TOOL

Plotting Tip Sheet: Matplotlib, Pandas, and Seaborn

Plotting in Python is supported by many different packages. This tip sheet focuses on the core functionality provided by Matplotlib, as well as the use of Matplotlib by both Pandas and Seaborn to provide convenient interfaces for plotting data from DataFrames and visually characterizing statistical properties of data. Links to documentation for each of these packages include:

- Matplotlib documentation
- Seaborn documentation
- Pandas plotting documentation

Matplotlib

- General information

```
import matplotlib.pyplot as plt       # import the pyplot interface and name it plt
```

  - Multiple function calls using `plt` can be made to build up a figure in stages (e.g., plot multiple data sets in the same figure) or customize the plot attributes (axis labels, tick marks, etc.).
  - At any given time, a particular figure is active, meaning that new `plt` commands are directed to that figure. If one is generating multiple figures at once, which figure is active can be switched with the `plt.figure` function as described below.
  - Some code examples are given below, but many more possibilities exist, so consult the available documentation for more information.

- `plt.figure:` Create a new figure.

```
plt.figure()                    # create a new figure
plt.figure(figsize=(10,10))     # create a new figure with specified size (width, height in inches)
plt.figure(2)                   # either create a new figure numbered 2, or make figure 2 active if it
                                  already exists
fig, ax = plt.subplots(2,2)     # create grid of subplots with specified number of rows and columns (e.g.,
                                  (2,2))
ax[0,0].plot(x, y)              # make plot in subplot by indexing into array of axes produced by plt.
                                  subplots
```

- `plt.plot:` Create a line plot.

```
plt.plot(y)                 # plot data in list or array y; x-axis defaults to range(len(y))
plt.plot(x, y)              # plot data in y against data in x; requires x and y to have the same length
plt.plot(x, y, 'bo')        # plot data in y against data in x, using format string to plot blue (b) circles (o)
plt.plot(x, y, 'rs-')       # plot data in y against data in x, using format string to plot red (r) squares (s) connected by lines (-)
plt.plot(a[:,0], a[:,1])    # plot data in column 1 of 2-dimensional array a against data in column 0
plt.errorbar(x, y, yerr)    # similar to plt.plot, but with error bars around data in y as specified in yerr
```

- `plt.scatter:` Create a scatter plot.

```
plt.scatter(x, y)           # scatter plot data y against the data in x
plt.scatter(x, y, s=5,      # scatter plot data y against the data in x, with markers of size 5 and
c='b')                        color blue ("b")
plt.scatter(x, y,           # scatter plot data y against the data in x, using marker style "^"
marker='^')                   (triangle pointing up)
```

- `plt.bar / plt.barh:` Create a bar chart.

```
plt.bar(x, height)          # vertical bar chart of data in height, with bars positioned according to
                              data in x
plt.bar(x, y, color='r')    # vertical bar chart of data in height, with bars positioned according to
                              data in x, coloring the bars red ("r")
plt.barh(y, width)          # horizontal bar chart of data in width, with bars positioned according to
                              data in y
```

- `plt.hist:` Create a histogram .

```
plt.hist(x)                    # plot a histogram (with vertical bars), putting data from x into default set of bins
plt.hist(x, bins=50)           # plot a histogram, putting data from x into specified number of bins (e.g., 50)
plt.hist(x,
bins=range(0,100))             # plot a histogram, putting data from x into bins with specified bin edges
```

- Customizing figures (providing additional commands to modify figures already generated with a plotting command):

```
plt.xlim(0, 100)               # set the limits of the x-axis to specified range (e.g., (0, 100))
plt.ylim(0.1, 0.9)             # set the limits of the y-axis to specified range (e.g., (0.1, 0.9))
plt.xlabel('year')             # set the label of the x-axis to specified text (e.g., "year")
plt.ylabel('cost')             # set the label of the y-axis to specified text (e.g., "cost")
plt.semilogx()                 # make the x-axis logarithmic
plt.semilogy()                 # make the y-axis logarithmic
plt.loglog()                   # make both axes logarithmic
plt.savefig('mydata.png')      # save the current active figure to specified file; file format is detected from file suffix (e.g., ".png")
plt.tight_layout()             # automatically adjust plot parameters, typically to reduce plot margins
```

