# seaborn Documentation

***Release 0.12.0.dev0***

**Michael Waskom**

**Dec 27, 2020**

# CONTENTS

Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

For a brief introduction to the ideas behind the library, you can read the introductory notes. Visit the *installation page* to see how you can download the package and get started with it. You can browse the *example gallery* to see what you can do with seaborn, and then check out the *tutorial* and *API reference* to find out how.

To see the code or report a bug, please visit the GitHub repository. General support questions are most at home on stackoverflow or discourse, which have dedicated channels for seaborn.

CHAPTER

ONE

# WHAT'S NEW IN EACH VERSION

This page contains information about what has changed in each new version of `seaborn`. Each release is also marked with a DOI from Zenodo, which can be used to cite the library.

## 1.1 v0.12.0 (Unreleased)

- Made `scipy` an optional dependency and added `pip install seaborn[all]` as a method for ensuring the availability of compatible `scipy` and `statsmodels` libraries. This has a few minor implications for existing code, which are explained in the Github pull request (#2398).
- Following NEP29, dropped support for Python 3.6 and bumped the minimally-supported versions of the library dependencies.
- Removed several previously-deprecated utility functions (`iqr`, `percentiles`, `pmf_hist`, and `sort_df`).

## 1.2 v0.11.1 (December 2020)

DOI 10.5281/zenodo.4379347

This a bug fix release and is a recommended upgrade for all users on v0.11.0.

- [ENHANCEMENT] Reduced the use of matplotlib global state in the *multi-grid classes* (#2388).
- [FIX] Restored support for using tuples or numeric keys to reference fields in a long-form `data` object (#2386).
- [FIX] Fixed a bug in `lineplot()` where NAs were propagating into the confidence interval, sometimes erasing it from the plot (#2273).
- [FIX] Fixed a bug in `PairGrid`/`pairplot()` where diagonal axes would be empty when the grid was not square and the diagonal axes did not contain the marginal plots (#2270).
- [FIX] Fixed a bug in `PairGrid`/`pairplot()` where off-diagonal plots would not appear when column names in `data` had non-string type (#2368).
- [FIX] Fixed a bug where categorical dtype information was ignored when data consisted of boolean or boolean-like values (#2379).
- [FIX] Fixed a bug in `FacetGrid` where interior tick labels would be hidden when only the orthogonal axis was shared (#2347).
- [FIX] Fixed a bug in `FacetGrid` that caused an error when `legend_out=False` was set (#2304).
- [FIX] Fixed a bug in `kdeplot()` where `common_norm=True` was ignored if `hue` was not assigned (#2378).

- [FIX] Fixed a bug in *displot()* where the `row_order` and `col_order` parameters were not used (#2262).
- [FIX] Fixed a bug in *PairGrid*/*pairplot()* that caused an exception when using `corner=True` and `diag_kind=None` (#2382).
- [FIX] Fixed a bug in *clustermap()* where `annot=False` was ignored (#2323).
- [FIX] Fixed a bug in *clustermap()* where row/col color annotations could not have a categorical dtype (#2389).
- [FIX] Fixed a bug in *boxenplot()* where the `linewidth` parameter was ignored (#2287).
- [FIX] Raise a more informative error in *PairGrid*/*pairplot()* when no variables can be found to define the rows/columns of the grid (#2382).
- [FIX] Raise a more informative error from *clustermap()* if row/col color objects have semantic index but data object does not (#2313).

## 1.3 v0.11.0 (September 2020)

DOI 10.5281/zenodo.4019146

This is a major release with several important new features, enhancements to existing functions, and changes to the library. Highlights include an overhaul and modernization of the distributions plotting functions, more flexible data specification, new colormaps, and better narrative documentation.

For an overview of the new features and a guide to updating, see this Medium post.

### 1.3.1 Required keyword arguments

[API]

Most plotting functions now require all of their parameters to be specified using keyword arguments. To ease adaptation, code without keyword arguments will trigger a `FutureWarning` in v0.11. In a future release (v0.12 or v0.13, depending on release cadence), this will become an error. Once keyword arguments are fully enforced, the signature of the plotting functions will be reorganized to accept `data` as the first and only positional argument (#2052#2081).

### 1.3.2 Modernization of distribution functions

The distribution module has been completely overhauled, modernizing the API and introducing several new functions and features within existing functions. Some new features are explained here; the tutorial documentation has also been rewritten and serves as a good introduction to the functions.

#### New plotting functions

[FEATURE] [ENHANCEMENT]

First, three new functions, *displot()*, *histplot()* and *ecdfplot()* have been added (#2157, #2125, #2141).

The figure-level *displot()* function is an interface to the various distribution plots (analogous to *relplot()* or *catplot()*). It can draw univariate or bivariate histograms, density curves, ECDFs, and rug plots on a *FacetGrid*.

The axes-level *histplot()* function draws univariate or bivariate histograms with a number of features, including:

- mapping multiple distributions with a `hue` semantic

- normalization to show density, probability, or frequency statistics
- flexible parameterization of bin size, including proper bins for discrete variables
- adding a KDE fit to show a smoothed distribution over all bin statistics
- experimental support for histograms over categorical and datetime variables.

The axes-level *ecdfplot()* function draws univariate empirical cumulative distribution functions, using a similar interface.

### Changes to existing functions

[API] [FEATURE] [ENHANCEMENT] [DEFAULTS]

Second, the existing functions *kdeplot()* and *rugplot()* have been completely overhauled (#2060#2104).

The overhauled functions now share a common API with the rest of seaborn, they can show conditional distributions by mapping a third variable with a `hue` semantic, and they have been improved in numerous other ways. The github pull request (#2104) has a longer explanation of the changes and the motivation behind them.

This is a necessarily API-breaking change. The parameter names for the positional variables are now `x` and `y`, and the old names have been deprecated. Efforts were made to handle and warn when using the deprecated API, but it is strongly suggested to check your plots carefully.

Additionally, the statsmodels-based computation of the KDE has been removed. Because there were some inconsistencies between the way different parameters (specifically, `bw`, `clip`, and `cut`) were implemented by each backend, this may cause plots to look different with non-default parameters. Support for using non-Gaussian kernels, which was available only in the statsmodels backend, has been removed.

Other new features include:

- several options for representing multiple densities (using the `multiple` and `common_norm` parameters)
- weighted density estimation (using the new `weights` parameter)
- better control over the smoothing bandwidth (using the new `bw_adjust` parameter)
- more meaningful parameterization of the contours that represent a bivariate density (using the `thresh` and `levels` parameters)
- log-space density estimation (using the new `log_scale` parameter, or by scaling the data axis before plotting)
- “bivariate” rug plots with a single function call (by assigning both `x` and `y`)

### Deprecations

[API]

Finally, the *distplot()* function is now formally deprecated. Its features have been subsumed by *displot()* and *histplot()*. Some effort was made to gradually transition *distplot()* by adding the features in *displot()* and handling backwards compatibility, but this proved to be too difficult. The similarity in the names will likely cause some confusion during the transition, which is regrettable.

### Related enhancements and changes

[API] [FEATURE] [ENHANCEMENT] [DEFAULTS]

These additions facilitated new features (and forced changes) in *jointplot()* and *JointGrid* (#2210) and in *pairplot()* and *PairGrid* (#2234).

- Added support for the `hue` semantic in *jointplot()*/*JointGrid*. This support is lightweight and simply delegates the mapping to the underlying axes-level functions.
- Delegated the handling of `hue` in *PairGrid*/*pairplot()* to the plotting function when it understands `hue`, meaning that (1) the zorder of scatterplot points will be determined by row in dataframe, (2) additional options for resolving hue (e.g. the `multiple` parameter) can be used, and (3) numeric hue variables can be naturally mapped when using *scatterplot()*.
- Added `kind="hist"` to *jointplot()*, which draws a bivariate histogram on the joint axes and univariate histograms on the marginal axes, as well as both `kind="hist"` and `kind="kde"` to *pairplot()*, which behaves likewise.
- The various modes of *jointplot()* that plot marginal histograms now use *histplot()* rather than *distplot()*. This slightly changes the default appearance and affects the valid keyword arguments that can be passed to customize the plot. Likewise, the marginal histogram plots in *pairplot()* now use *histplot()*.

## 1.3.3 Standardization and enhancements of data ingest

[FEATURE] [ENHANCEMENT] [DOCS]

The code that processes input data has been refactored and enhanced. In v0.11, this new code takes effect for the relational and distribution modules; other modules will be refactored to use it in future releases (#2071).

These changes should be transparent for most use-cases, although they allow a few new features:

- Named variables for long-form data can refer to the named index of a `pandas.DataFrame` or to levels in the case of a multi-index. Previously, it was necessary to call `pandas.DataFrame.reset_index()` before using index variables (e.g., after a groupby operation).
- *relplot()* now has the same flexibility as the axes-level functions to accept data in long- or wide-format and to accept data vectors (rather than named variables) in long-form mode.
- The data parameter can now be a Python `dict` or an object that implements that interface. This is a new feature for wide-form data. For long-form data, it was previously supported but not documented.
- A wide-form data object can have a mixture of types; the non-numeric types will be removed before plotting. Previously, this caused an error.
- There are better error messages for other instances of data mis-specification.

See the new user guide chapter on data formats for more information about what is supported.

### 1.3.4 Other changes

#### Documentation improvements

- [Docs] Added two new chapters to the user guide, one giving an overview of the types of functions in seaborn, and one discussing the different data formats that seaborn understands.
- [Docs] Expanded the color palette tutorial to give more background on color theory and better motivate the use of color in statistical graphics.
- [Docs] Added more information to the *installation guidelines* and streamlined the introduction page.
- [Docs] Improved cross-linking within the seaborn docs and between the seaborn and matplotlib docs.

#### Theming

- [API] The `set()` function has been renamed to `set_theme()` for more clarity about what it does. For the foreseeable future, `set()` will remain as an alias, but it is recommended to update your code.

#### Relational plots

- [Enhancement] [Defaults] Reduced some of the surprising behavior of relational plot legends when using a numeric hue or size mapping (#2229):
  - Added an "auto" mode (the new default) that chooses between "brief" and "full" legends based on the number of unique levels of each variable.
  - Modified the ticking algorithm for a "brief" legend to show up to 6 values and not to show values outside the limits of the data.
  - Changed the approach to the legend title: the normal matplotlib legend title is used when only one variable is assigned a semantic mapping, whereas the old approach of adding an invisible legend artist with a subtitle label is used only when multiple semantic variables are defined.
  - Modified legend subtitles to be left-aligned and to be drawn in the default legend title font size.
- [Enhancement] [Defaults] Changed how functions that use different representations for numeric and categorical data handle vectors with an `object` data type. Previously, data was considered numeric if it could be coerced to a float representation without error. Now, object-typed vectors are considered numeric only when their contents are themselves numeric. As a consequence, numbers that are encoded as strings will now be treated as categorical data (#2084).
- [Enhancement] [Defaults] Plots with a `style` semantic can now generate an infinite number of unique dashes and/or markers by default. Previously, an error would be raised if the `style` variable had more levels than could be mapped using the default lists. The existing defaults were slightly modified as part of this change; if you need to exactly reproduce plots from earlier versions, refer to the old defaults (#2075).
- [Defaults] Changed how `scatterplot()` sets the default linewidth for the edges of the scatter points. New behavior is to scale with the point sizes themselves (on a plot-wise, not point-wise basis). This change also slightly reduces the default width when point sizes are not varied. Set `linewidth=0.75` to reproduce the previous behavior. (#2708).
- [Enhancement] Improved support for datetime variables in `scatterplot()` and `lineplot()` (#2138).
- [Fix] Fixed a bug where `lineplot()` did not pass the `linestyle` parameter down to matplotlib (#2095).
- [Fix] Adapted to a change in matplotlib that prevented passing vectors of literal values to `c` and `s` in `scatterplot()` (#2079).

### Categorical plots

- [ENHANCEMENT] [DEFAULTS] [FIX] Fixed a few computational issues in `boxenplot()` and improved its visual appearance (#2086):
  - Changed the default method for computing the number of boxes to``k_depth="tukey", as the previous default (``k_depth="proportion") is based on a heuristic that produces too many boxes for small datasets.
  - Added the option to specify the specific number of boxes (e.g. `k_depth=6`) or to plot boxes that will cover most of the data points (`k_depth="full"`).
  - Added a new parameter, `trust_alpha`, to control the number of boxes when `k_depth="trustworthy"`.
  - Changed the visual appearance of `boxenplot()` to more closely resemble `boxplot()`. Notably, thin boxes will remain visible when the edges are white.
- [ENHANCEMENT] Allowed `catplot()` to use different values on the categorical axis of each facet when axis sharing is turned off (e.g. by specifying `sharex=False`) (#2196).
- [ENHANCEMENT] Improved the error messages produced when categorical plots process the orientation parameter.
- [ENHANCEMENT] Added an explicit warning in `swarmplot()` when more than 5% of the points overlap in the "gutters" of the swarm (#2045).

### Multi-plot grids

- [FEATURE] [ENHANCEMENT] [DEFAULTS] A few small changes to make life easier when using `PairGrid` (#2234):
  - Added public access to the legend object through the `legend` attribute (also affects `FacetGrid`).
  - The `color` and `label` parameters are no longer passed to the plotting functions when `hue` is not used.
  - The data is no longer converted to a numpy object before plotting on the marginal axes.
  - It is possible to specify only one of `x_vars` or `y_vars`, using all variables for the unspecified dimension.
  - The `layout_pad` parameter is stored and used every time you call the `PairGrid.tight_layout()` method.
- [FEATURE] Added a `tight_layout` method to `FacetGrid` and `PairGrid`, which runs the `matplotlib.pyplot.tight_layout()` algorithm without interference from the external legend (#2073).
- [FEATURE] Added the `axes_dict` attribute to `FacetGrid` for named access to the component axes (#2046).
- [ENHANCEMENT] Made `FacetGrid.set_axis_labels()` clear labels from "interior" axes (#2046).
- [FEATURE] Added the `marginal_ticks` parameter to `JointGrid` which, if set to `True`, will show ticks on the count/density axis of the marginal plots (#2210).
- [ENHANCEMENT] Improved `FacetGrid.set_titles()` with `margin_titles=True`, such that texts representing the original row titles are removed before adding new ones (#2083).
- [DEFAULTS] Changed the default value for `dropna` to `False` in `FacetGrid`, `PairGrid`, `JointGrid`, and corresponding functions. As all or nearly all seaborn and matplotlib plotting functions handle missing data well, this option is no longer useful, but it causes problems in some edge cases. It may be deprecated in the future. (#2204).
- [FIX] Fixed a bug in `PairGrid` that appeared when setting `corner=True` and `despine=False` (#2203).

### 1.3.5 Color palettes

- [Docs] Improved and modernized the color palettes chapter of the seaborn tutorial.
- [Feature] Added two new perceptually-uniform colormaps: "flare" and "crest". The new colormaps are similar to "rocket" and "mako", but their luminance range is reduced. This makes them well suited to numeric mappings of line or scatter plots, which need contrast with the axes background at the extremes (#2237).
- [Enhancement] [Defaults] Enhanced numeric colormap functionality in several ways (#2237):
  - Added string-based access within the `color_palette()` interface to `dark_palette()`, `light_palette()`, and `blend_palette()`. This means that anywhere you specify a palette in seaborn, a name like `"dark:blue"` will use `dark_palette()` with the input `"blue"`.
  - Added the `as_cmap` parameter to `color_palette()` and changed internal code that uses a continuous colormap to take this route.
  - Tweaked the `light_palette()` and `dark_palette()` functions to use an endpoint that is a very desaturated version of the input color, rather than a pure gray. This produces smoother ramps. To exactly reproduce previous plots, use `blend_palette()` with `".13"` for dark or `".95"` for light.
  - Changed `diverging_palette()` to have a default value of `sep=1`, which gives better results.
- [Enhancement] Added a rich HTML representation to the object returned by `color_palette()` (#2225).
- [Fix] Fixed the `"{palette}_d"` logic to modify reversed colormaps and to use the correct direction of the luminance ramp in both cases.

#### Deprecations and removals

- [Enhancement] Removed an optional (and undocumented) dependency on BeautifulSoup (#2190) in `get_dataset_names()`.
- [API] Deprecated the `axlabel` function; use `ax.set(xlabel=, ylabel=)` instead.
- [API] Deprecated the `iqr` function; use `scipy.stats.iqr()` instead.
- [API] Final removal of the previously-deprecated `annotate` method on `JointGrid`, along with related parameters.
- [API] Final removal of the `lvplot` function (the previously-deprecated name for `boxenplot()`).

## 1.4 v0.10.1 (April 2020)

DOI 10.5281/zenodo.3767070

This is minor release with bug fixes for issues identified since 0.10.0.

- Fixed a bug that appeared within the bootstrapping algorithm on 32-bit systems.
- Fixed a bug where `regplot()` would crash on singleton inputs. Now a crash is avoided and regression estimation/plotting is skipped.
- Fixed a bug where `heatmap()` would ignore user-specified under/over/bad values when recentering a colormap.
- Fixed a bug where `heatmap()` would use values from masked cells when computing default colormap limits.
- Fixed a bug where `despine()` would cause an error when trying to trim spines on a matplotlib categorical axis.

- Adapted to a change in matplotlib that caused problems with single swarm plots.
- Added the `showfliers` parameter to `boxenplot()` to suppress plotting of outlier data points, matching the API of `boxplot()`.
- Avoided seeing an error from statmodels when data with an IQR of 0 is passed to `kdeplot()`.
- Added the `legend.title_fontsize` to the `plotting_context()` definition.
- Deprecated several utility functions that are no longer used internally (`percentiles`, `sig_stars`, `pmf_hist`, and `sort_df`).

## 1.5 v0.10.0 (January 2020)

DOI 10.5281/zenodo.3629446

This is a major update that is being released simultaneously with version 0.9.1. It has all of the same features (and bugs!) as 0.9.1, but there are important changes to the dependencies.

Most notably, all support for Python 2 has now been dropped. Support for Python 3.5 has also been dropped. Seaborn is now strictly compatible with Python 3.6+.

Minimally supported versions of the dependent PyData libraries have also been increased, in some cases substantially. While seaborn has tended to be very conservative about maintaining compatibility with older dependencies, this was causing increasing pain during development. At the same time, these libraries are now much easier to install. Going forward, seaborn will likely stay close to the Numpy community guidelines for version support.

This release also removes a few previously-deprecated features:

- The `tsplot` function and `seaborn.timeseries` module have been removed. Recall that `tsplot` was replaced with `lineplot()`.
- The `seaborn.apionly` entry-point has been removed.
- The `seaborn.linearmodels` module (previously renamed to `seaborn.regression`) has been removed.

### 1.5.1 Looking forward

Now that seaborn is a Python 3 library, it can take advantage of keyword-only arguments. It is likely that future versions will introduce this syntax, potentially in a breaking way. For guidance, most seaborn functions have a signature that looks like

```
func(x, y, ..., data=None, **kwargs)
```

where the `**kwargs` are specified in the function. Going forward it will likely be necessary to specify `data` and all subsequent arguments with an explicit `key=value` mapping. This style has long been used throughout the documentation, and the formal requirement will not be introduced until at least the next major release. Adding this feature will make it possible to enhance some older functions with more modern capabilities (e.g., adding a native `hue` semantic within functions like `jointplot()` and `regplot()`) and will allow parameters that control new features to be situated nearby related, making them more discoverable.

## 1.6 v0.9.1 (January 2020)

DOI 10.5281/zenodo.3629445

This is a minor release with a number of bug fixes and adaptations to changes in seaborn's dependencies. There are also several new features.

This is the final version of seaborn that will support Python 2.7 or 3.5.

### 1.6.1 New features

- Added more control over the arrangement of the elements drawn by `clustermap()` with the `{dendrogram,colors}_ratio` and `cbar_pos` parameters. Additionally, the default organization and scaling with different figure sizes has been improved.
- Added the `corner` option to `PairGrid` and `pairplot()` to make a grid without the upper triangle of bivariate axes.
- Added the ability to seed the random number generator for the bootstrap used to define error bars in several plots. Relevant functions now have a `seed` parameter, which can take either fixed seed (typically an `int`) or a numpy random number generator object (either the newer `numpy.random.Generator` or the older `numpy.random.mtrand.RandomState`).
- Generalized the idea of "diagonal" axes in `PairGrid` to any axes that share an x and y variable.
- In `PairGrid`, the `hue` variable is now excluded from the default list of variables that make up the rows and columns of the grid.
- Exposed the `layout_pad` parameter in `PairGrid` and set a smaller default than what matptlotlib sets for more efficient use of space in dense grids.
- It is now possible to force a categorical interpretation of the `hue` variable in a relational plot by passing the name of a categorical palette (e.g. `"deep"`, or `"Set2"`). This complements the (previously supported) option of passing a list/dict of colors.
- Added the `tree_kws` parameter to `clustermap()` to control the properties of the lines in the dendrogram.
- Added the ability to pass hierarchical label names to the `FacetGrid` legend, which also fixes a bug in `relplot()` when the same label appeared in different semantics.
- Improved support for grouping observations based on pandas index information in categorical plots.

### 1.6.2 Bug fixes and adaptations

- Avoided an error when singular data is passed to `kdeplot()`, issuing a warning instead. This makes `pairplot()` more robust.
- Fixed the behavior of `dropna` in `PairGrid` to properly exclude null datapoints from each plot when set to `True`.
- Fixed an issue where `regplot()` could interfere with other axes in a multi-plot matplotlib figure.
- Semantic variables with a `category` data type will always be treated as categorical in relational plots.
- Avoided a warning about color specifications that arose from `boxenplot()` on newer matplotlibs.
- Adapted to a change in how matplotlib scales axis margins, which caused multiple calls to `regplot()` with `truncate=False` to progressively expand the x axis limits. Because there are currently limitations on how autoscaling works in matplotlib, the default value for `truncate` in seaborn has also been changed to `True`.

- Relational plots no longer error when hue/size data are inferred to be numeric but stored with a string datatype.
- Relational plots now consider semantics with only a single value that can be interpreted as boolean (0 or 1) to be categorical, not numeric.
- Relational plots now handle list or dict specifications for `sizes` correctly.
- Fixed an issue in `pointplot()` where missing levels of a hue variable would cause an exception after a recent update in matplotlib.
- Fixed a bug when setting the rotation of x tick labels on a `FacetGrid`.
- Fixed a bug where values would be excluded from categorical plots when only one variable was a pandas `Series` with a non-default index.
- Fixed a bug when using `Series` objects as arguments for `x_partial` or `y_partial` in `regplot()`.
- Fixed a bug when passing a `norm` object and using color annotations in `clustermap()`.
- Fixed a bug where annotations were not rearranged to match the clustering in `clustermap()`.
- Fixed a bug when trying to call `set()` while specifying a list of colors for the palette.
- Fixed a bug when resetting the color code short-hands to the matplotlib default.
- Avoided errors from stricter type checking in upcoming `numpy` changes.
- Avoided error/warning in `lineplot()` when plotting categoricals with empty levels.
- Allowed `colors` to be passed through to a bivariate `kdeplot()`.
- Standardized the output format of custom color palette functions.
- Fixed a bug where legends for numerical variables in a relational plot could show a surprisingly large number of decimal places.
- Improved robustness to missing values in distribution plots.
- Made it possible to specify the location of the `FacetGrid` legend using matplotlib keyword arguments.

## 1.7 v0.9.0 (July 2018)

DOI 10.5281/zenodo.1313201

This is a major release with several substantial and long-desired new features. There are also updates/modifications to the themes and color palettes that give better consistency with matplotlib 2.0 and some notable API changes.

### 1.7.1 New relational plots

Three completely new plotting functions have been added: `relplot()`, `scatterplot()`, and `lineplot()`. The first is a figure-level interface to the latter two that combines them with a `FacetGrid`. The functions bring the high-level, dataset-oriented API of the seaborn categorical plotting functions to more general plots (scatter plots and line plots).

These functions can visualize a relationship between two numeric variables while mapping up to three additional variables by modifying `hue`, `size`, and/or `style` semantics. The common high-level API is implemented differently in the two functions. For example, the size semantic in `scatterplot()` scales the area of scatter plot points, but in `lineplot()` it scales width of the line plot lines. The API is dataset-oriented, meaning that in both cases you pass the variable in your dataset rather than directly specifying the matplotlib parameters to use for point area or line width.

Another way the relational functions differ from existing seaborn functionality is that they have better support for using numeric variables for `hue` and `size` semantics. This functionality may be propagated to other functions that can add a `hue` semantic in future versions; it has not been in this release.

The *`lineplot()`* function also has support for statistical estimation and is replacing the older `tsplot` function, which still exists but is marked for removal in a future release. *`lineplot()`* is better aligned with the API of the rest of the library and more flexible in showing relationships across additional variables by modifying the size and style semantics independently. It also has substantially improved support for date and time data, a major pain factor in `tsplot`. The cost is that some of the more esoteric options in `tsplot` for representing uncertainty (e.g. a colormapped KDE of the bootstrap distribution) have not been implemented in the new function.

There is quite a bit of new documentation that explains these new functions in more detail, including detailed examples of the various options in the *API reference* and a more verbose tutorial.

These functions should be considered in a "stable beta" state. They have been thoroughly tested, but some unknown corner cases may remain to be found. The main features are in place, but not all planned functionality has been implemented. There are planned improvements to some elements, particularly the default legend, that are a little rough around the edges in this release. Finally, some of the default behavior (e.g. the default range of point/line sizes) may change somewhat in future releases.

### 1.7.2 Updates to themes and palettes

Several changes have been made to the seaborn style themes, context scaling, and color palettes. In general the aim of these changes was to make the seaborn styles more consistent with the style updates in matplotlib 2.0 and to leverage some of the new style parameters for better implementation of some aspects of the seaborn styles. Here is a list of the changes:

- Reorganized and updated some *`axes_style()`*/*`plotting_context()`* parameters to take advantage of improvements in the matplotlib 2.0 update. The biggest change involves using several new parameters in the "style" spec while moving parameters that used to implement the corresponding aesthetics to the "context" spec. For example, axes spines and ticks are now off instead of having their width/length zeroed out for the darkgrid style. That means the width/length of these elements can now be scaled in different contexts. The effect is a more cohesive appearance of the plots, especially in larger contexts. These changes include only minimal support for the 1.x matplotlib series. Users who are stuck on matplotlib 1.5 but wish to use seaborn styling may want to use the seaborn parameters that can be accessed through the matplotlib stylesheet interface.
- Updated the seaborn palettes ("deep", "muted", "colorblind", etc.) to correspond with the new 10-color matplotlib default. The legacy palettes are now available at "deep6", "muted6", "colorblind6", etc. Additionally, a few individual colors were tweaked for better consistency, aesthetics, and accessibility.
- Calling *`color_palette()`* (or *`set_palette()`*) with a named qualitative palettes (i.e. one of the seaborn palettes, the colorbrewer qualitative palettes, or the matplotlib matplotlib tableau-derived palettes) and no specified number of colors will return all of the colors in the palette. This means that for some palettes, the returned list will have a different length than it did in previous versions.
- Enhanced *`color_palette()`* to accept a parameterized specification of a cubehelix palette in in a string, prefixed with `"ch:"` (e.g. `"ch:-.1,.2,l=.7"`). Note that keyword arguments can be spelled out or referenced using only their first letter. Reversing the palette is accomplished by appending `"_r"`, as with other matplotlib colormaps. This specification will be accepted by any seaborn function with a `palette=` parameter.
- Slightly increased the base font sizes in *`plotting_context()`* and increased the scaling factors for `"talk"` and `"poster"` contexts.
- Calling *`set()`* will now call *`set_color_codes()`* to re-assign the single letter color codes by default

### 1.7.3 API changes

A few functions have been renamed or have had changes to their default parameters.

- The `factorplot` function has been renamed to *catplot()*. The new name ditches the original R-inflected terminology to use a name that is more consistent with terminology in pandas and in seaborn itself. This change should hopefully make *catplot()* easier to discover, and it should make more clear what its role is. `factorplot` still exists and will pass its arguments through to *catplot()* with a warning. It may be removed eventually, but the transition will be as gradual as possible.
- The other reason that the `factorplot` name was changed was to ease another alteration which is that the default `kind` in *catplot()* is now `"strip"` (corresponding to *stripplot()*). This plots a categorical scatter plot which is usually a much better place to start and is more consistent with the default in *relplot()*. The old default style in `factorplot` (`"point"`, corresponding to *pointplot()*) remains available if you want to show a statistical estimation.
- The `lvplot` function has been renamed to *boxenplot()*. The "letter-value" terminology that was used to name the original kind of plot is obscure, and the abbreviation to `lv` did not help anything. The new name should make the plot more discoverable by describing its format (it plots multiple boxes, also known as "boxen"). As with `factorplot`, the `lvplot` function still exists to provide a relatively smooth transition.
- Renamed the `size` parameter to `height` in multi-plot grid objects (*FacetGrid*, *PairGrid*, and *JointGrid*) along with functions that use them (`factorplot`, *lmplot()*, *pairplot()*, and *jointplot()*) to avoid conflicts with the `size` parameter that is used in `scatterplot` and `lineplot` (necessary to make *relplot()* work) and also makes the meaning of the parameter a bit more clear.
- Changed the default diagonal plots in *pairplot()* to use func:`kdeplot` when a `"hue"` dimension is used.
- Deprecated the statistical annotation component of *JointGrid*. The method is still available but will be removed in a future version.
- Two older functions that were deprecated in earlier versions, `coefplot` and `interactplot`, have undergone final removal from the code base.

### 1.7.4 Documentation improvements

There has been some effort put into improving the documentation. The biggest change is that the introduction to the library has been completely rewritten to provide much more information and, critically, examples. In addition to the high-level motivation, the introduction also covers some important topics that are often sources of confusion, like the distinction between figure-level and axes-level functions, how datasets should be formatted for use in seaborn, and how to customize the appearance of the plots.

Other improvements have been made throughout, most notably a thorough re-write of the categorical tutorial.

### 1.7.5 Other small enhancements and bug fixes

- Changed *rugplot()* to plot a matplotlib `LineCollection` instead of many `Line2D` objects, providing a big speedup for large arrays.
- Changed the default off-diagonal plots to use *scatterplot()*. (Note that the `"hue"` currently draws three separate scatterplots instead of using the hue semantic of the scatterplot function).
- Changed color handling when using *kdeplot()* with two variables. The default colormap for the 2D density now follows the color cycle, and the function can use `color` and `label` kwargs, adding more flexibility and avoiding a warning when using with multi-plot grids.
- Added the `subplot_kws` parameter to *PairGrid* for more flexibility.

- Removed a special case in *PairGrid* that defaulted to drawing stacked histograms on the diagonal axes.
- Fixed *jointplot()*/*JointGrid* and *regplot()* so that they now accept list inputs.
- Fixed a bug in *FacetGrid* when using a single row/column level or using `col_wrap=1`.
- Fixed functions that set axis limits so that they preserve auto-scaling state on matplotlib 2.0.
- Avoided an error when using matplotlib backends that cannot render a canvas (e.g. PDF).
- Changed the install infrastructure to explicitly declare dependencies in a way that `pip` is aware of. This means that `pip install seaborn` will now work in an empty environment. Additionally, the dependencies are specified with strict minimal versions.
- Updated the testing infrastructure to execute tests with pytest (although many individual tests still use nose assertion).

## 1.8 v0.8.1 (September 2017)

DOI 10.5281/zenodo.883859

- Added a warning in *FacetGrid* when passing a categorical plot function without specifying `order` (or `hue_order` when `hue` is used), which is likely to produce a plot that is incorrect.
- Improved compatibility between *FacetGrid* or *PairGrid* and interactive matplotlib backends so that the legend no longer remains inside the figure when using `legend_out=True`.
- Changed categorical plot functions with small plot elements to use *dark_palette()* instead of *light_palette()* when generating a sequential palette from a specified color.
- Improved robustness of *kdeplot()* and *distplot()* to data with fewer than two observations.
- Fixed a bug in *clustermap()* when using `yticklabels=False`.
- Fixed a bug in *pointplot()* where colors were wrong if exactly three points were being drawn.
- Fixed a bug in *pointplot()* where legend entries for missing data appeared with empty markers.
- Fixed a bug in *clustermap()* where an error was raised when annotating the main heatmap and showing category colors.
- Fixed a bug in *clustermap()* where row labels were not being properly rotated when they overlapped.
- Fixed a bug in *kdeplot()* where the maximum limit on the density axes was not being updated when multiple densities were drawn.
- Improved compatibility with future versions of pandas.

## 1.9 v0.8.0 (July 2017)

DOI 10.5281/zenodo.824567

- The default style is no longer applied when seaborn is imported. It is now necessary to explicitly call *set()* or one or more of *set_style()*, *set_context()*, and *set_palette()*. Correspondingly, the `seaborn.apionly` module has been deprecated.
- Changed the behavior of *heatmap()* (and by extension *clustermap()*) when plotting divergent dataesets (i.e. when the `center` parameter is used). Instead of extending the lower and upper limits of the colormap to be symmetrical around the `center` value, the colormap is modified so that its middle color corresponds to

center. This means that the full range of the colormap will not be used (unless the data or specified vmin and vmax are symmetric), but the upper and lower limits of the colorbar will correspond to the range of the data. See the Github pull request (#1184) for examples of the behavior.

- Removed automatic detection of diverging data in *heatmap()* (and by extension *clustermap()*). If you want the colormap to be treated as diverging (see above), it is now necessary to specify the center value. When no colormap is specified, specifying center will still change the default to be one that is more appropriate for displaying diverging data.
- Added four new colormaps, created using viscm for perceptual uniformity. The new colormaps include two sequential colormaps ("rocket" and "mako") and two diverging colormaps ("icefire" and "vlag"). These colormaps are registered with matplotlib on seaborn import and the colormap objects can be accessed in the seaborn.cm namespace.
- Changed the default *heatmap()* colormaps to be "rocket" (in the case of sequential data) or "icefire" (in the case of diverging data). Note that this change reverses the direction of the luminance ramp from the previous defaults. While potentially confusing and disruptive, this change better aligns the seaborn defaults with the new matplotlib default colormap ("viridis") and arguably better aligns the semantics of a "heat" map with the appearance of the colormap.
- Added "auto" as a (default) option for tick labels in *heatmap()* and *clustermap()*. This will try to estimate how many ticks can be labeled without the text objects overlapping, which should improve performance for larger matrices.
- Added the dodge parameter to *boxplot()*, *violinplot()*, and *barplot()* to allow use of hue without changing the position or width of the plot elements, as when the hue varible is not nested within the main categorical variable.
- Correspondingly, the split parameter for *stripplot()* and *swarmplot()* has been renamed to dodge for consistency with the other categorical functions (and for differentiation from the meaning of split in *violinplot()*).
- Added the ability to draw a colorbar for a bivariate *kdeplot()* with the cbar parameter (and related cbar_ax and cbar_kws parameters).
- Added the ability to use error bars to show standard deviations rather than bootstrap confidence intervals in most statistical functions by putting ci="sd".
- Allow side-specific offsets in *despine()*.
- Figure size is no longer part of the seaborn plotting context parameters.
- Put a cap on the number of bins used in *jointplot()* for type=="hex" to avoid hanging when the reference rule prescribes too many.
- Changed the y axis in *heatmap()*. Instead of reversing the rows of the data internally, the y axis is now inverted. This may affect code that draws on top of the heatmap in data coordinates.
- Turn off dendrogram axes in *clustermap()* rather than setting the background color to white.
- New matplotlib qualitative palettes (e.g. "tab10") are now handled correctly.
- Some modules and functions have been internally reorganized; there should be no effect on code that uses the seaborn namespace.
- Added a deprecation warning to tsplot function to indicate that it will be removed or replaced with a substantially altered version in a future release.
- The interactplot and coefplot functions are officially deprecated and will be removed in a future release.

## 1.10 v0.7.1 (June 2016)

DOI 10.5281/zenodo.54844

- Added the ability to put "caps" on the error bars that are drawn by *barplot()* or *pointplot()* (and, by extension, `factorplot`). Additionally, the line width of the error bars can now be controlled. These changes involve the new parameters `capsize` and `errwidth`. See the github pull request (#898) for examples of usage.
- Improved the row and column colors display in *clustermap()*. It is now possible to pass Pandas objects for these elements and, when possible, the semantic information in the Pandas objects will be used to add labels to the plot. When Pandas objects are used, the color data is matched against the main heatmap based on the index, not on position. This is more accurate, but it may lead to different results if current code assumed positional matching.
- Improved the luminance calculation that determines the annotation color in *heatmap()*.
- The `annot` parameter of *heatmap()* now accepts a rectangular dataset in addition to a boolean value. If a dataset is passed, its values will be used for the annotations, while the main dataset will be used for the heatmap cell colors.
- Fixed a bug in *FacetGrid* that appeared when using `col_wrap` with missing `col` levels.
- Made it possible to pass a tick locator object to the *heatmap()* colorbar.
- Made it possible to use different styles (e.g., step) for *PairGrid* histograms when there are multiple hue levels.
- Fixed a bug in scipy-based univariate kernel density bandwidth calculation.
- The *reset_orig()* function (and, by extension, importing `seaborn.apionly`) resets matplotlib rcParams to their values at the time seaborn itself was imported, which should work better with rcParams changed by the jupyter notebook backend.
- Removed some objects from the top-level `seaborn` namespace.
- Improved unicode compatibility in *FacetGrid*.

## 1.11 v0.7.0 (January 2016)

DOI 10.5281/zenodo.45133

This is a major release from 0.6. The main new feature is *swarmplot()* which implements the beeswarm approach for drawing categorical scatterplots. There are also some performance improvements, bug fixes, and updates for compatibility with new versions of dependencies.

- Added the *swarmplot()* function, which draws beeswarm plots. These are categorical scatterplots, similar to those produced by *stripplot()*, but position of the points on the categorical axis is chosen to avoid overlapping points. See the categorical plot tutorial for more information.
- Changed some of the *stripplot()* defaults to be closer to *swarmplot()*. Points are now somewhat smaller, have no outlines, and are not split by default when using `hue`. These settings remain customizable through function parameters.
- Added an additional rule when determining category order in categorical plots. Now, when numeric variables are used in a categorical role, the default behavior is to sort the unique levels of the variable (i.e they will be in proper numerical order). This can still be overridden by the appropriate `{*_}order` parameter, and variables with a `category` datatype will still follow the category order even if the levels are strictly numerical.

- Changed how *stripplot()* draws points when using hue nesting with split=False so that the different hue levels are not drawn strictly on top of each other.
- Improve performance for large dendrograms in *clustermap()*.
- Added font.size to the plotting context definition so that the default output from plt.text will be scaled appropriately.
- Fixed a bug in *clustermap()* when fastcluster is not installed.
- Fixed a bug in the zscore calculation in *clustermap()*.
- Fixed a bug in *distplot()* where sometimes the default number of bins would not be an integer.
- Fixed a bug in *stripplot()* where a legend item would not appear for a hue level if there were no observations in the first group of points.
- Heatmap colorbars are now rasterized for better performance in vector plots.
- Added workarounds for some matplotlib boxplot issues, such as strange colors of outlier points.
- Added workarounds for an issue where violinplot edges would be missing or have random colors.
- Added a workaround for an issue where only one *heatmap()* cell would be annotated on some matplotlib backends.
- Fixed a bug on newer versions of matplotlib where a colormap would be erroneously applied to scatterplots with only three observations.
- Updated seaborn for compatibility with matplotlib 1.5.
- Added compatibility for various IPython (and Jupyter) versions in functions that use widgets.

## 1.12 v0.6.0 (June 2015)

DOI 10.5281/zenodo.19108

This is a major release from 0.5. The main objective of this release was to unify the API for categorical plots, which means that there are some relatively large API changes in some of the older functions. See below for details of those changes, which may break code written for older versions of seaborn. There are also some new functions (*stripplot()*, and *countplot()*), numerous enhancements to existing functions, and bug fixes.

Additionally, the documentation has been completely revamped and expanded for the 0.6 release. Now, the API docs page for each function has multiple examples with embedded plots showing how to use the various options. These pages should be considered the most comprehensive resource for examples, and the tutorial pages are now streamlined and oriented towards a higher-level overview of the various features.

### 1.12.1 Changes and updates to categorical plots

In version 0.6, the "categorical" plots have been unified with a common API. This new category of functions groups together plots that show the relationship between one numeric variable and one or two categorical variables. This includes plots that show distribution of the numeric variable in each bin (*boxplot()*, *violinplot()*, and *stripplot()*) and plots that apply a statistical estimation within each bin (*pointplot()*, *barplot()*, and *countplot()*). There is a new tutorial chapter that introduces these functions.

The categorical functions now each accept the same formats of input data and can be invoked in the same way. They can plot using long- or wide-form data, and can be drawn vertically or horizontally. When long-form data is used, the orientation of the plots is inferred from the types of the input data. Additionally, all functions natively take a hue variable to add a second layer of categorization.

With the (in some cases new) API, these functions can all be drawn correctly by *FacetGrid*. However, factorplot can also now create faceted verisons of any of these kinds of plots, so in most cases it will be unnecessary to use *FacetGrid* directly. By default, factorplot draws a point plot, but this is controlled by the kind parameter.

Here are details on what has changed in the process of unifying these APIs:

- Changes to *boxplot()* and *violinplot()* will probably be the most disruptive. Both functions maintain backwards-compatibility in terms of the kind of data they can accept, but the syntax has changed to be more similar to other seaborn functions. These functions are now invoked with x and/or y parameters that are either vectors of data or names of variables in a long-form DataFrame passed to the new data parameter. You can still pass wide-form DataFrames or arrays to data, but it is no longer the first positional argument. See the github pull request (#410) for more information on these changes and the logic behind them.
- As *pointplot()* and *barplot()* can now plot with the major categorical variable on the y axis, the x_order parameter has been renamed to order.
- Added a hue argument to *boxplot()* and *violinplot()*, which allows for nested grouping the plot elements by a third categorical variable. For *violinplot()*, this nesting can also be accomplished by splitting the violins when there are two levels of the hue variable (using split=True). To make this functionality feasible, the ability to specify where the plots will be draw in data coordinates has been removed. These plots now are drawn at set positions, like (and identical to) *barplot()* and *pointplot()*.
- Added a palette parameter to *boxplot()*/*violinplot()*. The color parameter still exists, but no longer does double-duty in accepting the name of a seaborn palette. palette supersedes color so that it can be used with a *FacetGrid*.

Along with these API changes, the following changes/enhancements were made to the plotting functions:

- The default rules for ordering the categories has changed. Instead of automatically sorting the category levels, the plots now show the levels in the order they appear in the input data (i.e., the order given by Series.unique()). Order can be specified when plotting with the order and hue_order parameters. Additionally, when variables are pandas objects with a "categorical" dtype, the category order is inferred from the data object. This change also affects *FacetGrid* and *PairGrid*.
- Added the scale and scale_hue parameters to *violinplot()*. These control how the width of the violins are scaled. The default is area, which is different from how the violins used to be drawn. Use scale='width' to get the old behavior.
- Used a different style for the box kind of interior plot in *violinplot()*, which shows the whisker range in addition to the quartiles. Use inner='quartile' to get the old style.

### 1.12.2 New plotting functions

- Added the *stripplot()* function, which draws a scatterplot where one of the variables is categorical. This plot has the same API as *boxplot()* and *violinplot()*. It is useful both on its own and when composed with one of these other plot kinds to show both the observations and underlying distribution.
- Added the *countplot()* function, which uses a bar plot representation to show counts of variables in one or more categorical bins. This replaces the old approach of calling *barplot()* without a numeric variable.

### 1.12.3 Other additions and changes

- The `corrplot()` and underlying `symmatplot()` functions have been deprecated in favor of *heatmap()*, which is much more flexible and robust. These two functions are still available in version 0.6, but they will be removed in a future version.
- Added the *set_color_codes()* function and the `color_codes` argument to *set()* and *set_palette()*. This changes the interpretation of shorthand color codes (i.e. "b", "g", k", etc.) within matplotlib to use the values from one of the named seaborn palettes (i.e. "deep", "muted", etc.). That makes it easier to have a more uniform look when using matplotlib functions directly with seaborn imported. This could be disruptive to existing plots, so it does not happen by default. It is possible this could change in the future.
- The *color_palette()* function no longer trims palettes that are longer than 6 colors when passed into it.
- Added the `as_hex` method to color palette objects, to return a list of hex codes rather than rgb tuples.
- *jointplot()* now passes additional keyword arguments to the function used to draw the plot on the joint axes.
- Changed the default `linewidths` in *heatmap()* and *clustermap()* to 0 so that larger matrices plot correctly. This parameter still exists and can be used to get the old effect of lines demarcating each cell in the heatmap (the old default `linewidths` was 0.5).
- *heatmap()* and *clustermap()* now automatically use a mask for missing values, which previously were shown with the "under" value of the colormap per default `plt.pcolormesh` behavior.
- Added the `seaborn.crayons` dictionary and the *crayon_palette()* function to define colors from the 120 box (!) of Crayola crayons.
- Added the `line_kws` parameter to *residplot()* to change the style of the lowess line, when used.
- Added open-ended `**kwargs` to the `add_legend` method on *FacetGrid* and *PairGrid*, which will pass additional keyword arguments through when calling the legend function on the `Figure` or `Axes`.
- Added the `gridspec_kws` parameter to *FacetGrid*, which allows for control over the size of individual facets in the grid to emphasize certain plots or account for differences in variable ranges.
- The interactive palette widgets now show a continuous colorbar, rather than a discrete palette, when `as_cmap` is True.
- The default Axes size for *pairplot()* and *PairGrid* is now slightly smaller.
- Added the `shade_lowest` parameter to *kdeplot()* which will set the alpha for the lowest contour level to 0, making it easier to plot multiple bivariate distributions on the same axes.
- The `height` parameter of *rugplot()* is now interpreted as a function of the axis size and is invariant to changes in the data scale on that axis. The rug lines are also slightly narrower by default.
- Added a catch in *distplot()* when calculating a default number of bins. For highly skewed data it will now use sqrt(n) bins, where previously the reference rule would return "infinite" bins and cause an exception in matplotlib.
- Added a ceiling (50) to the default number of bins used for *distplot()* histograms. This will help avoid confusing errors with certain kinds of datasets that heavily violate the assumptions of the reference rule used to get a default number of bins. The ceiling is not applied when passing a specific number of bins.
- The various property dictionaries that can be passed to `plt.boxplot` are now applied after the seaborn restyling to allow for full customizability.
- Added a `savefig` method to *JointGrid* that defaults to a tight bounding box to make it easier to save figures using this class, and set a tight bbox as the default for the `savefig` method on other Grid objects.

- You can now pass an integer to the `xticklabels` and `yticklabels` parameter of *heatmap()* (and, by extension, *clustermap()*). This will make the plot use the ticklabels inferred from the data, but only plot every `n` label, where `n` is the number you pass. This can help when visualizing larger matrices with some sensible ordering to the rows or columns of the dataframe.
- Added `"figure.facecolor"` to the style parameters and set the default to white.
- The *load_dataset()* function now caches datasets locally after downloading them, and uses the local copy on subsequent calls.

### 1.12.4 Bug fixes

- Fixed bugs in *clustermap()* where the mask and specified ticklabels were not being reorganized using the dendrograms.
- Fixed a bug in *FacetGrid* and *PairGrid* that lead to incorrect legend labels when levels of the `hue` variable appeared in `hue_order` but not in the data.
- Fixed a bug in `FacetGrid.set_xticklabels()` or `FacetGrid.set_yticklabels()` when `col_wrap` is being used.
- Fixed a bug in *PairGrid* where the `hue_order` parameter was ignored.
- Fixed two bugs in *despine()* that caused errors when trying to trim the spines on plots that had inverted axes or no ticks.
- Improved support for the `margin_titles` option in *FacetGrid*, which can now be used with a legend.

## 1.13 v0.5.1 (November 2014)

This is a bugfix release that includes a workaround for an issue in matplotlib 1.4.2 and fixes for two bugs in functions that were new in 0.5.0.

- Implemented a workaround for a bug in matplotlib 1.4.2 that prevented point markers from being drawn when the seaborn styles had been set. See this github issue for more information.
- Fixed a bug in *heatmap()* where the mask was vertically reversed relative to the data.
- Fixed a bug in *clustermap()* when using nested lists of side colors.

## 1.14 v0.5.0 (November 2014)

This is a major release from 0.4. Highlights include new functions for plotting heatmaps, possibly while applying clustering algorithms to discover structured relationships. These functions are complemented by new custom colormap functions and a full set of IPython widgets that allow interactive selection of colormap parameters. The palette tutorial has been rewritten to cover these new tools and more generally provide guidance on how to use color in visualizations. There are also a number of smaller changes and bugfixes.

### 1.14.1 Plotting functions

- Added the `heatmap()` function for visualizing a matrix of data by color-encoding the values. See the docs for more information.
- Added the `clustermap()` function for clustering and visualizing a matrix of data, with options to label individual rows and columns by colors. See the docs for more information. This work was lead by Olga Botvinnik.
- `lmplot()` and `pairplot()` get a new keyword argument, `markers`. This can be a single kind of marker or a list of different markers for each level of the `hue` variable. Using different markers for different hues should let plots be more comprehensible when reproduced to black-and-white (i.e. when printed). See the github pull request (#323) for examples.
- More generally, there is a new keyword argument in `FacetGrid` and `PairGrid`, `hue_kws`. This similarly lets plot aesthetics vary across the levels of the hue variable, but more flexibily. `hue_kws` should be a dictionary that maps the name of keyword arguments to lists of values that are as long as the number of levels of the hue variable.
- The argument `subplot_kws` has been added to `FacetGrid`. This allows for faceted plots with custom projections, including maps with Cartopy.

### 1.14.2 Color palettes

- Added two new functions to create custom color palettes. For sequential palettes, you can use the `light_palette()` function, which takes a seed color and creates a ramp from a very light, desaturated variant of it. For diverging palettes, you can use the `diverging_palette()` function to create a balanced ramp between two endpoints to a light or dark midpoint. See the palette tutorial for more information.
- Added the ability to specify the seed color for `light_palette()` and `dark_palette()` as a tuple of `husl` or `hls` space values or as a named `xkcd` color. The interpretation of the seed color is now provided by the new `input` parameter to these functions.
- Added several new interactive palette widgets: `choose_colorbrewer_palette()`, `choose_light_palette()`, `choose_dark_palette()`, and `choose_diverging_palette()`. For consistency, renamed the cubehelix widget to `choose_cubehelix_palette()` (and fixed a bug where the cubehelix palette was reversed). These functions also now return either a color palette list or a matplotlib colormap when called, and that object will be live-updated as you play with the widget. This should make it easy to iterate over a plot until you find a good representation for the data. See the Github pull request or this notebook (download it to use the widgets) for more information.
- Overhauled the color palette tutorial to organize the discussion by class of color palette and provide more motivation behind the various choices one might make when choosing colors for their data.

### 1.14.3 Bug fixes

- Fixed a bug in `PairGrid` that gave incorrect results (or a crash) when the input DataFrame has a non-default index.
- Fixed a bug in `PairGrid` where passing columns with a date-like datatype raised an exception.
- Fixed a bug where `lmplot()` would show a legend when the hue variable was also used on either the rows or columns (making the legend redundant).
- Worked around a matplotlib bug that was forcing outliers in `boxplot()` to appear as blue.
- `kdeplot()` now accepts pandas Series for the `data` and `data2` arguments.

- Using a non-default correlation method in `corrplot()` now implies `sig_stars=False` as the permutation test used to significance values for the correlations uses a pearson metric.
- Removed `pdf.fonttype` from the style definitions, as the value used in version 0.4 resulted in very large PDF files.

## 1.15 v0.4.0 (September 2014)

This is a major release from 0.3. Highlights include new approaches for *quick, high-level dataset exploration* (along with a more *flexible interface*) and easy creation of perceptually-appropriate color palettes using the cubehelix system. Along with these additions, there are a number of smaller changes that make visualizing data with seaborn easier and more powerful.

### 1.15.1 Plotting functions

- A new object, `PairGrid`, and a corresponding function `pairplot()`, for drawing grids of pairwise relationships in a dataset. This style of plot is sometimes called a "scatterplot matrix", but the representation of the data in `PairGrid` is flexible and many styles other than scatterplots can be used. See the docs for more information. **Note:** due to a bug in older versions of matplotlib, you will have best results if you use these functions with matplotlib 1.4 or later.
- The rules for choosing default color palettes when variables are mapped to different colors have been unified (and thus changed in some cases). Now when no specific palette is requested, the current global color palette will be used, unless the number of variables to be mapped exceeds the number of unique colors in the palette, in which case the `"husl"` palette will be used to avoid cycling.
- Added a keyword argument `hist_norm` to `distplot()`. When a `distplot()` is now drawn without a KDE or parametric density, the histogram is drawn as counts instead of a density. This can be overridden by by setting `hist_norm` to `True`.
- When using `FacetGrid` with a `hue` variable, the legend is no longer drawn by default when you call `FacetGrid.map()`. Instead, you have to call `FacetGrid.add_legend()` manually. This should make it easier to layer multiple plots onto the grid without having duplicated legends.
- Made some changes to `factorplot` so that it behaves better when not all levels of the `x` variable are represented in each facet.
- Added the `logx` option to `regplot()` for fitting the regression in log space.
- When `violinplot()` encounters a bin with only a single observation, it will now plot a horizontal line at that value instead of erroring out.

### 1.15.2 Style and color palettes

- Added the `cubehelix_palette()` function for generating sequential palettes from the cubehelix system. See the palette docs for more information on how these palettes can be used. There is also the `choose_cubehelix()` which will launch an interactive app to select cubehelix parameters in the notebook.
- Added the `xkcd_palette()` and the `xkcd_rgb` dictionary so that colors can be specified with names from the xkcd color survey.
- Added the `font_scale` option to `plotting_context()`, `set_context()`, and `set()`. `font_scale` can independently increase or decrease the size of the font elements in the plot.

- Font-handling should work better on systems without Arial installed. This is accomplished by adding the `font.sans-serif` field to the `axes_style` definition with Arial and Liberation Sans prepended to matplotlib defaults. The font family can also be set through the `font` keyword argument in *set()*. Due to matplotlib bugs, this might not work as expected on matplotlib 1.3.
- The *despine()* function gets a new keyword argument `offset`, which replaces the deprecated `offset_spines()` function. You no longer need to offset the spines before plotting data.
- Added a default value for `pdf.fonttype` so that text in PDFs is editable in Adobe Illustrator.

### 1.15.3 Other API Changes

- Removed the deprecated `set_color_palette` and `palette_context` functions. These were replaced in version 0.3 by the *set_palette()* function and ability to use *color_palette()* directly in a `with` statement.
- Removed the ability to specify a `nogrid` style, which was renamed to `white` in 0.3.

## 1.16 v0.3.1 (April 2014)

This is a minor release from 0.3 with fixes for several bugs.

### 1.16.1 Plotting functions

- The size of the points in *pointplot()* and `factorplot` are now scaled with the linewidth for better aesthetics across different plotting contexts.
- The *pointplot()* glyphs for different levels of the hue variable are drawn at different z-orders so that they appear uniform.

### 1.16.2 Bug Fixes

- Fixed a bug in *FacetGrid* (and thus affecting lmplot and factorplot) that appeared when `col_wrap` was used with a number of facets that did not evenly divide into the column width.
- Fixed an issue where the support for kernel density estimates was sometimes computed incorrectly.
- Fixed a problem where `hue` variable levels that were not strings were missing in *FacetGrid* legends.
- When passing a color palette list in a `with` statement, the entire palette is now used instead of the first six colors.

## 1.17 v0.3.0 (March 2014)

This is a major release from 0.2 with a number of enhancements to the plotting capabilities and styles. Highlights include *FacetGrid*, `factorplot`, *jointplot()*, and an overhaul to style management. There is also lots of new documentation, including an *example gallery* and reorganized *tutorial*.

### 1.17.1 New plotting functions

- The *FacetGrid* class adds a new form of functionality to seaborn, providing a way to abstractly structure a grid of plots corresponding to subsets of a dataset. It can be used with a wide variety of plotting functions (including most of the matplotlib and seaborn APIs. See the tutorial for more information.
- Version 0.3 introduces the `factorplot` function, which is similar in spirit to *lmplot()* but intended for use when the main independent variable is categorical instead of quantitative. `factorplot` can draw a plot in either a point or bar representation using the corresponding Axes-level functions *pointplot()* and *barplot()* (which are also new). Additionally, the `factorplot` function can be used to draw box plots on a faceted grid. For examples of how to use these functions, you can refer to the tutorial.
- Another new function is *jointplot()*, which is built using the new *JointGrid* object. *jointplot()* generalizes the behavior of *regplot()* in previous versions of seaborn (*regplot()* has changed somewhat in 0.3; see below for details) by drawing a bivariate plot of the relationship between two variables with their marginal distributions drawn on the side of the plot. With *jointplot()*, you can draw a scatterplot or regression plot as before, but you can now also draw bivariate kernel densities or hexbin plots with appropriate univariate graphs for the marginal distributions. Additionally, it's easy to use *JointGrid* directly to build up more complex plots when the default methods offered by *jointplot()* are not suitable for your visualization problem. The tutorial for *JointGrid* has more examples of how this object can be useful.
- The *residplot()* function complements *regplot()* and can be quickly used to diagnose problems with a linear model by calculating and plotting the residuals of a simple regression. There is also a `"resid"` kind for *jointplot()*.

