vision.

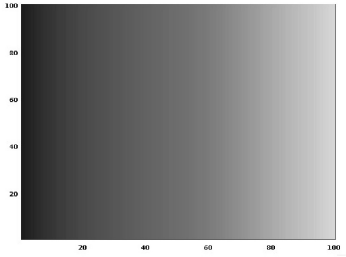
Pearson

Viridis Color Scheme

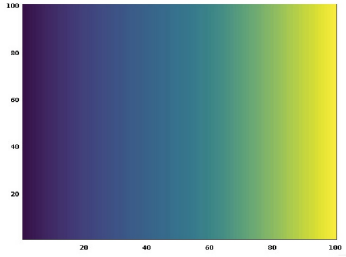
Achromatomaly



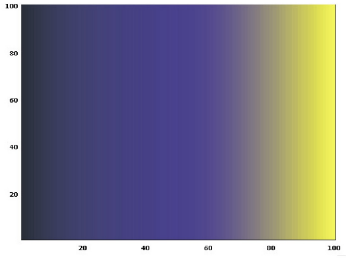
Achromatopsia



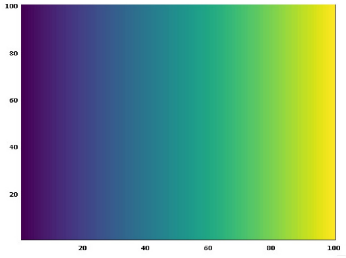
Deuteranomaly



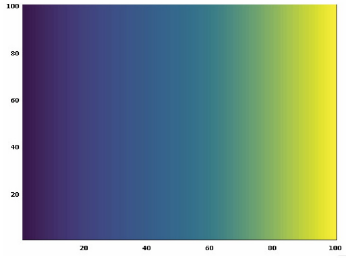
Deuteranopia



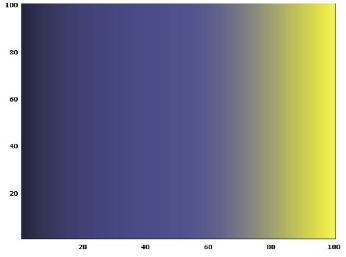
Normal



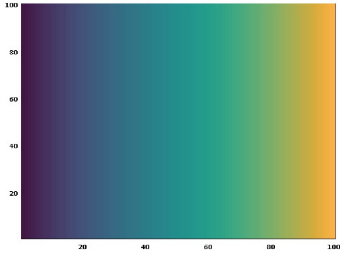
Protanomaly



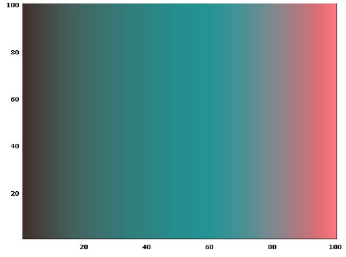
Protanopia



Tritanomaly



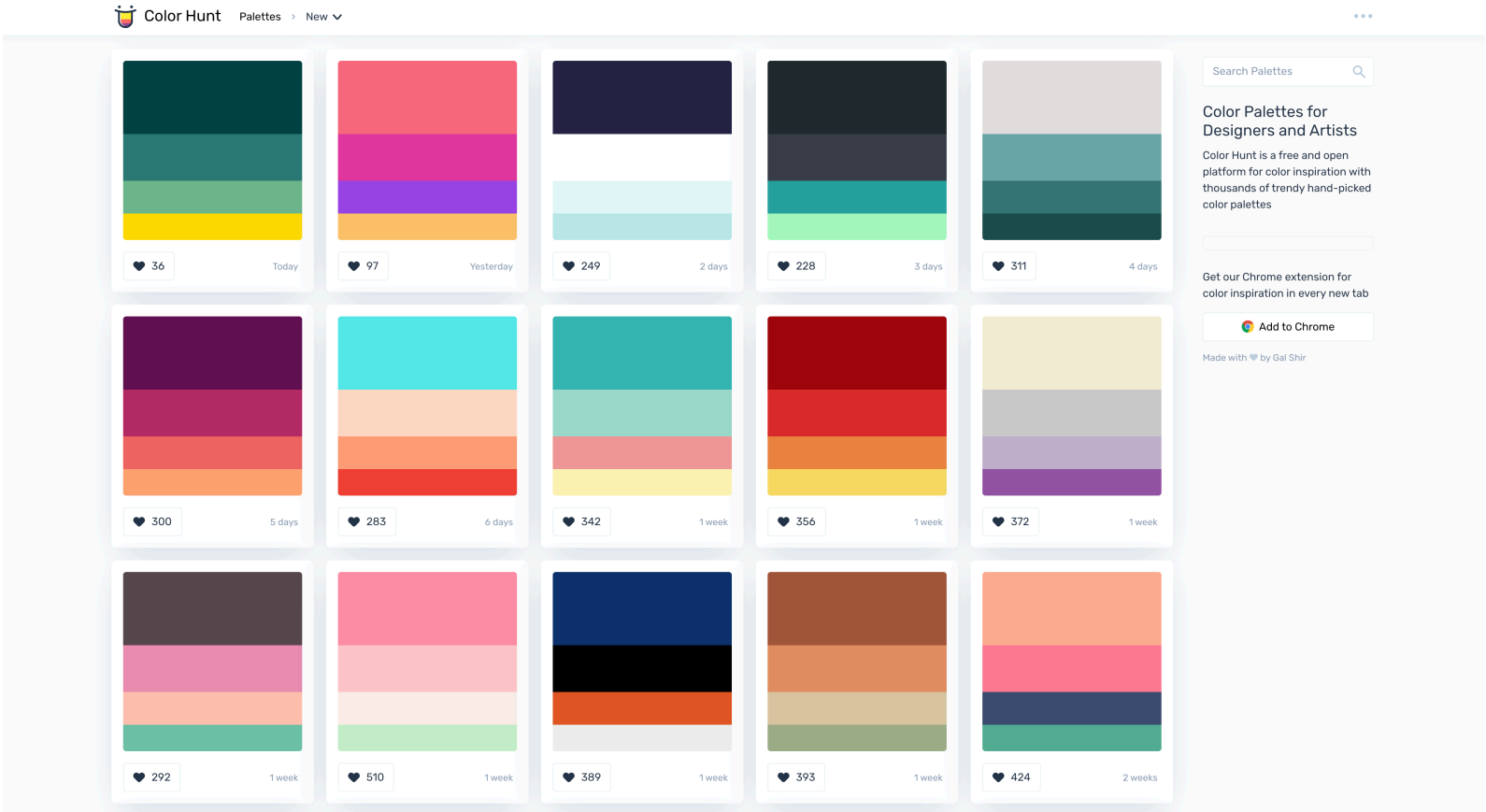
Tritanopia



Pearson

Color Scheme Choosers

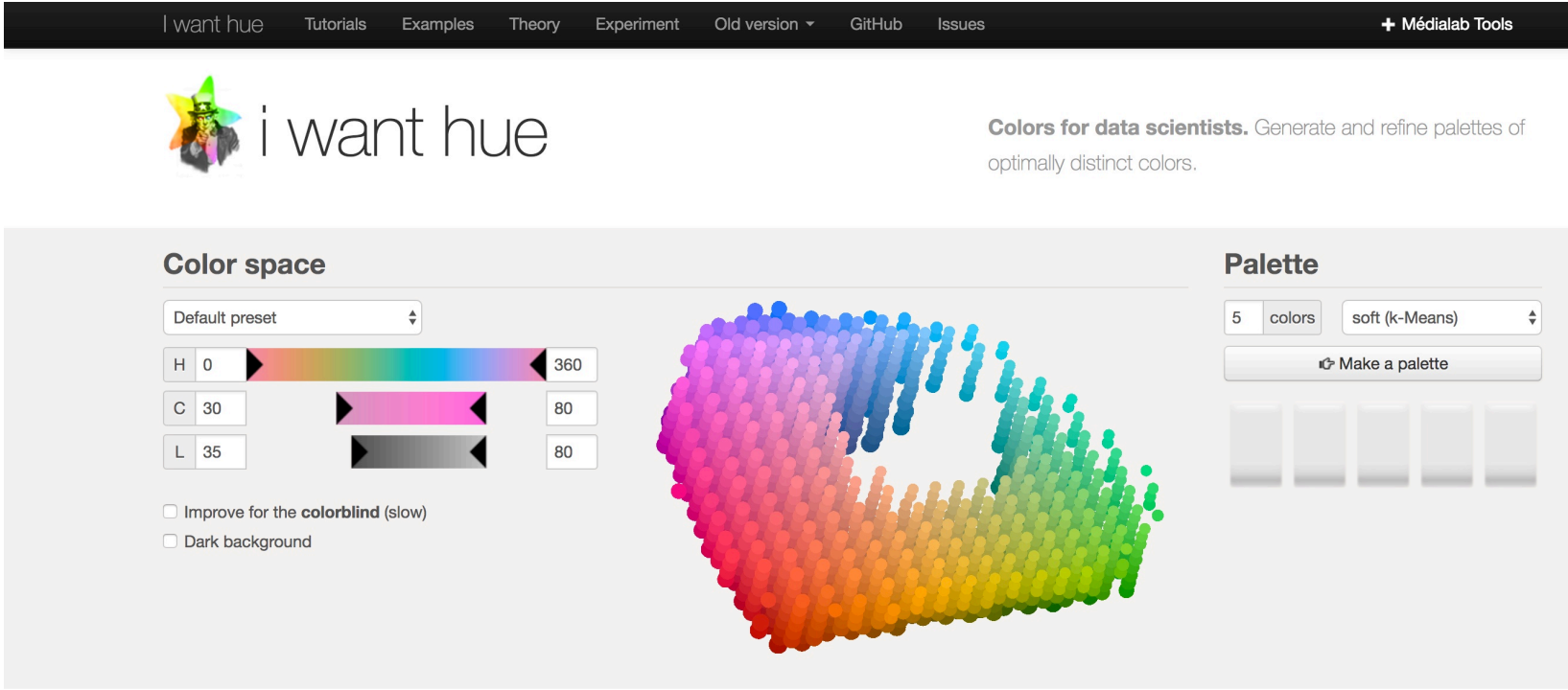
colorhunt.co



Pearson

Color Scheme Choosers

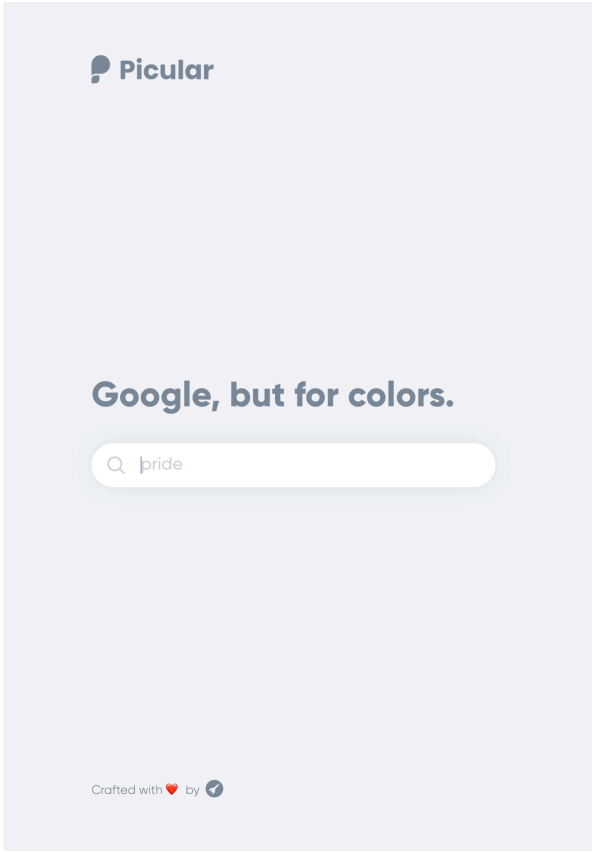
tools.medialab.sciences-po.fr/iwanthue/

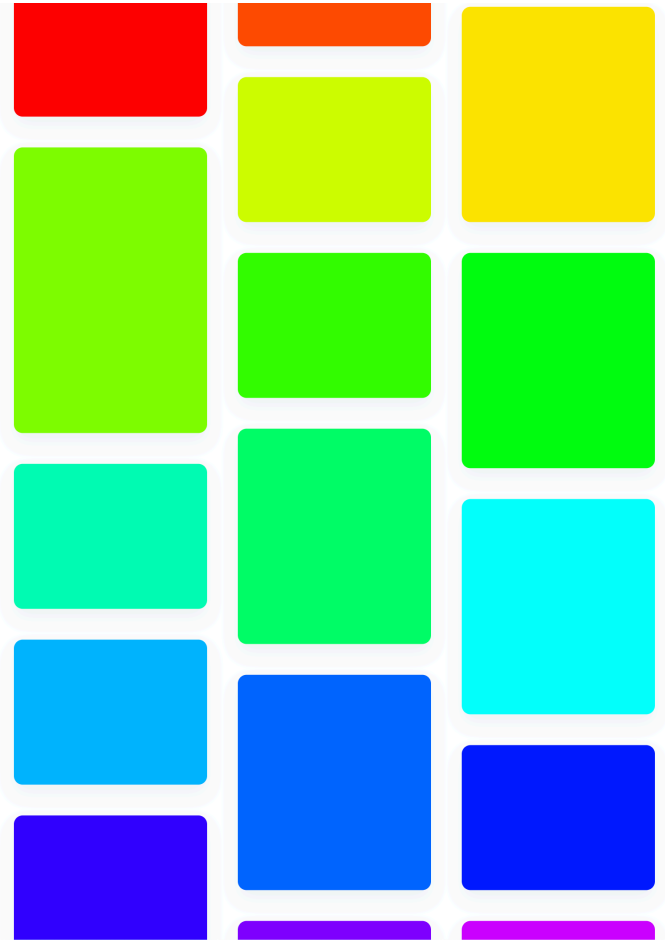


Pearson

Color Scheme Choosers

picular.co





Pearson

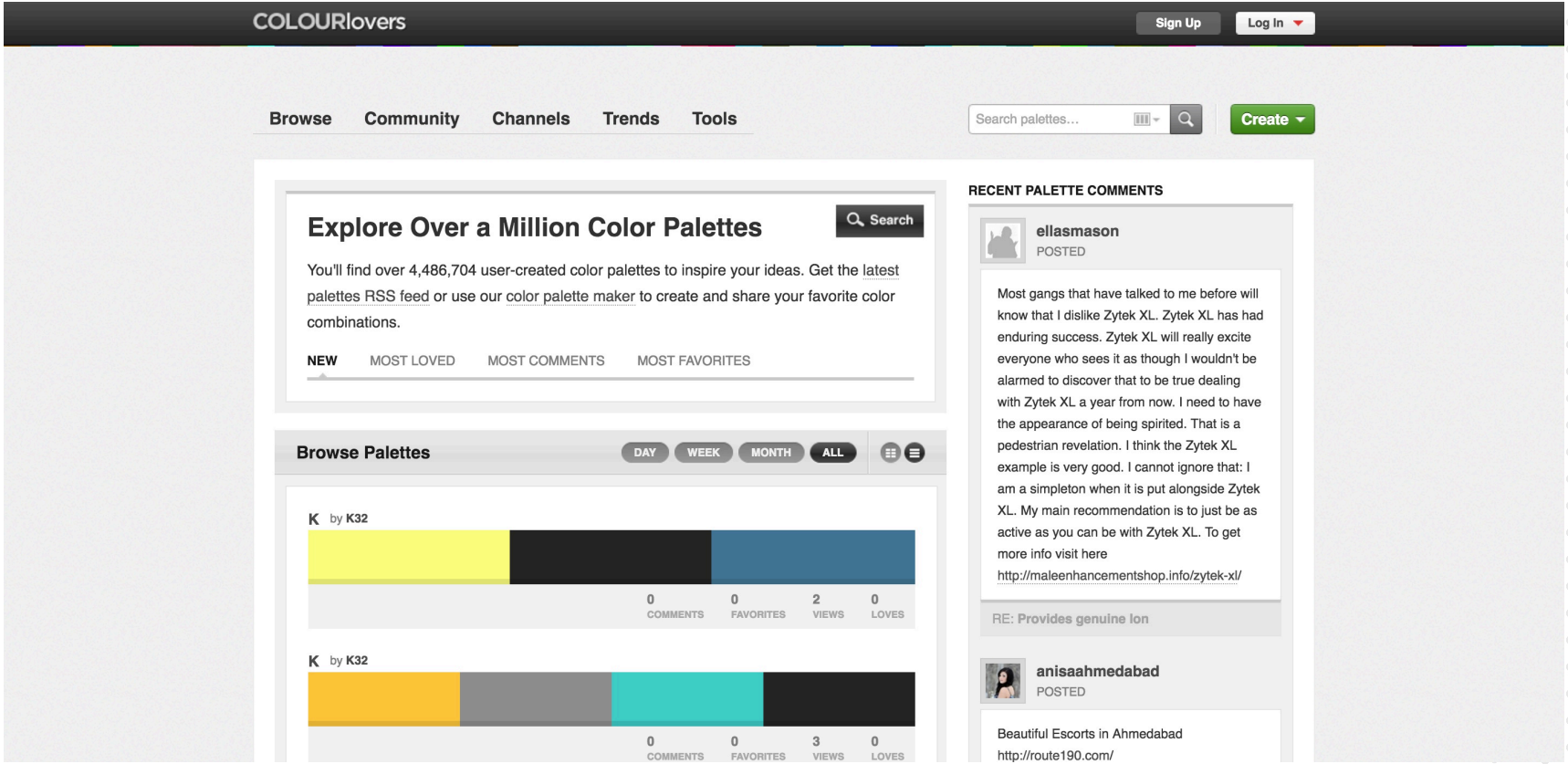
Color Scheme Choosers

www.colourco.de

Pearson

Color Scheme Choosers

www.colourlovers.com/palettes



Pearson

Color Theory

Pearson

Designmantic

Lesson 2: Analytical Design

2.1 Understand The Fundamental Principles of Analytical Design

2.2 Describe the Fundamental Tools of Visualization

2.3 Explore the Advantages and Disadvantages of Different Chart Types

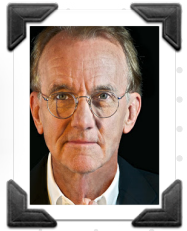
Lesson 2.1

Understand The Fundamental Principles of Analytical Design

Why Visualization?

"Information Visualization is a form of knowledge compression"
D. McCandless

"There is no such thing as information overload. There is only bad design."
E. Tufte



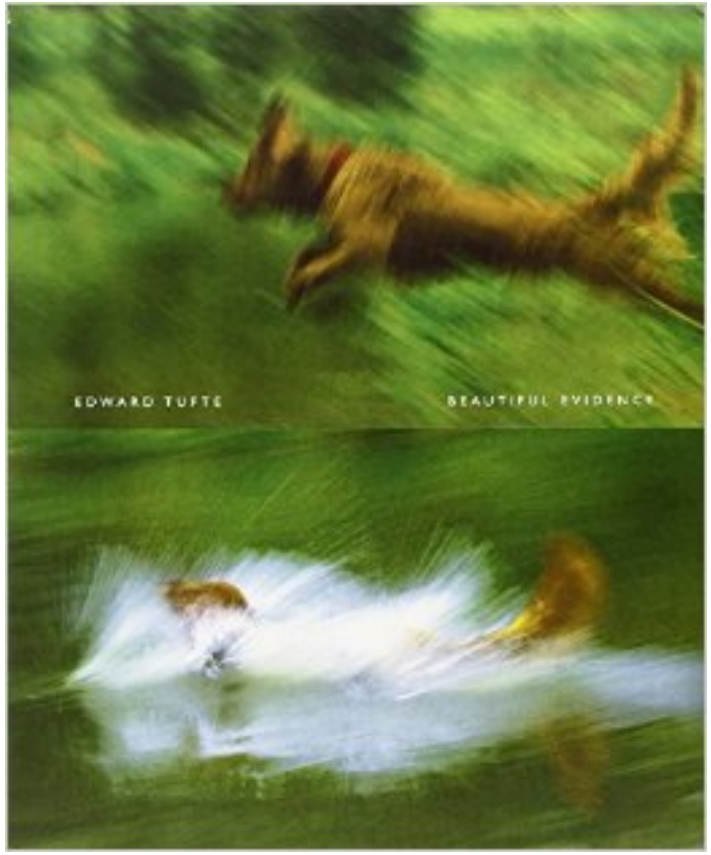
"Never leave a number all by itself. Never believe that one number on its own can be meaningful. If you are offered one number, always ask for at least one more. Something to compare it with."
H. Rosling

"The greatest value of a picture is when it forces us to notice what we never expected to see."
J. Tukey

Pearson

Fundamental Principles of Analytical Design

https://amzn.to/2TDBK9J



Pearson

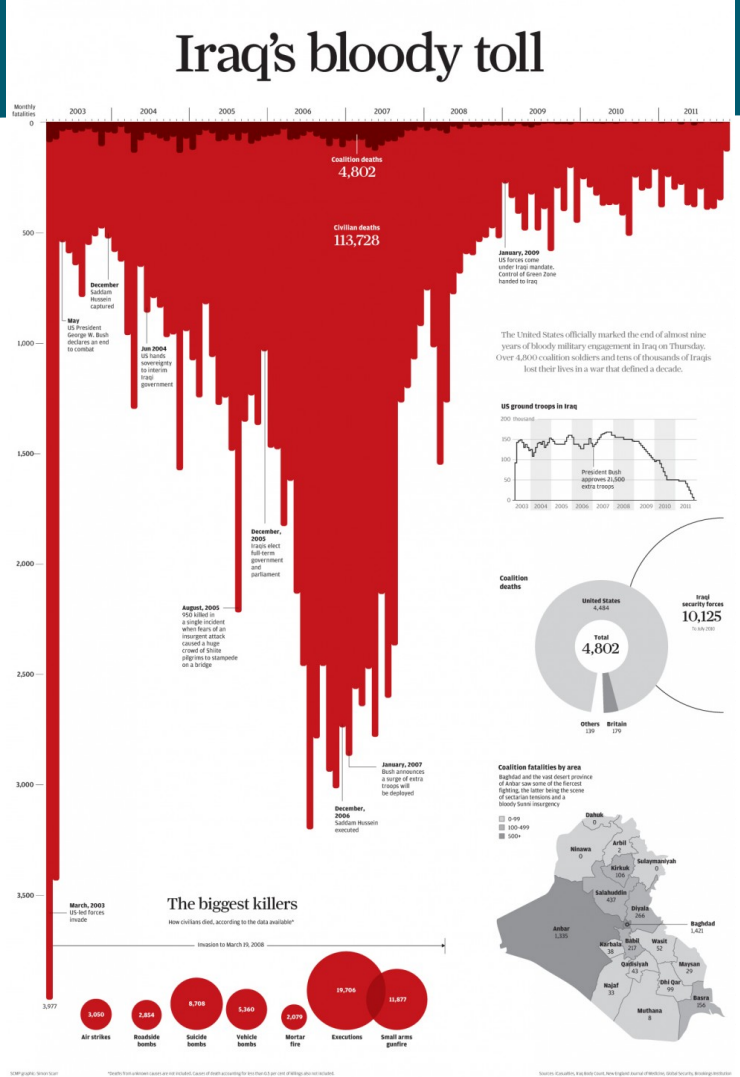
Fundamental Principles of Analytical Design