Pandas

- General information

```
import pandas as pd                    # import pandas with shorthand pd
```

  - Pandas uses Matplotlib to make plots of data in DataFrames and Series by calling plot methods on those objects rather than using the underlying plt interface directly.
  - The plt interface can be used directly, however, to further customize plots generated by Pandas.
  - The code examples below assume that there exists a DataFrame named 'df.'

- `df.plot:` Plot data in a DataFrame.

```
df.plot(x='year', y='cost')                          # make a line plot of the DataFrame column "cost" against the column "year" (assuming they exist)
df.plot(x='year', y='cost', kind='bar')              # make a bar plot of "cost" against "year"
df.plot.bar((x='year', y='cost')                     # same as previous example: bar plot of "cost" against "year"
df.plot(x='year', y='cost', kind='scatter')          # make a scatter plot of "cost" against "year"
df.plot.scatter(x='year', y='cost')                  # same as previous example
df.plot.hist('cost')                                 # plot a histogram of the data in column "cost"
df.plot.box()                                        # make a box plot (box and whisker) of all the columns in the DataFrame
df.groupby('category').mean().plot(x='year', y='cost')   # make a plot for a DataFrame derived through other operations such as groupby
```

Seaborn

- General information

```
import seaborn as sns                    # import seaborn with shorthand sns
```

  - Seaborn uses Matplotlib to make plots of data, usually in DataFrames and Series, by calling plot methods on those objects rather than using the underlying plt interface directly.
  - The plt interface can be used directly, however, to further customize plots generated by Seaborn.
  - Since Seaborn is focused largely on characterizing statistical properties of data, its functionality is broadly divided into subareas for: (1) visualizing statistical relationships; (2) plotting with categorical data; (3) visualizing the distribution of a data set; and (4) visualizing linear relationships.
  - Seaborn contains some predefined data sets that can be loaded using the `sns.load_dataset` function; for example:
    - `tips = sns.load_dataset('tips')`
    - `mpg = sns.load_dataset('mpg')`
    - `fmri = sns.load_dataset('fmri')`
    - `titanic = sns.load_dataset('titanic')`
    - `iris = sns.load_dataset('iris')`
  - The code examples below assume that these sample DataFrames have been loaded and are drawn from the material in the Official Seaborn Tutorial.

- `sns.relplot:` Plot relationship between variables in a DataFrame.

```
sns.relplot(x='total_bill', y='tip', data=tips)                    # for tips, make a scatter plot between "total_bill" and "tip" (scatter plot: default kind)
sns.relplot(x='total_bill', y='tip', hue='smoker', data=tips)      # same as above, but color each point by categorical data in "smoker"
sns.relplot(x='total_bill', y='tip', kind='line', data=tips)       # for tips, make a line plot between "total_bill" and "tip" (kind="line")
sns.relplot(x='timepoint', y='signal', kind='line', ci=95, data=fmri)   # for fmri, make a line plot with 95% confidence interval about mean
```

- `sns.catplot:` Plot with categorical data.

```
sns.catplot(x='day', y='total_bill', data=tips)                          # for each category in x ("day"), make scatter plot of data in y ("total_bill"), with jitter
sns.catplot(x='day', y='total_bill', kind='swarm', data=tips)            # as above, but splay points as in a "beeswarm" to prevent them from overlapping
sns.catplot(x='day', y='total_bill', kind='box', data=tips)              # for each category in x, make a box plot for data in y
sns.catplot(x='sex', y='survived', hue='class', kind='bar', data=titanic)  # make a bar chart, coloring each bar by "class" category
sns.catplot(x='deck', kind='count', palette='ch:.25', data=titanic)      # make "count" plot (i.e., histogram) for categories in x (with a specified color palette)
sns.catplot(x='day', y='total_bill', col='smoker', data=tips)            # make multiple catplots in a FacetGrid, in columns for each category in col="smoker"
```