### 1.17.2 API changes

- The most noticeable change will be that *regplot()* no longer produces a multi-component plot with distributions in marginal axes. Instead. *regplot()* is now an "Axes-level" function that can be plotted into any existing figure on a specific set of axes. *regplot()* and *lmplot()* have also been unified (the latter uses the former behind the scenes), so all options for how to fit and represent the regression model can be used for both functions. To get the old behavior of *regplot()*, use *jointplot()* with `kind="reg"`.
- As noted above, *lmplot()* has been rewritten to exploit the *FacetGrid* machinery. This involves a few changes. The `color` keyword argument has been replaced with `hue`, for better consistency across the package. The `hue` parameter will always take a variable *name*, while `color` will take a color name or (in some cases) a palette. The *lmplot()* function now returns the *FacetGrid* used to draw the plot instance.
- The functions that interact with matplotlib rc parameters have been updated and standardized. There are now three pairs of functions, *axes_style()* and *set_style()*, *plotting_context()* and *set_context()*, and *color_palette()* and *set_palette()*. In each case, the pairs take the exact same arguments. The first function defines and returns the parameters, and the second sets the matplotlib defaults. Additionally, the first function in each pair can be used in a `with` statement to temporarily change the defaults. Both the style and context functions also now accept a dictionary of matplotlib rc parameters to override the seaborn defaults, and *set()* now also takes a dictionary to update any of the matplotlib defaults. See the tutorial for more information.
- The `nogrid` style has been deprecated and changed to `white` for more uniformity (i.e. there are now `darkgrid`, `dark`, `whitegrid`, and `white` styles).

### 1.17.3 Other changes

#### Using the package

- If you want to use plotting functions provided by the package without setting the matplotlib style to a seaborn theme, you can now do `import seaborn.apionly as sns` or `from seaborn.apionly import lmplot`, etc. This is using the (also new) `reset_orig()` function, which returns the rc parameters to what they are at matplotlib import time — i.e. they will respect any custom `matplotlibrc` settings on top of the matplotlib defaults.
- The dependency load of the package has been reduced. It can now be installed and used with only `numpy`, `scipy`, `matplotlib`, and `pandas`. Although `statsmodels` is still recommended for full functionality, it is not required.

#### Plotting functions

- `lmplot()` (and `regplot()`) have two new options for fitting regression models: `lowess` and `robust`. The former fits a nonparametric smoother, while the latter fits a regression using methods that are less sensitive to outliers.
- The regression uncertainty in `lmplot()` and `regplot()` is now estimated with fewer bootstrap iterations, so plotting should be faster.
- The univariate `kdeplot()` can now be drawn as a *cumulative* density plot.
- Changed `interactplot()` to use a robust calculation of the data range when finding default limits for the contour colormap to work better when there are outliers in the data.

#### Style

- There is a new style, `dark`, which shares most features with `darkgrid` but does not draw a grid by default.
- There is a new function, `offset_spines()`, and a corresponding option in `despine()` called `trim`. Together, these can be used to make plots where the axis spines are offset from the main part of the figure and limited within the range of the ticks. This is recommended for use with the `ticks` style.
- Other aspects of the seaborn styles have been tweaked for more attractive plots.

## 1.18 v0.2.1 (December 2013)

This is a bugfix release, with no new features.

### 1.18.1 Bug fixes

- Changed the mechanics of `violinplot()` and `boxplot()` when using a `Series` object as data and performing a `groupby` to assign data to bins to address a problem that arises in Pandas 0.13.
- Additionally fixed the `groupby` code to work with all styles of group specification (specifically, using a dictionary or a function now works).
- Fixed a bug where artifacts from the kde fitting could undershoot and create a plot where the density axis starts below 0.
- Ensured that data used for kde fitting is double-typed to avoid a low-level statsmodels error.

- Changed the implementation of the histogram bin-width reference rule to take a ceiling of the estimated number of bins.

## 1.19 v0.2.0 (December 2013)

This is a major release from 0.1 with a number of API changes, enhancements, and bug fixes.

Highlights include an overhaul of timeseries plotting to work intelligently with dataframes, the new function `interactplot()` for visualizing continuous interactions, bivariate kernel density estimates in `kdeplot()`, and significant improvements to color palette handling.

Version 0.2 also introduces experimental support for Python 3.

In addition to the library enhancements, the documentation has been substantially rewritten to reflect the new features and improve the presentation of the ideas behind the package.

### 1.19.1 API changes

- The `tsplot()` function was rewritten to accept data in a long-form `DataFrame` and to plot different traces by condition. This introduced a relatively minor but unavoidable API change, where instead of doing `sns.tsplot(time, heights)`, you now must do `sns.tsplot(heights, time=time)` (the `time` parameter is now optional, for quicker specification of simple plots). Additionally, the `"obs_traces"` and `"obs_points"` error styles in `tsplot()` have been renamed to `"unit_traces"` and `"unit_points"`, respectively.
- Functions that fit kernel density estimates (`kdeplot()` and `violinplot()`) now use `statsmodels` instead of `scipy`, and the parameters that influence the density estimate have changed accordingly. This allows for increased flexibility in specifying the bandwidth and kernel, and smarter choices for defining the range of the support. Default options should produce plots that are very close to the old defaults.
- The `kdeplot()` function now takes a second positional argument of data for drawing bivariate densities.
- The `violin()` function has been changed to `violinplot()`, for consistency. In 0.2, `violin` will still work, but it will fire a `UserWarning`.

### 1.19.2 New plotting functions

- The `interactplot()` function draws a contour plot for an interactive linear model (i.e., the contour shows `y-hat` from the model `y ~ x1 * x2`) over a scatterplot between the two predictor variables. This plot should aid the understanding of an interaction between two continuous variables.
- The `kdeplot()` function can now draw a bivariate density estimate as a contour plot if provided with two-dimensional input data.
- The `palplot()` function provides a simple grid-based visualization of a color palette.

### 1.19.3 Other changes

#### Plotting functions

- The `corrplot()` function can be drawn without the correlation coefficient annotation and with variable names on the side of the plot to work with large datasets.
- Additionally, `corrplot()` sets the color palette intelligently based on the direction of the specified test.
- The `distplot()` histogram uses a reference rule to choose the bin size if it is not provided.
- Added the `x_bins` option in `lmplot()` for binning a continuous predictor variable, allowing for clearer trends with many datapoints.
- Enhanced support for labeling plot elements and axes based on `name` attributes in several distribution plot functions and `tsplot()` for smarter Pandas integration.
- Scatter points in `lmplot()` are slightly transparent so it is easy to see where observations overlap.
- Added the `order` parameter to `boxplot()` and `violinplot()` to control the order of the bins when using a Pandas object.
- When an `ax` argument is not provided to a plotting function, it grabs the currently active axis instead of drawing a new one.

#### Color palettes

- Added the `dark_palette()` and `blend_palette()` for on-the-fly creation of blended color palettes.
- The color palette machinery is now intelligent about qualitative ColorBrewer palettes (`Set1`, `Paired`, etc.), which are properly treated as discrete.
- Seaborn color palettes (`deep`, `muted`, etc.) have been standardized in terms of basic hue sequence, and all palettes now have 6 colors.
- Introduced `{mpl_palette}_d` palettes, which make a palette with the basic color scheme of the source palette, but with a sequential blend from dark instead of light colors for use with line/scatter/contour plots.
- Added the `palette_context()` function for blockwise color palettes controlled by a `with` statement.

#### Plot styling

- Added the `despine()` function for easily removing plot spines.
- A new plot style, `"ticks"` has been added.
- Tick labels are padded a bit farther from the axis in all styles, avoiding collisions at (0, 0).

#### General package issues

- Reorganized the package by breaking up the monolithic `plotobjs` module into smaller modules grouped by general objective of the constituent plots.
- Removed the `scikits-learn` dependency in `moss`.
- Installing with `pip` should automatically install most missing dependencies.
- The example notebooks are now used as an automated test suite.

### 1.19.4 Bug fixes

- Fixed a bug where labels did not match data for `boxplot()` and `violinplot()` when using a groupby.
- Fixed a bug in the `desaturate()` function.
- Fixed a bug in the `coefplot()` figure size calculation.
- Fixed a bug where `regplot()` choked on list input.
- Fixed buggy behavior when drawing horizontal boxplots.
- Specifying bins for the `distplot()` histogram now works.
- Fixed a bug where `kdeplot()` would reset the axis height and cut off existing data.
- All axis styling has been moved out of the top-level `seaborn.set()` function, so context or color palette can be cleanly changed.

CHAPTER

TWO

# INSTALLING AND GETTING STARTED

Official releases of seaborn can be installed from PyPI:

pip install seaborn

The basic invocation of `pip` will install seaborn and, if necessary, its mandatory dependencies. It is possible to include optional dependencies that give access to a few advanced features:

pip install seaborn[all]

The library is also included as part of the Anaconda distribution, and it can be installed with `conda`:

conda install seaborn

## 2.1 Dependencies

### 2.1.1 Supported Python versions

- Python 3.7+

### 2.1.2 Mandatory dependencies

- numpy
- pandas
- matplotlib

### 2.1.3 Optional dependencies

- statsmodels, for advanced regression plots
- scipy, for clustering matrices and some advanced options
- fastcluster, faster clustering of large matrices

## 2.2 Quickstart

Once you have seaborn installed, you're ready to get started. To test it out, you could load and plot one of the example datasets:

```
import seaborn as sns
df = sns.load_dataset("penguins")
sns.pairplot(df, hue="species")
```

If you're working in a Jupyter notebook or an IPython terminal with matplotlib mode enabled, you should immediately see *the plot*. Otherwise, you may need to explicitly call `matplotlib.pyplot.show()`:

```
import matplotlib.pyplot as plt
plt.show()
```

While you can get pretty far with only seaborn imported, having access to matplotlib functions is often useful. The tutorials and API documentation typically assume the following imports:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

## 2.3 Debugging install issues

The seaborn codebase is pure Python, and the library should generally install without issue. Occasionally, difficulties will arise because the dependencies include compiled code and link to system libraries. These difficulties typically manifest as errors on import with messages such as `"DLL load failed"`. To debug such problems, read through the exception trace to figure out which specific library failed to import, and then consult the installation docs for that package to see if they have tips for your particular system.

In some cases, an installation of seaborn will appear to succeed, but trying to import it will raise an error with the message `"No module named seaborn"`. This usually means that you have multiple Python installations on your system and that your `pip` or `conda` points towards a different installation than where your interpreter lives. Resolving this issue will involve sorting out the paths on your system, but it can sometimes be avoided by invoking `pip` with `python -m pip install seaborn`.

## 2.4 Getting help

If you think you've encountered a bug in seaborn, please report it on the GitHub issue tracker. To be useful, bug reports must include the following information:

- A reproducible code example that demonstrates the problem
- The output that you are seeing (an image of a plot, or the error message)
- A clear explanation of why you think something is wrong
- The specific versions of seaborn and matplotlib that you are working with

Bug reports are easiest to address if they can be demonstrated using one of the example datasets from the seaborn docs (i.e. with `load_dataset()`). Otherwise, it is preferable that your example generate synthetic data to reproduce the problem. If you can only demonstrate the issue with your actual dataset, you will need to share it, ideally as a csv.

If you've encountered an error, searching the specific text of the message before opening a new issue can often help you solve the problem quickly and avoid making a duplicate report.

Because matplotlib handles the actual rendering, errors or incorrect outputs may be due to a problem in matplotlib rather than one in seaborn. It can save time if you try to reproduce the issue in an example that uses only matplotlib, so that you can report it in the right place. But it is alright to skip this step if it's not obvious how to do it.

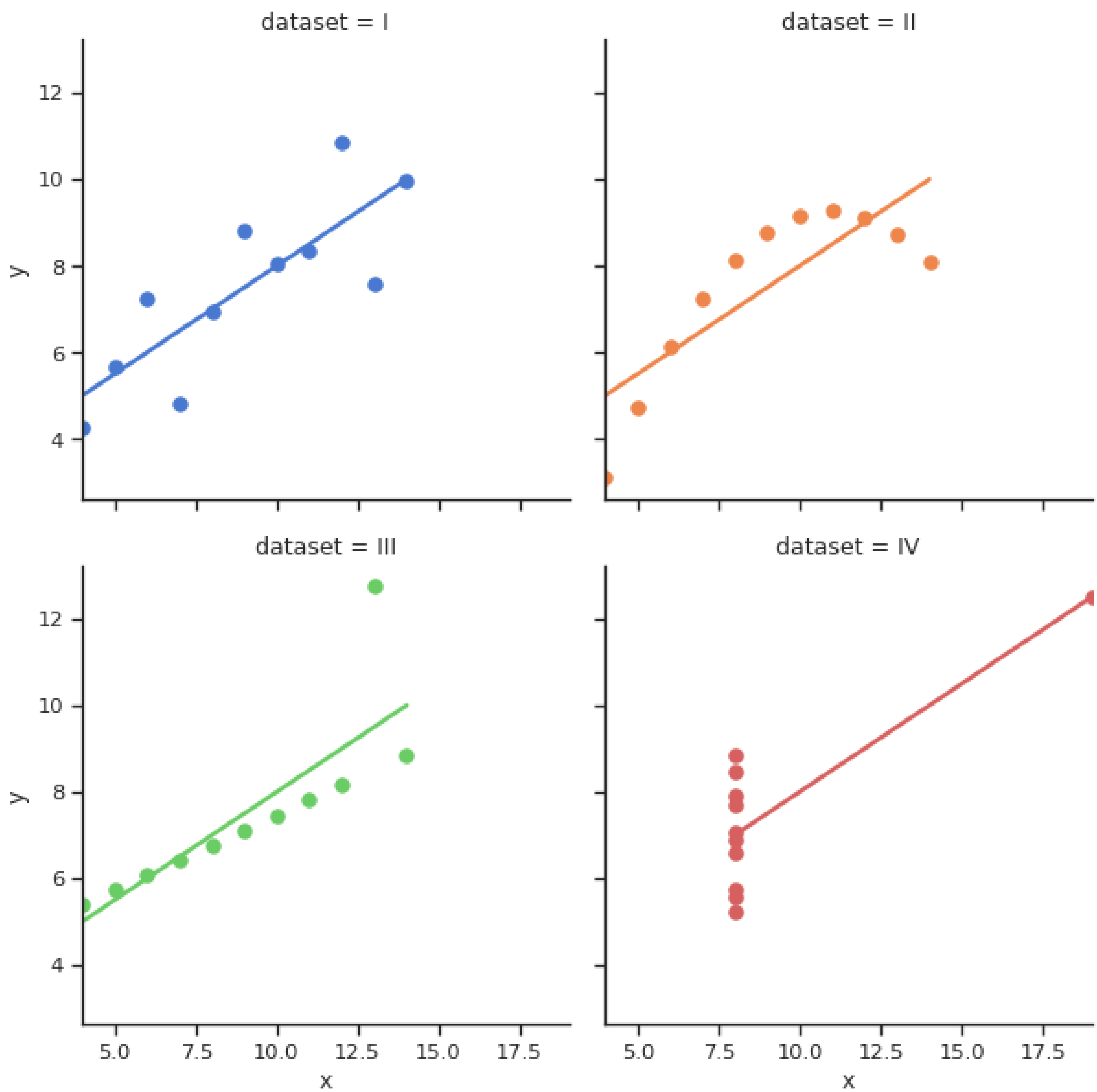
General support questions are more at home on either stackoverflow or discourse, which have a larger audience of people who will see your post and may be able to offer assistance. StackOverflow is better for specific issues, while discourse is better for more open-ended discussion. Your chance of getting a quick answer will be higher if you include runnable code, a precise statement of what you are hoping to achieve, and a clear explanation of the problems that you have encountered.

CHAPTER

THREE

# EXAMPLE GALLERY

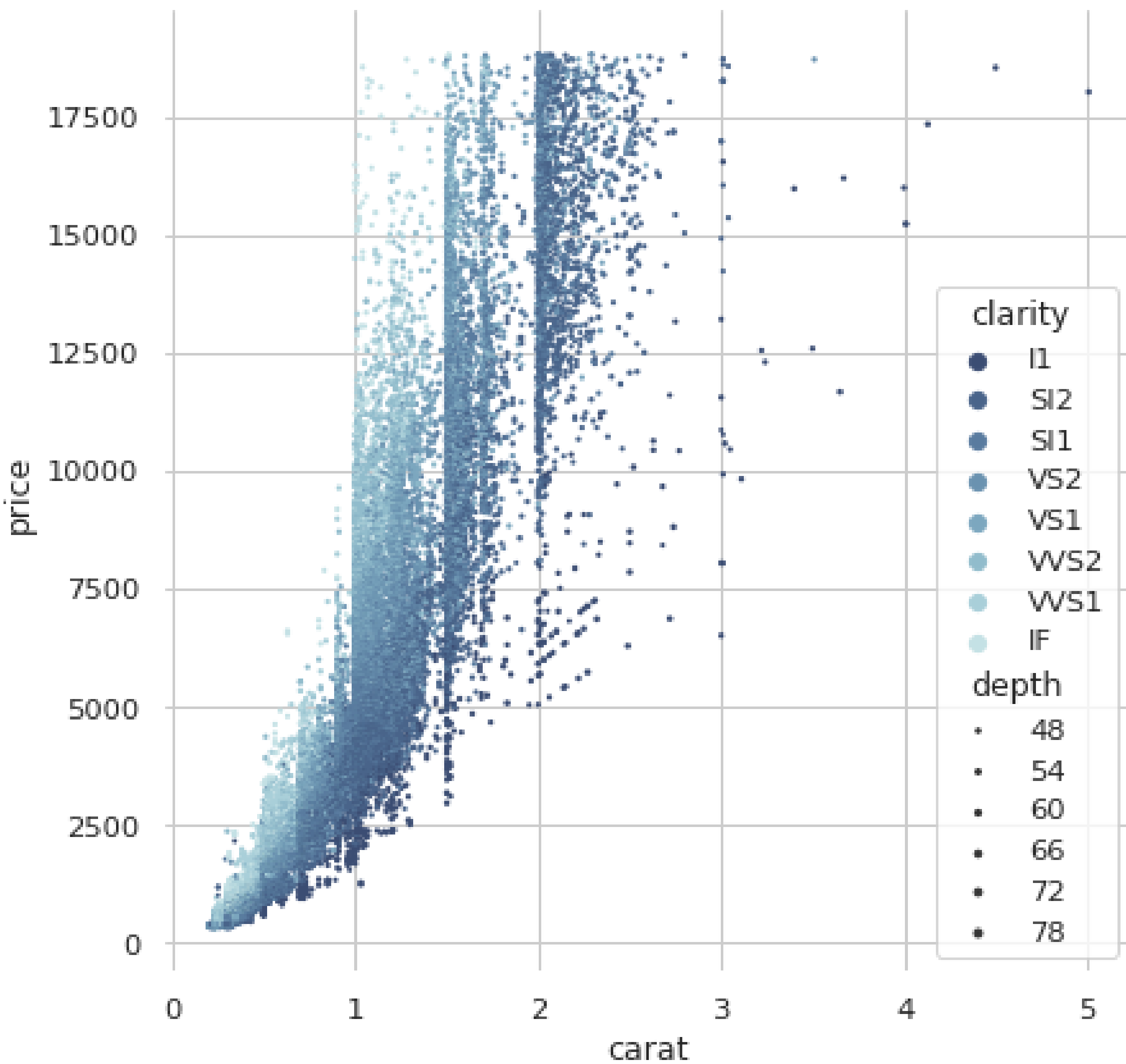
## 3.1 Anscombe's quartet



**seaborn components used:** *set_theme()*, *load_dataset()*, *lmplot()*

```
import seaborn as sns
sns.set_theme(style="ticks")

# Load the example dataset for Anscombe's quartet
df = sns.load_dataset("anscombe")

# Show the results of a linear regression within each dataset
sns.lmplot(x="x", y="y", col="dataset", hue="dataset", data=df,
           col_wrap=2, ci=None, palette="muted", height=4,
           scatter_kws={"s": 50, "alpha": 1})
```

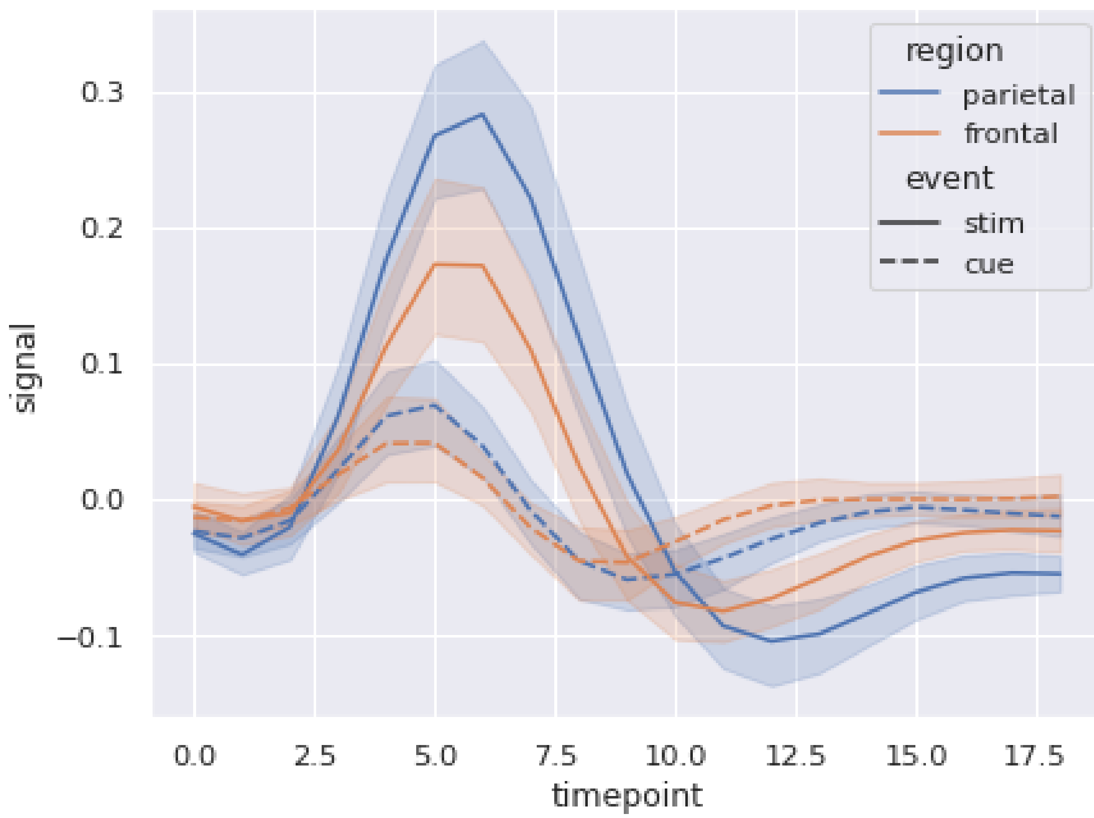
## 3.2 Scatterplot with multiple semantics




**seaborn components used:** *set_theme()*, *load_dataset()*, *despine()*, *scatterplot()*

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="whitegrid")

# Load the example diamonds dataset
diamonds = sns.load_dataset("diamonds")

# Draw a scatter plot while assigning point colors and sizes to different
# variables in the dataset
f, ax = plt.subplots(figsize=(6.5, 6.5))
sns.despine(f, left=True, bottom=True)
clarity_ranking = ["I1", "SI2", "SI1", "VS2", "VS1", "VVS2", "VVS1", "IF"]
sns.scatterplot(x="carat", y="price",
                hue="clarity", size="depth",
                palette="ch:r=-.2,d=.3_r",
                hue_order=clarity_ranking,
                sizes=(1, 8), linewidth=0,
                data=diamonds, ax=ax)
```

## 3.3 Timeseries plot with error bands




**seaborn components used:** *set_theme()*, *load_dataset()*, *lineplot()*

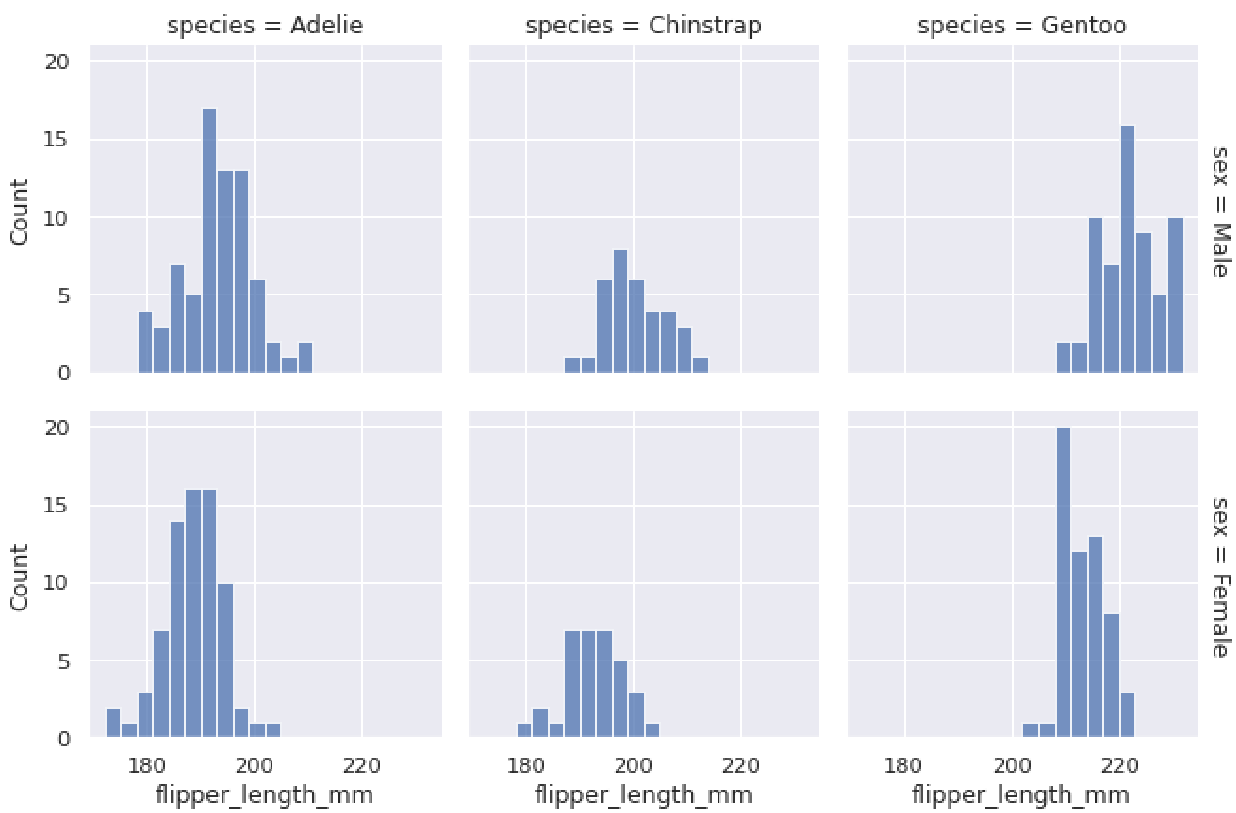
```
import seaborn as sns
sns.set_theme(style="darkgrid")
```

(continues on next page)

(continued from previous page)

```
# Load an example dataset with long-form data
fmri = sns.load_dataset("fmri")

# Plot the responses for different events and regions
sns.lineplot(x="timepoint", y="signal",
             hue="region", style="event",
             data=fmri)
```

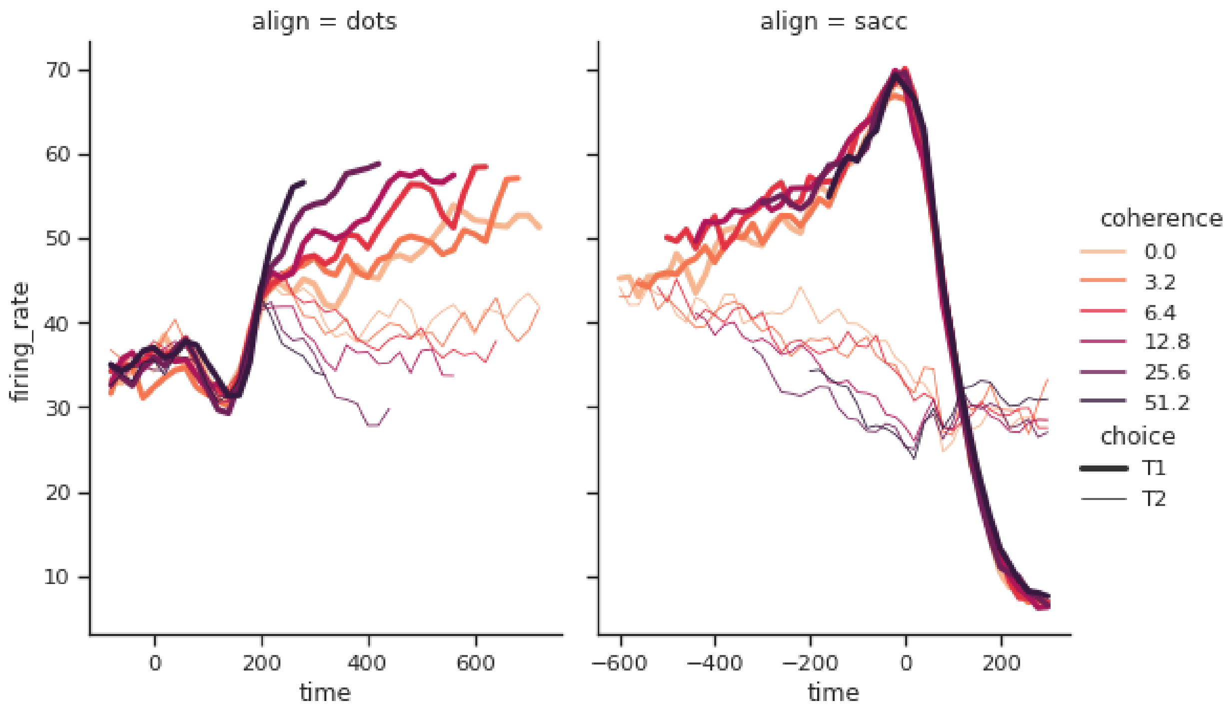
## 3.4 Facetting histograms by subsets of data




**seaborn components used:** *set_theme()*, *load_dataset()*, *displot()*

```
import seaborn as sns

sns.set_theme(style="darkgrid")
df = sns.load_dataset("penguins")
sns.displot(
    df, x="flipper_length_mm", col="species", row="sex",
    binwidth=3, height=3, facet_kws=dict(margin_titles=True),
)
```

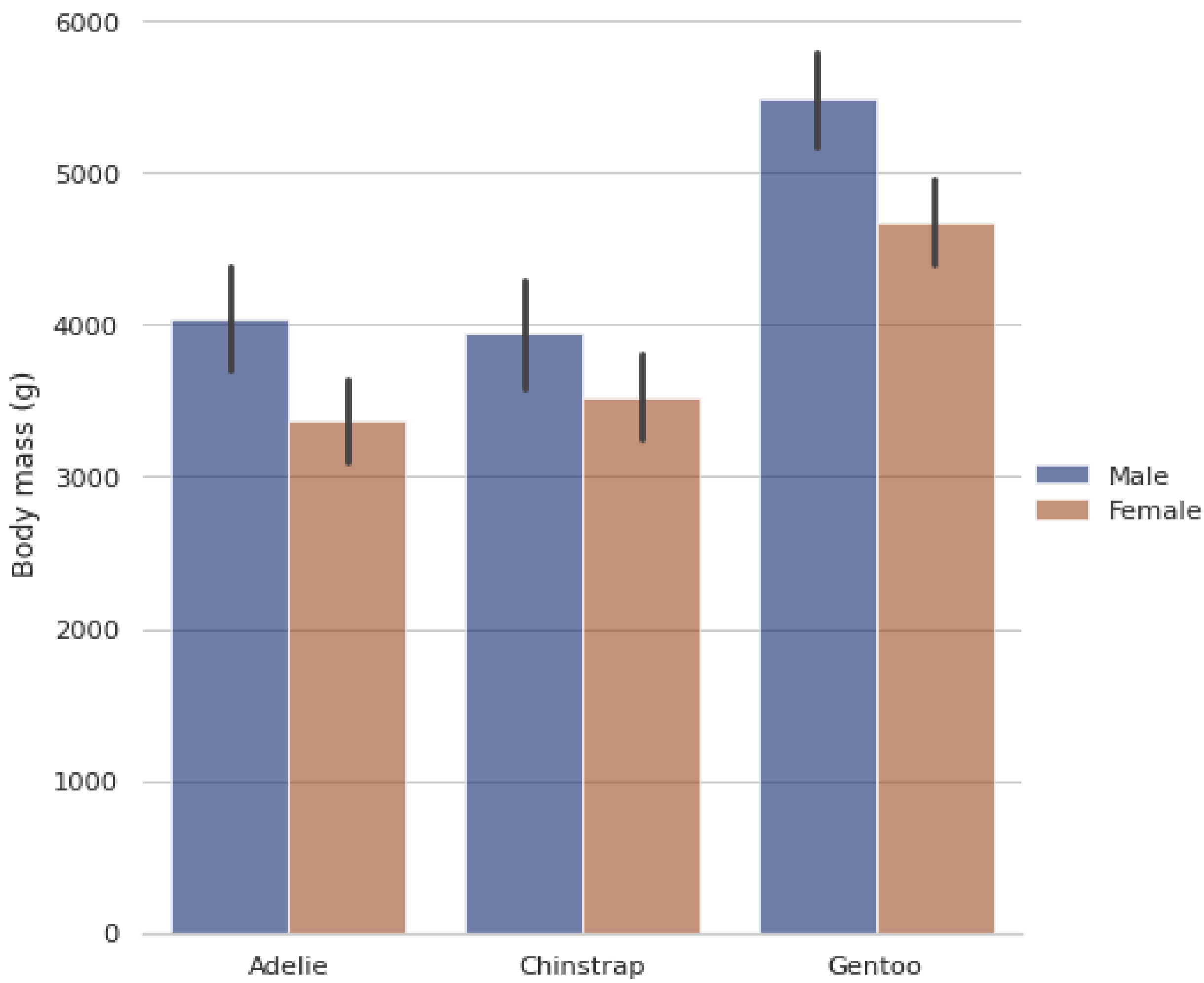
## 3.5 Line plots on multiple facets




**seaborn components used:** *set_theme()*, *load_dataset()*, *color_palette()*, *relplot()*

```
import seaborn as sns
sns.set_theme(style="ticks")

dots = sns.load_dataset("dots")

# Define the palette as a list to specify exact values
palette = sns.color_palette("rocket_r")

# Plot the lines on two facets
sns.relplot(
    data=dots,
    x="time", y="firing_rate",
    hue="coherence", size="choice", col="align",
    kind="line", size_order=["T1", "T2"], palette=palette,
    height=5, aspect=.75, facet_kws=dict(sharex=False),
)
```

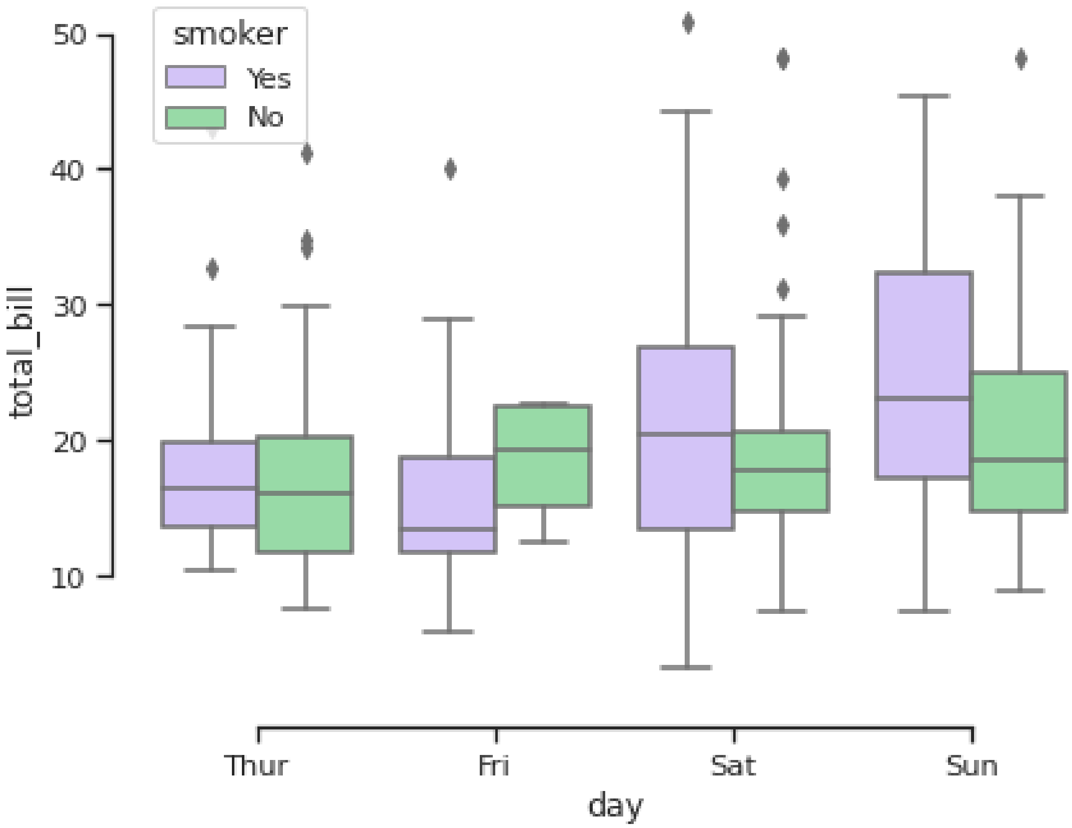
## 3.6 Grouped barplots




**seaborn components used:** *set_theme()*, *load_dataset()*, *catplot()*

```
import seaborn as sns
sns.set_theme(style="whitegrid")

penguins = sns.load_dataset("penguins")

# Draw a nested barplot by species and sex
g = sns.catplot(
    data=penguins, kind="bar",
    x="species", y="body_mass_g", hue="sex",
    ci="sd", palette="dark", alpha=.6, height=6
)
g.despine(left=True)
g.set_axis_labels("", "Body mass (g)")
g.legend.set_title("")
```

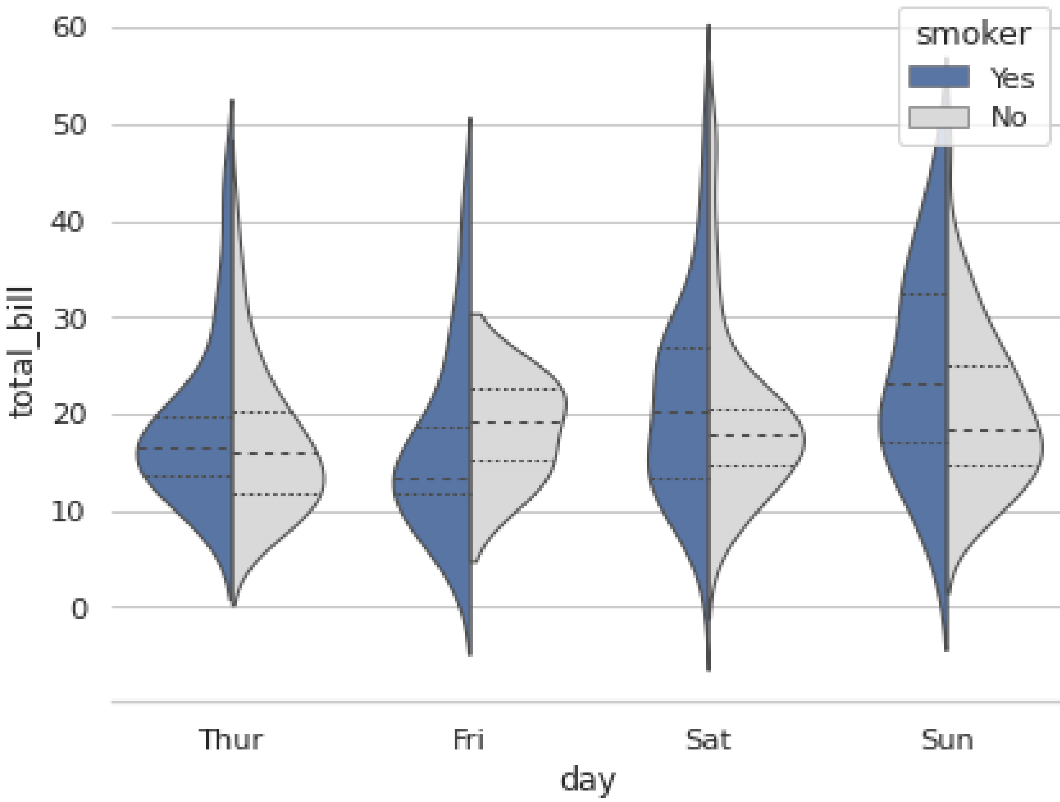
## 3.7 Grouped boxplots




**seaborn components used:** *set_theme()*, *load_dataset()*, *boxplot()*, *despine()*

```
import seaborn as sns
sns.set_theme(style="ticks", palette="pastel")

# Load the example tips dataset
tips = sns.load_dataset("tips")

# Draw a nested boxplot to show bills by day and time
sns.boxplot(x="day", y="total_bill",
            hue="smoker", palette=["m", "g"],
            data=tips)
sns.despine(offset=10, trim=True)
```

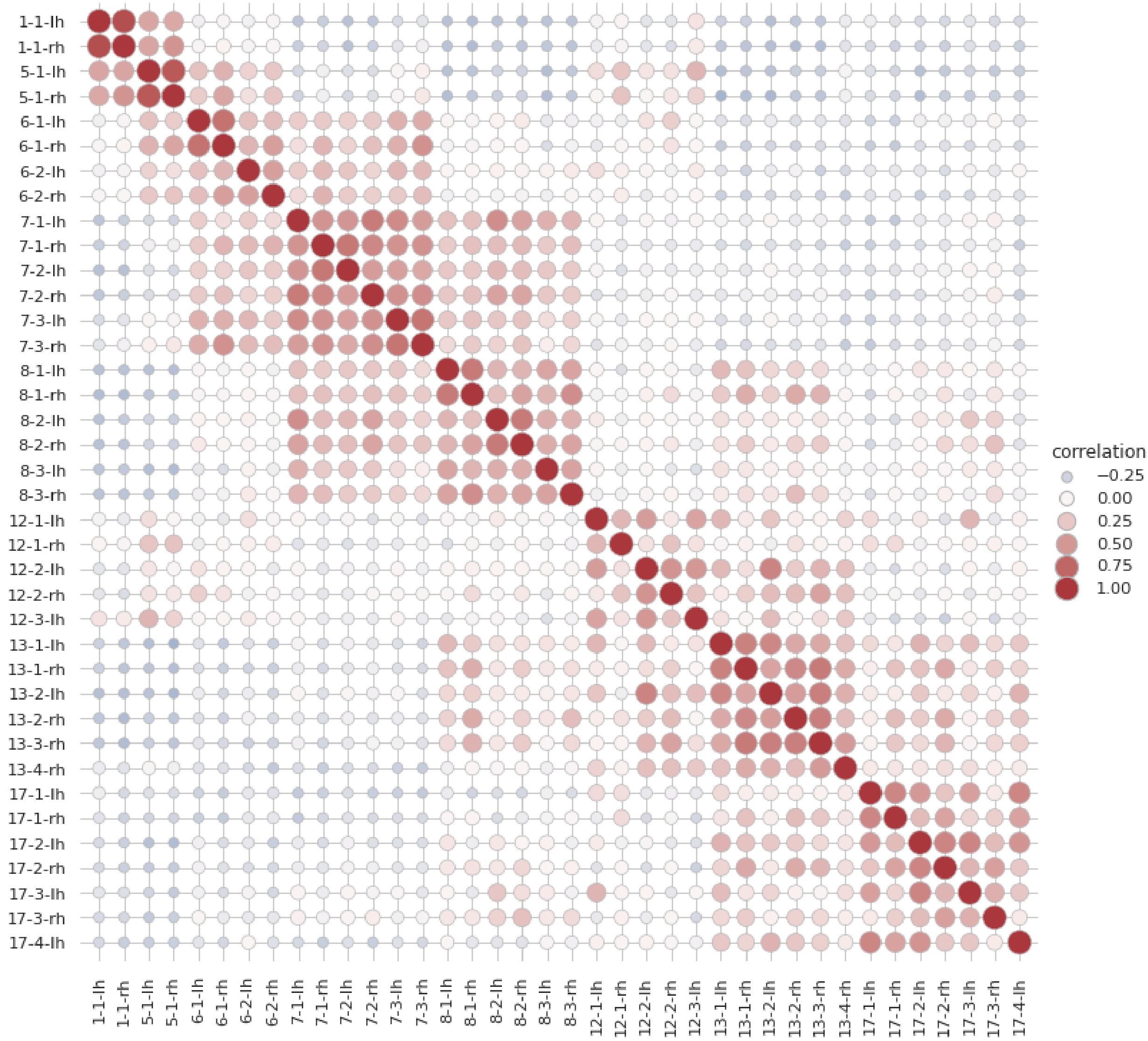
## 3.8 Grouped violinplots with split violins




**seaborn components used:** *set_theme()*, *load_dataset()*, *violinplot()*, *despine()*

```
import seaborn as sns
sns.set_theme(style="whitegrid")

# Load the example tips dataset
tips = sns.load_dataset("tips")

# Draw a nested violinplot and split the violins for easier comparison
sns.violinplot(data=tips, x="day", y="total_bill", hue="smoker",
               split=True, inner="quart", linewidth=1,
               palette={"Yes": "b", "No": ".85"})
sns.despine(left=True)
```

## 3.9 Scatterplot heatmap




**seaborn components used:** *set_theme()*, *load_dataset()*, *relplot()*

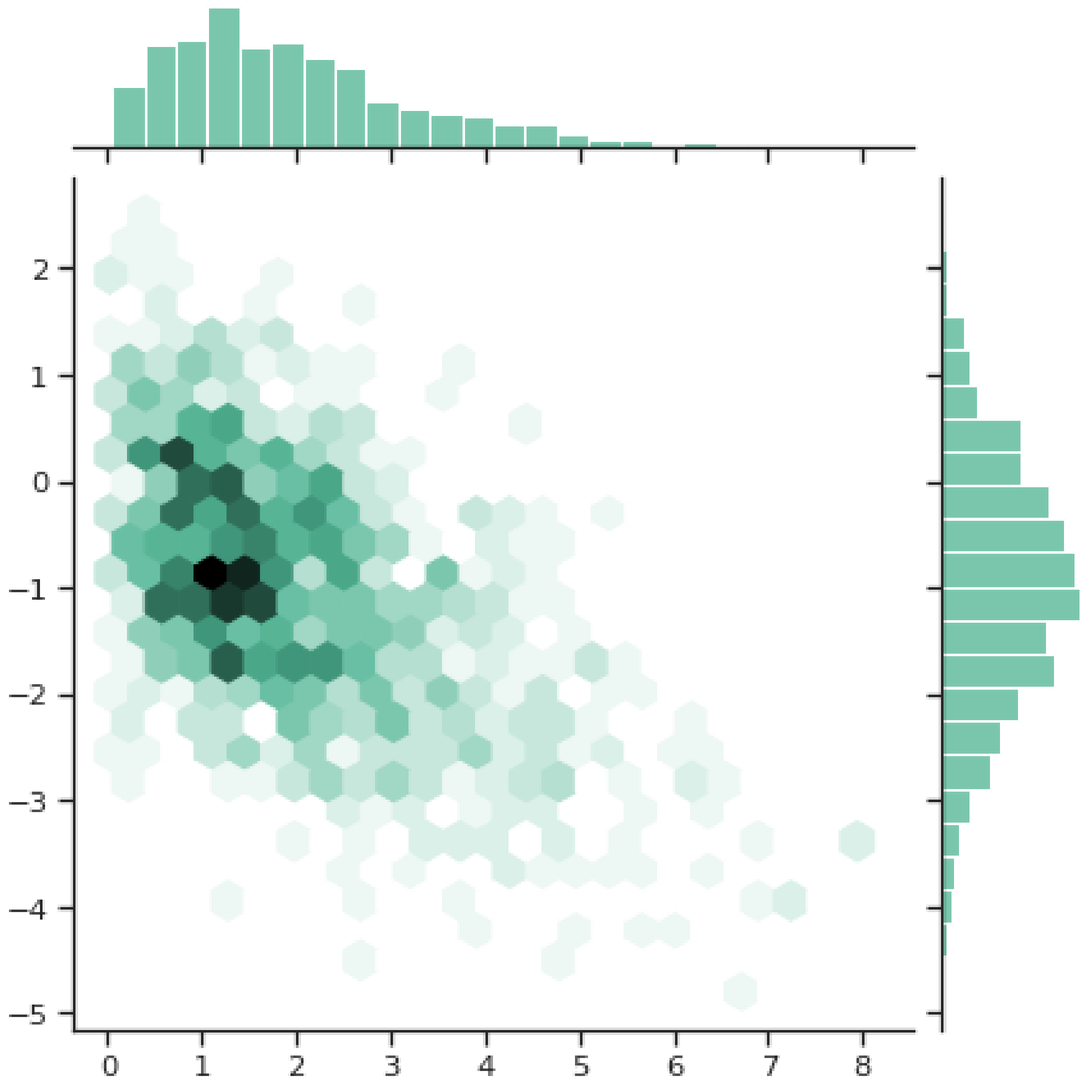
```
import seaborn as sns
sns.set_theme(style="whitegrid")

# Load the brain networks dataset, select subset, and collapse the multi-index
df = sns.load_dataset("brain_networks", header=[0, 1, 2], index_col=0)

used_networks = [1, 5, 6, 7, 8, 12, 13, 17]
used_columns = (df.columns
                  .get_level_values("network")
                  .astype(int)
                  .isin(used_networks))
df = df.loc[:, used_columns]

df.columns = df.columns.map("-".join)
```

(continues on next page)

(continued from previous page)

```
# Compute a correlation matrix and convert to long-form
corr_mat = df.corr().stack().reset_index(name="correlation")

# Draw each cell as a scatter point with varying size and color
g = sns.relplot(
    data=corr_mat,
    x="level_0", y="level_1", hue="correlation", size="correlation",
    palette="vlag", hue_norm=(-1, 1), edgecolor=".7",
    height=10, sizes=(50, 250), size_norm=(-.2, .8),
)

# Tweak the figure to finalize
g.set(xlabel="", ylabel="", aspect="equal")
g.despine(left=True, bottom=True)
g.ax.margins(.02)
for label in g.ax.get_xticklabels():
    label.set_rotation(90)
for artist in g.legend.legendHandles:
    artist.set_edgecolor(".7")
```

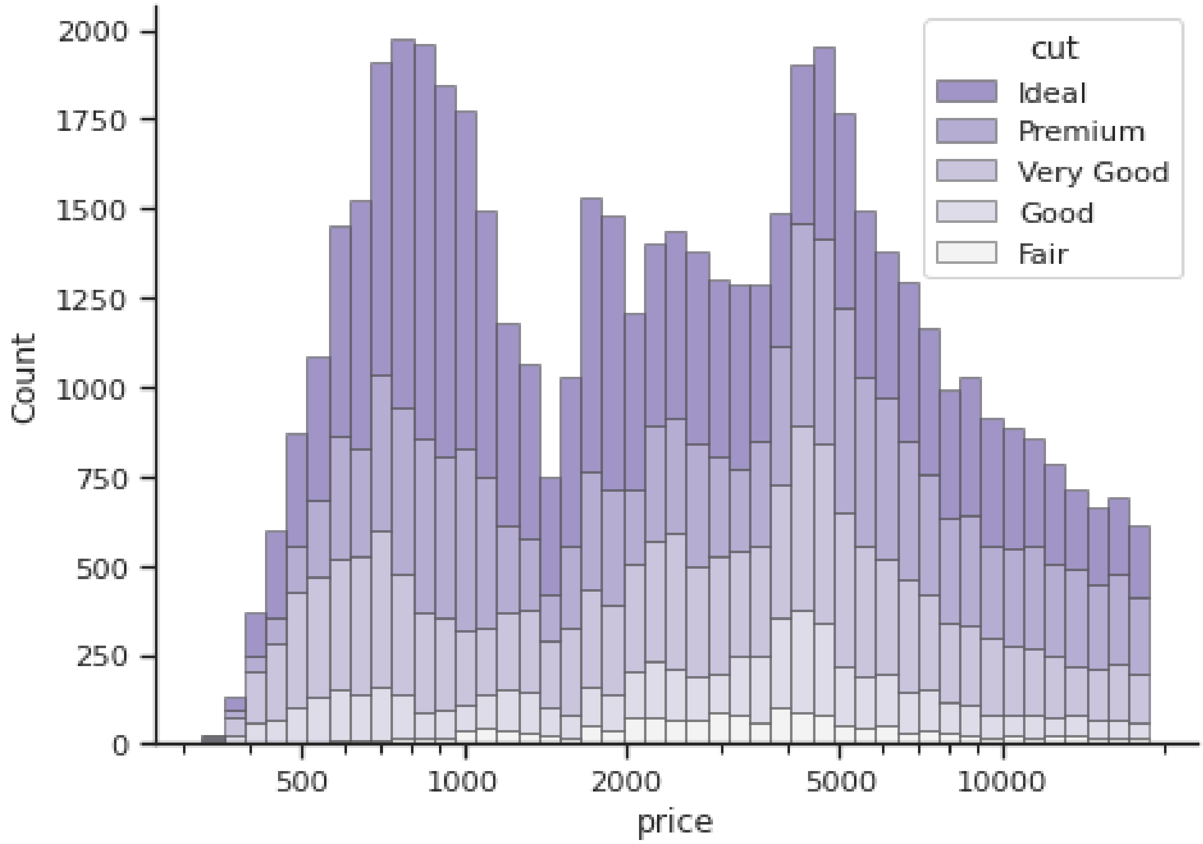
## 3.10 Hexbin plot with marginal distributions




**seaborn components used:** *set_theme()*, *jointplot()*

```
import numpy as np
import seaborn as sns
sns.set_theme(style="ticks")

rs = np.random.RandomState(11)
x = rs.gamma(2, size=1000)
y = -.5 * x + rs.normal(size=1000)

sns.jointplot(x=x, y=y, kind="hex", color="#4CB391")
```

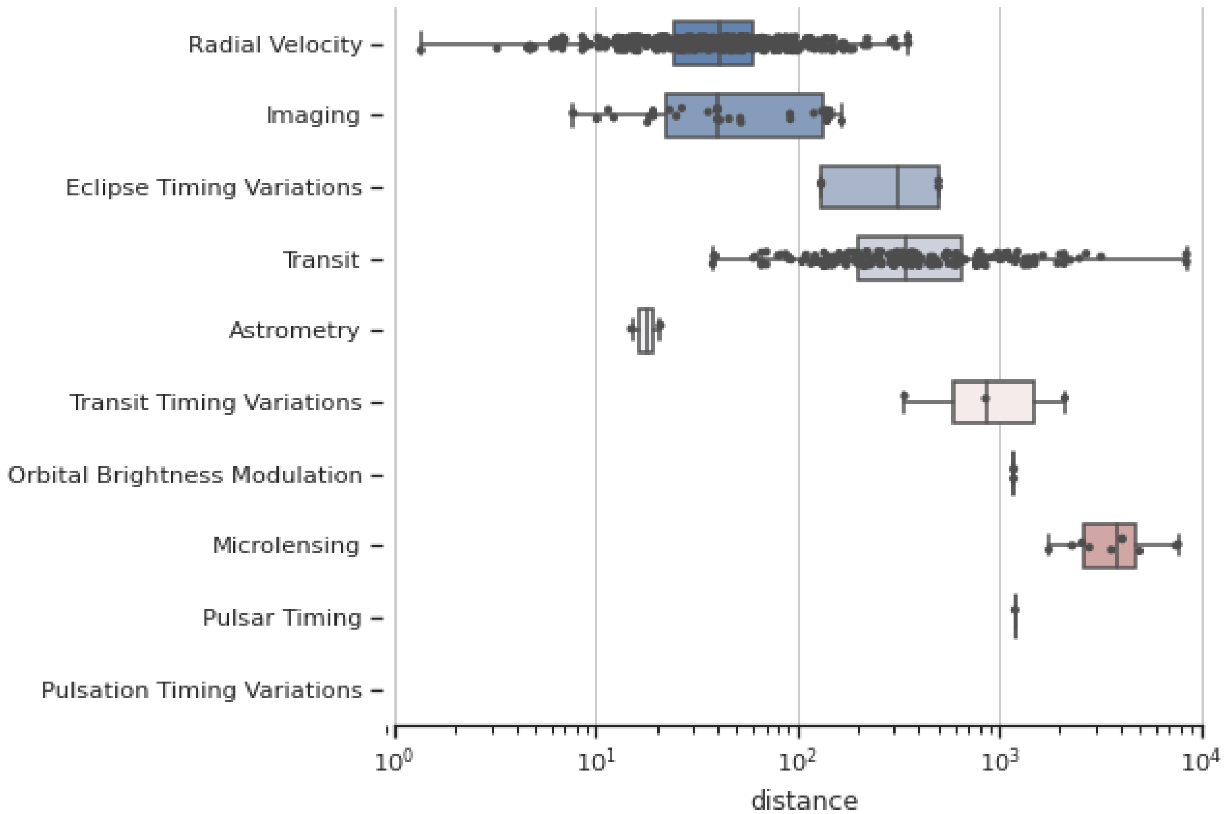
## 3.11 Stacked histogram on a log scale




**seaborn components used:** *set_theme()*, *load_dataset()*, *despine()*, *histplot()*