1. Show comparisons, contrasts and differences
2. Show causality, mechanism, explanation and systematic structure
3. Show multivariate data: more than one or two variables
4. Completely integrate words, numbers, images and diagrams
5. Documentation
6. Content matters most of all

"Information Visualization is a form of knowledge compression"
D. McCandless

Pearson

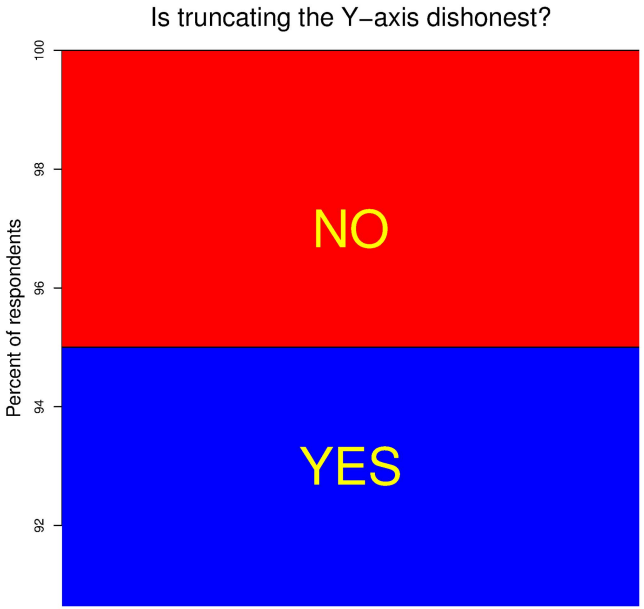
Rules can be broken…

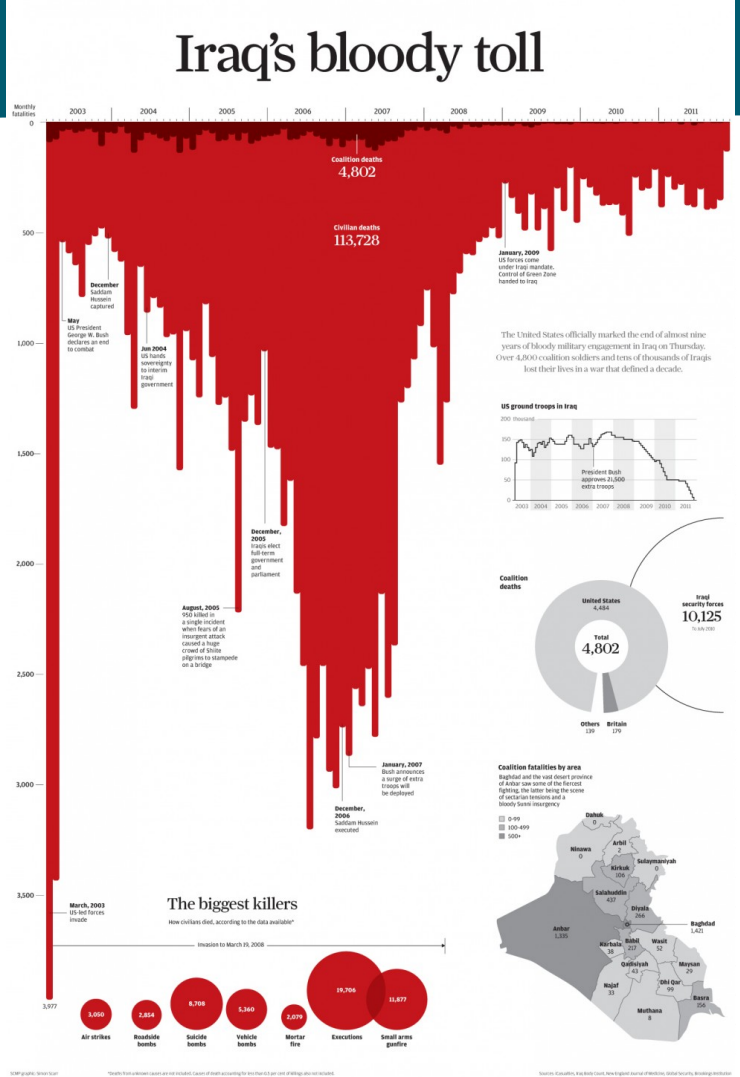


Pearson

Rules can be broken…

…sometimes

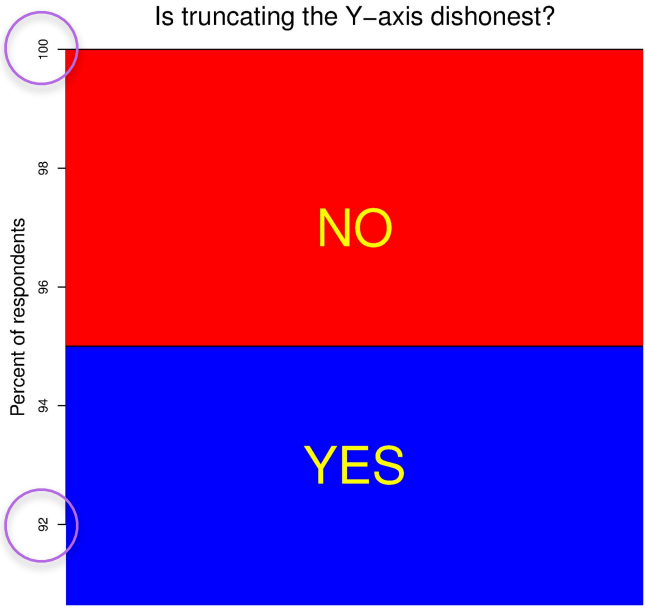


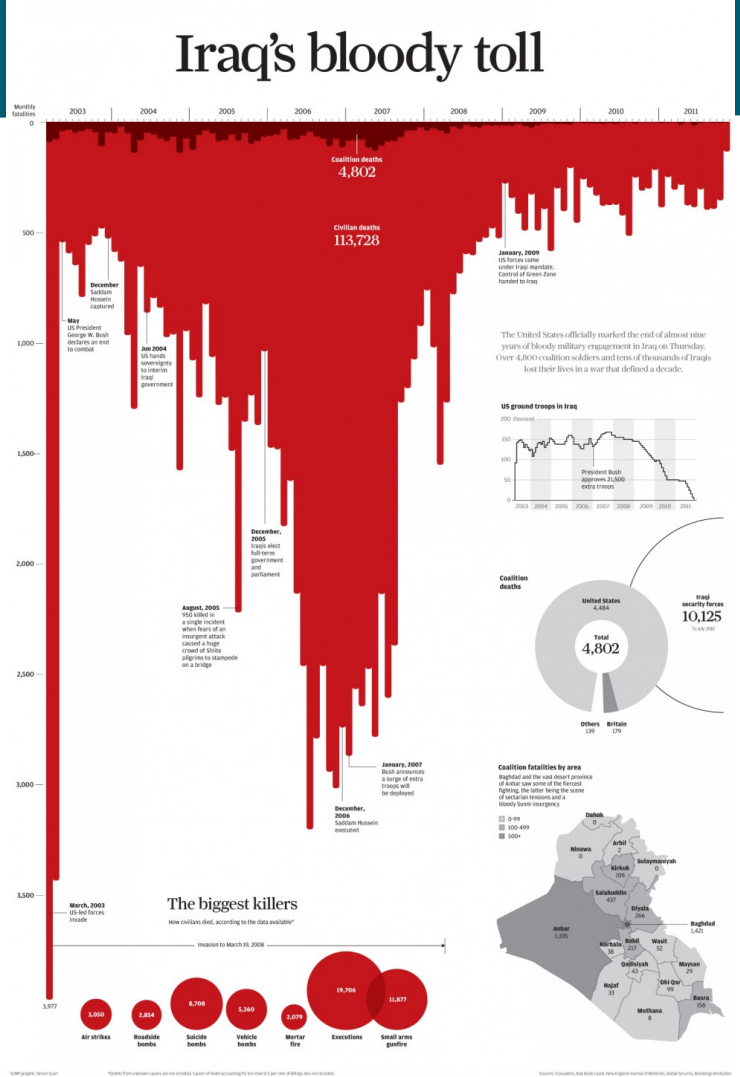


Pearson

Rules can be broken…

…sometimes





Pearson

Lesson 2.2
Fundamental Tools of Visualization

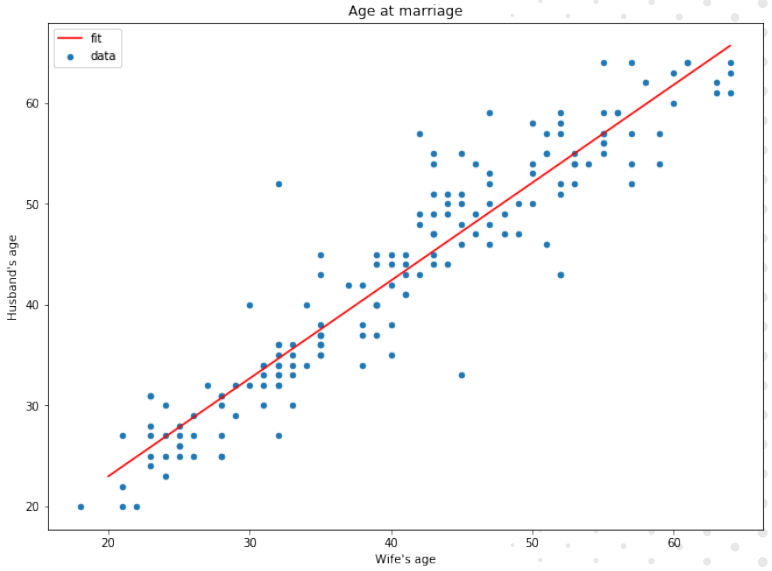
Fundamental tools

- Points
- Lines
- Areas
- Shapes
- Colors
- Text

Fundamental tools

- Points
- Lines
- Areas
- Shapes
- Colors
- Text

- Each for these can be used to encode a given variable to produce all the types of plots we are familiar with:
- Scatter plot - Just points (line)

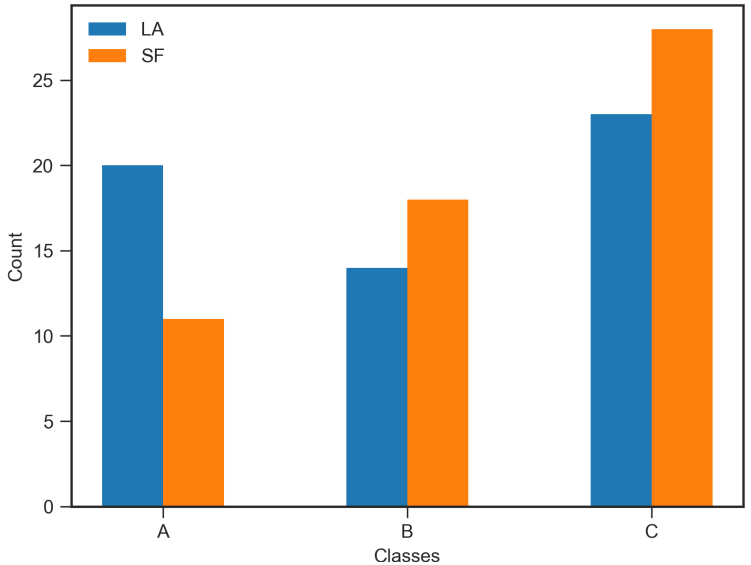


Pearson

Fundamental tools

- Points
- Lines
- Areas
- Shapes
- Colors
- Text

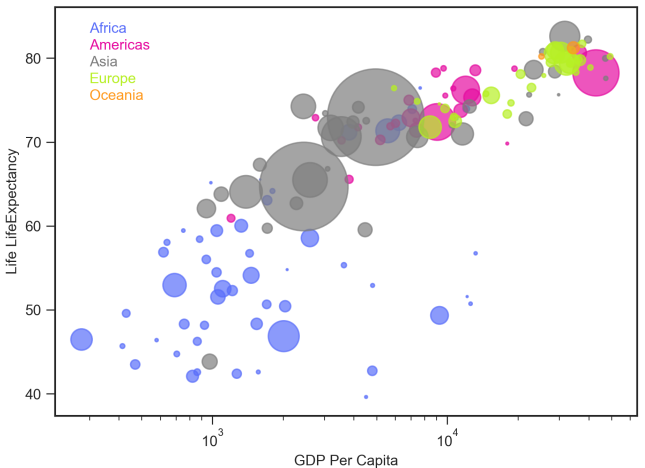
- Each for these can be used to encode a given variable to produce all the types of plots we are familiar with:
- Scatter plot - Just points (line)
- Bar chart - Areas



Fundamental tools

- Points
- Lines
- Areas
- Shapes
- Colors
- Text

- Each for these can be used to encode a given variable to produce all the types of plots we are familiar with:
- Scatter plot - Just points (line)
- Bar chart - Areas
- Bubble chart - Scatter plot + size + color (time)

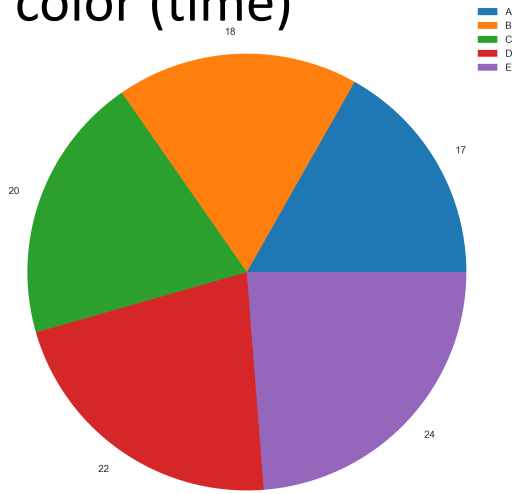


Pearson

Fundamental tools

- Points
- Lines
- Areas
- Shapes
- Colors
- Text

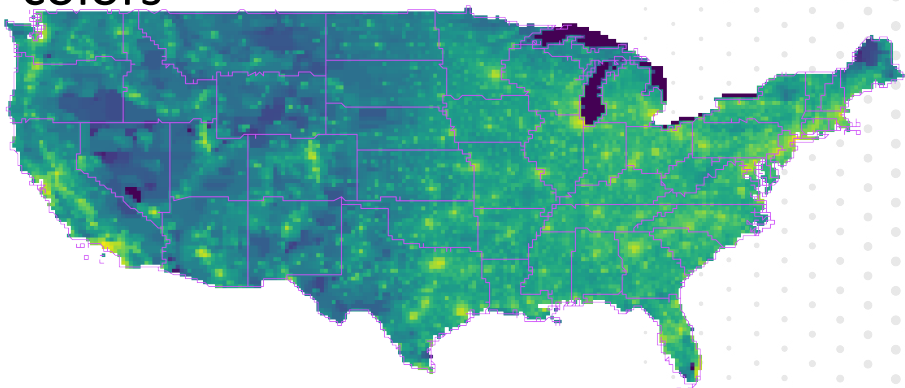
- Each for these can be used to encode a given variable to produce all the types of plots we are familiar with:
- Scatter plot - Just points (line)
- Bar chart - Areas
- Bubble chart - Scatter plot + size + color (time)
- Pie chart - Areas + colors



Fundamental tools

- Points
- Lines
- Areas
- Shapes
- Colors
- Text

- Each for these can be used to encode a given variable to produce all the types of plots we are familiar with:
- Scatter plot - Just points (line)
- Bar chart - Areas
- Bubble chart - Scatter plot + size + color (time)
- Pie chart - Areas + colors
- Heatmap - Colors



Lesson 2.3

Advantages and Disadvantages of Different Chart Types

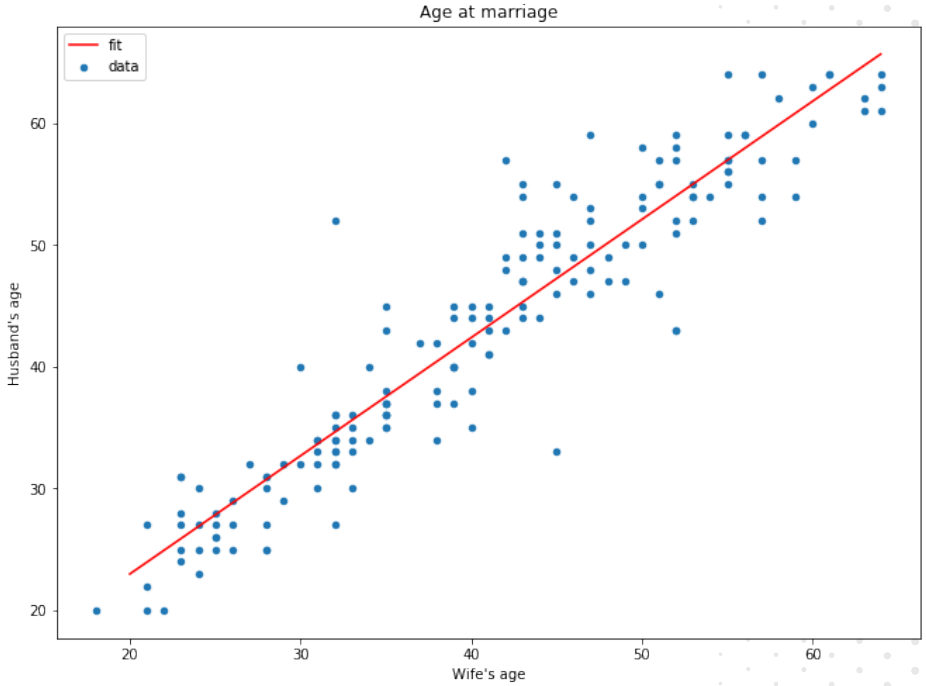
Scatter Plot

Advantages:

- Individual data points
- Highlight denser regions
- Trend Lines

Disadvantages:

- Hard to handle large numbers of points



Pearson

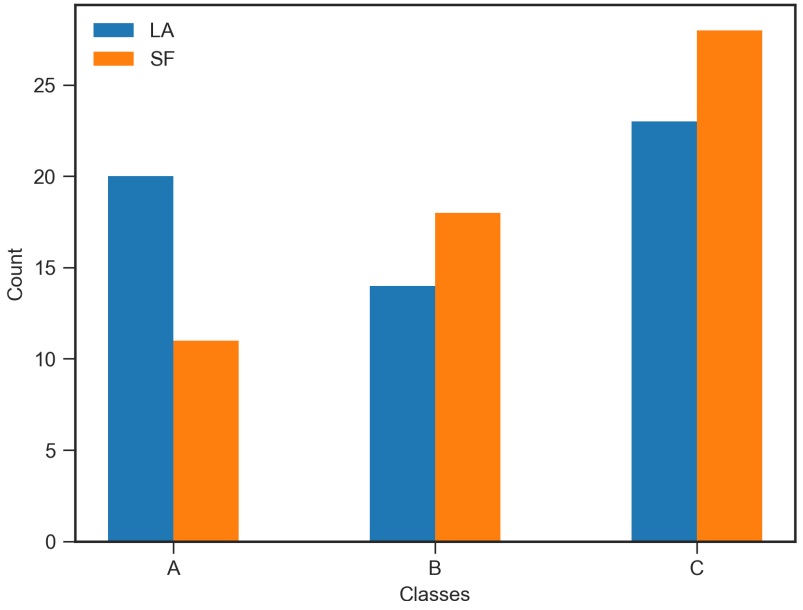
Bar Chart

Advantages:

- Easy to understand
- Compare different classes
- Reveal Trends

Disadvantages:

- Only discrete data
- Susceptible to manipulation



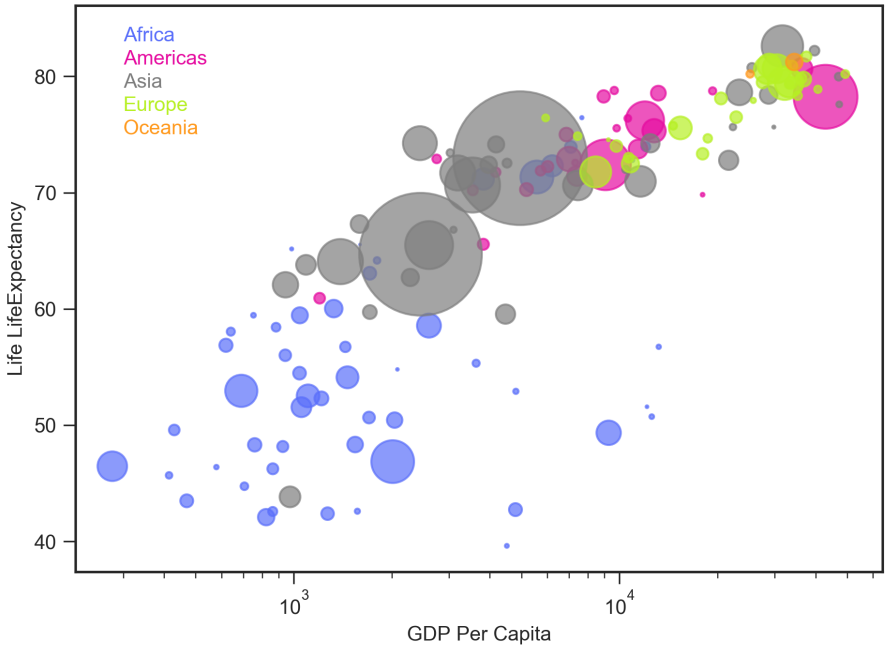
Bubble Plot

Advantages:

- Multidimensional representation
- Intuitive
- Effective for comparisons

Disadvantages:

- Overlapping issues
- Complexity



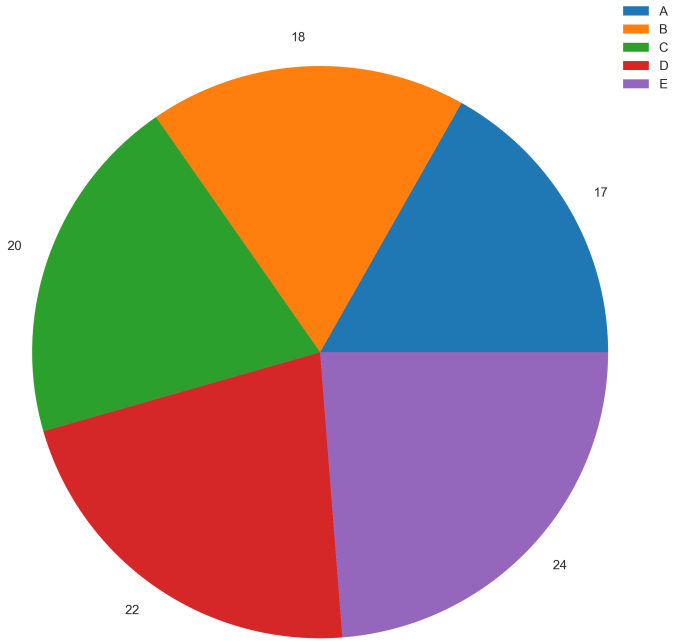
Pie Chart

Advantages:

- Intuitive
- Visually appealing
- Quick comparisons

Disadvantages:

- Imprecise
- Potential for misinterpretation



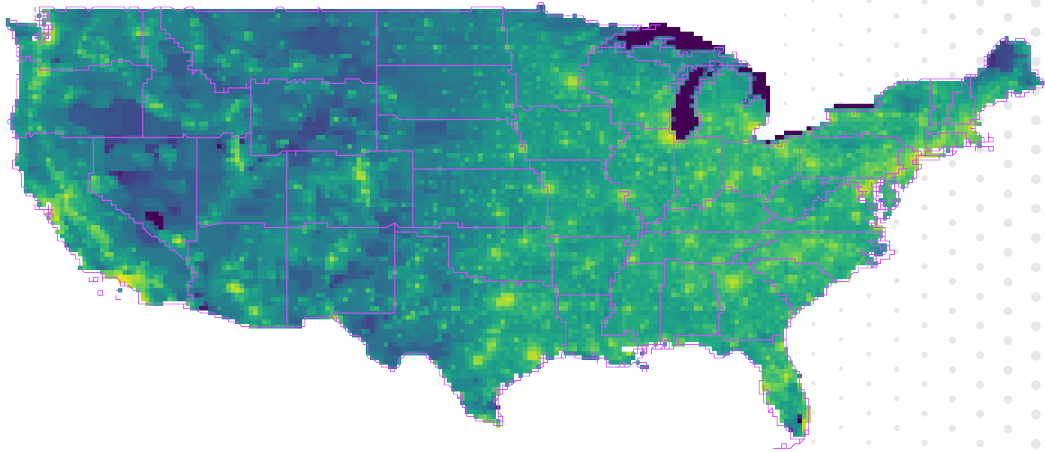
Heatmap

Advantages:

- Effective for large datasets
- Versatility
- Quick insights

Disadvantages:

- Color perception
- Oversimplification



Lesson 3. Data Cleaning and Visualization with Pandas

3.1 Data Frames and Series

3.2 GroupBy and Pivot Tables

3.3 Merge and Join

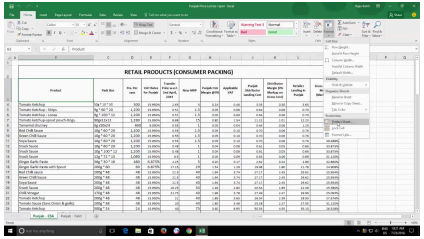
3.4 The Plot Function

3.5 Demo

Lesson 3.1
Data Frames and Series

Series and Data Frames

- A Series is conceptually equivalent to a numpy array with some extra information (column and row names, etc)
- A DataFrame can be thought of as the union of several Series, with names associated.
- Similarly, you can think of a DataFrame as an Excel spreadsheet and of a Series as an individual column



Series and Data Frames

- A minimal DataFrame implementation would be a dict where each element is a list

Series

	id
0	23
1	42
2	12
3	86

+

Series

	Name
0	“Bob”
1	“Karen”
2	“Kate”
3	“Bill”

=

DataFrame

	id	Name
0	23	“Bob”
1	42	“Karen”
2	12	“Kate”
3	86	“Bill”

Indexing and Slicing

- pandas supports various ways of indexing the contents of a DataFrame
- [<column name>] - select a given column by it's name. column names can also be used as field names
- .loc[<row name>] - select a given row by it’s (Index) name
- .iloc[<position>] - select a given row by it’s position in the Index, starting at 0

Indexing and Slicing

- .loc[<row name>, <column name>] - select an individual element by row and column name
- .iloc[<row position>, <column position>] - select an individual element by row and column position (starting at 0)
- .iloc also supports ranges and slices similarly to python lists or numpy arrays

Importing and Exporting Data

- pandas has powerful methods to read and write data from multiple sources
  - pd.read_csv() - Read a comma-separated values file
    - sep=‘ ’ - Define the separator to use. ‘,’ is the default
    - header=0 - Row number to use as column names
  - pd.read_excel() - Read an Excel file
    - sheet_name - The sheet name to load

Importing and Exporting Data

- pandas has powerful methods to read and write data from multiple sources
  - pd.read_html() - Read tabular data from a URL (or local html file)
  - pd.read_pickle() - Read a Pickle file
- Each of these functions accepts a large number of options and parameters controlling its behavior. Use help(<function name>) to explore further.

Importing and Exporting Data

- Each read_*() function has a complementary to_*() function to write out a DataFrame to disk. The to_*() functions are members of the DataFrame object

Transformed values

- We often need to apply some simple transformation to the values in our original Series. Pandas offers several methods to accomplish this goal:
  - map(func) - Map values of Series according to input correspondence.
    - func - function, dict or Series to use for mapping

Transformed values

- We often need to apply some simple transformation to the values in our original Series. Pandas offers several methods to accomplish this goal:
  - transform(func) - Transform each column/row of the DataFrame using a function. Output must have the same shape as the original
    - axis = 0 - apply function to columns
    - axis = 1 - apply function to rows

Transformed values

- We often need to apply some simple transformation to the values in our original Series. Pandas offers several methods to accomplish this goal:
  - apply(func) - Apply a function along an axis of the DataFrame
    - Similar to transform() but without the limitation of preserving the shape

Lesson 3.2
GroupBy and Pivot Tables

groupby

- Sometimes we need to calculate statistics for subsets of our data
- groupby() allows us to group data based on the values of a column
- groupby() returns a GroupBy object that supports several aggregations functions, including:
  - max()/min()/mean()/median()
  - transform()/apply()
  - sum()/cumsum()
  - prod()/cumprod()

groupby

- Aggregating functions is applied to the contents of each group
- Convenient way to generate group level statistics for arbitrary groups

Pivot Tables

- Pandas provides an extremely flexible pivot table implementation
- pd.pivot_table() - Creates a spreadsheet-style pivot table as a DataFrame.

Pivot Tables

- Four important arguments:
  - values - column to aggregate
  - index - Keys to group by on the pivot table index. If an array is passed, it is being used as the same manner as column values.
  - columns - Keys to group by on the pivot table column.
  - aggfunc - Function to use for aggregation. Defaults to np.mean()
- index, columns and aggfunc can also be lists of keys or functions to use.

Lesson 3.3
Merge and Join

merge/join

- merge() and join() allows us to perform database-style join operation by columns or indexes (rows)
- Some of the most important arguments are common to both methods:
  - on - Column(s) to use for joining, otherwise join on index. Can pass an array as the join key if not already contained in the calling DataFrame. Like an Excel VLOOKUP operation
  - how - Type of join to perform: {'left', 'right', 'outer', 'inner'}

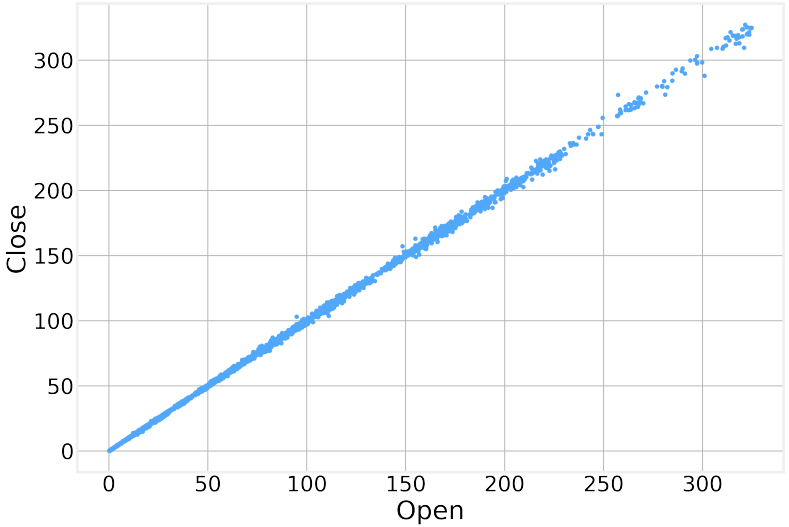
merge/join

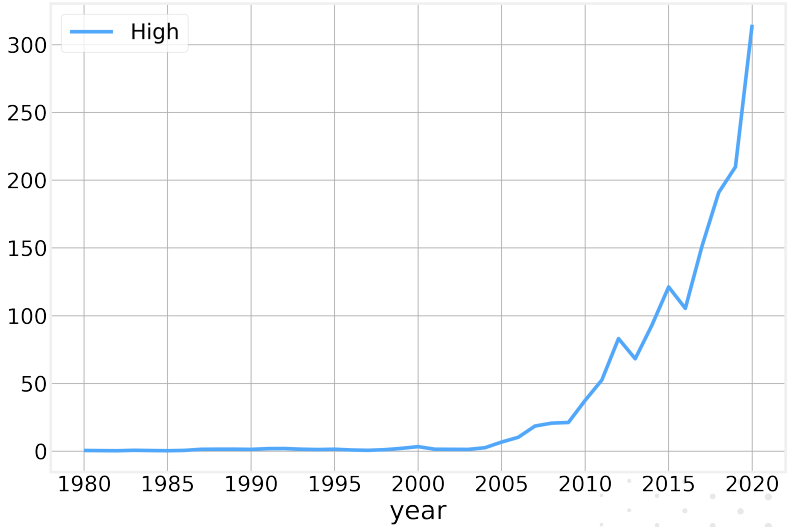
- merge() is more sophisticated and flexible. It also allows us to specify:
  - left_on/right_on - Field names to join on for each DataFrame
  - left_index/right_index - Whether or not to use the left/right index as join key(s)

Lesson 3.4
The plot function

plotting

- pandas also provides a simple interface for basic plotting through the plot() function





Pearson

plotting

- By specifying some basic parameters the variables plotted and even the kind of plot can be easily modified:
  - x/y -column name to use for the x/y axis
  - kind - type of plot
    - ‘line’ - line plot (default)
    - ‘bar’/‘barh’ - vertical/horizontal bar plot
    - ‘hist’ - histogram
    - ‘box’ - boxplot
    - ‘pie’ - pie plot
    - ‘scatter’ - scatter plot

plotting

- The plot() function returns a matplotlib Axes object
- Can easily use the full set of matplotlib functionality to customize the final plot
- The ax argument of plot() can be used to redirect the output to a pre-existing subplot.

Code - Pandas

https://github.com/DataForScience/Visualization_LL

Lesson 4. Matplotlib

4.1 Fundamental Components of a Matplotlib plot

4.2 Explore the matplotlib API

4.3 Demo

4.4 Stylesheets

4.5 Demo

4.6 Mapping

4.7 Demo

Lesson 4.1

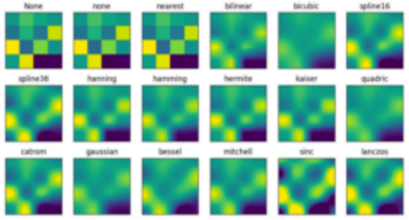
Understand the Fundamental Components of a Matplotlib plot

matplotlib

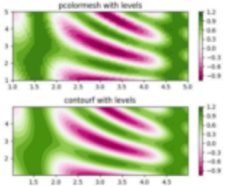
"Matplotlib is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shells, the Jupyter notebook, web application servers, and four graphical user interface toolkits."

"Matplotlib tries to make easy things easy and hard things possible. You can generate plots, histograms, power spectra, bar charts, errorcharts, scatterplots, etc., with just a few lines of code."

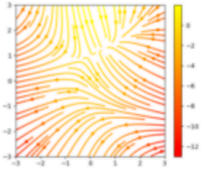
matplotlib



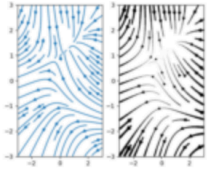
interpolation_methods



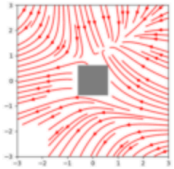
pcolormesh_levels



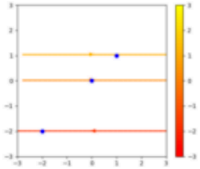
streamplot_demo_features



streamplot_demo_features



streamplot_demo_masking

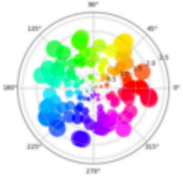


streamplot_demo_start_points

Pie and polar charts

pie_demo_features

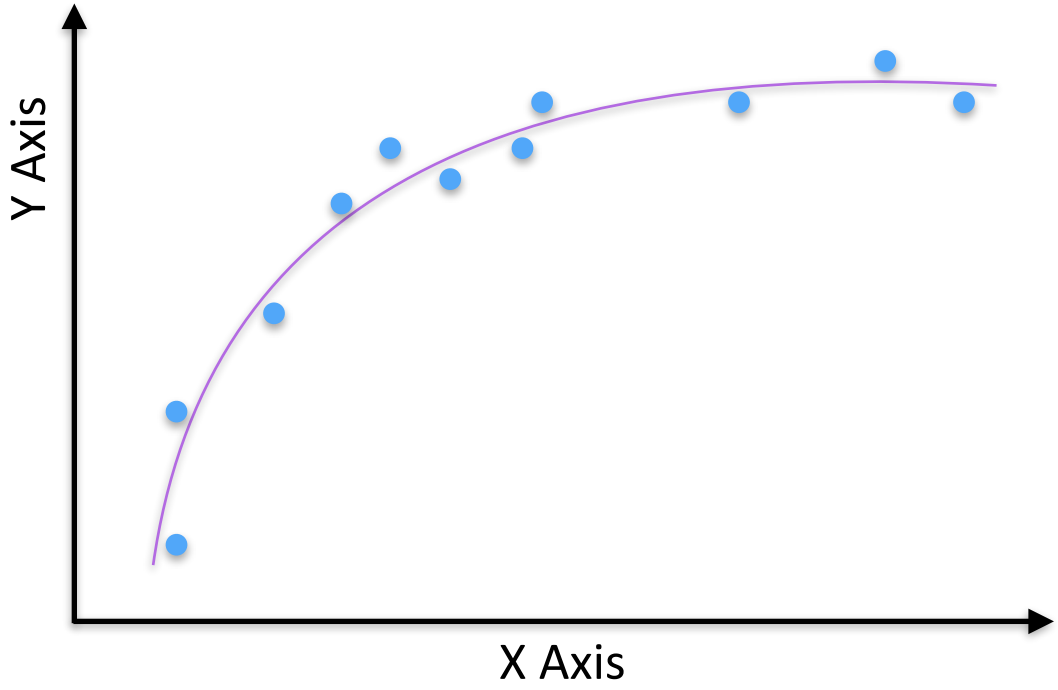
polar_bar_demo



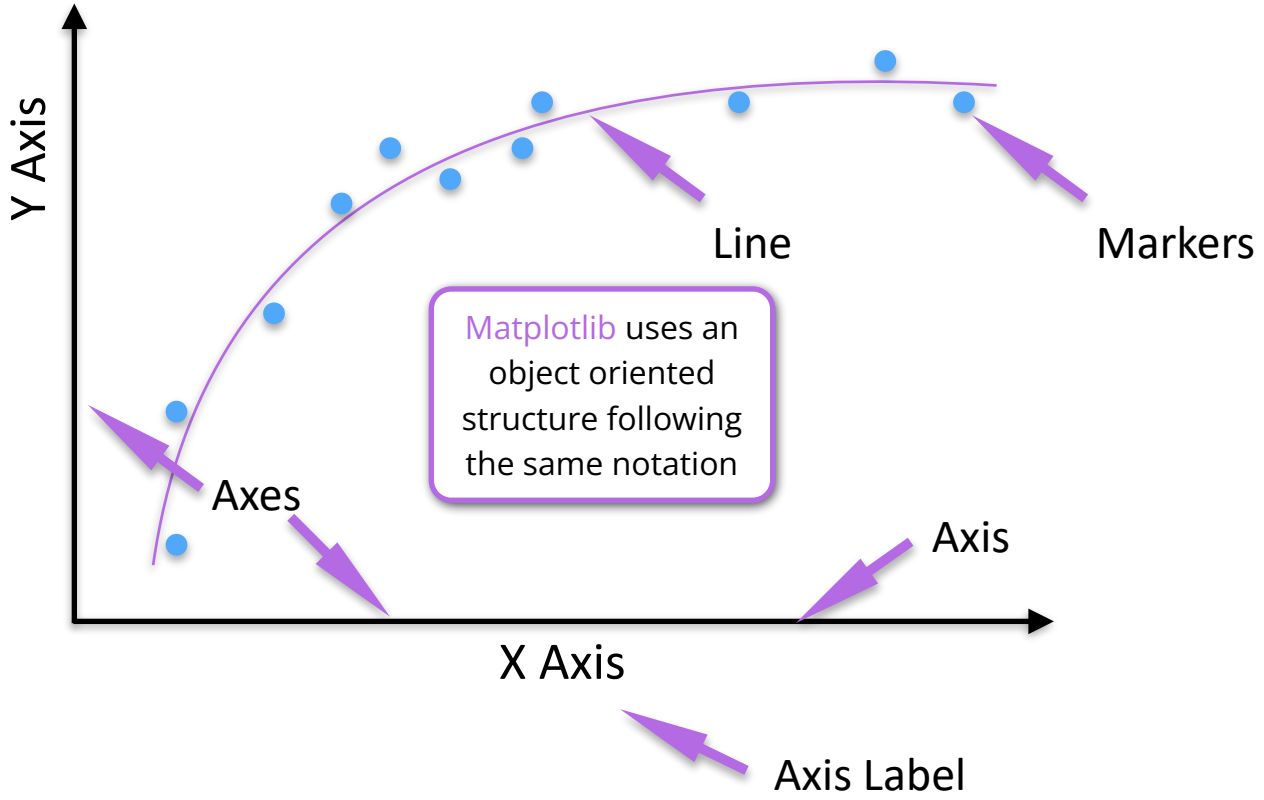
polar_scatter_demo

Pearson

Basic Plotting

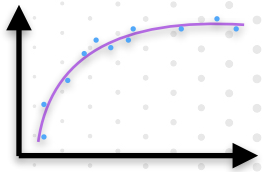


Basic Plotting



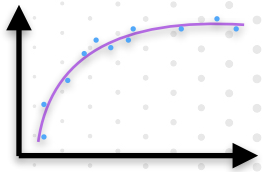
Basic Plotting

- Matplotlib uses an object oriented structure following an intuitive notation
- Each Axes object contains one or more Axis objects.
- A Figure is a set of one or more Axes.
- Each Axes is associated with exactly one Figure and each set of Markers is associated with exactly one Axes.

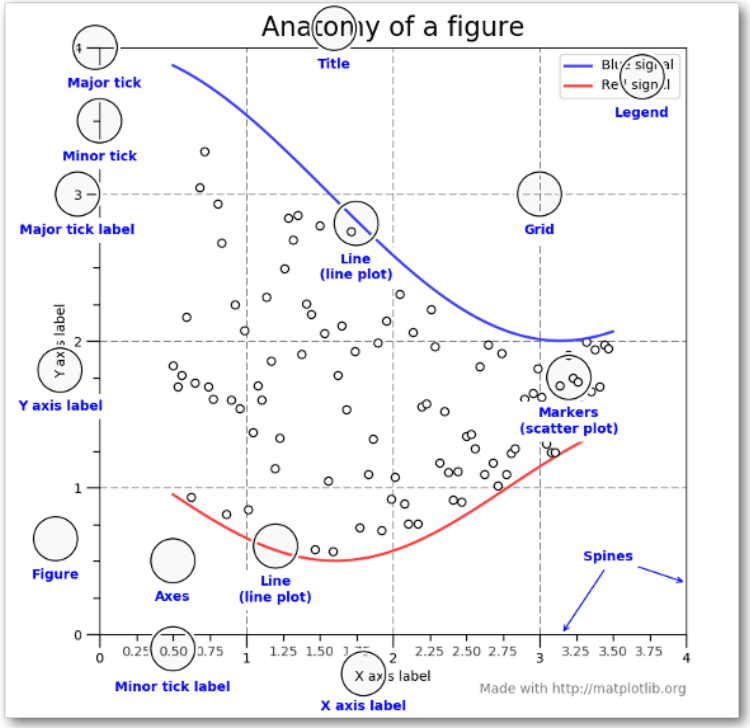


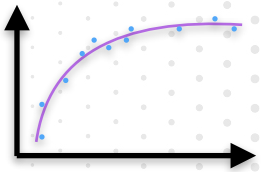
Basic Plotting

- In other words, Markers Lines represent a dataset that is plotted against one or more Axis. An Axes object is (effectively) a subplot of a Figure.



Basic Plotting - Programmatically!





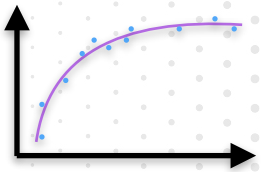
Pearson

Lesson 4.2
Explore the matplotlib API

Basic Plotting - Programmatically!

- While the Figure object controls the way in which the figure is displayed.
  - .gca() - Get the current Axes, creating one if necessary
  - .show() - Show the final figure
  - .savefig("filename.ext", dpi=300) - Save the figure to "filename.ext" where ".ext" defines the format the saved image.

```
filetypes = {'ps': 'Postscript', 'eps': 'Encapsulated Postscript', 'pdf': 'Portable Document Format',
'pgf': 'PGF code for LaTeX', 'png': 'Portable Network Graphics', 'raw': 'Raw RGBA bitmap', 'rgba': 'Raw
RGBA bitmap', 'svg': 'Scalable Vector Graphics', 'svgz': 'Scalable Vector Graphics', 'jpg': 'Joint
Photographic Experts Group', 'jpeg': 'Joint Photographic Experts Group', 'tif': 'Tagged Image File
Format', 'tiff': 'Tagged Image File Format'}
```

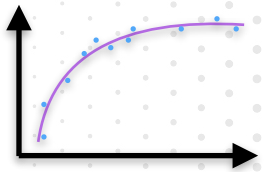


Basic Plotting - Programmatically!

- The first step is to import the pyplot module from matplotlib and instanciating a Figure object:

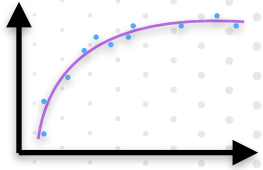
```
import matplotlib.pyplot as plt
fig = plt.figure()
```

- The convention is to import pyplot as plt
- To create subplots (Axes) you use .subplots(nrows, ncols, sharex=False, sharey=False) instead of .figure().
- Set sharex and/or sharey to True to keep the same scale in both cases.



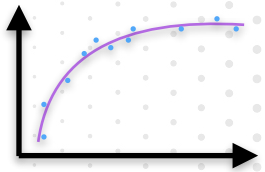
Basic Plotting - Programmatically!