- **`sns.distplot:`** Plot a univariate distribution.

```
sns.distplot(tips['total_bill'])                      # plot a histogram, along with a line representing kernel density estimation (kde=True by default)
sns.distplot(tips['total_bill'], kde=False, rug=True) # plot a histogram, along with a "rug" plot showing individual values, but no kde
sns.distplot(tips['total_bill'], bins=20, rug=True)   # change the number of bins
sns.distplot(tips['total_bill'], hist=False, rug=True) # no histogram, but kde and rug plot
```

- **`sns.jointplot:`** Plot bivariate distributions.

```
sns.jointplot(x='total_bill', y='tip', data=tips)              # make a scatter plot of x and y, with histograms for each variable along each axis
sns.jointplot(x='total_bill', y='tip', kind='hex', data=tips)  # make a "hexbin" plot of x and y, with histograms for each variable along each axis
sns.jointplot(x='total_bill', y='tip', kind='kde', data=tips)  # make a kde-based contour plot of x and y, with kde plots for each variable along each axis
```

- **`sns.pairplot:`** Plot pairwise relationships among variables.

```
sns.pairplot(iris)                    # make a pairwise scatter plot grid of columns in "iris" data set
sns.pairplot(iris, hue='species')     # color each category of "species" by a distinct color
```

- **`sns.regplot and sns.lmplot:`** Plot regression models among variables.

```
sns.regplot(x='total_bill', y='tip', data=tips)              # scatter plot of x and y, along with best-fit linear regression and 95% confidence interval
sns.lmplot(x='total_bill', y='tip', hue='smoker', data=tips) # multiple scatter plots and regression lines, for each category in hue="smoker"
```

SEABORN

CREATING HISTOGRAM: seaborn.histplot( )

seaborn.histplot (*data=None, *, x=None, y=None, hue=None, weights=None, stat='count', bins='auto', binwidth=None, binrange=None, discrete=None, cumulative=False, common_bins=True, common_norm=True, multiple='layer', element='bars', fill=True, shrink=1, kde=False, kde_kws=None, line_kws=None, thresh=0, pthresh=None, pmax=None, cbar=False, cbar_ax=None, cbar_kws=None, palette=None, hue_order=None, hue_norm=None, color=None, log_scale=None, legend=True, ax=None, **kwargs*)

Parameters:

data : *pandas.DataFrame , numpy.ndarray , mapping, or sequence*

Input data structure. Either a long-form collection of vectors that can be assigned to named variables or a wide-form dataset that will be internally reshaped.

x, y : *vectors or keys in `data`*

Variables that specify positions on the x and y axes.

hue : *vector or key in `data`*

Semantic variable that is mapped to determine the color of plot elements.

weights : *vector or key in `data`*

If provided, weight the contribution of the corresponding data points towards the count in each bin by these factors.

stat : *str*

Aggregate statistic to compute in each bin.

- `count` : show the number of observations in each bin
- `frequency` : show the number of observations divided by the bin width
- `probability` : or `proportion` : normalize such that bar heights sum to 1
- `percent` : normalize such that bar heights sum to 100
- `density` : normalize such that the total area of the histogram equals 1

bins : *str, number, vector, or a pair of such values*

Generic bin parameter that can be the name of a reference rule, the number of bins, or the breaks of the bins. Passed to `numpy.histogram_bin_edges()` .

binwidth : *number or pair of numbers*

Width of each bin, overrides `bins` but can be used with `binrange` .

binrange : *pair of numbers or a pair of pairs*

Lowest and highest value for bin edges; can be used either with `bins` or `binwidth` . Defaults to data extremes.

discrete : *bool*

If True, default to `binwidth=1` and draw the bars so that they are centered on their corresponding data points. This avoids "gaps" that may otherwise appear when using discrete (integer) data.

