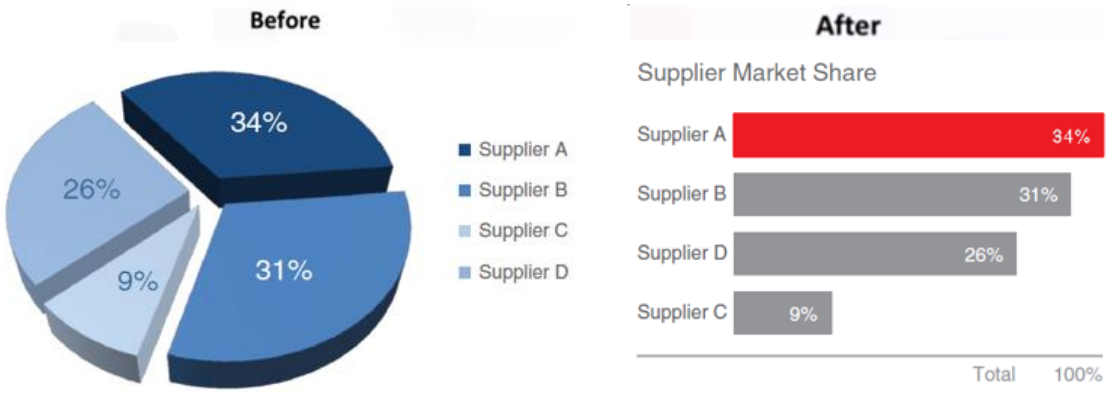
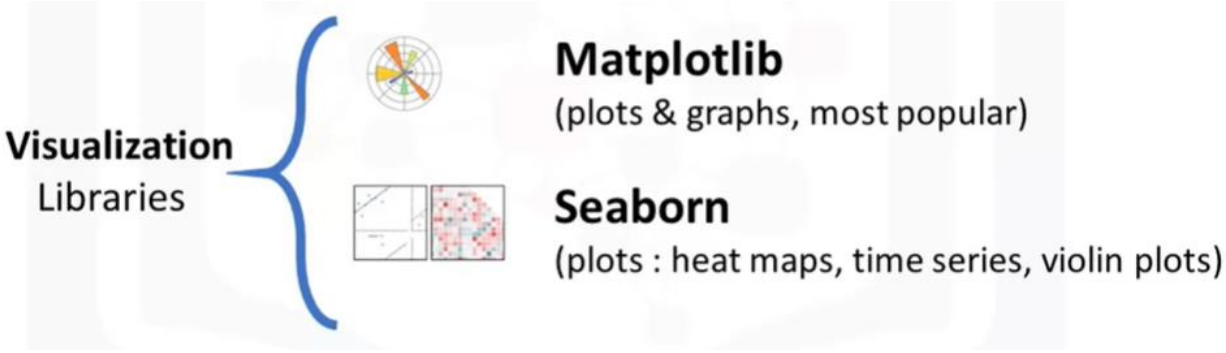
VISUAL REPRESENTATIONS & APIs



[# Removed bg, border, redundant legend, 3-D, & wedges, unordered categories; create simple bar plot for a cleaner & straightforward message on a linear and common baseline ref]

(every customization must be important aspect of the data, meet accessibility std, or requested criteria in vis —at minimalistic approach: **less is more attractive, effective, and impactive**). (Refer. Presentation)



Matplotlib- associated libraries are recommended for creating static graphics due to its compatibility with the analytical steps & comprehensive layouts customizations.

(*ie.* The libs support easy access & operate directly onto a Pandas object.)

—Otherwise, select the best visualization tool / API based on its functionality, advantages & limitations for a given problem and audience.

[# More comprehensive visualizations with code can be found on- Python Graph Gallery, sorted by Python package used: Seaborn, Matplotlib, or Plotly.]

The success of communication depends on how well the following are presented,

(A) Begin by figuring out these questions,
- Who is the audience? • How can I best communicate what they need to know?
- What do you need them to know or do?

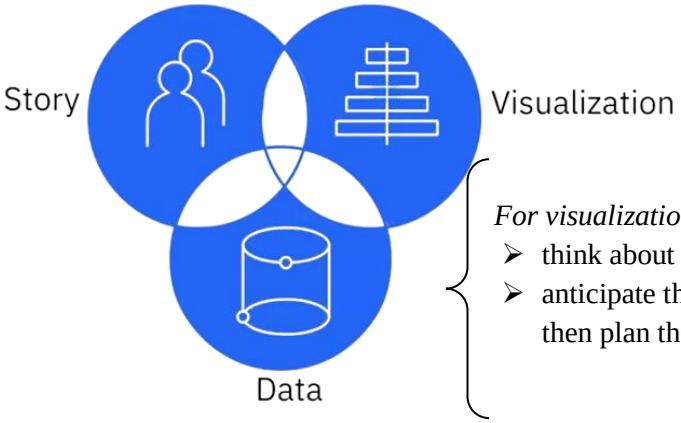
(B) Speak in the language of the organization's business domain to build connection with the audience.
- Understand different metrics used by the industry to meet specific goal in data
  —eg. *ROI tells how well an investment is doing in business*

(C) Identify the best form of communication.
(eg. email -see. email sample / live presentation, periodic report, dashboard)

Basic organizing strategies for a presentation,

- **Chronological** – useful for data that is best understood following a sequence in time.
- **Generic-to-specific** – helps audience to consider an issue before describing how it affects them.
- **Specific-to-generic** – highlights impacts the data can have on a broader scale.

—*eg.* Highlighting the national trend before targeting the effect in specific states / locations.



For visualization to be of value,
- think about the key takeaway for your audience, and
- anticipate their information needs & questions, then plan the visualization accordingly.

(See. Effective_Visuals)

Note: 1. The presentation needs to be framed around the level of information of the audience.
2. Avoid pairing red & green, but blue & orange as it can be difficult for color blind reader;
if a table can show the story at a glance, stick with it for clarity instead of needless graphs.

" It's easy to fall back on the assumption that we all know what we're here for, but reflecting on understanding of the problem and the outcome that needs to be achieved, is a great first step in winning attention : confidence. "

Matplotlib backend mode:

(1) %matplotlib inline -embedding static images of plot

:: the figure cannot be modified once rendered, *ie*. plt.show(); re-generate a new plot if needed.

(2) %matplotlib -embedding interactive plots & plot menu

:: always check & apply to an active figure when a plot action, otherwise renders a new figure.

Note: - The magic command only needs to be configured once per kernel session.
- Plots are reset after each cell is evaluated, so all plotting commands must be in a single cell.

```
# Check & Apply plot background style:
plt.style.available ; plt.style.use('seaborn-whitegrid')
```

Common vector & raster graphic formats to export visualization supported by Matplotlib:

```
fig.canvas.get.get_supported_filetypes()
:: {'eps': 'Encapsulated Postscript',
    'jpeg': 'Joint Photographic Experts Group',
    'jpg'/'jpeg': 'Joint Photographic Experts Group',
    'pdf': 'Portable Document Format',
    'pgf': 'PGF code for LaTex',
    'png': 'Portable Network Graphics',
    'ps': 'Postscript',
    'raw'/'rgba': 'Raw RGBA bitmap',
    'svg': 'Scalable Vector Graphics',
    'svgz': 'Compressed Scalable Vector Graphics',
    'tif'/'tiff': 'Tagged Image File Format'}
```

* *Some file format has multiple extensions due to old windows restricting to 3-letter extension.*

For saving figure to file,

```
fig.savefig('fname.png', dpi=600, facecolor='auto', edgecolor='auto')
```

:: the facecolor & edgecolor are colors of the figure background outside subplots to save, defaulted to follow the current figure automatically.

I. Dual Interfaces.

- Matplotlib was originally written as Python alternative for MATLAB users with analogous scripting interface in the 'pyplot'.

(*however,* the change of reference when updating an overlapping context of plots is clunky)

1. **Method of procedural construct:**

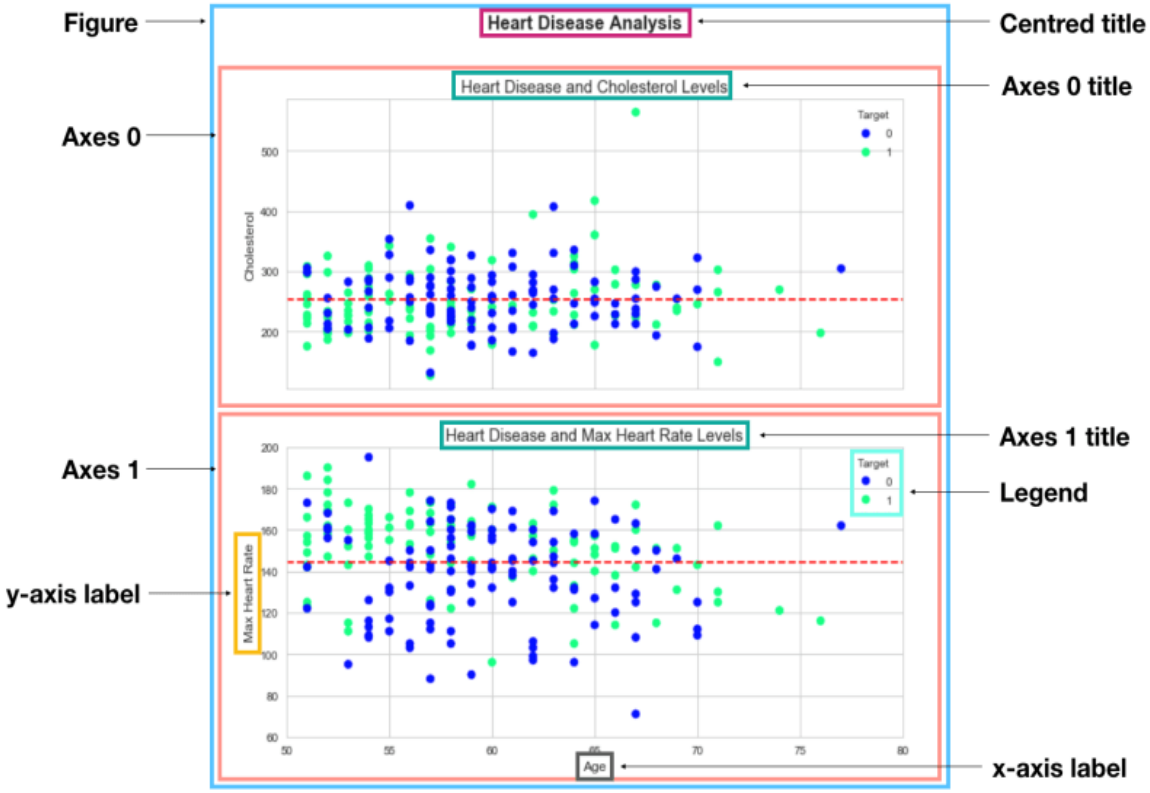
```
# create a blank canvas
plt.figure()

# initialize grid & axes subplot
# (consistent w/ MATLAB's 1-base indexing)
plt.subplot(2, 1, 1)
plt.plot(x, np.sin(x))

# initialize 2nd axes subplot
plt.subplot(2, 1, 2)  # (rows, cols, grid no.)
plt.plot(x, np.cos(x))
```

2. **Method of explicit Figure & Axes spaces:**

```
# initialize a figure instance
fig, ax = plt.subplots(nrows=1, ncols=2)

# Adding plots based on the grid sys
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x))
```

@ Object-oriented approach is a precise reference for constructing multiple complicated plots, but may otherwise exchange for conveniences of a procedural in simple plots.

(Figure vs. Axes)



In Matplotlib's object hierarchy plot,
{figure}, instance of 'plt.Figure' class— a canvas containing generic graphics, text, and labels.
{axes}, instance of 'plt.Axes' class— a bounding space of coordinate system for plot elements.

Note: Do not confuse axes with subplot, which the later is a grid containing axes.