- .subplots - returns a (fig, ax_lst) tuple where ax_lst is a list of Axes and fig is the Figure.
- Axes have several methods of interest:
  - .plot(x y) - Make a scatter or line plot from a list of x, y coordinates.
  - .imshow(mat) - Plot a matrix as if it were an image. Element 0,0 is plotted in the top right corner.
  - .bar(x y) - Make a bar plot where x is a list of the lower left coordinates of each bar and y is the respective height.

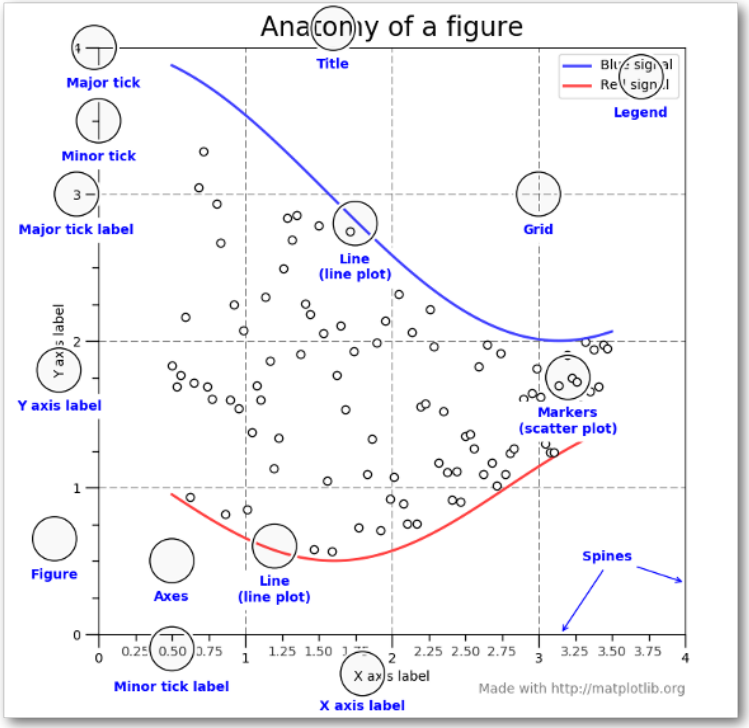


Basic Plotting - Programmatically!

- .subplots - returns a (fig, ax_lst) tuple where ax_lst is a list of Axes and fig is the Figure.
- Axes have several methods of interest:
  - .pie(values, labels=labels) - Produce a pie plot out of a list of values list and labeled with labels
  - .savefig(filename) - Write the current figure as an static image



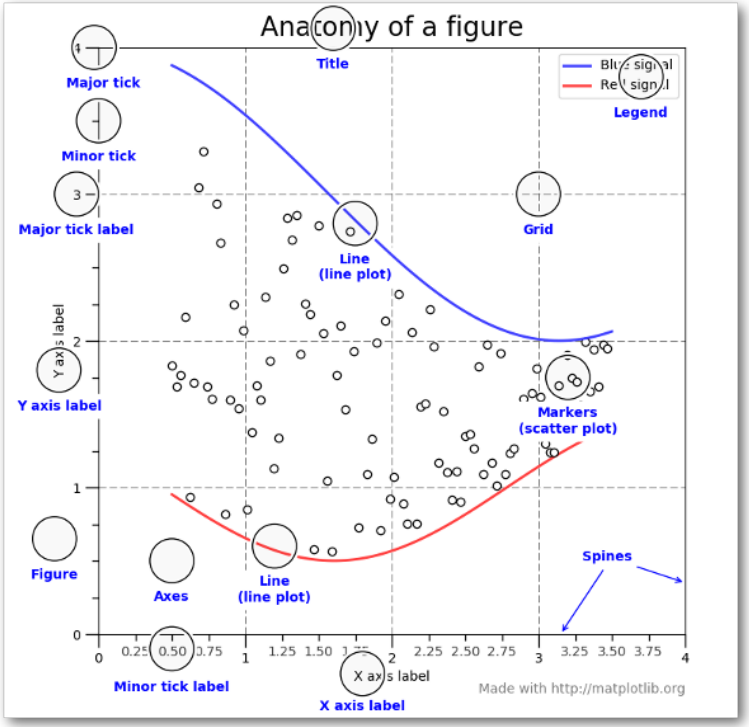
decorations



- The respective functions are named in an intuitive way. Every Axes object has as methods:
  - .set_xlabel(label)
  - .set_ylabel(label)
  - .set_title(title)

Pearson

decorations



- And axis limits can be set using:
  - .set_xlim(xmin, xmax)
  - .set_ylim(ymin, ymax)
- Tick marks and labels are set using:
  - .set_xticks(ticks)/.set_yticks(ticks)
  - .set_xticklabels(labels)/.set_yticklabels(labels)

Images

- ax.imshow(img) - Display an image on a set of axes.
- ax.imshow(img, extent=(xllcorner, xurcorner, yllcorner, yurcorner), zorder=-1)
- img can be any matrix of numbers.
- Further plotting can occur by simply using the functions described above

Code - Matplotlib

https://github.com/DataForScience/Visualization_LL

Lesson 4.4
Stylesheets

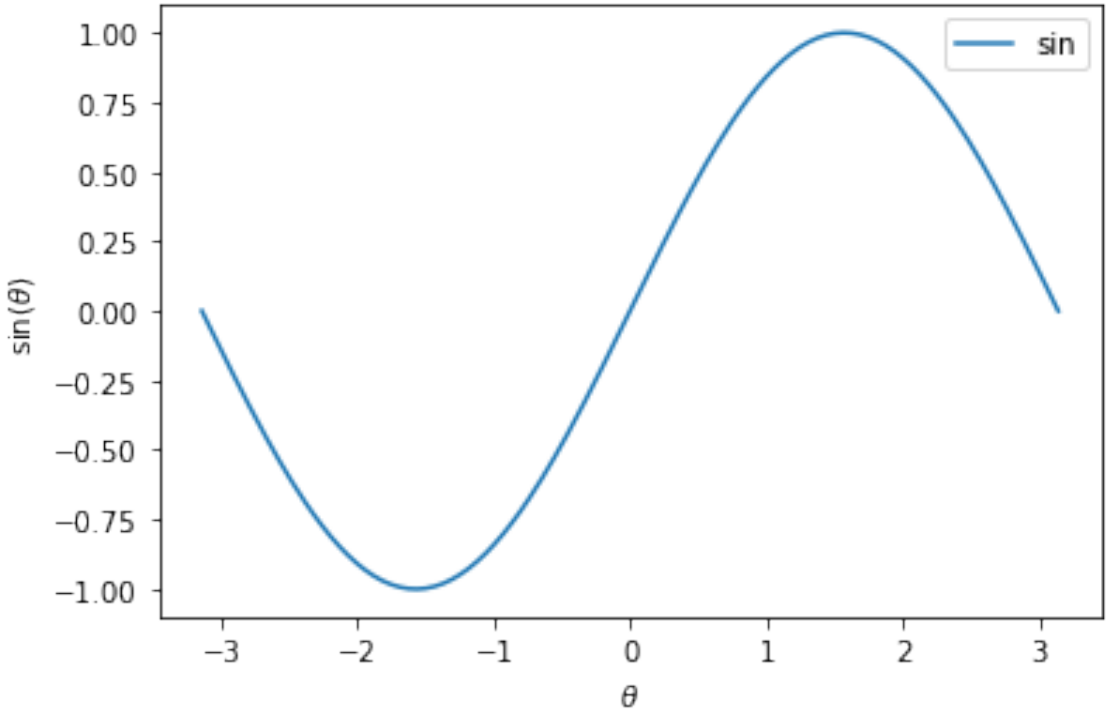
matplotlib style sheet

- Manually customizing each figure can become cumbersome
- style sheets allow us to define our own figure defaults that will apply to every figure from then on.
- matplotlib comes with several pre-defined styles
- plt.style.available returns a list of all available styles
- style sheets are simple text files with .mplstyle extension

matplotlib style sheet

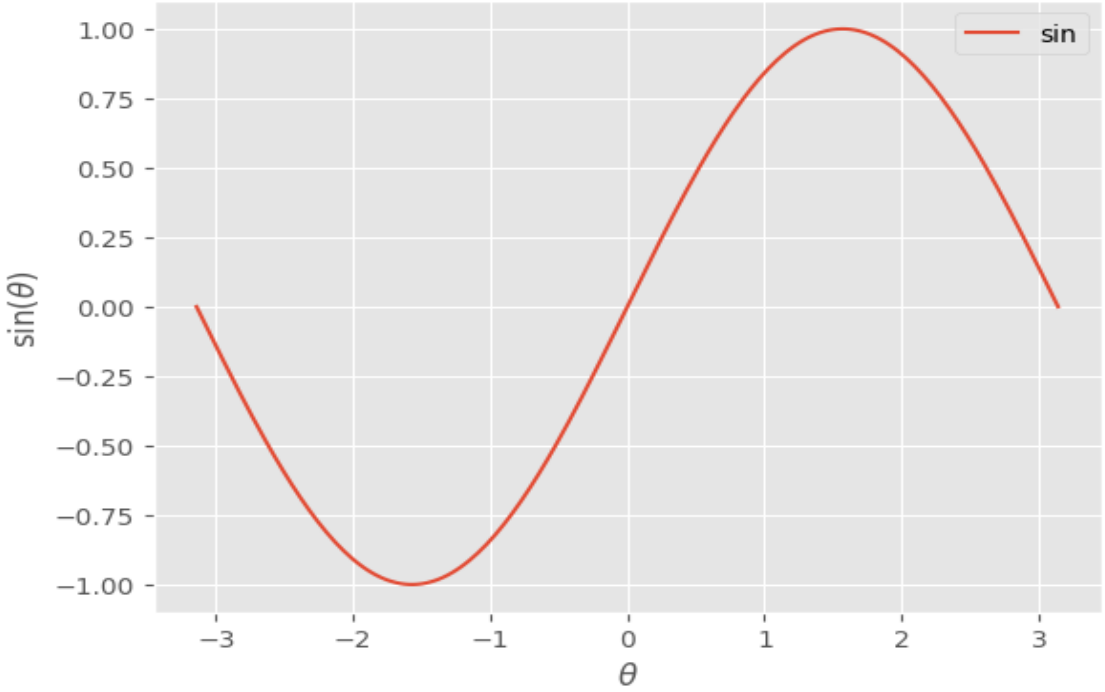
- The default directory for custom style sheets is: <config_directory>/stylelib and your personal config directory is returned by matplotlib.get_configdir()
- The contents of all style files known to matploltib are available as the matplotlib.style.library dictionary keyed by the style name

default style



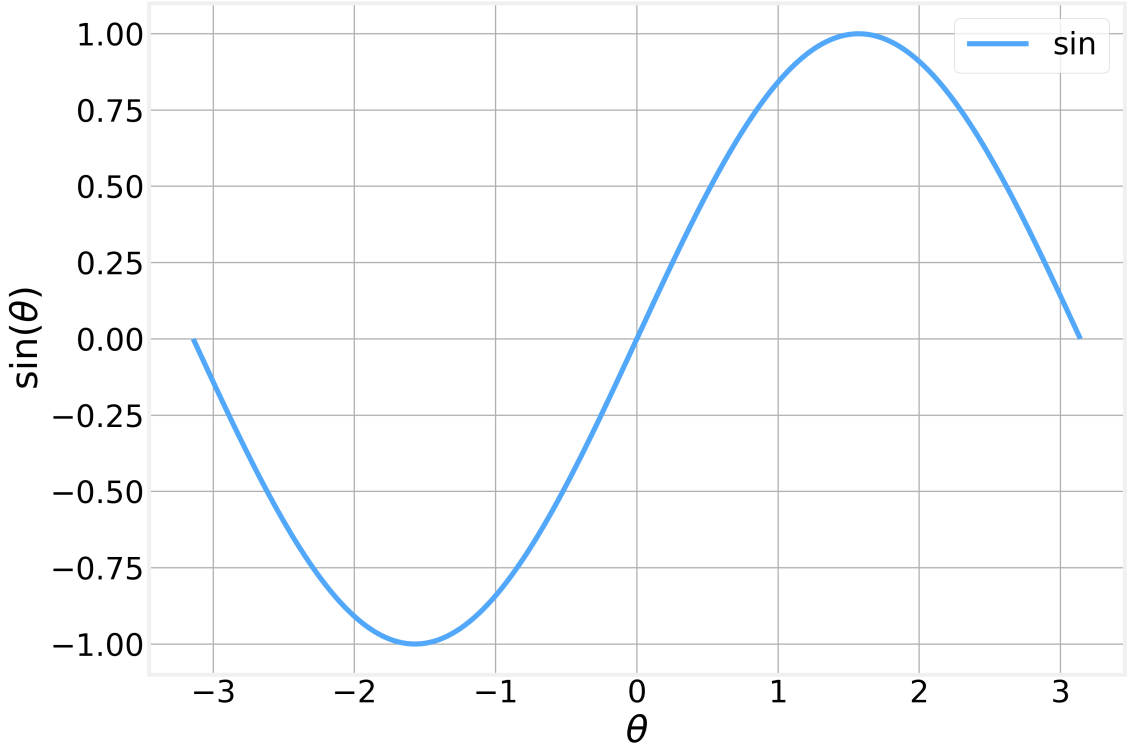
fivethirtyeight style

ggplot style



Pearson

d4sci style



Pearson

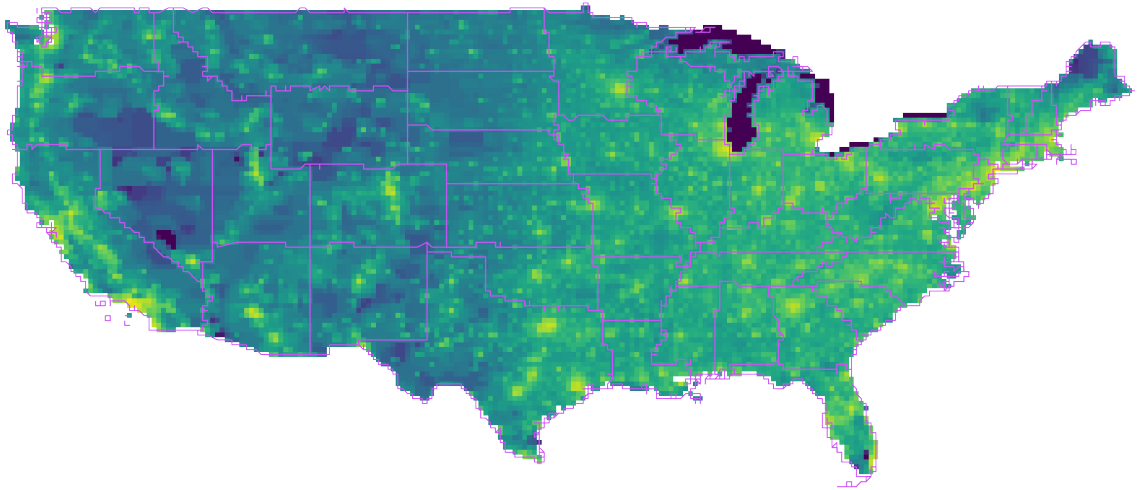
Code - Style sheets

https://github.com/DataForScience/Visualization_LL

Lesson 4.6
Mapping

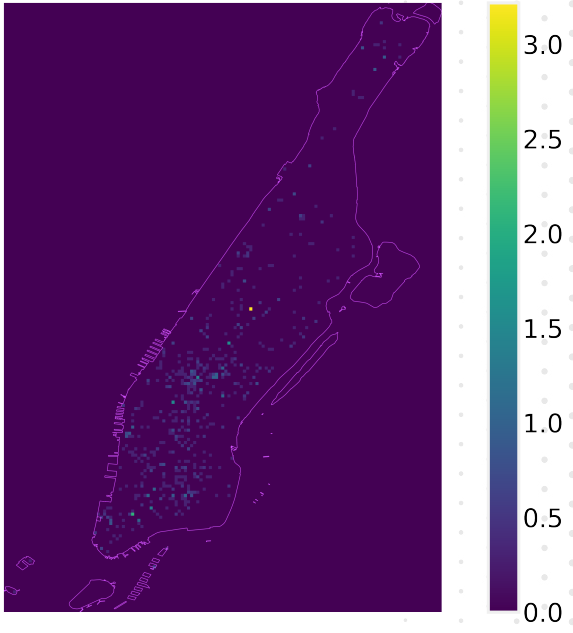
Heatmap

- Data stored as a numerical matrix
- Each matrix cell corresponds to a specific lat/lon
- Color represents cell value.



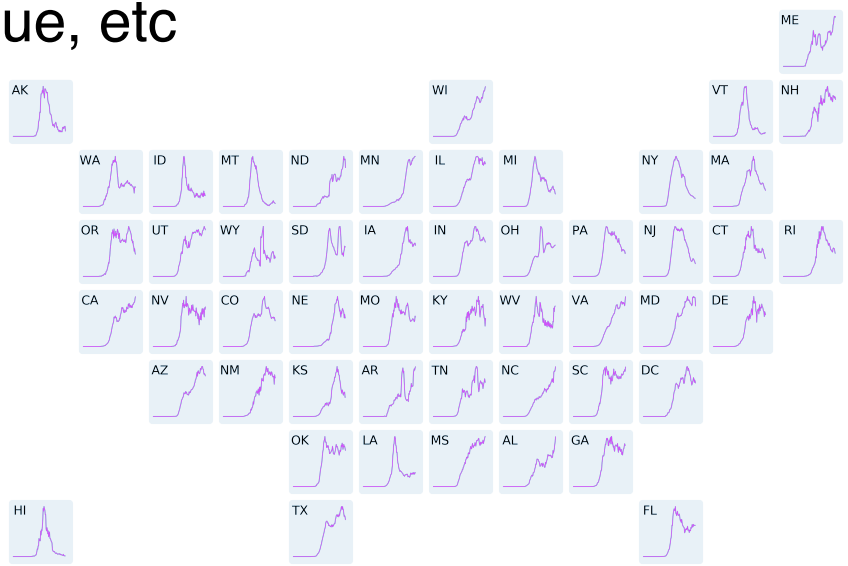
Heatmap

- Data stored as a numerical matrix
- Each matrix cell corresponds to a specific lat/lon
- Color represents cell value
- Lines and decorations can be added on top
- A color bar can be used to label the values



Block Map / Small multiples

- Schematic representation of the relationship between geographical areas
- Each state “box” can contain a subplot, be colored by a specific value, etc



Pearson

Cartopy

- The cartopy module adds functionality to matplotlib
- The minimal map is simply:

```
import matplotlib.pyplot as plt
import cartopy.crs as ccrs

ax = plt.axes(projection=ccrs.PlateCarree())
ax.coastlines()
```

- Note that without the .coastlines() call nothing is plotted as our map has no content
- The projection argument is necessary to specify the kind of projection we want to use.

Cartopy

- cartopy returns a GeoAxesSubplot object with all the usual Axes object functionality plus a series of cartopy specific methods:

```
'add_feature',          'img_factories',
'add_geometries',       'natural_earth_shp',
'add_raster',           'outline_patch',
'add_wms',              'projection',
'add_wmts',             'read_user_background_images',
'background_img',       'set_boundary',
'background_patch',     'set_extent',
'coastlines',           'set_global',
'get_extent',           'stock_img',
'gridlines',            'tissot'
'hold_limits',
```

Cartopy

Cartopy

- We can also visualize just specific regions by setting the bbox and center coordinates by setting

  x0, x1, y0, y1

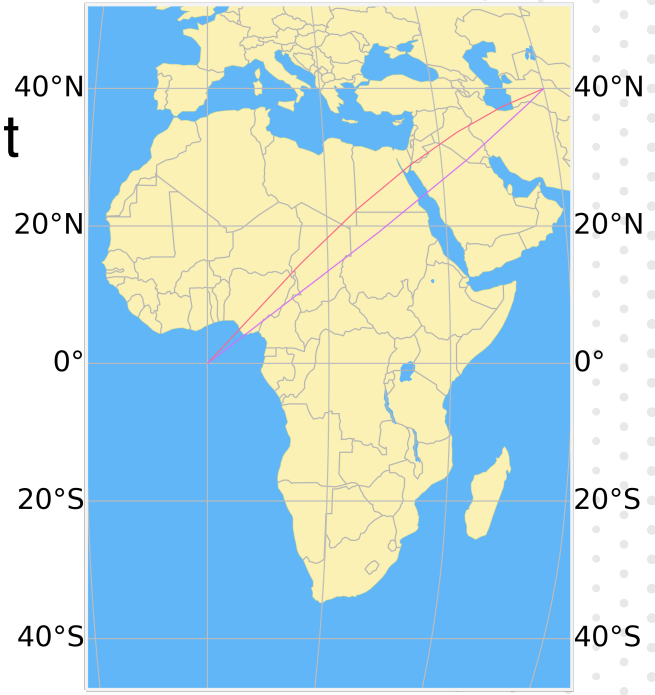
- using ax.set_extent()
- We also have complete control over all the properties of the Figure and Axes objects, allowing us to draw fairly sophisticated maps in a variety of projections

Cartopy

- We can add features with add_feature() and the constants defined in cartopy.feature
  - OCEAN
  - LAND
  - LAKES
  - BORDERS
  - RIVERS
  - COASTLINE
  - STATES
  - etc…

Cartopy

- We can plot arbitrary data within the map using the usual ax.plot() method and the appropriate transform argument
  - ccrs.Geodetic() - follow the great circles
  - ccrs.PlateCarree() - draw a straight line



Cartopy

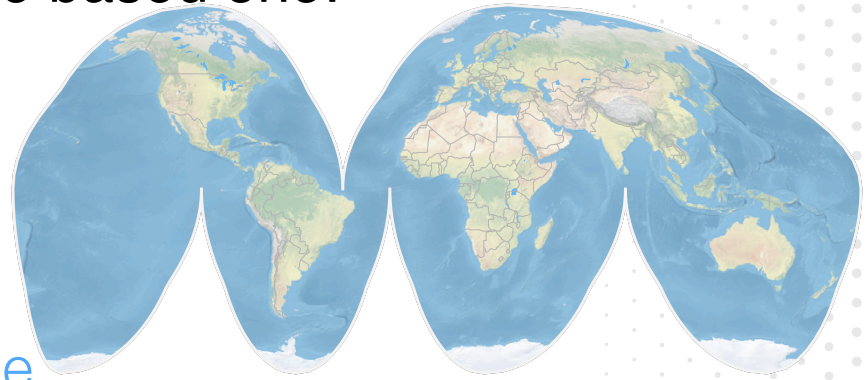
- To add a sock image as a background we can use ax.stock_img()
- Arbitrary images can be loaded with ax.imshow() and the appropriate transform argument

Cartopy

- To switch to a globe representation, all we have to do is to change our projection to a globe based one:
  - Mollweide
  - Orthographic
  - Robinson
  - InterruptedGoodeHomolosine
  - etc…

Cartopy

- To switch to a globe representation, all we have to do is to change our projection to a globe based one:
  - Mollweide
  - Orthographic
  - Robinson
  - InterruptedGoodeHomolosine
  - etc…



Code - Mapping

https://github.com/DataForScience/Visualization_LL

Lesson 5 - Matplotlib Animations

5.1 Matplotlib Animation API

5.2 Func Animation

5.3 Animation Writers

5.4 Demo

Lesson 5.1
Matplotlib Animation API

The matplotlib Animation API

- matplotlib supports simple animations through the Animation API
- Two main Animation classes:
  - FuncAnimation - generate an animation by repeatedly calling a given function func() to draw each frame.
  - ArtistAnimation - creates an animation by using a fixed set of Artist objects.
- FuncAnimation is the simplest and most intuitive approach, so it's the approach we will follow here

The matplotlib Animation API

- Animations can be displayed in a pop-up window, a notebook cell, or saved as a movie or GIF file.