```
import seaborn as sns
import matplotlib as mpl
import matplotlib.pyplot as plt

sns.set_theme(style="ticks")

diamonds = sns.load_dataset("diamonds")

f, ax = plt.subplots(figsize=(7, 5))
sns.despine(f)

sns.histplot(
    diamonds,
    x="price", hue="cut",
    multiple="stack",
    palette="light:m_r",
    edgecolor=".3",
    linewidth=.5,
    log_scale=True,
)
ax.xaxis.set_major_formatter(mpl.ticker.ScalarFormatter())
ax.set_xticks([500, 1000, 2000, 5000, 10000])
```

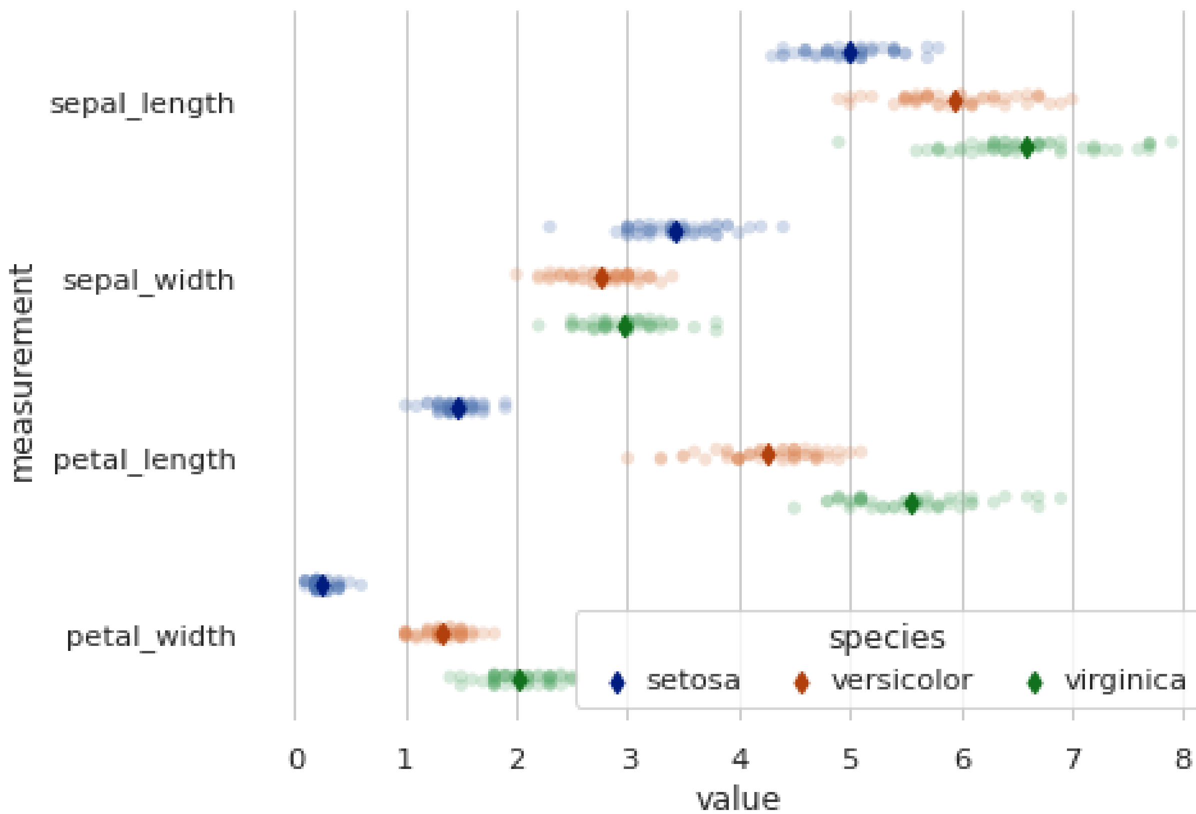
## 3.12 Horizontal boxplot with observations




**seaborn components used:** *set_theme()*, *load_dataset()*, *boxplot()*, *stripplot()*, *despine()*

```
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="ticks")

# Initialize the figure with a logarithmic x axis
f, ax = plt.subplots(figsize=(7, 6))
ax.set_xscale("log")

# Load the example planets dataset
planets = sns.load_dataset("planets")

# Plot the orbital period with horizontal boxes
sns.boxplot(x="distance", y="method", data=planets,
            whis=[0, 100], width=.6, palette="vlag")

# Add in points to show each observation
sns.stripplot(x="distance", y="method", data=planets,
              size=4, color=".3", linewidth=0)

# Tweak the visual presentation
ax.xaxis.grid(True)
ax.set(ylabel="")
sns.despine(trim=True, left=True)
```

## 3.13 Conditional means with observations




**seaborn components used:** *set_theme()*, *load_dataset()*, *despine()*, *stripplot()*, *pointplot()*

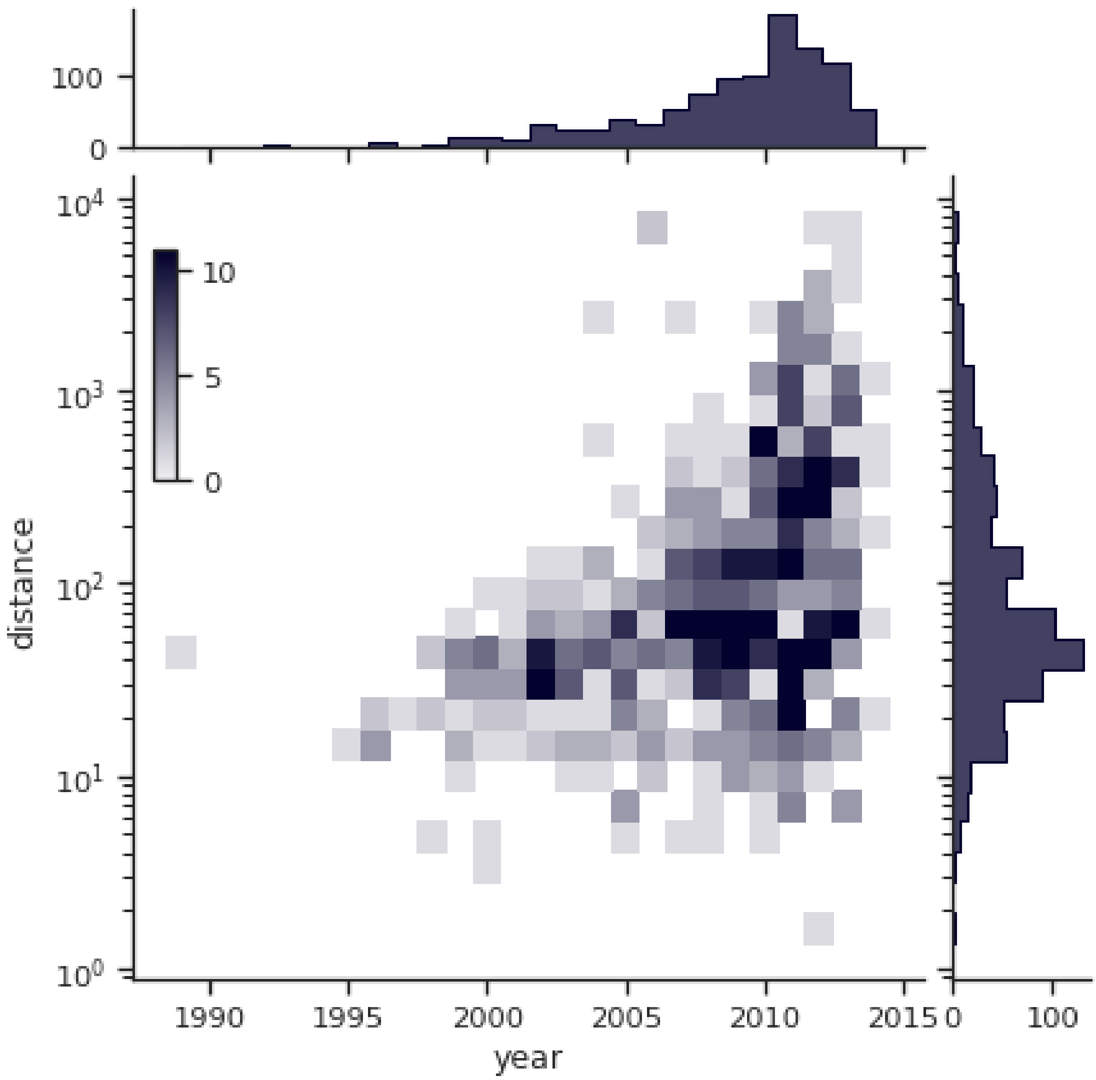
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="whitegrid")
iris = sns.load_dataset("iris")

# "Melt" the dataset to "long-form" or "tidy" representation
iris = pd.melt(iris, "species", var_name="measurement")

# Initialize the figure
f, ax = plt.subplots()
sns.despine(bottom=True, left=True)

# Show each observation with a scatterplot
sns.stripplot(x="value", y="measurement", hue="species",
              data=iris, dodge=True, alpha=.25, zorder=1)

# Show the conditional means
sns.pointplot(x="value", y="measurement", hue="species",
              data=iris, dodge=.532, join=False, palette="dark",
              markers="d", scale=.75, ci=None)
```

(continues on next page)

(continued from previous page)

```
# Improve the legend
handles, labels = ax.get_legend_handles_labels()
ax.legend(handles[3:], labels[3:], title="species",
          handletextpad=0, columnspacing=1,
          loc="lower right", ncol=3, frameon=True)
```

## 3.14 Joint and marginal histograms




**seaborn components used:** *set_theme()*, *load_dataset()*, *JointGrid*

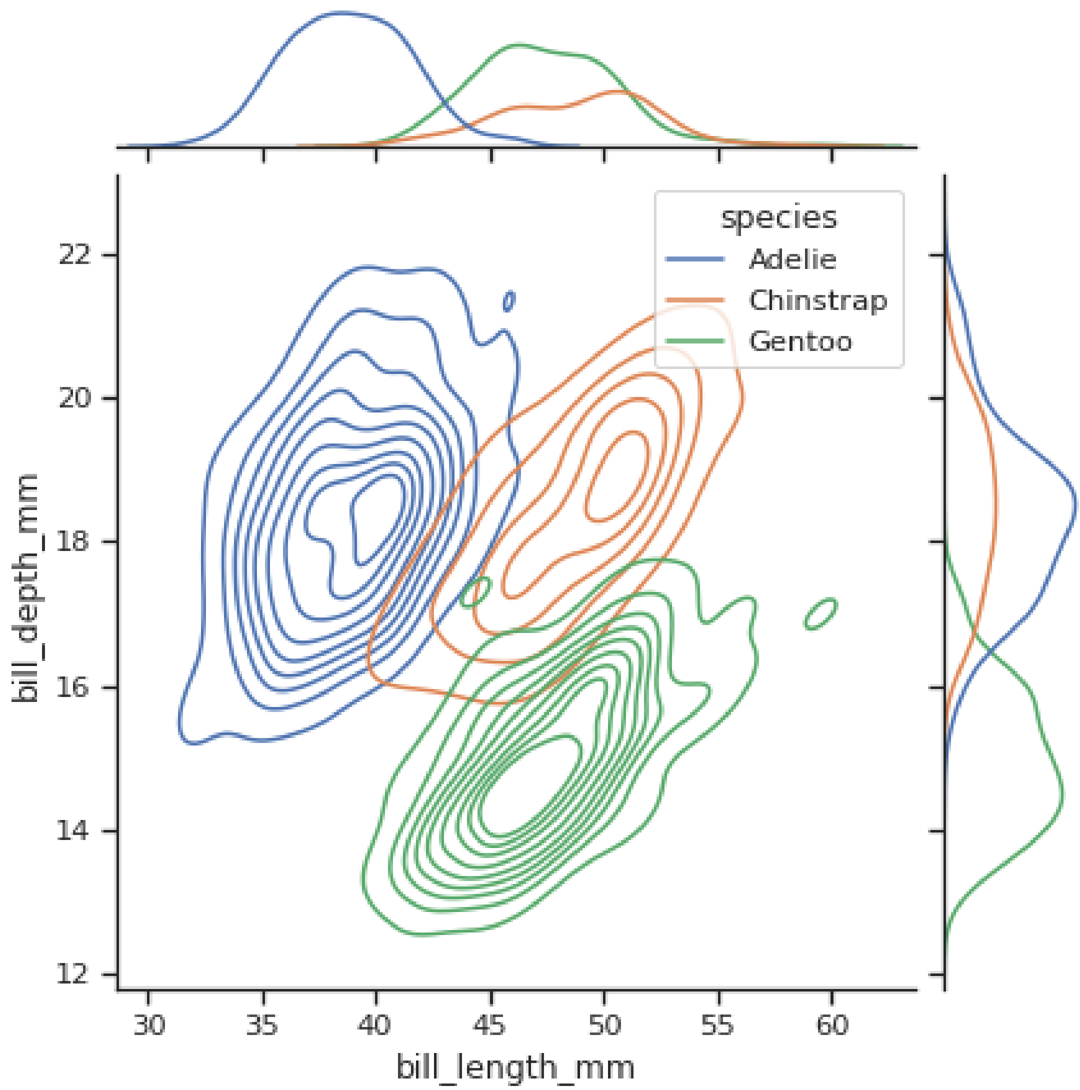
```
import seaborn as sns
sns.set_theme(style="ticks")

# Load the planets dataset and initialize the figure
planets = sns.load_dataset("planets")
```

(continues on next page)

(continued from previous page)

```
g = sns.JointGrid(data=planets, x="year", y="distance", marginal_ticks=True)

# Set a log scaling on the y axis
g.ax_joint.set(yscale="log")

# Create an inset legend for the histogram colorbar
cax = g.fig.add_axes([.15, .55, .02, .2])

# Add the joint and marginal histogram plots
g.plot_joint(
    sns.histplot, discrete=(True, False),
    cmap="light:#03012d", pmax=.8, cbar=True, cbar_ax=cax
)
g.plot_marginals(sns.histplot, element="step", color="#03012d")
```

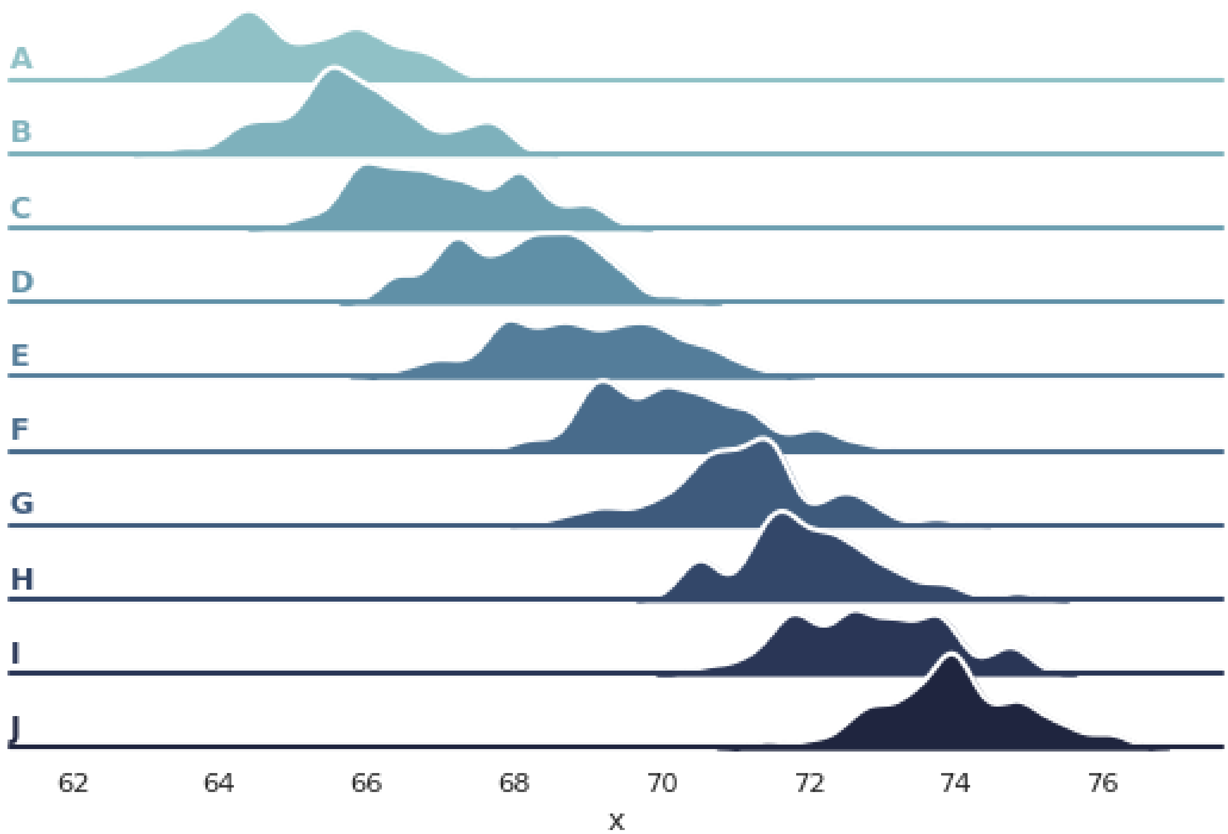
## 3.15 Joint kernel density estimate




**seaborn components used:** *set_theme()*, *load_dataset()*, *jointplot()*

```
import seaborn as sns
sns.set_theme(style="ticks")

# Load the penguins dataset
penguins = sns.load_dataset("penguins")

# Show the joint distribution using kernel density estimation
g = sns.jointplot(
    data=penguins,
    x="bill_length_mm", y="bill_depth_mm", hue="species",
    kind="kde",
)
```

## 3.16 Overlapping densities ('ridge plot')




**seaborn components used:** *set_theme()*, *cubehelix_palette()*, *FacetGrid*

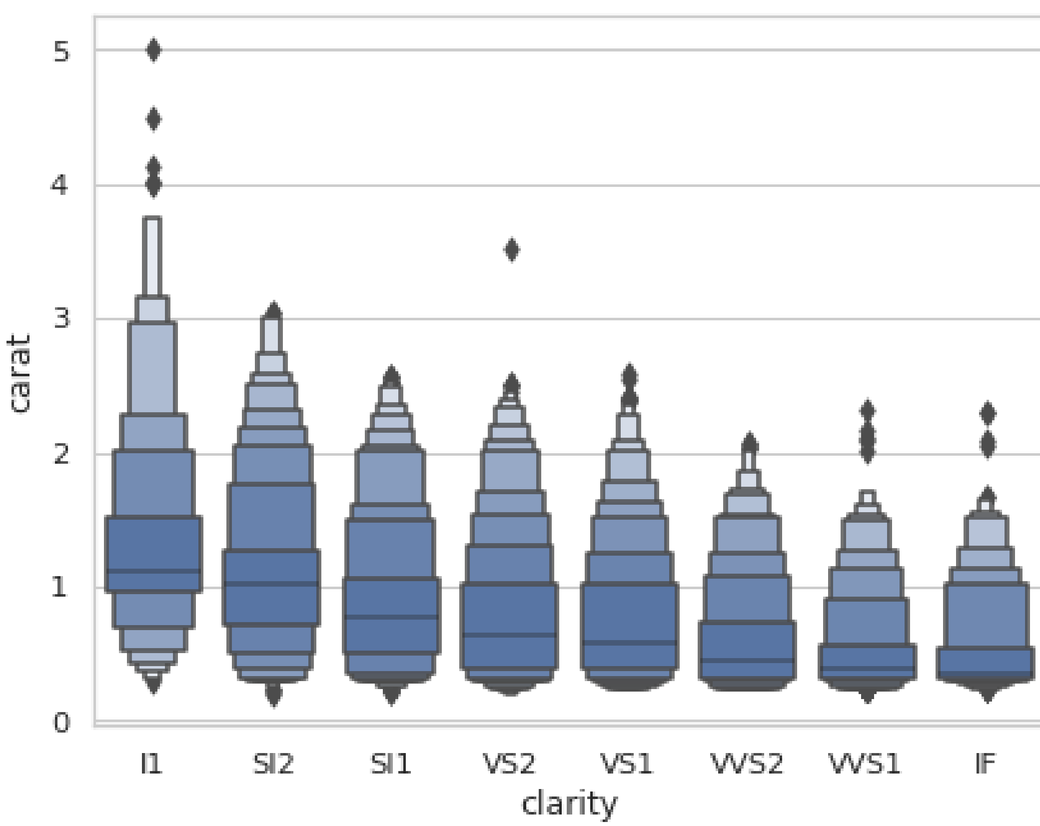
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="white", rc={"axes.facecolor": (0, 0, 0, 0)})

# Create the data
rs = np.random.RandomState(1979)
x = rs.randn(500)
g = np.tile(list("ABCDEFGHIJ"), 50)
df = pd.DataFrame(dict(x=x, g=g))
m = df.g.map(ord)
df["x"] += m

# Initialize the FacetGrid object
pal = sns.cubehelix_palette(10, rot=-.25, light=.7)
g = sns.FacetGrid(df, row="g", hue="g", aspect=15, height=.5, palette=pal)

# Draw the densities in a few steps
g.map(sns.kdeplot, "x",
      bw_adjust=.5, clip_on=False,
      fill=True, alpha=1, linewidth=1.5)
```

(continues on next page)

(continued from previous page)

```
g.map(sns.kdeplot, "x", clip_on=False, color="w", lw=2, bw_adjust=.5)
g.map(plt.axhline, y=0, lw=2, clip_on=False)


# Define and use a simple function to label the plot in axes coordinates
def label(x, color, label):
    ax = plt.gca()
    ax.text(0, .2, label, fontweight="bold", color=color,
            ha="left", va="center", transform=ax.transAxes)


g.map(label, "x")

# Set the subplots to overlap
g.fig.subplots_adjust(hspace=-.25)

# Remove axes details that don't play well with overlap
g.set_titles("")
g.set(yticks=[])
g.despine(bottom=True, left=True)
```

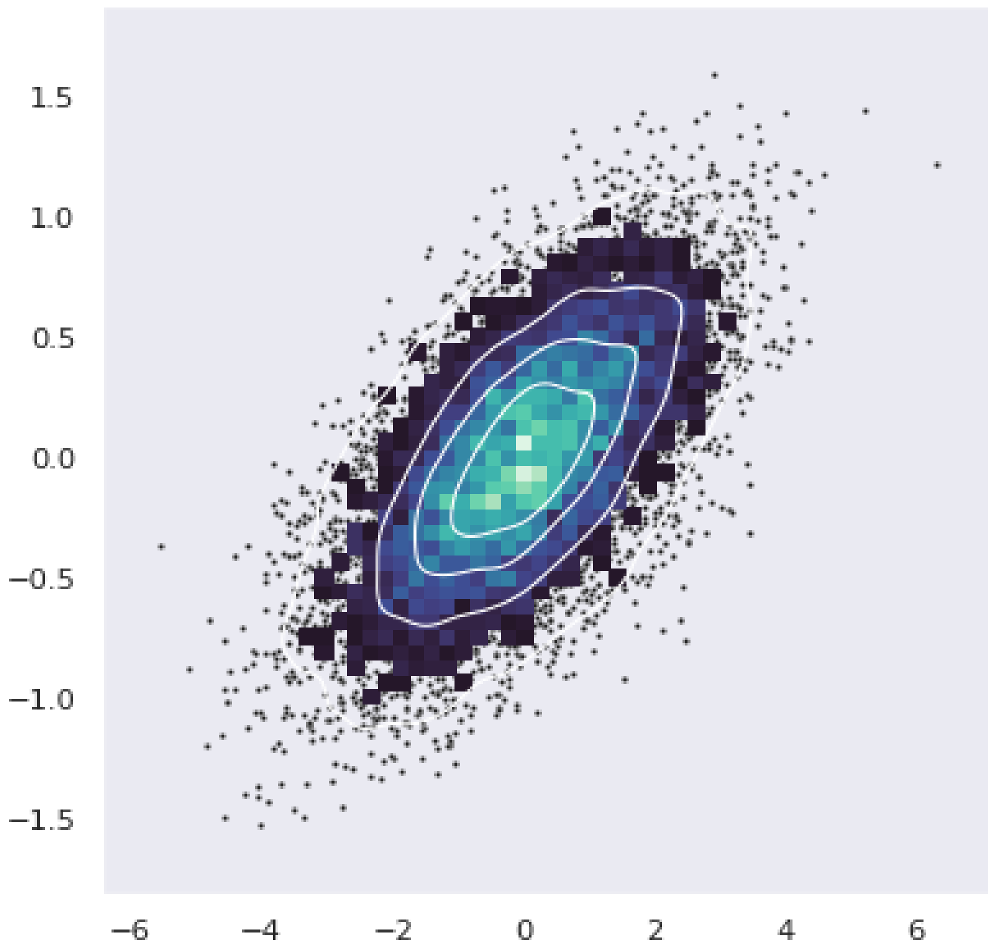
## 3.17 Plotting large distributions




**seaborn components used:** *set_theme()*, *load_dataset()*, *boxenplot()*

```
import seaborn as sns
sns.set_theme(style="whitegrid")

diamonds = sns.load_dataset("diamonds")
clarity_ranking = ["I1", "SI2", "SI1", "VS2", "VS1", "VVS2", "VVS1", "IF"]

sns.boxenplot(x="clarity", y="carat",
              color="b", order=clarity_ranking,
              scale="linear", data=diamonds)
```

## 3.18 Bivariate plot with multiple elements




**seaborn components used:** *set_theme()*, *scatterplot()*, *histplot()*, *kdeplot()*

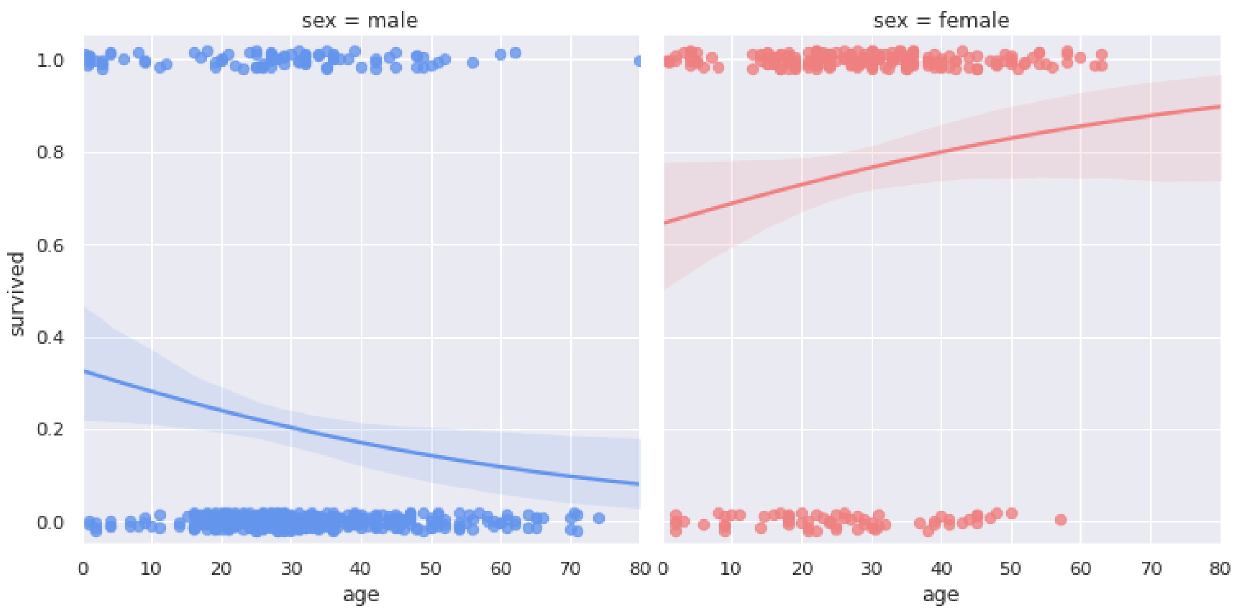
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="dark")

# Simulate data from a bivariate Gaussian
```

(continues on next page)

(continued from previous page)

```
n = 10000
mean = [0, 0]
cov = [(2, .4), (.4, .2)]
rng = np.random.RandomState(0)
x, y = rng.multivariate_normal(mean, cov, n).T

# Draw a combo histogram and scatterplot with density contours
f, ax = plt.subplots(figsize=(6, 6))
sns.scatterplot(x=x, y=y, s=5, color=".15")
sns.histplot(x=x, y=y, bins=50, pthresh=.1, cmap="mako")
sns.kdeplot(x=x, y=y, levels=5, color="w", linewidths=1)
```

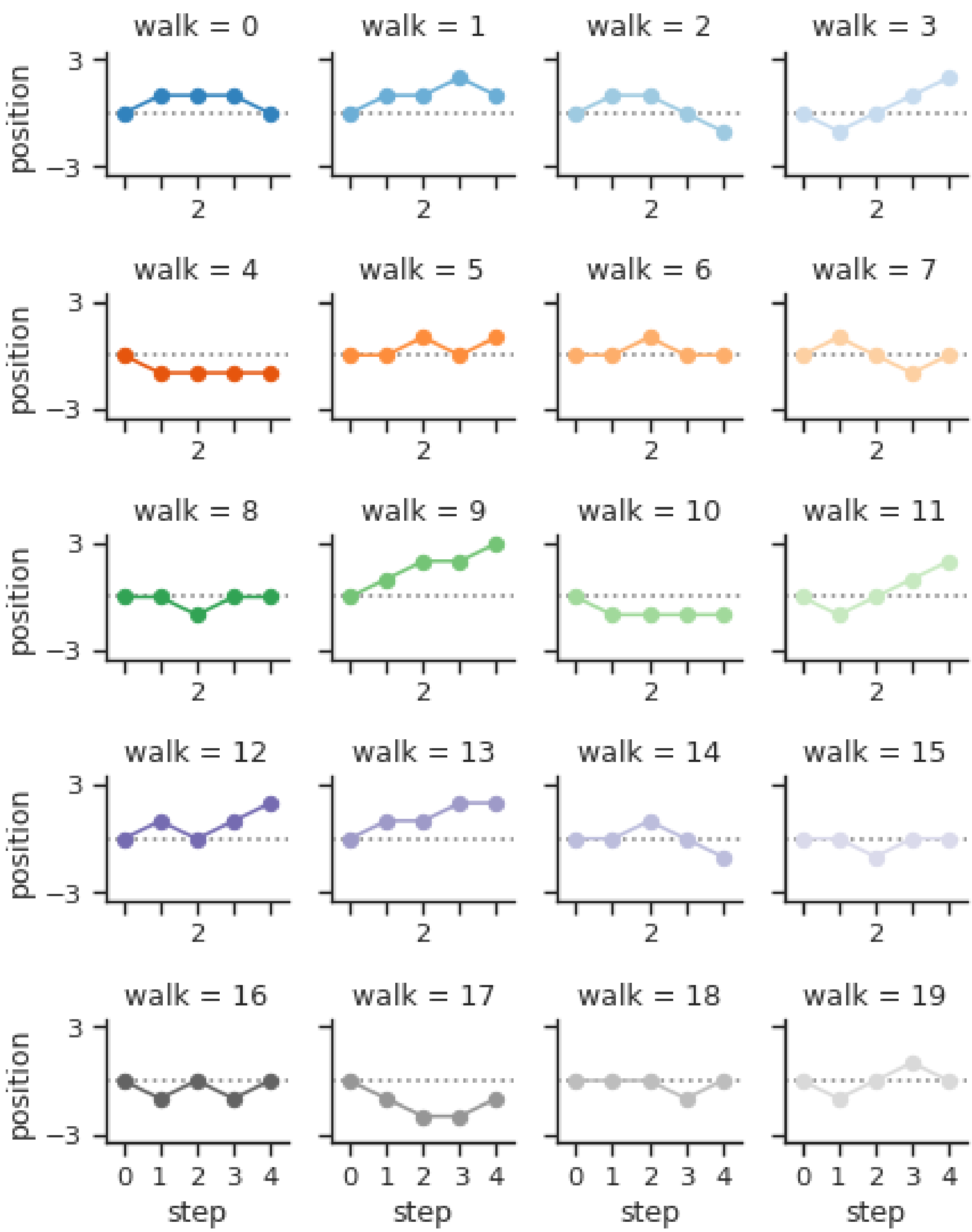
## 3.19 Faceted logistic regression




**seaborn components used:** *set_theme()*, *load_dataset()*, *lmplot()*

```
import seaborn as sns
sns.set_theme(style="darkgrid")

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

# Make a custom palette with gendered colors
pal = dict(male="#6495ED", female="#F08080")

# Show the survival probability as a function of age and sex
g = sns.lmplot(x="age", y="survived", col="sex", hue="sex", data=df,
               palette=pal, y_jitter=.02, logistic=True, truncate=False)
g.set(xlim=(0, 80), ylim=(-.05, 1.05))
```

## 3.20 Plotting on a large number of facets




**seaborn components used:** *set_theme()*, *FacetGrid*

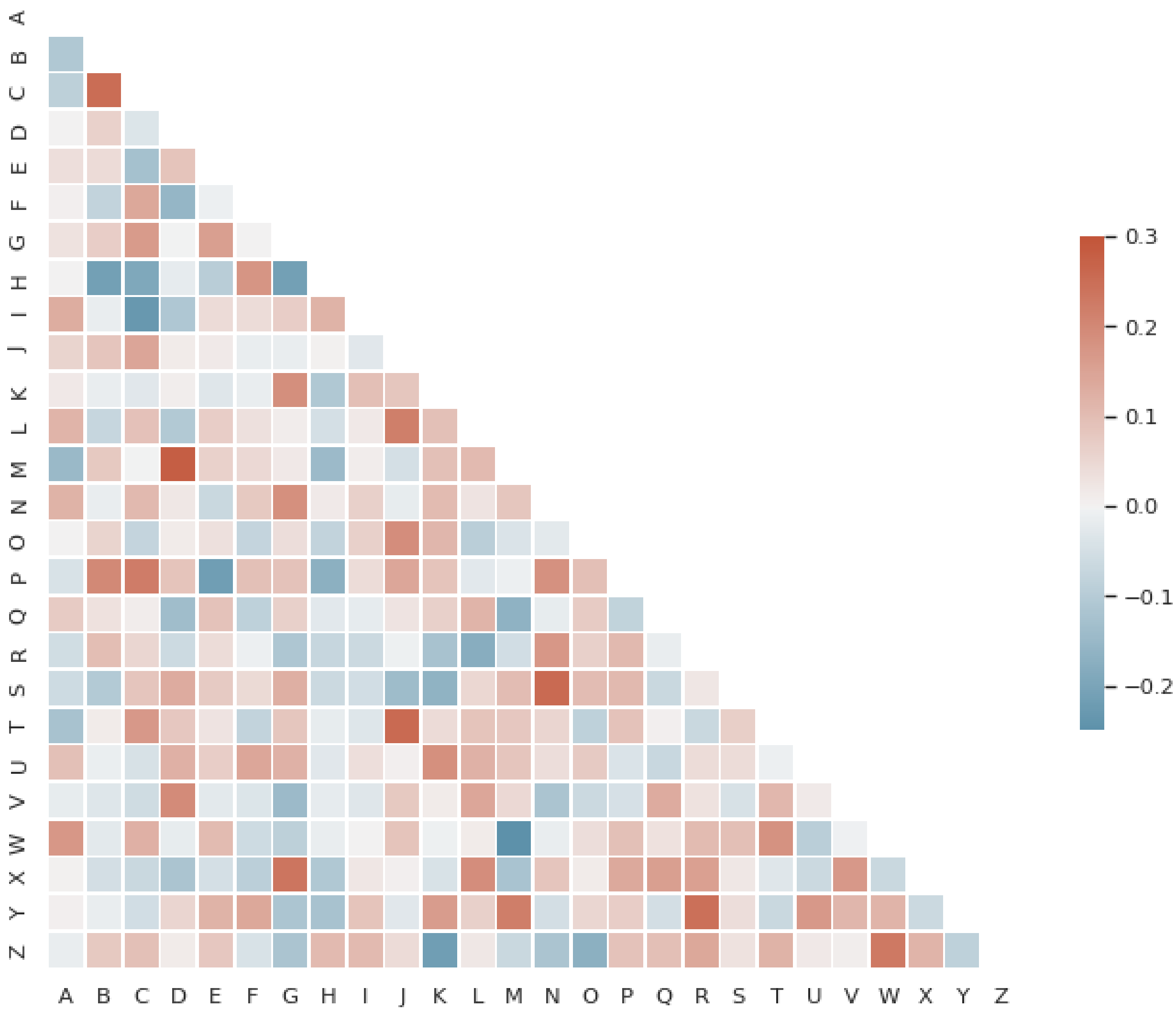
```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
sns.set_theme(style="ticks")

# Create a dataset with many short random walks
rs = np.random.RandomState(4)
pos = rs.randint(-1, 2, (20, 5)).cumsum(axis=1)
pos -= pos[:, 0, np.newaxis]
step = np.tile(range(5), 20)
walk = np.repeat(range(20), 5)
df = pd.DataFrame(np.c_[pos.flat, step, walk],
                  columns=["position", "step", "walk"])

# Initialize a grid of plots with an Axes for each walk
grid = sns.FacetGrid(df, col="walk", hue="walk", palette="tab20c",
                     col_wrap=4, height=1.5)

# Draw a horizontal line to show the starting point
grid.map(plt.axhline, y=0, ls=":", c=".5")

# Draw a line plot to show the trajectory of each random walk
grid.map(plt.plot, "step", "position", marker="o")

# Adjust the tick positions and labels
grid.set(xticks=np.arange(5), yticks=[-3, 3],
         xlim=(-.5, 4.5), ylim=(-3.5, 3.5))

# Adjust the arrangement of the plots
grid.fig.tight_layout(w_pad=1)
```

## 3.21 Plotting a diagonal correlation matrix




**seaborn components used:** *set_theme()*, *diverging_palette()*, *heatmap()*

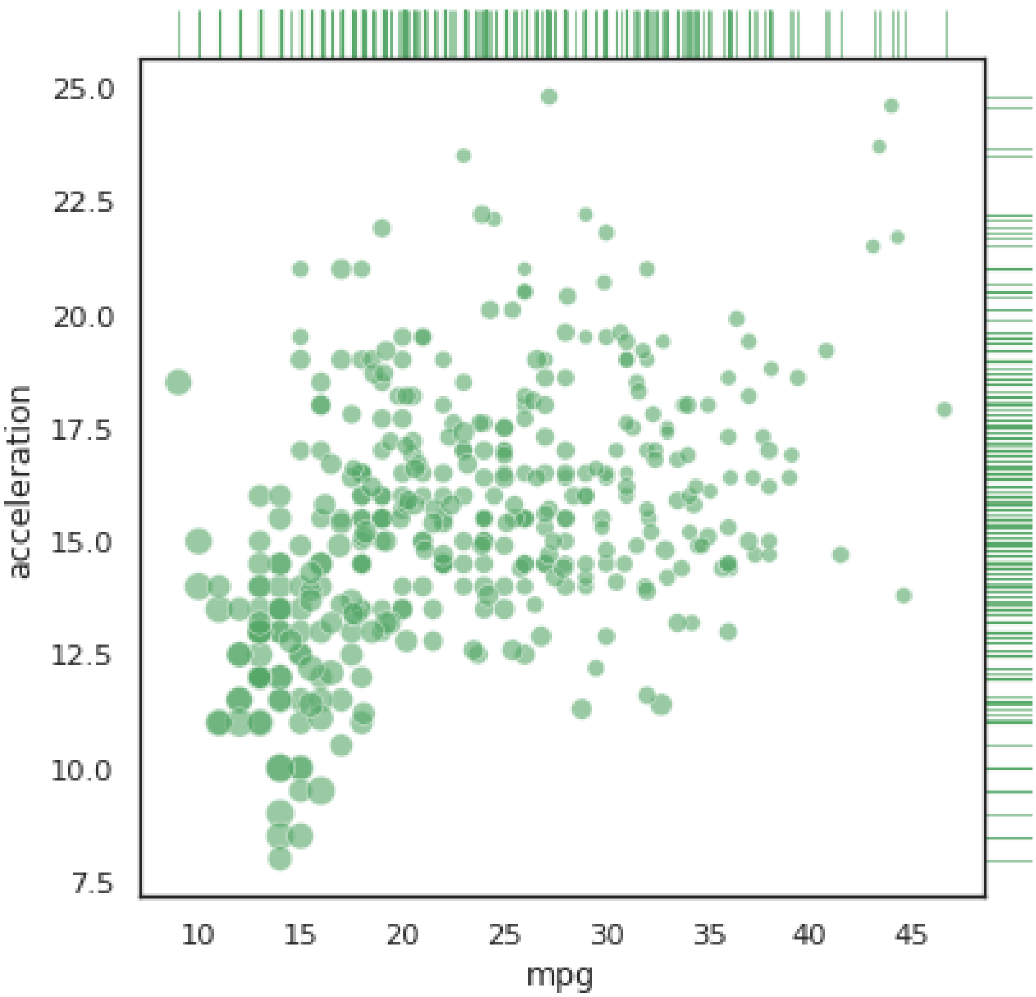
```
from string import ascii_letters
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="white")

# Generate a large random dataset
rs = np.random.RandomState(33)
d = pd.DataFrame(data=rs.normal(size=(100, 26)),
                 columns=list(ascii_letters[26:]))

# Compute the correlation matrix
corr = d.corr()
```

(continues on next page)

(continued from previous page)

```
# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 9))

# Generate a custom diverging colormap
cmap = sns.diverging_palette(230, 20, as_cmap=True)

# Draw the heatmap with the mask and correct aspect ratio
sns.heatmap(corr, mask=mask, cmap=cmap, vmax=.3, center=0,
            square=True, linewidths=.5, cbar_kws={"shrink": .5})
```

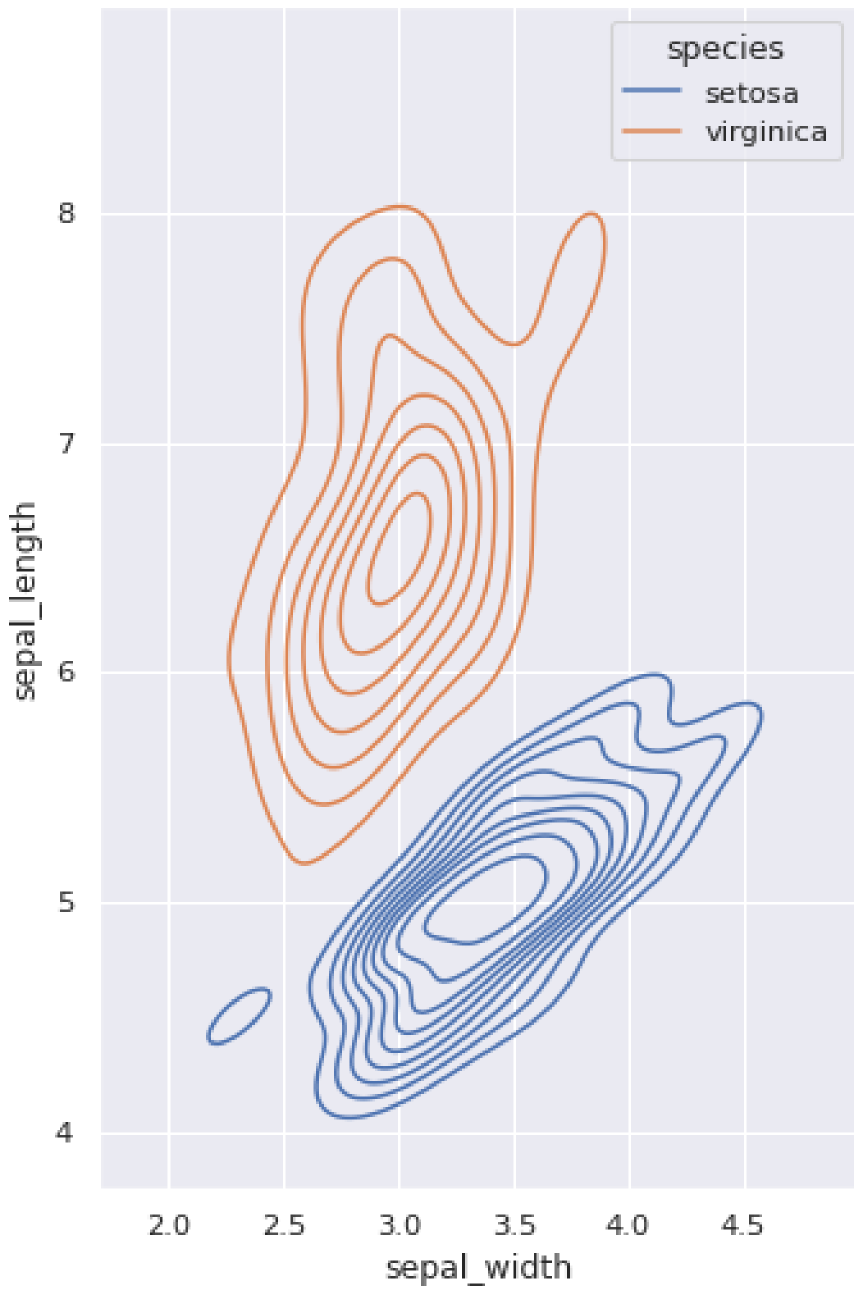
## 3.22 Scatterplot with marginal ticks




**seaborn components used:** *set_theme()*, *load_dataset()*, *JointGrid*

```
import seaborn as sns
sns.set_theme(style="white", color_codes=True)
mpg = sns.load_dataset("mpg")

# Use JointGrid directly to draw a custom plot
g = sns.JointGrid(data=mpg, x="mpg", y="acceleration", space=0, ratio=17)
g.plot_joint(sns.scatterplot, size=mpg["horsepower"], sizes=(30, 120),
             color="g", alpha=.6, legend=False)
g.plot_marginals(sns.rugplot, height=1, color="g", alpha=.6)
```

## 3.23 Multiple bivariate KDE plots




**seaborn components used:** *set_theme()*, *load_dataset()*, *kdeplot()*

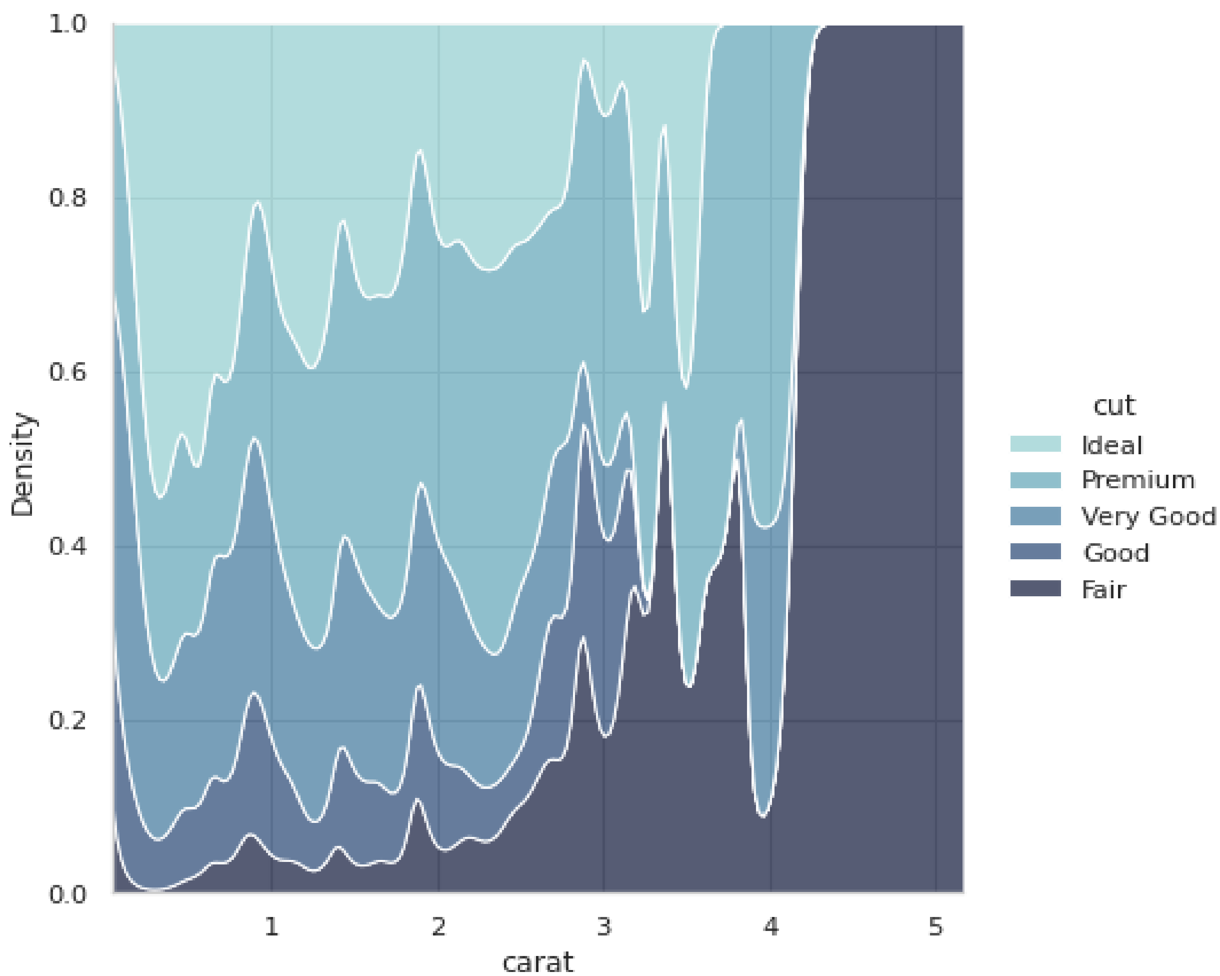
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="darkgrid")
iris = sns.load_dataset("iris")

# Set up the figure
```

(continues on next page)

(continued from previous page)

```
f, ax = plt.subplots(figsize=(8, 8))
ax.set_aspect("equal")

# Draw a contour plot to represent each bivariate density
sns.kdeplot(
    data=iris.query("species != 'versicolor'"),
    x="sepal_width",
    y="sepal_length",
    hue="species",
    thresh=.1,
)
```

## 3.24 Conditional kernel density estimate




**seaborn components used:** *set_theme()*, *load_dataset()*, *displot()*

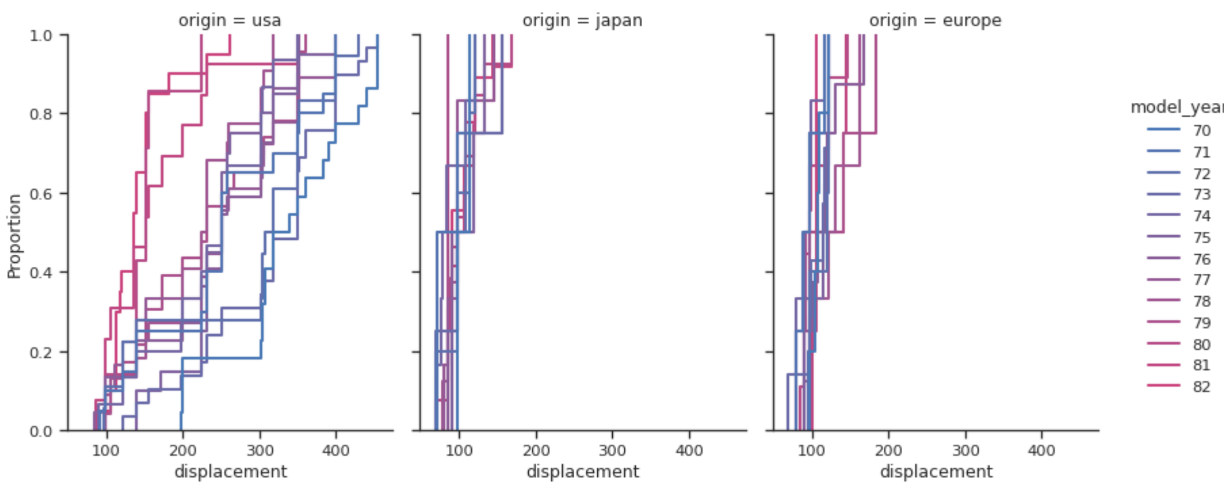
```
import seaborn as sns
sns.set_theme(style="whitegrid")
```

(continues on next page)

(continued from previous page)

```
# Load the diamonds dataset
diamonds = sns.load_dataset("diamonds")

# Plot the distribution of clarity ratings, conditional on carat
sns.displot(
    data=diamonds,
    x="carat", hue="cut",
    kind="kde", height=6,
    multiple="fill", clip=(0, None),
    palette="ch:rot=-.25,hue=1,light=.75",
)
```

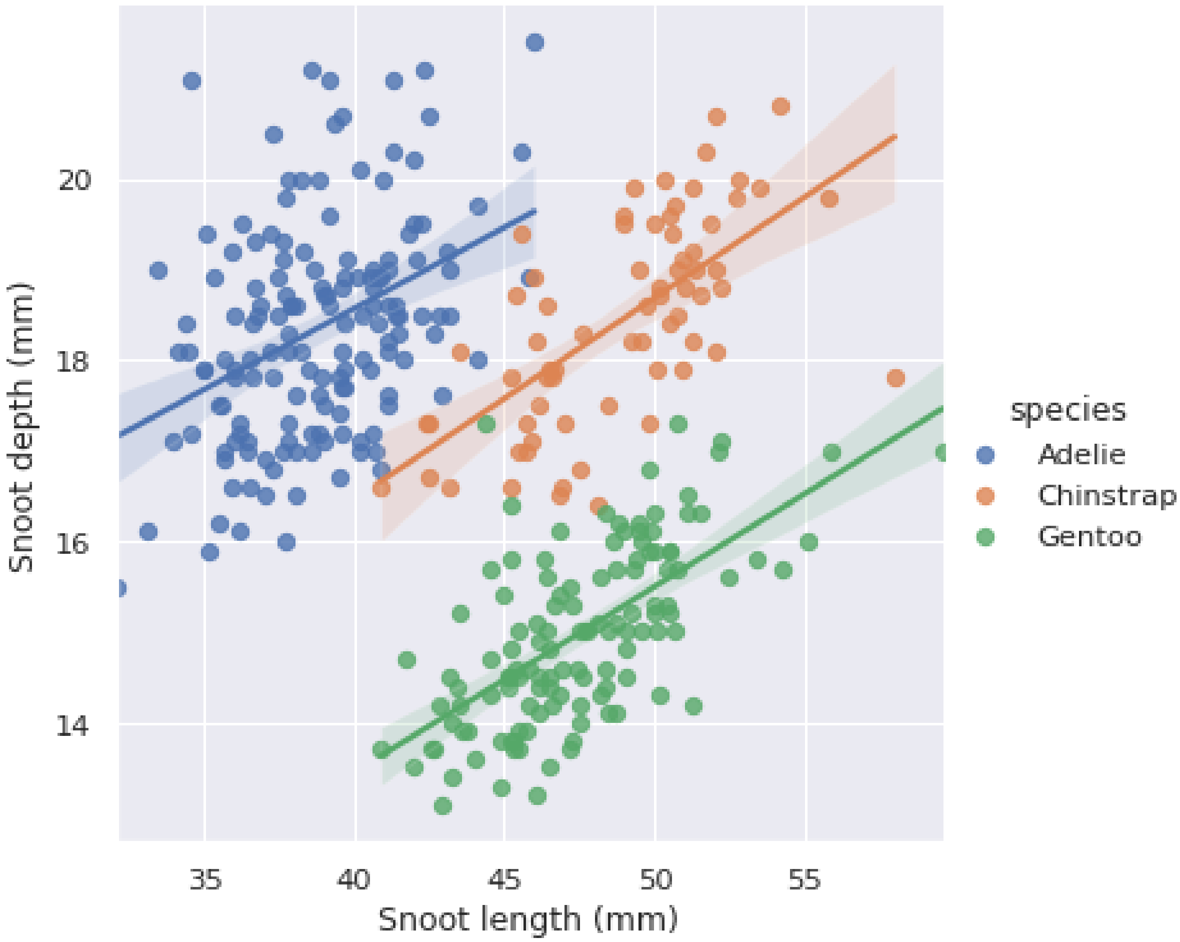
## 3.25 Facetted ECDF plots




**seaborn components used:** *set_theme()*, *load_dataset()*, *blend_palette()*, *displot()*