For scripting plot elements setting,

- `plt.xlabel()` • `plt.ylabel()` • `plt.xticks()` • `plt.xlim()` • `plt.title()`

For single / multiple axes elements setting,

```
ax.set(...) / ax.set_axisbelow(True)
ax.xaxis.set(...) # axis ticks / ax.set_xscale('log')  # axis scale
ax.spines.set(...)/ ax.spines[['top', 'right']].set_visible(True)
```

II. plt.plot

```
# Simple line plots.
plt.plot(x, np.sin(x), '-g', label='sin(x)')  # combined ls & color context
plt.plot(x, np.cos(x), linestyle='dotted'/ ':', linewidth=4, label='cos(x)')
```

*Named linestyles only offers 4-basic options, refer: Matplotlib documentation, @ linestyle.

```
plt.axis('equal')  # graphic scale
plt.axis([-1, 11, -1.5, 1.5])  # axes limit: [xmin, xmax, ymin, ymax]
```

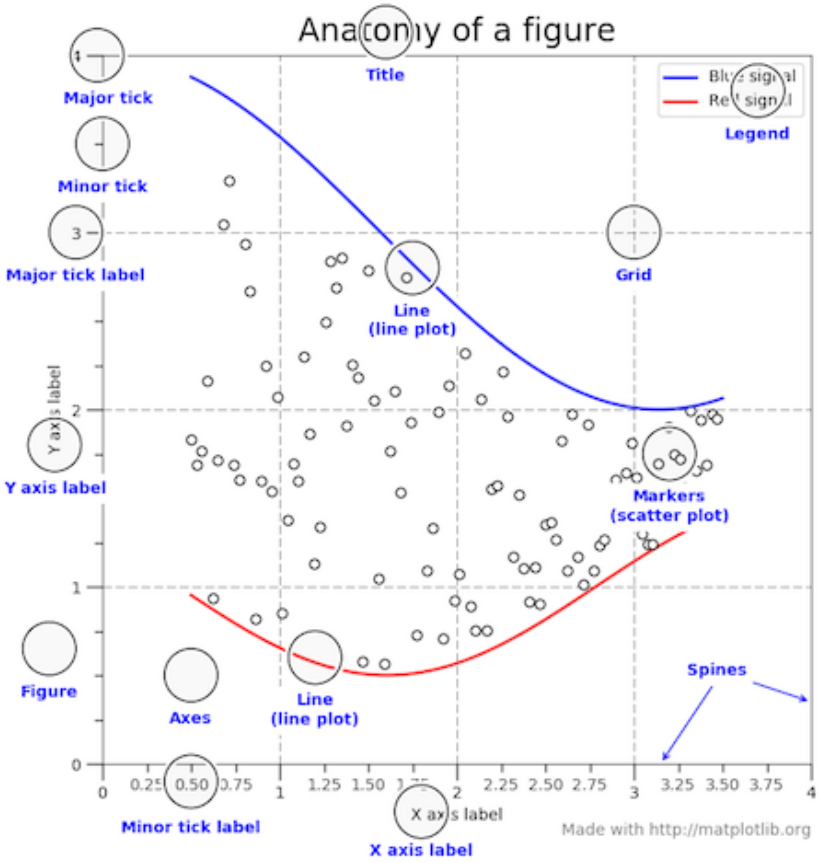
*To display either axis in reverse, simply reverse the order of arguments.

```
plt.title('Trigonometry Curves')
plt.xlabel('x') ; plt.ylabel('f(x)')
plt.legend(loc='best', frameon=False, ncol=2)  # default legend table for labeled plots
```

*Optional parameters can be defined to alter the position, size, and style of the legend table.

```
# Creating step plot with drawstyle.
plt.plot(x, y, drawstyle='steps-post') OR plt.step(x, y, where='post')
```

III. Plot Anatomy



(A) Color Contexts

```
ax.plot(x, np.sin(x - 0), color='blue')          # specify by X11/CSS4 HTML color name
ax.plot(x, np.sin(x - 1), color='xkcd:blue')     # perceived luminance (sharper) color
ax.plot(x, np.sin(x - 2), color='g')             # short color code [rgbcmykw]
ax.plot(x, np.sin(x - 3), color='0.75')          # Grayscale scalar: [0-1]
ax.plot(x, np.sin(x - 4), color='#FFDD44')       # Hex code (RRGGBB): [00-FF]
ax.plot(x, np.sin(x - 5), color=(1.0,2.0,0.3)) # RGB tuple: [0-1]
ax.plot(x, np.sin(x - 6), color='C0')            # Cn color cycle index specification
```

[blue; orange; green; red; purple; brown; pink; gray; olive; cyan]

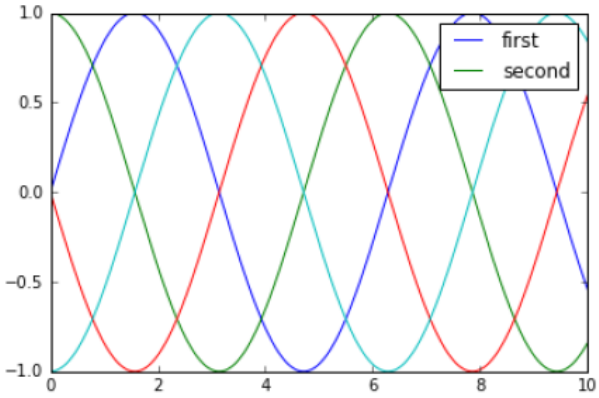
Refer. — [1] Matplotlib(2) CheatSheet for different color names ;
[2] Full listing at- `for name, hex in matplotlib.colors.cnames.items(): print(name, hex)`

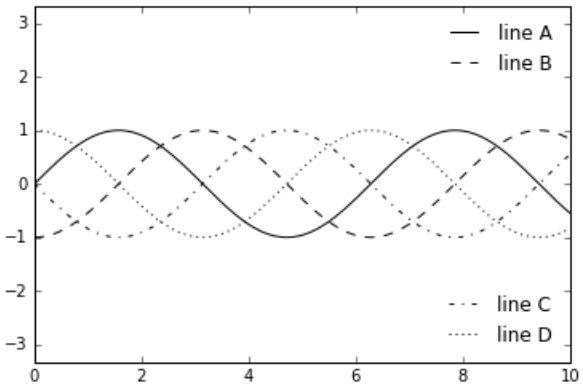
(B) Customizing Legends

Selectively define what plot labels appear in the legend.

```
# Defining legend elements by—
## Labels on plots                          ## Specification in legend
ax.plot(x, y[:, 0], label='first')          lines = ax.plot(x, y)  # multiple lines
ax.plot(x, y[:, 1], label='second')         ax.legend(lines[:2], ['first', 'second'])
ax.plot(x, y[:, 2:])
ax.legend(framealpha=1, frameon=True)
```

[# fancybox & borderpad parameters can be added for round box w/ nicely spaced pad around text]





Note: Pass label='_nolegend_', if 'underfined' label is shown on any excluded element.

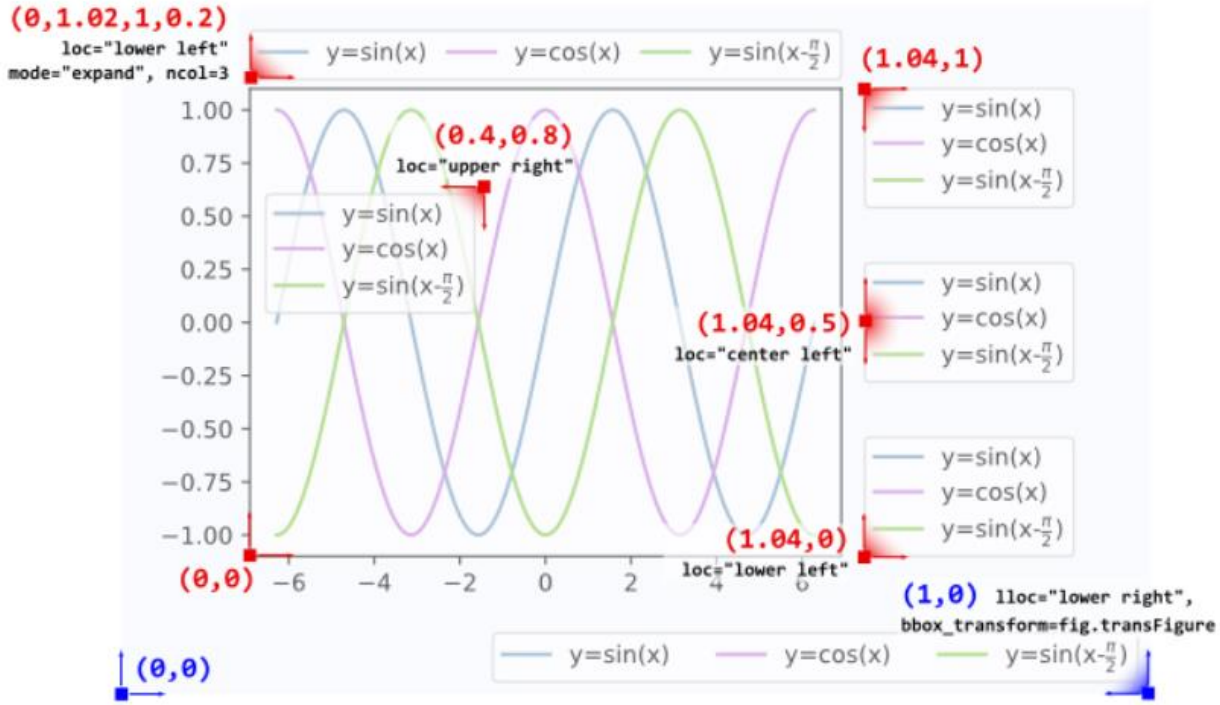
For multiple legend box,

```
# (First legend)
ax.legend(lines[:2], ['line A', 'line B'], loc='upper right', frameon=False)

# (Second legend)
from matplotlib.legend import Legend
leg = Legend(ax, lines[2:], ['line C', 'line D'], loc='lower right', frameon=False)
ax.add_artist(leg)
```

[# Work around by first initiating the built-in legend function, and add the 2nd by a non-integrated, lower-level 'ax.add_artist()' method]

On top, the coordinate of legend can be manipulated w/ aligned loc to the bounding pts / box :

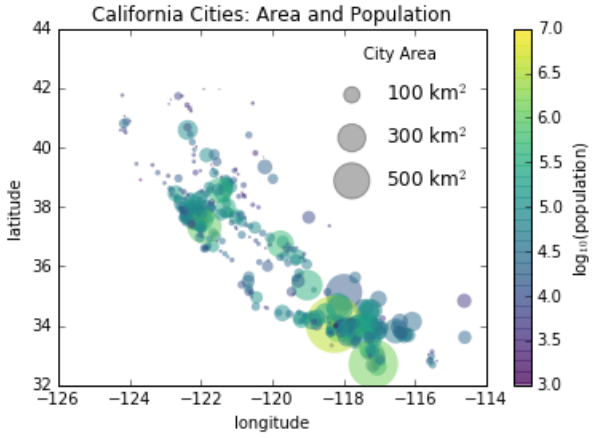
```
ax.legend(bbox_to_anchor=(1.04, 1), loc="left", bbox_transform=ax.transAxes)
```



(Refer. @stackoverflow, for more details about 4-elements tuple of bbox_to_anchor)

For a legend that specifies the scale w/ the size of pts,

```
Scatter the points, using size and color but no label
plt.scatter(lon, lat, label=None, c=np.log10(population), cmap='viridis',
            s=area, linewidth=0, alpha=0.5)

# Create an empty allocation plot to associate customized labels
# as '.legend' will always reference the labeled elements on plt.
for area in [100, 300, 500]:
    plt.scatter([], [], c='k', alpha=0.3, s=area, label=str(area) + ' km$^2$')
    plt.legend(scatterpoints=1, frameon=False, labelspacing=1, title='City Area')
```



*Matplotlib interprets Cifrão ($) as the beginning (or ending) of a LaTex math mode.

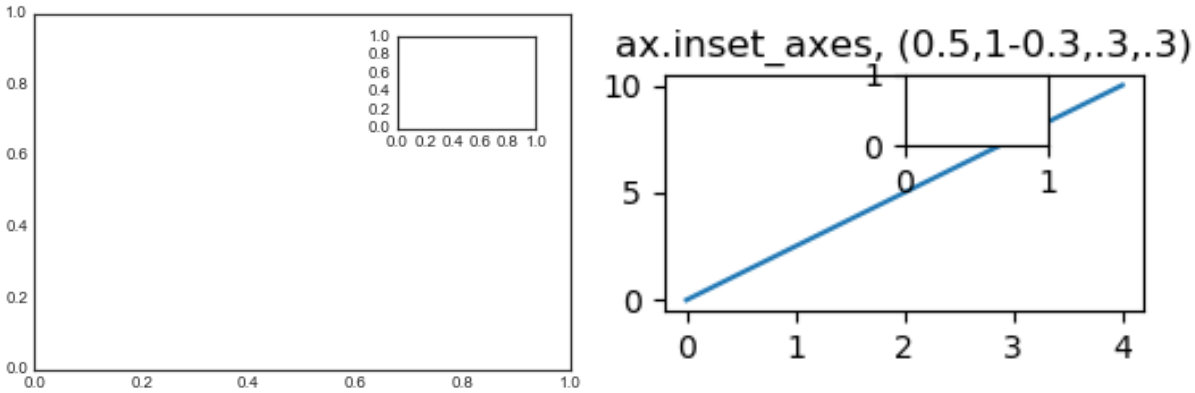
(C) Multiple Subplots

i. plt.axes: Inset axes

```
from mpl_toolkits.axes_grid1.set_locator import inset_axes
```

[1]
```
ax = plt.subplots()   # standard axes
ax_ins = inset_axes(ax, width="100%", height="100%", loc='lower left',
                    bbox_to_anchor=(0.65, 0.65, 0.2, 0.2), bbox_transform=ax.transAxes)
```

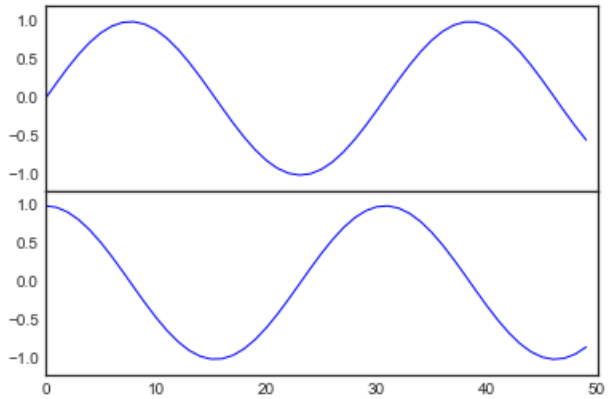
[2]
```
fig = plt.figure()
ax = fig.add_subplot()
ax_ins = ax.inset_axes([x0, y0, w, h], xticklabels, xlim, ...)   # starting pt & relative size
```



[# The prior allows for a padding around the axes (applied by default), but the later is more concise.]