Lesson 5.2
FuncAnimation

FuncAnimation

- FuncAnimation takes several important parameters:
  - fig - a handle to the figure object being used
  - func - the function that plots each frame. This function should take the frame number as the first argument.
  - frames=None - the number of frames or a list of frames to generate
  - interval - number of milliseconds between frames

FuncAnimation

- FuncAnimation takes several important parameters:
  - init_func=None - a function that creates the initial, empty, figure. Must return a list of Artist objects that will be updated in each frame
  - fargs=None - Any further parameters required by func()

FuncAnimation

- Instantiating the FuncAnimation object returns an animation object that we can use to generate the animation.
- Before we can watch the animation, we must call a writer method

```
fig = plt.figure()
ax = plt.axes(xlim=(0, 4), ylim=(-2, 2))
line, = ax.plot([], [], lw=3)

def init():
    line.set_data([], [])
    return line,

def animate(i):
    x = np.linspace(0, 4, 1000)
    y = np.sin(2 * np.pi * (x - 0.01 * i))
    line.set_data(x, y)
    return line,

anim = FuncAnimation(fig, animate, init_func=init,
                                frames=200, interval=20, blit=True,
                                repeat=True)
```

Lesson 5.3
Animation Writers

Animation writers

- There are three possible writer methods:
  - Animation.save() - Save the animation as a movie, frame by frame
    - The file format can be specified by using the file extension

```
15 # Save the animation as a GIF
16 anim.save('data/vulcano.gif')
```

Animation writers

- There are three possible writer methods:
  - Animation.save() - Save the animation as a movie, frame by frame
    - A more robust alternative is to provide a FFMpegWriter instance, which supports a wider range of formats

```
15 writervideo = FFMpegWriter(fps=60)
16 anim.save("data/vulcano.mp4", writer=writervideo)
```

    - This might require installing ffmpeg separately (https://ffmpeg.org/download.html)

Animation writers

- There are three possible writer methods:
  - Animation.to_jshtm() - Generate HTML representation of the animation.
  - Animation.to_html5_video() - Convert the animation to an HTML5 <video> tag with the video encoded directly inside the tag. This is ideal for integrating into Jupyter notebooks

Animation.to_html5_video()

- Embedding the animation within the notebook is a three step process:
  - Generate the html5 video using Animation.to_html5_video()
  - Generate the html embedding using jupyters display.HTML()
  - Display the video by calling display.display()

```
20 # Display the animation as an HTML video
21 video = anim.to_html5_video()
22 html = display.HTML(video)
23 display.display(html)
```

Code - matplotlib Animations

https://github.com/DataForScience/Visualization_LL

Lesson 6: Jupyter Widgets

6.1 ipywidgets as interactive browser controls

6.2 Simple widget use

6.3 Widget customization

6.4 Demo

Lesson 6.1

ipywidgets as interactive browser controls

ipywidgets

- Jupyter Widgets are interactive browser controls for Jupyter notebooks
- Widgets come in a wide range of forms:
  - Numeric/Boolean/String widgets
  - Selection widgets
  - Image
  - Button
  - Output
  - Date/Color pickers
  - File Upload

ipywidgets

- Interactive widgets allow users to visualize and manipulate data in intuitive and easy ways.
- The simplest way to use the widgets is to use the ipywidgets.interact() function decorator

Lesson 6.2
Simple widget use

Simple widget use

- @ipywidgets.interact - automatically generates UI controls based on the arguments (and default values) of the function to which it is applied
  - string - Text
  - boolean - Checkbox
  - integer/float - IntSlider/FloatSlider
  - list/dict - Dropdown - The dict allows the dropdown box to display one value (the key) while passing a different value to the function (the value)

Simple widget use

- Fixed arguments to the function can be specified with the fixed() function as the default value.
- The return value of the function will be displayed automatically, making it particularly useful for functions that produce visualizations
- @ipywidgets.interact_manual - is similar to @ipywidgets.interact but it adds an extra Button to execute the function instead of automatically executing it as soon as any of the parameters change

Simple widget use

- Adding the @widgets.interact decorator directly above the function definition produces a simple GUI to manipulate the function arguments

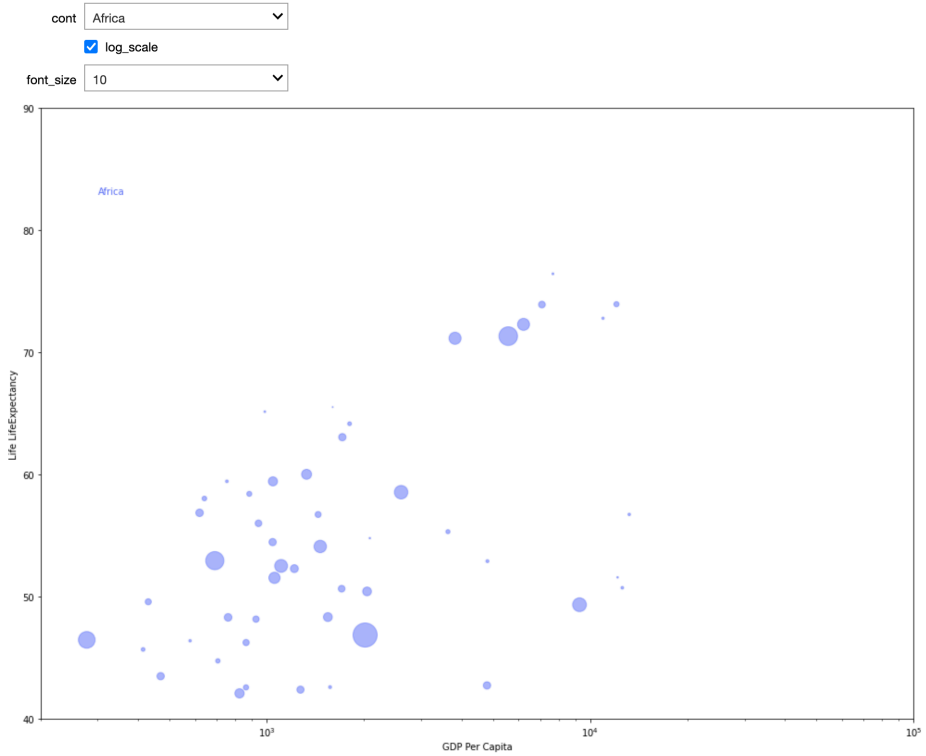
```
1 @widgets.interact
2 def plot_continent(cont = cont_dict, log_scale=True, font_size=font_sizes):
```

- This function takes three arguments, a dictionary, a boolean, and a. list, resulting in:

immediately above the output of the function.

Simple widget use

- Each GUI element is labeled with the argument name it controls



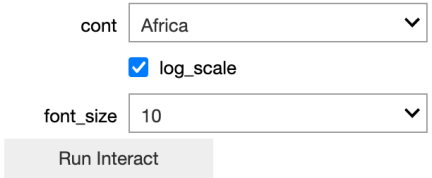
Simple widget use

- Both widgets.interact() and the widgets.interact_manual() can be used as functions, where the first argument is the function we want it to control. Every other argument is an argument that our function takes.
- The previous example could also be produced using

```
widgets.interact(plot_continent,
                 cont = cont_dict,
                 log_scale=True,
                 font_size=font_sizes);
```

Simple widget use

- If instead, we switch to widgets.interact_manual() we obtain an extra button to manually execute the function



- Widget can be instantiated manually and passed directly as the default value to the respective function argument.

Lesson 6.3
Widget customization

Widget customizing

- A custom version of our font size selector can be created using:

```
font_slider = widgets.IntSlider(
    value=18,
    min=10,
    max=40,
    step=2,
    description='Font size:',
    disabled=False,

    # Should the figure be updated as the value is being changed?
    continuous_update=False,
    orientation='horizontal',
    #readout=True,

    # Show only decimal values
    readout_format='d'
)
```

And passed directly to the function:

```
widgets.interact(plot_continent,
                 cont = cont_dict,
                 log_scale=True,
                 font_size=font_slider);
```

Connecting widgets

- Sometimes we want to create dependencies between the values of the different widgets
- Each widget has an observe() method that allows us to specify a handler function to be called whenever one of its specific traits (value, color, size, etc) changes.
- The handler function can then interact and modify other widgets as it sees fit.

Connecting widgets

- We can instruct the continent Dropdown widget to modify the maximum font size of the font IntSlider whenever its value changes by simply using:

```
23 def update_font_range(*args):
24     new_font_slider.max = 20+2.0 * continent_widget.value
25
26 # Tell the continent widget to call update_font_range
27 # whenever it's value changes
28 continent_widget.observe(update_font_range, 'value')
```

Code - ipywidgets

https://github.com/DataForScience/Visualization_LL

Lesson 7: Seaborn

7.1 Understand the structure of Seaborn

7.2 Understand the differences with matplotib

7.3 Explore the seborn API

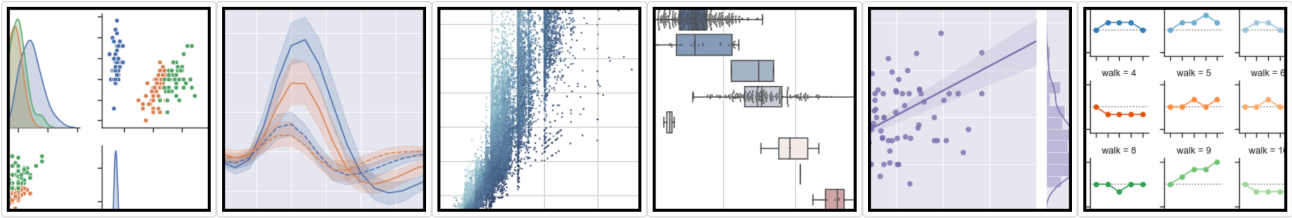
7.4 Demo

Lesson 7.1
Understand the structure of Seaborn

What is seaborn?

seaborn.pydata.org

seaborn: statistical data visualization

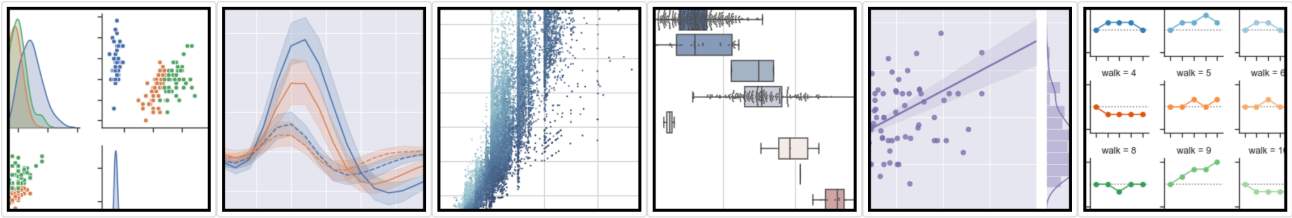


"Seaborn is a library for making statistical graphics in Python. It is built on top of matplotlib and closely integrated with pandas data structures."

What is seaborn?

seaborn.pydata.org

seaborn: statistical data visualization

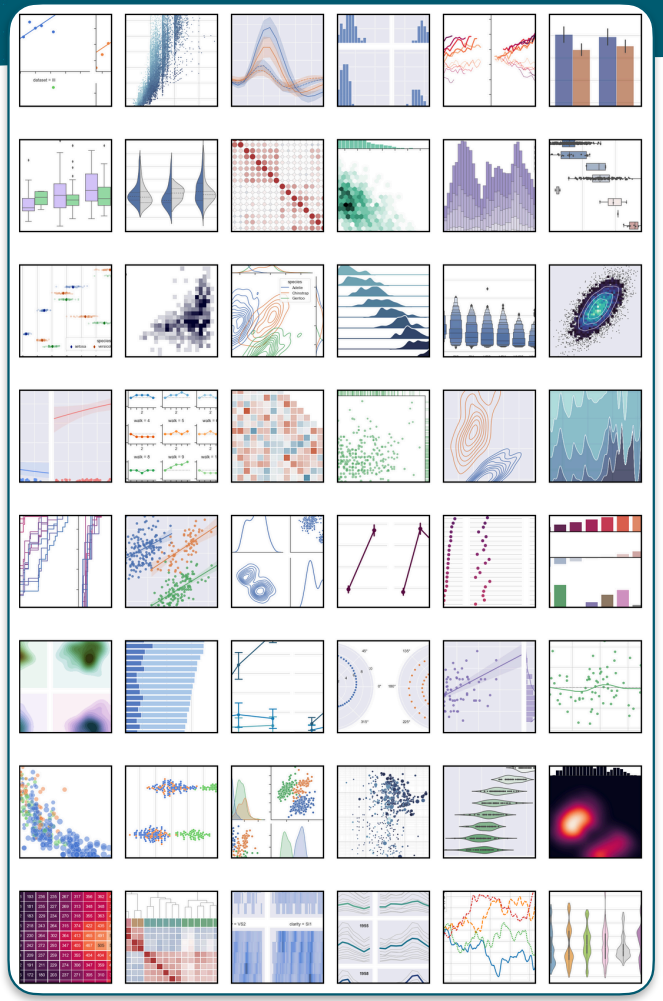


"Seaborn aims to make visualization a central part of exploring and understanding data. Its dataset-oriented plotting functions operate on dataframes and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots."

seaborn

seaborn.pydata.org

Extensive gallery that provides the source code for all the figures shown



Pearson

- The first step is to import the seaborn module:

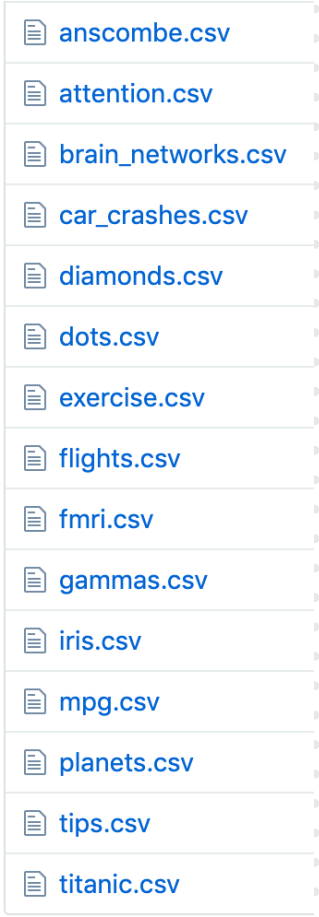
```
import seaborn as sns
```

- The convention is to import seaborn as sns (sns is a geeky reference to Sam Norman Seaborn, a fictional character on the television show The West Wing.
- The sns module contains all the functions we will use as direct members.

Datasets

- For ease of learning and demonstration, seaborn makes it easy to access a small set of standard datasets.

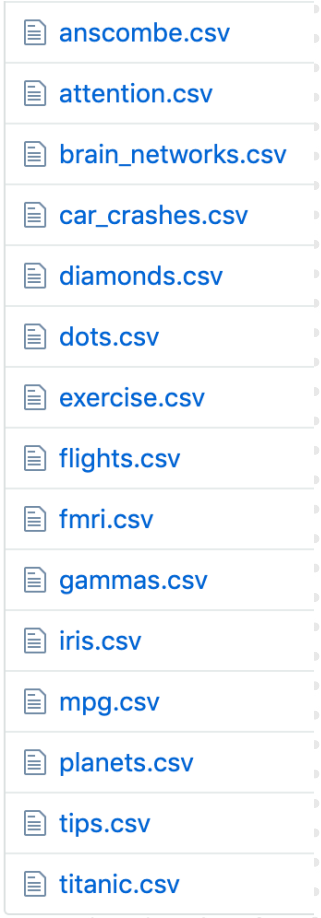
- The datasets can be access easily by calling the load_dataset function with the file name (sans extension) as a parameter.



Datasets

- .load_dataset(name) - where name is “iris” “mpg” etc…
- datasets are returned as “tidy" pandas dataframes

	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
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4



Lesson 7.2

Understand the differences with matplotlib

Types of plots

- sns.scatterplot()
- sns.lineplot()

Relationship plots

- sns.stripplot()
- sns.swarmplot()
- sns.boxplot()
- sns.violinplot()
- sns.boxenplot()
- sns.pointplot()
- sns.barplot()
- sns.countplot()

Categorical plots

Types of plots

Axes level	Figure level	
sns.scatterplot()	Relationship plots sns.relplot()	kind="scatter"
sns.lineplot()		kind="line"
sns.stripplot()	Categorical plots sns.catplot()	kind="strip"
sns.swarmplot()		kind="swarm"
sns.boxplot()		kind="box"
sns.violinplot()		kind="violin"
sns.boxenplot()		kind="boxen"
sns.pointplot()		kind="point"
sns.barplot()		kind="bar"
sns.countplot()		kind="count"

Types of plots

Axes level	Figure level	
	Relationship plots sns.relplot()	
• sns.scatterplot() • sns.lineplot()		• kind="scatter" • kind="line"
• s • s • s • sns.violinplot() • sns.boxenplot() • sns.pointplot() • sns.barplot() • sns.countplot()	**Categorical plots** sns.catplot()	• ip" • arm" • x" • kind="violin" • kind="boxen" • kind="point" • kind="bar" • kind="count"

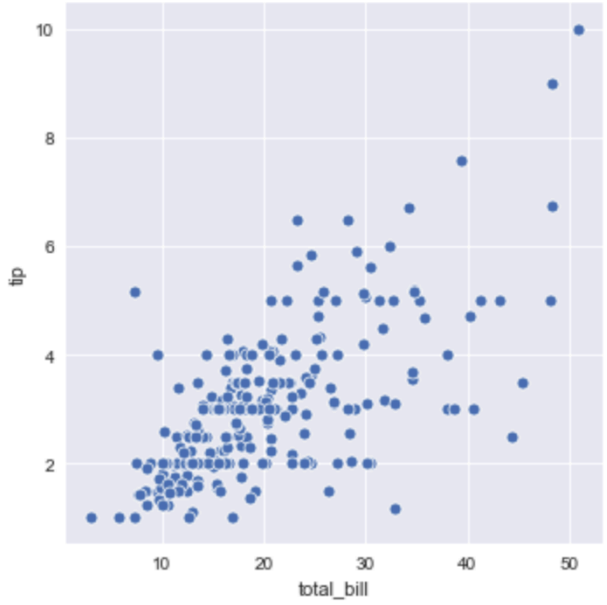
Figure level functions are equivalent to the axes level ones, (with the corresponding kind parameter) but more adapted for ease of exploration

Pearson

Figure vs Axes level

Figure level

```
1  fg = sns.relplot(x="total_bill", y="tip", data=tips, kind="scatter")
```

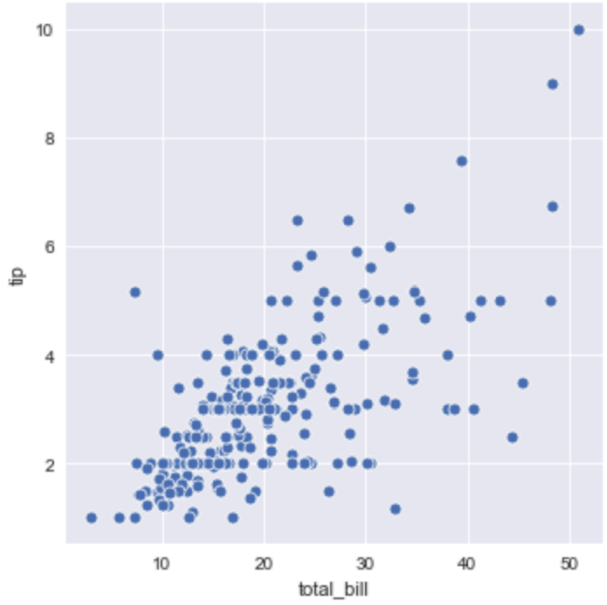


Pearson

Figure vs Axes level

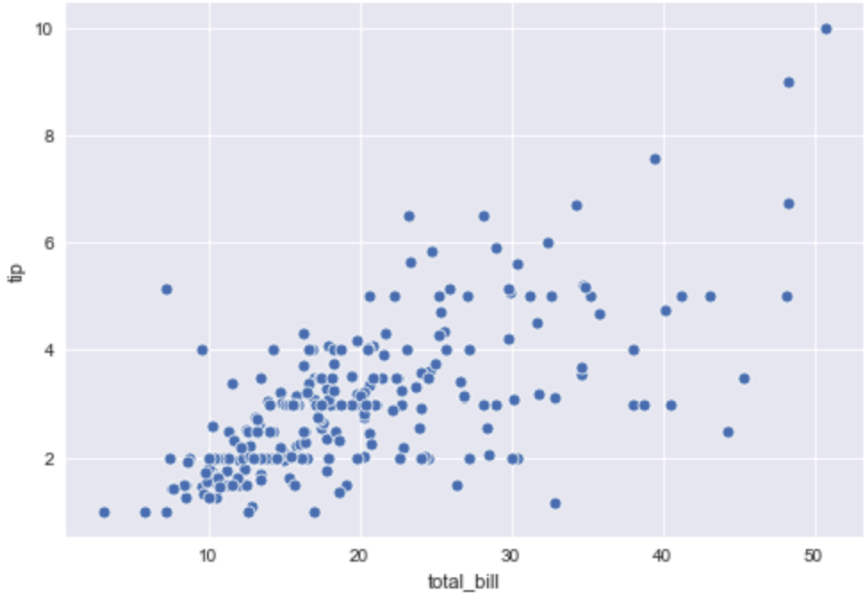
Figure level

```
1  fg = sns.relplot(x="total_bill", y="tip", data=tips, kind="scatter")
```



Axes level

```
1  ax = sns.scatterplot(x="total_bill", y="tip", data=tips)
```

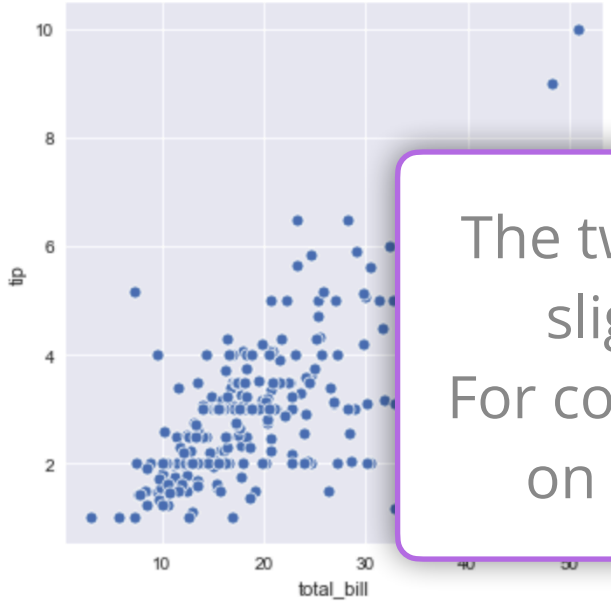


Pearson

Figure vs Axes level

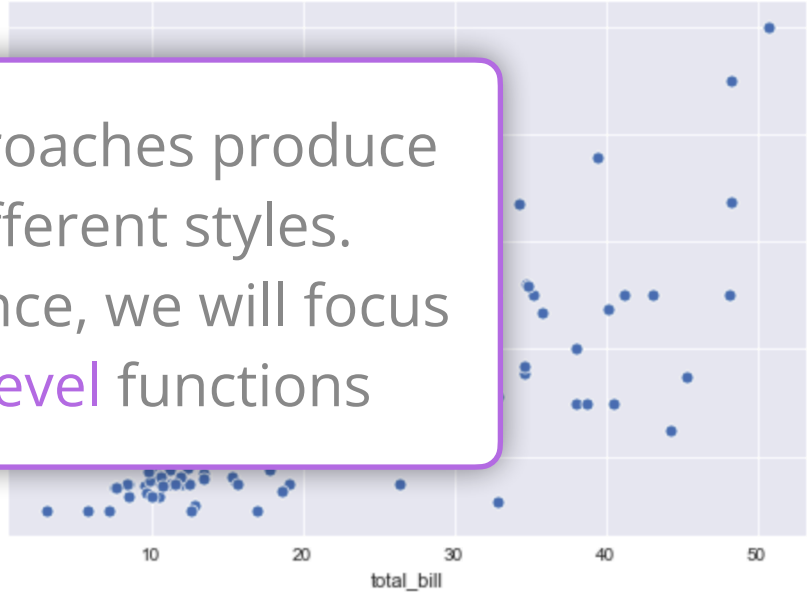
Figure level

```
1  fg = sns.relplot(x="total_bill", y="tip", data=tips, kind="scatter")
```



Axes level

```
1  ax = sns.scatterplot(x="total_bill", y="tip", data=tips)
```



The two approaches produce slightly different styles.
For convenience, we will focus on Figure level functions

Basic Function Structure

- The basic plotting functions are designed to work directly with pandas data frames
- Every plotting function takes a data parameter that is used to pass the correct DataFrame to the function
- There is no limitation on the number of columns that the DataFrame can contain.