```
import seaborn as sns
sns.set_theme(style="ticks")
mpg = sns.load_dataset("mpg")

colors = (250, 70, 50), (350, 70, 50)
cmap = sns.blend_palette(colors, input="husl", as_cmap=True)
sns.displot(
    mpg,
    x="displacement", col="origin", hue="model_year",
    kind="ecdf", aspect=.75, linewidth=2, palette=cmap,
)
```

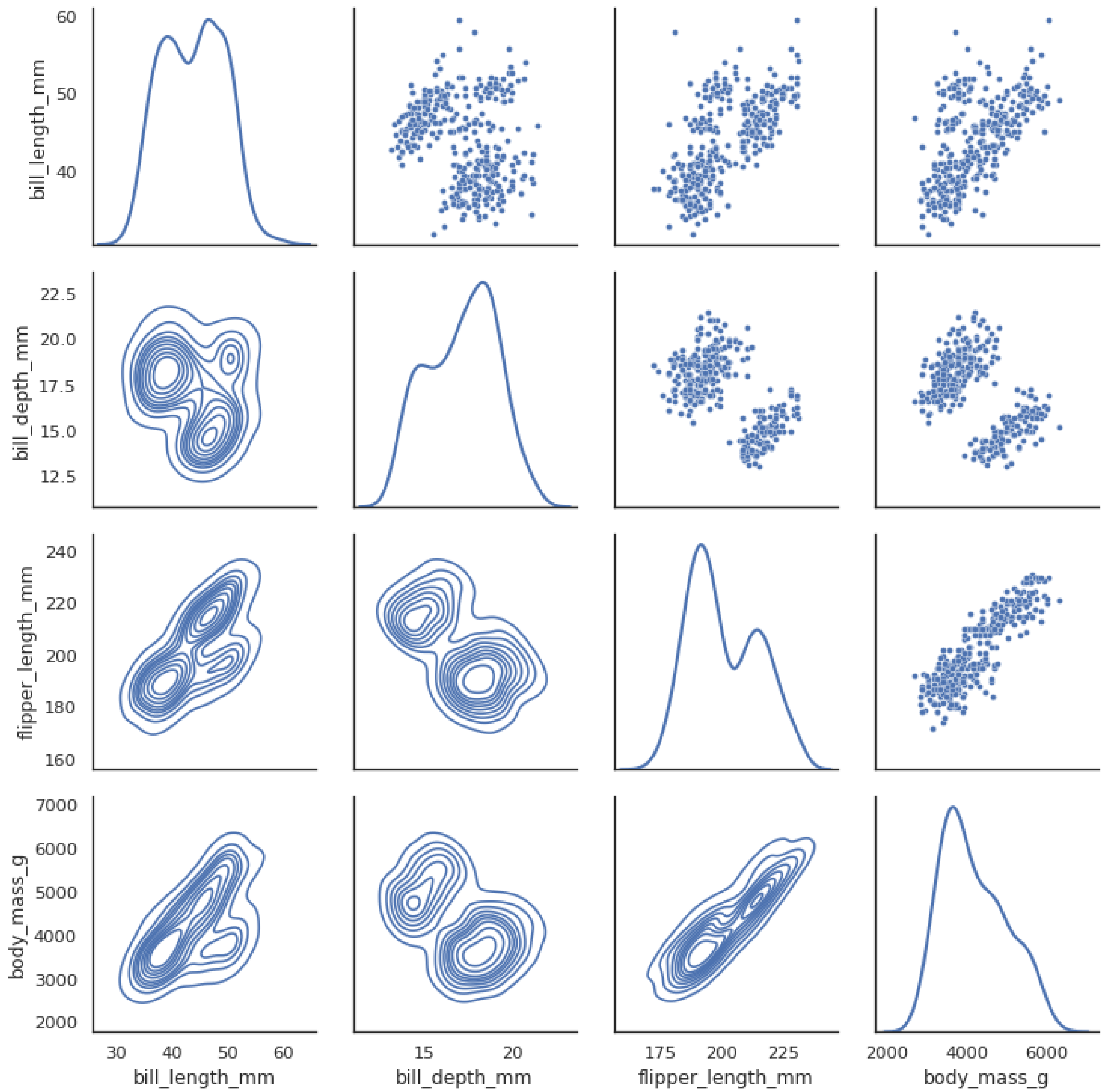
## 3.26 Multiple linear regression




**seaborn components used:** *set_theme()*, *load_dataset()*, *lmplot()*

```
import seaborn as sns
sns.set_theme()

# Load the penguins dataset
penguins = sns.load_dataset("penguins")

# Plot sepal width as a function of sepal_length across days
g = sns.lmplot(
    data=penguins,
    x="bill_length_mm", y="bill_depth_mm", hue="species",
    height=5
)

# Use more informative axis labels than are provided by default
g.set_axis_labels("Snoot length (mm)", "Snoot depth (mm)")
```

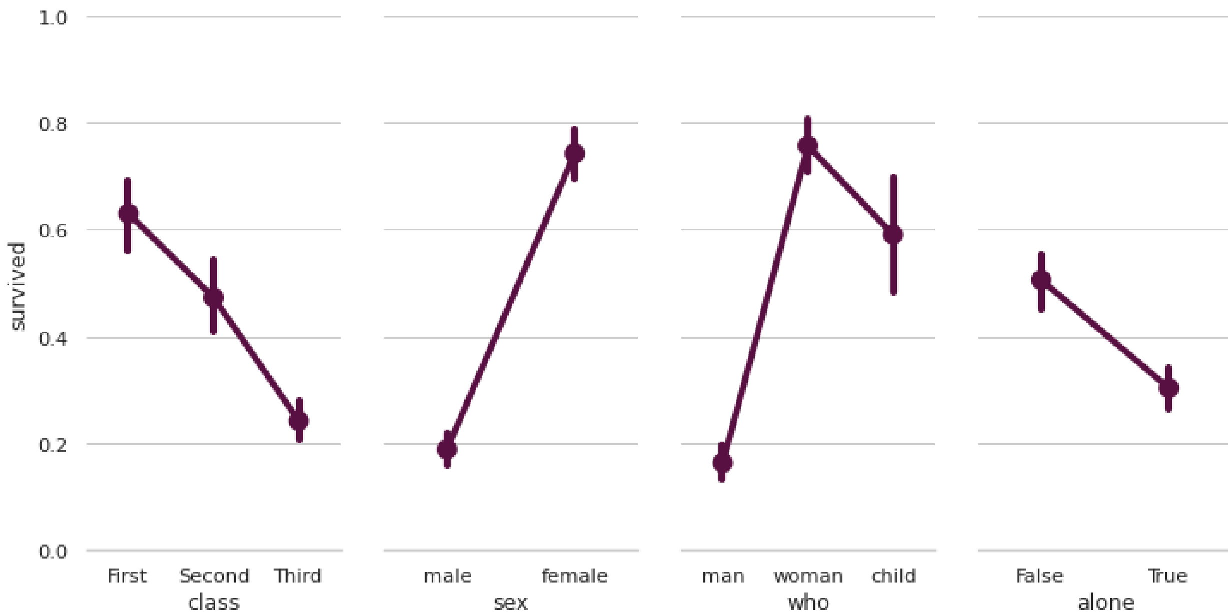
## 3.27 Paired density and scatterplot matrix




**seaborn components used:** `set_theme()`, `load_dataset()`, `PairGrid`

```python
import seaborn as sns
sns.set_theme(style="white")

df = sns.load_dataset("penguins")

g = sns.PairGrid(df, diag_sharey=False)
g.map_upper(sns.scatterplot, s=15)
g.map_lower(sns.kdeplot)
g.map_diag(sns.kdeplot, lw=2)
```

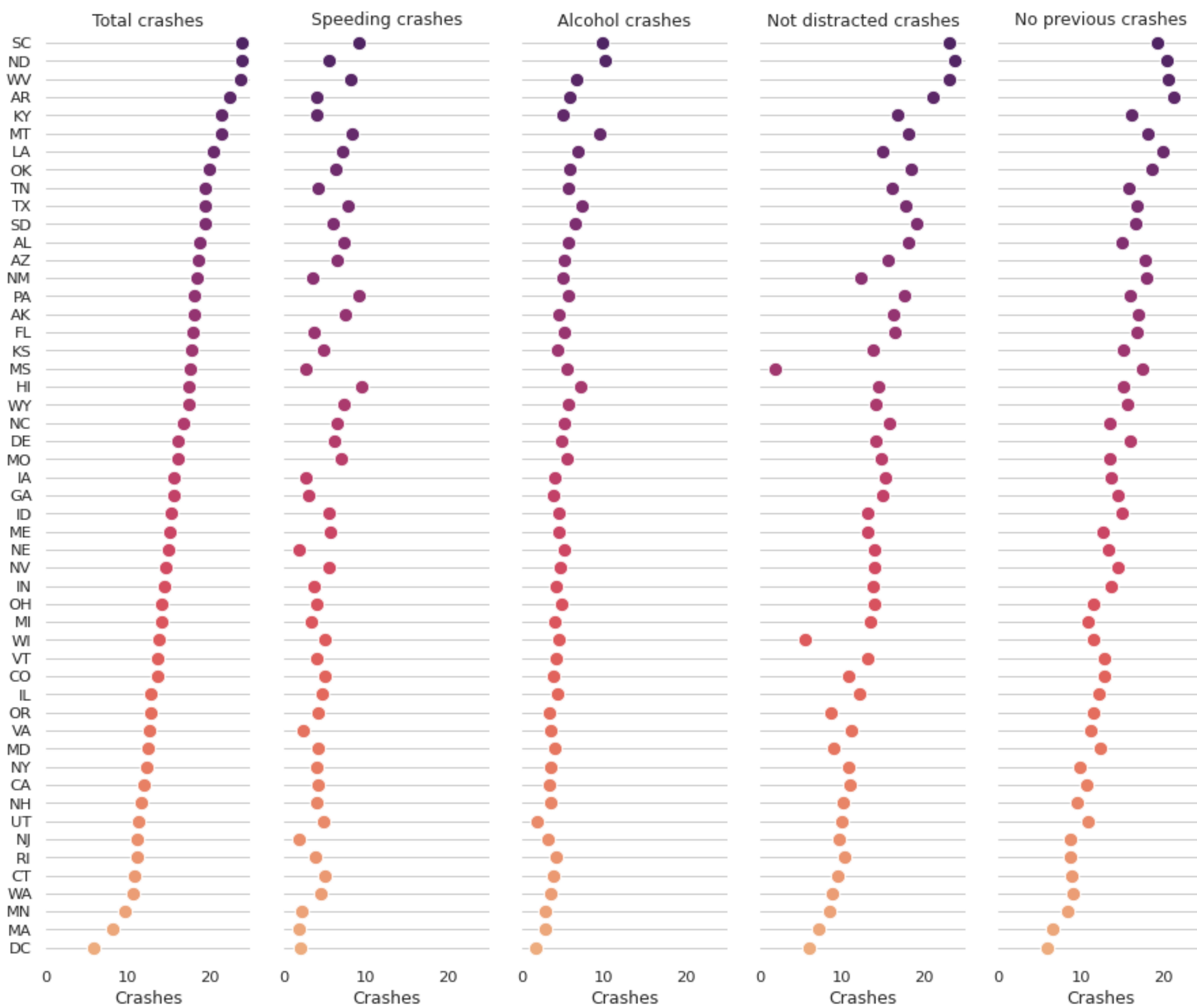
## 3.28 Paired categorical plots




**seaborn components used:** *set_theme()*, *load_dataset()*, *PairGrid*, *despine()*

```
import seaborn as sns
sns.set_theme(style="whitegrid")

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

# Set up a grid to plot survival probability against several variables
g = sns.PairGrid(titanic, y_vars="survived",
                 x_vars=["class", "sex", "who", "alone"],
                 height=5, aspect=.5)

# Draw a seaborn pointplot onto each Axes
g.map(sns.pointplot, scale=1.3, errwidth=4, color="xkcd:plum")
g.set(ylim=(0, 1))
sns.despine(fig=g.fig, left=True)
```

## 3.29 Dot plot with several variables




**seaborn components used:** `set_theme()`, `load_dataset()`, `PairGrid`, `despine()`

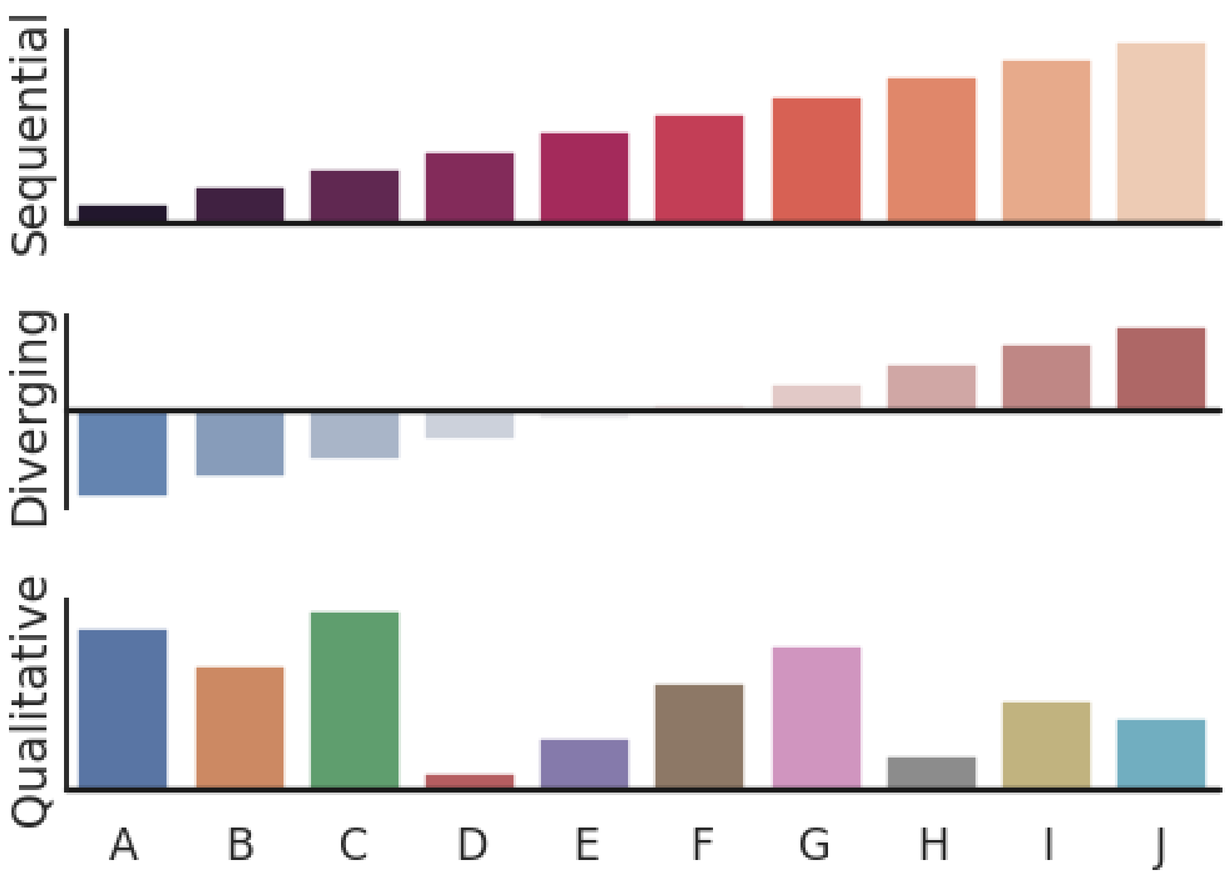
```
import seaborn as sns
sns.set_theme(style="whitegrid")

# Load the dataset
crashes = sns.load_dataset("car_crashes")

# Make the PairGrid
g = sns.PairGrid(crashes.sort_values("total", ascending=False),
                 x_vars=crashes.columns[:-3], y_vars=["abbrev"],
                 height=10, aspect=.25)

# Draw a dot plot using the stripplot function
g.map(sns.stripplot, size=10, orient="h", jitter=False,
      palette="flare_r", linewidth=1, edgecolor="w")

# Use the same x axis limits on all columns and add better labels
```

(continues on next page)

(continued from previous page)

```
g.set(xlim=(0, 25), xlabel="Crashes", ylabel="")

# Use semantically meaningful titles for the columns
titles = ["Total crashes", "Speeding crashes", "Alcohol crashes",
          "Not distracted crashes", "No previous crashes"]

for ax, title in zip(g.axes.flat, titles):

    # Set a different title for each axes
    ax.set(title=title)

    # Make the grid horizontal instead of vertical
    ax.xaxis.grid(False)
    ax.yaxis.grid(True)

sns.despine(left=True, bottom=True)
```

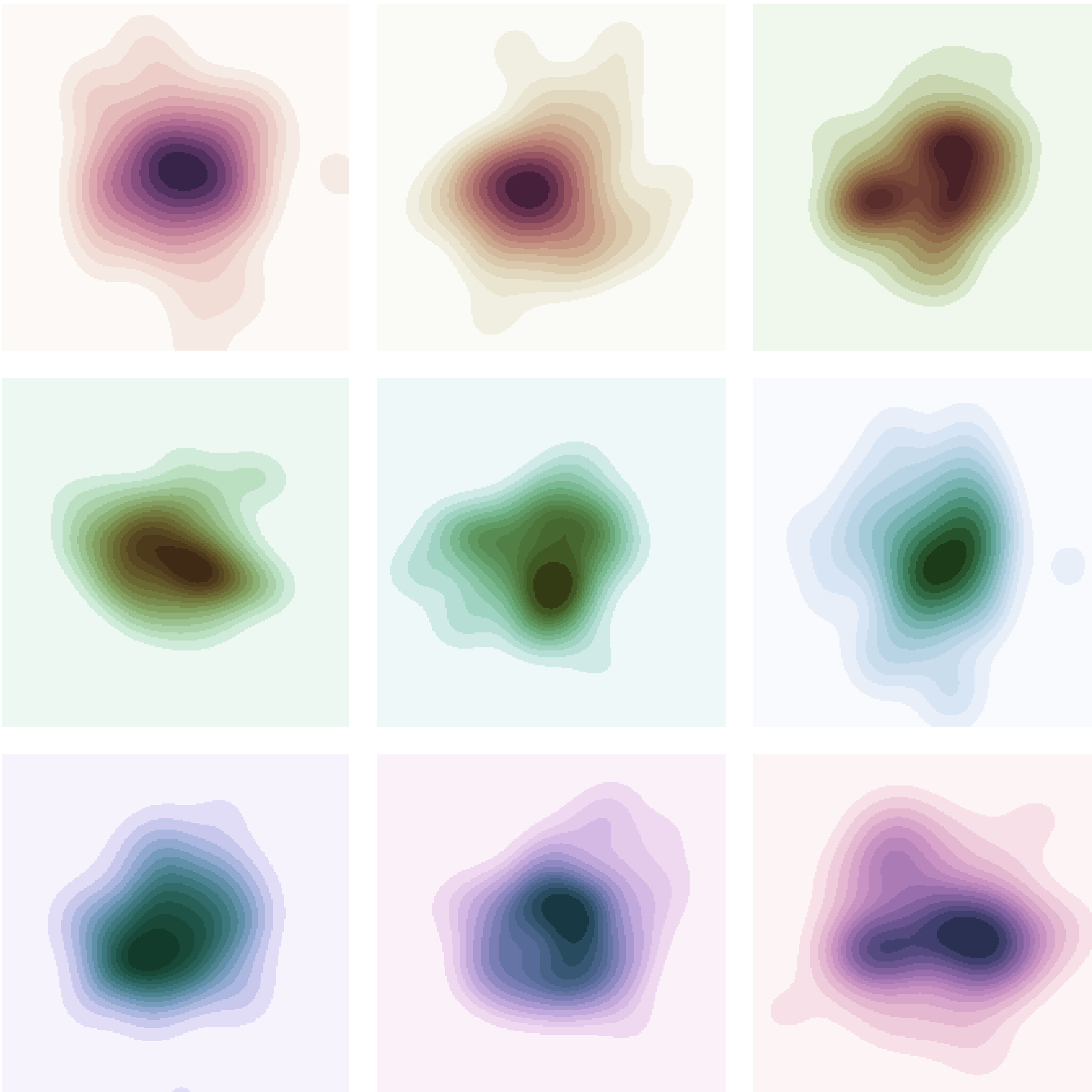
## 3.30 Color palette choices




**seaborn components used:** *set_theme()*, *barplot()*, *barplot()*, *barplot()*, *despine()*

```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="white", context="talk")
rs = np.random.RandomState(8)

# Set up the matplotlib figure
f, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(7, 5), sharex=True)

# Generate some sequential data
x = np.array(list("ABCDEFGHIJ"))
y1 = np.arange(1, 11)
sns.barplot(x=x, y=y1, palette="rocket", ax=ax1)
ax1.axhline(0, color="k", clip_on=False)
ax1.set_ylabel("Sequential")

# Center the data to make it diverging
y2 = y1 - 5.5
sns.barplot(x=x, y=y2, palette="vlag", ax=ax2)
ax2.axhline(0, color="k", clip_on=False)
ax2.set_ylabel("Diverging")

# Randomly reorder the data to make it qualitative
y3 = rs.choice(y1, len(y1), replace=False)
sns.barplot(x=x, y=y3, palette="deep", ax=ax3)
ax3.axhline(0, color="k", clip_on=False)
ax3.set_ylabel("Qualitative")

# Finalize the plot
sns.despine(bottom=True)
plt.setp(f.axes, yticks=[])
plt.tight_layout(h_pad=2)
```

## 3.31 Different cubehelix palettes



**seaborn components used:** *set_theme()*, *cubehelix_palette()*, *kdeplot()*

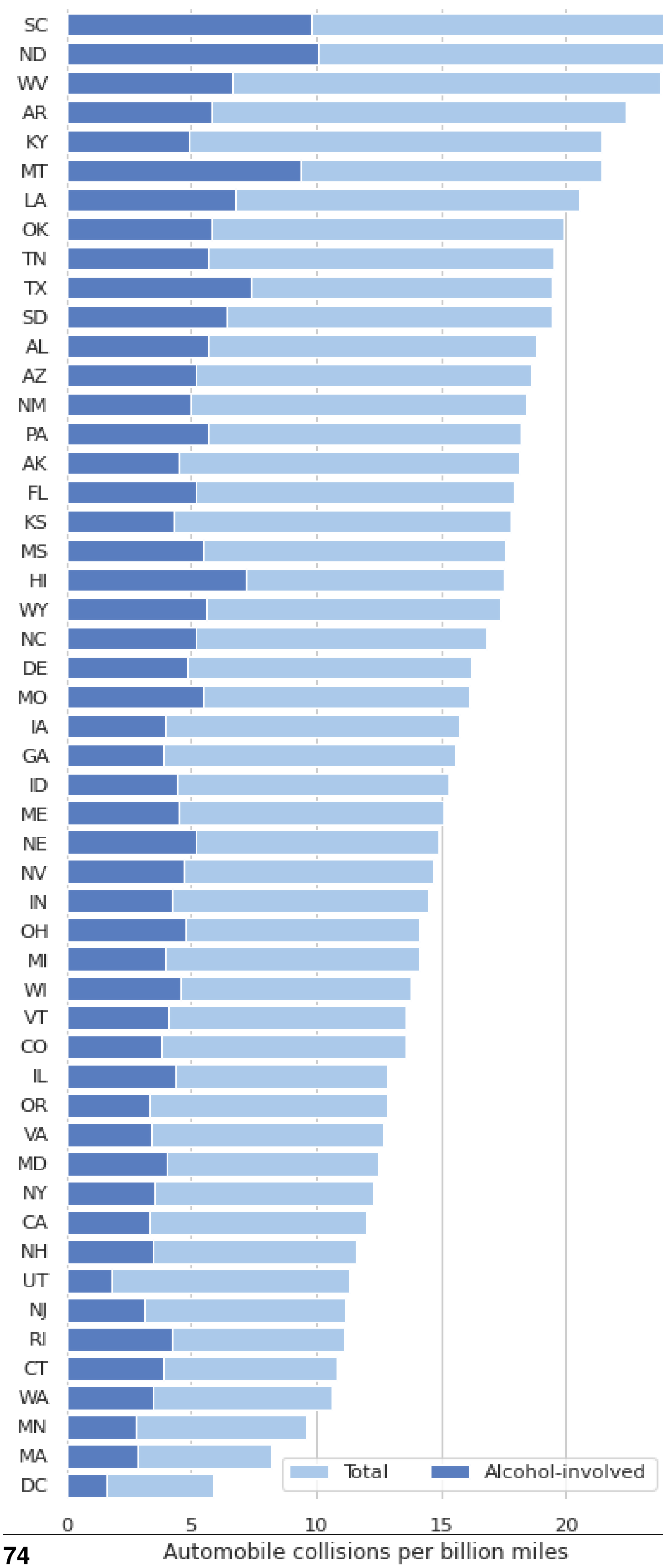
```
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_theme(style="white")
rs = np.random.RandomState(50)

# Set up the matplotlib figure
f, axes = plt.subplots(3, 3, figsize=(9, 9), sharex=True, sharey=True)
```

(continues on next page)

(continued from previous page)

```
# Rotate the starting point around the cubehelix hue circle
for ax, s in zip(axes.flat, np.linspace(0, 3, 10)):

    # Create a cubehelix colormap to use with kdeplot
    cmap = sns.cubehelix_palette(start=s, light=1, as_cmap=True)

    # Generate and plot a random bivariate dataset
    x, y = rs.normal(size=(2, 50))
    sns.kdeplot(
        x=x, y=y,
        cmap=cmap, fill=True,
        clip=(-5, 5), cut=10,
        thresh=0, levels=15,
        ax=ax,
    )
    ax.set_axis_off()

ax.set(xlim=(-3.5, 3.5), ylim=(-3.5, 3.5))
f.subplots_adjust(0, 0, 1, 1, .08, .08)
```

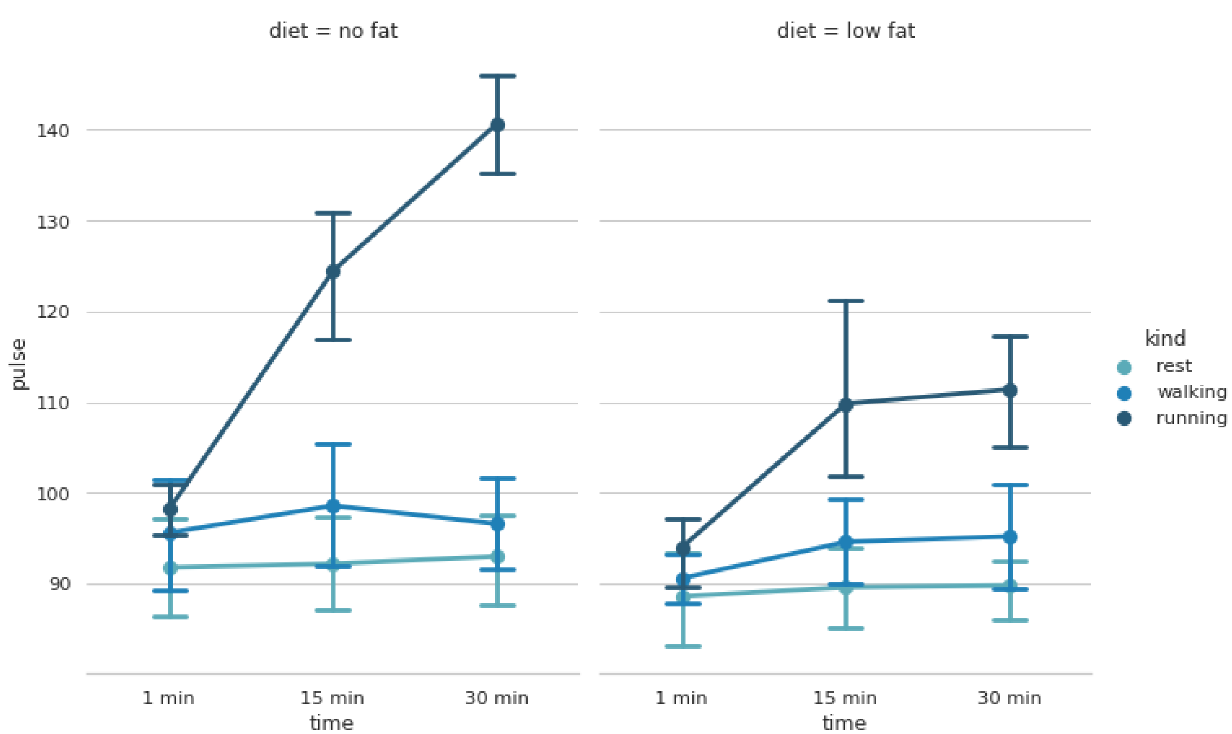
## 3.32 Horizontal bar plots



**seaborn components used:** *set_theme()*, *load_dataset()*, *set_color_codes()*, *barplot()*, *set_color_codes()*, *barplot()*, *despine()*

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="whitegrid")

# Initialize the matplotlib figure
f, ax = plt.subplots(figsize=(6, 15))

# Load the example car crash dataset
crashes = sns.load_dataset("car_crashes").sort_values("total", ascending=False)

# Plot the total crashes
sns.set_color_codes("pastel")
sns.barplot(x="total", y="abbrev", data=crashes,
            label="Total", color="b")

# Plot the crashes where alcohol was involved
sns.set_color_codes("muted")
sns.barplot(x="alcohol", y="abbrev", data=crashes,
            label="Alcohol-involved", color="b")

# Add a legend and informative axis label
ax.legend(ncol=2, loc="lower right", frameon=True)
ax.set(xlim=(0, 24), ylabel="",
       xlabel="Automobile collisions per billion miles")
sns.despine(left=True, bottom=True)
```

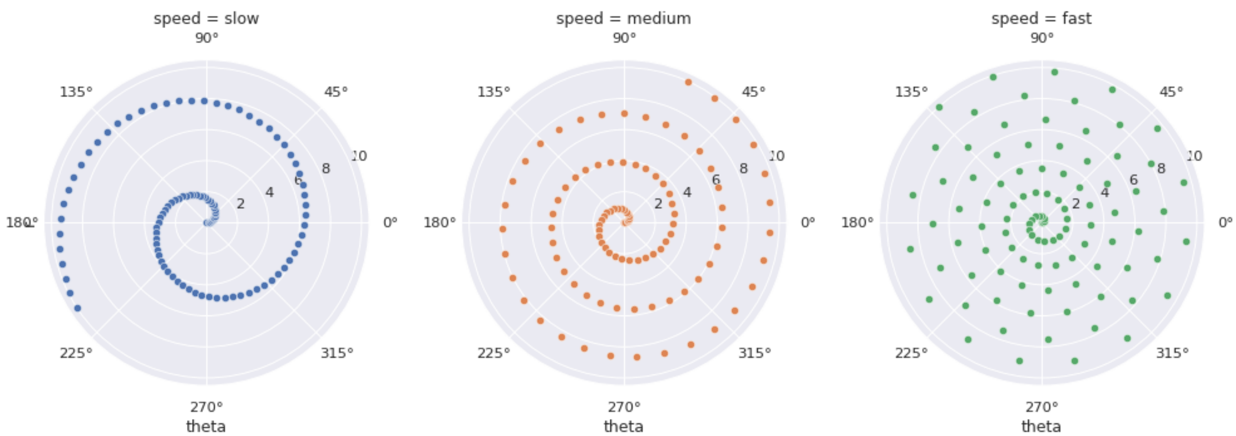
## 3.33 Plotting a three-way ANOVA



**seaborn components used:** *set_theme()*, *load_dataset()*, *catplot()*

```
import seaborn as sns
sns.set_theme(style="whitegrid")

# Load the example exercise dataset
df = sns.load_dataset("exercise")

# Draw a pointplot to show pulse as a function of three categorical factors
g = sns.catplot(x="time", y="pulse", hue="kind", col="diet",
                capsize=.2, palette="YlGnBu_d", height=6, aspect=.75,
                kind="point", data=df)
g.despine(left=True)
```

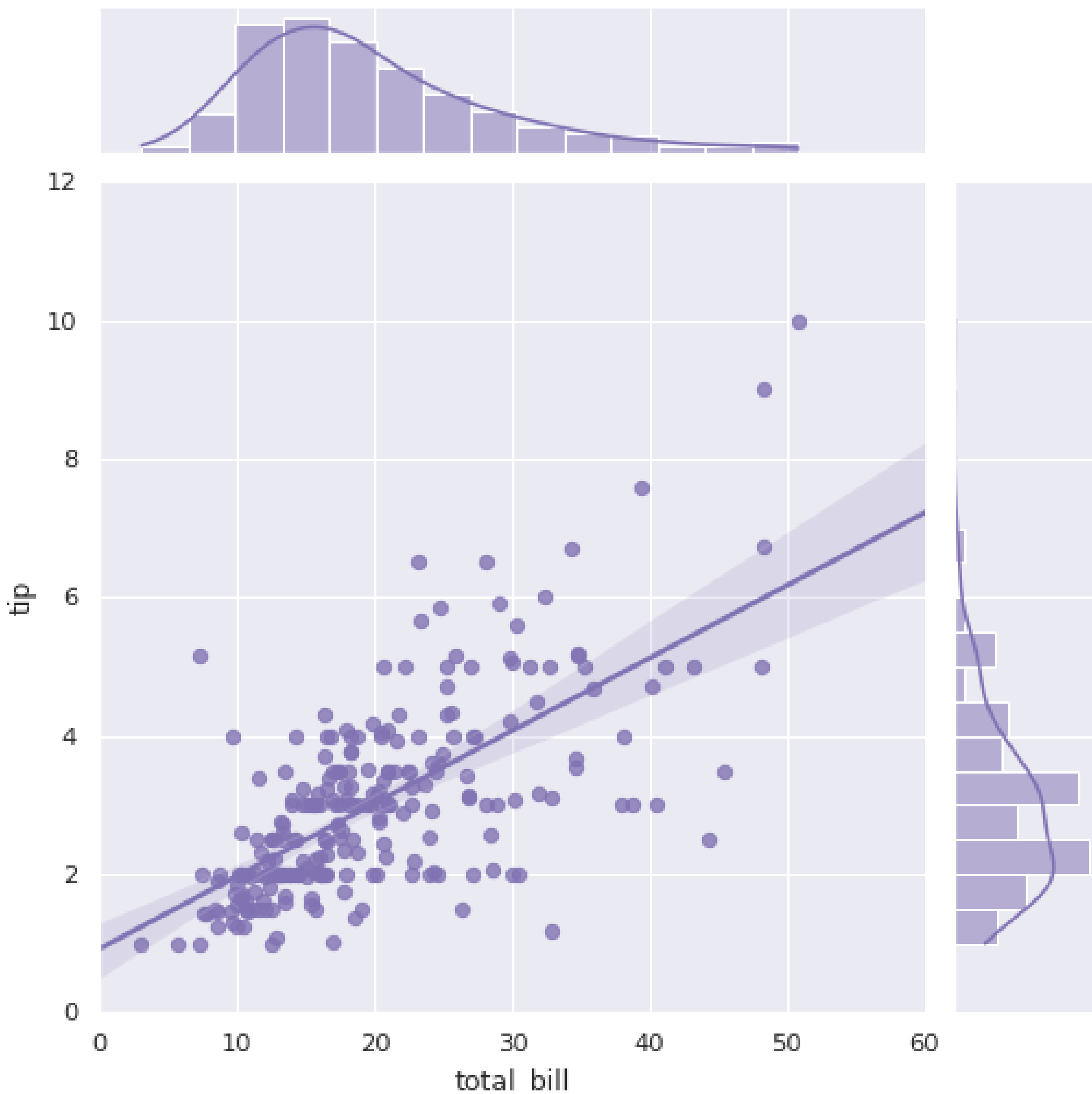
## 3.34 FacetGrid with custom projection




**seaborn components used:** *set_theme()*, *FacetGrid*

```
import numpy as np
import pandas as pd
import seaborn as sns

sns.set_theme()

# Generate an example radial datast
r = np.linspace(0, 10, num=100)
df = pd.DataFrame({'r': r, 'slow': r, 'medium': 2 * r, 'fast': 4 * r})

# Convert the dataframe to long-form or "tidy" format
df = pd.melt(df, id_vars=['r'], var_name='speed', value_name='theta')

# Set up a grid of axes with a polar projection
g = sns.FacetGrid(df, col="speed", hue="speed",
                  subplot_kws=dict(projection='polar'), height=4.5,
                  sharex=False, sharey=False, despine=False)

# Draw a scatterplot onto each axes in the grid
g.map(sns.scatterplot, "theta", "r")
```

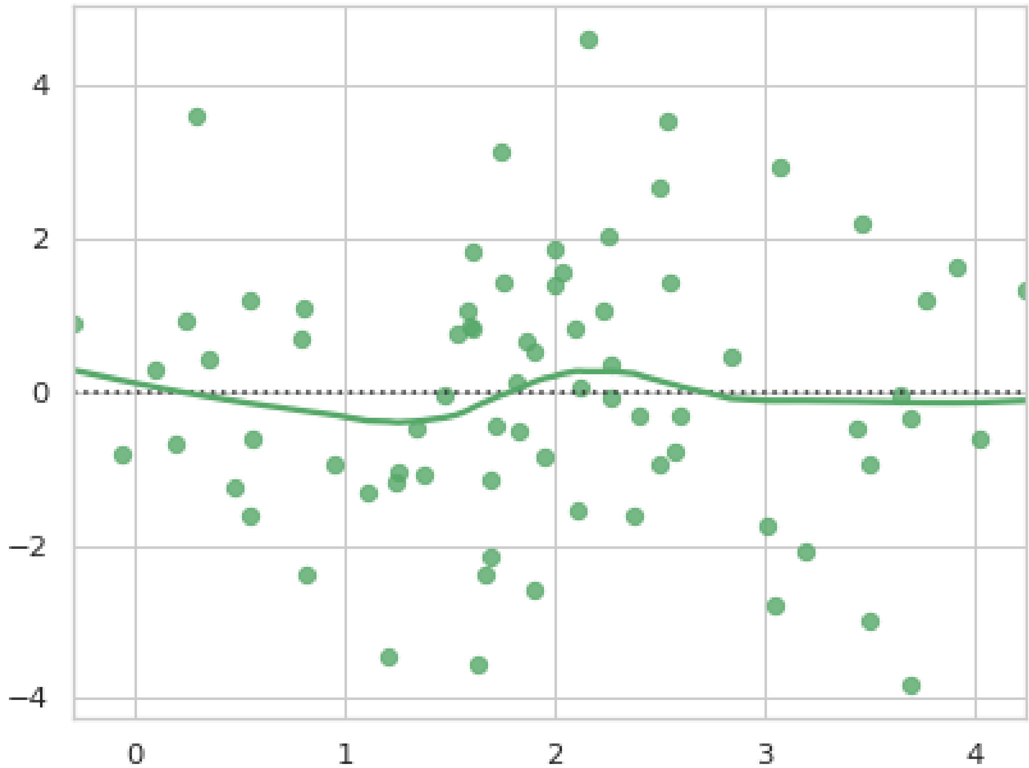
## 3.35 Linear regression with marginal distributions




**seaborn components used:** *set_theme()*, *load_dataset()*, *jointplot()*

```
import seaborn as sns
sns.set_theme(style="darkgrid")

tips = sns.load_dataset("tips")
g = sns.jointplot(x="total_bill", y="tip", data=tips,
                  kind="reg", truncate=False,
                  xlim=(0, 60), ylim=(0, 12),
                  color="m", height=7)
```

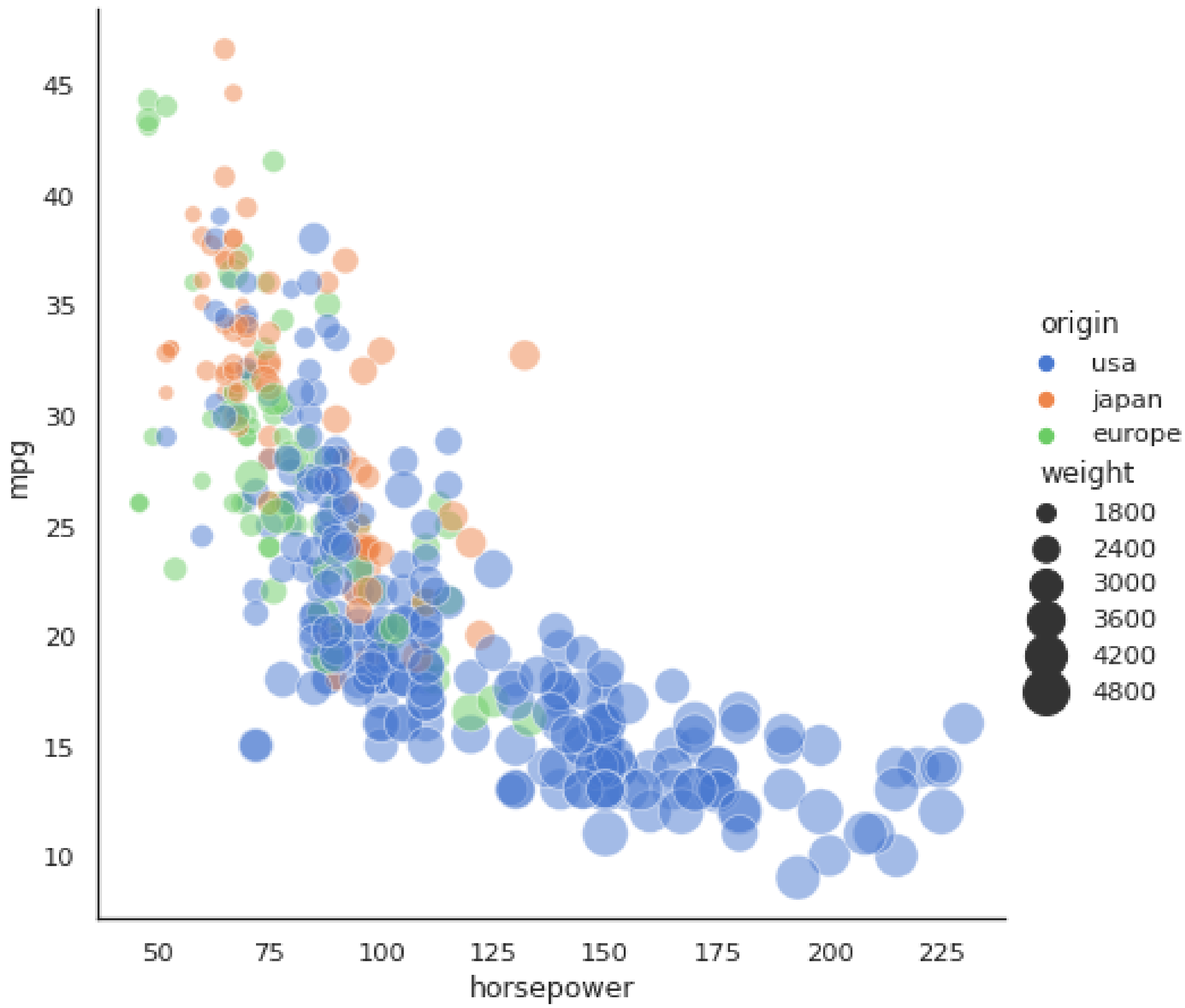
## 3.36 Plotting model residuals




**seaborn components used:** *set_theme()*, *residplot()*

```
import numpy as np
import seaborn as sns
sns.set_theme(style="whitegrid")

# Make an example dataset with y ~ x
rs = np.random.RandomState(7)
x = rs.normal(2, 1, 75)
y = 2 + 1.5 * x + rs.normal(0, 2, 75)

# Plot the residuals after fitting a linear model
sns.residplot(x=x, y=y, lowess=True, color="g")
```

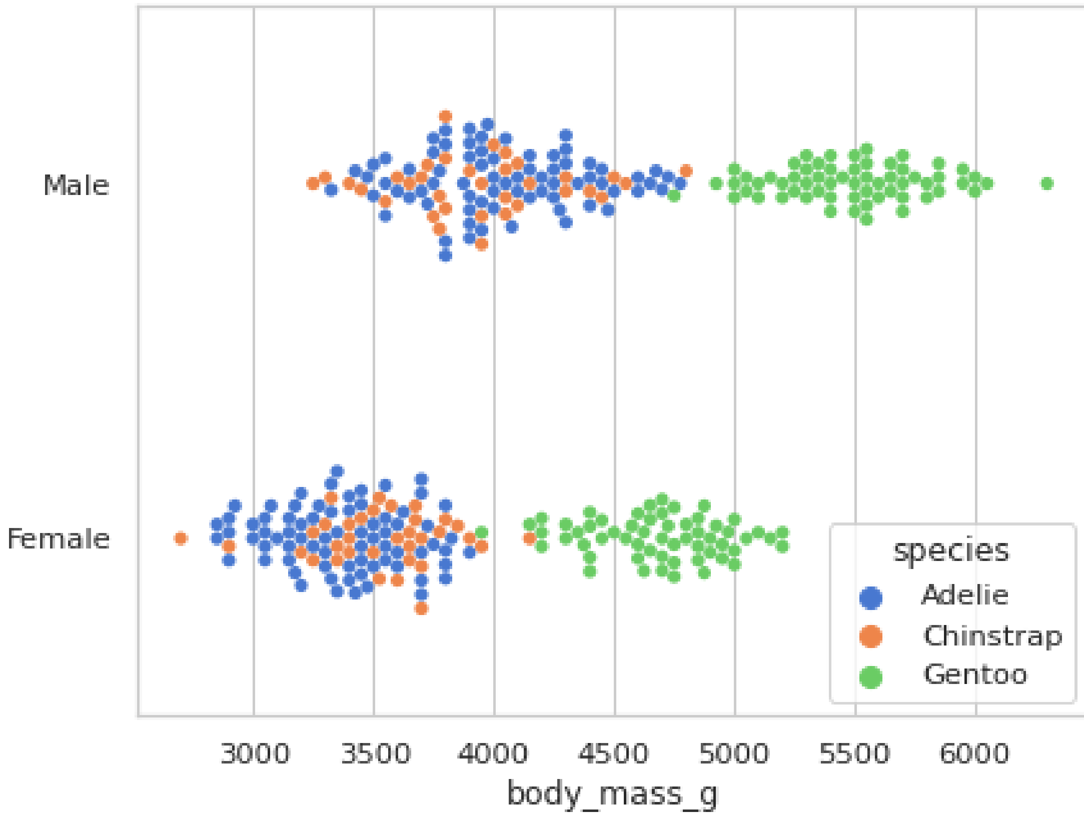
## 3.37 Scatterplot with varying point sizes and hues




**seaborn components used:** *set_theme()*, *load_dataset()*, *relplot()*

```
import seaborn as sns
sns.set_theme(style="white")

# Load the example mpg dataset
mpg = sns.load_dataset("mpg")

# Plot miles per gallon against horsepower with other semantics
sns.relplot(x="horsepower", y="mpg", hue="origin", size="weight",
            sizes=(40, 400), alpha=.5, palette="muted",
            height=6, data=mpg)
```

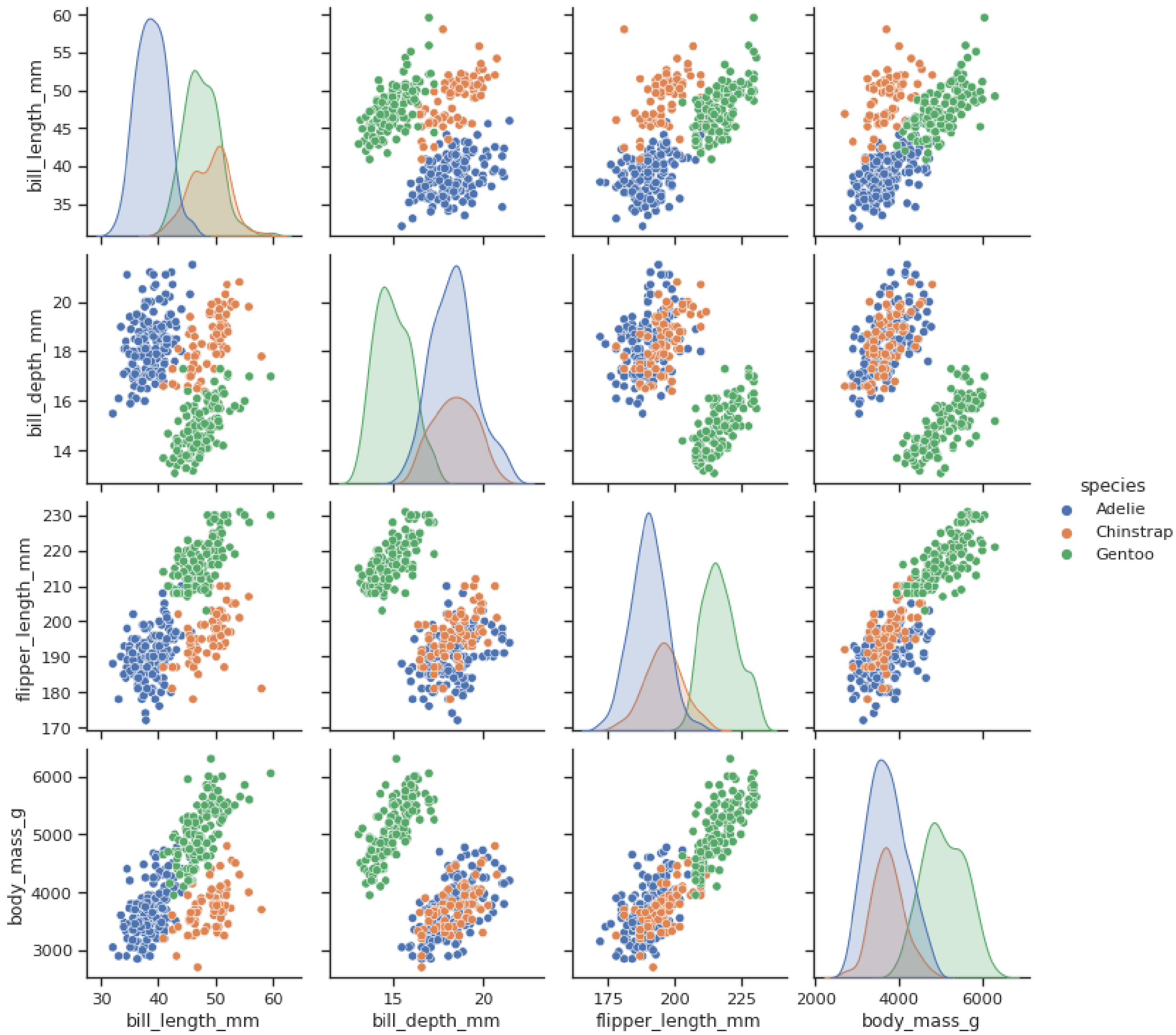
## 3.38 Scatterplot with categorical variables




**seaborn components used:** *set_theme()*, *load_dataset()*, *swarmplot()*

```
import seaborn as sns
sns.set_theme(style="whitegrid", palette="muted")

# Load the penguins dataset
df = sns.load_dataset("penguins")

# Draw a categorical scatterplot to show each observation
ax = sns.swarmplot(data=df, x="body_mass_g", y="sex", hue="species")
ax.set(ylabel="")
```

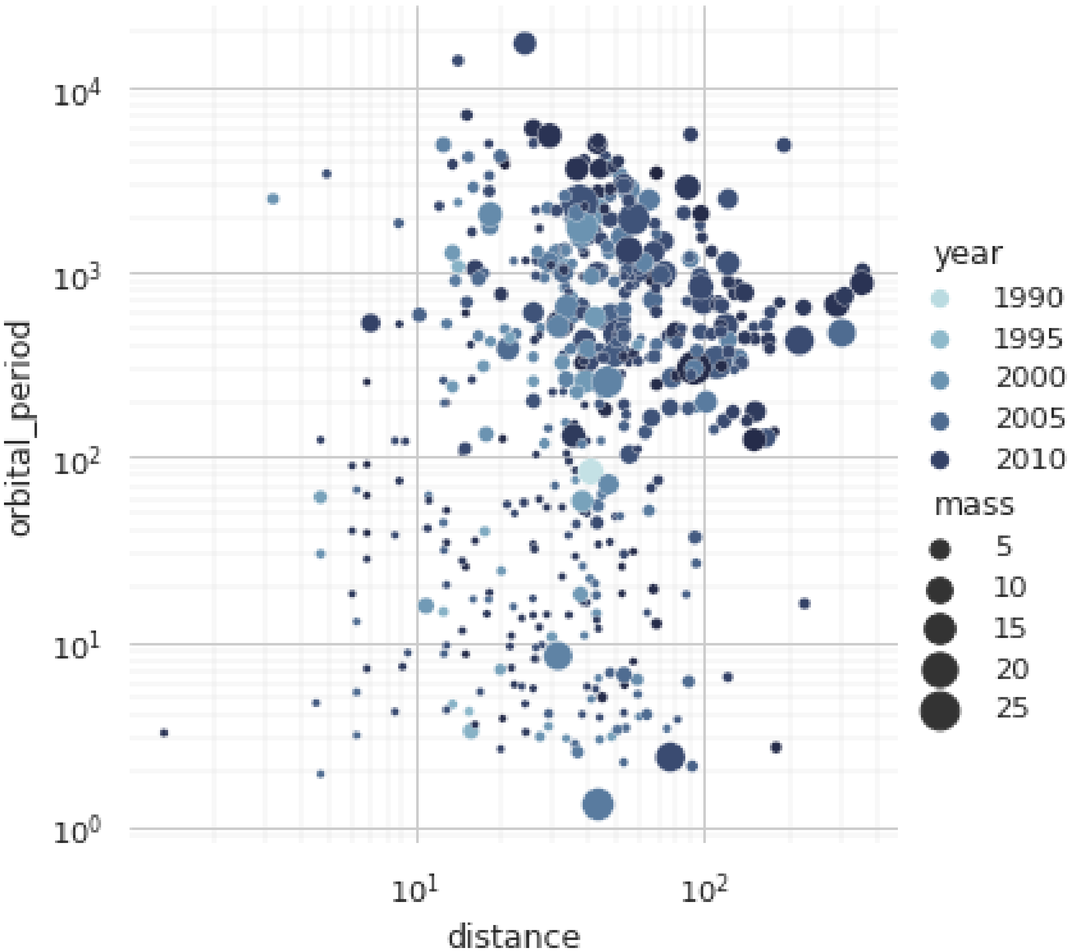
## 3.39 Scatterplot Matrix




**seaborn components used:** `set_theme()`, `load_dataset()`, `pairplot()`

```
import seaborn as sns
sns.set_theme(style="ticks")

df = sns.load_dataset("penguins")
sns.pairplot(df, hue="species")
```

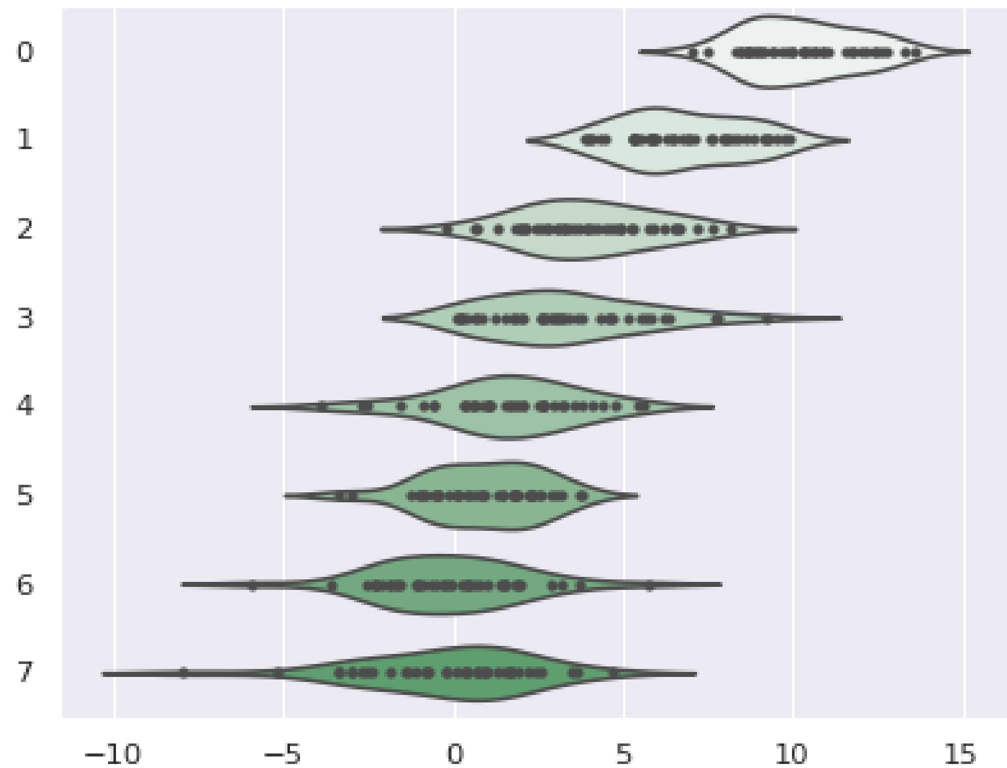
## 3.40 Scatterplot with continuous hues and sizes