For arbitrary axes shapes, adding axes individually can be a more convenient interface.

```
fig = plt.figure()
ax1 = fig.add_axes([0.1, 0.5, 0.8, 0.4], xticklabels=[], ylim=(-1.2, 1.2))
ax2 = fig.add_axes([0.1, 0.1, 0.8, 0.4], ylim=(-1.2, 1.2))
```

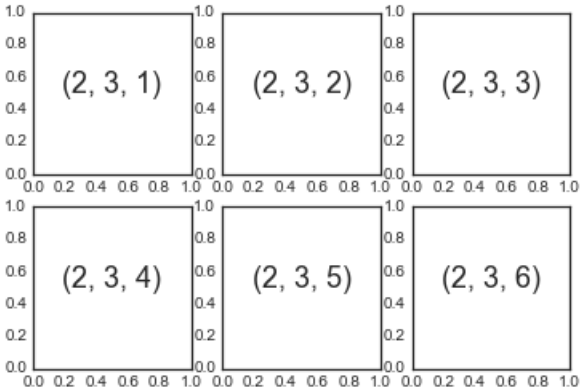


[# [x, y, width, height] at pos 10% & 50% {0.1 + 0.4} of fig. height.]

ii. plt.subplots: Grids of subplots

- create a full grid of subplots, and returning them in a NumPy array.

```
fig, ax = plt.subplots(nrows=2, ncols=3, sharex='col', sharey='row')
# axes are in 2-D array, indexed by [r, c]
for i in range(2):
    for j in range(3):
        ax[i, j].text(0.5, 0.5, str((i, j)), fontsize=18, ha='center')
```



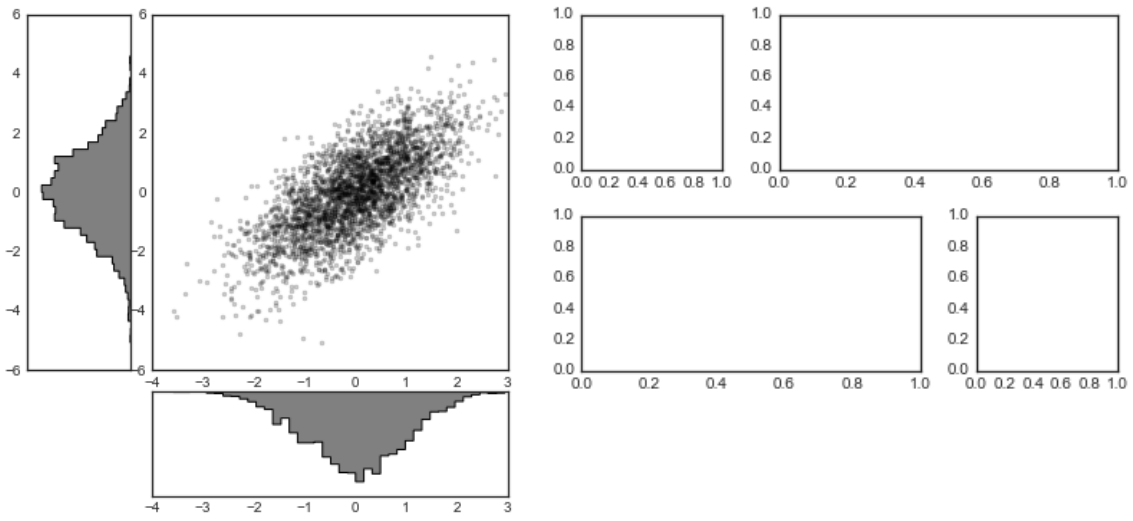
Note: Each subplot can be selected by indexing to have its own tick locator & formatter objects.

iii. plt.GridSpec: Arbitrary grid arrangements

- create subplots that span multiple rows grid's rows & columns.

Useful for creating multi-axes histogram plots like:

```
fig = plt.figure(figsize=(6, 6))
grid = plt.GridSpec(4, 4, hspace=0.2, wspace=0.2)   # rows, cols & pct figure spacing
main_ax = fig.add_subplot(grid[:-1, 1:])
y_hist = fig.add_subplot(grid[:-1, 0], xticklabels=[], sharey=main_ax)
x_hist = fig.add_subplot(grid[-1, 1:], yticklabels=[], sharex=main_ax)

# Scatter points on main axes
main_ax.plot(x, y, 'ok', markersize=3, alpha=0.2)

# Histogram on the attached axes
x_hist.hist(x, 40, histtype='stepfilled', orientation='vertical', color='gray')
x_hist.invert_yaxis()

y_hist.hist(y, 40, histtype='stepfilled', orientation='horizontal', color='gray')
y_hist.invert_xaxis()
```



(D) Customizing Ticks Scale

Matplotlib's `ScaleBase` has a variety of scale definition which can be defined for an axis.

```
fig, ax = plt.subplots()
ax.set(xscale='log', yscale='log', …)
```

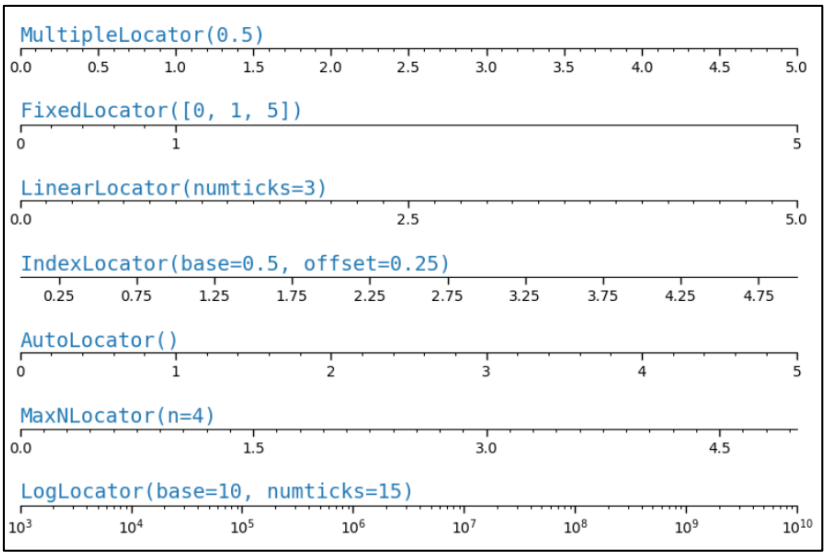
*the common scale specification incl.- 'linear', 'log', 'logit', 'asinh', 'symlog', 'function' *-custom scale function*

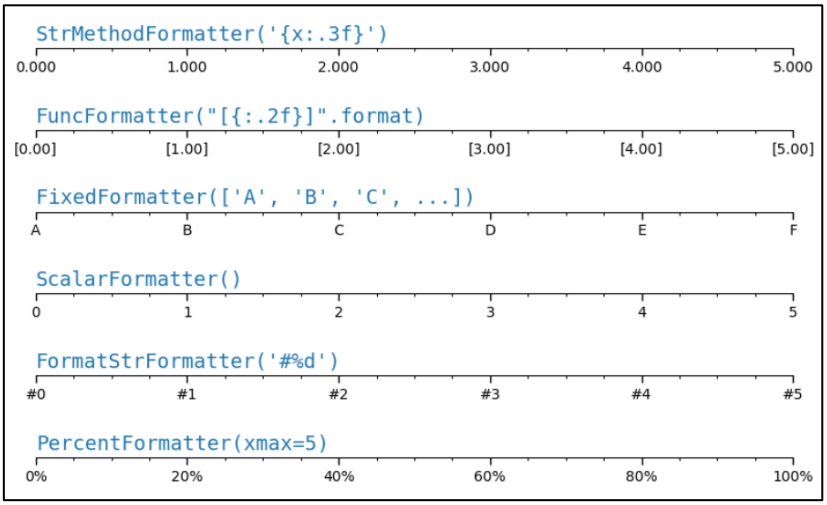
Similarly, the major & minor axes' ticks can be manipulated at—

```
# Check major & minor ticks / labels properties
ax.xaxis.get_major_locator()  ;   ax.xaxis.get_major_formatter()
ax.xaxis.get_minor_locator()  ;   ax.xaxis.get_minor_formatter()
::<matplotlib.ticker.LogLocator object at 0x107530cc0>
::<matplotlib.ticker.LogFormatterMathtext object at 0x107512780>
```

*The existing tickers are often setup, but we can freely customize by changing the formatter & locator objects of each axis.

```
# Customizing tick locations & labels:
ax.yaxis.set_major_locator(plt.NullLocator())    or,    ax.set_yticks( ) # Arbitrary
ax.xaxis.set_major_formatter(plt.NullFormatter())       ax.set_xlabel( )
```



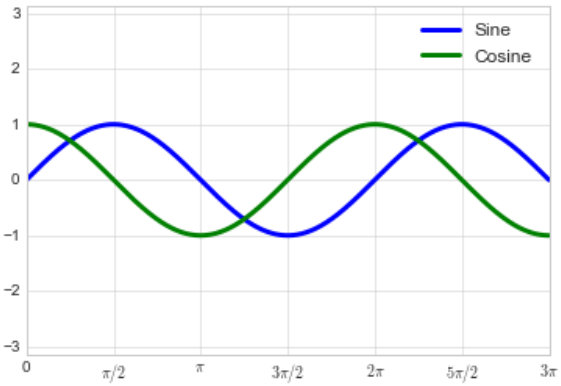


[1] For simple string formatting- '{x} km' or, typical list of labels- 'ax.set_xlabel([…])' can be used.

[2] StrMethodFormatter – *new style* f-string / .format ; FormatStrFormatter – *old style* (%) formatting

To employ fancy string labels from a user-defined function,

```
def format_func(value, tick_number):
    # find number of multiples of pi/2
    N = int(np.round(2 * value / np.pi))
    if N == 0:
        return "0"
    elif N == 1:
        return r"$\pi/2$"
    elif N == 2:
        return r"${0}\pi/2$".format(N)
    else:
        return r"${0}\pi$".format(N // 2)
```

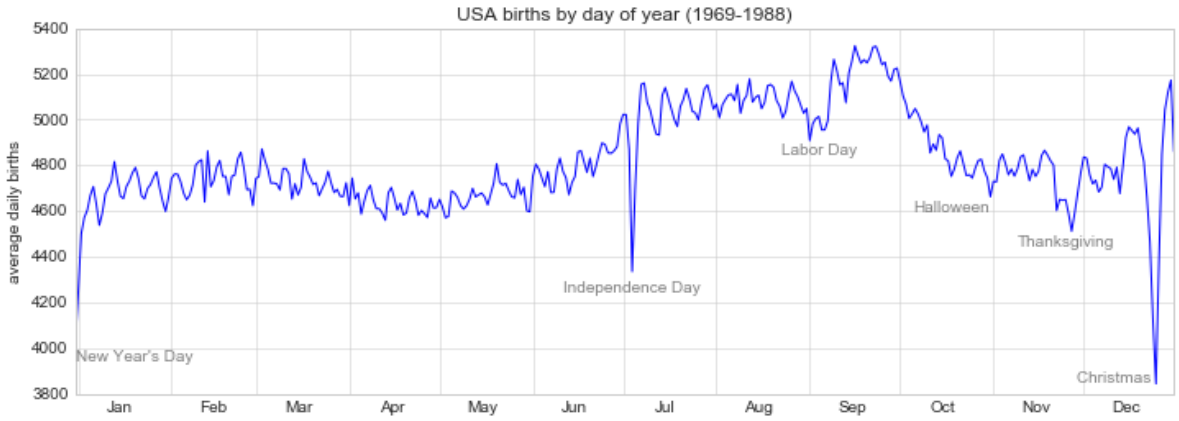



```
axis.xaxis.set_major_formatter(plt.FuncFormatter(format_func))
fig  # trigger figure updates
```

(E) Annotation & Shapes

- incl. text, arrows, or shapes as pointer / container for stressing on critical information.

```
# Consolidated font style
style = dict(fontsize=10, color='gray')

# Anchor text to position on plot by: [1]index (str) / [2]coordinate (int, float) axes labels
ax.text('2012-1-1', 3950, "New Year's Day", **style, transform=ax.transData)
ax.text('2012-7-4', 4250, "Independence Day", ha='center', **style)
ax.text('2012-9-4', 4850, "Labor Day", ha='center', **style)
ax.text('2012-10-31', 4600, "Halloween", ha='right', **style)
ax.text('2012-11-25', 4450, "Thanksgiving", ha='center', **style)
ax.text('2012-12-25', 3850, "Christmas", ha='right', **style)


# Manipulate x-axis ticks scale centered to month
ax.xaxis.set_major_locator(mpl.dates.MonthLocator())  # following ts-locator by month
ax.xaxis.set_minor_locator(mpl.dates.MonthLocator(bymonthday=15))
ax.xaxis.set_major_formatter(plt.NullFormatter())
ax.xaxis.set_minor_formatter(mpl.dates.DateFormatter('%b'))
```



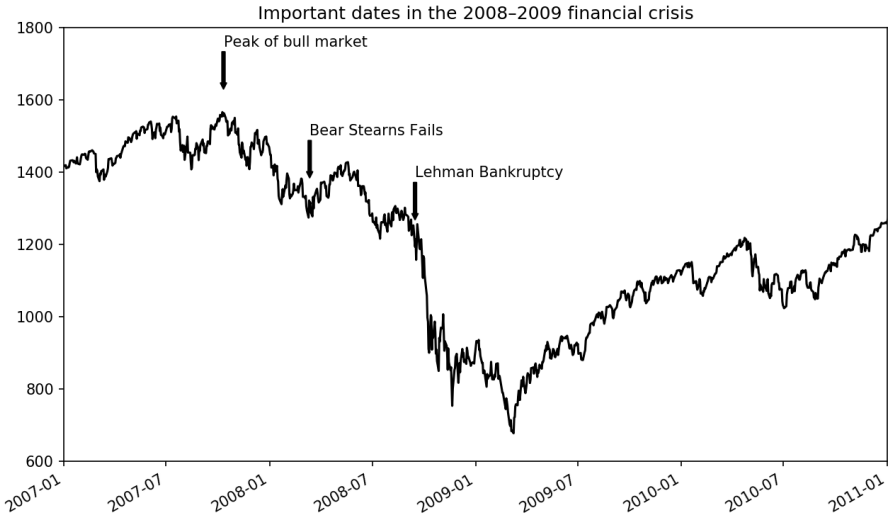
! Often, it is more effective to label a data visualization instead of using a legend as:

✓ it can be placed near the data, more accessible by not relying on ability to interpret color, and it allows for text explanations to be placed directly on the visualization.