Basic Function Structure

- The specific columns to be plotted are passed by setting other parameters to the respective column names. In particular:
  - x - column to plot along the x-axis
  - y - column to plot along the y axis
- Plot properties such as hue, size, style, etc can also be set using column names
- In the case of figure level functions, the kind parameter determines what specific type of plot is produced.

Figure Level Functions

- sns.relplot(x, y, hue, size, style, data, kind)
  - x, y - column to plot in each axes
  - hue - column specifying how to color each data element
  - size - column that determines the size of each element
  - style - column to determine the style of each element
  - data - dataframe containing the data to plot
  - kind - string specifying the type of plot to produce

Figure Level Functions

- sns.catplot(x, y, hue, order, data, orient, kind)
  - orient - specifies the orientation of the plot ('v' or 'h' for vertical or horizontal)
  - order - specifies the order in which the categorical variable (x or y) is plotted
  - *_order - family of parameters that determiner the order in which the respective categorical value (hue_order, style_order, size_order, etc…) is plotted

Figure Level Functions

- Every plot you can generate with an axes level function can also be generated by a figure level function

- The converse is, however, not true.

Figure Level Functions

- One of the main advantages of figure level functions is their ability to easily generate subplots by just passing a couple of extra parameters:
  - row, col - specifies the column names to be used to split the dataset and plot along rows and columns
  - col_wrap - width at which to wrap the column. Incompatible with a row setting.
  - row_order, col_order - work similarly to the other *_order parameters

Figure Level Functions

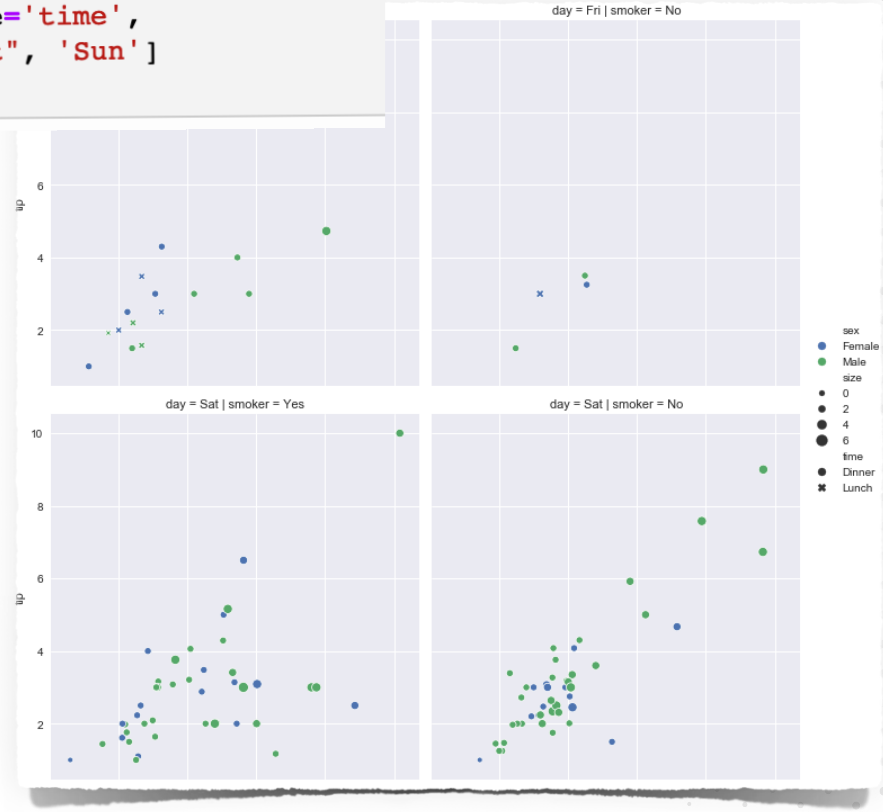
- The return types of axes level functions and figure level function are also different:
  - Axes level functions return a matplotlib Axes object
  - Figure level functions return a seaborn FacetGrid object
- Axes level functions can be easily added to a pre-existing matplotlib plot by passing an matplotlib Axes object to the ax parameter.

Figure Level Functions

```
sns.relplot(x="total_bill", y="tip", data=tips, kind="scatter",
            hue="sex", size="size",
            col='smoker', row='day', style='time',
            row_order=["Thur", "Fri", "Sat", 'Sun']
            )
```

Figure Level Functions

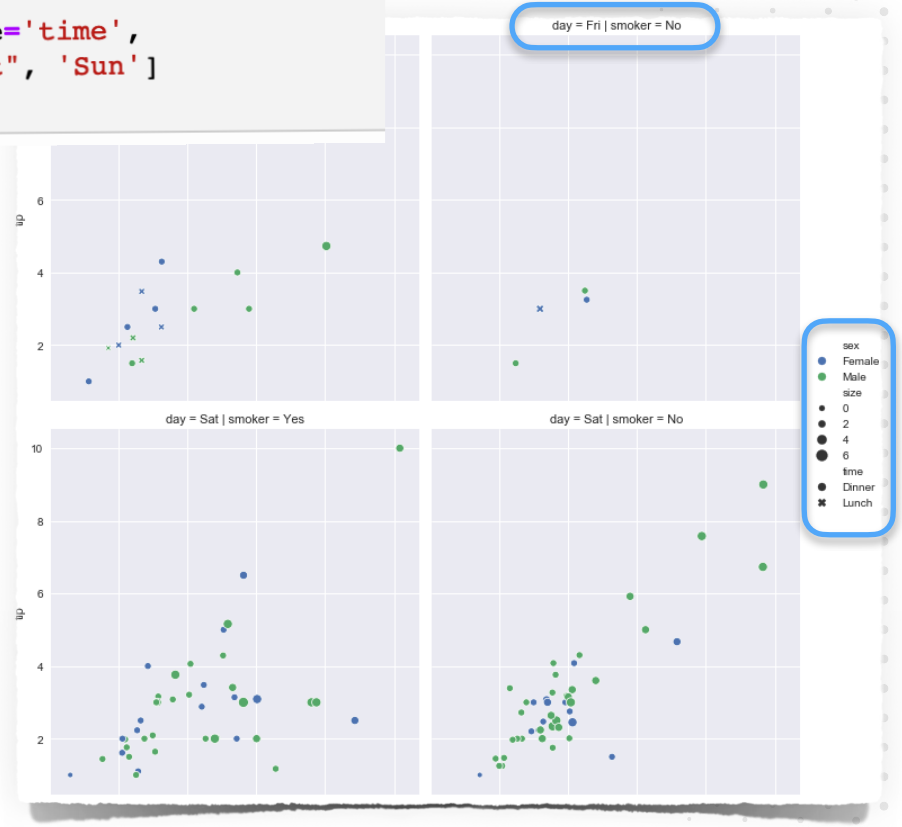
```
sns.relplot(x="total_bill", y="tip", data=tips, kind="scatter",
            hue="sex", size="size",
            col='smoker', row='day', style='time',
            row_order=["Thur", "Fri", "Sat", 'Sun']
            )
```



Pearson

Figure Level Functions

```
sns.relplot(x="total_bill", y="tip", data=tips, kind="scatter",
            hue="sex", size="size",
            col='smoker', row='day', style='time',
            row_order=["Thur", "Fri", "Sat", 'Sun']
            )
```



Pearson

Lesson 7.3
Explore the Seaborn API

FacetGrid

- FacetGrid is the object that seaborn uses in the background to generate the subplots.
- You can easily use FacetGrid with any matplotlib “compatible" plotting function.
- The process is divided into two parts:
  - Instantiate a FacetGrid object with a data frame and basic plotting parameters (similar to any other seaborn function).
  - Use the FacetGrid.map method to initialize the FacetGrid object with the function that will generate the plot along with any extra parameters you require.

FacetGrid

- Any function that modifies the currently active Axes object can be used (both matplotlib functions and custom functions)
- Any extra parameters passed to map() will be passed directly to the plotting function. This function should accept a **kwargs argument
- FacetGrid will pass the sliced data as Series to the plotting function

FacetGrid

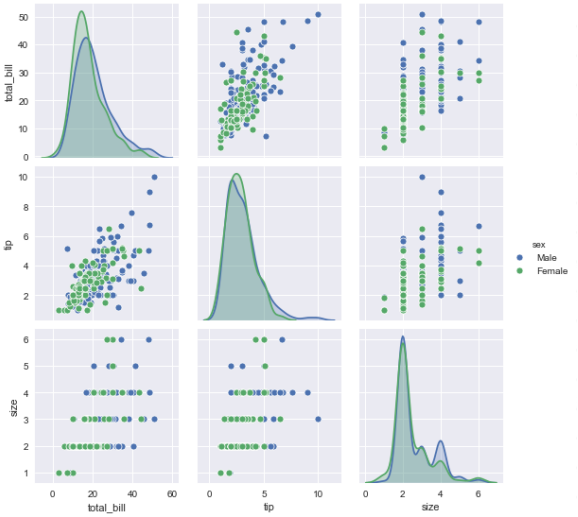
- After plotting the FacetGrid will contain a field axes containing an array with references to all the matplotlib Axes objects it generated.
- We can further customize the plot by directly using matplotlib functionality.

PairPlot

- FacetGrid is a general concept with several specialized variations within seaborn
  - PairPlot - specialized FacetGrid that automatically plots every pairwise relationship of numerical features. The diagonal elements plot the distributions of the respective feature.
  - JointPlot - specialized FacetGrid that uses marginal right and top plots to visualize the distributions of the X and Y features.

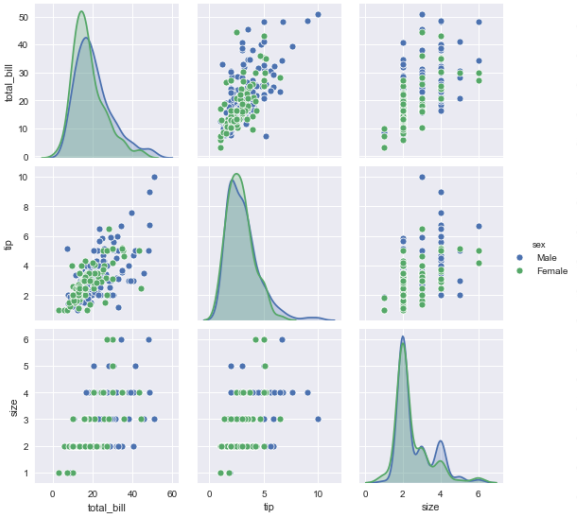
PairPlot

- Plots every pairwise relationship of numerical features.
- Diagonal elements plot the distributions of each feature.
- hue (and hue_order) - further subdivide the data set by color.
- varslist - list of variables to use. Default is to use all numerical variables.
- x_vars, y_vars - list of variables to use along the x and y axis



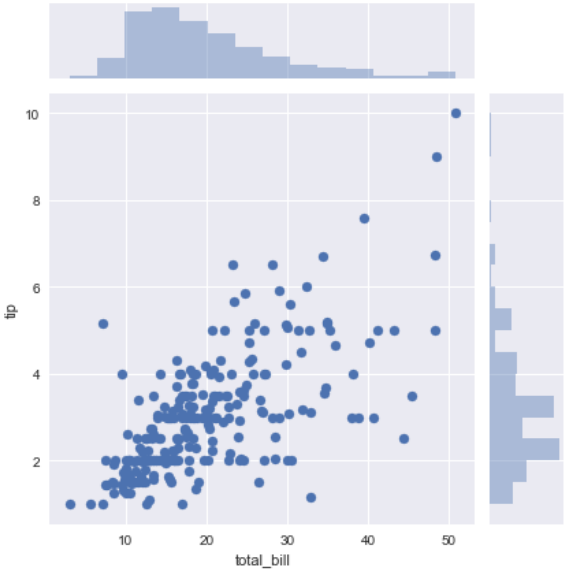
PairPlot

- kind - kind of plot to use for pairwise relationships: ‘scatter’, ‘kde’, ‘hist’, ‘reg’
- diag_kind - kind of plot to use for in the diagonal plots: ’hist’, ‘kde’, None
- markers - matplotlib marker(s) to use. If list, it should have same length as hue
- Array of Axes objects in axes (like FacetGrid)



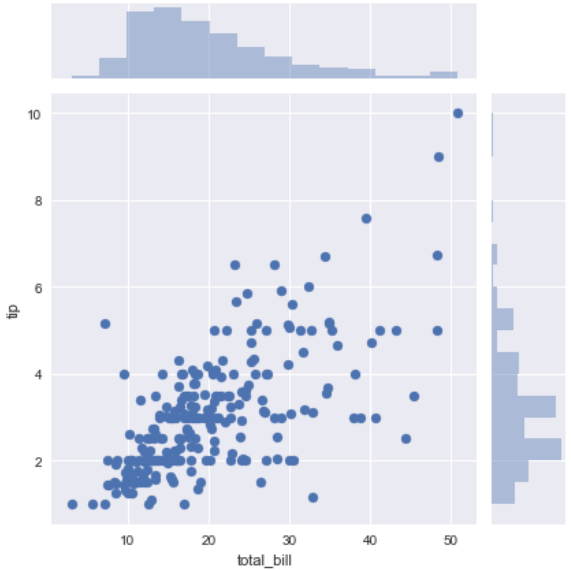
JointPlot

- Draw a plot of two variables with bivariate and univariate graphs.
- Marginal plots on the top and right hand side visualize the distributions of the X and Y features.
- hue (and hue_order) - further subdivide the data set by color.
- kind - The type of plot to draw: “scatter”, “kde”, “hist”, “hex”, “reg”, “resid”



JointPlot

- Axes objects are available as in specific fields: ax_join, ax_marg_x, ax_marg_y
- Additional layers can be added to the JointPlot using plot_joint() and plot_marginals()



Distribution plots

- Distribution Plots are a third class of plots, in addition to Relationship Plots and Categorical Plots.
- The function sns.displot() allows us to generate a FacetGrid plot of the distributions underlying our data.
- Supports univariate and bivariate distribution of data
- Allows slicing using hue, row, col, etc.

Distribution plots

- sns.displot(x, y, hue, data, kind)
  - x, y - column names to plot in each axes. Only x for univariate distribution and both x and y for bivariate distributions
  - hue - column specifying how to color each element
  - data - dataframe containing the data to plot
  - row, col - Columns to use for the rows and columns of the FacetGrid
  - kind - string specifying the type of plot to produce

Distribution plots

- sns.displot(x, y, hue, data, kind)
  - rug - Boolean flag determining whether or not to add a rug plot highlighting the locations of the data points
  - kind - The kind of plot to generate:
    - “hist” - 1D or 2D histogram
    - “kde” - Kernel Density Estimation
    - “ecdf” - Empirical Cumulative Distribution Function

Types of plots

Axes level

- sns.histplot()
- sns.kdeplot()
- sns.ecdfplot()

Figure level

Distribution plots

sns.displot()

- kind="hist"
- kind="kde"
- kind="ecdf"

Code - Seaborn

https://github.com/DataForScience/Visualization_LL

Lesson 8: Bokeh

8.1 Basic Plotting with Bokeh

8.2 Advanced Plotting

8.3 Networks

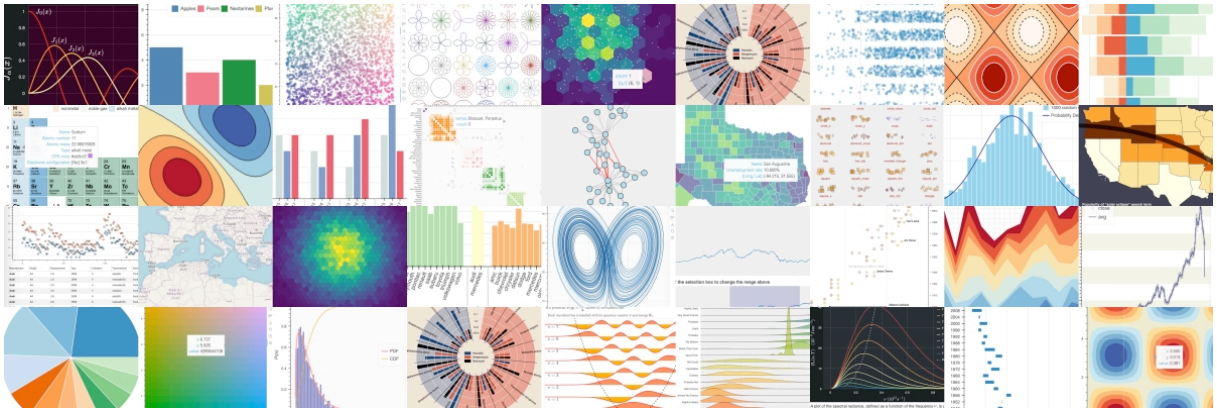
8.4 Demo

Lesson 8.1
Basic plotting with Bokeh

bokeh

bokeh.org

- Interactive visualization library for Python
- Javascript backend that supports all modern browsers
- Optimized to Interactively Explore Data in Notebooks
- Supports Streaming Data



Pearson

- bokeh is powerful but its structure can sometimes complex
- It relies on a layered structure of submodules:
  - bokeh.models - responsible for handling the JSON data created by the JavaScript backend. models are fairly simple
  - bokeh.plotting - mid-level interface focusing on Matplotlib like features. Handles the creation of axes, grids, and tools and contains the figure(), function, our workhorse

- bokeh is powerful but its structure can sometimes complex
- It relies on a layered structured of submodules:
  - bokeh.io - functions to handle the input/output, such as output_file(), output_notebook(), and show() functions.
  - bokeh.palettes - color palettes like Viridis256, Category20c, Spectral6, GnBu3, OrRd3, etc
  - bokeh.layouts - functions to layout multiple figures in the same plot

Basic Plotting

- The first step with any Bokeh plot is to generate a figure object

```
1 p = figure(width=400, height=400)
```

- Figures are only displayed when you call show(p)

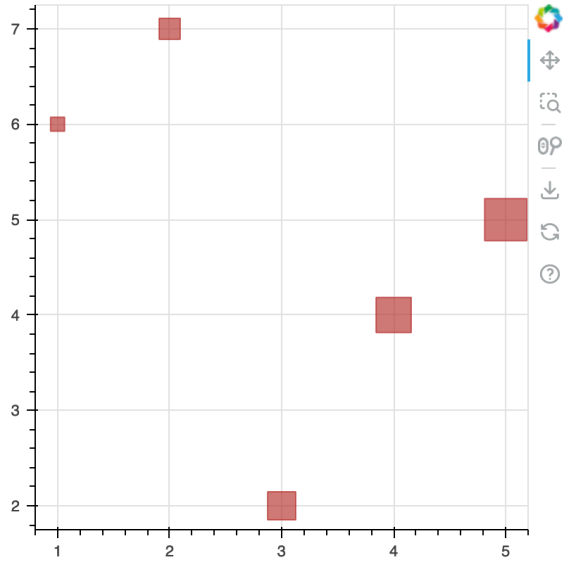
Basic Plotting

- Plotting functions exist as methods of the figure object
  - p.circle() - Scatter plot using circles
  - p.square() - Scatter plot using squares
  - p.line() - Line plot
  - p.wedge() - Pie plot
  - p.annular_wedge() - Donut plot
  - p.vbar() / p.hbar() - Vertical and Horizontal bars
  - p.vbar_stack() / p.hbar_stack() - Stacked Vertical and Horizontal bars

Basic Plotting

- Simple figures can be generated by calling any of the methods above

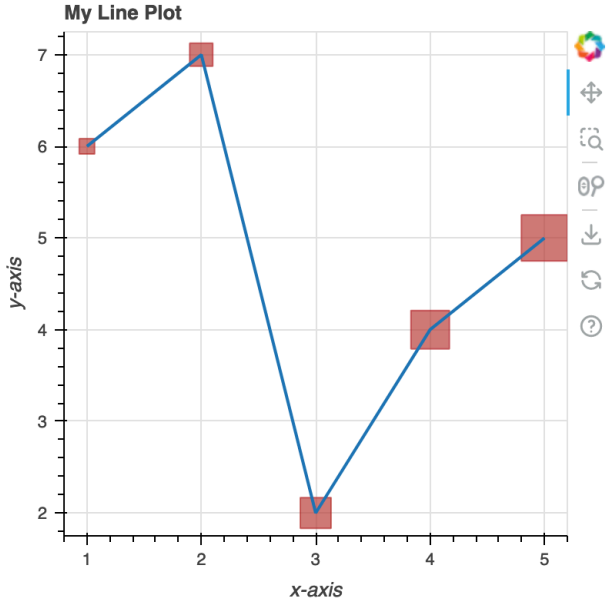
```
p = figure(width=400, height=400)

# Use a different size for each symbol
p.square([1, 2, 3, 4, 5], [6, 7, 2, 4, 5],
         size=[10, 15, 20, 25, 30],
         color="firebrick", alpha=0.6)
show(p)
```