cumulative : *bool*

If True, plot the cumulative counts as bins increase.

common_bins : *bool*

If True, use the same bins when semantic variables produce multiple plots. If using a reference rule to determine the bins, it will be computed with the full dataset.

common_norm : *bool*

If True and using a normalized statistic, the normalization will apply over the full dataset. Otherwise, normalize each histogram independently.

multiple : *{"layer", "dodge", "stack", "fill"}*

Approach to resolving multiple elements when semantic mapping creates subsets. Only relevant with univariate data.

element : *{"bars", "step", "poly"}*

Visual representation of the histogram statistic. Only relevant with univariate data.

fill : *bool*

If True, fill in the space under the histogram. Only relevant with univariate data.

shrink : *number*

Scale the width of each bar relative to the binwidth by this factor. Only relevant with univariate data.

kde : *bool*

If True, compute a kernel density estimate to smooth the distribution and show on the plot as (one or more) line(s). Only relevant with univariate data.

kde_kws : *dict*

Parameters that control the KDE computation, as in `kdeplot()` .

line_kws : *dict*

Parameters that control the KDE visualization, passed to `matplotlib.axes.Axes.plot()` .

thresh : *number or None*

Cells with a statistic less than or equal to this value will be transparent. Only relevant with bivariate data.

pthresh : *number or None*

Like `thresh` , but a value in [0, 1] such that cells with aggregate counts (or other statistics, when used) up to this proportion of the total will be transparent.

pmax : *number or None*

A value in [0, 1] that sets that saturation point for the colormap at a value such that cells below is constistute this proportion of the total count (or other statistic, when used).

cbar : *bool*

If True, add a colorbar to annotate the color mapping in a bivariate plot. Note: Does not currently support plots with a `hue` variable well.

cbar_ax : *`matplotlib.axes.Axes`*

Pre-existing axes for the colorbar.

cbar_kws : *dict*

Additional parameters passed to `matplotlib.figure.Figure.colorbar()` .

palette : *string, list, dict, or `matplotlib.colors.Colormap`*

Method for choosing the colors to use when mapping the `hue` semantic. String values are passed to `color_palette()` . List or dict values imply categorical mapping, while a colormap object implies numeric mapping.

hue_order : *vector of strings*

Specify the order of processing and plotting for categorical levels of the `hue` semantic.

hue_norm : *tuple or `matplotlib.colors.Normalize`*

Either a pair of values that set the normalization range in data units or an object that will map from data units into a [0, 1] interval. Usage implies numeric mapping.

color : *`matplotlib color`*

Single color specification for when hue mapping is not used. Otherwise, the plot will try to hook into the matplotlib property cycle.

log_scale : *bool or number, or pair of bools or numbers*

Set axis scale(s) to log. A single value sets the data axis for univariate distributions and both axes for bivariate distributions. A pair of values sets each axis independently. Numeric values are interpreted as the desired base (default 10). If `False` , defer to the existing Axes scale.

legend : *bool*

If False, suppress the legend for semantic variables.

ax : *`matplotlib.axes.Axes`*

Pre-existing axes for the plot. Otherwise, call `matplotlib.pyplot.gca()` internally.

kwargs

Other keyword arguments are passed to one of the following matplotlib functions:

- `matplotlib.axes.Axes.bar()` (univariate, element="bars")
- `matplotlib.axes.Axes.fill_between()` (univariate, other element, fill=True)
- `matplotlib.axes.Axes.plot()` (univariate, other element, fill=False)
- `matplotlib.axes.Axes.pcolormesh()` (bivariate)

Returns: `matplotlib.axes.Axes`

The matplotlib axes containing the plot.

RESCALING THE PLOT: matplotlib.pyplot.ylim( ): get or set the y limits of the current axis

`matplotlib.pyplot.ylim(*args, **kwargs)` [source]

Get or set the y-limits of the current axes.