Note: Every palette also has a reversed version with the same name plus the suffix _r (eg. RdBu_r).

For simple annotation of combined text & arrow:

```
crisis_data = [(datetime(2007, 10, 11), 'Peak of bull market'),
               (datetime(2008, 3, 12), 'Bear Stearns Fails'),
               (datetime(2008, 9, 15), 'Lehman Bankruptcy')]

for date, label in crisis_data:
    ax.annotate(label, xy=(date, spx.asof(date) + 75), xytext=(date, spx.asof(date) + 225),
                arrowprops=dict(fc='black', headwidth=4, width=2, headlength=4),
                horizontalalignment='left', verticalalignment='top')  #
```



Similarly, the anchor reference can be modified to follow a different scheme:

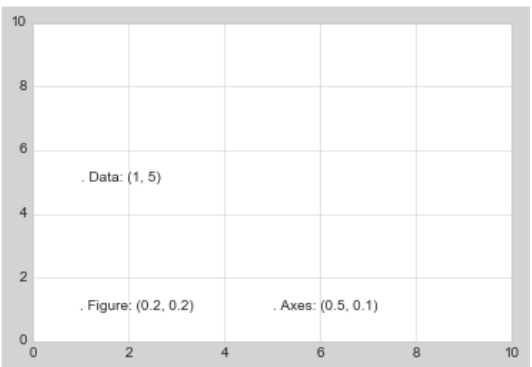
(*a methematical transformation for translating btw coordinate sys. & pixel values represented on screen*)

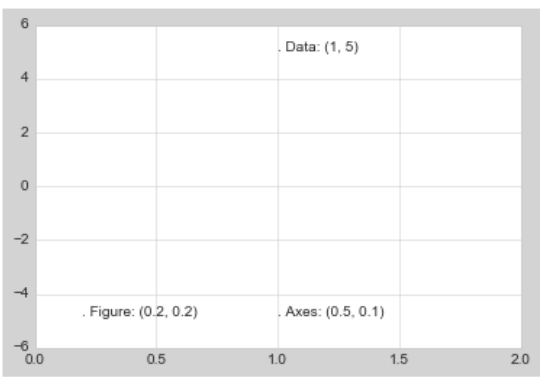
i. ax.transData— Transform associated with data on *x*- & *y*- plot labels.

ii. ax.transAxes— Transform associated with fraction of the axes size.

iii. fig.transFigure— Transform associated with fraction the figure dimension.

```
fig, ax = plt.subplots(facecolor='lightgray')
ax.axis([0, 10, 0, 10])

# Default- allocate according to data labels:
ax.text(1, 5, ". Data: (1, 5)", transform=ax.transData)
# At (50%, 20%) arbitrary axes range:
ax.text(0.5, 0.1, ". Axes: (0.5, 0.1)", transform=ax.transAxes)
% Defining 20% figure dim; useful for annotation outside of axes range:
ax.text(0.2, 0.2, ". Figure: (0.2, 0.2)", transform=fig.transFigure)
```

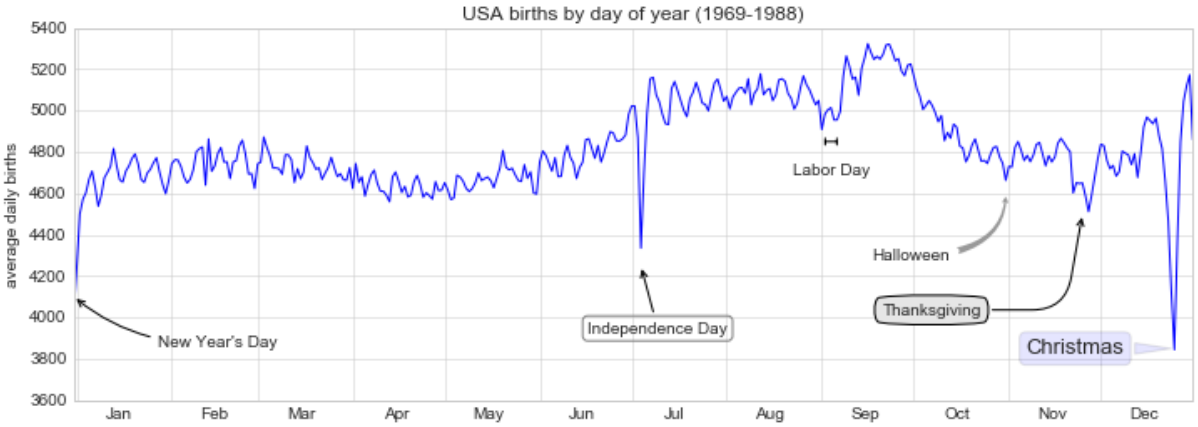




Note: Every palette also has a reversed version with the same name plus the suffix _r (eg. RdBu_r).

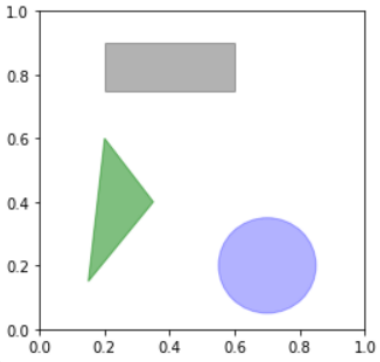
Besides the default standard, a variety of fancy annotations may be defined as:

```
ax.annotate("New Year's Day", xy=('2012-1-1', 4100), xycoords='data',
            xytext=(50, -30), textcoords='offset points',
            arrowprops=dict(arrowstyle="->", connectionsstyle="arc3,rad=-0.2"))

# Define arrow & text separately for greater text spacing (from bounded to arrow tail)
ax.annotate("Labor Day", xy=('2012-9-4', 4850), xycoords='data', ha='center',
            xytext=(0, -20), textcoords='offset points')
ax.annotate('', xy=('2012-9-1', 4850), xytext=('2012-9-7', 4850),
            xycoords='data', textcoords='data',
            arrowprops={'arrowstyle': '|-|,widthA=0.2,widthB=0.2'})

ax.annotate("Halloween", xy=('2012-10-31', 4600), xycoords='data',
            xytext=(-80, -40), textcoords='offset points',
            bbox=dict(boxstyle="round4,pad=.5", fc="0.9"),
            arrowprops=dict(arrowstyle="fancy", fc="0.6", ec="none"
                            connectionstyle="angle,angleA=0,angleB=80,rad=20"))

ax.annotate("Christmas", xy=('2012-12-25', 3850), xycoords='data', xytext=(-30, 0),
            textcoords='offset points', size=13, ha='right', va='center',
            bbox=dict(boxstyle="round", alpha=0.1),
                  arrowprops=dict(arrowstyle="wedge,tail_width=0.5", alpha=0.1))
```



To draw different types of shape (*ie.* geometries) on plot,

```
from matplotlib.patches import Rectangle, Circle, Polygon

fig, ax = plt.subplots()
# Shapes are referred as patches in Matplotlib
rect = Rectangle((0.2, 0.75), 0.4, 0.15,
                 color='black', alpha=0.3)
circ = Circle((0.7, 0.2), 0.15, color='blue', alpha=0.3)
pgon = Polygon([[0.15, 0.15], [0.35, 0.4], [0.2, 0.6]],
               color='green', alpha=0.5)

ax.add_patch(rect); ax.add_patch(circ); ax.add_patch(pgon);
```



[# Unlike plot, shapes are primitive artist class in – 'matplotlib.patches']

(F) Customizing Colorbars

- a separate axes scale based on color for plot of points, lines, or regions.

Tersely, the color palettes are classified in three broad categories:

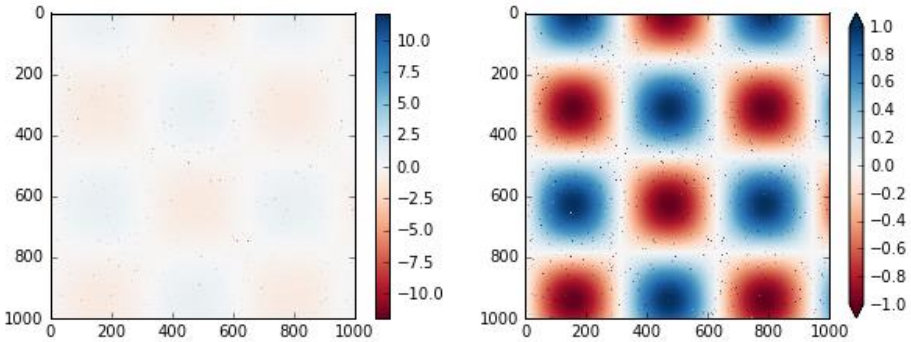
- **Sequential colormaps**— *uniform intensity progression, good for representing numeric data w/ continuous relative values* (*eg.* cubehelix / viridis)
- **Divergent colormaps**— *two dominant hues, good for representing numeric data w/ interesting high {+ve} & low {-ve} from an insignificant midpoint* (*eg.* RdBu/ PuOr)
- **Qualitative colormaps**— *Mix color variations, good for representing categorical data* (*eg.* rainbow/ jet)

[# Refer. plt.cm.<TAB> / Matplotlib's user guide, @Choosing Colormaps]

(Color Limits & Extensions)

- to manipulate color range & indicate the existence of unbounded outliers/ noise from range of interest with arrow extends.

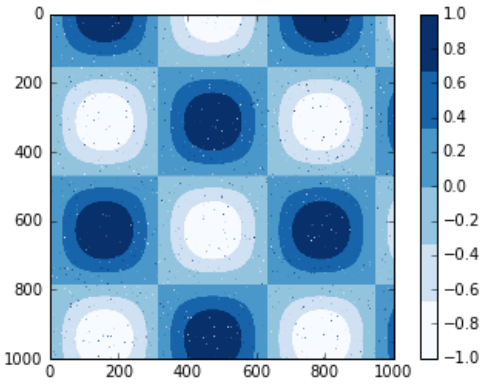
```
plt.subplot(1, 2, 1)
plt.imshow(I, cmap='RdBu')
plt.colorbar()

# Specify color limits in the range of interest & extended arrowhead
plt.subplot(1, 2, 2)
plt.imshow(I, cmap='RdBu')
plt.colorbar(extend='both')
plt.clim(-1, 1)
```



[# To prevent extreme values from diminishing the color sensitivity in the effective range]

To divide color density into discrete finite ranks,

```
plt.imshow(I, cmap=plt.cm.get_cmap('Blues', 6))
plt.colorbar(ticks=range(6), location='right',
             orientation='vertical')
plt.clim(-1, 1)
```



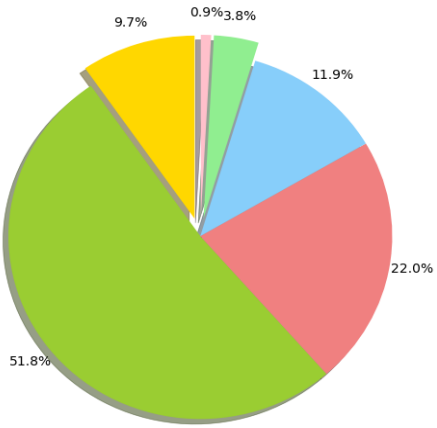
Note: Every palette also has a reversed version with the same name plus the suffix _r (eg. RdBu_r).