**seaborn components used:** *set_theme()*, *load_dataset()*, *cubehelix_palette()*, *relplot()*

```
import seaborn as sns
sns.set_theme(style="whitegrid")

# Load the example planets dataset
planets = sns.load_dataset("planets")

cmap = sns.cubehelix_palette(rot=-.2, as_cmap=True)
g = sns.relplot(
    data=planets,
    x="distance", y="orbital_period",
    hue="year", size="mass",
    palette=cmap, sizes=(10, 200),
)
g.set(xscale="log", yscale="log")
g.ax.xaxis.grid(True, "minor", linewidth=.25)
g.ax.yaxis.grid(True, "minor", linewidth=.25)
g.despine(left=True, bottom=True)
```

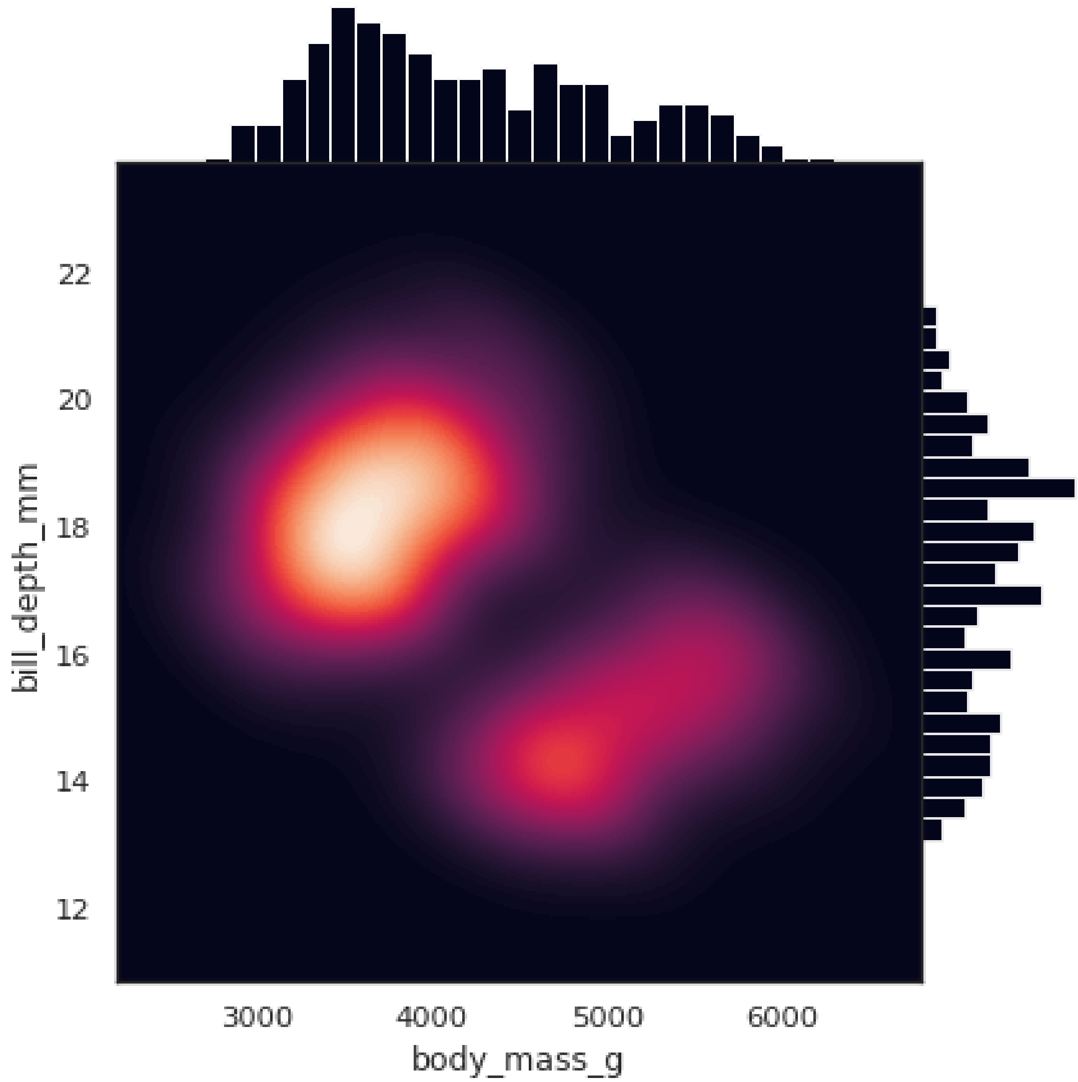
## 3.41 Violinplots with observations




**seaborn components used:** *set_theme()*, *violinplot()*

```
import numpy as np
import seaborn as sns

sns.set_theme()

# Create a random dataset across several variables
rs = np.random.default_rng(0)
n, p = 40, 8
d = rs.normal(0, 2, (n, p))
d += np.log(np.arange(1, p + 1)) * -5 + 10

# Show each distribution with both violins and points
sns.violinplot(data=d, palette="light:g", inner="points", orient="h")
```

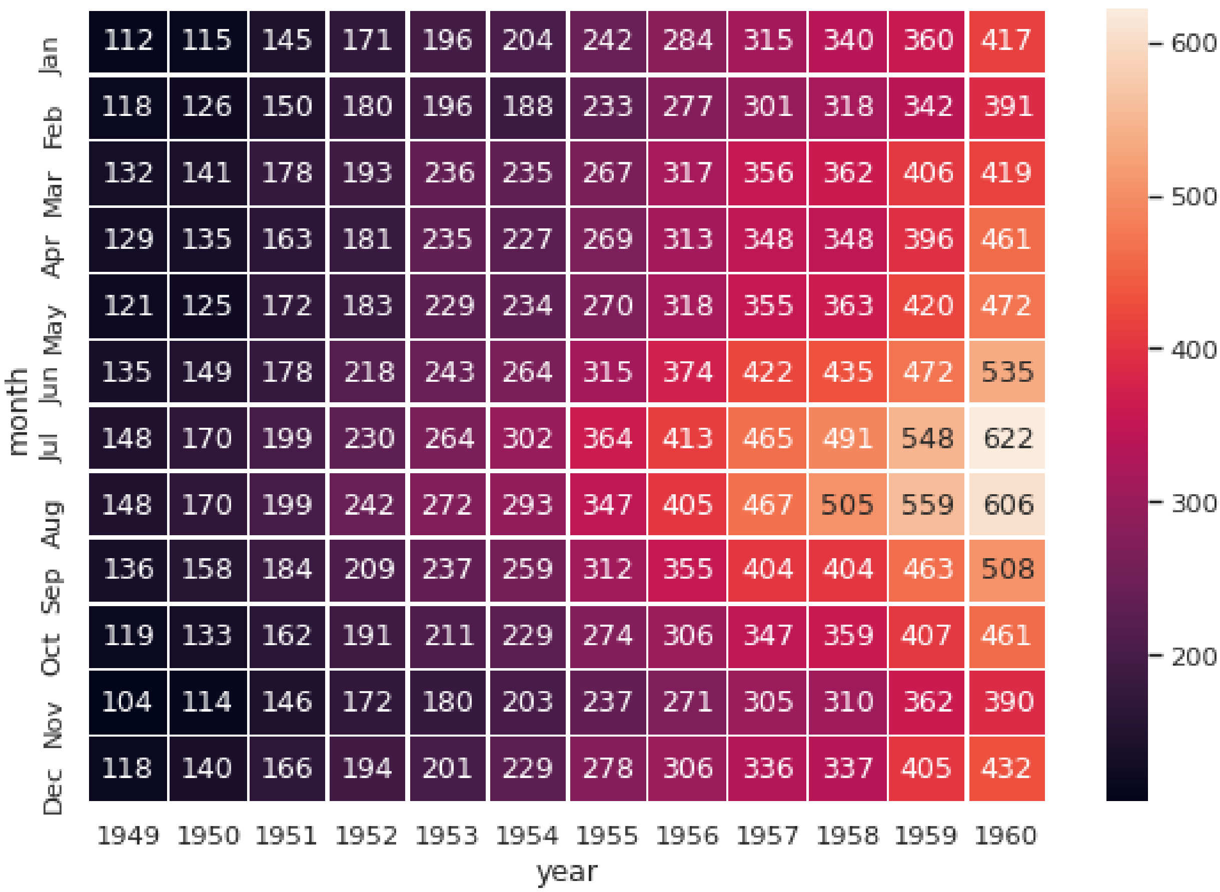
## 3.42 Smooth kernel density with marginal histograms




**seaborn components used:** *set_theme()*, *load_dataset()*, *JointGrid*

```
import seaborn as sns
sns.set_theme(style="white")

df = sns.load_dataset("penguins")

g = sns.JointGrid(data=df, x="body_mass_g", y="bill_depth_mm", space=0)
g.plot_joint(sns.kdeplot,
             fill=True, clip=((2200, 6800), (10, 25)),
             thresh=0, levels=100, cmap="rocket")
g.plot_marginals(sns.histplot, color="#03051A", alpha=1, bins=25)
```

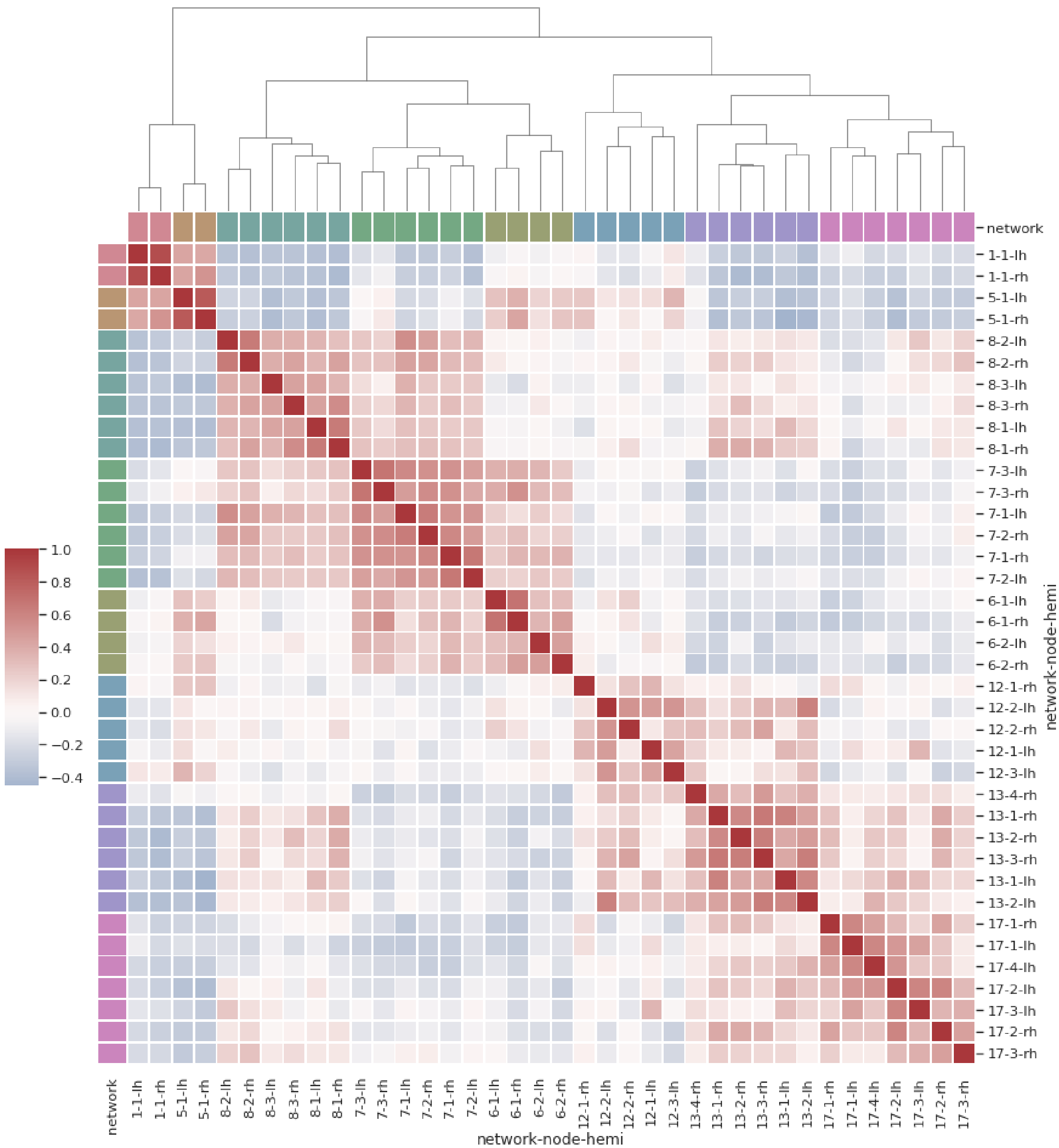
## 3.43 Annotated heatmaps




**seaborn components used:** *set_theme()*, *load_dataset()*, *heatmap()*

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme()

# Load the example flights dataset and convert to long-form
flights_long = sns.load_dataset("flights")
flights = flights_long.pivot("month", "year", "passengers")

# Draw a heatmap with the numeric values in each cell
f, ax = plt.subplots(figsize=(9, 6))
sns.heatmap(flights, annot=True, fmt="d", linewidths=.5, ax=ax)
```

## 3.44 Discovering structure in heatmap data




**seaborn components used:** *set_theme()*, *load_dataset()*, *husl_palette()*, *clustermap()*

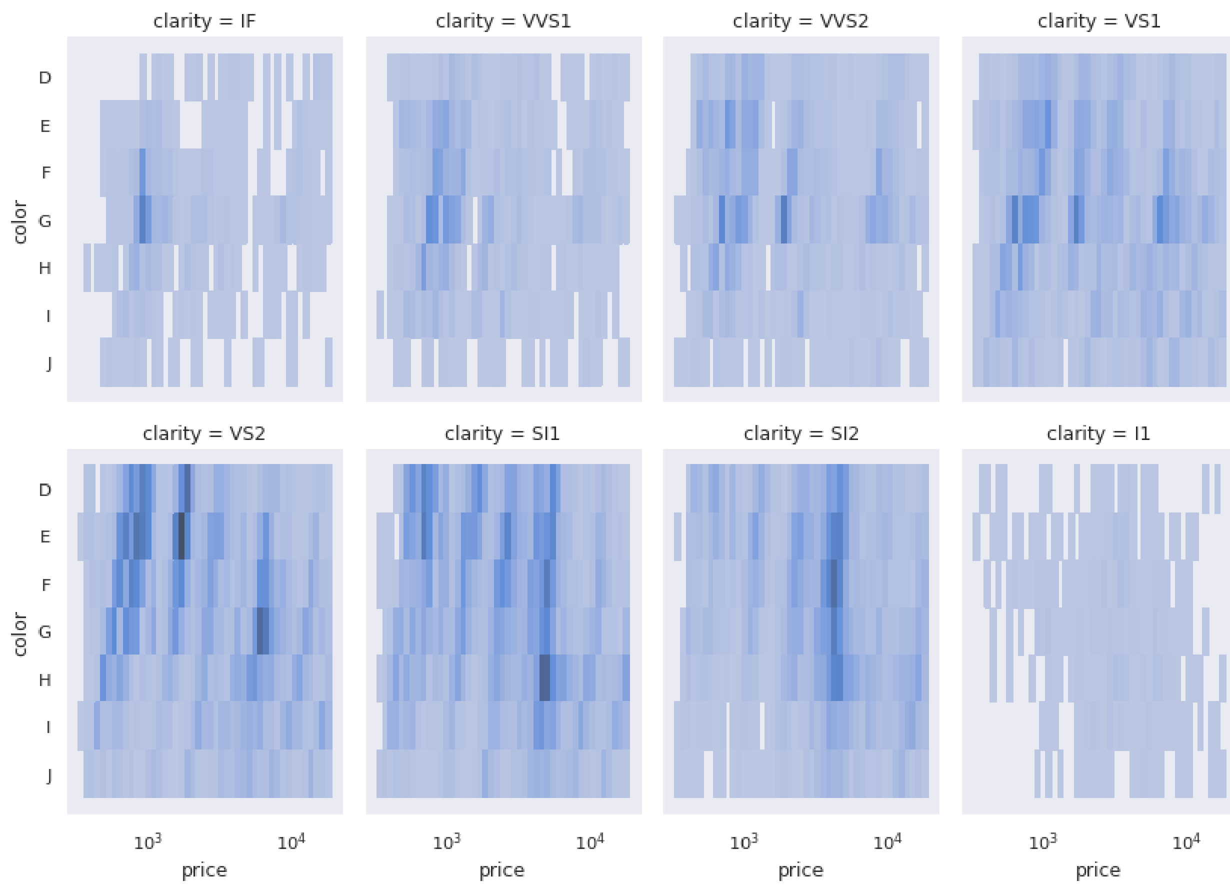
```
import pandas as pd
import seaborn as sns
sns.set_theme()

# Load the brain networks example dataset
df = sns.load_dataset("brain_networks", header=[0, 1, 2], index_col=0)
```

(continues on next page)

(continued from previous page)

```
# Select a subset of the networks
used_networks = [1, 5, 6, 7, 8, 12, 13, 17]
used_columns = (df.columns.get_level_values("network")
                          .astype(int)
                          .isin(used_networks))
df = df.loc[:, used_columns]

# Create a categorical palette to identify the networks
network_pal = sns.husl_palette(8, s=.45)
network_lut = dict(zip(map(str, used_networks), network_pal))

# Convert the palette to vectors that will be drawn on the side of the matrix
networks = df.columns.get_level_values("network")
network_colors = pd.Series(networks, index=df.columns).map(network_lut)

# Draw the full plot
g = sns.clustermap(df.corr(), center=0, cmap="vlag",
                   row_colors=network_colors, col_colors=network_colors,
                   dendrogram_ratio=(.1, .2),
                   cbar_pos=(.02, .32, .03, .2),
                   linewidths=.75, figsize=(12, 13))

g.ax_row_dendrogram.remove()
```

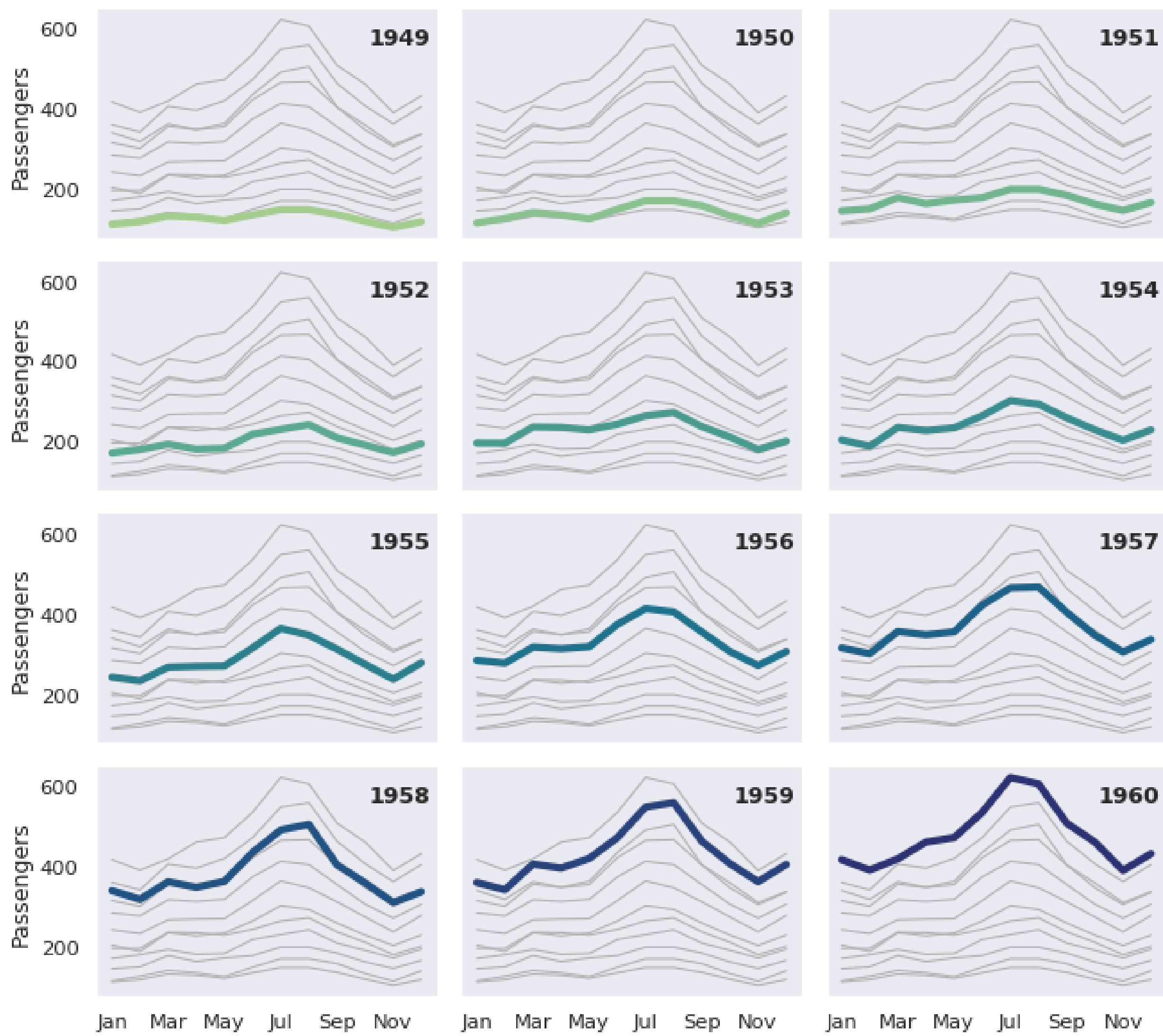
## 3.45 Trivariate histogram with two categorical variables




**seaborn components used:** *set_theme()*, *load_dataset()*, *displot()*

```
import seaborn as sns
sns.set_theme(style="dark")

diamonds = sns.load_dataset("diamonds")
sns.displot(
    data=diamonds, x="price", y="color", col="clarity",
    log_scale=(True, False), col_wrap=4, height=4, aspect=.7,
)
```

## 3.46 Small multiple time series




**seaborn components used:** *set_theme()*, *load_dataset()*, *relplot()*, *lineplot()*

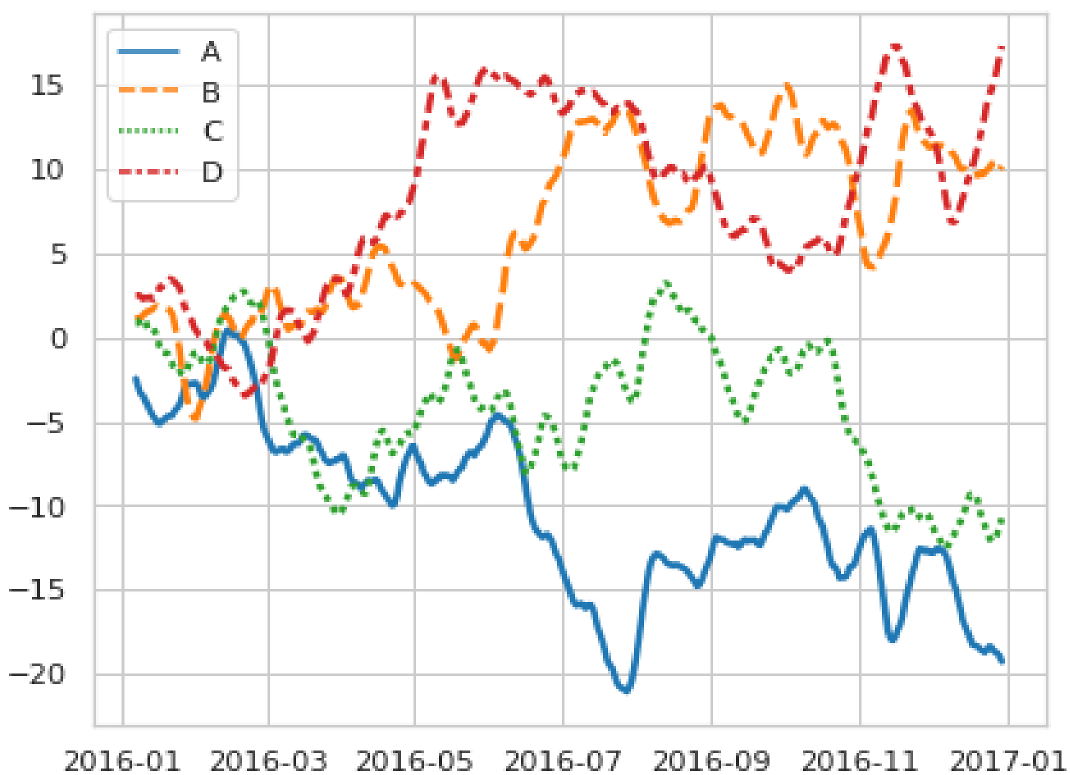
```
import seaborn as sns

sns.set_theme(style="dark")
flights = sns.load_dataset("flights")

# Plot each year's time series in its own facet
g = sns.relplot(
    data=flights,
    x="month", y="passengers", col="year", hue="year",
    kind="line", palette="crest", linewidth=4, zorder=5,
    col_wrap=3, height=2, aspect=1.5, legend=False,
)

# Iterate over each subplot to customize further
```

(continues on next page)

(continued from previous page)

```
for year, ax in g.axes_dict.items():

    # Add the title as an annotation within the plot
    ax.text(.8, .85, year, transform=ax.transAxes, fontweight="bold")

    # Plot every year's time series in the background
    sns.lineplot(
        data=flights, x="month", y="passengers", units="year",
        estimator=None, color=".7", linewidth=1, ax=ax,
    )

# Reduce the frequency of the x axis ticks
ax.set_xticks(ax.get_xticks()[::2])

# Tweak the supporting aspects of the plot
g.set_titles("")
g.set_axis_labels("", "Passengers")
g.tight_layout()
```

## 3.47 Lineplot from a wide-form dataset




**seaborn components used:** `set_theme()`, `lineplot()`

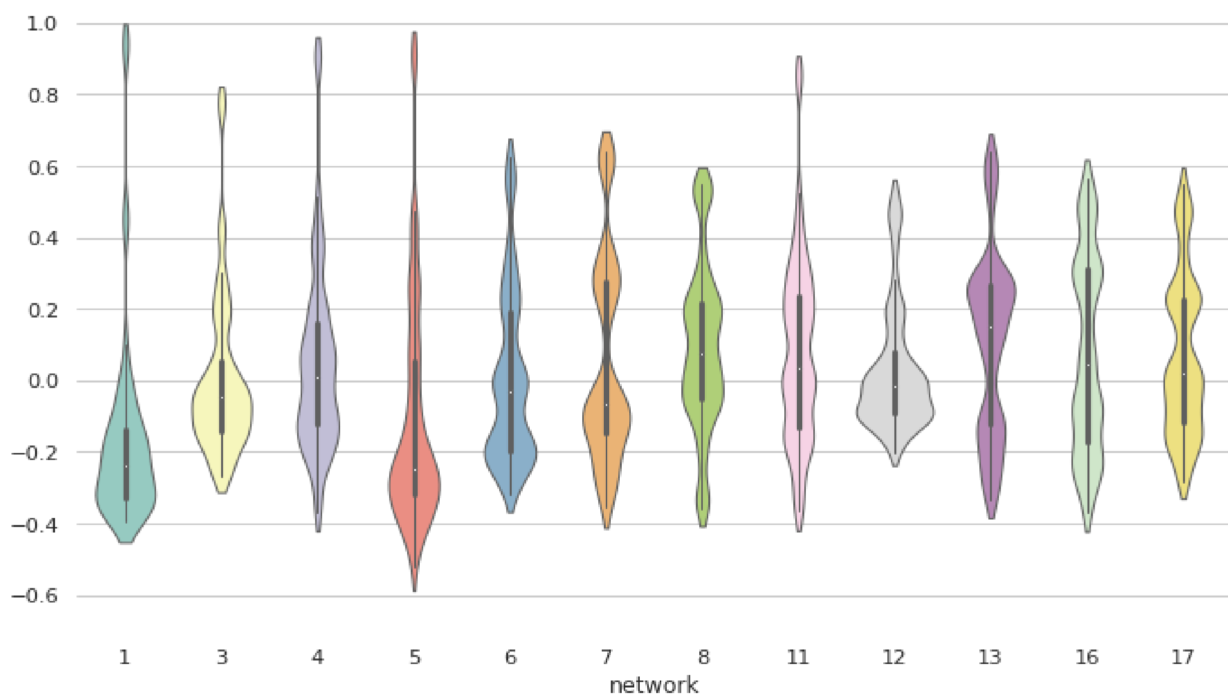
```
import numpy as np
import pandas as pd
import seaborn as sns
```

(continues on next page)

(continued from previous page)

```
sns.set_theme(style="whitegrid")

rs = np.random.RandomState(365)
values = rs.randn(365, 4).cumsum(axis=0)
dates = pd.date_range("1 1 2016", periods=365, freq="D")
data = pd.DataFrame(values, dates, columns=["A", "B", "C", "D"])
data = data.rolling(7).mean()

sns.lineplot(data=data, palette="tab10", linewidth=2.5)
```

## 3.48 Violinplot from a wide-form dataset




**seaborn components used:** *set_theme()*, *load_dataset()*, *violinplot()*, *despine()*

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="whitegrid")

# Load the example dataset of brain network correlations
df = sns.load_dataset("brain_networks", header=[0, 1, 2], index_col=0)

# Pull out a specific subset of networks
used_networks = [1, 3, 4, 5, 6, 7, 8, 11, 12, 13, 16, 17]
used_columns = (df.columns.get_level_values("network")
                          .astype(int)
                          .isin(used_networks))
df = df.loc[:, used_columns]
```

(continues on next page)

(continued from previous page)

```
# Compute the correlation matrix and average over networks
corr_df = df.corr().groupby(level="network").mean()
corr_df.index = corr_df.index.astype(int)
corr_df = corr_df.sort_index().T

# Set up the matplotlib figure
f, ax = plt.subplots(figsize=(11, 6))

# Draw a violinplot with a narrower bandwidth than the default
sns.violinplot(data=corr_df, palette="Set3", bw=.2, cut=1, linewidth=1)

# Finalize the figure
ax.set(ylim=(-.7, 1.05))
sns.despine(left=True, bottom=True)
```

CHAPTER

# FOUR

# USER GUIDE AND TUTORIAL

CHAPTER

FIVE

# API REFERENCE

## 5.1 Relational plots

| | |
|---|---|
| *relplot* | Figure-level interface for drawing relational plots onto a FacetGrid. |
| *scatterplot* | Draw a scatter plot with possibility of several semantic groupings. |
| *lineplot* | Draw a line plot with possibility of several semantic groupings. |

### 5.1.1 seaborn.relplot

seaborn.**relplot**(**, x=None, y=None, hue=None, size=None, style=None, data=None, row=None, col=None, col_wrap=None, row_order=None, col_order=None, palette=None, hue_order=None, hue_norm=None, sizes=None, size_order=None, size_norm=None, markers=None, dashes=None, style_order=None, legend='auto', kind='scatter', height=5, aspect=1, facet_kws=None, units=None, **kwargs*)

Figure-level interface for drawing relational plots onto a FacetGrid.

This function provides access to several different axes-level functions that show the relationship between two variables with semantic mappings of subsets. The `kind` parameter selects the underlying axes-level function to use:

- *scatterplot()* (with `kind="scatter"`; the default)
- *lineplot()* (with `kind="line"`)

Extra keyword arguments are passed to the underlying function, so you should refer to the documentation for each to see kind-specific options.

The relationship between `x` and `y` can be shown for different subsets of the data using the `hue`, `size`, and `style` parameters. These parameters control what visual semantics are used to identify the different subsets. It is possible to show up to three dimensions independently by using all three semantic types, but this style of plot can be hard to interpret and is often ineffective. Using redundant semantics (i.e. both `hue` and `style` for the same variable) can be helpful for making graphics more accessible.

See the tutorial for more information.

The default treatment of the `hue` (and to a lesser extent, `size`) semantic, if present, depends on whether the variable is inferred to represent "numeric" or "categorical" data. In particular, numeric variables are represented with a sequential colormap by default, and the legend entries show regular "ticks" with values that may or may not exist in the data. This behavior can be controlled through various parameters, as described and illustrated below.

After plotting, the *FacetGrid* with the plot is returned and can be used directly to tweak supporting plot details or add other layers.

Note that, unlike when using the underlying plotting functions directly, data must be passed in a long-form DataFrame with variables specified by passing strings to x, y, and other parameters.

**Parameters**

**x, y** [vectors or keys in data] Variables that specify positions on the x and y axes.

**hue** [vector or key in data] Grouping variable that will produce elements with different colors. Can be either categorical or numeric, although color mapping will behave differently in latter case.

**size** [vector or key in data] Grouping variable that will produce elements with different sizes. Can be either categorical or numeric, although size mapping will behave differently in latter case.

**style** [vector or key in data] Grouping variable that will produce elements with different styles. Can have a numeric dtype but will always be treated as categorical.

**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.

**row, col** [vectors or keys in data] Variables that define subsets to plot on different facets.

**col_wrap** [int] "Wrap" the column variable at this width, so that the column facets span multiple rows. Incompatible with a row facet.

**row_order, col_order** [lists of strings] Order to organize the rows and/or columns of the grid in, otherwise the orders are inferred from the data objects.

**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.

**sizes** [list, dict, or tuple] An object that determines how sizes are chosen when size is used. It can always be a list of size values or a dict mapping levels of the size variable to sizes. When size is numeric, it can also be a tuple specifying the minimum and maximum size to use such that other values are normalized within this range.

**size_order** [list] Specified order for appearance of the size variable levels, otherwise they are determined from the data. Not relevant when the size variable is numeric.

**size_norm** [tuple or Normalize object] Normalization in data units for scaling plot objects when the size variable is numeric.

**style_order** [list] Specified order for appearance of the style variable levels otherwise they are determined from the data. Not relevant when the style variable is numeric.

**dashes** [boolean, list, or dictionary] Object determining how to draw the lines for different levels of the style variable. Setting to True will use default dash codes, or you can pass a list of dash codes or a dictionary mapping levels of the style variable to dash codes.

Setting to `False` will use solid lines for all subsets. Dashes are specified as in matplotlib: a tuple of `(segment, gap)` lengths, or an empty string to draw a solid line.

**markers** [boolean, list, or dictionary] Object determining how to draw the markers for different levels of the `style` variable. Setting to `True` will use default markers, or you can pass a list of markers or a dictionary mapping levels of the `style` variable to markers. Setting to `False` will draw marker-less lines. Markers are specified as in matplotlib.

**legend** ["auto", "brief", "full", or False] How to draw the legend. If "brief", numeric `hue` and `size` variables will be represented with a sample of evenly spaced values. If "full", every group will get an entry in the legend. If "auto", choose between brief or full representation based on number of levels. If `False`, no legend data is added and no legend is drawn.

**kind** [string] Kind of plot to draw, corresponding to a seaborn relational plot. Options are {`scatter` and `line`}.

**height** [scalar] Height (in inches) of each facet. See also: `aspect`.

**aspect** [scalar] Aspect ratio of each facet, so that `aspect * height` gives the width of each facet in inches.

**facet_kws** [dict] Dictionary of other keyword arguments to pass to *`FacetGrid`*.

**units** [vector or key in `data`] Grouping variable identifying sampling units. When used, a separate line will be drawn for each unit with appropriate semantics, but no legend entry will be added. Useful for showing distribution of experimental replicates when exact identities are not needed.

**kwargs** [key, value pairings] Other keyword arguments are passed through to the underlying plotting function.

**Returns**

***`FacetGrid`*** An object managing one or more subplots that correspond to conditional data subsets with convenient methods for batch-setting of axes attributes.

#### Examples

### 5.1.2 seaborn.scatterplot

`seaborn.`**`scatterplot`**`(`*, *x=None*, *y=None*, *hue=None*, *style=None*, *size=None*, *data=None*, *palette=None*, *hue_order=None*, *hue_norm=None*, *sizes=None*, *size_order=None*, *size_norm=None*, *markers=True*, *style_order=None*, *x_bins=None*, *y_bins=None*, *units=None*, *estimator=None*, *ci=95*, *n_boot=1000*, *alpha=None*, *x_jitter=None*, *y_jitter=None*, *legend='auto'*, *ax=None*, ***kwargs*`)`

Draw a scatter plot with possibility of several semantic groupings.

The relationship between `x` and `y` can be shown for different subsets of the data using the `hue`, `size`, and `style` parameters. These parameters control what visual semantics are used to identify the different subsets. It is possible to show up to three dimensions independently by using all three semantic types, but this style of plot can be hard to interpret and is often ineffective. Using redundant semantics (i.e. both `hue` and `style` for the same variable) can be helpful for making graphics more accessible.

See the tutorial for more information.

The default treatment of the `hue` (and to a lesser extent, `size`) semantic, if present, depends on whether the variable is inferred to represent "numeric" or "categorical" data. In particular, numeric variables are represented with a sequential colormap by default, and the legend entries show regular "ticks" with values that may or may not exist in the data. This behavior can be controlled through various parameters, as described and illustrated below.

**Parameters**

**x, y** [vectors or keys in data] Variables that specify positions on the x and y axes.

**hue** [vector or key in data] Grouping variable that will produce points with different colors. Can be either categorical or numeric, although color mapping will behave differently in latter case.

**size** [vector or key in data] Grouping variable that will produce points with different sizes. Can be either categorical or numeric, although size mapping will behave differently in latter case.

**style** [vector or key in data] Grouping variable that will produce points with different markers. Can have a numeric dtype but will always be treated as categorical.

**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.

**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.

**sizes** [list, dict, or tuple] An object that determines how sizes are chosen when size is used. It can always be a list of size values or a dict mapping levels of the size variable to sizes. When size is numeric, it can also be a tuple specifying the minimum and maximum size to use such that other values are normalized within this range.

**size_order** [list] Specified order for appearance of the size variable levels, otherwise they are determined from the data. Not relevant when the size variable is numeric.

**size_norm** [tuple or Normalize object] Normalization in data units for scaling plot objects when the size variable is numeric.

**markers** [boolean, list, or dictionary] Object determining how to draw the markers for different levels of the style variable. Setting to True will use default markers, or you can pass a list of markers or a dictionary mapping levels of the style variable to markers. Setting to False will draw marker-less lines. Markers are specified as in matplotlib.

**style_order** [list] Specified order for appearance of the style variable levels otherwise they are determined from the data. Not relevant when the style variable is numeric.

**{x,y}_bins** [lists or arrays or functions] *Currently non-functional.*

**units** [vector or key in data] Grouping variable identifying sampling units. When used, a separate line will be drawn for each unit with appropriate semantics, but no legend entry will be added. Useful for showing distribution of experimental replicates when exact identities are not needed. *Currently non-functional.*

**estimator** [name of pandas method or callable or None] Method for aggregating across multiple observations of the y variable at the same x level. If None, all observations will be drawn. *Currently non-functional.*

**ci** [int or "sd" or None] Size of the confidence interval to draw when aggregating with an estimator. "sd" means to draw the standard deviation of the data. Setting to `None` will skip bootstrapping. *Currently non-functional.*

**n_boot** [int] Number of bootstraps to use for computing the confidence interval. *Currently non-functional.*

**alpha** [float] Proportional opacity of the points.

**{x,y}_jitter** [booleans or floats] *Currently non-functional.*

**legend** ["auto", "brief", "full", or False] How to draw the legend. If "brief", numeric `hue` and `size` variables will be represented with a sample of evenly spaced values. If "full", every group will get an entry in the legend. If "auto", choose between brief or full representation based on number of levels. If `False`, no legend data is added and no legend is drawn.

**ax** [`matplotlib.axes.Axes`] Pre-existing axes for the plot. Otherwise, call `matplotlib.pyplot.gca()` internally.

**kwargs** [key, value mappings] Other keyword arguments are passed down to `matplotlib.axes.Axes.scatter()`.

**Returns**

**`matplotlib.axes.Axes`** The matplotlib axes containing the plot.

**See also:**

***`lineplot`*** Plot data using lines.

***`stripplot`*** Plot a categorical scatter with jitter.

***`swarmplot`*** Plot a categorical scatter with non-overlapping points.

#### Examples

### 5.1.3 seaborn.lineplot

seaborn.**lineplot**(*, *x=None*, *y=None*, *hue=None*, *size=None*, *style=None*, *data=None*, *palette=None*, *hue_order=None*, *hue_norm=None*, *sizes=None*, *size_order=None*, *size_norm=None*, *dashes=True*, *markers=None*, *style_order=None*, *units=None*, *estimator='mean'*, *ci=95*, *n_boot=1000*, *seed=None*, *sort=True*, *err_style='band'*, *err_kws=None*, *legend='auto'*, *ax=None*, ***kwargs*)

Draw a line plot with possibility of several semantic groupings.

The relationship between `x` and `y` can be shown for different subsets of the data using the `hue`, `size`, and `style` parameters. These parameters control what visual semantics are used to identify the different subsets. It is possible to show up to three dimensions independently by using all three semantic types, but this style of plot can be hard to interpret and is often ineffective. Using redundant semantics (i.e. both `hue` and `style` for the same variable) can be helpful for making graphics more accessible.

See the tutorial for more information.

The default treatment of the `hue` (and to a lesser extent, `size`) semantic, if present, depends on whether the variable is inferred to represent "numeric" or "categorical" data. In particular, numeric variables are represented with a sequential colormap by default, and the legend entries show regular "ticks" with values that may or may not exist in the data. This behavior can be controlled through various parameters, as described and illustrated below.

By default, the plot aggregates over multiple `y` values at each value of `x` and shows an estimate of the central tendency and a confidence interval for that estimate.

**Parameters**

**x, y** [vectors or keys in `data`] Variables that specify positions on the x and y axes.

**hue** [vector or key in `data`] Grouping variable that will produce lines with different colors. Can be either categorical or numeric, although color mapping will behave differently in latter case.

**size** [vector or key in `data`] Grouping variable that will produce lines with different widths. Can be either categorical or numeric, although size mapping will behave differently in latter case.

**style** [vector or key in `data`] Grouping variable that will produce lines with different dashes and/or markers. Can have a numeric dtype but will always be treated as categorical.

**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.

**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.

**sizes** [list, dict, or tuple] An object that determines how sizes are chosen when `size` is used. It can always be a list of size values or a dict mapping levels of the `size` variable to sizes. When `size` is numeric, it can also be a tuple specifying the minimum and maximum size to use such that other values are normalized within this range.

**size_order** [list] Specified order for appearance of the `size` variable levels, otherwise they are determined from the data. Not relevant when the `size` variable is numeric.

**size_norm** [tuple or Normalize object] Normalization in data units for scaling plot objects when the `size` variable is numeric.

**dashes** [boolean, list, or dictionary] Object determining how to draw the lines for different levels of the `style` variable. Setting to `True` will use default dash codes, or you can pass a list of dash codes or a dictionary mapping levels of the `style` variable to dash codes. Setting to `False` will use solid lines for all subsets. Dashes are specified as in matplotlib: a tuple of `(segment, gap)` lengths, or an empty string to draw a solid line.

**markers** [boolean, list, or dictionary] Object determining how to draw the markers for different levels of the `style` variable. Setting to `True` will use default markers, or you can pass a list of markers or a dictionary mapping levels of the `style` variable to markers. Setting to `False` will draw marker-less lines. Markers are specified as in matplotlib.

**style_order** [list] Specified order for appearance of the `style` variable levels otherwise they are determined from the data. Not relevant when the `style` variable is numeric.

**units** [vector or key in `data`] Grouping variable identifying sampling units. When used, a separate line will be drawn for each unit with appropriate semantics, but no legend entry will be added. Useful for showing distribution of experimental replicates when exact identities are not needed.

**estimator** [name of pandas method or callable or None] Method for aggregating across multiple observations of the `y` variable at the same `x` level. If `None`, all observations will be drawn.

**ci** [int or "sd" or None] Size of the confidence interval to draw when aggregating with an estimator. "sd" means to draw the standard deviation of the data. Setting to `None` will skip bootstrapping.

**n_boot** [int] Number of bootstraps to use for computing the confidence interval.

**seed** [int, numpy.random.Generator, or numpy.random.RandomState] Seed or random number generator for reproducible bootstrapping.

**sort** [boolean] If True, the data will be sorted by the x and y variables, otherwise lines will connect points in the order they appear in the dataset.

**err_style** ["band" or "bars"] Whether to draw the confidence intervals with translucent error bands or discrete error bars.

**err_kws** [dict of keyword arguments] Additional paramters to control the aesthetics of the error bars. The kwargs are passed either to `matplotlib.axes.Axes.fill_between()` or `matplotlib.axes.Axes.errorbar()`, depending on `err_style`.

**legend** ["auto", "brief", "full", or False] How to draw the legend. If "brief", numeric `hue` and `size` variables will be represented with a sample of evenly spaced values. If "full", every group will get an entry in the legend. If "auto", choose between brief or full representation based on number of levels. If `False`, no legend data is added and no legend is drawn.

**ax** [`matplotlib.axes.Axes`] Pre-existing axes for the plot. Otherwise, call `matplotlib.pyplot.gca()` internally.

**kwargs** [key, value mappings] Other keyword arguments are passed down to `matplotlib.axes.Axes.plot()`.

**Returns**

**`matplotlib.axes.Axes`** The matplotlib axes containing the plot.

**See also:**

***`scatterplot`*** Plot data using points.

***`pointplot`*** Plot point estimates and CIs using markers and lines.

### Examples

## 5.2 Distribution plots

| | |
|---|---|
| *`displot`* | Figure-level interface for drawing distribution plots onto a FacetGrid. |
| *`histplot`* | Plot univariate or bivariate histograms to show distributions of datasets. |
| *`kdeplot`* | Plot univariate or bivariate distributions using kernel density estimation. |
| *`ecdfplot`* | Plot empirical cumulative distribution functions. |
| *`rugplot`* | Plot marginal distributions by drawing ticks along the x and y axes. |

continues on next page

Table 2 – continued from previous page

| | |
|---|---|
| *distplot* | DEPRECATED: Flexibly plot a univariate distribution of observations. |

### 5.2.1 seaborn.displot

seaborn.**displot**(*data=None*, ***, *x=None*, *y=None*, *hue=None*, *row=None*, *col=None*, *weights=None*, *kind='hist'*, *rug=False*, *rug_kws=None*, *log_scale=None*, *legend=True*, *palette=None*, *hue_order=None*, *hue_norm=None*, *color=None*, *col_wrap=None*, *row_order=None*, *col_order=None*, *height=5*, *aspect=1*, *facet_kws=None*, ***kwargs*)

Figure-level interface for drawing distribution plots onto a FacetGrid.

This function provides access to several approaches for visualizing the univariate or bivariate distribution of data, including subsets of data defined by semantic mapping and faceting across multiple subplots. The `kind` parameter selects the approach to use:

- *histplot()* (with `kind="hist"`; the default)
- *kdeplot()* (with `kind="kde"`)
- *ecdfplot()* (with `kind="ecdf"`; univariate-only)

Additionally, a *rugplot()* can be added to any kind of plot to show individual observations.

Extra keyword arguments are passed to the underlying function, so you should refer to the documentation for each to understand the complete set of options for making plots with this interface.

See the distribution plots tutorial for a more in-depth discussion of the relative strengths and weaknesses of each approach. The distinction between figure-level and axes-level functions is explained further in the user guide.

**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.

**row, col** [vectors or keys in `data`] Variables that define subsets to plot on different facets.

**kind** [{"hist", "kde", "ecdf"}] Approach for visualizing the data. Selects the underlying plotting function and determines the additional set of valid parameters.

**rug** [bool] If True, show each observation with marginal ticks (as in *rugplot()*).

**rug_kws** [dict] Parameters to control the appearance of the rug plot.

**log_scale** [bool or number, or pair of bools or numbers] Set a log scale on the data axis (or axes, with bivariate data) with the given base (default 10), and evaluate the KDE in log space.

**legend** [bool] If False, suppress the legend for semantic variables.

**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.

**col_wrap** [int] "Wrap" the column variable at this width, so that the column facets span multiple rows. Incompatible with a `row` facet.

**{row,col}_order** [vector of strings] Specify the order in which levels of the `row` and/or `col` variables appear in the grid of subplots.

**height** [scalar] Height (in inches) of each facet. See also: `aspect`.

**aspect** [scalar] Aspect ratio of each facet, so that `aspect * height` gives the width of each facet in inches.

**facet_kws** [dict] Additional parameters passed to *`FacetGrid`*.

**kwargs** Other keyword arguments are documented with the relevant axes-level function:

- *`histplot()`* (with `kind="hist"`)
- *`kdeplot()`* (with `kind="kde"`)
- *`ecdfplot()`* (with `kind="ecdf"`)

**Returns**

***`FacetGrid`*** An object managing one or more subplots that correspond to conditional data subsets with convenient methods for batch-setting of axes attributes.

**See also:**

***`histplot`*** Plot a histogram of binned counts with optional normalization or smoothing.

***`kdeplot`*** Plot univariate or bivariate distributions using kernel density estimation.

***`rugplot`*** Plot a tick at each observation value along the x and/or y axes.

***`ecdfplot`*** Plot empirical cumulative distribution functions.

***`jointplot`*** Draw a bivariate plot with univariate marginal distributions.

### Examples

See the API documentation for the axes-level functions for more details about the breadth of options available for each plot kind.

### 5.2.2 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*)

Plot univariate or bivariate histograms to show distributions of datasets.

A histogram is a classic visualization tool that represents the distribution of one or more variables by counting the number of observations that fall within disrete bins.

This function can normalize the statistic computed within each bin to estimate frequency, density or probability mass, and it can add a smooth curve obtained using a kernel density estimate, similar to *kdeplot()*.

More information is provided in the user guide.

**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** [{"count", "frequency", "density", "probability"}] Aggregate statistic to compute in each bin.

- count shows the number of observations
- frequency shows the number of observations divided by the bin width
- density normalizes counts so that the area of the histogram is 1
- probability normalizes counts so that the sum of the bar heights is 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 a log scale on the data axis (or axes, with bivariate data) with the given base (default 10), and evaluate the KDE in log space.

**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.

**See also:**

***displot*** Figure-level interface to distribution plot functions.

***kdeplot*** Plot univariate or bivariate distributions using kernel density estimation.

***rugplot*** Plot a tick at each observation value along the x and/or y axes.

***ecdfplot*** Plot empirical cumulative distribution functions.

***jointplot*** Draw a bivariate plot with univariate marginal distributions.

#### Notes

The choice of bins for computing and plotting a histogram can exert substantial influence on the insights that one is able to draw from the visualization. If the bins are too large, they may erase important features. On the other hand, bins that are too small may be dominated by random variability, obscuring the shape of the true underlying distribution. The default bin size is determined using a reference rule that depends on the sample size and variance. This works well in many cases, (i.e., with "well-behaved" data) but it fails in others. It is always a good to try different bin sizes to be sure that you are not missing something important. This function allows you to specify bins in several different ways, such as by setting the total number of bins to use, the width of each bin, or the specific locations where the bins should break.

#### Examples

### 5.2.3 seaborn.kdeplot

seaborn.**kdeplot**(*x=None*, ***, *y=None*, *shade=None*, *vertical=False*, *kernel=None*, *bw=None*, *gridsize=200*, *cut=3*, *clip=None*, *legend=True*, *cumulative=False*, *shade_lowest=None*, *cbar=False*, *cbar_ax=None*, *cbar_kws=None*, *ax=None*, *weights=None*, *hue=None*, *palette=None*, *hue_order=None*, *hue_norm=None*, *multiple='layer'*, *common_norm=True*, *common_grid=False*, *levels=10*, *thresh=0.05*, *bw_method='scott'*, *bw_adjust=1*, *log_scale=None*, *color=None*, *fill=None*, *data=None*, *data2=None*, ***kwargs*)

Plot univariate or bivariate distributions using kernel density estimation.

A kernel density estimate (KDE) plot is a method for visualizing the distribution of observations in a dataset, analagous to a histogram. KDE represents the data using a continuous probability density curve in one or more dimensions.

The approach is explained further in the user guide.

Relative to a histogram, KDE can produce a plot that is less cluttered and more interpretable, especially when drawing multiple distributions. But it has the potential to introduce distortions if the underlying distribution is bounded or not smooth. Like a histogram, the quality of the representation also depends on the selection of good smoothing parameters.

**Parameters**

**x, y** [vectors or keys in `data`] Variables that specify positions on the x and y axes.

**shade** [bool] Alias for `fill`. Using `fill` is recommended.

**vertical** [bool] Orientation parameter.

Deprecated since version 0.11.0: specify orientation by assigning the `x` or `y` variables.

**kernel** [str] Function that defines the kernel.

Deprecated since version 0.11.0: support for non-Gaussian kernels has been removed.

**bw** [str, number, or callable] Smoothing parameter.

Deprecated since version 0.11.0: see `bw_method` and `bw_adjust`.

**gridsize** [int] Number of points on each dimension of the evaluation grid.

**cut** [number, optional] Factor, multiplied by the smoothing bandwidth, that determines how far the evaluation grid extends past the extreme datapoints. When set to 0, truncate the curve at the data limits.

**clip** [pair of numbers None, or a pair of such pairs] Do not evaluate the density outside of these limits.

**legend** [bool] If False, suppress the legend for semantic variables.

**cumulative** [bool, optional] If True, estimate a cumulative distribution function. Requires scipy.

**shade_lowest** [bool] If False, the area below the lowest contour will be transparent

Deprecated since version 0.11.0: see `thresh`.

**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()`.

**ax** [`matplotlib.axes.Axes`] Pre-existing axes for the plot. Otherwise, call `matplotlib.pyplot.gca()` internally.

**weights** [vector or key in `data`] If provided, weight the kernel density estimation using these values.

**hue** [vector or key in `data`] Semantic variable that is mapped to determine the color of plot elements.

**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.

**multiple** [{{"layer", "stack", "fill"}}] Method for drawing multiple elements when semantic mapping creates subsets. Only relevant with univariate data.

**common_norm** [bool] If True, scale each conditional density by the number of observations such that the total area under all densities sums to 1. Otherwise, normalize each density independently.

**common_grid** [bool] If True, use the same evaluation grid for each kernel density estimate. Only relevant with univariate data.

**levels** [int or vector] Number of contour levels or values to draw contours at. A vector argument must have increasing values in [0, 1]. Levels correspond to iso-proportions of the density: e.g., 20% of the probability mass will lie below the contour drawn for 0.2. Only relevant with bivariate data.

**thresh** [number in [0, 1]] Lowest iso-proportion level at which to draw a contour line. Ignored when `levels` is a vector. Only relevant with bivariate data.

**bw_method** [string, scalar, or callable, optional] Method for determining the smoothing bandwidth to use; passed to `scipy.stats.gaussian_kde`.

**bw_adjust** [number, optional] Factor that multiplicatively scales the value chosen using `bw_method`. Increasing will make the curve smoother. See Notes.

**log_scale** [bool or number, or pair of bools or numbers] Set a log scale on the data axis (or axes, with bivariate data) with the given base (default 10), and evaluate the KDE in log space.

**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.

**fill** [bool or None] If True, fill in the area under univariate density curves or between bivariate contours. If None, the default depends on `multiple`.

**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.

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

- `matplotlib.axes.Axes.plot()` (univariate, `fill=False`),
- `matplotlib.axes.Axes.fill_between()` (univariate, `fill=True`),
- `matplotlib.axes.Axes.contour()` (bivariate, `fill=False`),
- `matplotlib.axes.contourf()` (bivariate, `fill=True`).

**Returns**

**`matplotlib.axes.Axes`** The matplotlib axes containing the plot.

**See also:**

***`displot`*** Figure-level interface to distribution plot functions.

***`histplot`*** Plot a histogram of binned counts with optional normalization or smoothing.

***`ecdfplot`*** Plot empirical cumulative distribution functions.

***`jointplot`*** Draw a bivariate plot with univariate marginal distributions.

***`violinplot`*** Draw an enhanced boxplot using kernel density estimation.

### Notes

The *bandwidth*, or standard deviation of the smoothing kernel, is an important parameter. Misspecification of the bandwidth can produce a distorted representation of the data. Much like the choice of bin width in a histogram, an over-smoothed curve can erase true features of a distribution, while an under-smoothed curve can create false features out of random variability. The rule-of-thumb that sets the default bandwidth works best when the true distribution is smooth, unimodal, and roughly bell-shaped. It is always a good idea to check the default behavior by using `bw_adjust` to increase or decrease the amount of smoothing.

Because the smoothing algorithm uses a Gaussian kernel, the estimated density curve can extend to values that do not make sense for a particular dataset. For example, the curve may be drawn over negative values when smoothing data that are naturally positive. The `cut` and `clip` parameters can be used to control the extent of the curve, but datasets that have many observations close to a natural boundary may be better served by a different visualization method.

Similar considerations apply when a dataset is naturally discrete or "spiky" (containing many repeated observations of the same value). Kernel density estimation will always produce a smooth curve, which would be misleading in these situations.