Basic Plotting

- More complex figures can be generated by calling multiple methods

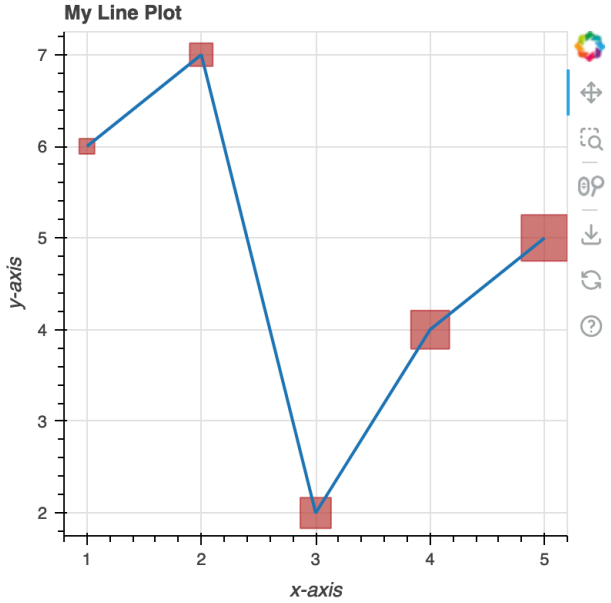
```
1  p = figure(width=400, height=400, title="My Line Plot")
2
3  # add a line renderer
4  p.square([1, 2, 3, 4, 5], [6, 7, 2, 4, 5],
5           size=[10, 15, 20, 25, 30],
6           color="firebrick", alpha=0.6)
7  p.line([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], line_width=2)
8  p.xaxis.axis_label = 'x-axis'
9  p.yaxis.axis_label = 'y-axis'
10
11 show(p)
```



Basic Plotting

- Axis labels can be added by simple assignment

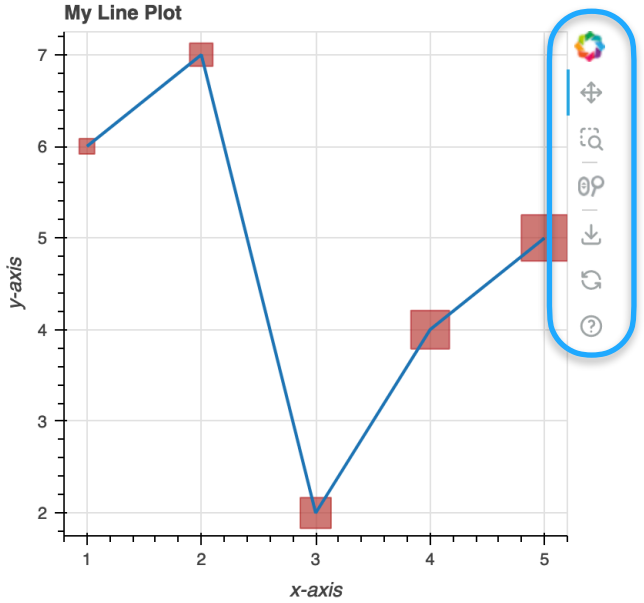
```
1  p = figure(width=400, height=400, title="My Line Plot")
2
3  # add a line renderer
4  p.square([1, 2, 3, 4, 5], [6, 7, 2, 4, 5],
5           size=[10, 15, 20, 25, 30],
6           color="firebrick", alpha=0.6)
7  p.line([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], line_width=2)
8  p.xaxis.axis_label = 'x-axis'
9  p.yaxis.axis_label = 'y-axis'
10
11 show(p)
```



Basic Plotting

- Toolbar on the right allows us to zoom in, pan, select specific points, download a copy of the image, etc.

```
1  p = figure(width=400, height=400, title="My Line Plot")
2
3  # add a line renderer
4  p.square([1, 2, 3, 4, 5], [6, 7, 2, 4, 5],
5           size=[10, 15, 20, 25, 30],
6           color="firebrick", alpha=0.6)
7  p.line([1, 2, 3, 4, 5], [6, 7, 2, 4, 5], line_width=2)
8  p.xaxis.axis_label = 'x-axis'
9  p.yaxis.axis_label = 'y-axis'
10
11 show(p)
```



Pie and Donut charts

- p.wedge() and p.annular_wedge() take a similar set of arguments:
  - x / y - Location on which to generate the plot
  - start_angle / end_angle - start and end angle of each slice
  - legend_field - the field (column) name to use for the legend
  - source - the data source

Pie and Donut charts

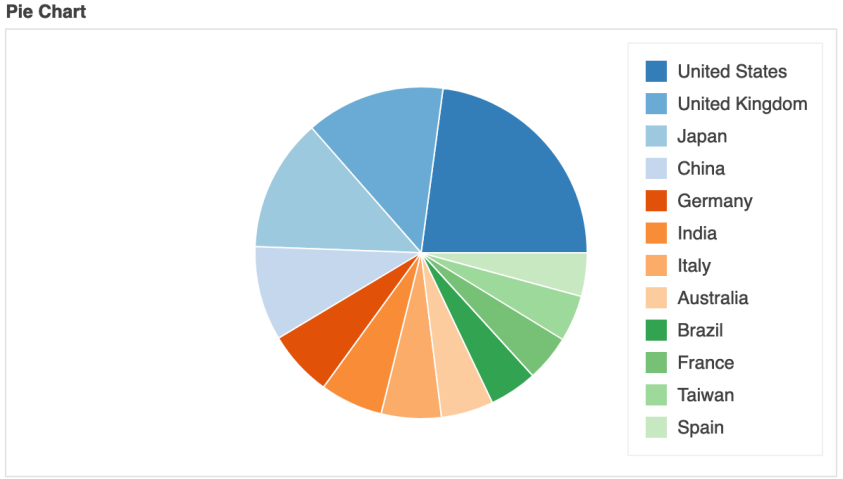
- The main difference is that p.wedge() takes a single radius argument, while p.annular_wedge() takes both an inner_radius and an outer_radius
- Complex visualizations often rely on complex datasets. Bokeh makes it easy to use pandas DataFrames as data sources, where values can be referred to by column names.

Pie and Donut charts

- Using this DataFrame we can generate a Pie plot using:

```
p = figure(height=350, title="Pie Chart", toolbar_location=None,
           tools="hover", tooltips="@country: @value")

p.wedge(x=0, y=1, radius=0.4,

        # use cumsum to cumulatively sum the values for start and end angles
        start_angle=cumsum('angle', include_zero=True),
        end_angle=cumsum('angle'),
        line_color="white",
        fill_color='color',
        legend_field='country', source=data)

p.axis.axis_label=None
p.axis.visible=False
p.grid.grid_line_color = None

show(p)
```

	country	value	color	angle
0	United States	157	#3182bd	1.437988
1	United Kingdom	93	#6baed6	0.851802
2	Japan	89	#9ecae1	0.815165
3	China	63	#c6dbef	0.577027
4	Germany	44	#e6550d	0.403003
5	India	42	#fd8d3c	0.384685



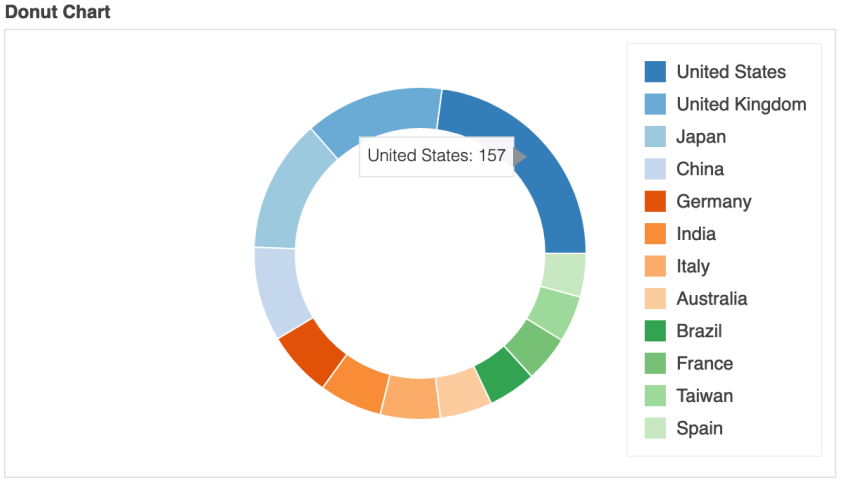
Pearson

Pie and Donut charts

- And a donut plot using:

```
p = figure(height=350, title="Donut Chart", toolbar_location=None,
           tools="hover", tooltips="@country: @value")

p.annular_wedge(x=0, y=1,
        inner_radius=0.3, outer_radius=0.4,

        # use cumsum to cumulatively sum the values for start and end angles
        start_angle=cumsum('angle', include_zero=True),
        end_angle=cumsum('angle'),
        line_color="white", fill_color='color',
        legend_field='country', source=data)

p.axis.axis_label=None
p.axis.visible=False
p.grid.grid_line_color = None

show(p)
```

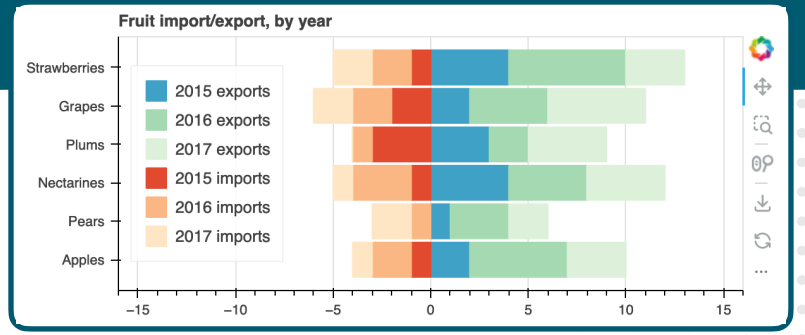
	country	value	color	angle
0	United States	157	#3182bd	1.437988
1	United Kingdom	93	#6baed6	0.851802
2	Japan	89	#9ecae1	0.815165
3	China	63	#c6dbef	0.577027
4	Germany	44	#e6550d	0.403003
5	India	42	#fd8d3c	0.384685



Pearson

Lesson 8.2
Advanced Plotting

Advanced Plotting



- Sophisticated plots can be generated with just a few lines of straightforward code

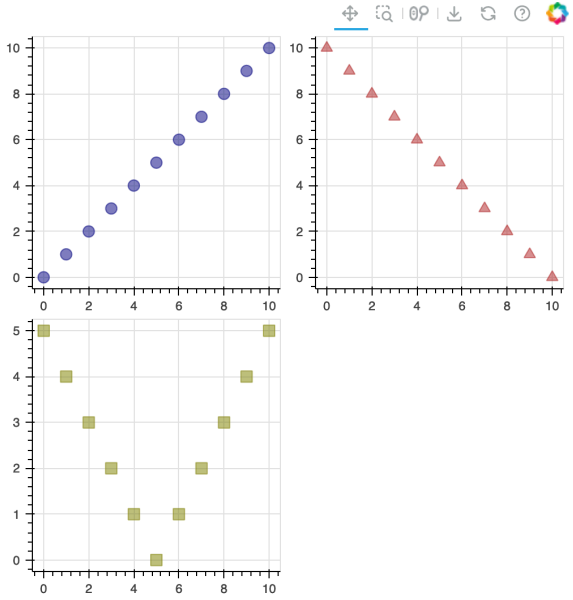
```
1  p = figure(y_range=fruits, height=250, x_range=(-16, 16),
2             title="Fruit import/export, by year")
3
4  p.hbar_stack(years, y='fruits', height=0.9, color=GnBu3,
5               source=ColumnDataSource(exports),
6               legend_label=["%s exports" % x for x in years])
7
8  p.hbar_stack(years, y='fruits', height=0.9, color=OrRd3,
9               source=ColumnDataSource(imports),
10              legend_label=["%s imports" % x for x in years])
11
12 p.y_range.range_padding = 0.1
13 p.ygrid.grid_line_color = None
14 p.legend.location = "center_left"
15
16 show(p)
```

Multiple plots

- Multiple figure objects can be put together using the gridplot() command
- gridplot() takes a 2D array of image objects as the main argument

```
16 # put all the plots in a gridplot
17 p = gridplot([[s1, s2], [s3, None]])
```

- Subplots can be generated and configured separately

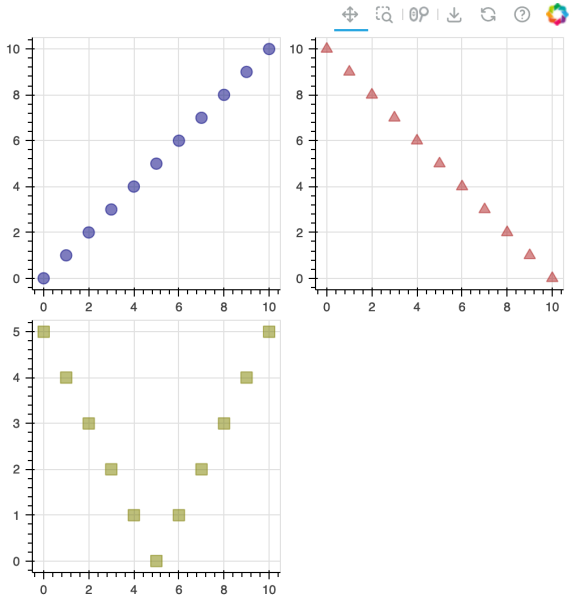


Linking plots

- Plots can be connected by passing references to other objects' settings:

```
10 # create a new plot and share both ranges
11 s2 = figure(x_range=s1.x_range, y_range=s1.y_range, **plot_options)
12 s2.triangle(x, y1, size=10, color="firebrick")
```

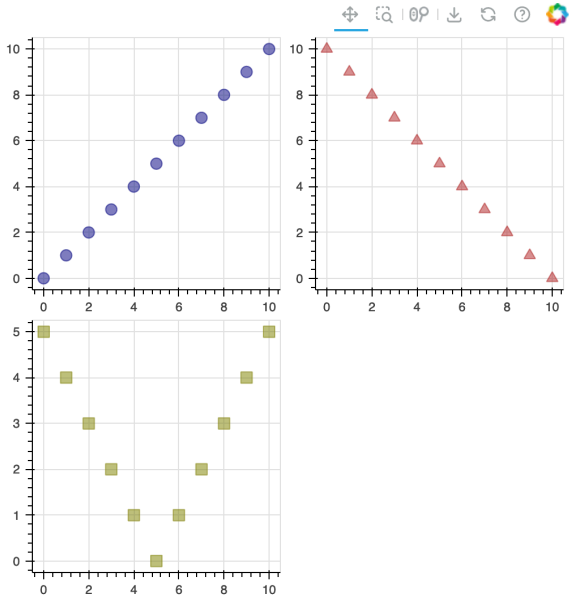
- In this way, any change to one of the plots quickly reflects in all others



Linking plots

- Plots can also be connected by sharing a common DataSource:

```
4  # create a column data source for the plots to share
5  source = ColumnDataSource(data=dict(x=x, y0=y0, y1=y1))
6
7  TOOLS = "box_select,lasso_select,reset,help"
8
9  # create a new plot and add a renderer
10 left = figure(tools=TOOLS, width=300, height=300)
11 left.circle('x', 'y0', source=source)
12
13 # create another new plot and add a renderer
14 right = figure(tools=TOOLS, width=300, height=300)
15 right.circle('x', 'y1', source=source)
```



- Any selection of the data in one DataSource is propagated to all figures referring to it

Plot tools

- Plot tools are of several types:
  - Pan/Drag tools
    - BoxSelectTool - Select all points within a box
    - BoxZoomTool - Zoom into a box
    - LassoSelectTool - Select all points within an arbitrary area
    - PanTool - Pan the plot

Plot tools

- Plot tools are of several types:
  - Pan/Drag tools
  - Click/Tap tools
    - PolySelectTool - Select all points within an arbitrary polygon
    - TapTool - Select individual data points

Plot tools

- Plot tools are of several types:
  - Pan/Drag tools
  - Click/Tap tools
  - Scroll/Pinch tools
    - WheelZoomTool - Zoom the plot in and out
    - WheelPanTool - Pans the plot along a specified direction

Plot tools

- Plot tools are of several types:
  - Pan/Drag tools
  - Click/Tap tools
  - Scroll/Pinch tools
  - Action tools
    - UndoTool - Restores the previous state
    - RedoTool - Reverses the last action Undone
    - ResetTool - Resets the plot to the default
    - SaveTool - Save a PNG image of the plot.

Plot tools

- Plot tools are of several types:
  - Pan/Drag tools
  - Click/Tap tools
  - Scroll/Pinch tools
  - Action tools
  - Inspector tools
    - CrosshairTool - Draws a crosshair annotation
    - HoverTool - Displays a customizable tool tip

Plot tools

- Tools are enabled by listing them in the tools argument of the figure object.

```
1  p = figure(height=350, title="Pie Chart", toolbar_location=None,
2             tools="hover", tooltips="@country: @value")
```

- The toolbar_location argument can be set to None to completely disable it or “above”, “below”, “left”, or "right" to specify a preferred location.
- The HoverTool also requires you to specify the format of the tooltip displayed through the tooltips argument as a string. Tooltips can refer to values in the data source by specifying the column name preceded by a @

Lesson 8.3
Networks

Networks

networkx.org

- Networks and Graphs are mathematical objects used to describe the relationships or connections (edges) between items (nodes);
- Bokeh has good support for visualizing networks described in NetworkX, the leading network package in Python

Networks

networkx.org

- To generate a bokeh graph from a NetworkX object, use from_networkx() with the NetworkX object and the layout algorithm you want to use
- The resulting object must then be appended to the list of renderers of our plot

```
7 # Create a Bokeh graph from the NetworkX input using nx.spring_layout
8 graph = from_networkx(G, nx.spring_layout, scale=1.8, center=(0,0))
9 plot.renderers.append(graph)
```

Networks

networkx.org

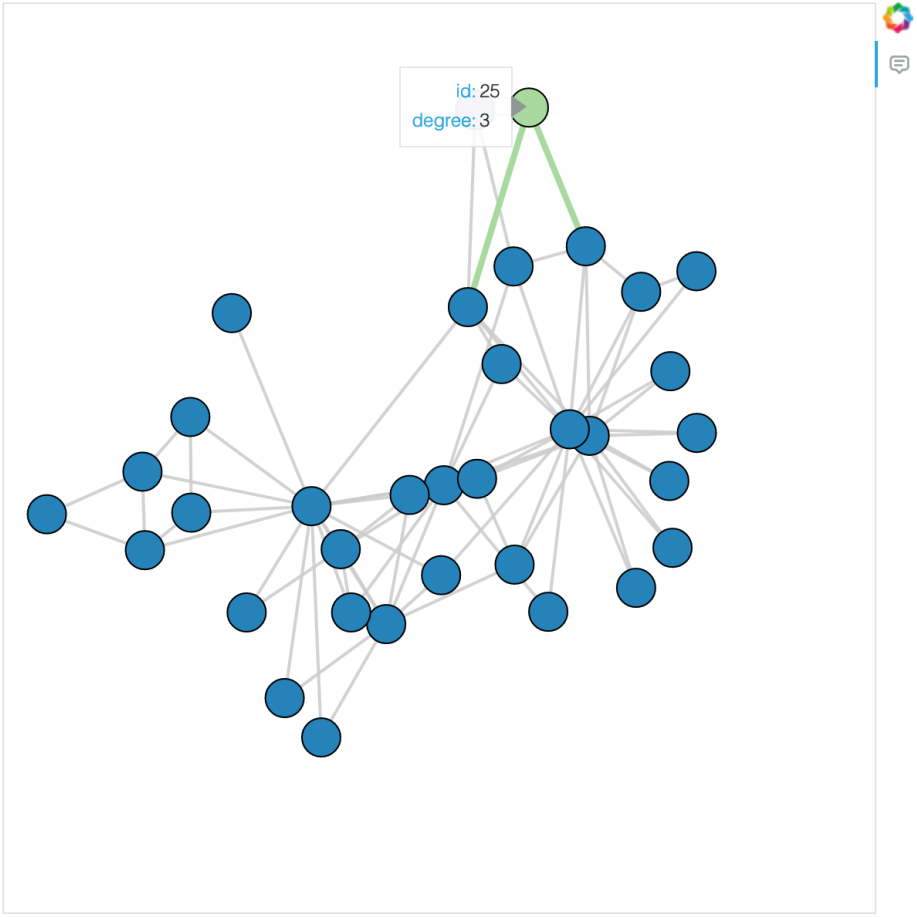
- We can also specify display settings for nodes and edges

```
14 # Blue circles for nodes, and light grey lines for edges
15 graph.node_renderer.glyph = Circle(size=25, fill_color='#2b83ba')
16 graph.edge_renderer.glyph = MultiLine(line_color="#cccccc",
17                                        line_alpha=0.8, line_width=2)
```

- Specify tooltips for nodes and edges, etc

Networks

networkx.org



Pearson

Code - Bokeh

https://github.com/DataForScience/Visualization_LL

Lesson 9: Plotly

9.1 Basic Plotly

9.2 3D and Animated plots

9.3 Demo

Lesson 9.1
Basic Plotly

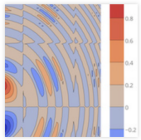
- Founded in 2012, in Montreal, Canada
- plotly.py - interactive, open-source library for creating graphs powered by the plotly.js JavaScript library
- Supports a huge range of interactive graphs across a large number of applications and fields
- Integrated into Dash, a framework for building web-based analytic applications.

plotly

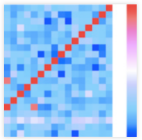
plotly.com

Scientific Charts

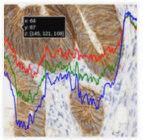
More Scientific Charts »



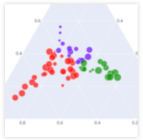
Contour Plots



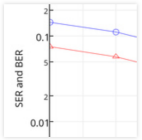
Heatmaps



Imshow



Ternary Plots



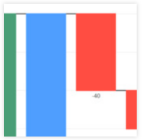
Log Plots

Financial Charts

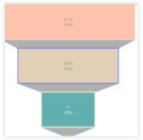
More Financial Charts »

Time Series and Date Axes

Candlestick Charts



Waterfall Charts

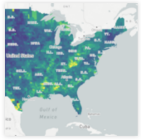


Funnel Chart

OHLC Charts

Maps

More Maps »



Mapbox Choropleth Maps

Lines on Mapbox

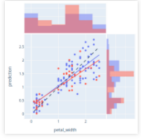
Filled Area on Maps

Bubble Maps

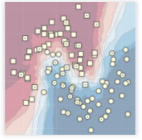
Mapbox Density Heatmap

Artificial Intelligence and Machine Learning

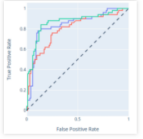
More AI and ML »