MATPLOTLIB PLOTS & VISUALIZATIONS

@Pandas has a DataFrame-specific plot: `df.plot` that uses Matplotlib in the default backend for various kind— [bar, barh, hist, box, kde, area, pie, scatter, hexbin];
taking *index* as *x-axis*, *value* as *y-axis* & *columns* as *groups*. (*transpose the data if needed*)

I. Pie / Bar Plot

```
expl_list = [0.1, 0, 0, 0, 0.1, 0.1]
ax = df['c'].plot.pie(figsize=(15,6), autopct='%1.1f%%', startangle=90,
                      labels=None, pctdistance=1.12, explode=expl_list)
```



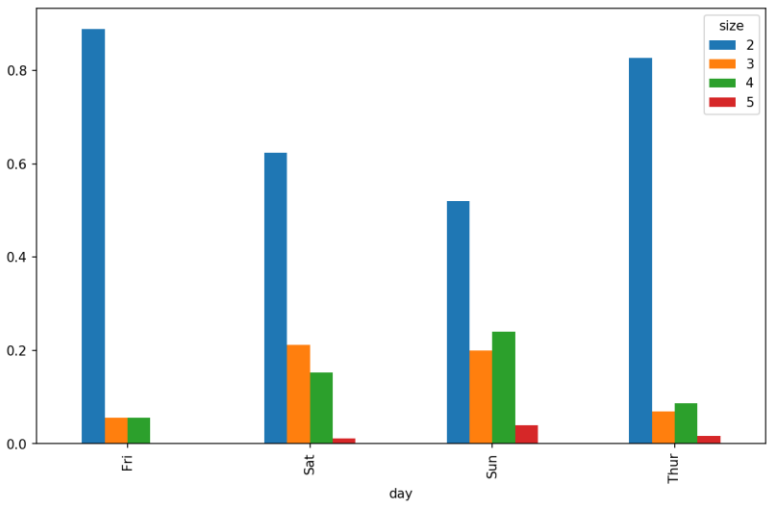
autopct – format string for the labels
explode – protruding ratio of the classes in sequence
pctdistance – radial pct distance of label from the center

To create a bar plot on fraction of parties by size on each day,

```
party_counts = (pd.crosstab(df['day'], df['size'])
                .reindex(index=['Thur', 'Fri', 'Sat', 'Sun']))
```

```
:: size  1   2   3   4  5  6
   day
   Thur  1  48   4   5  1  3
   Fri   1  16   1   1  0  0
   Sat   2  53  18  13  1  0
   Sun   0  39  15  18  3  1
```

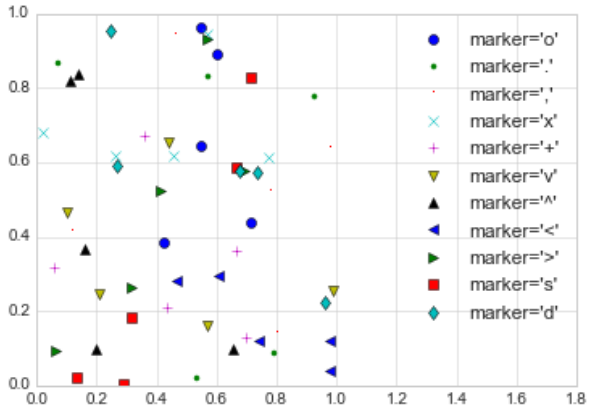
```
party_counts = party_counts.loc[:, 2:5]   #excl. insignificant 1 & 6 pax parties
party_pcts = party_counts.div(party_counts.sum(axis='columns'))   # normalize to sum
party_pcts.plot.bar(stacked=False)
```



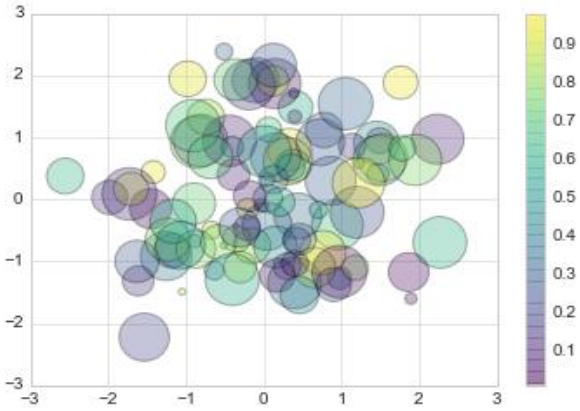
[# Otherwise, use Seaborn API for simpler operation w/o needing to summarize data first]

II. Scatterplot

```
# Points Marker  –fmt allows basic formatting for color, marker & linestyle. ( NOT INCLUDED IN Line2D )
(1) plt.plot(x, y, fmt='o', markersize=15, markerfacecolor='white',
             markeredgecolor='gray', markeredgewidth=2)
(2) plt.scatter(x, y, facecolors='white', edgecolors='gray', linewidths=2)
```



plt.plot



plt.scatter

- More efficient with large (n>100) records.
  - Allows only one property for each plot.
- More work is required for specifying properties.
  - Detailed list of sizes & colors for each pt.

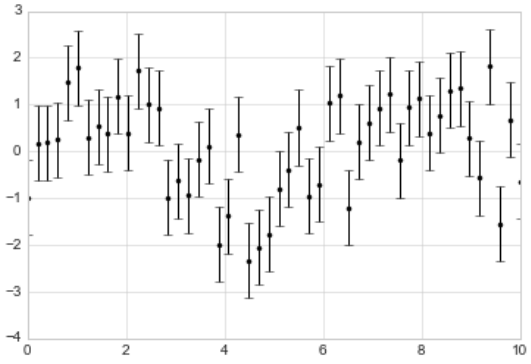
**To look at scatterplots among groups of vars, it is easier to use Seaborn 'sns.pairplot' instead.

III. Visualizing Errors

- to accurately account for the scale of variations / errors in any scientific measurement.
  (*eg.* report information with range of uncertainty: 74 ± 2.5)

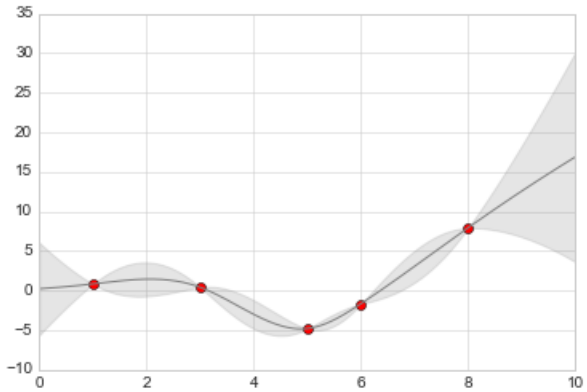
(Basic Errorbars)

```
x = np.linspace(0, 10, 50)
dy = 0.8   # xerr can be added if needed
y = np.sin(x) + dy * np.random.randn(50)
plt.errorbar(x, y, yerr=dy, fmt= fmt='.k',
             elinewidth=2)
```



(Continuous Errors)

For a continuous errors from a model fitting, a combined *markers* & *filled curves* can be used.



```
x = df['predictors']; y = df['target'])
yhat = model.predict(x)
dy = 2 * np.sqrt(mean_squared_error(y, yhat))
# 2 * sigma ~ 95% C.I.

plt.plot(x, y, 'or')
plt.plot(x, yhat, '-', color='gray')

plt.fill_between(x, yhat – dy, yhat + dy,
                 color='gray', alpha=0.2)
```

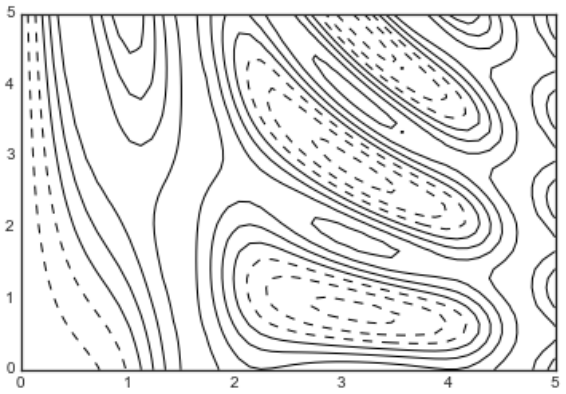
IV. Density & Contour Plot

- to visualize the property of 2-d function (eg. multi-dimensional heat conduction).

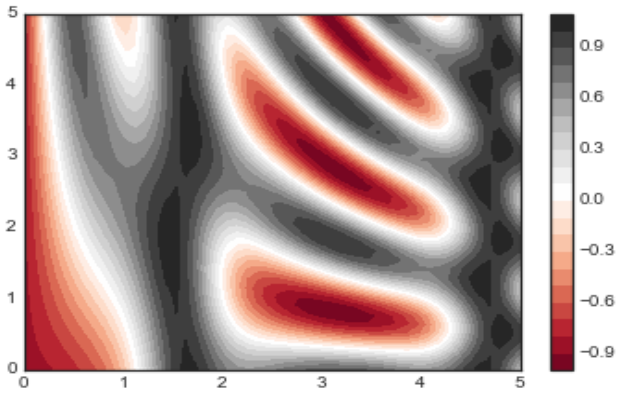
! All input data is required in the form of 2-D coordinated grids, with Z evaluated for each pt.

```
func = lambda x, y: np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
x = np.linspace(0, 5, 51)
y = np.linspace(0, 5, 41)
X, Y = np.meshgrid(x, y)
Z = func(X, Y)
```

```
# Contour plot
plt.contour(X, Y, Z, colors='k')
```



```
# Density plot
plt.contourf(X, Y, Z, 20, cmap='RdGy')
plt.colorbar()   # 20 equally spaced Z-intervals
```

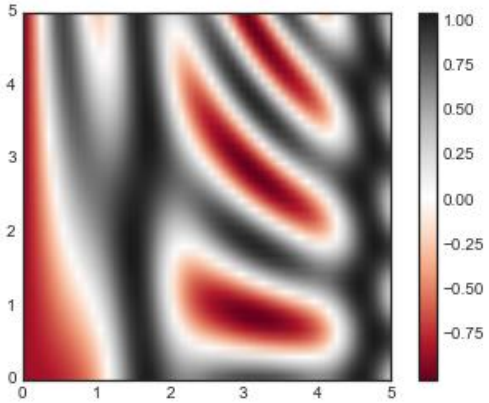


[# The coordinate represents ordered (x, y) values, while the contour line/ color density represents z values]

However, the discrete color steps are not ideal as representation of a continuous Z- values.

(*ie.* setting to a very high no. of intervals is possible but inefficient, as the number of new polygons to be rendered grows with each level)

Alternatively, an image plot can be used.

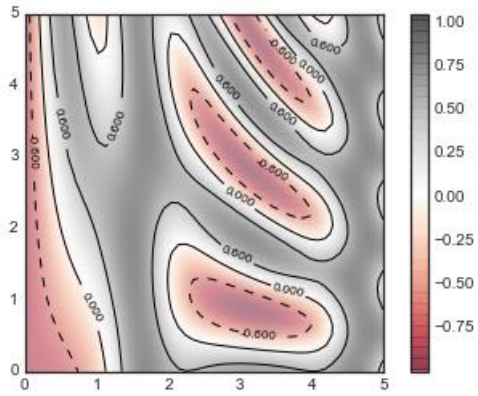



```
plt.imshow(Z, extent[0, 5, 0, 5],
           origin='lower', cmap='RdGy')
plt.colorbar()
plt.axis(aspect='image')
```

- Requires manual specification range [x_{min}, x_{max}, y_{min}, y_{max}]
- Specify origin $(0,0)$ at lower rather than upper left corner.
- Adjust aspect ratio to follow image for matching x & y units

For a combined contour & image plot:

```
contours = plt.contour(X, Y, Z, 3, colors='black')
plt.clabel(contours, inline=True, fontsize=8)
plt.imshow(Z, extent=[0, 5, 0, 5],
           origin='lower', cmap='RdGy', alpha=0.5)
plt.colorbar()
```

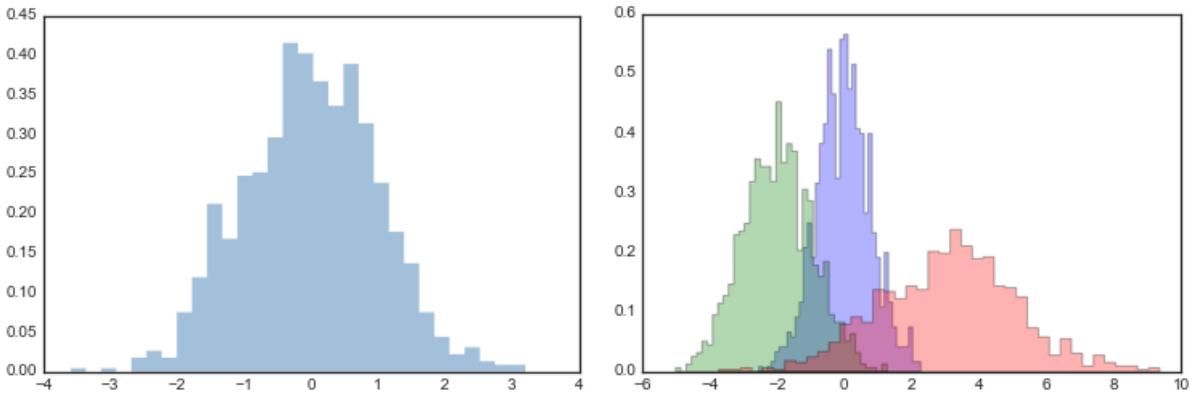


V. Histogram Binnings

- 1-D : dividing continuous values into intervals ; 2-D : dividing coordinated pts into areas

```
counts, bin_edges = np.histogram(data, bins=30)
```

```
# Simply Probability Density Histogram
plt.hist(data, bins=30, density=True, alpha=0.5,
         histtype='stepfilled', color='steelblue', edgecolor=None)
```



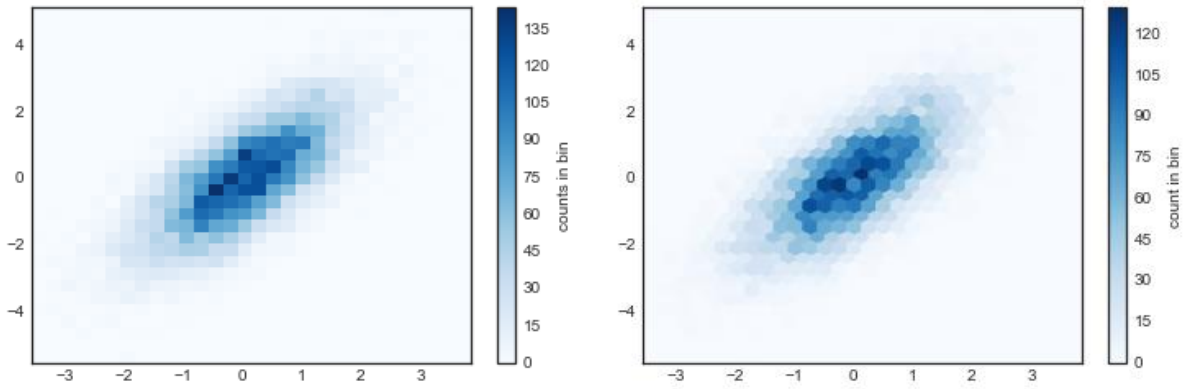
(1) The combined settings of 'stepfilled' & transparency ~ 0.5 can be useful for comparing several histogram plots that are likely to be overlapping.