The units on the density axis are a common source of confusion. While kernel density estimation produces a probability distribution, the height of the curve at each point gives a density, not a probability. A probability can be obtained only by integrating the density across a range. The curve is normalized so that the integral over all possible values is 1, meaning that the scale of the density axis depends on the data values.

### Examples

## 5.2.4 seaborn.ecdfplot

`seaborn.`**`ecdfplot`**(*data=None*, ***, *x=None*, *y=None*, *hue=None*, *weights=None*, *stat='proportion'*, *complementary=False*, *palette=None*, *hue_order=None*, *hue_norm=None*, *log_scale=None*, *legend=True*, *ax=None*, ***kwargs*)

Plot empirical cumulative distribution functions.

An ECDF represents the proportion or count of observations falling below each unique value in a dataset. Compared to a histogram or density plot, it has the advantage that each observation is visualized directly, meaning that there are no binning or smoothing parameters that need to be adjusted. It also aids direct comparisons between multiple distributions. A downside is that the relationship between the appearance of the plot and the basic properties of the distribution (such as its central tendency, variance, and the presence of any bimodality) may not be as intuitive.

More information is provided in the user guide.

**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 cumulative distribution using these values.

**stat** [{{"proportion", "count"}}] Distribution statistic to compute.

**complementary** [bool] If True, use the complementary CDF (1 - CDF)

**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.

**log_scale** [bool or number, or pair of bools or numbers] Set a log scale on the data axis (or axes, with bivariate data) with the given base (default 10), and evaluate the KDE in log space.

**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 `matplotlib.axes.Axes.plot()`.

**Returns**

**`matplotlib.axes.Axes`** The matplotlib axes containing the plot.

**See also:**

***`displot`*** Figure-level interface to distribution plot functions.

***`histplot`*** Plot a histogram of binned counts with optional normalization or smoothing.

***`kdeplot`*** Plot univariate or bivariate distributions using kernel density estimation.

***`rugplot`*** Plot a tick at each observation value along the x and/or y axes.

#### Examples

### 5.2.5 seaborn.rugplot

`seaborn.`**`rugplot`**(*x=None*, ***, *height=0.025*, *axis=None*, *ax=None*, *data=None*, *y=None*, *hue=None*, *palette=None*, *hue_order=None*, *hue_norm=None*, *expand_margins=True*, *legend=True*, *a=None*, ***kwargs*)

Plot marginal distributions by drawing ticks along the x and y axes.

This function is intended to complement other plots by showing the location of individual observations in an unobstrusive way.

**Parameters**

**x, y** [vectors or keys in `data`] Variables that specify positions on the x and y axes.

**height** [number] Proportion of axes extent covered by each rug element.

**axis** [{“x”, “y”}] Axis to draw the rug on.

Deprecated since version 0.11.0: specify axis by assigning the `x` or `y` variables.

**ax** [`matplotlib.axes.Axes`] Pre-existing axes for the plot. Otherwise, call `matplotlib.pyplot.gca()` internally.

**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.

**hue** [vector or key in `data`] Semantic variable that is mapped to determine the color of plot elements.

**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.

**expand_margins** [bool] If True, increase the axes margins by the height of the rug to avoid overlap with other elements.

**legend** [bool] If False, do not add a legend for semantic variables.

**kwargs** Other keyword arguments are passed to `matplotlib.collections.LineCollection()`

**Returns**

**`matplotlib.axes.Axes`** The matplotlib axes containing the plot.

### Examples

## 5.2.6 seaborn.distplot

`seaborn.`**`distplot`**(*a=None*, *bins=None*, *hist=True*, *kde=True*, *rug=False*, *fit=None*, *hist_kws=None*, *kde_kws=None*, *rug_kws=None*, *fit_kws=None*, *color=None*, *vertical=False*, *norm_hist=False*, *axlabel=None*, *label=None*, *ax=None*, *x=None*)

DEPRECATED: Flexibly plot a univariate distribution of observations.

> **Warning:** This function is deprecated and will be removed in a future version. Please adapt your code to use one of two new functions:
>
> - *`displot()`*, a figure-level function with a similar flexibility over the kind of plot to draw
> - *`histplot()`*, an axes-level function for plotting histograms, including with kernel density smoothing

This function combines the matplotlib `hist` function (with automatic calculation of a good default bin size) with the seaborn *`kdeplot()`* and *`rugplot()`* functions. It can also fit `scipy.stats` distributions and plot the estimated PDF over the data.

**Parameters**

**a** [Series, 1d-array, or list.] Observed data. If this is a Series object with a `name` attribute, the name will be used to label the data axis.

**bins** [argument for matplotlib hist(), or None, optional] Specification of hist bins. If unspecified, as reference rule is used that tries to find a useful default.

**hist** [bool, optional] Whether to plot a (normed) histogram.

**kde** [bool, optional] Whether to plot a gaussian kernel density estimate.

**rug** [bool, optional] Whether to draw a rugplot on the support axis.

**fit** [random variable object, optional] An object with `fit` method, returning a tuple that can be passed to a `pdf` method a positional arguments following a grid of values to evaluate the pdf on.

**hist_kws** [dict, optional] Keyword arguments for `matplotlib.axes.Axes.hist()`.

**kde_kws** [dict, optional] Keyword arguments for `kdeplot()`.

**rug_kws** [dict, optional] Keyword arguments for `rugplot()`.

**color** [matplotlib color, optional] Color to plot everything but the fitted curve in.

**vertical** [bool, optional] If True, observed values are on y-axis.

**norm_hist** [bool, optional] If True, the histogram height shows a density rather than a count. This is implied if a KDE or fitted density is plotted.

**axlabel** [string, False, or None, optional] Name for the support axis label. If None, will try to get it from a.name if False, do not set a label.

**label** [string, optional] Legend label for the relevant component of the plot.

**ax** [matplotlib axis, optional] If provided, plot on this axis.

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot for further tweaking.

**See also:**

`kdeplot` Show a univariate or bivariate distribution with a kernel density estimate.

`rugplot` Draw small vertical lines to show each observation in a distribution.

## Examples

Show a default plot with a kernel density estimate and histogram with bin size determined automatically with a reference rule:

```
>>> import seaborn as sns, numpy as np
>>> sns.set_theme(); np.random.seed(0)
>>> x = np.random.randn(100)
>>> ax = sns.distplot(x)
```

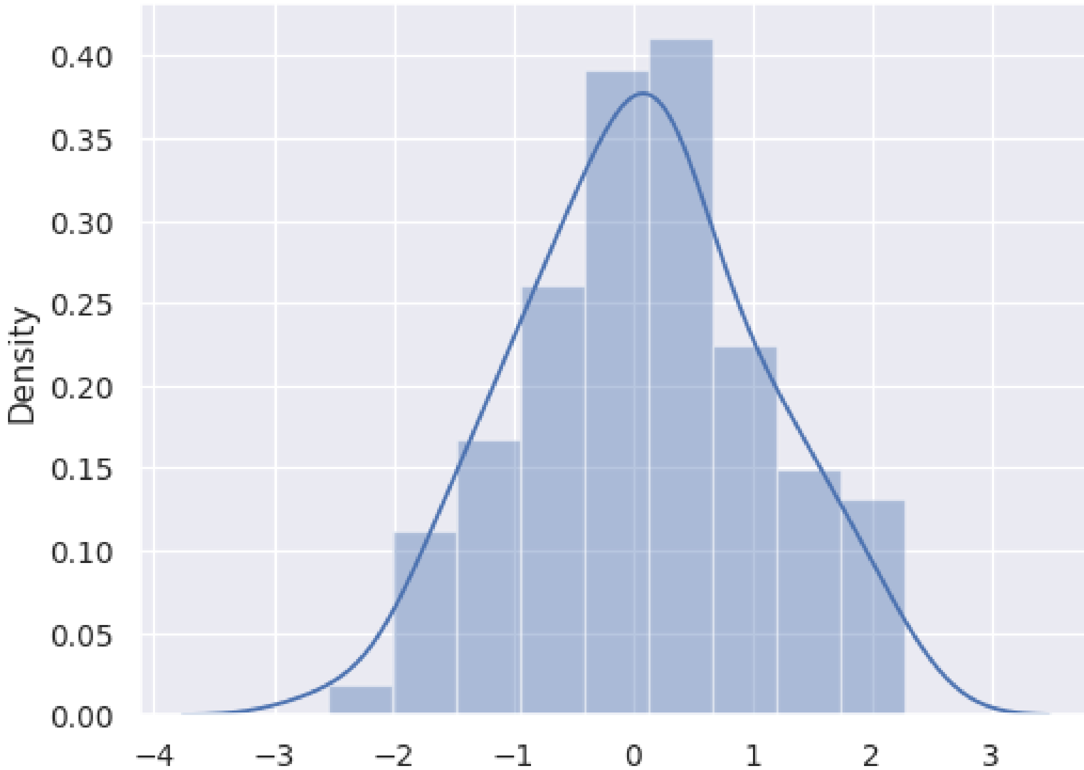
Use Pandas objects to get an informative axis label:

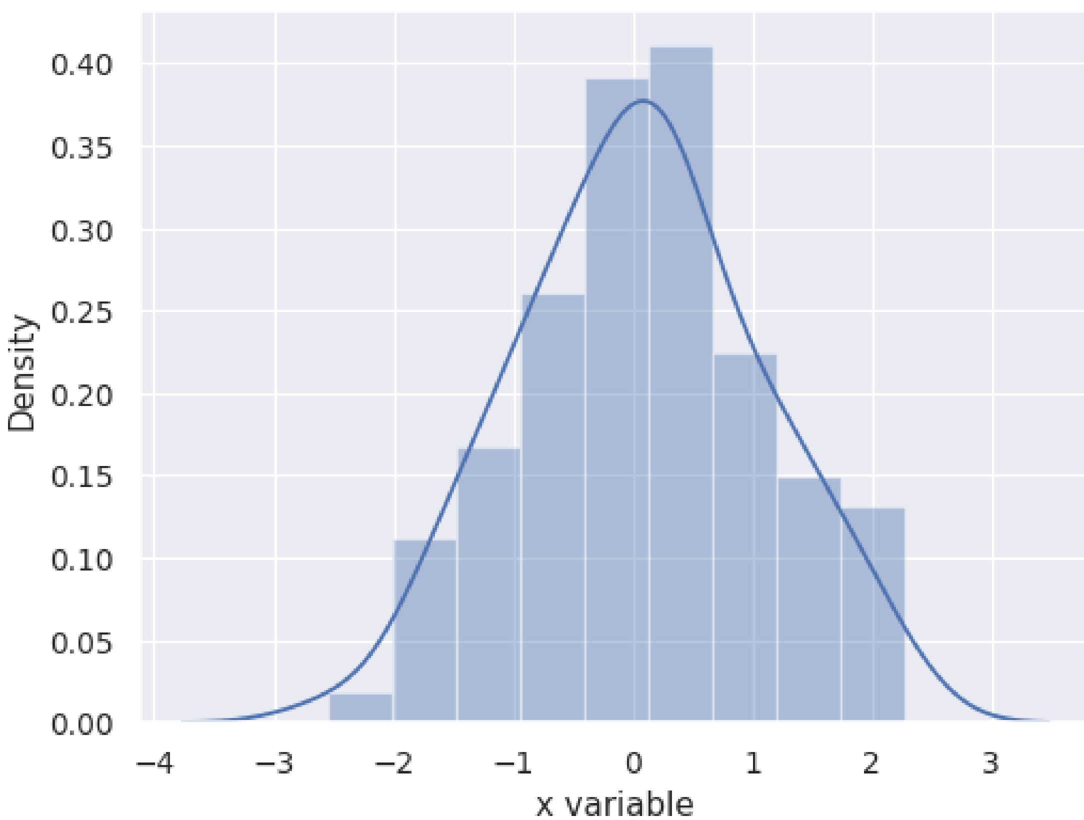
```
>>> import pandas as pd
>>> x = pd.Series(x, name="x variable")
>>> ax = sns.distplot(x)
```

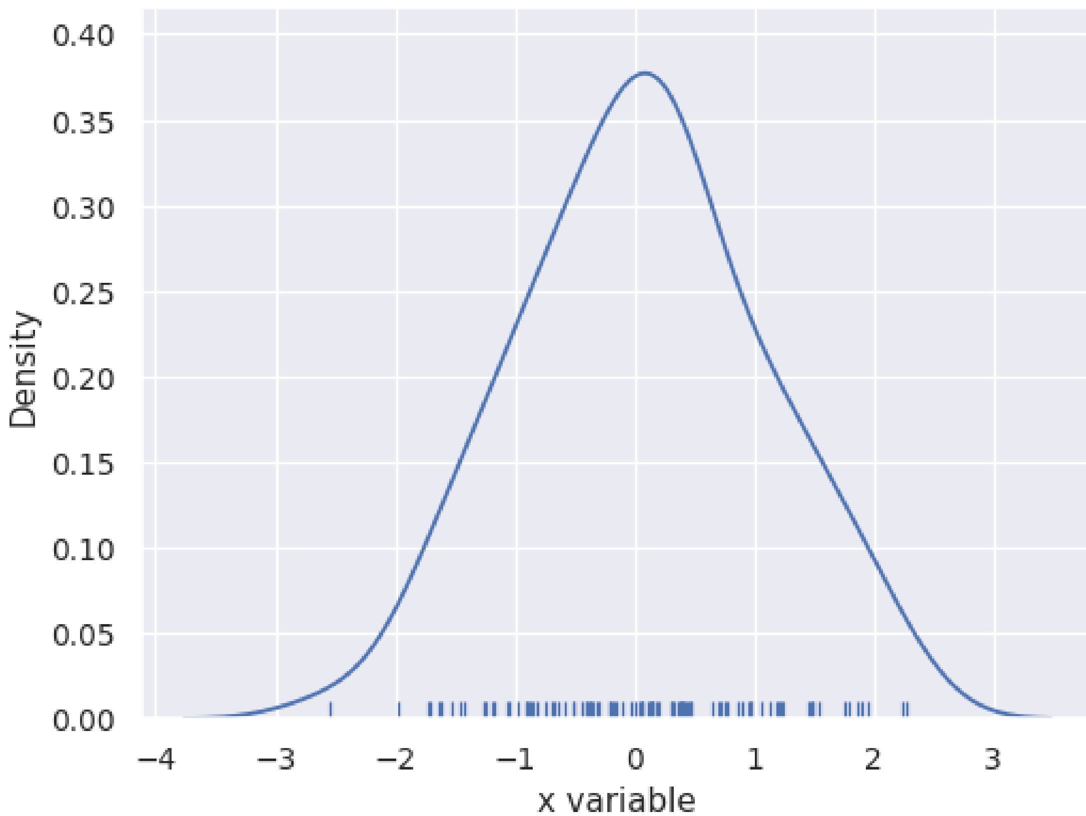
Plot the distribution with a kernel density estimate and rug plot:

```
>>> ax = sns.distplot(x, rug=True, hist=False)
```

Plot the distribution with a histogram and maximum likelihood gaussian distribution fit:


Density
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4
−3
−2
−1
0
1
2
3


Density
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4
−3
−2
−1
0
1
2
3
x variable




```
>>> from scipy.stats import norm
>>> ax = sns.distplot(x, fit=norm, kde=False)
```

Plot the distribution on the vertical axis:

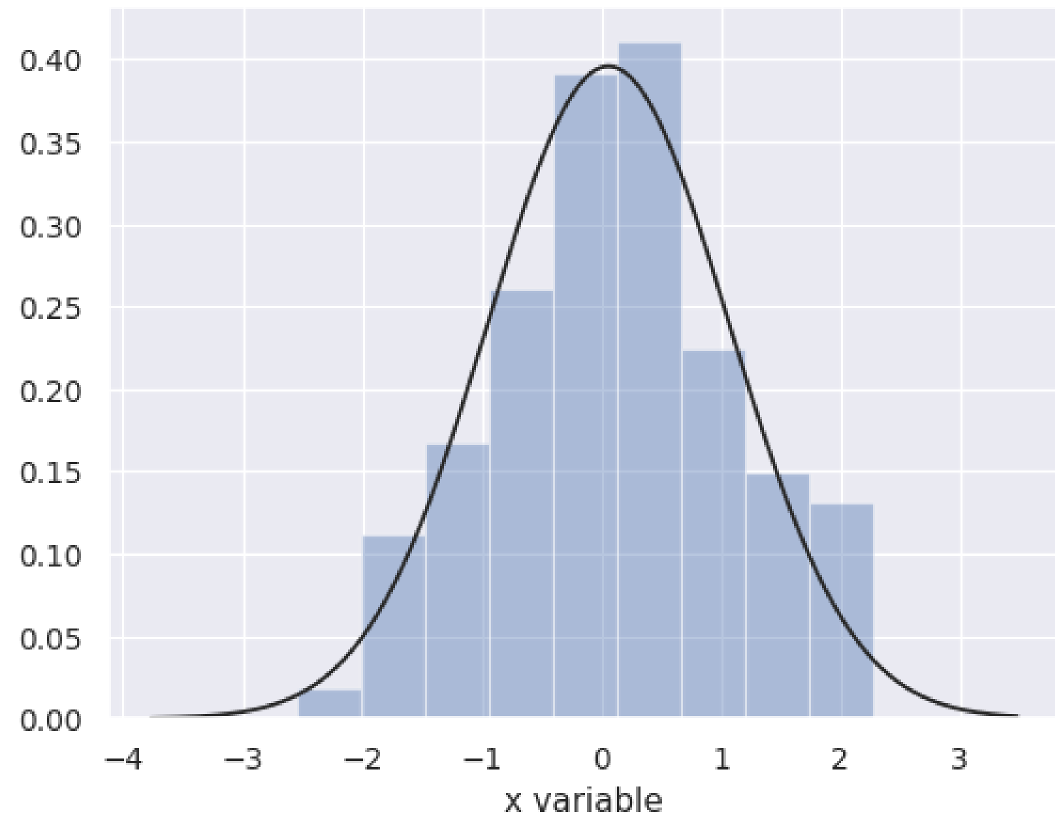
```
>>> ax = sns.distplot(x, vertical=True)
```

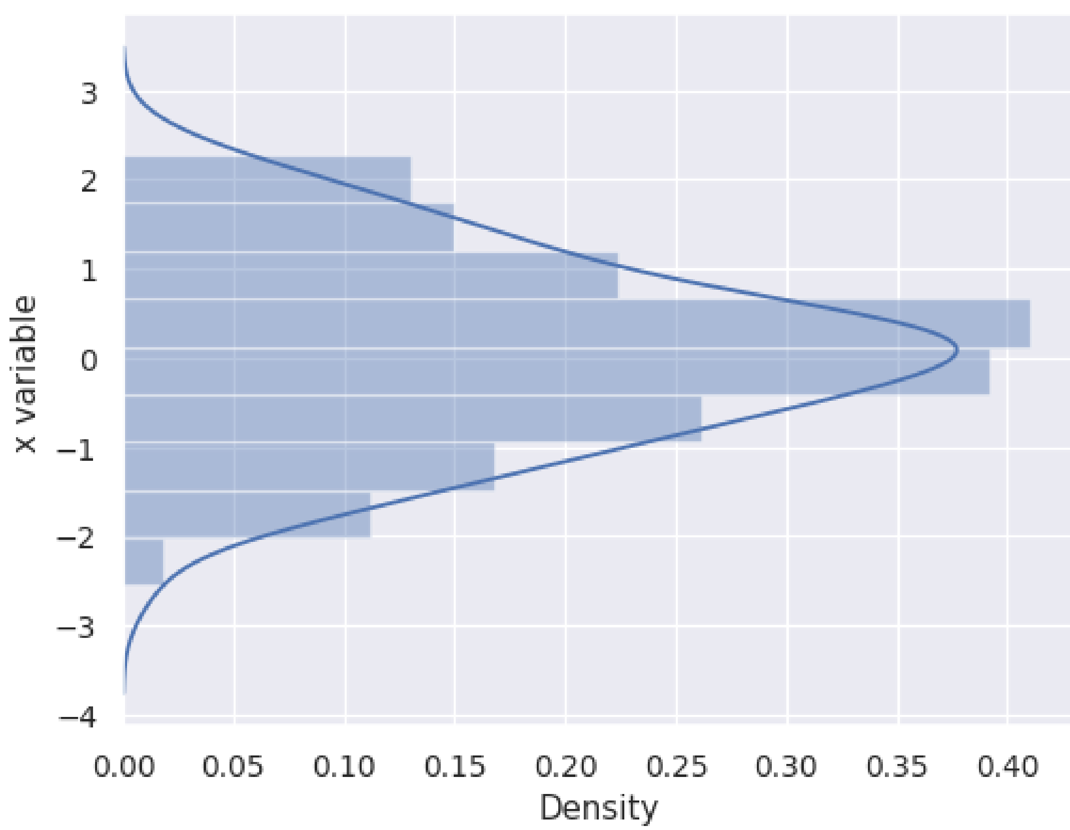
Change the color of all the plot elements:

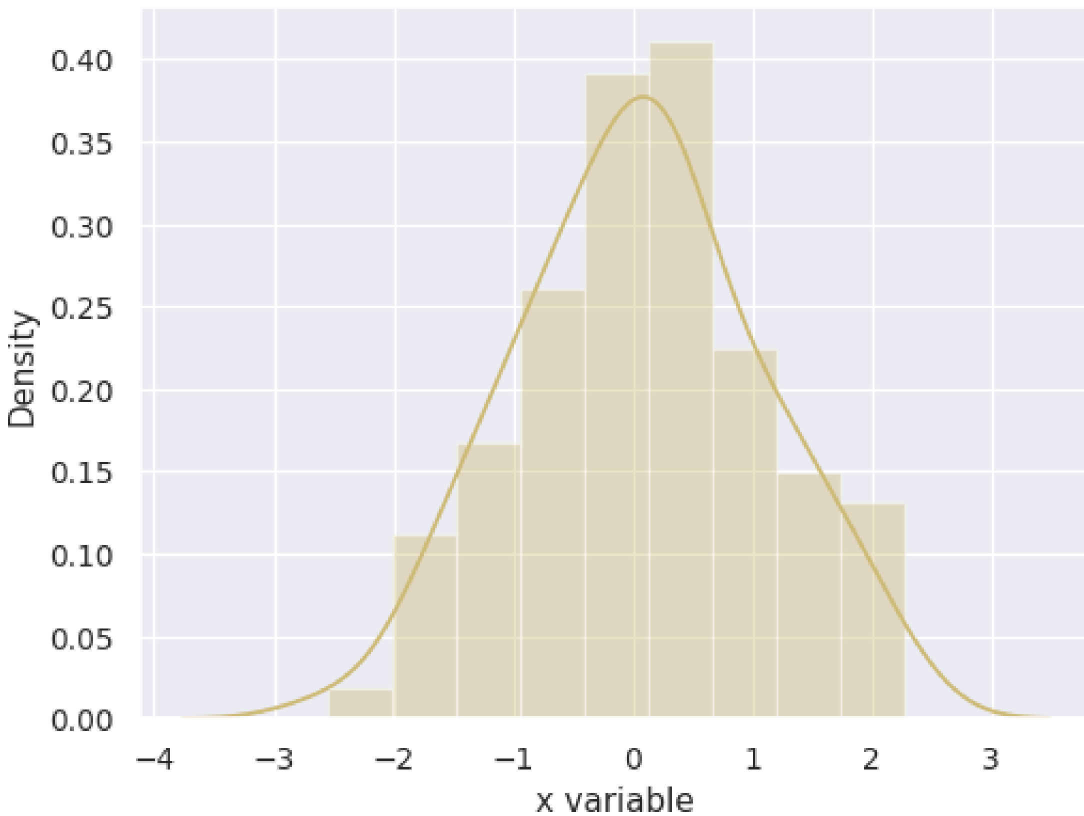
```
>>> sns.set_color_codes()
>>> ax = sns.distplot(x, color="y")
```

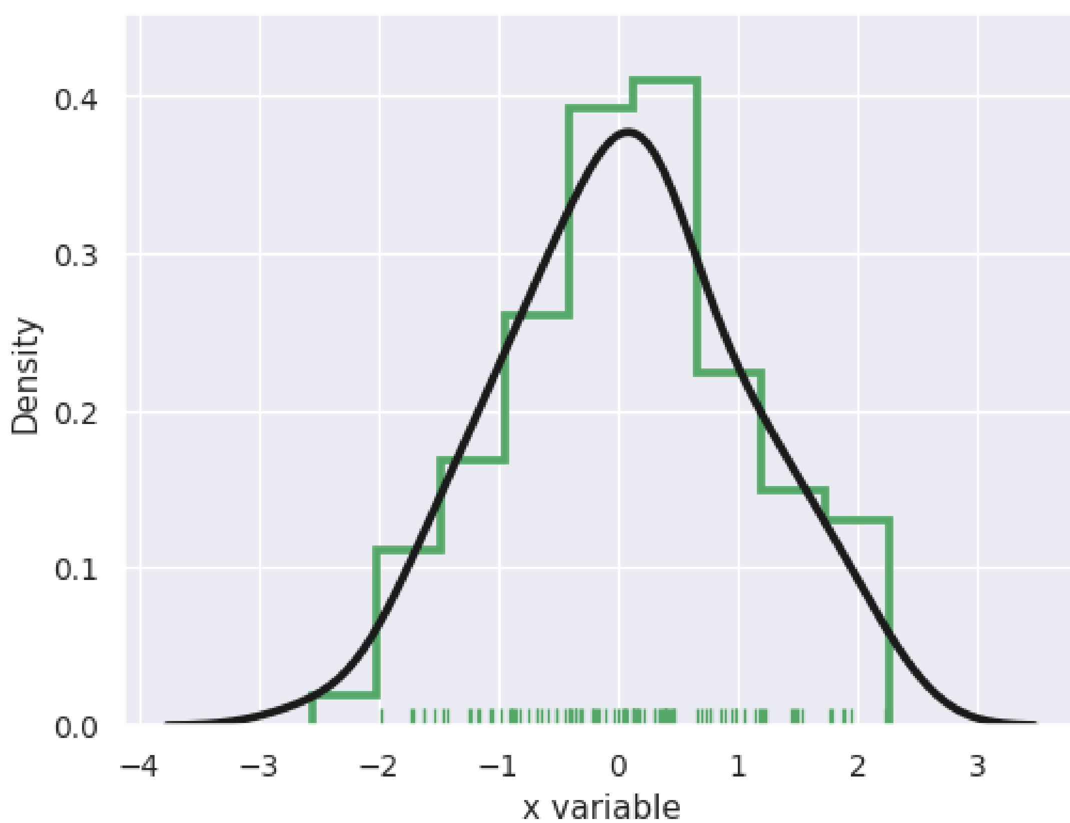
Pass specific parameters to the underlying plot functions:

```
>>> ax = sns.distplot(x, rug=True, rug_kws={"color": "g"},
...                   kde_kws={"color": "k", "lw": 3, "label": "KDE"},
...                   hist_kws={"histtype": "step", "linewidth": 3,
...                             "alpha": 1, "color": "g"})
```


0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−4
−3
−2
−1
0
1
2
3
x variable


3
2
1
0
−1
−2
−3
−4
x variable
0.00
0.05
0.10
0.15
0.20
0.25
0.30
0.35
0.40
Density


0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
Density
−4
−3
−2
−1
0
1
2
3
x variable


0.4
0.3
0.2
0.1
0.0
Density
−4
−3
−2
−1
0
1
2
3
x variable

## 5.3 Categorical plots

| | |
|---|---|
| *catplot* | Figure-level interface for drawing categorical plots onto a FacetGrid. |
| *stripplot* | Draw a scatterplot where one variable is categorical. |
| *swarmplot* | Draw a categorical scatterplot with non-overlapping points. |
| *boxplot* | Draw a box plot to show distributions with respect to categories. |
| *violinplot* | Draw a combination of boxplot and kernel density estimate. |
| *boxenplot* | Draw an enhanced box plot for larger datasets. |
| *pointplot* | Show point estimates and confidence intervals using scatter plot glyphs. |
| *barplot* | Show point estimates and confidence intervals as rectangular bars. |
| *countplot* | Show the counts of observations in each categorical bin using bars. |

### 5.3.1 seaborn.catplot

seaborn.**catplot**(**, x=None, y=None, hue=None, data=None, row=None, col=None, col_wrap=None, estimator=<function mean>, ci=95, n_boot=1000, units=None, seed=None, order=None, hue_order=None, row_order=None, col_order=None, kind='strip', height=5, aspect=1, orient=None, color=None, palette=None, legend=True, legend_out=True, sharex=True, sharey=True, margin_titles=False, facet_kws=None, **kwargs*)

Figure-level interface for drawing categorical plots onto a FacetGrid.

This function provides access to several axes-level functions that show the relationship between a numerical and one or more categorical variables using one of several visual representations. The `kind` parameter selects the underlying axes-level function to use:

Categorical scatterplots:

- `stripplot()` (with `kind="strip"`; the default)
- `swarmplot()` (with `kind="swarm"`)

Categorical distribution plots:

- `boxplot()` (with `kind="box"`)
- `violinplot()` (with `kind="violin"`)
- `boxenplot()` (with `kind="boxen"`)

Categorical estimate plots:

- `pointplot()` (with `kind="point"`)
- `barplot()` (with `kind="bar"`)
- `countplot()` (with `kind="count"`)

Extra keyword arguments are passed to the underlying function, so you should refer to the documentation for each to see kind-specific options.

Note that unlike when using the axes-level functions directly, data must be passed in a long-form DataFrame with variables specified by passing strings to `x`, `y`, `hue`, etc.

As in the case with the underlying plot functions, if variables have a `categorical` data type, the levels of the categorical variables, and their order will be inferred from the objects. Otherwise you may have to use alter the dataframe sorting or use the function parameters (`orient`, `order`, `hue_order`, etc.) to set up the plot correctly.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, . . . n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

After plotting, the *`FacetGrid`* with the plot is returned and can be used directly to tweak supporting plot details or add other layers.

**Parameters**

**x, y, hue** [names of variables in `data`] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame] Long-form (tidy) dataset for plotting. Each column should correspond to a variable, and each row should correspond to an observation.

**row, col** [names of variables in `data`, optional] Categorical variables that will determine the faceting of the grid.

**col_wrap** [int] "Wrap" the column variable at this width, so that the column facets span multiple rows. Incompatible with a `row` facet.

**estimator** [callable that maps vector -> scalar, optional] Statistical function to estimate within each categorical bin.

**ci** [float or "sd" or None, optional] Size of confidence intervals to draw around estimated values. If "sd", skip bootstrapping and draw the standard deviation of the observations. If `None`, no bootstrapping will be performed, and error bars will not be drawn.

**n_boot** [int, optional] Number of bootstrap iterations to use when computing confidence intervals.

**units** [name of variable in `data` or vector data, optional] Identifier of sampling units, which will be used to perform a multilevel bootstrap and account for repeated measures design.

**seed** [int, numpy.random.Generator, or numpy.random.RandomState, optional] Seed or random number generator for reproducible bootstrapping.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**row_order, col_order** [lists of strings, optional] Order to organize the rows and/or columns of the grid in, otherwise the orders are inferred from the data objects.

**kind** [str, optional] The kind of plot to draw, corresponds to the name of a categorical axes-level plotting function. Options are: "strip", "swarm", "box", "violin", "boxen", "point", "bar", or "count".

**height** [scalar] Height (in inches) of each facet. See also: `aspect`.

**aspect** [scalar] Aspect ratio of each facet, so that `aspect * height` gives the width of each facet in inches.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both `x` and `y` are numeric or when plotting wide-form data.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by *color_palette()*, or a dictionary mapping hue levels to matplotlib colors.

**legend** [bool, optional] If `True` and there is a `hue` variable, draw a legend on the plot.

**legend_out** [bool] If `True`, the figure size will be extended, and the legend will be drawn outside the plot on the center right.

**share{x,y}** [bool, 'col', or 'row' optional] If true, the facets will share y axes across columns and/or x axes across rows.

**margin_titles** [bool] If `True`, the titles for the row variable are drawn to the right of the last column. This option is experimental and may not work in all cases.

**facet_kws** [dict, optional] Dictionary of other keyword arguments to pass to *FacetGrid*.

**kwargs** [key, value pairings] Other keyword arguments are passed through to the underlying plotting function.

**Returns**

**g** [*FacetGrid*] Returns the *FacetGrid* object with the plot on it for further tweaking.

## Examples

Draw a single facet to use the *FacetGrid* legend placement:

```
>>> import seaborn as sns
>>> sns.set_theme(style="ticks")
>>> exercise = sns.load_dataset("exercise")
>>> g = sns.catplot(x="time", y="pulse", hue="kind", data=exercise)
```

Use a different plot kind to visualize the same data:

```
>>> g = sns.catplot(x="time", y="pulse", hue="kind",
...                data=exercise, kind="violin")
```

Facet along the columns to show a third categorical variable:

```
>>> g = sns.catplot(x="time", y="pulse", hue="kind",
...                 col="diet", data=exercise)
```

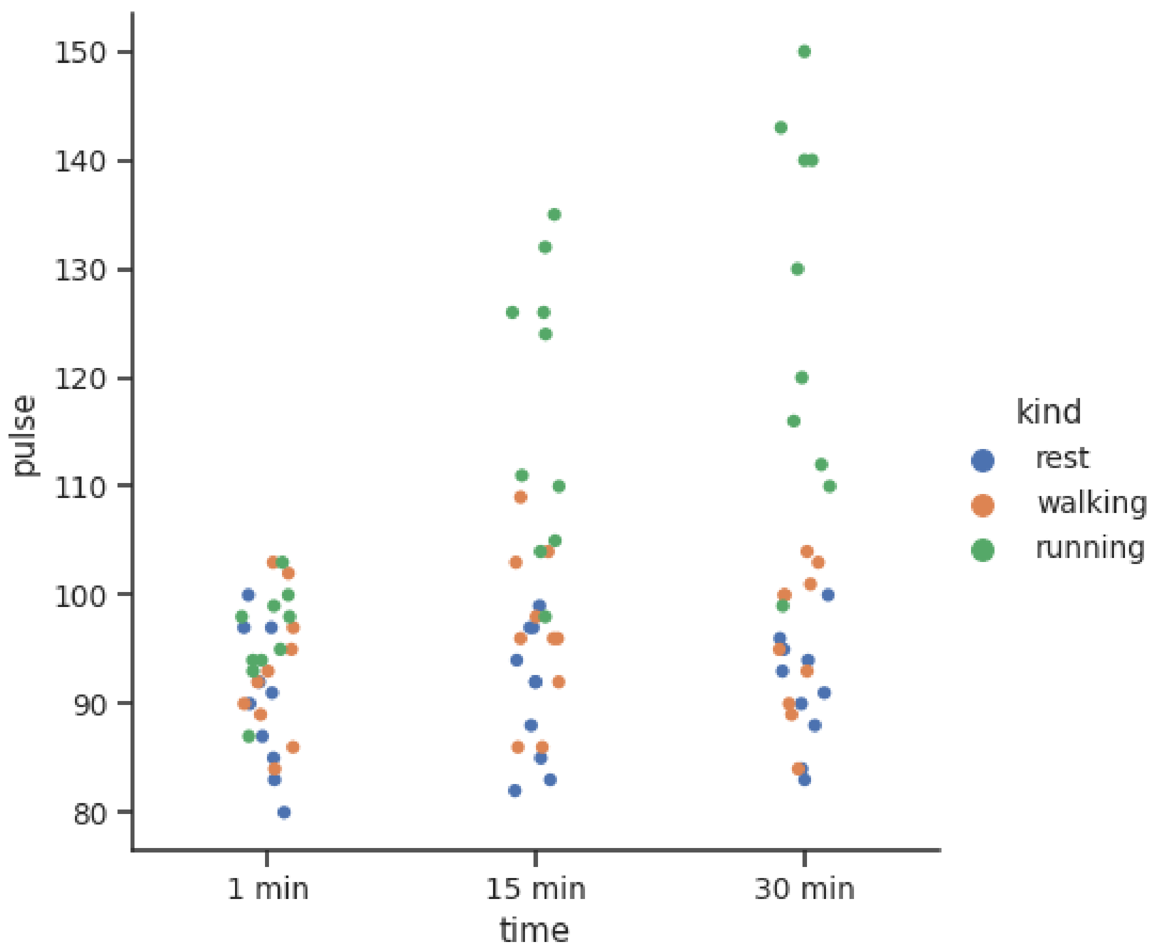
Use a different height and aspect ratio for the facets:

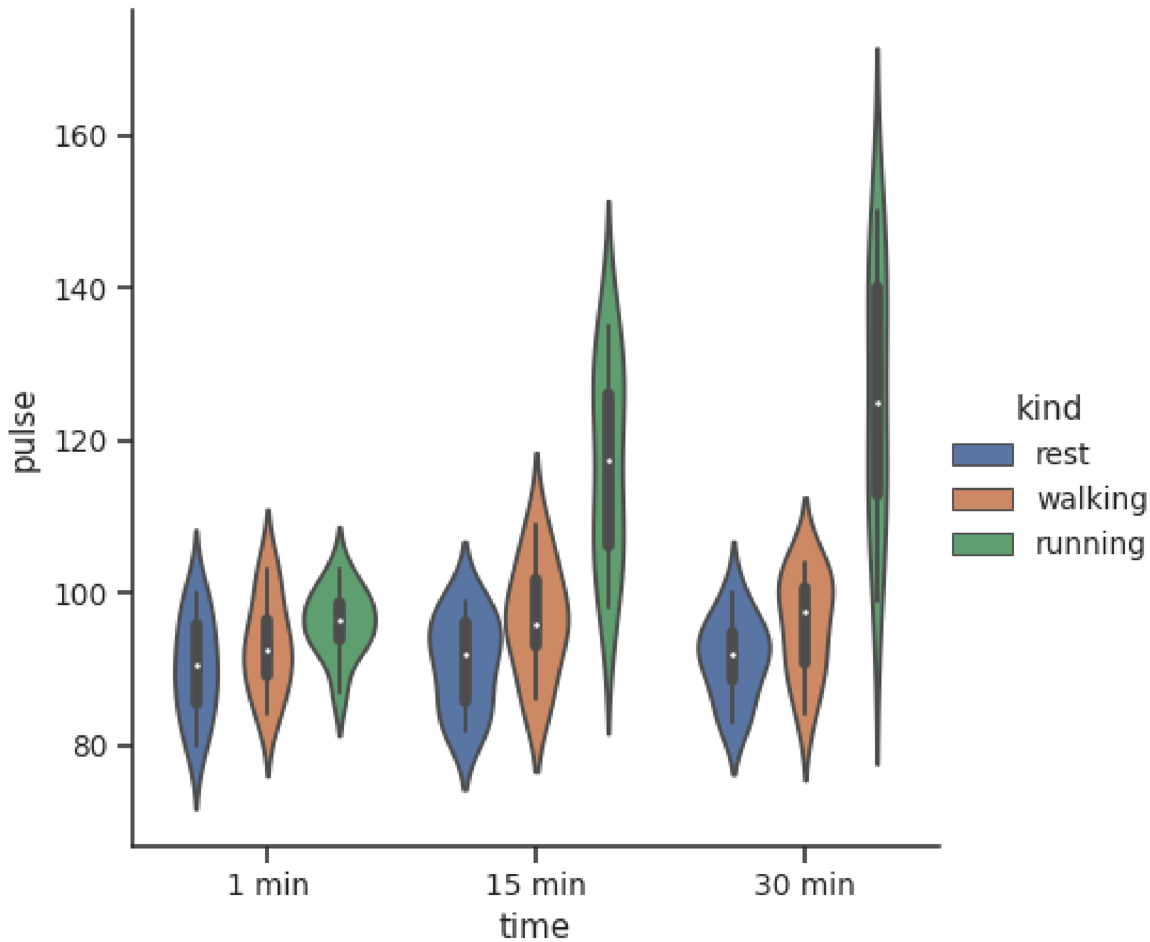
```
>>> g = sns.catplot(x="time", y="pulse", hue="kind",
...                 col="diet", data=exercise,
...                 height=5, aspect=.8)
```

Make many column facets and wrap them into the rows of the grid:

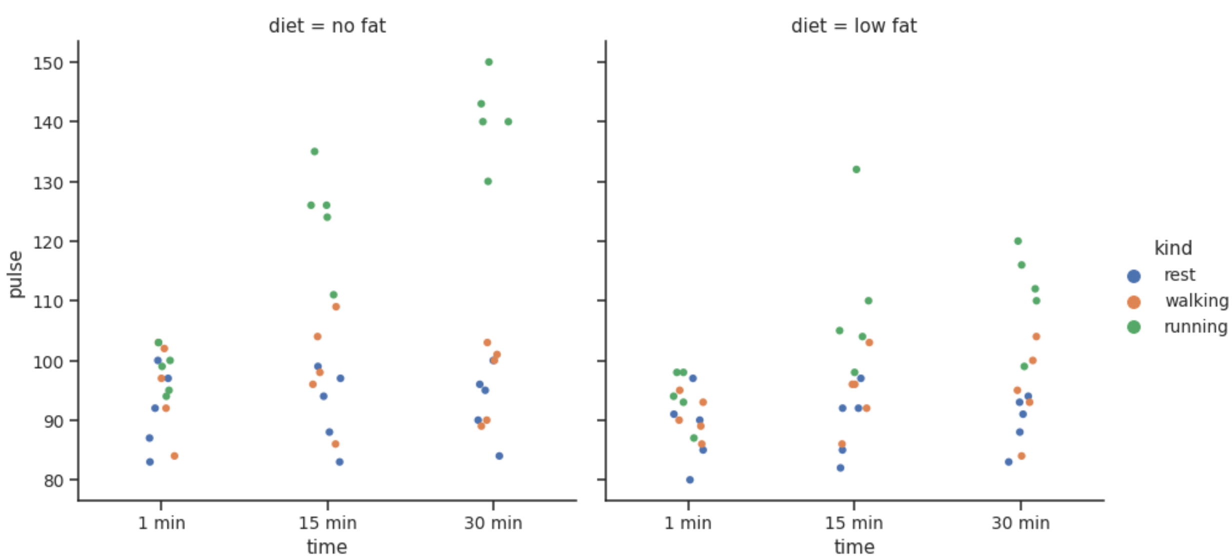
```
>>> titanic = sns.load_dataset("titanic")
>>> g = sns.catplot(x="alive", col="deck", col_wrap=4,
...                 data=titanic[titanic.deck.notnull()],
...                 kind="count", height=2.5, aspect=.8)
```

Plot horizontally and pass other keyword arguments to the plot function:

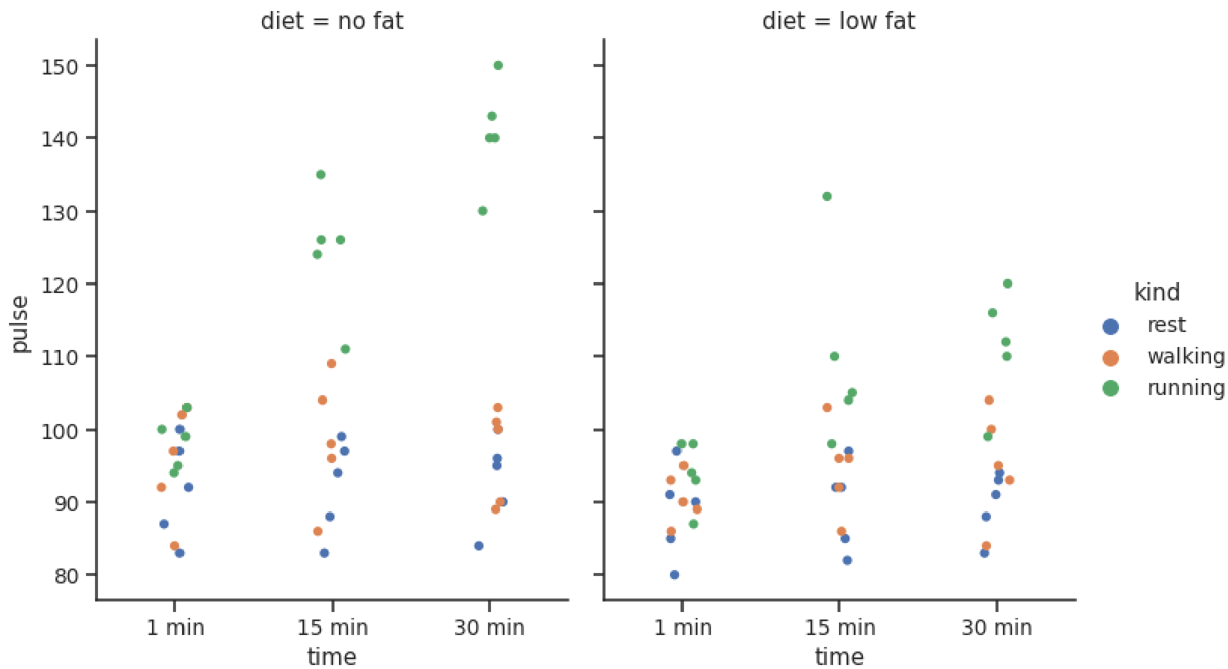

150
140
130
120
110
100
90
80
pulse
1 min
15 min
30 min
time
kind
rest
walking
running

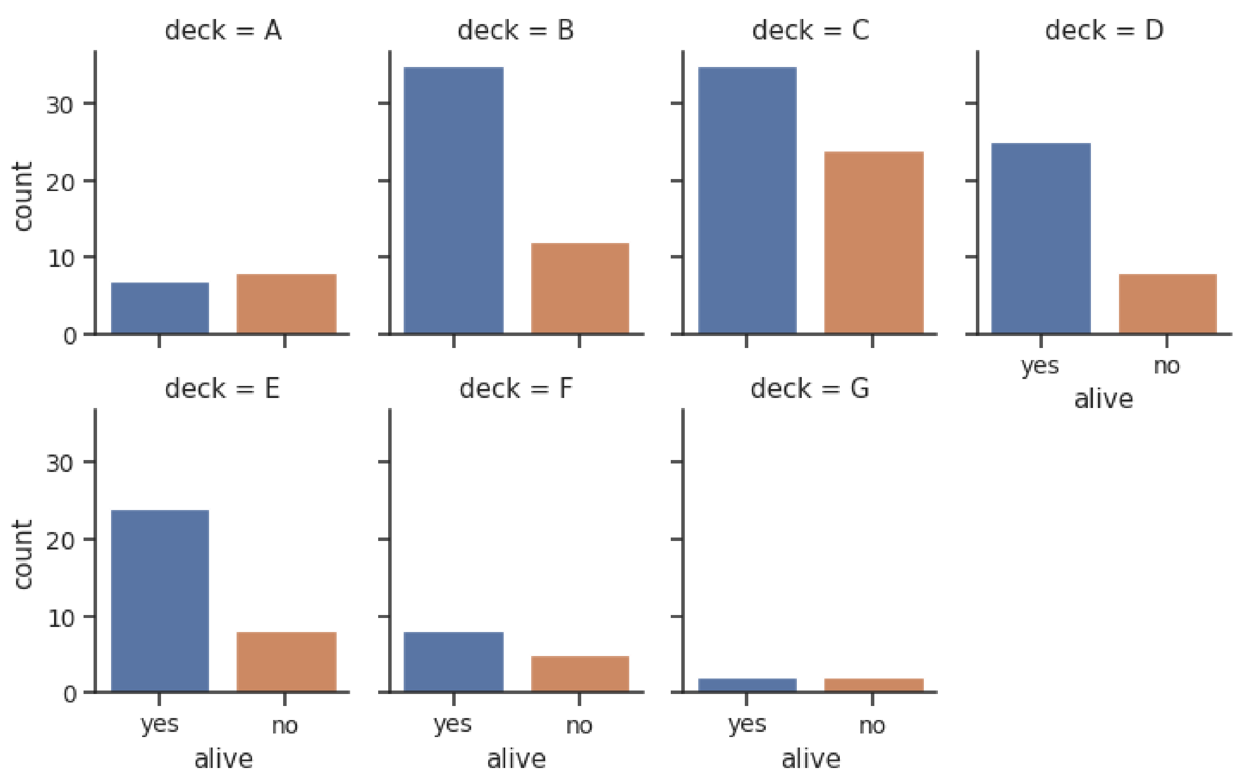


pulse
160
140
120
100
80
1 min
15 min
30 min
time
kind
rest
walking
running

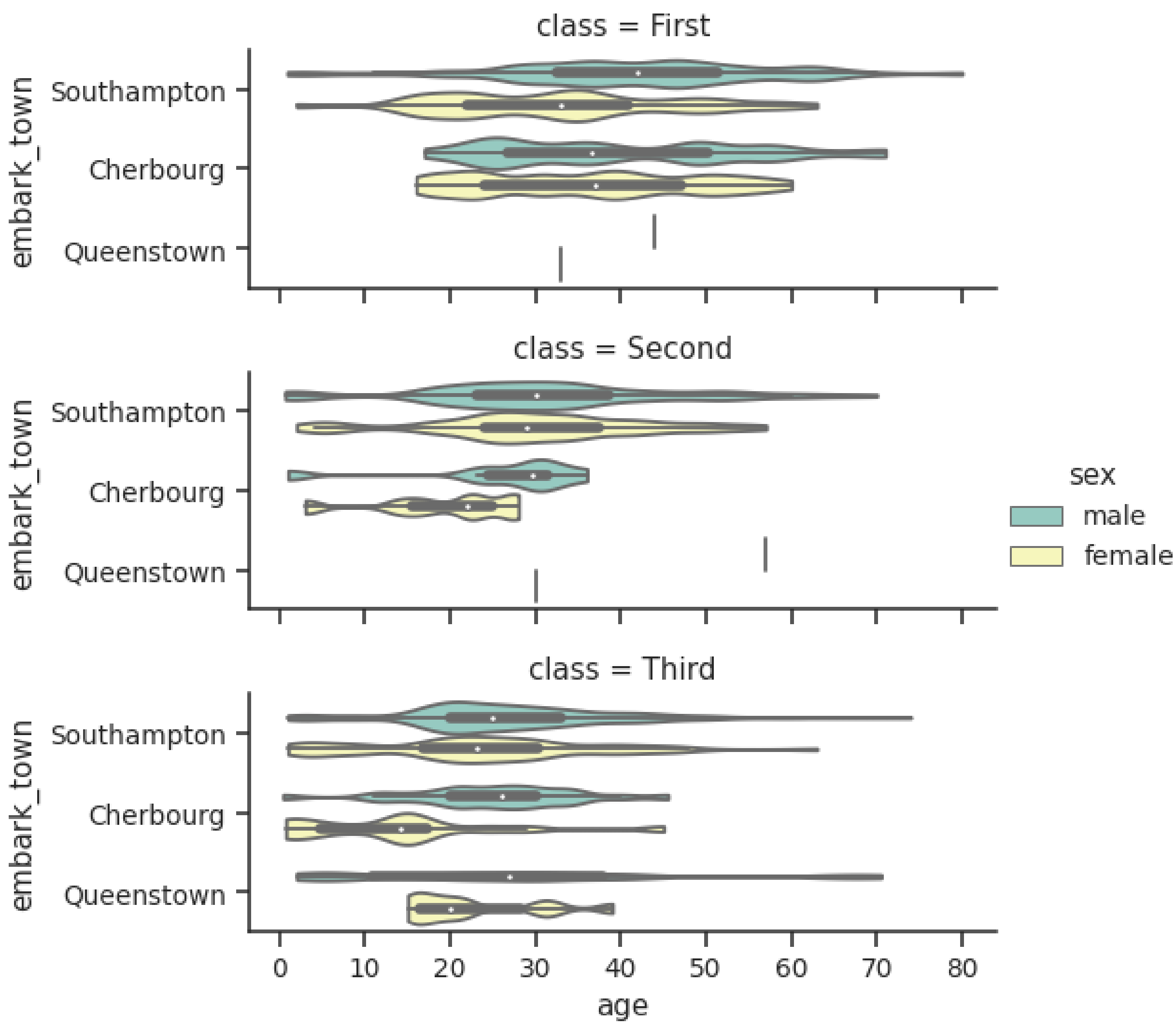



diet = no fat
diet = low fat
pulse
150
140
130
120
110
100
90
80
1 min
15 min
30 min
time
kind
rest
walking
running


diet = no fat
diet = low fat
pulse
150
140
130
120
110
100
90
80
1 min
15 min
30 min
time
kind
rest
walking
running

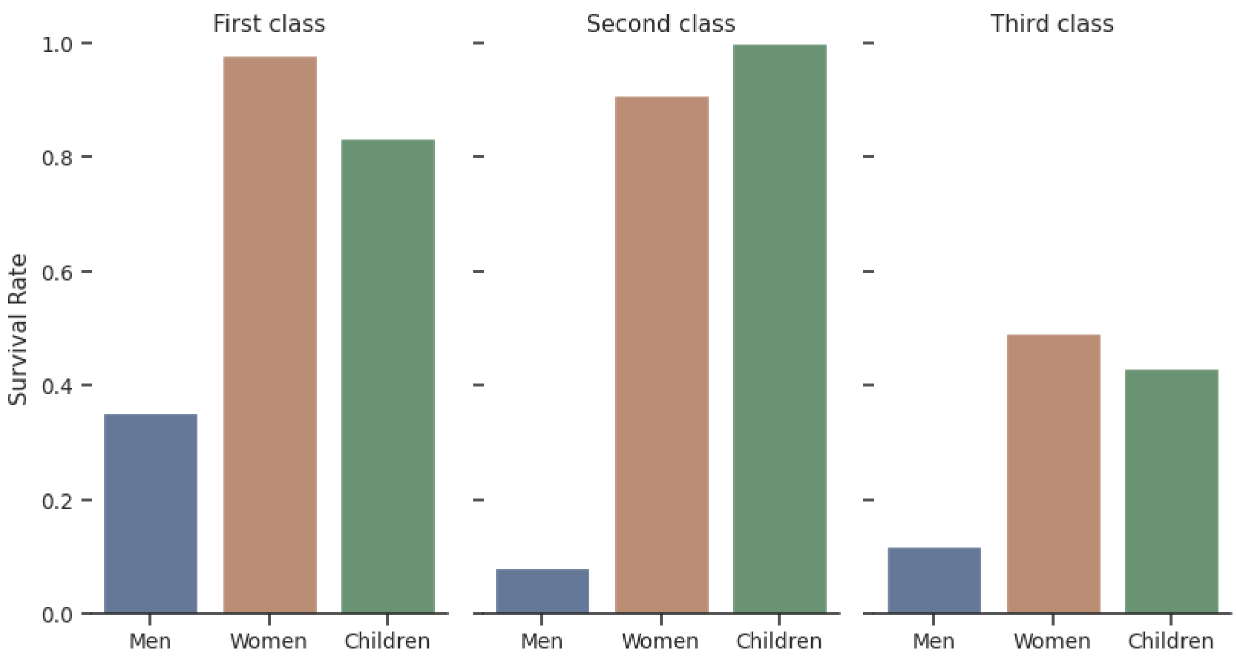

deck = A
deck = B
deck = C
deck = D
deck = E
deck = F
deck = G
count
30
20
10
0
yes
no
alive

```
>>> g = sns.catplot(x="age", y="embark_town",
...                 hue="sex", row="class",
...                 data=titanic[titanic.embark_town.notnull()],
...                 orient="h", height=2, aspect=3, palette="Set3",
...                 kind="violin", dodge=True, cut=0, bw=.2)
```




Use methods on the returned *FacetGrid* to tweak the presentation:

```
>>> g = sns.catplot(x="who", y="survived", col="class",
...                 data=titanic, saturation=.5,
...                 kind="bar", ci=None, aspect=.6)
>>> (g.set_axis_labels("", "Survival Rate")
...   .set_xticklabels(["Men", "Women", "Children"])
...   .set_titles("{col_name} {col_var}")
...   .set(ylim=(0, 1))
...   .despine(left=True))
<seaborn.axisgrid.FacetGrid object at 0x...>
```




### 5.3.2 seaborn.stripplot

seaborn.**stripplot**(**, x=None, y=None, hue=None, data=None, order=None, hue_order=None, jitter=True, dodge=False, orient=None, color=None, palette=None, size=5, edgecolor='gray', linewidth=0, ax=None, **kwargs*)

Draw a scatterplot where one variable is categorical.

A strip plot can be drawn on its own, but it is also a good complement to a box or violin plot in cases where you want to show all observations along with some representation of the underlying distribution.

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`, and/or `hue` parameters.
- A "long-form" DataFrame, in which case the `x`, `y`, and `hue` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, … n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

**Parameters**

**x, y, hue** [names of variables in `data` or vector data, optional] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame, array, or list of arrays, optional] Dataset for plotting. If `x` and `y` are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**jitter** [float, `True`/`1` is special-cased, optional] Amount of jitter (only along the categorical axis) to apply. This can be useful when you have many points and they overlap, so that it is easier to see the distribution. You can specify the amount of jitter (half the width of the uniform random variable support), or just use `True` for a good default.