Call signatures:

```
bottom, top = ylim()  # return the current ylim
ylim((bottom, top))   # set the ylim to bottom, top
ylim(bottom, top)     # set the ylim to bottom, top
```

If you do not specify args, you can alternatively pass *bottom* or *top* as kwargs, i.e.:

```
ylim(top=3)  # adjust the top leaving bottom unchanged
ylim(bottom=-1)  # adjust the bottom leaving top unchanged
```

Setting limits turns autoscaling off for the y-axis.

Returns: **bottom, top**

A tuple of the new y-axis limits.

seaborn.pairplot():

seaborn.pairplot (*data, *, hue=None, hue_order=None, palette=None, vars=None, x_vars=None, y_vars=None, kind='scatter', diag_kind='auto', markers=None, height=2.5, aspect=1, corner=False, dropna=False, plot_kws=None, diag_kws=None, grid_kws=None, size=None*)

Plot pairwise relationships in a dataset.

By default, this function will create a grid of Axes such that each numeric variable in `data` will by shared across the y-axes across a single row and the x-axes across a single column. The diagonal plots are treated differently: a univariate distribution plot is drawn to show the marginal distribution of the data in each column.

It is also possible to show a subset of variables or plot different variables on the rows and columns.

This is a high-level interface for `PairGrid` that is intended to make it easy to draw a few common styles. You should use `PairGrid` directly if you need more flexibility.

Parameters:

data : *pandas.DataFrame*

Tidy (long-form) dataframe where each column is a variable and each row is an observation.

hue : *name of variable in `data`*

Variable in `data` to map plot aspects to different colors.

hue_order : *list of strings*

Order for the levels of the hue variable in the palette

palette : *dict or seaborn color palette*

Set of colors for mapping the `hue` variable. If a dict, keys should be values in the `hue` variable.

vars : *list of variable names*

Variables within `data` to use, otherwise use every column with a numeric datatype.

{x, y}_vars : *lists of variable names*

Variables within `data` to use separately for the rows and columns of the figure; i.e. to make a non-square plot.

kind : *{'scatter', 'kde', 'hist', 'reg'}*

Kind of plot to make.

diag_kind : *{'auto', 'hist', 'kde', None}*

Kind of plot for the diagonal subplots. If 'auto', choose based on whether or not `hue` is used.

markers : *single matplotlib marker code or list*

Either the marker to use for all scatterplot points or a list of markers with a length the same as the number of levels in the hue variable so that differently colored points will also have different scatterplot markers.

height : *scalar*

Height (in inches) of each facet.

aspect : *scalar*

Aspect * height gives the width (in inches) of each facet.

aspect : *scalar*

Aspect * height gives the width (in inches) of each facet.

corner : *bool*

If True, don't add axes to the upper (off-diagonal) triangle of the grid, making this a "corner" plot.

dropna : *boolean*

Drop missing values from the data before plotting.

{plot, diag, grid}_kws : *dicts*

Dictionaries of keyword arguments. `plot_kws` are passed to the bivariate plotting function, `diag_kws` are passed to the univariate plotting function, and `grid_kws` are passed to the `PairGrid` constructor.

Returns:

grid : *PairGrid*

Returns the underlying `PairGrid` instance for further tweaking.

SCIPY

scipy.stats.mstats.winsorize(): Returns winsorized version of the input array.

```
scipy.stats.mstats.winsorize(a, limits=None, inclusive=(True, True), inplace=False, axis=None, nan_policy='propagate')
```

- `a`: input array
- `limits`: None OR tuple of floats [OPTIONAL] — Tuples of percentages to cut on each side of the array with respect to the number of unmasked data
- `inclusive`: [OPTIONAL] Tuple indicating whether the number of data being masked on each side should be truncated (True) or rounded (False).
- `inplace`: [OPTIONAL] Whether to winsorize in place (True) or to use a copy (False)
- `axis`: [OPTIONAL] None OR int — axis along to trim.
- `nan_policy`: [OPTIONAL]