(2) Refer the @Patch properties for more available shape parameters' setting.

```
counts, xedges, yedges = np.histogram2d(x, y, bins=30)
```

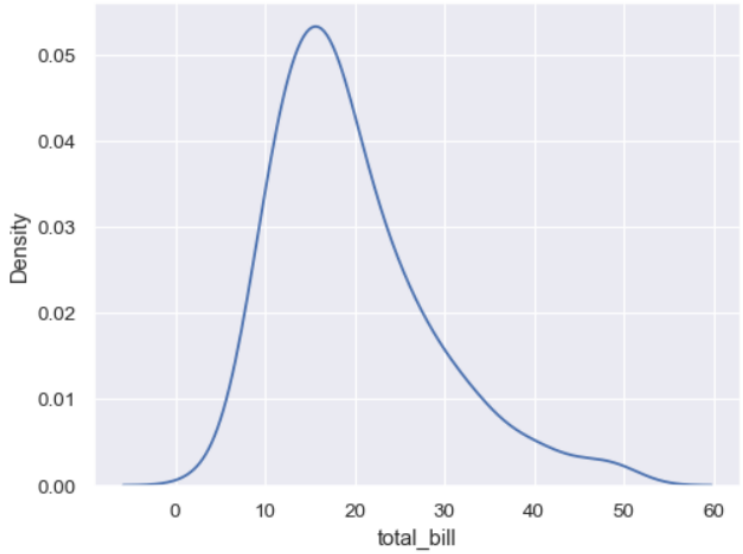
```
# 2-D Histogram {Square binnings}
plt.hist2d(x, y, bins=30, cmap='Blues')
cb = plt.colorbar()
cb.set_label('counts in bin')
```

```
# 2-D Histogram {Hexagonal binnings}
plt.hexbin(x, y, gridsize=30, cmap='Blues')
cb = plt.colorbar(label='count in bin')
```



For general feature values distribution plot, **Seaborn's KDE** visualization is more terse & easier implementation than Matplotlib.

```
ax = sns.displot(kind='kde', data=df, x='total_bill',
                 bw_method='scott', fill=False)
```



Kernel Density Estimation – an estimated *smooth density for the numerical feature distribution*.

> KDE has smoothing bandwidth that effectively adjusts the bias-variance trade-off on density values estimation.
> —by **'scott'** (default) / **'silverman'** / **'normal_reference'** smoothing algorithm, each with its own strengths & weaknesses in locating the optimal bandwidth for the data.

**" The units on the density axes is a common source of confusion.
The height of the curve at each point gives an estimated density, not exact probability;
a normalized proportion of frequency to make area under the distribution equal to 1.
A probability is obtained only by integrating the density across a range. "**

VI. Area / Waffle Chart

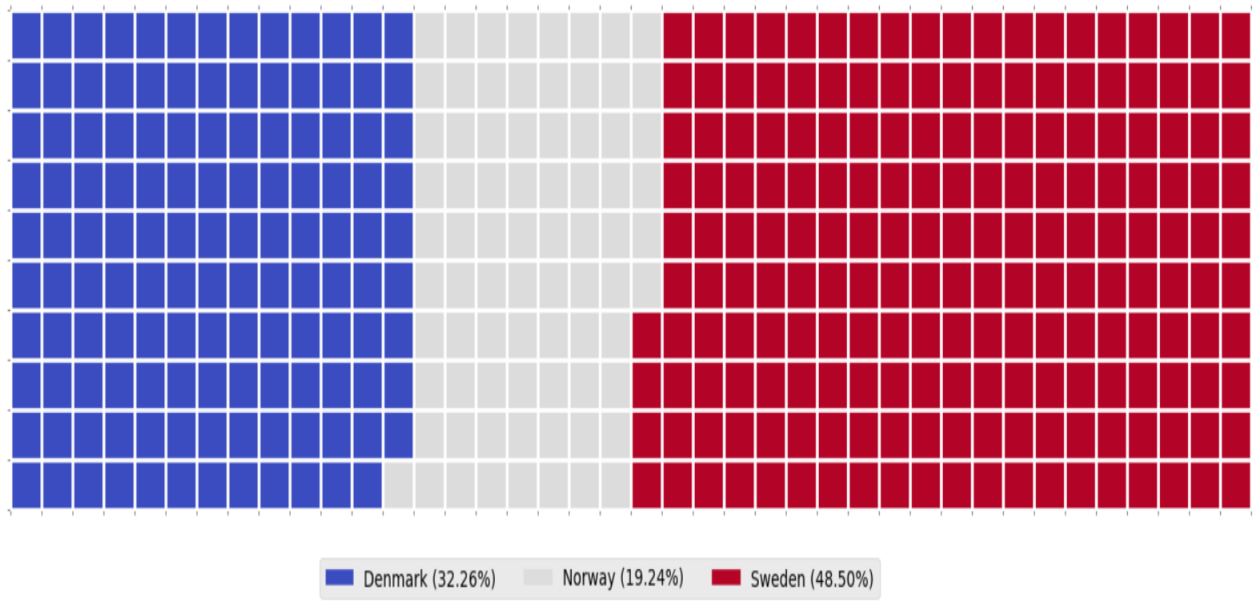
- manual configuration of color cells is required as it is not built into any of the Python visualization libs.

```
def create_waffle_chart(classes, values, *, height, width, colormap, value_sign=' ')
    # Compute proportion of each class w.r.t total frequency/discrete measure.
    total_value = sum(values)
    class_proportions = [(float(value) / total_value) for value in values]

    # Configure the desired overall waffle size & class fraction
    tot_tiles = width * height
    tiles_per_class = [round(proportion * tot_tiles) for proportion in class_proportions]

    # Create & populate the value matrix as representation of waffle chart
    waffle_mat = np.zeros((height, width), dtype=np.int)

    category_index, tile_index = 0, 0
    for col in range(width):
      for row in range(height):
            tile_index +=1

            if tile_index > sum(tiles_per_class[0:category_index]):
              category_index +=1

            waffle_chart[row, col] = category_index

    print('Waffle Matrix is populated')

    # Map the waffle matrix into visual
    fig = plt.figure( )

    colormap = plt.cm.get_map(colormap, len(classes))
    plt.matshow(waffle_mat, cmap=colormap)
    plt.show( )

    ax = plt.gca( )

    # set minor ticks
    ax.set_xticks(np.arange(-.5, width, 1), minor=True)
    ax.set_yticks(np.arange(-.5, height, 1), minor=True)

    # add gridlines based on minor ticks
    ax.grid(which='minor', color='w', ls='-', lw=2)

    plt.xticks([ ]); plt.yticks([ ])
```

```
    # create legend
    legend_handles = [ ]
    for i, class in enumerate(classes):
      if value_sign == '%':
        label_str = category + '(' + f '{class_proportions[i]*100:.2f}' + value_sign + ')'
      else:
            label_str = category + '(' + value_sign + str(values[i]) + ')'
      color_val = colormap(float(values_cumsum[i])/total_values)
      legend_handles.append(mpl.patches.Patch(color=colormap(i), label=label_str))

    # add legend to chart
    plt.legend(
      handles=legend_handles,
      loc='lower center',
      ncol=len(classes),
      bbox_to_anchor=(0., -0.2, 0.95, 0.1)
    )
    plt.show( )

# Create visualization from frequency / discrete measured data :
create_waffle_chart(df.index.values, df['Total'], width=40, height=10,
                    colormap='coolwarm', value_sign='%')
```

1. **categories**: Unique categories or classes in dataframe.
2. **values**: Values corresponding to categories or classes.
3. **height**: Defined height of waffle chart.
4. **width**: Defined width of waffle chart.
5. **colormap**: Colormap class
6. **value_sign**: In order to make our function more generalizable, we will add this parameter to address signs that could be associated with a value such as %, $, and so on. **value_sign** has a default value of empty string.

```
Total number of tiles is 400
Denmark: 129
Norway: 77
Sweden: 194
<Figure size 640x480 with 0 Axes>
```



SEABORN API

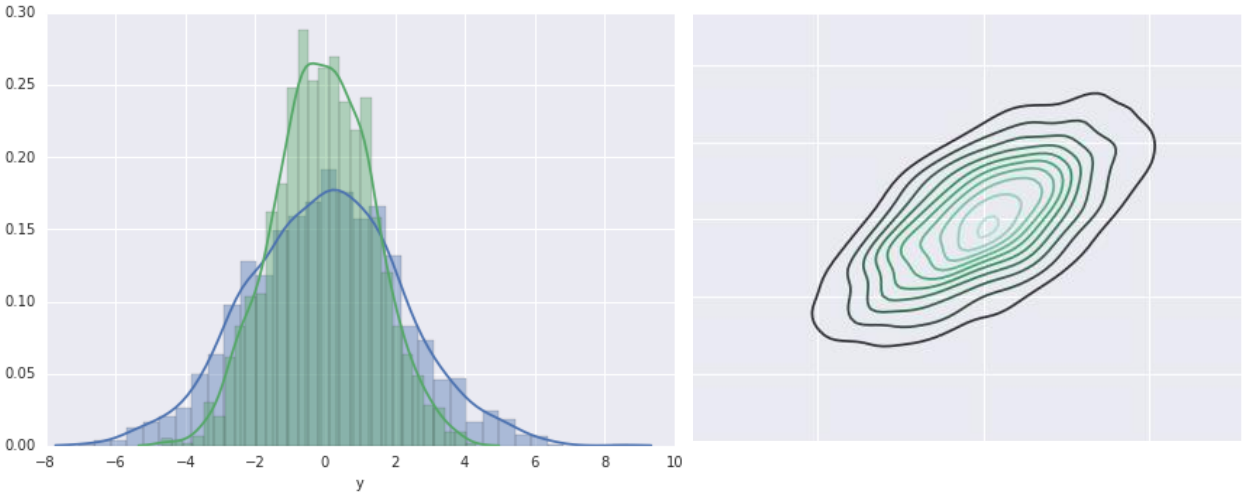
—In contrast to Matplotlib, Seaborn has simpler syntax for creating statistical graphics that require aggregations/ summarizations before plotting.

```
import seaborn as sns
# Useful aesthetics of plot for Black-and-White printed medium:
sns.set_theme(style='whitegrid', palette='Greys_r')
sns.set(font_scale=1.2)   # adjust the font in graphic
```

I. Distribution Plots – *'hist' / 'kde' / 'ecdf' / 'rugplot'*

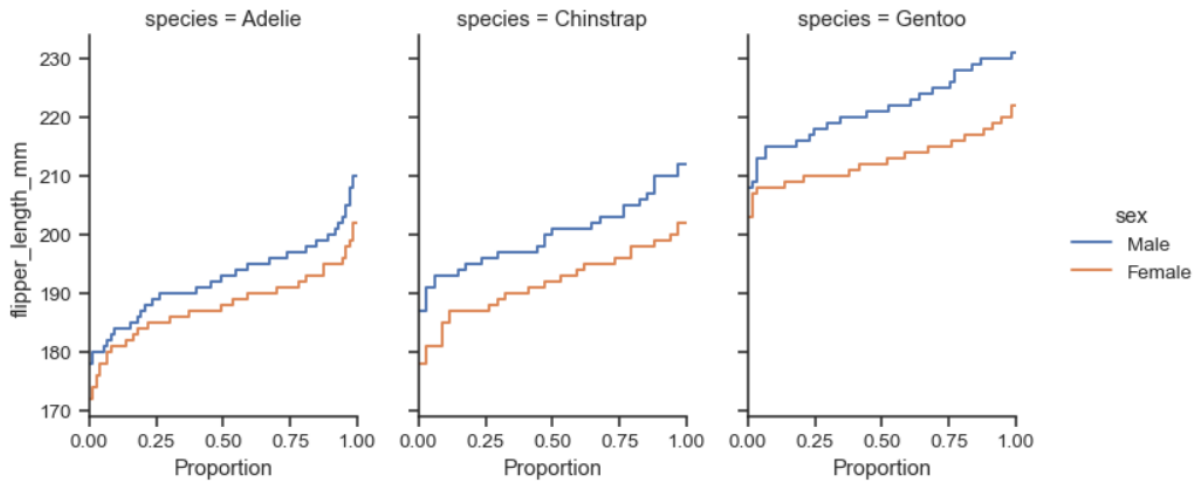
Seabon has Figure-level interface that generalizes to all related types of plot & returns a FacetGrid object, which allow conditional plots defined in parameters to row & col. (Otherwise, employ the typical plot kind for Matplotlib axes)

```
# Combined Histograms & KDEs
sns.displot(kind='hist', df['col'], binwidth=None, binrange=None, kde=True)
```