**dodge** [bool, optional] When using `hue` nesting, setting this to `True` will separate the strips for different hue levels along the categorical axis. Otherwise, the points for each level will be plotted on top of each other.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both `x` and `y` are numeric or when plotting wide-form data.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by `color_palette()`, or a dictionary mapping hue levels to matplotlib colors.

**size** [float, optional] Radius of the markers, in points.

**edgecolor** [matplotlib color, "gray" is special-cased, optional] Color of the lines around each point. If you pass `"gray"`, the brightness is determined by the color palette used for the body of the points.

**linewidth** [float, optional] Width of the gray lines that frame the plot elements.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**kwargs** [key, value mappings] Other keyword arguments are passed through to `matplotlib.axes.Axes.scatter()`.

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot drawn onto it.

**See also:**

**`swarmplot`** A categorical scatterplot where the points do not overlap. Can be used with other plots to show each observation.

**`boxplot`** A traditional box-and-whisker plot with a similar API.

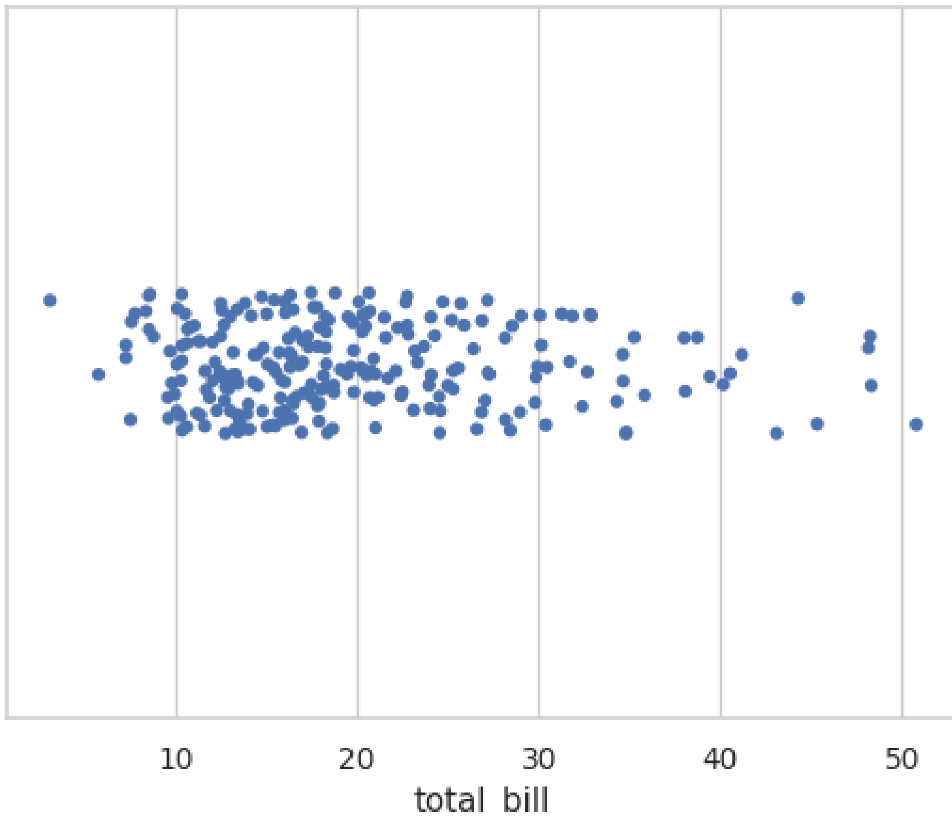
**`violinplot`** A combination of boxplot and kernel density estimation.

**`catplot`** Combine a categorical plot with a `FacetGrid`.

### Examples

Draw a single horizontal strip plot:

```
>>> import seaborn as sns
>>> sns.set_theme(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.stripplot(x=tips["total_bill"])
```




Group the strips by a categorical variable:

```
>>> ax = sns.stripplot(x="day", y="total_bill", data=tips)
```

Use a smaller amount of jitter:

```
>>> ax = sns.stripplot(x="day", y="total_bill", data=tips, jitter=0.05)
```

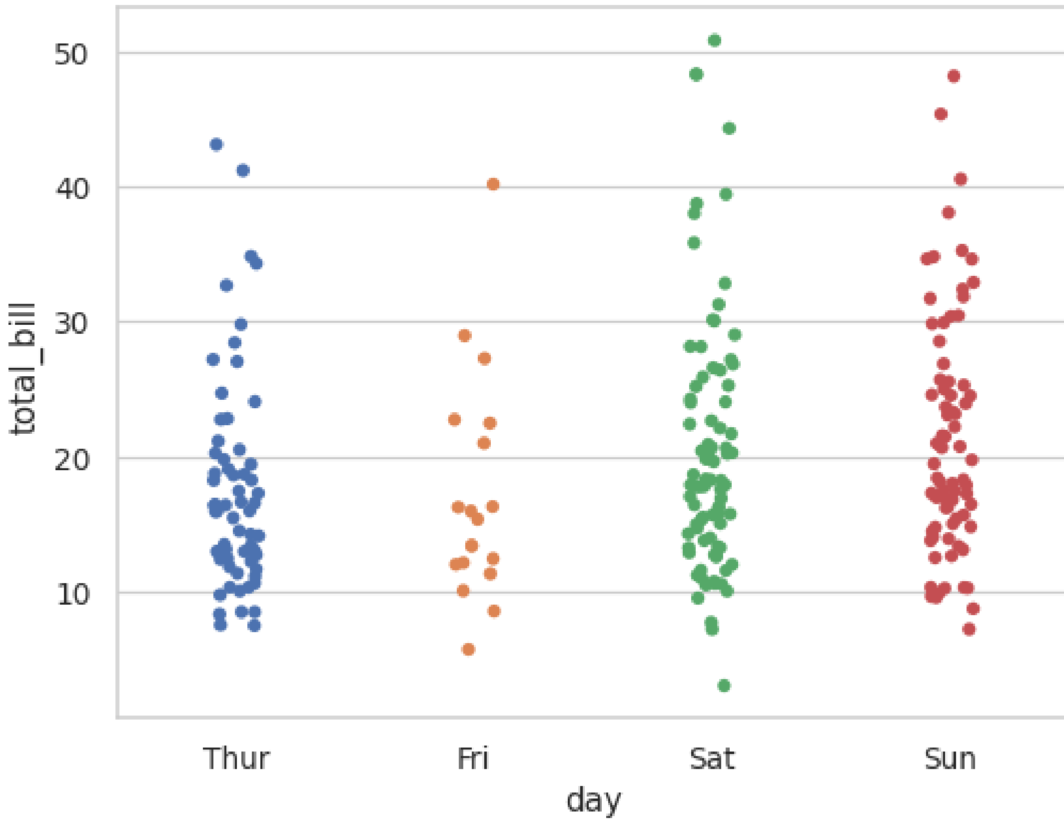
Draw horizontal strips:

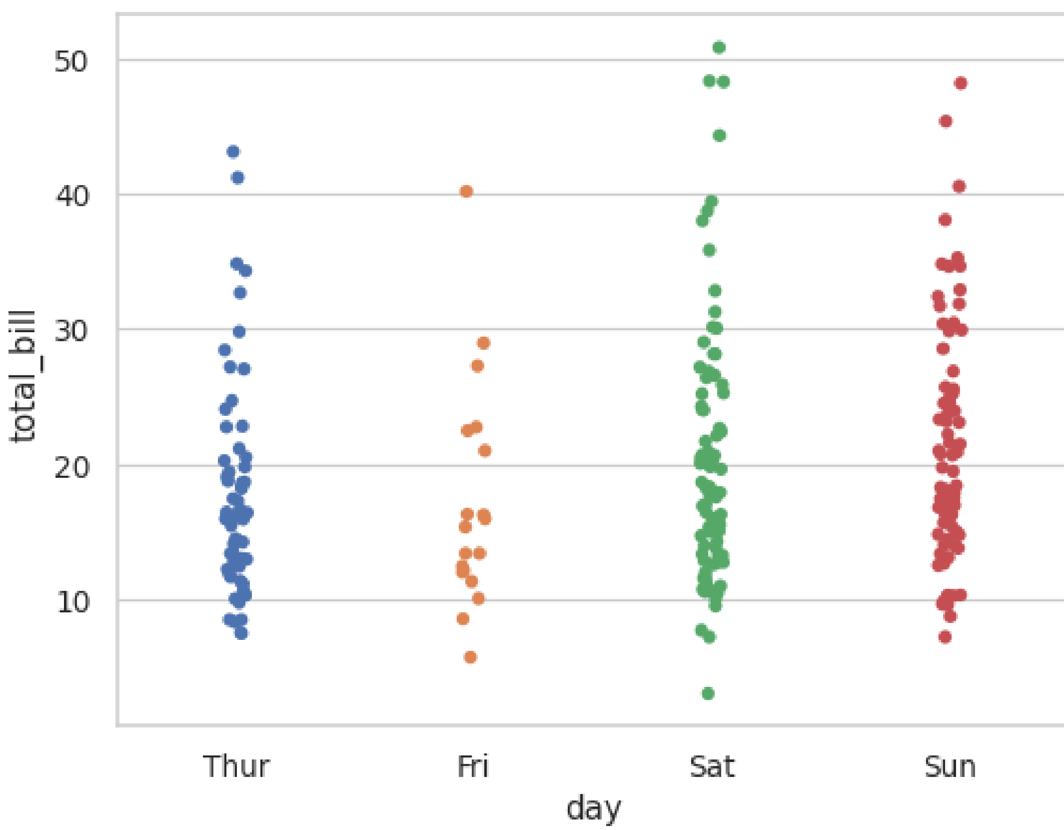
```
>>> ax = sns.stripplot(x="total_bill", y="day", data=tips)
```

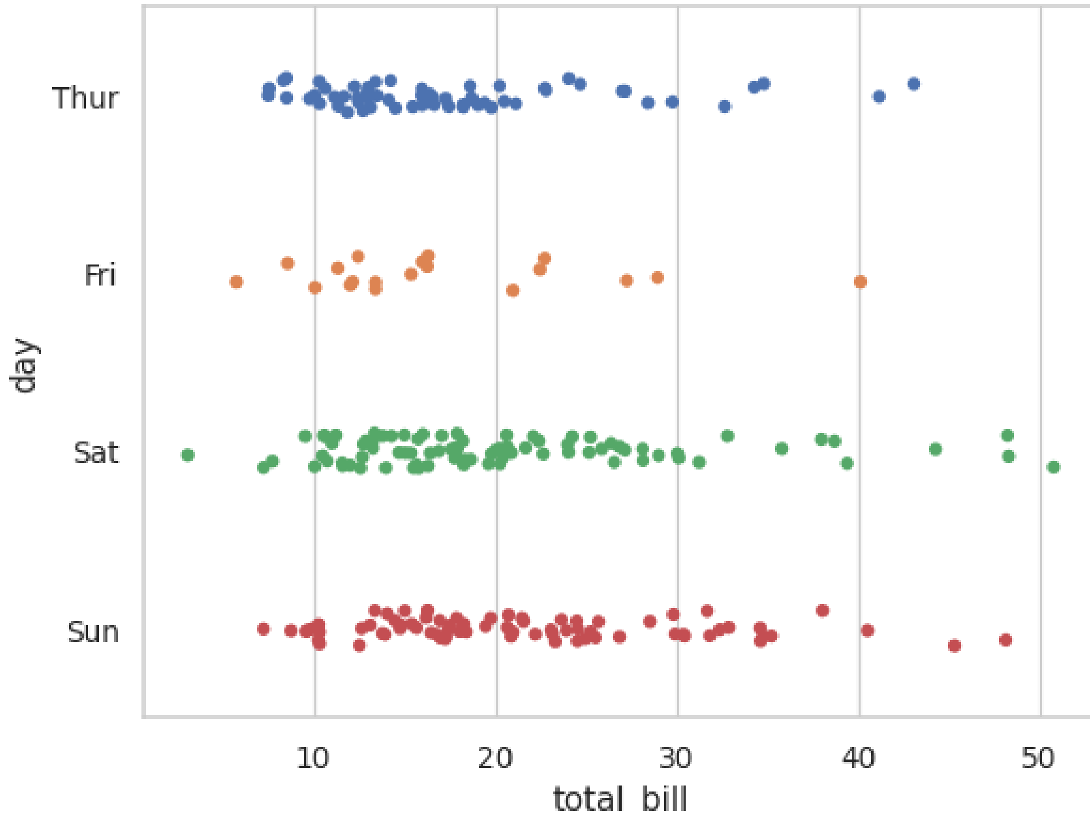
Draw outlines around the points:

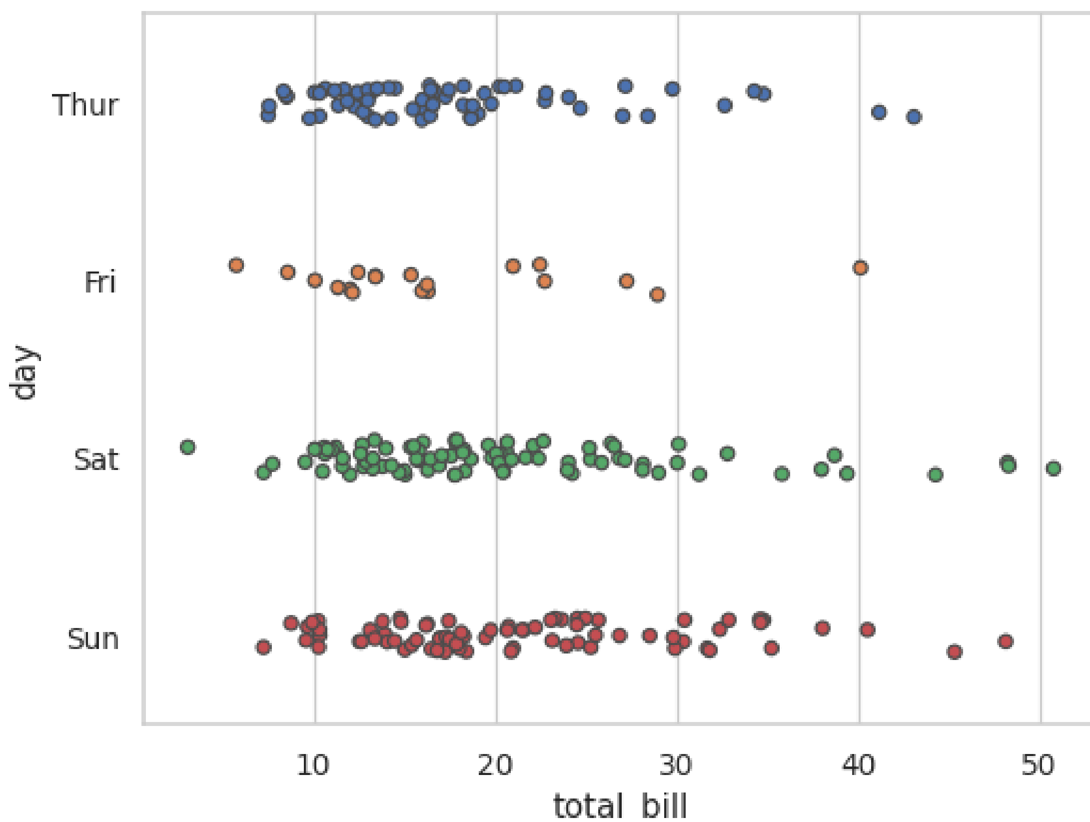
```
>>> ax = sns.stripplot(x="total_bill", y="day", data=tips,
...                    linewidth=1)
```

Nest the strips within a second categorical variable:

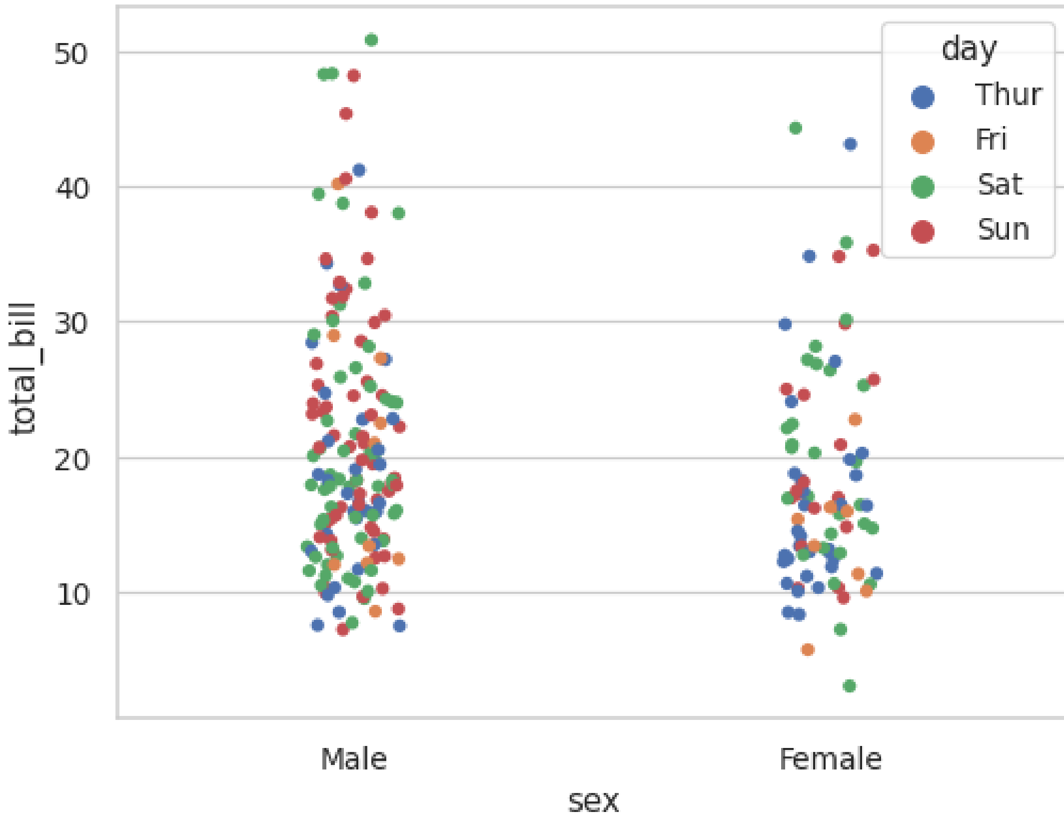

50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day


50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day


Thur
Fri
Sat
Sun
day
10
20
30
40
50
total_bill


Thur
Fri
Sat
Sun
day
10
20
30
40
50
total_bill

```
>>> ax = sns.stripplot(x="sex", y="total_bill", hue="day", data=tips)
```




Draw each level of the hue variable at different locations on the major categorical axis:

```
>>> ax = sns.stripplot(x="day", y="total_bill", hue="smoker",
...                    data=tips, palette="Set2", dodge=True)
```

Control strip order by passing an explicit order:

```
>>> ax = sns.stripplot(x="time", y="tip", data=tips,
...                    order=["Dinner", "Lunch"])
```

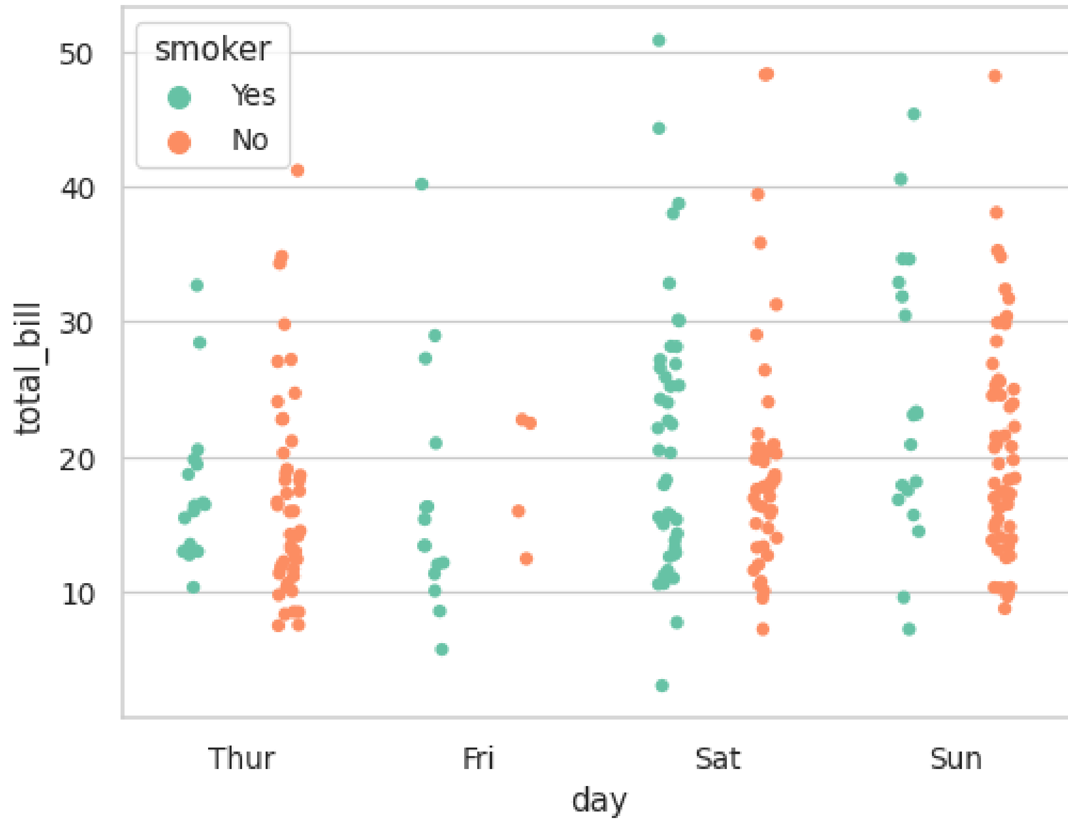
Draw strips with large points and different aesthetics:

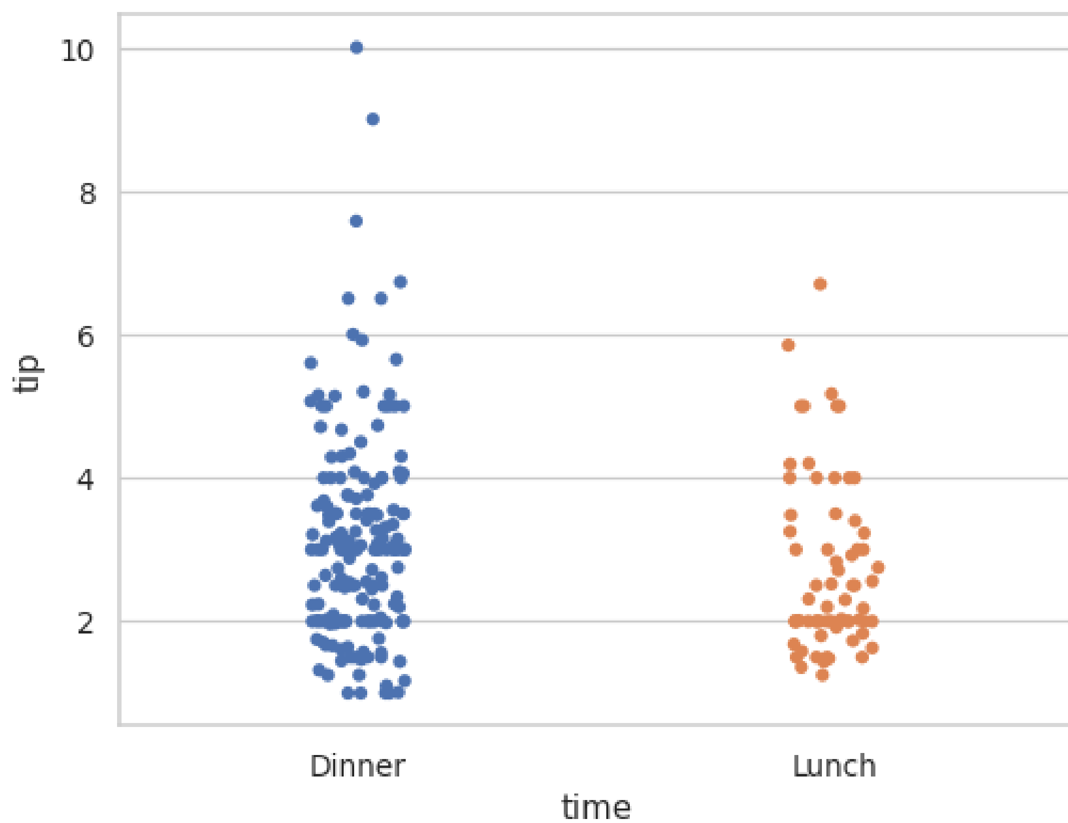
```
>>> ax =  sns.stripplot(x="day", y="total_bill", hue="smoker",
...                    data=tips, palette="Set2", size=20, marker="D",
...                    edgecolor="gray", alpha=.25)
```

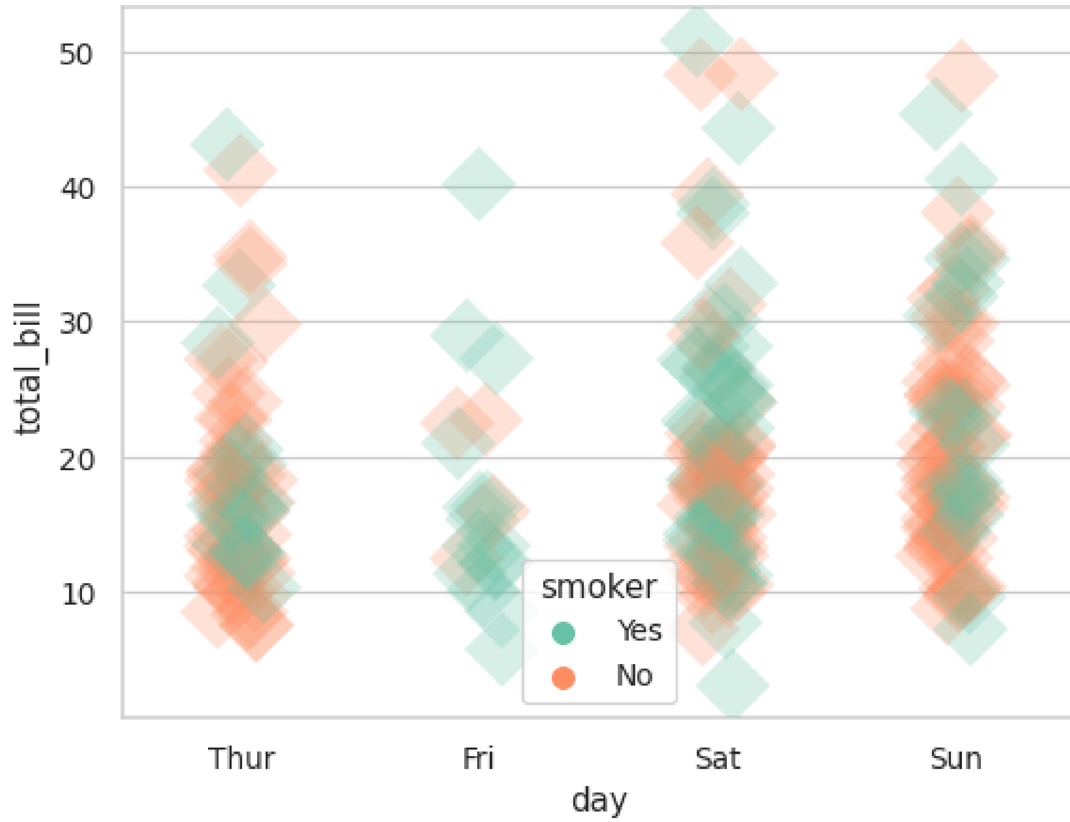
Draw strips of observations on top of a box plot:

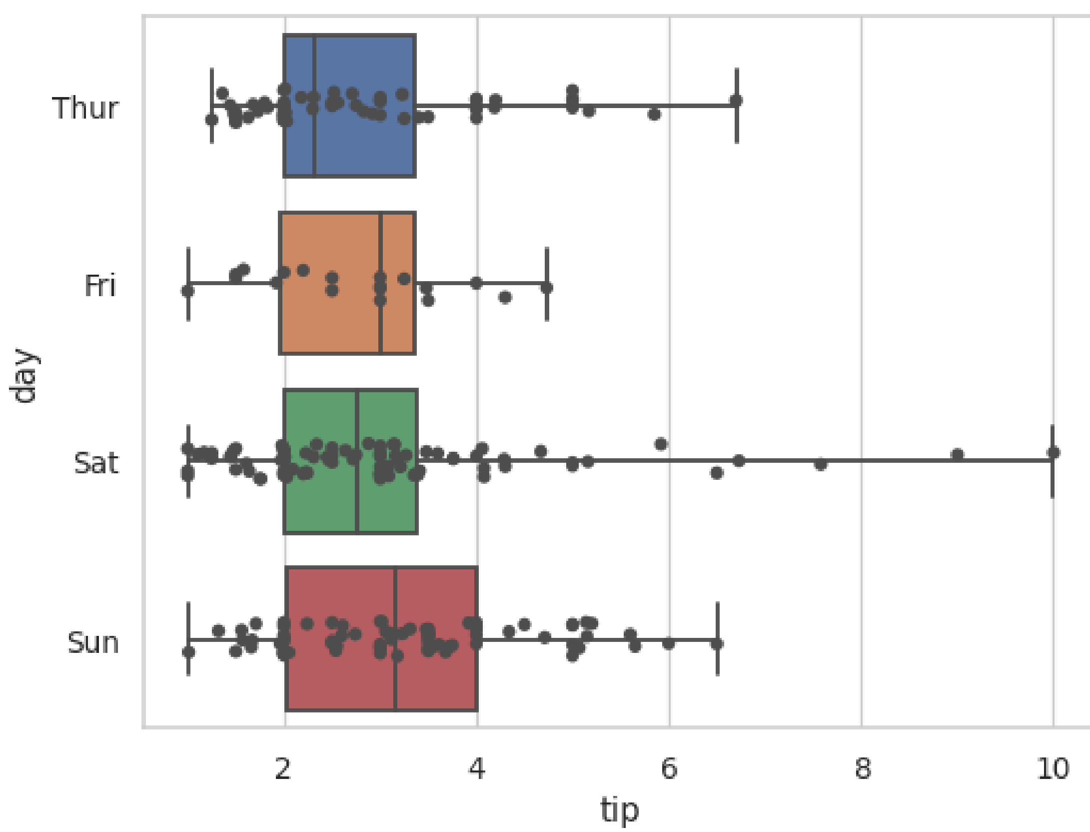
```
>>> import numpy as np
>>> ax = sns.boxplot(x="tip", y="day", data=tips, whis=np.inf)
>>> ax = sns.stripplot(x="tip", y="day", data=tips, color=".3")
```

Draw strips of observations on top of a violin plot:

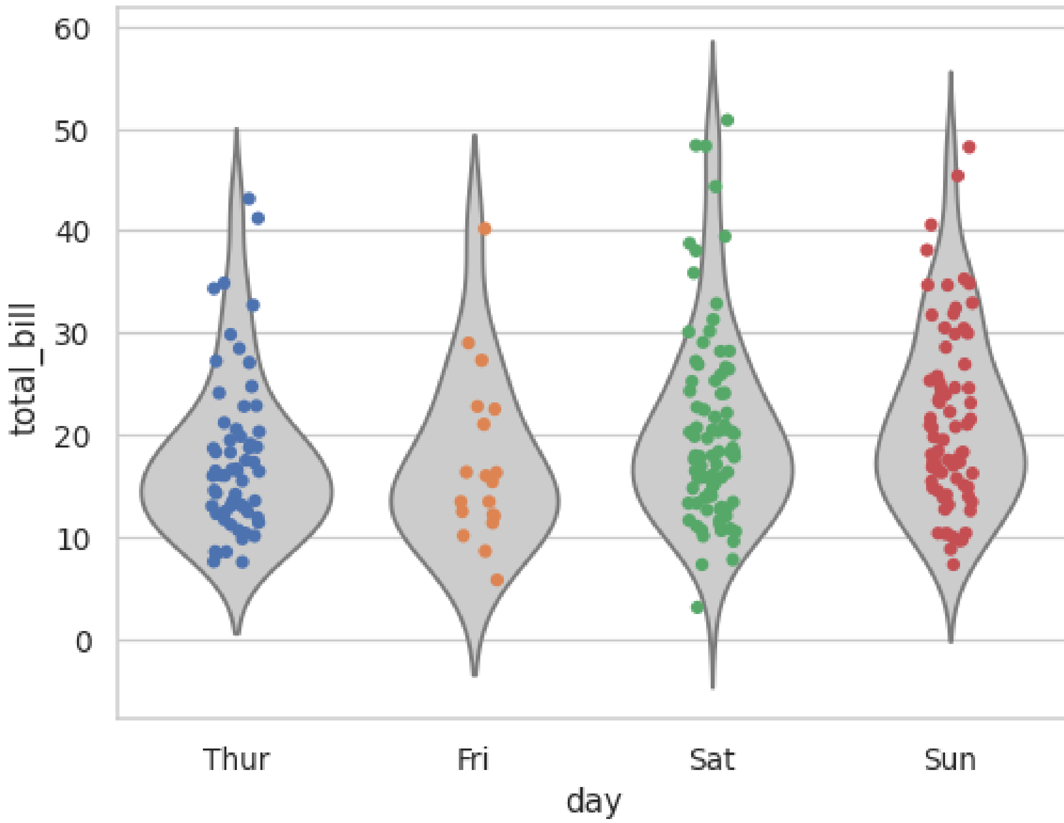

smoker
Yes
No
50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day


10
8
6
4
2
tip
Dinner
Lunch
time


50
40
30
20
10
total_bill
smoker
Yes
No
Thur
Fri
Sat
Sun
day


Thur
Fri
Sat
Sun
day
2
4
6
8
10
tip

```
>>> ax = sns.violinplot(x="day", y="total_bill", data=tips,
...                     inner=None, color=".8")
>>> ax = sns.stripplot(x="day", y="total_bill", data=tips)
```




Use `catplot()` to combine a `stripplot()` and a `FacetGrid`. This allows grouping within additional categorical variables. Using `catplot()` is safer than using `FacetGrid` directly, as it ensures synchronization of variable order across facets:

```
>>> g = sns.catplot(x="sex", y="total_bill",
...                 hue="smoker", col="time",
...                 data=tips, kind="strip",
...                 height=4, aspect=.7);
```

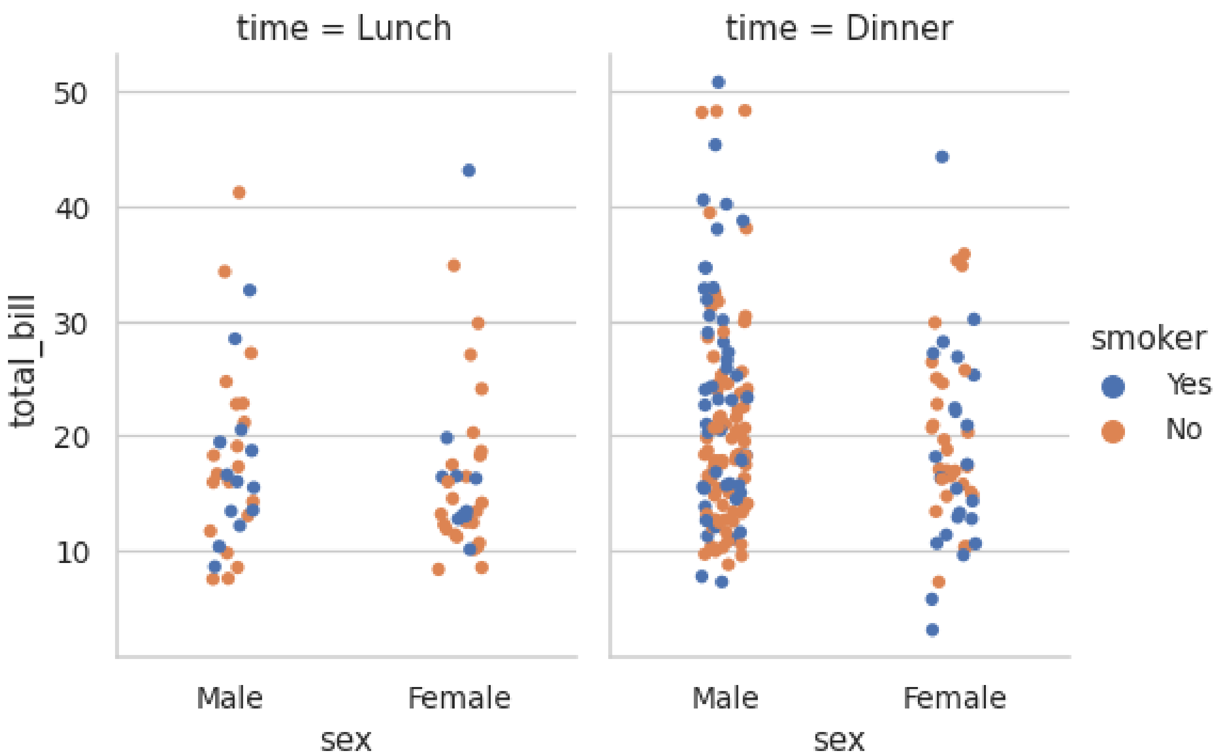
### 5.3.3 seaborn.swarmplot

seaborn.**swarmplot**(**, x=None, y=None, hue=None, data=None, order=None, hue_order=None, dodge=False, orient=None, color=None, palette=None, size=5, edgecolor='gray', linewidth=0, ax=None, **kwargs*)

Draw a categorical scatterplot with non-overlapping points.

This function is similar to `stripplot()`, but the points are adjusted (only along the categorical axis) so that they don't overlap. This gives a better representation of the distribution of values, but it does not scale well to large numbers of observations. This style of plot is sometimes called a "beeswarm".

A swarm plot can be drawn on its own, but it is also a good complement to a box or violin plot in cases where you want to show all observations along with some representation of the underlying distribution.




Arranging the points properly requires an accurate transformation between data and point coordinates. This means that non-default axis limits must be set *before* drawing the plot.

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`, and/or `hue` parameters.
- A "long-form" DataFrame, in which case the `x`, `y`, and `hue` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, . . . n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

**Parameters**

**x, y, hue** [names of variables in `data` or vector data, optional] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame, array, or list of arrays, optional] Dataset for plotting. If `x` and `y` are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**dodge** [bool, optional] When using `hue` nesting, setting this to `True` will separate the strips for different hue levels along the categorical axis. Otherwise, the points for each level will be plotted in one swarm.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both `x` and `y` are numeric or when plotting wide-form data.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by `color_palette()`, or a dictionary mapping hue levels to matplotlib colors.

**size** [float, optional] Radius of the markers, in points.

**edgecolor** [matplotlib color, "gray" is special-cased, optional] Color of the lines around each point. If you pass `"gray"`, the brightness is determined by the color palette used for the body of the points.

**linewidth** [float, optional] Width of the gray lines that frame the plot elements.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**kwargs** [key, value mappings] Other keyword arguments are passed through to `matplotlib.axes.Axes.scatter()`.

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot drawn onto it.

**See also:**

**`boxplot`** A traditional box-and-whisker plot with a similar API.

**`violinplot`** A combination of boxplot and kernel density estimation.

**`stripplot`** A scatterplot where one variable is categorical. Can be used in conjunction with other plots to show each observation.

**`catplot`** Combine a categorical plot with a `FacetGrid`.

### Examples

Draw a single horizontal swarm plot:

```
>>> import seaborn as sns
>>> sns.set_theme(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.swarmplot(x=tips["total_bill"])
```

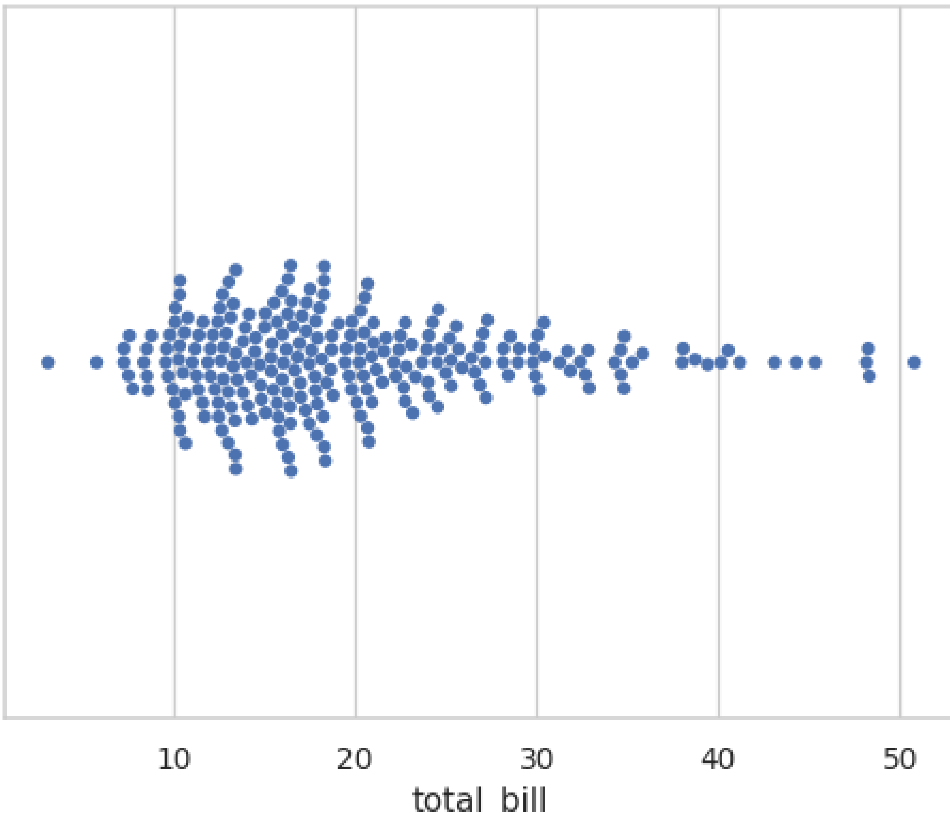
Group the swarms by a categorical variable:

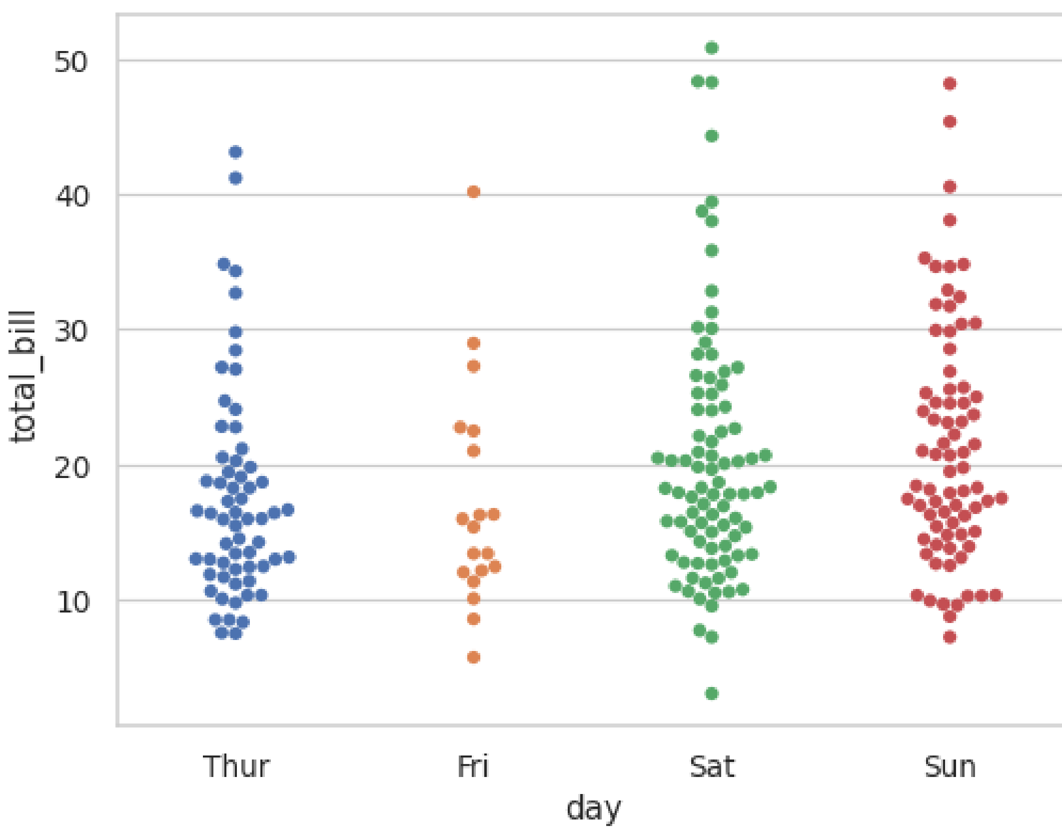
```
>>> ax = sns.swarmplot(x="day", y="total_bill", data=tips)
```

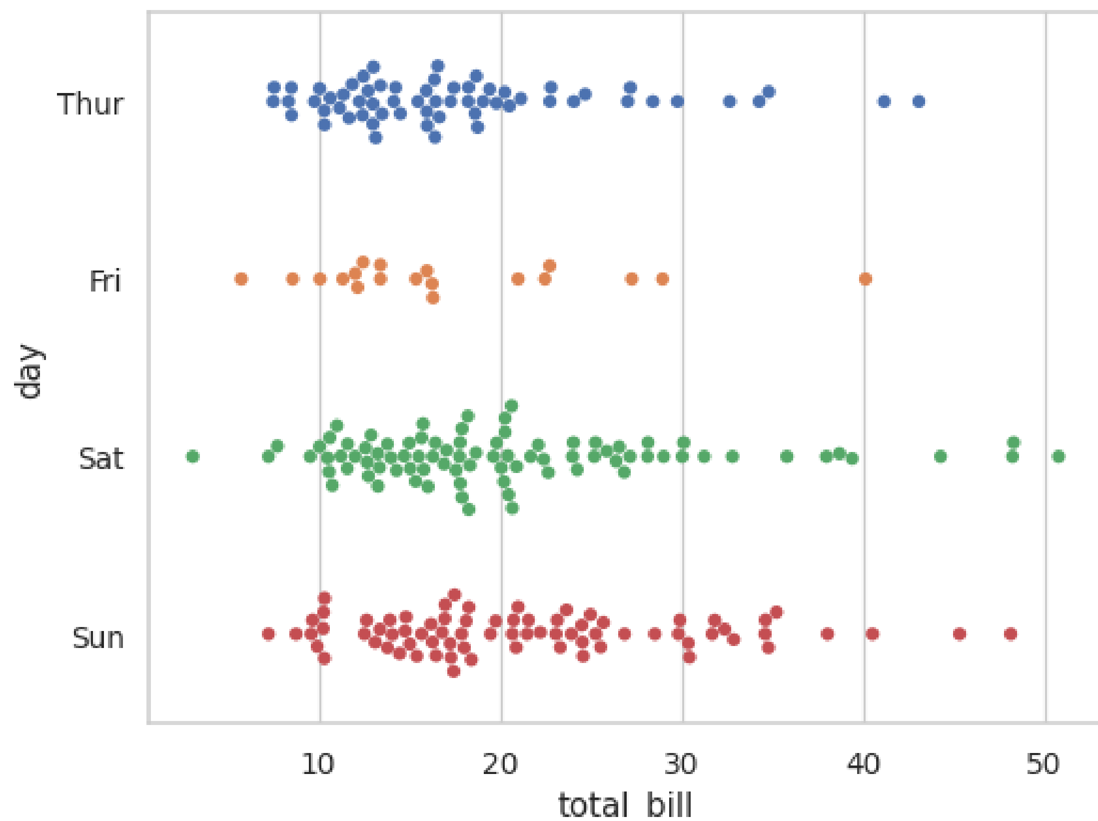
Draw horizontal swarms:

```
>>> ax = sns.swarmplot(x="total_bill", y="day", data=tips)
```

Color the points using a second categorical variable:


10
20
30
40
50
total_bill


50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day




```
>>> ax = sns.swarmplot(x="day", y="total_bill", hue="sex", data=tips)
```

Split each level of the hue variable along the categorical axis:

```
>>> ax = sns.swarmplot(x="day", y="total_bill", hue="smoker",
...                    data=tips, palette="Set2", dodge=True)
```

Control swarm order by passing an explicit order:

```
>>> ax = sns.swarmplot(x="time", y="total_bill", data=tips,
...                    order=["Dinner", "Lunch"])
```

Plot using larger points:

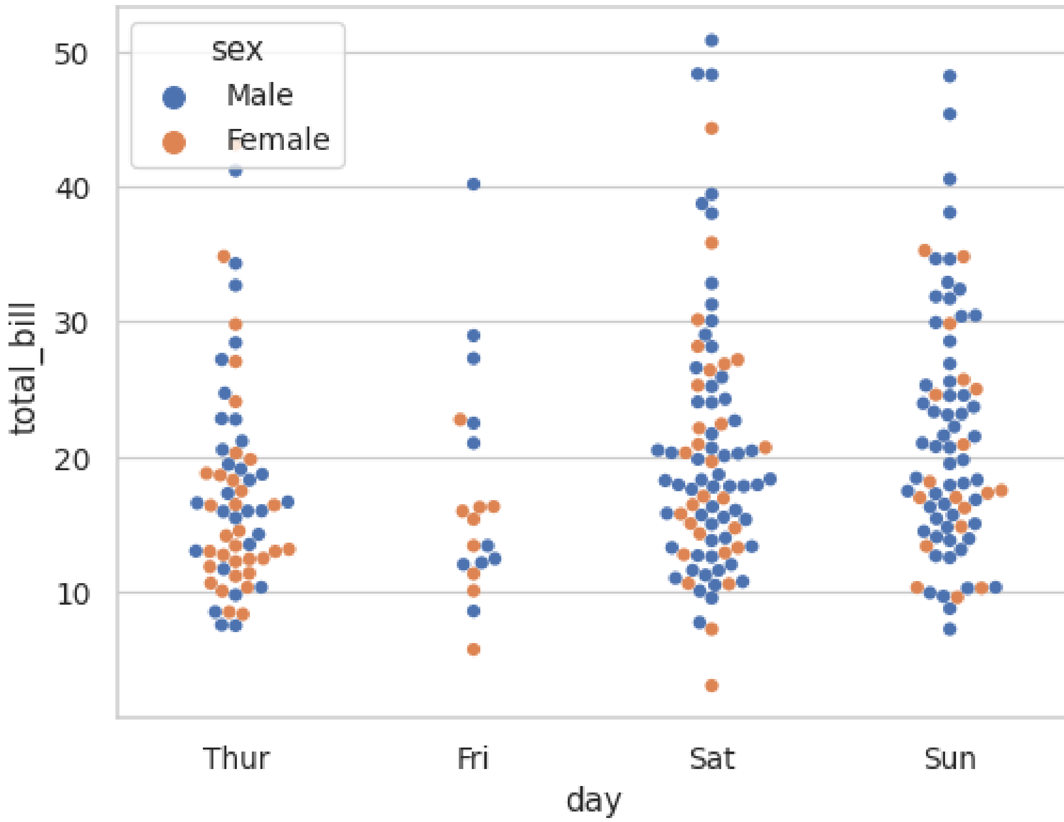
```
>>> ax = sns.swarmplot(x="time", y="total_bill", data=tips, size=6)
```

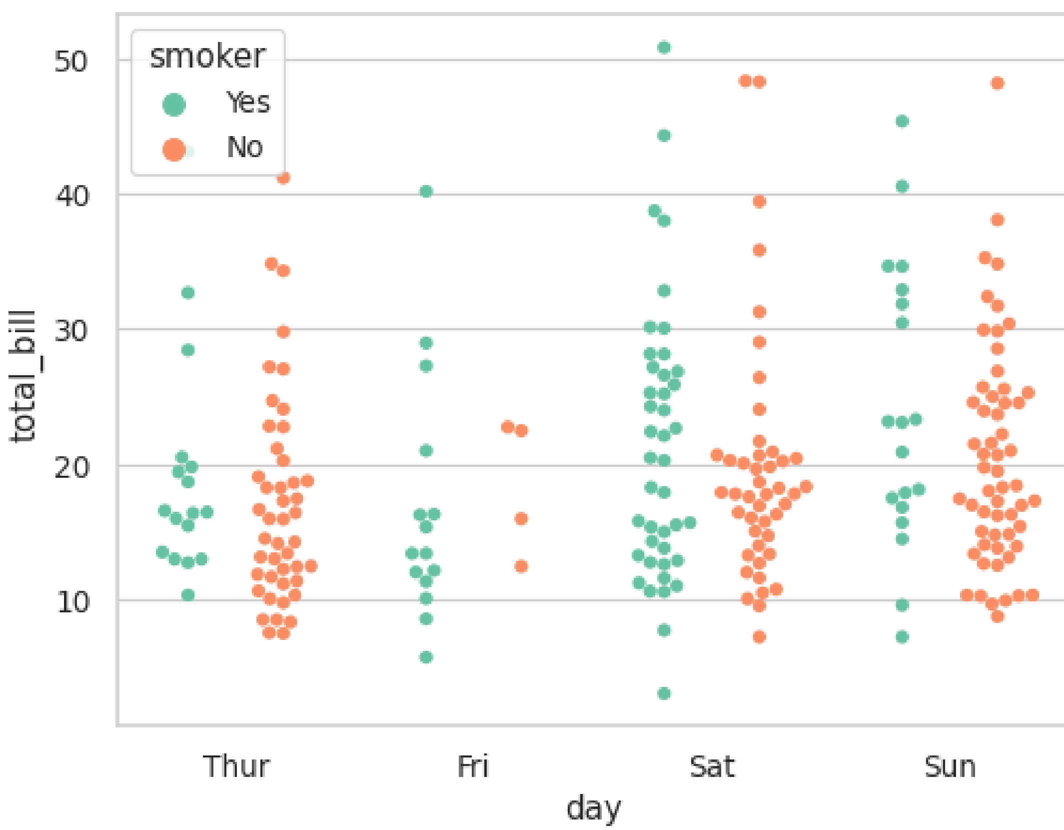
Draw swarms of observations on top of a box plot:

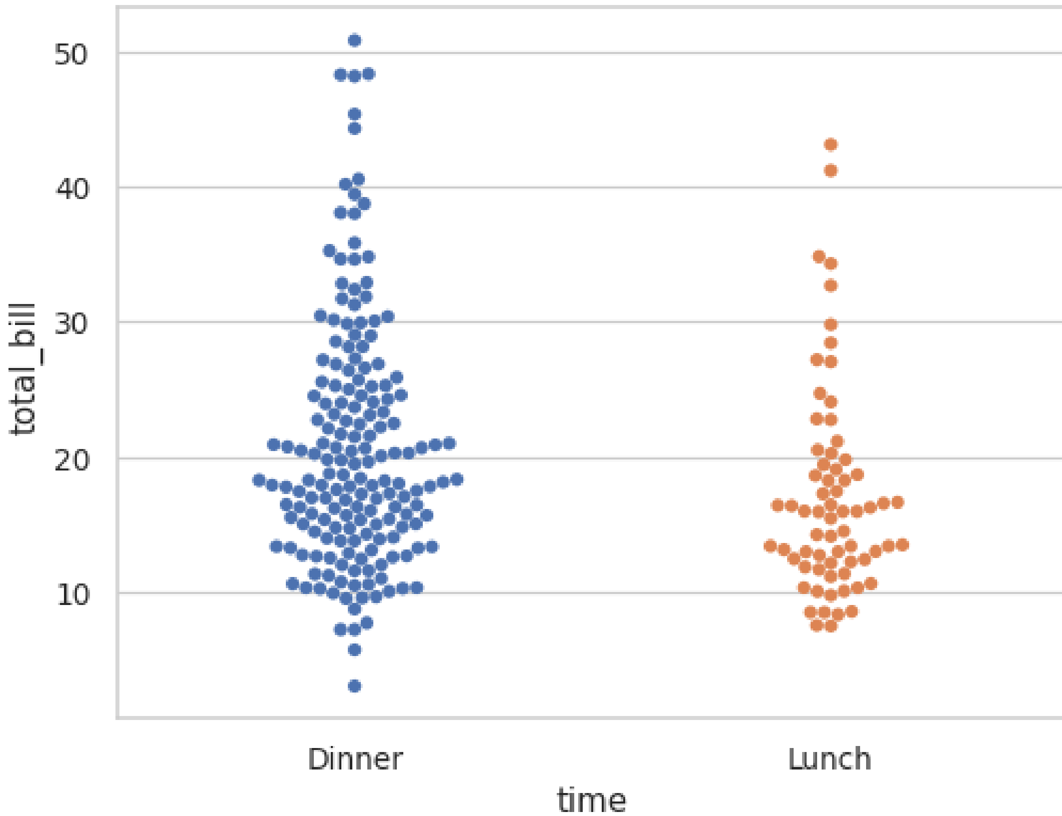
```
>>> ax = sns.boxplot(x="total_bill", y="day", data=tips, whis=np.inf)
>>> ax = sns.swarmplot(x="total_bill", y="day", data=tips, color=".2")
```