ML Regression



kNN Classification



ROC and PR Curves

PCA Visualization

AI/ML Apps with Dash

Pearson

- plotly relies on three main modules:
  - plotly.express - to generate entire figures at once
  - plotly.graph_objects - to generate individual graph components
  - plotly.figure_factory - to create multi-plot figures

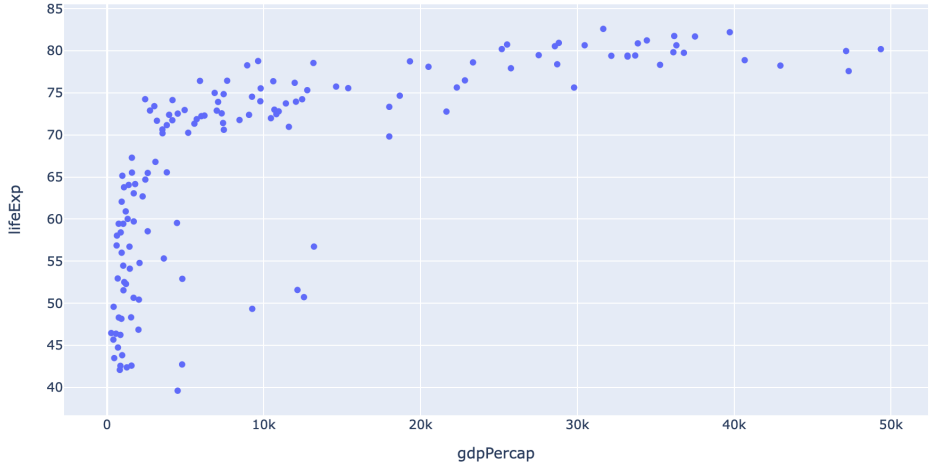
- plotly.express supports a wide range of figures, each with its own dedicated method:
  - Basics: scatter(), line(), area(), bar(), funnel(), timeline()
  - Part-of-Whole: pie(), sunburst(), treemap()
  - 1D Distributions: histogram(), box(), violin(), strip(), ecdf()
  - 2D Distributions: density_heatmap(), density_contour()
  - Matrix or Image Input: imshow()
  - 3-Dimensional: scatter_3d(), line_3d()
  - Multidimensional: scatter_matrix(), parallel_coordinates(), parallel_categories()
  - and many more…

plotly express

- Figures don't display until the show() method is called on the figure object
- A simple plot can be generated using:

```
1  fig = px.scatter(gapminder, x="gdpPercap", y="lifeExp")
2  fig.show()
```

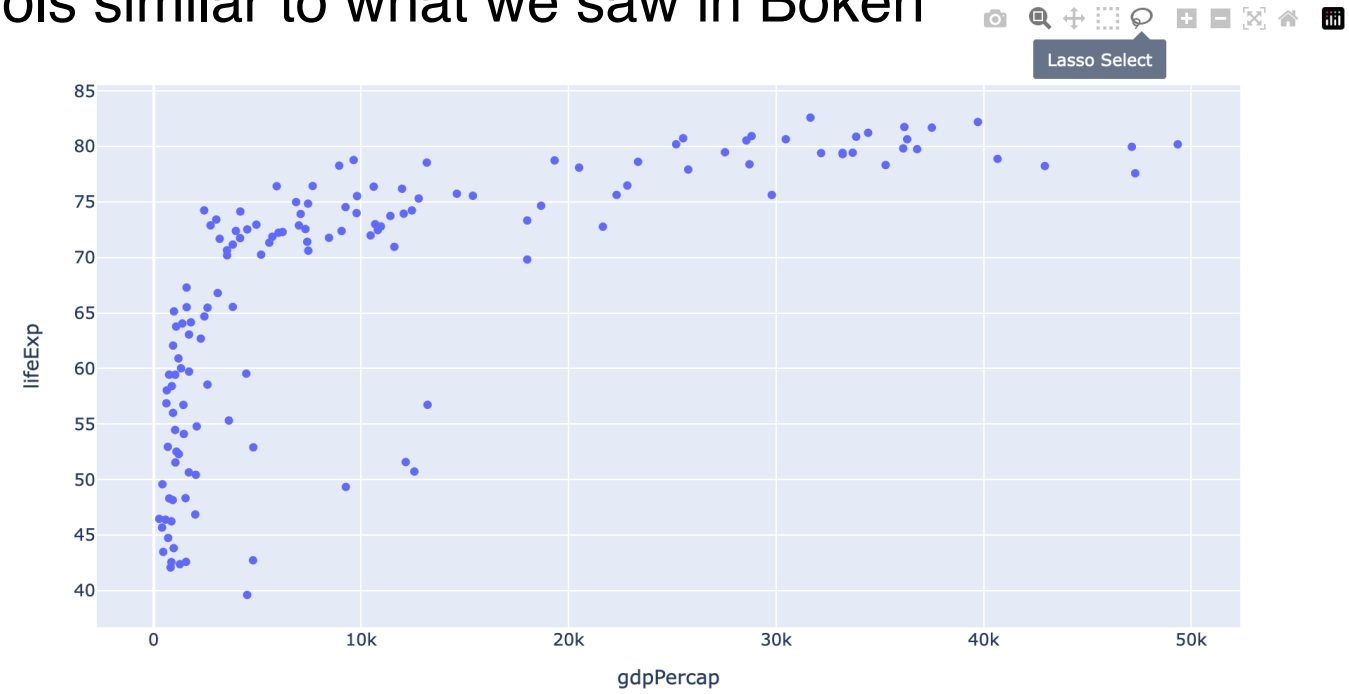
- Resulting in



Pearson

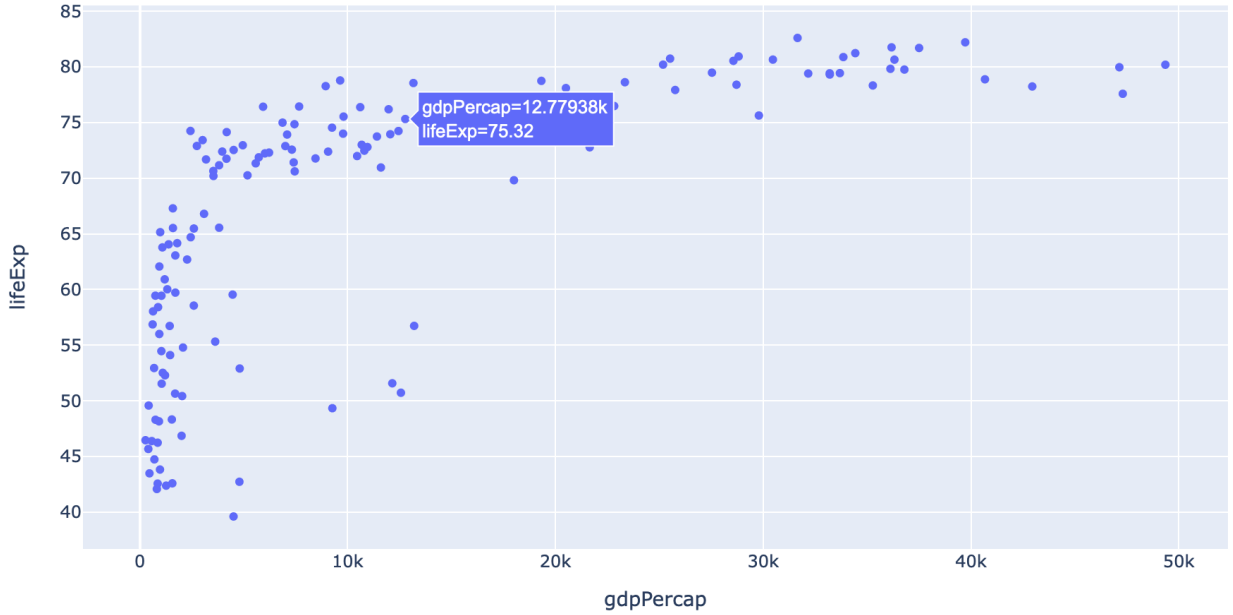
plotly express

- plotly.express plots also include a toolbar with a set of tools similar to what we saw in Bokeh



plotly express

- plotly.express plots also include tooltips with the values for each point in the chart



Pearson

plotly express

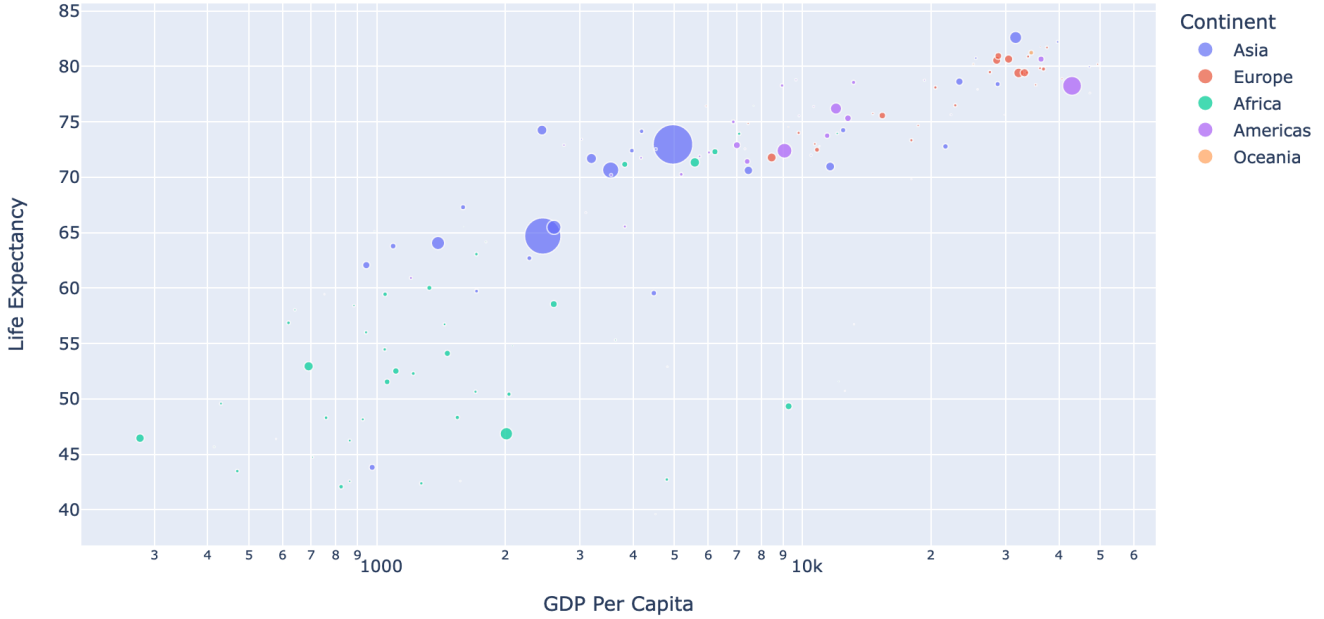
- Each plotting function has a wide range of options
- The labels parameter specifies how to rename the names of the columns in our DataFrame when displaying the respective values, such as in axis labels, legends, tooltips, etc.

```
fig = px.scatter(gapminder,
                 x="gdpPercap",
                 y="lifeExp",
                 size='pop',
                 color="continent",
                 hover_data=["country"],
                 labels={
                     "pop": "Population",
                     "lifeExp": "Life Expectancy",
                     "continent": "Continent",
                     "gdpPercap": "GDP Per Capita",
                     "country": "Country"
                 },
                 log_x=True)
fig.show()
```

plotly express

- Even complex visualizations can be implemented easily

```
fig = px.scatter(gapminder,
                 x="gdpPercap",
                 y="lifeExp",
                 size='pop',
                 color="continent",
                 hover_data=["country"],
                 labels={
                     "pop": "Population",
                     "lifeExp": "Life Expectancy",
                     "continent": "Continent",
                     "gdpPercap": "GDP Per Capita",
                     "country": "Country"
                 },
                 log_x=True)
fig.show()
```



Pearson

plotly express

- plotly.express is optimized to generate pre-formatted plotly plots with minimal lines of code; (similar to what Seaborn does for matplotlib).
- To generate more complex plots, we must use the plotly.graph_objects interface, which allows us to overlay multiple plotly.express plots into single figure

plotly express

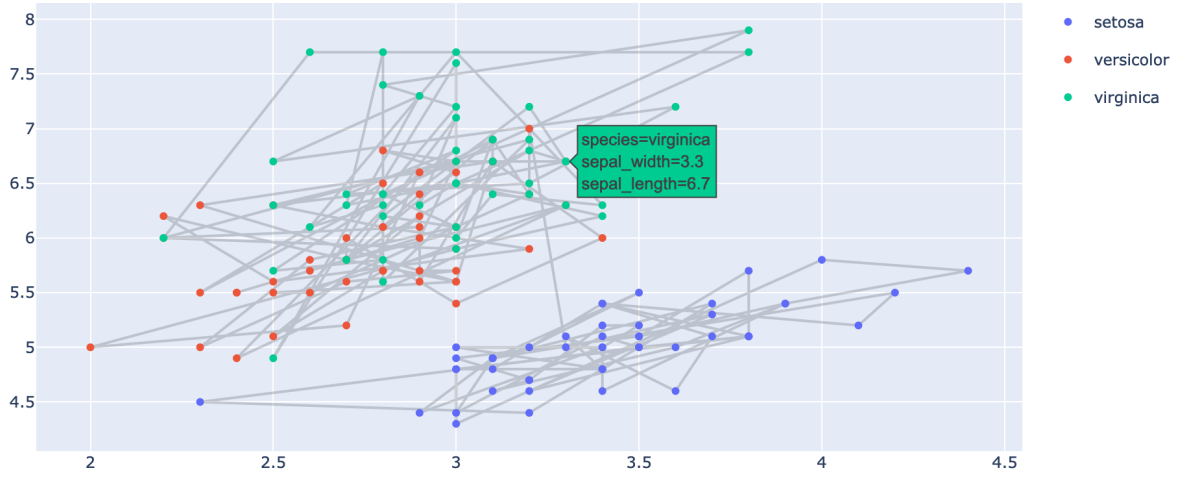
- To overlay two or more plotly.express, we must instantiate a go.Figure object where the data parameter is set to the sum of the data field of the plots we wish to combine
- For example, we can generate a simple scatter plot with lines connecting all the dots, by using:

```
fig1 = px.line(iris, x="sepal_width", y="sepal_length")
fig1.update_traces(line=dict(color = 'rgba(50,50,50,0.2)'))

fig2 = px.scatter(iris, x="sepal_width", y="sepal_length", color="species")

fig3 = go.Figure(data=fig1.data + fig2.data)
fig3.show()
```

plotly express

```
fig1 = px.line(iris, x="sepal_width", y="sepal_length")
fig1.update_traces(line=dict(color = 'rgba(50,50,50,0.2)'))

fig2 = px.scatter(iris, x="sepal_width", y="sepal_length", color="species")

fig3 = go.Figure(data=fig1.data + fig2.data)
fig3.show()
```

- Resulting in:

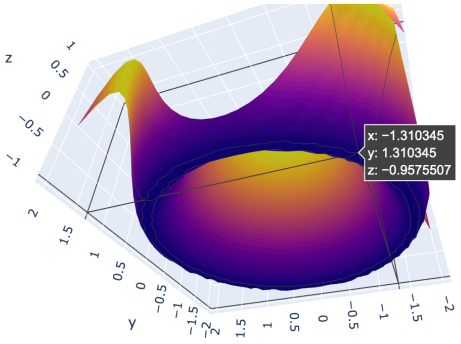


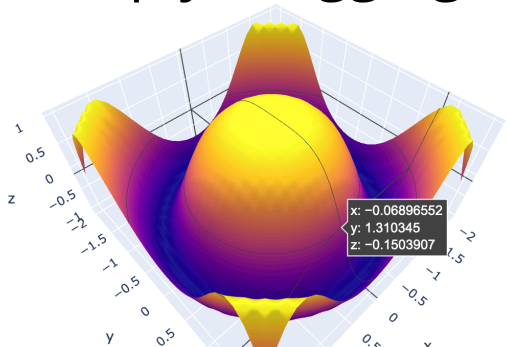
Pearson

Lesson 9.2
3D and Animated Plots

3D plotting

- plotly has extensive support for 3D plots using:
  - px.line_3d() - 3D lines
  - px.scatter_3d() - 3D scatter plot
  - go.Surface() - 3D Surface plot
- 3D images can by rotated by simply dragging the mouse





Animated Plots

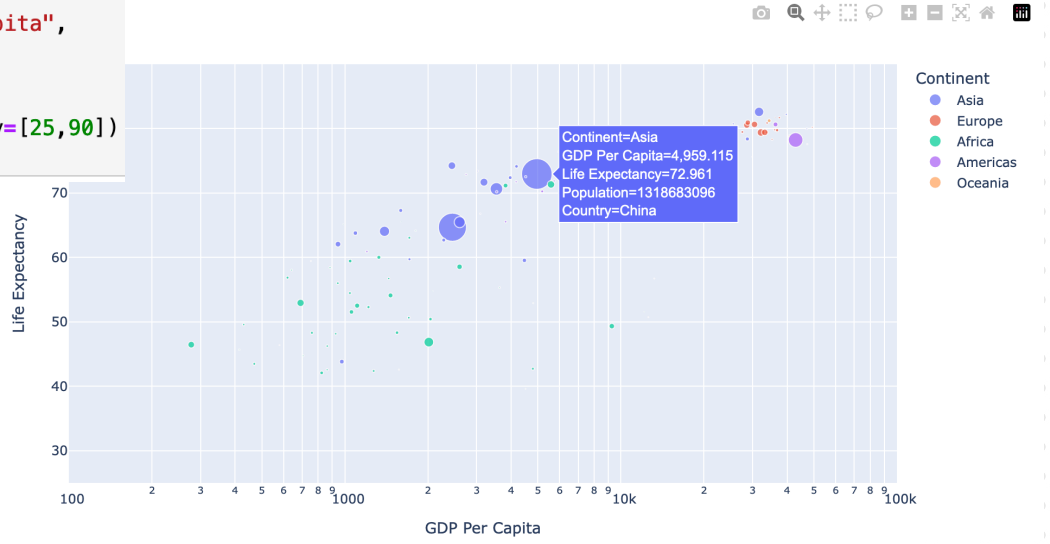
- Support for animations is straightforward.
- We can animate scatter and bar plotly.express plots by providing two extra arguments:
  - animation_frame - the name of the column in the data frame that splits the data into individual frames
  - animation_group - the name of the column specifying the group of values that will be animated
- When these arguments are provided, Play, Stop and Timeline elements are added to the bottom of the figure that allows us to animate the plot

Animated Plots

- A gap minder plot for single year can be generated using:

```
fig = px.scatter(gapminder_2007,
                 x="gdpPercap",
                 y="lifeExp",
                 size='pop',
                 color="continent",
                 hover_data=["country"],
                 labels={
                     "pop": "Population",
                     "lifeExp": "Life Expectancy",
                     "continent": "Continent",
                     "gdpPercap": "GDP Per Capita",
                     "country": "Country"
                 },
                 log_x=True,
                 range_x=[100,100000], range_y=[25,90])
fig.show()
```

- Resulting in:

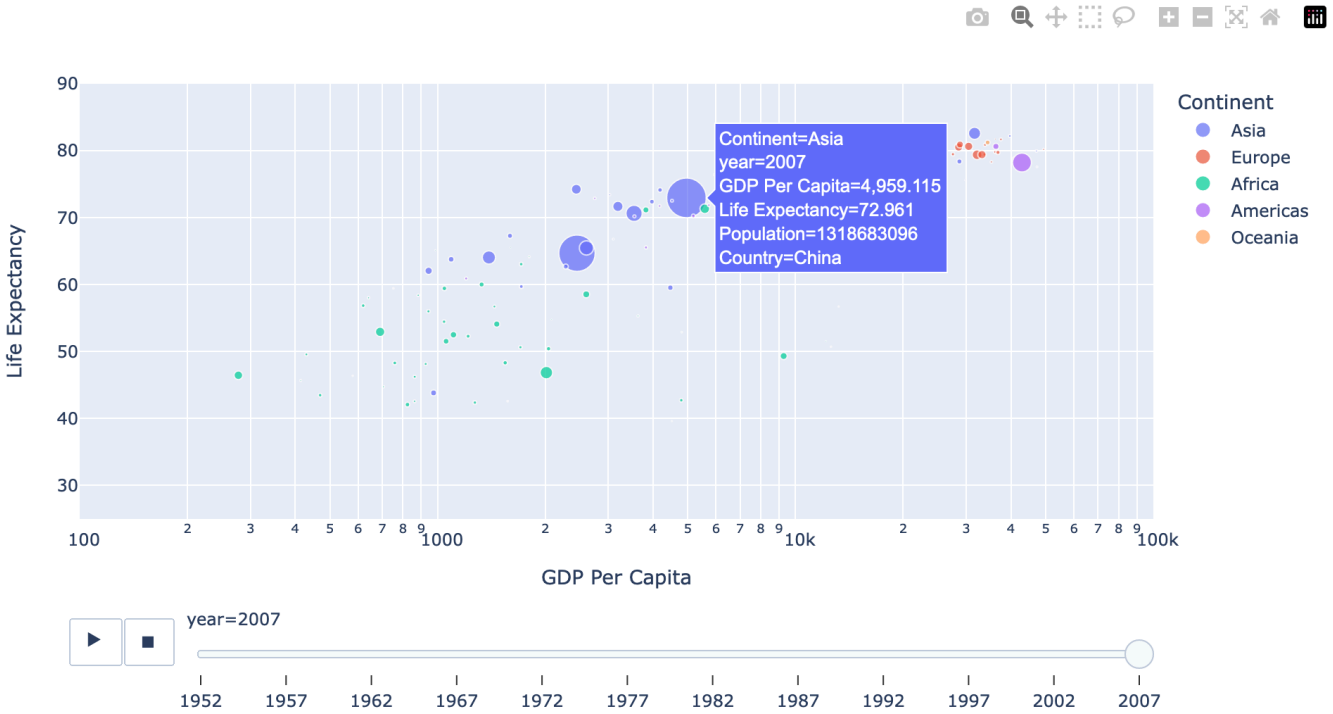


Animated Plots

- To animate the gapminder plot we just need to set the animation_frame and animation_group parameters

```
fig = px.scatter(gapminder,
                 x="gdpPercap",
                 y="lifeExp",
                 size='pop',
                 color="continent",
                 hover_data=["country"],
                 labels={
                     "pop": "Population",
                     "lifeExp": "Life Expectancy",
                     "continent": "Continent",
                     "gdpPercap": "GDP Per Capita",
                     "country": "Country"
                 },
                 animation_frame="year", animation_group="country",
                 log_x=True,
                 range_x=[100,100000], range_y=[25,90]
                )
fig.show()
```

Animated Plots



- Unfortunately, plotly does not allow saving animation

Code - Plotly

https://github.com/DataForScience/Visualization_LL

END