**If a 2-D data is passed to the function, a contour plot will be returned.

```
# Empirical Cumulative Distribution Function
sns.displot(data=penguins, y='flipper_length_mm', hue='sex', col='species',
            kind='ecdf', height=4, aspect=0.7)
```



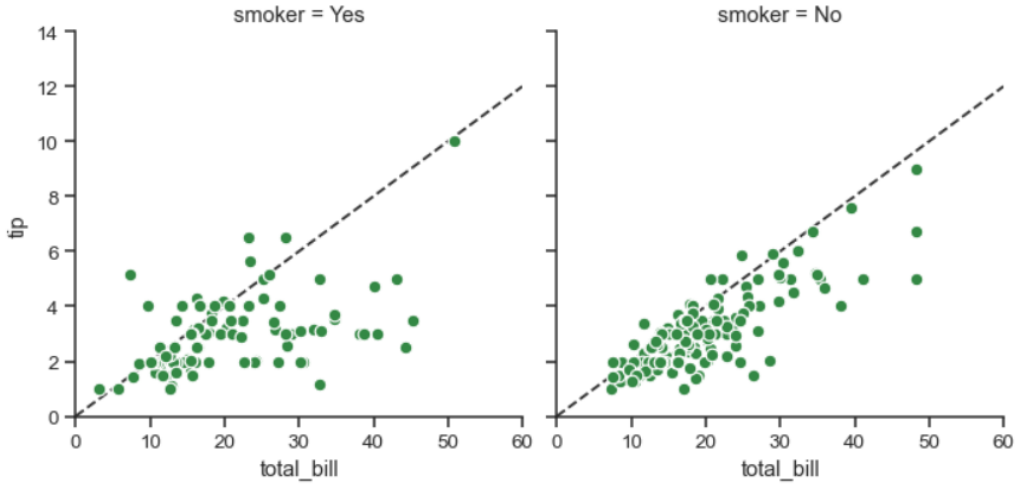
- For FacetGrid, the size & shape can be adjusted with 'height' and 'aspect' parameter.

II. Relational Plots – 'scatter' / 'line'

Unlike Matplotlib, Seaborn has relatively simple interface for specifying features in parameters w/o needing to summarize & create a separate plot for different groups.

—*ie*. in Matplotlib, ax = df.plot.scatter(x='a', y='b', color='blue', label='Group 1')
df.plot.scatter(x='c', y='d', color='green', label='Group 2', ax=ax2)

```
# Scatter plot
graph = sns.relplot(x='total_bill', y='tips', col='smoker', data=df, kind='scatter',
                    color='#338844', ec='white', height=4, s=50, lw=1)
for ax in graph.axes_dict.values():
    ax.axline((0, 0), slope=.2, c=".2", ls="--", zorder=0)
graph.set(xlim=(0, 60), ylim=(0, 14))
```



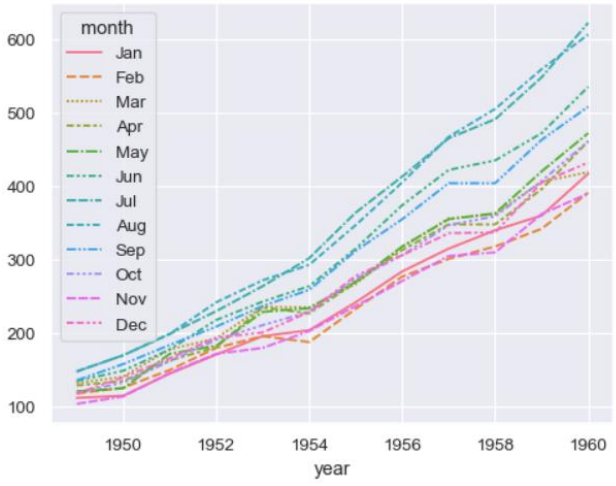
[# This is useful for exploring the interaction effect of a Cat_Var on the bivariate relationship]

See. `'lmplot'` if regression fitting is needed to ascertain the result.

Note: [1] Using the generalized interface (eg. 'relplot') is safer w/ relevant parameter specifications than FacetGrid class, unless more flexible customization on the grids is needed.

[2] `'zorder'` determines the drawing order of an artist w.r.t. others (line at above or below pts).

```
# Line Plot
sns.lineplot(data=flights_wide)
```

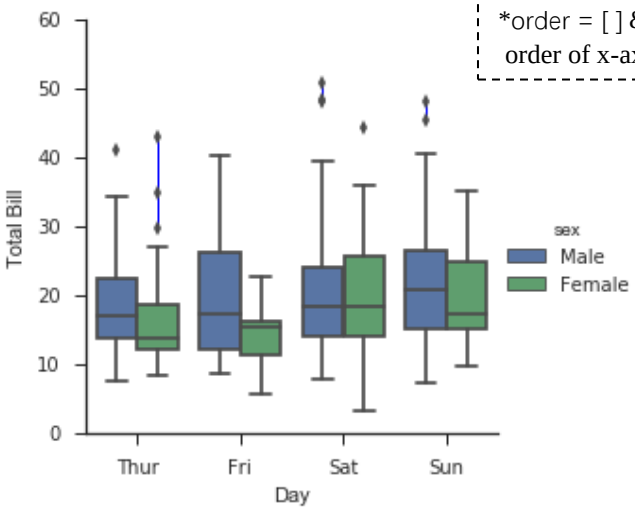


month	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
year												
1949	112	118	132	129	121	135	148	148	136	119	104	118
1950	115	126	141	135	125	149	170	170	158	133	114	140
1951	145	150	178	163	172	178	199	199	184	162	146	166
1952	171	180	193	181	183	218	230	242	209	191	172	194
1953	196	196	236	235	229	243	264	272	237	211	180	201

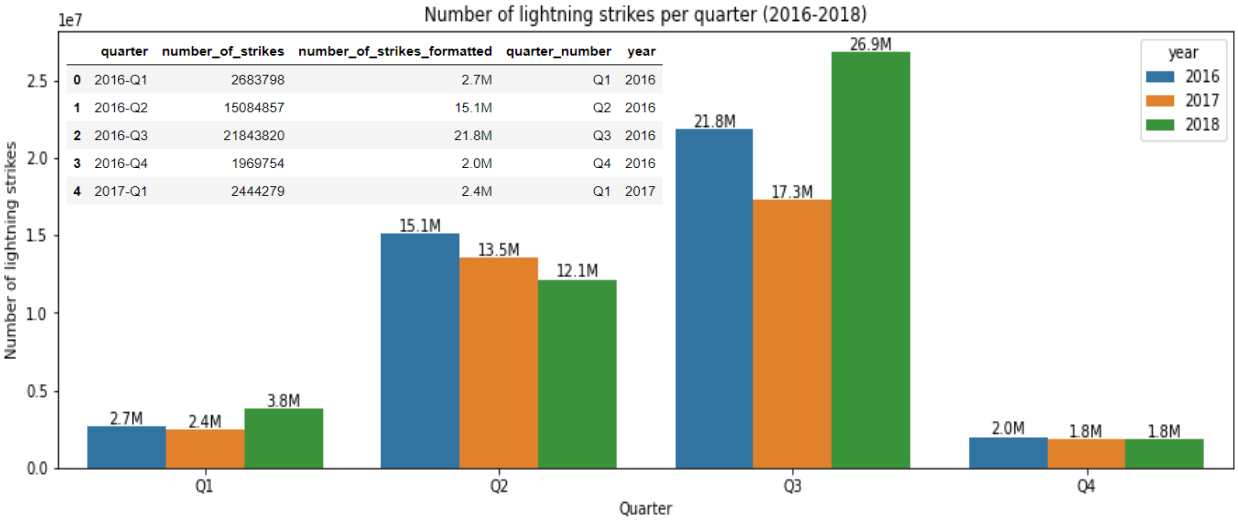
III. Categorical Plots – 'strip' / 'swamp' / 'box' / 'violin' / 'boxen' / 'point' / 'bar' / 'count'

```
# Box Plot
with sns.axes_style(style='ticks'):
    graph = sns.catplot("day", "total_bill", "sex", data=tips, kind="box")
    graph.set_axis_labels("Day", "Total Bill")
```

*order = [ ] & showfliers = **True**
order of x-axis / excl. outliers



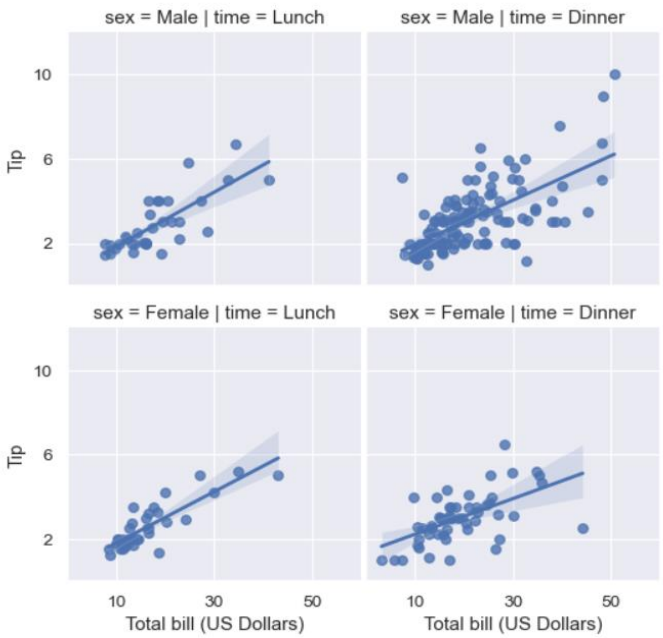

```
# Bar Plot: shows the frequency of observations in each class
plt.figure(figsize=(15,5))
# pal = sns.color_palette('Greens_d', grouped_df.nunique( ))  - add discrete colors as per ranks
# rank = grouped_df['feature'].argsort( ).argsort( )  - retrieve order of indexes
p = sns.barplot(data=df_by_quarter, x='quarter_number',
                y='no_of_strikes', hue='year')   # incl. palette=np.array(pal)[rank]
for b in p.patches:
    p.annotate(str(round(b.get_height()/1000000, 1)) + 'M',
               (b.get_x() + b.get_width() / 2., b.get_height() + 1.2e6),
               ha='center', va='bottom', xytext=(0, -12), textcoords='offset points')
plt.xlabel("Quarter"); plt.ylabel("Number of lightning strikes")
plt.title("Number of lightning strikes per quarter (2016-2018)")
```



	quarter	number_of_strikes	number_of_strikes_formatted	quarter_number	year
0	2016-Q1	2683798	2.7M	Q1	2016
1	2016-Q2	15084857	15.1M	Q2	2016
2	2016-Q3	21843820	21.8M	Q3	2016
3	2016-Q4	1969754	2.0M	Q4	2016
4	2017-Q1	2444279	2.4M	Q1	2017

IV. Regression Plots – 'regplot' / 'residplot'

```
# Combined regplot & FacetGrid:
graph = sns.lmplot(x="total_bill", y="tip", row="sex", col="time", data=tips)
(graph.set_axis_labels("Total bill (US Dollars)", "Tip")
 .set(xlim=(0, 60), ylim=(0, 12), xticks=[10, 30, 50], yticks=[2, 6, 10])
 .fig.subplots_adjust(wspace=0.02))
```



@ For visualizing the residual of a regression model,

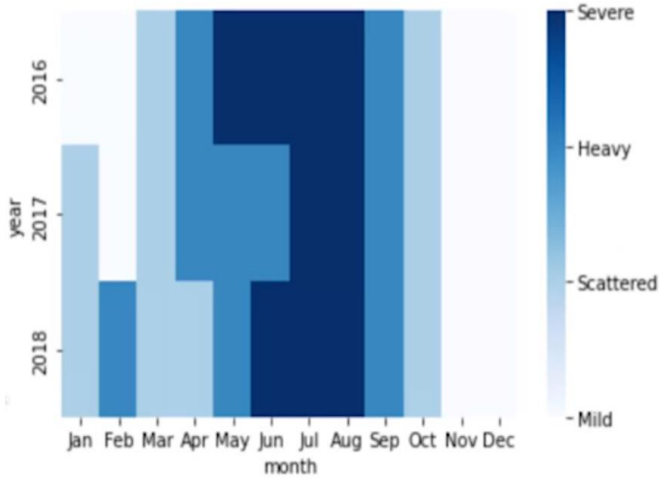
- 'sns.residplot' can be used to regress y on x, then draw a scatterplot of the residual.
- 'sns.jointplot' with kind="resid" to draw residual plot & resp. marginal distributions.

* Unlike others, regression plot has no Figure-level interface that integrates all kind of plots as one.

V. Matrix Plots – 'heatplot' / 'clusterplot'

```
df_by_month_plot = df.pivot_table('strike_level_code', index='year', column='month')
ax = sns.heatmap(df_by_month_plot, cmap='Blues')
colorbar = ax.collections[0].colorbar
colorbar.set_ticks([0, 1, 2, 3])
colorbar.set_tickslabels(['Mild', 'Scattered', 'Heavy', 'Severe'])
```

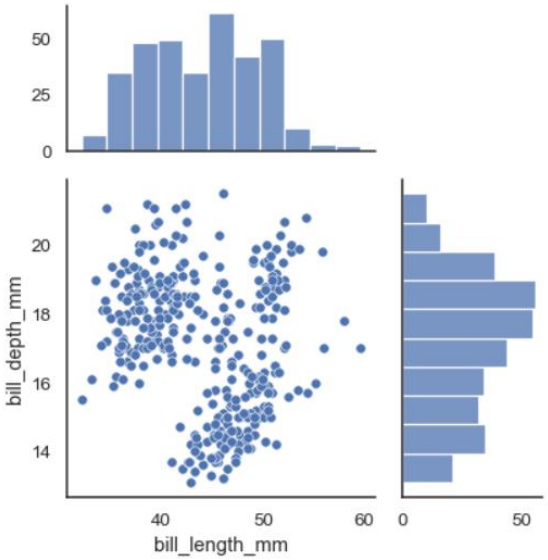
	year	month	number_of_strikes	strike_level	strike_level_code
0	2016	Jan	313595	Mild	0
1	2016	Feb	312676	Mild	0
2	2016	Mar	2057527	Scattered	1
3	2016	Apr	2636427	Heavy	2
4	2016	May	5800500	Severe	3



VI. Joint Grids

Besides the conventional multi-grid subplots, Seaborn offers a JointGrid class that combines bivariate & univariate numerical visualizations in a single plot.

```
with sns.axes_style('white'):
    graph = sns.jointplot(x='predictor', y='target', data, marginal_ticks=True,
                          kind='kde'/'hex'/'scatter'/'reg'/'resid',
                          maginal_kws=dict(bins=10, fill=True))
```

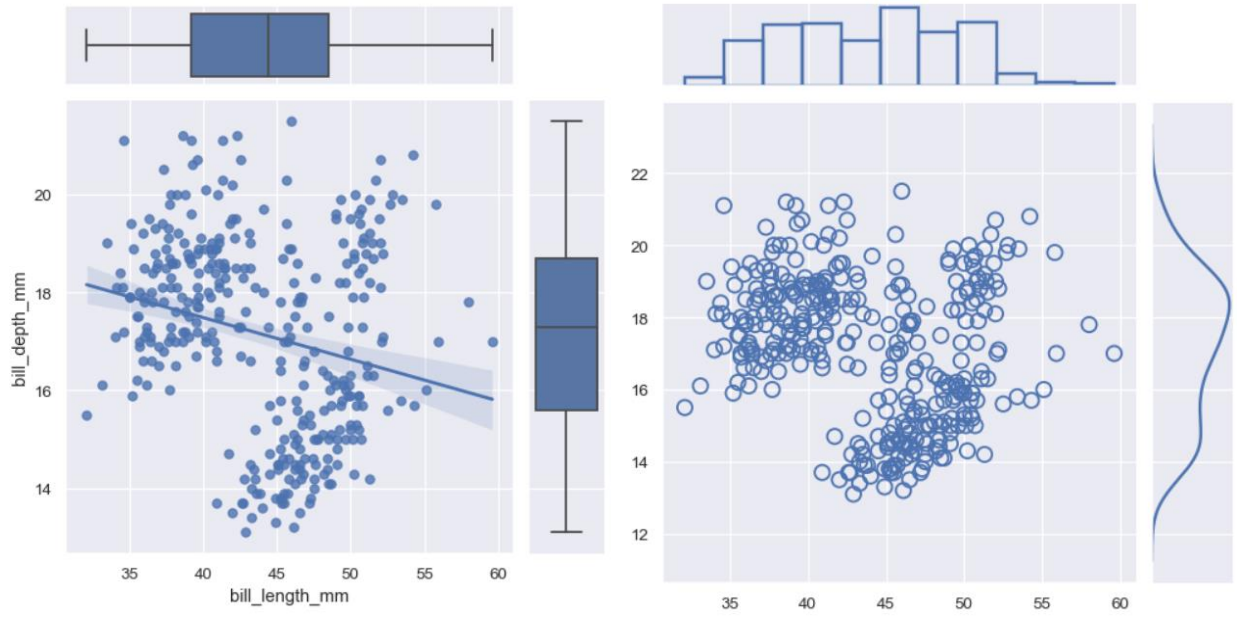


Alternately, independent subplots can be defined to the grids manually at—

```
graph = sns.JointGrid(data=df, x='bill_length_mm', y='bill_length_mm')
graph.plot(sns.regplot, sns.boxplot)
```

Or,

```
graph = sns.JointGrid()  # customizing each individually
sns.scatterplot(x=x, y=y, ec='b', fc='none', s=100, lw=1.5, ax=graph.ax_joint)
sns.histplot(x=x, fill=False, lw=2, ax=graph.ax_marg_x)
sns.kdeplot(y=y, lw=2, ax=graph_marg_y)
```



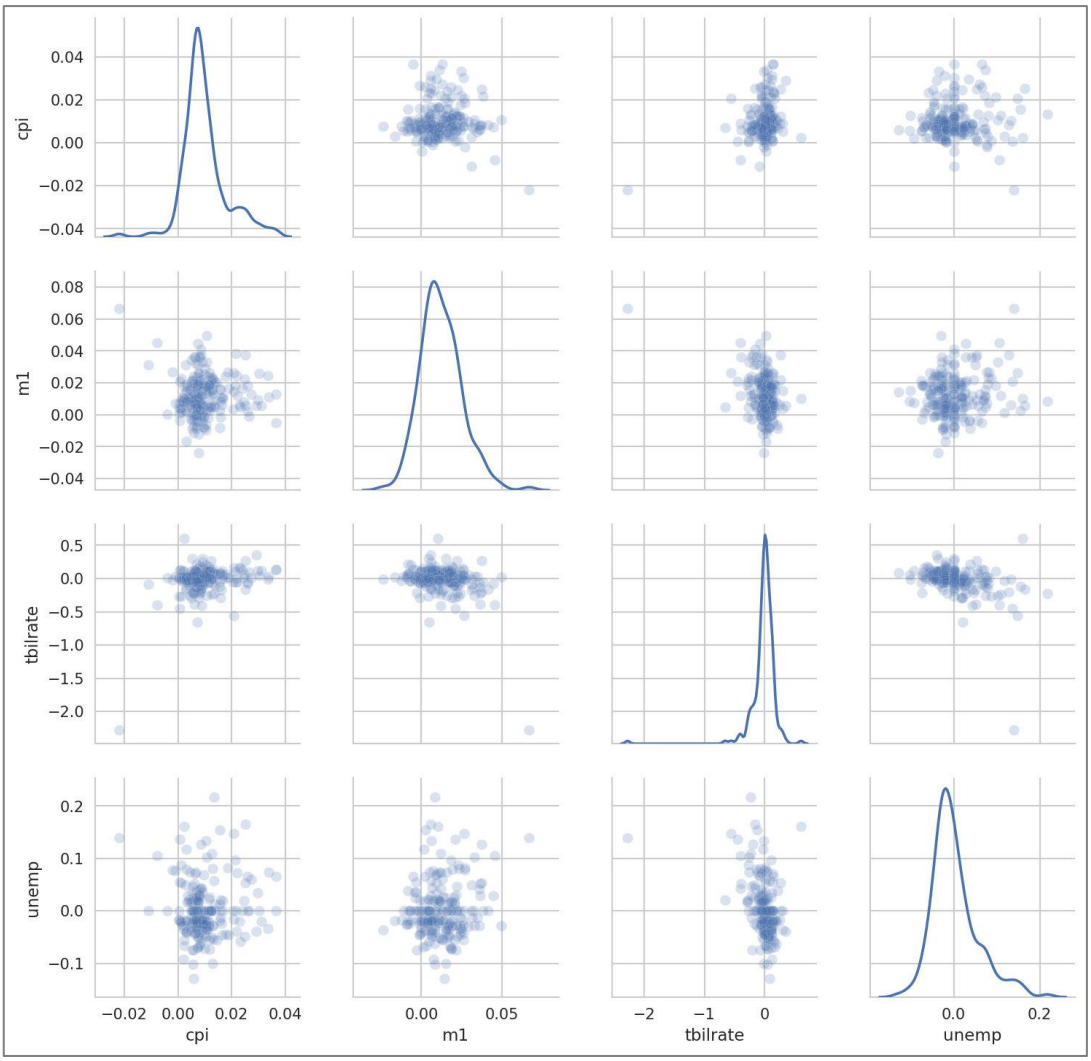
**Parameters— 'ax_joint', 'ax_marg_x', 'ax_marg_y' indicate the resp. distribution plots.

VII. Pair Grids

Often in EDA, a quick exploration among the numerical features is desired.

Seaborn's PairGrid class is handy, which illustrates an identical plot kind (*eg.* scatterplot matrix) via rows of small grids.

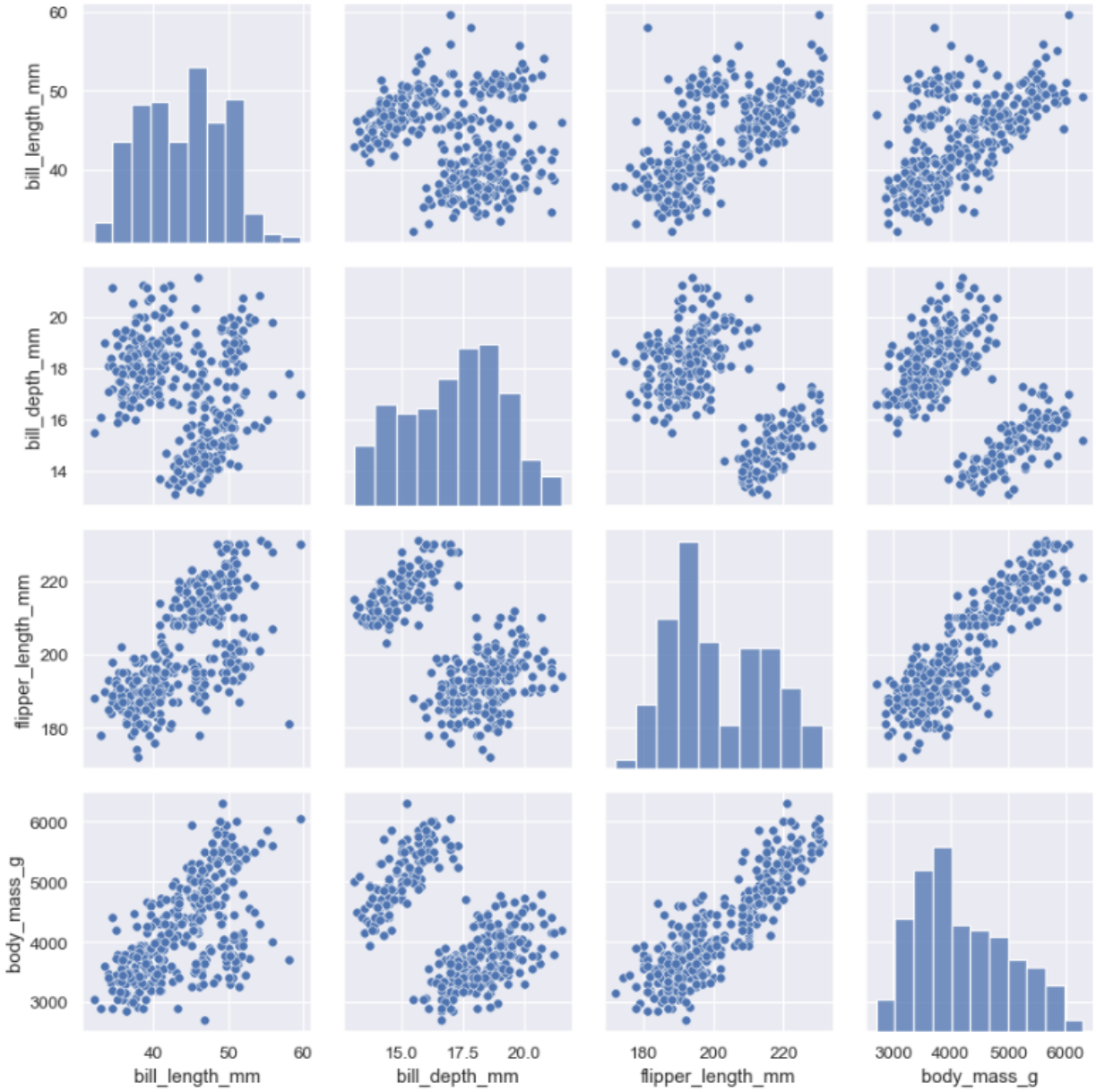
```
sns.pairplot(df, vars=['c1','c2',...], diag_kind='kde', plot_kws={'alpha': 0.2})
```



[# The plot illustrates relationships btw each var & marginal distribution along diagonal of the grids.]

To customize the pair of variables subplots & the diagonal subplots individually,

```
graph = sns.PairGrid(data=df, vars=['…'], hue=None, palette=None)
graph.map_diag(sns.histplot)
graph.map_offdiag(sns.scatterplot)
```



(Refer. Seaborn tutorial, @Building structured multi-plot grids for more details on multi-plots)

GEOSPATIAL VISUALIZATIONS

Visualize geographic data by $(\text{lat}, \text{long})$ values passed to related APIs $^{-\{Folium\ /\ Tableau\}}$ in order to generate maps of any location in the world.