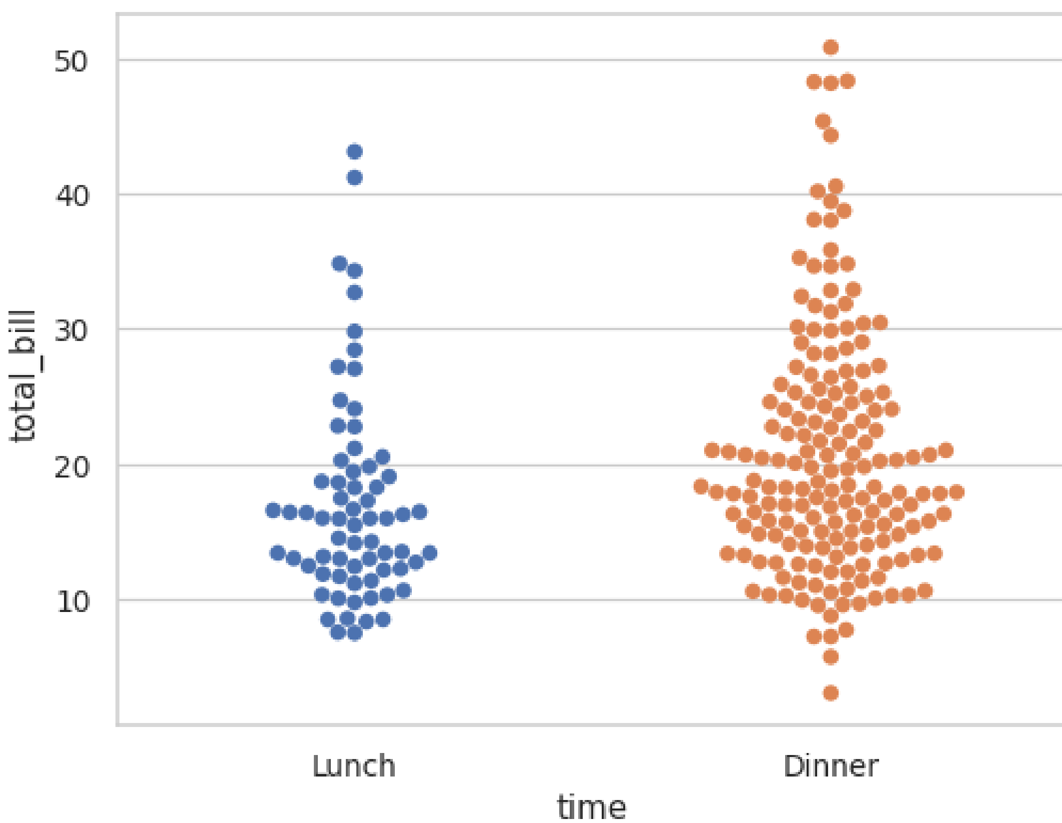
Draw swarms of observations on top of a violin plot:

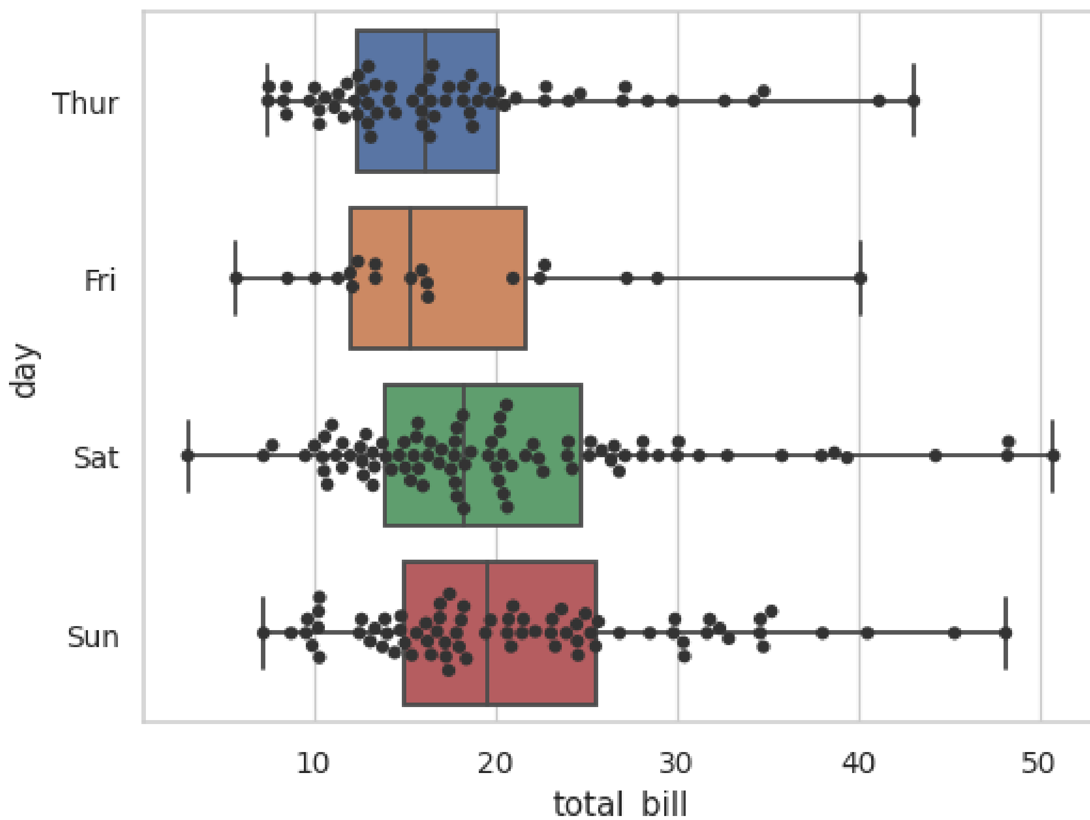
```
>>> ax = sns.violinplot(x="day", y="total_bill", data=tips, inner=None)
>>> ax = sns.swarmplot(x="day", y="total_bill", data=tips,
...                    color="white", edgecolor="gray")
```

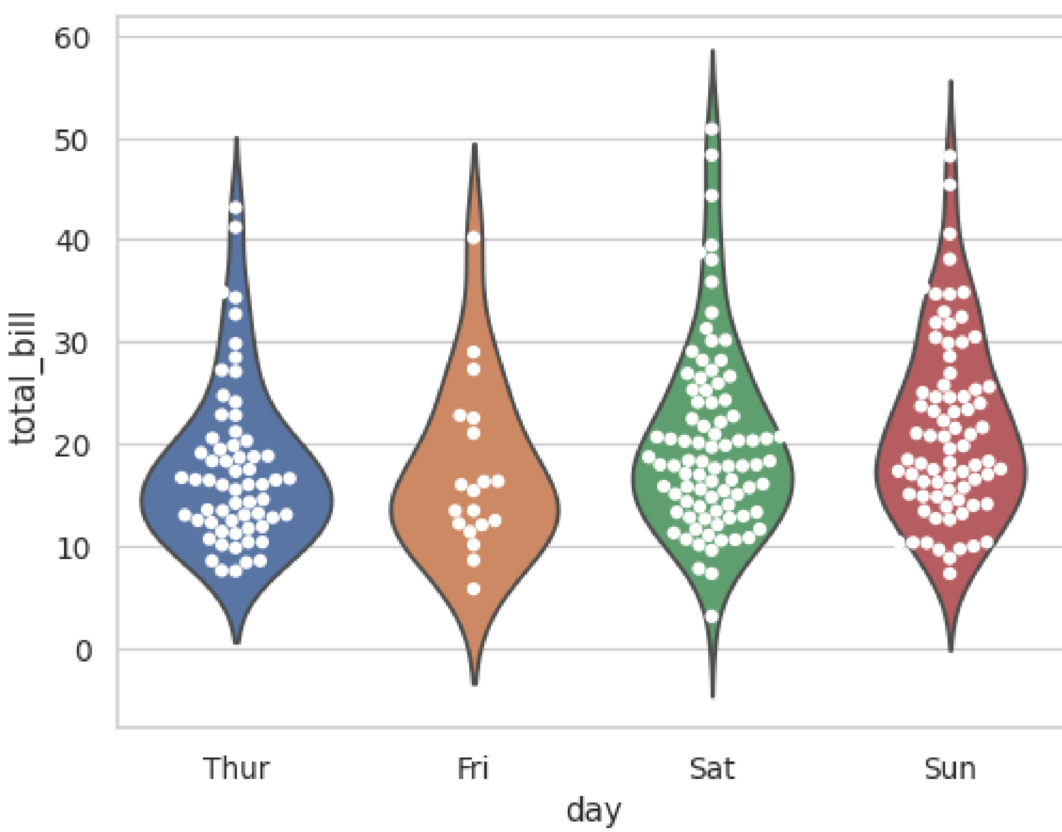

sex
Male
Female
total_bill
50
40
30
20
10
Thur
Fri
Sat
Sun
day


smoker
Yes
No
total_bill
50
40
30
20
10
Thur
Fri
Sat
Sun
day


50
40
30
20
10
total_bill
Dinner
Lunch
time

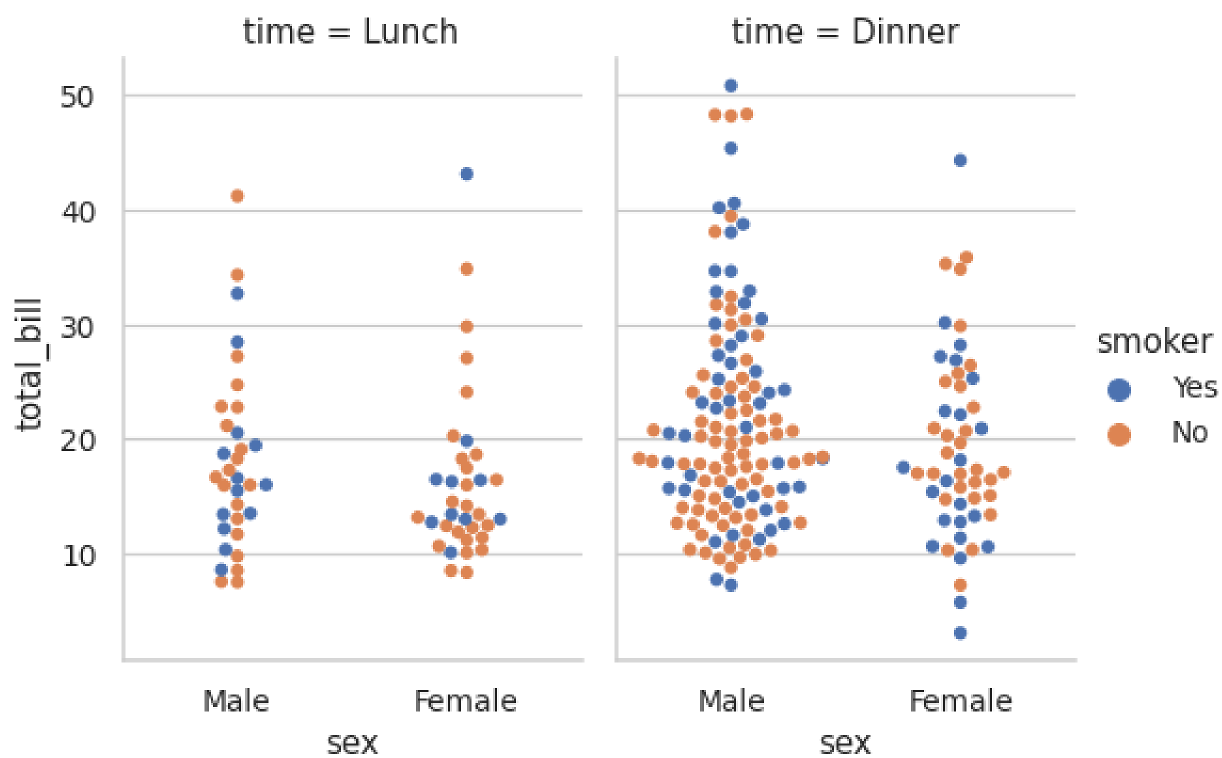

50
40
30
20
10
total_bill
Lunch
Dinner
time


Thur
Fri
Sat
Sun
day
10
20
30
40
50
total_bill


60
50
40
30
20
10
0
total_bill
Thur
Fri
Sat
Sun
day

Use *catplot()* to combine a *swarmplot()* and a *FacetGrid*. This allows grouping within additional categorical variables. Using *catplot()* is safer than using *FacetGrid* directly, as it ensures synchronization of variable order across facets:

```
>>> g = sns.catplot(x="sex", y="total_bill",
...                 hue="smoker", col="time",
...                 data=tips, kind="swarm",
...                 height=4, aspect=.7);
```




### 5.3.4 seaborn.boxplot

seaborn.**boxplot**(*, *x=None*, *y=None*, *hue=None*, *data=None*, *order=None*, *hue_order=None*, *orient=None*, *color=None*, *palette=None*, *saturation=0.75*, *width=0.8*, *dodge=True*, *fliersize=5*, *linewidth=None*, *whis=1.5*, *ax=None*, ***kwargs*)

Draw a box plot to show distributions with respect to categories.

A box plot (or box-and-whisker plot) shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution, except for points that are determined to be "outliers" using a method that is a function of the inter-quartile range.

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`, and/or `hue` parameters.
- A "long-form" DataFrame, in which case the `x`, `y`, and `hue` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.

- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, … n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

**Parameters**

**x, y, hue** [names of variables in `data` or vector data, optional] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame, array, or list of arrays, optional] Dataset for plotting. If `x` and `y` are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both `x` and `y` are numeric or when plotting wide-form data.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by `color_palette()`, or a dictionary mapping hue levels to matplotlib colors.

**saturation** [float, optional] Proportion of the original saturation to draw colors at. Large patches often look better with slightly desaturated colors, but set this to `1` if you want the plot colors to perfectly match the input color spec.

**width** [float, optional] Width of a full element when not using hue nesting, or width of all the elements for one level of the major grouping variable.

**dodge** [bool, optional] When hue nesting is used, whether elements should be shifted along the categorical axis.

**fliersize** [float, optional] Size of the markers used to indicate outlier observations.

**linewidth** [float, optional] Width of the gray lines that frame the plot elements.

**whis** [float, optional] Proportion of the IQR past the low and high quartiles to extend the plot whiskers. Points outside this range will be identified as outliers.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**kwargs** [key, value mappings] Other keyword arguments are passed through to `matplotlib.axes.Axes.boxplot()`.

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot drawn onto it.

**See also:**

**`violinplot`** A combination of boxplot and kernel density estimation.

**`stripplot`** A scatterplot where one variable is categorical. Can be used in conjunction with other plots to show each observation.

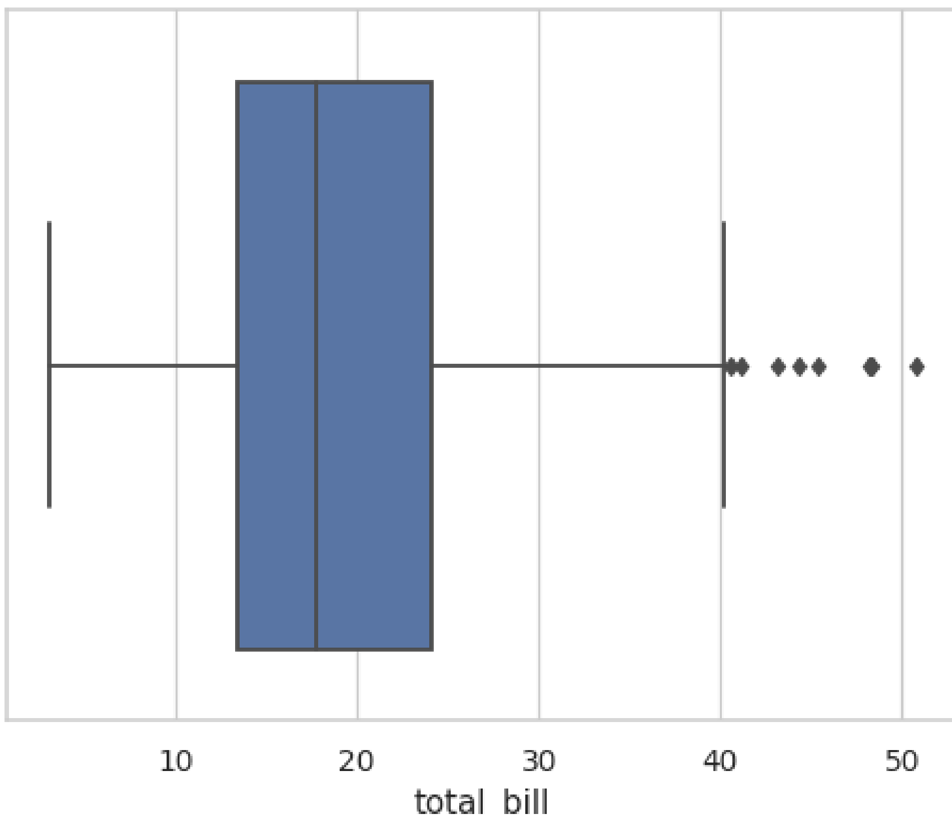
***swarmplot*** A categorical scatterplot where the points do not overlap. Can be used with other plots to show each observation.

***catplot*** Combine a categorical plot with a *FacetGrid*.

### Examples

Draw a single horizontal boxplot:

```
>>> import seaborn as sns
>>> sns.set_theme(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.boxplot(x=tips["total_bill"])
```




Draw a vertical boxplot grouped by a categorical variable:

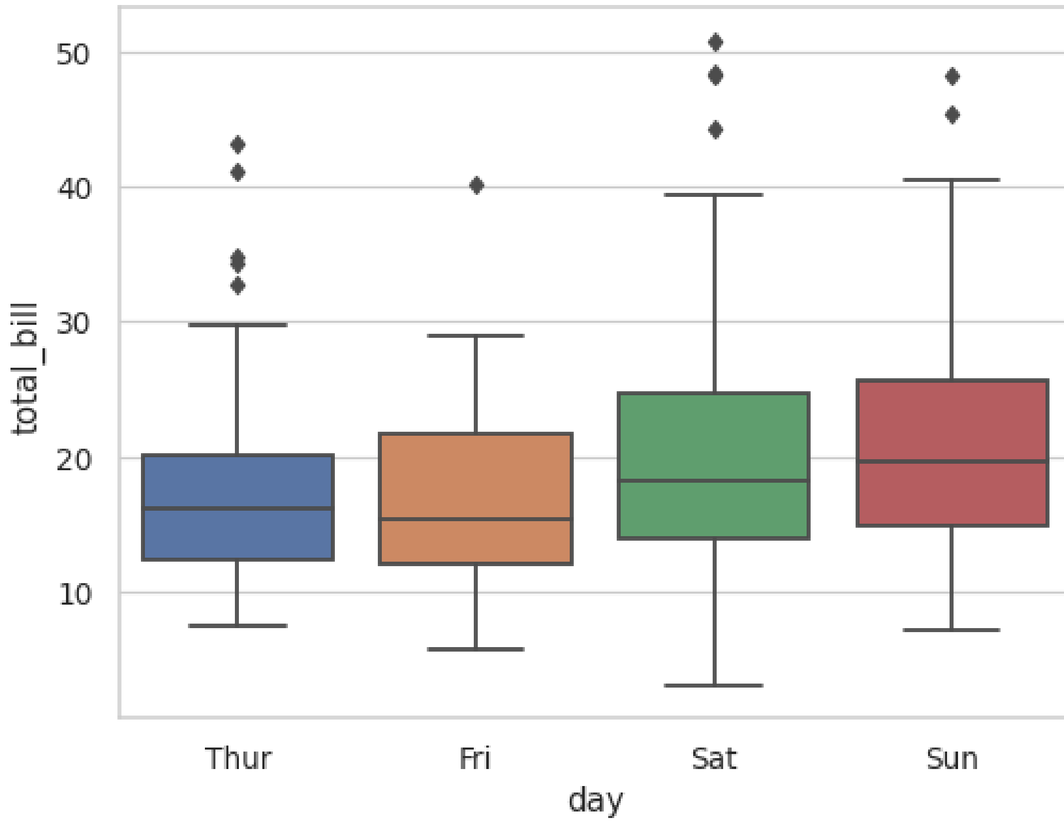
```
>>> ax = sns.boxplot(x="day", y="total_bill", data=tips)
```

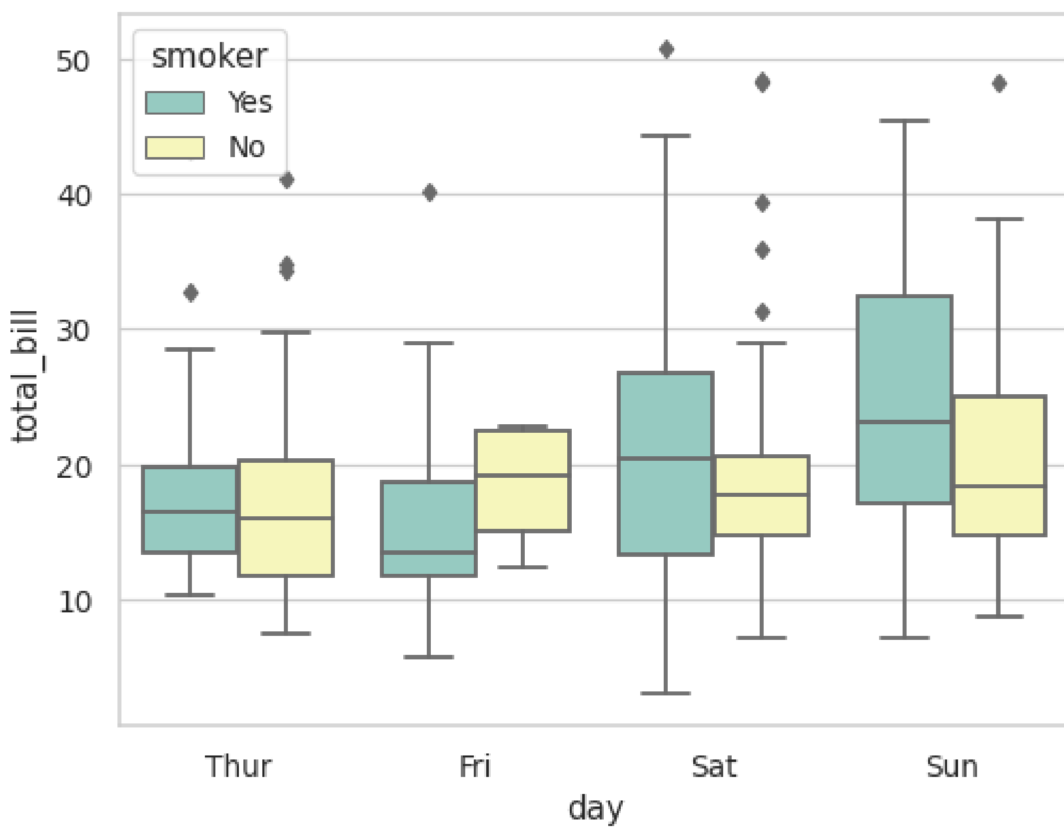
Draw a boxplot with nested grouping by two categorical variables:

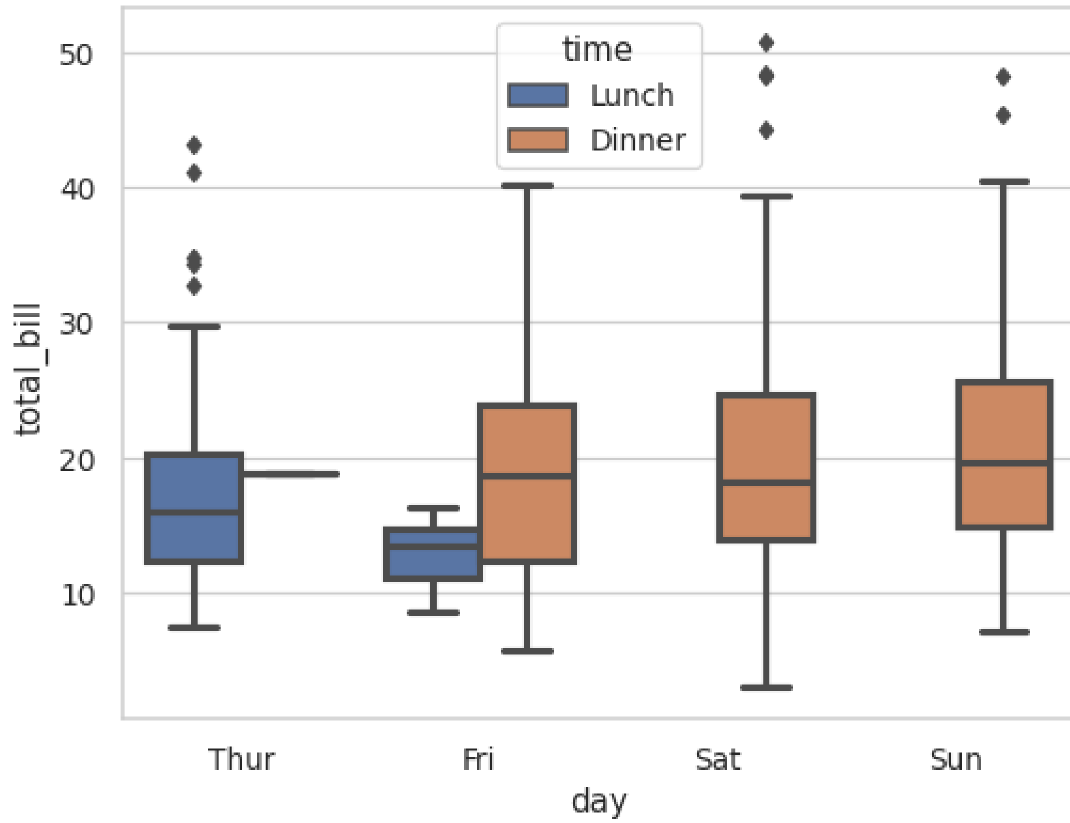
```
>>> ax = sns.boxplot(x="day", y="total_bill", hue="smoker",
...                  data=tips, palette="Set3")
```

Draw a boxplot with nested grouping when some bins are empty:

```
>>> ax = sns.boxplot(x="day", y="total_bill", hue="time",
...                  data=tips, linewidth=2.5)
```


50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day


smoker
Yes
No
50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day




Control box order by passing an explicit order:

```
>>> ax = sns.boxplot(x="time", y="tip", data=tips,
...                  order=["Dinner", "Lunch"])
```

Draw a boxplot for each numeric variable in a DataFrame:

```
>>> iris = sns.load_dataset("iris")
>>> ax = sns.boxplot(data=iris, orient="h", palette="Set2")
```

Use `hue` without changing box position or width:

```
>>> tips["weekend"] = tips["day"].isin(["Sat", "Sun"])
>>> ax = sns.boxplot(x="day", y="total_bill", hue="weekend",
...                  data=tips, dodge=False)
```

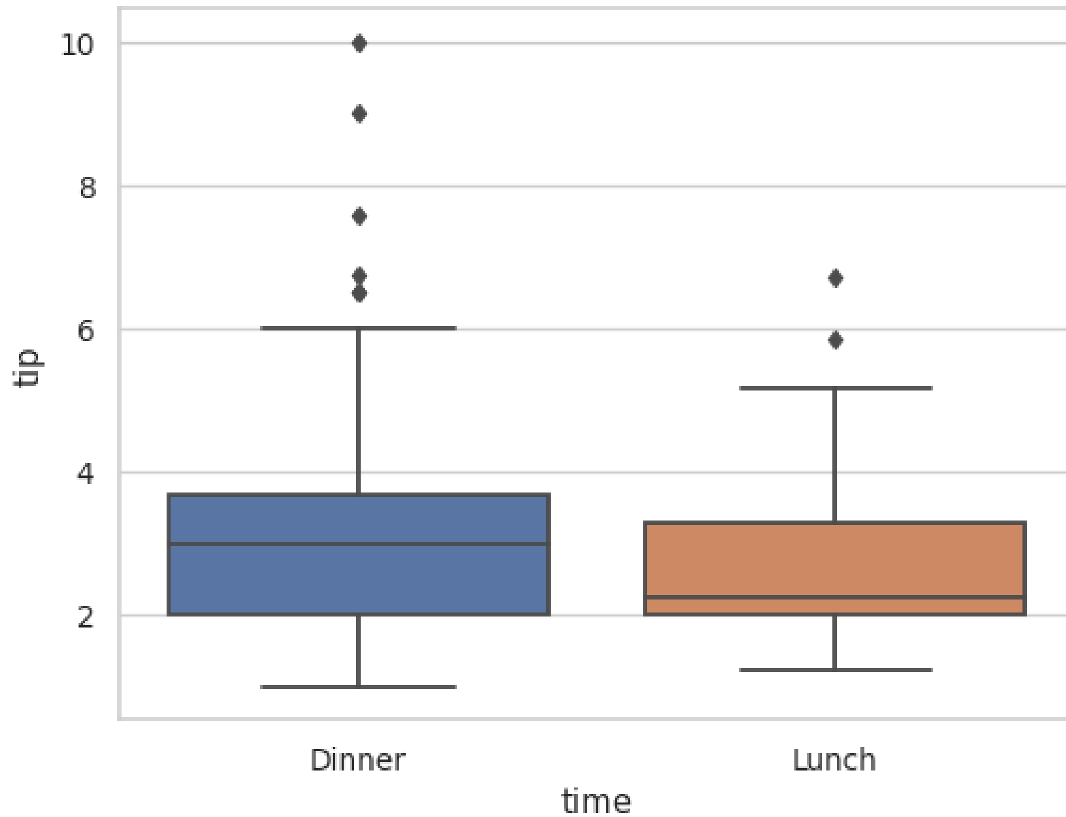
Use `swarmplot()` to show the datapoints on top of the boxes:

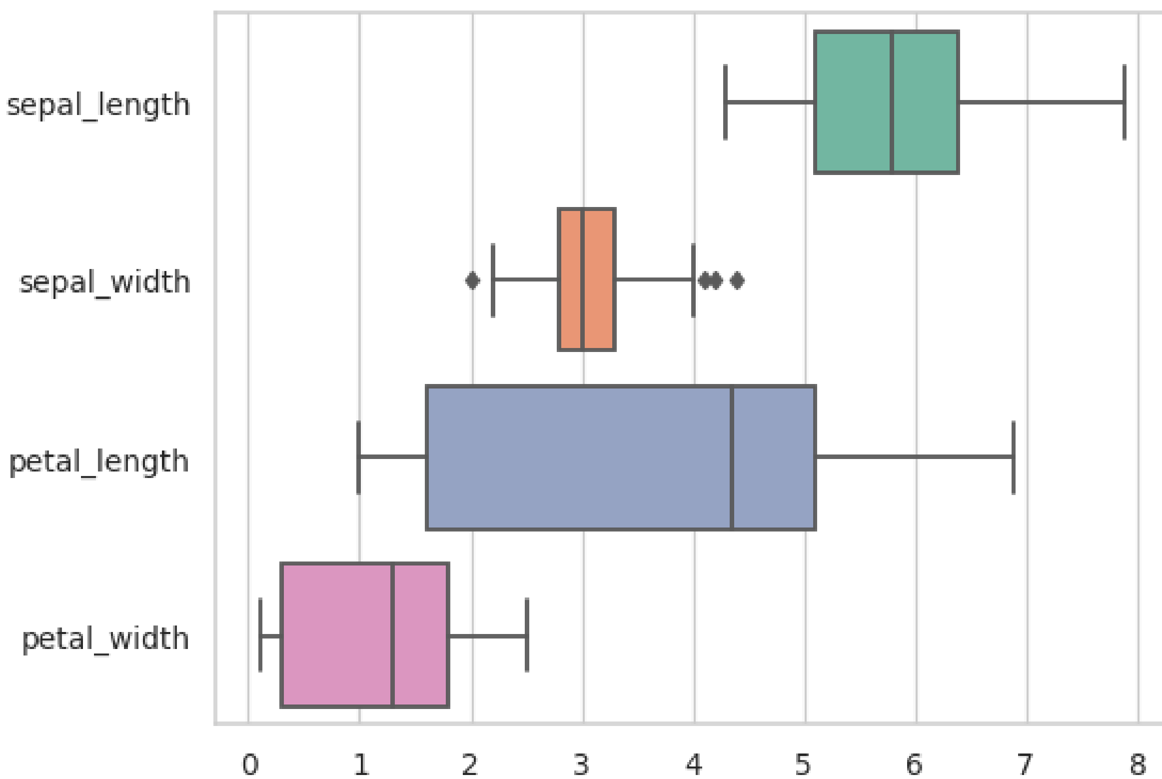
```
>>> ax = sns.boxplot(x="day", y="total_bill", data=tips)
>>> ax = sns.swarmplot(x="day", y="total_bill", data=tips, color=".25")
```

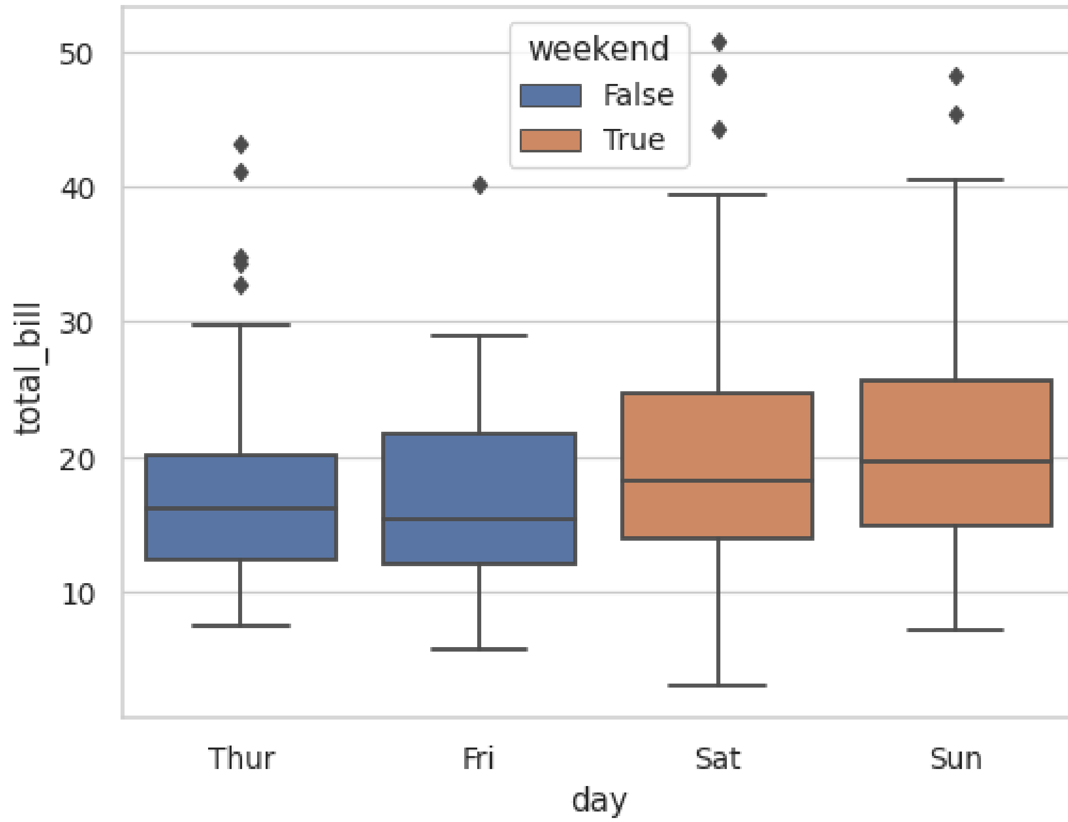
Use `catplot()` to combine a `boxplot()` and a `FacetGrid`. This allows grouping within additional categorical variables. Using `catplot()` is safer than using `FacetGrid` directly, as it ensures synchronization of variable order across facets:

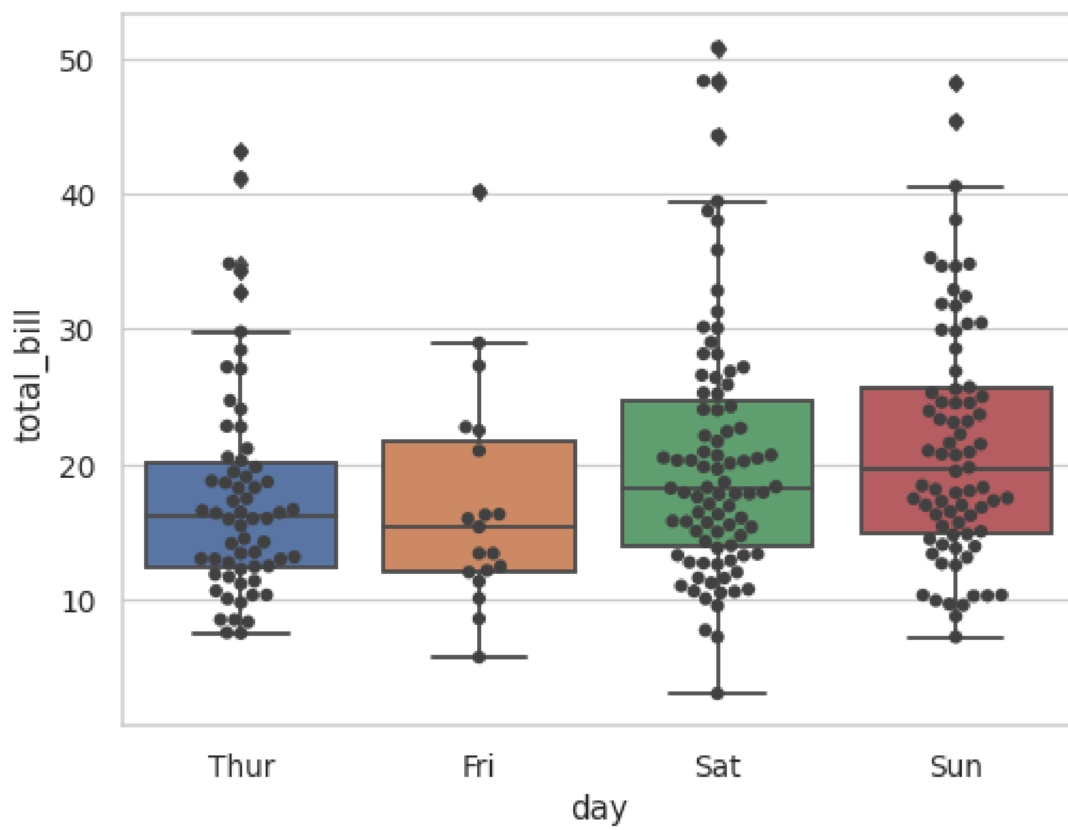
```
>>> g = sns.catplot(x="sex", y="total_bill",
...                 hue="smoker", col="time",
```

(continues on next page)


10
8
6
4
2
tip
Dinner
Lunch
time

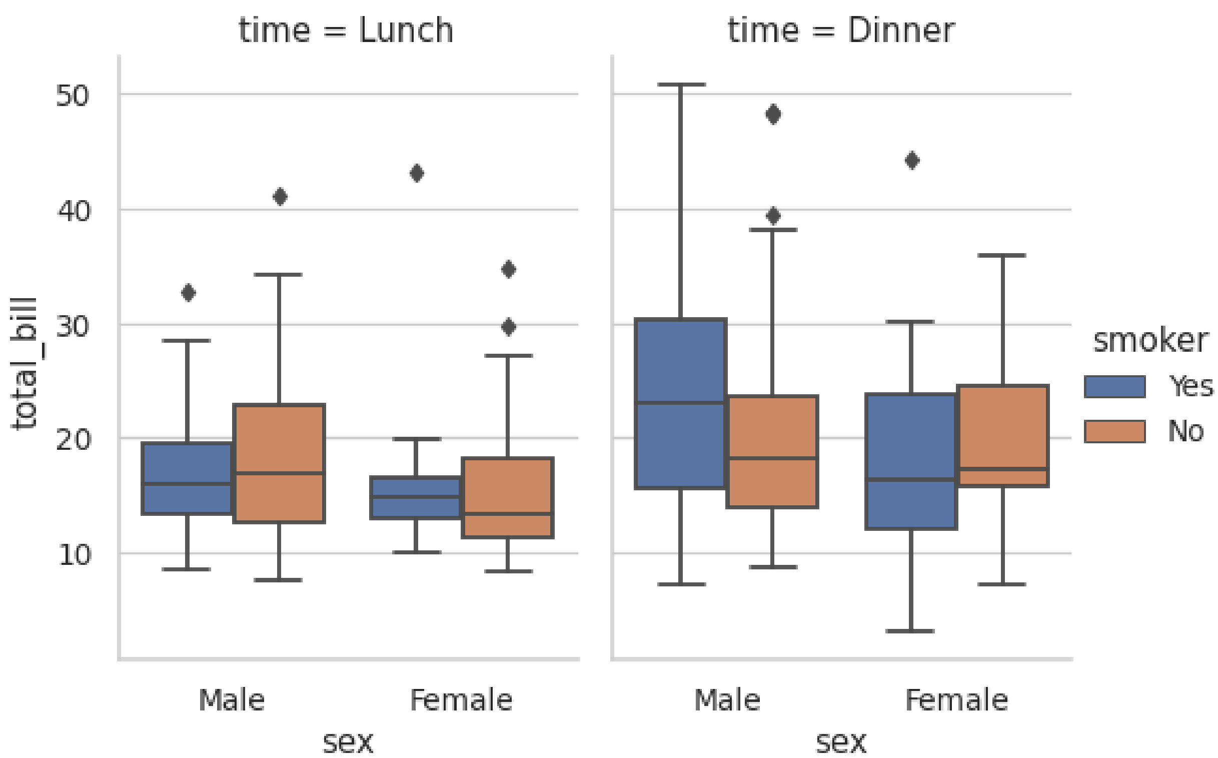

sepal_length
sepal_width
petal_length
petal_width
0
1
2
3
4
5
6
7
8


weekend
False
True
50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day


50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day

(continued from previous page)

```
...                    data=tips, kind="box",
...                    height=4, aspect=.7);
```




### 5.3.5 seaborn.violinplot

seaborn.**violinplot**(*, *x=None*, *y=None*, *hue=None*, *data=None*, *order=None*, *hue_order=None*, *bw='scott'*, *cut=2*, *scale='area'*, *scale_hue=True*, *gridsize=100*, *width=0.8*, *inner='box'*, *split=False*, *dodge=True*, *orient=None*, *linewidth=None*, *color=None*, *palette=None*, *saturation=0.75*, *ax=None*, ***kwargs*)

Draw a combination of boxplot and kernel density estimate.

A violin plot plays a similar role as a box and whisker plot. It shows the distribution of quantitative data across several levels of one (or more) categorical variables such that those distributions can be compared. Unlike a box plot, in which all of the plot components correspond to actual datapoints, the violin plot features a kernel density estimation of the underlying distribution.

This can be an effective and attractive way to show multiple distributions of data at once, but keep in mind that the estimation procedure is influenced by the sample size, and violins for relatively small samples might look misleadingly smooth.

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`, and/or `hue` parameters.
- A "long-form" DataFrame, in which case the `x`, `y`, and `hue` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.

- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, … n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

**Parameters**

**x, y, hue** [names of variables in `data` or vector data, optional] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame, array, or list of arrays, optional] Dataset for plotting. If `x` and `y` are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**bw** [{'scott', 'silverman', float}, optional] Either the name of a reference rule or the scale factor to use when computing the kernel bandwidth. The actual kernel size will be determined by multiplying the scale factor by the standard deviation of the data within each bin.

**cut** [float, optional] Distance, in units of bandwidth size, to extend the density past the extreme datapoints. Set to 0 to limit the violin range within the range of the observed data (i.e., to have the same effect as `trim=True` in `ggplot`.

**scale** [{"area", "count", "width"}, optional] The method used to scale the width of each violin. If `area`, each violin will have the same area. If `count`, the width of the violins will be scaled by the number of observations in that bin. If `width`, each violin will have the same width.

**scale_hue** [bool, optional] When nesting violins using a `hue` variable, this parameter determines whether the scaling is computed within each level of the major grouping variable (`scale_hue=True`) or across all the violins on the plot (`scale_hue=False`).

**gridsize** [int, optional] Number of points in the discrete grid used to compute the kernel density estimate.

**width** [float, optional] Width of a full element when not using hue nesting, or width of all the elements for one level of the major grouping variable.

**inner** [{"box", "quartile", "point", "stick", None}, optional] Representation of the datapoints in the violin interior. If `box`, draw a miniature boxplot. If `quartiles`, draw the quartiles of the distribution. If `point` or `stick`, show each underlying datapoint. Using `None` will draw unadorned violins.

**split** [bool, optional] When using hue nesting with a variable that takes two levels, setting `split` to True will draw half of a violin for each level. This can make it easier to directly compare the distributions.

**dodge** [bool, optional] When hue nesting is used, whether elements should be shifted along the categorical axis.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both `x` and `y` are numeric or when plotting wide-form data.

**linewidth** [float, optional] Width of the gray lines that frame the plot elements.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by *color_palette()*, or a dictionary mapping hue levels to matplotlib colors.

**saturation** [float, optional] Proportion of the original saturation to draw colors at. Large patches often look better with slightly desaturated colors, but set this to `1` if you want the plot colors to perfectly match the input color spec.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot drawn onto it.

**See also:**

***boxplot*** A traditional box-and-whisker plot with a similar API.

***stripplot*** A scatterplot where one variable is categorical. Can be used in conjunction with other plots to show each observation.

***swarmplot*** A categorical scatterplot where the points do not overlap. Can be used with other plots to show each observation.

***catplot*** Combine a categorical plot with a *FacetGrid*.

### Examples

Draw a single horizontal violinplot:

```
>>> import seaborn as sns
>>> sns.set_theme(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.violinplot(x=tips["total_bill"])
```

Draw a vertical violinplot grouped by a categorical variable:

```
>>> ax = sns.violinplot(x="day", y="total_bill", data=tips)
```

Draw a violinplot with nested grouping by two categorical variables:

```
>>> ax = sns.violinplot(x="day", y="total_bill", hue="smoker",
...                     data=tips, palette="muted")
```

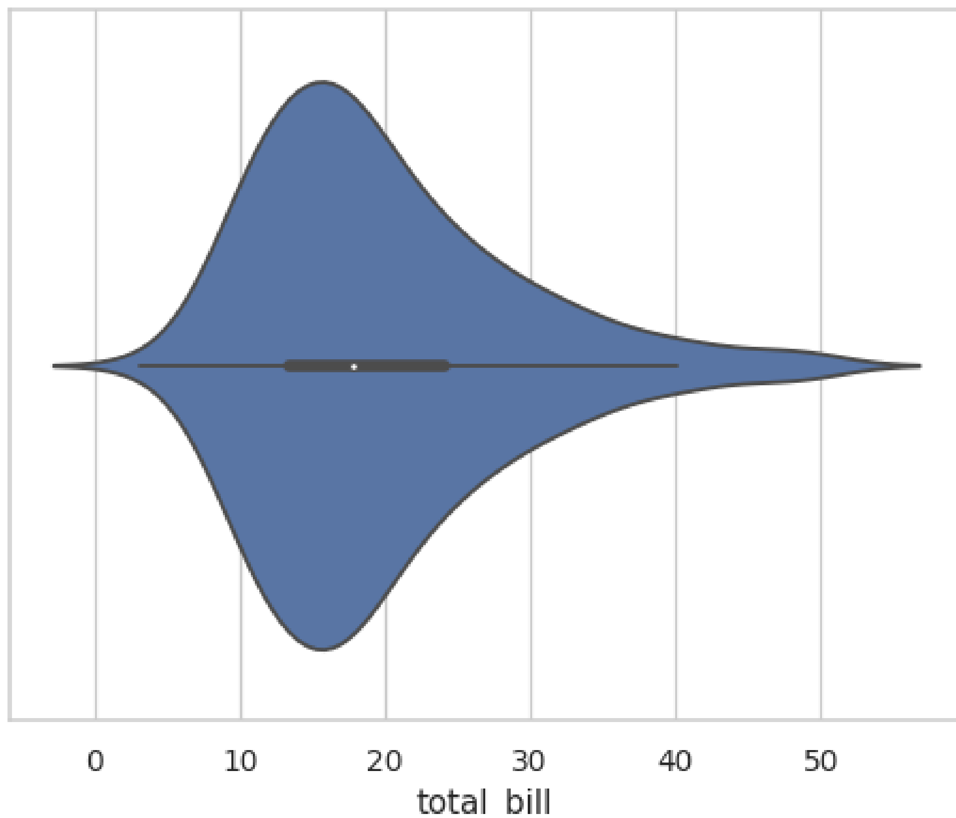
Draw split violins to compare the across the hue variable:

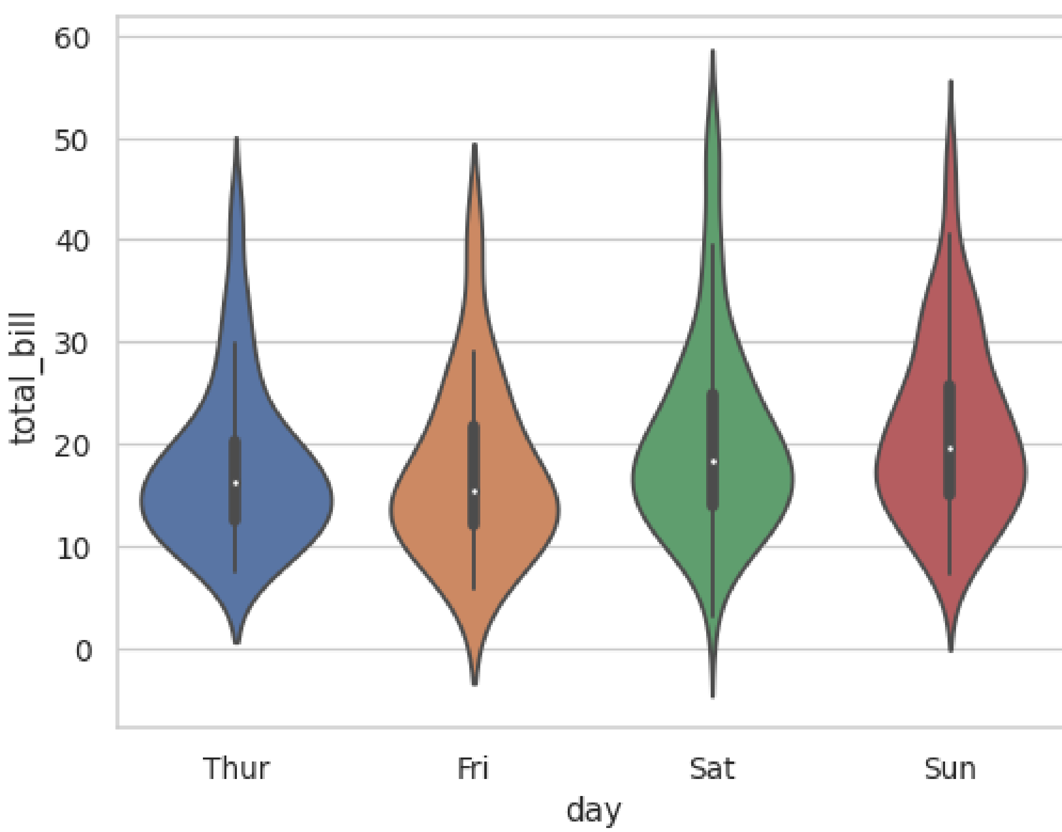
```
>>> ax = sns.violinplot(x="day", y="total_bill", hue="smoker",
...                     data=tips, palette="muted", split=True)
```

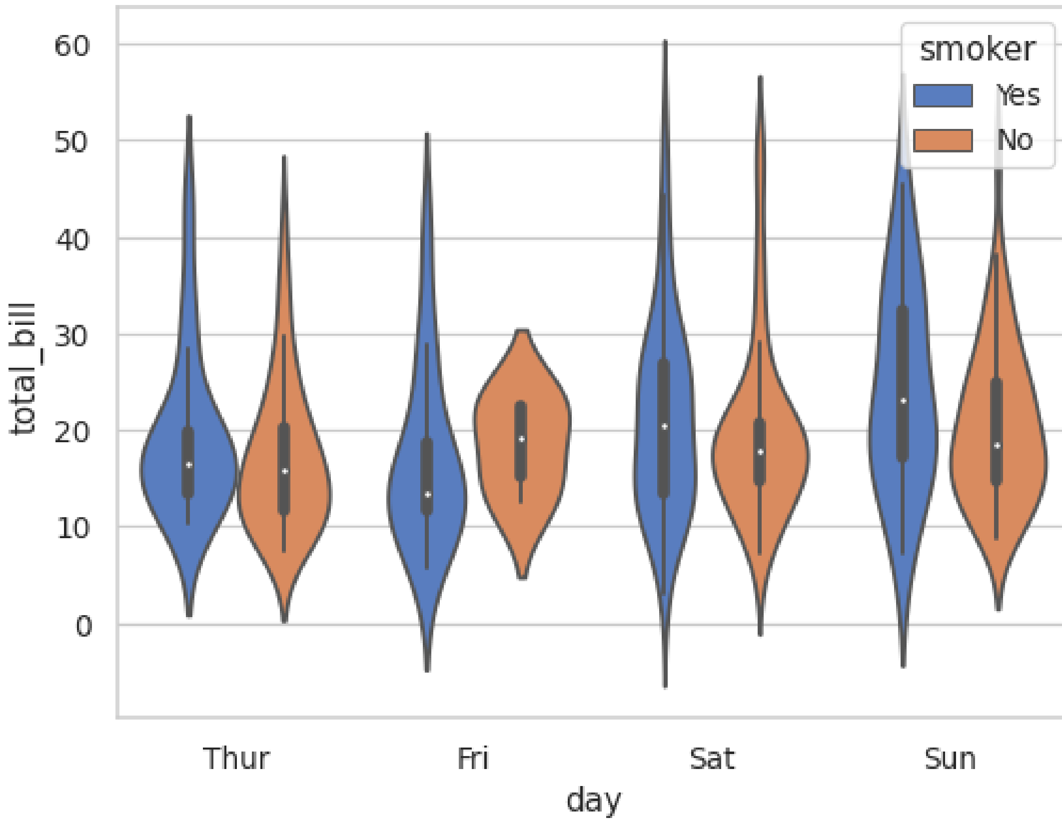
Control violin order by passing an explicit order:

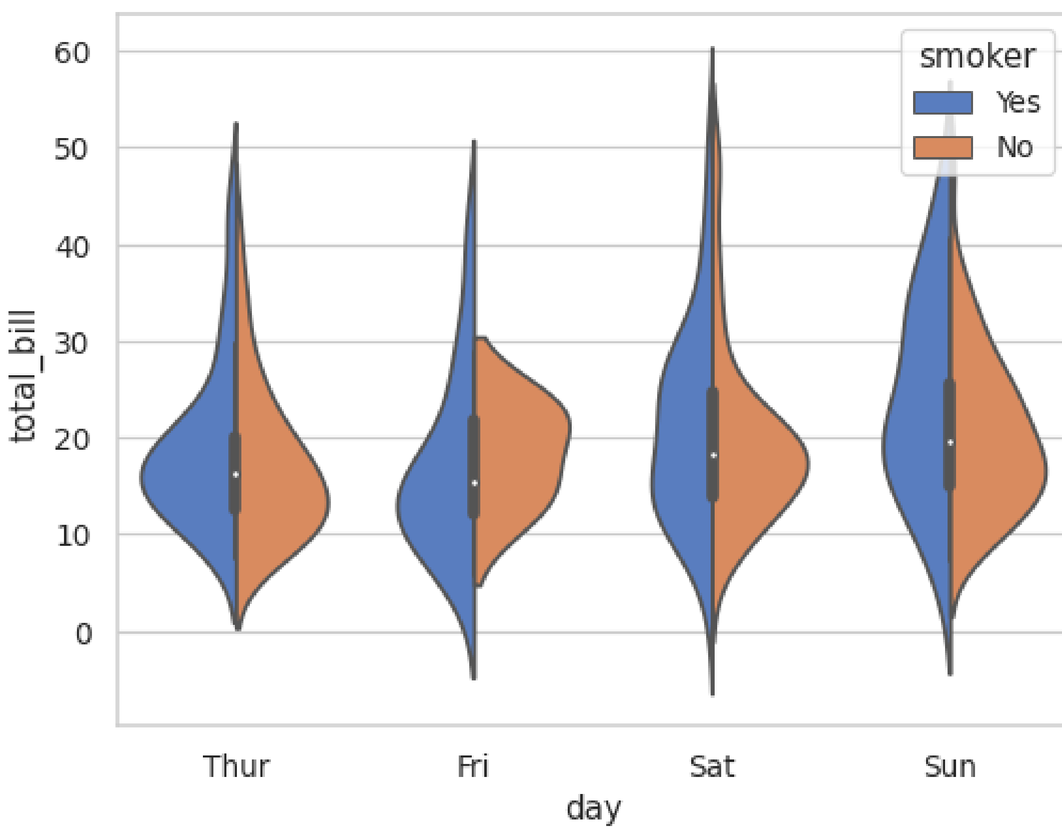
```
>>> ax = sns.violinplot(x="time", y="tip", data=tips,
...                     order=["Dinner", "Lunch"])
```

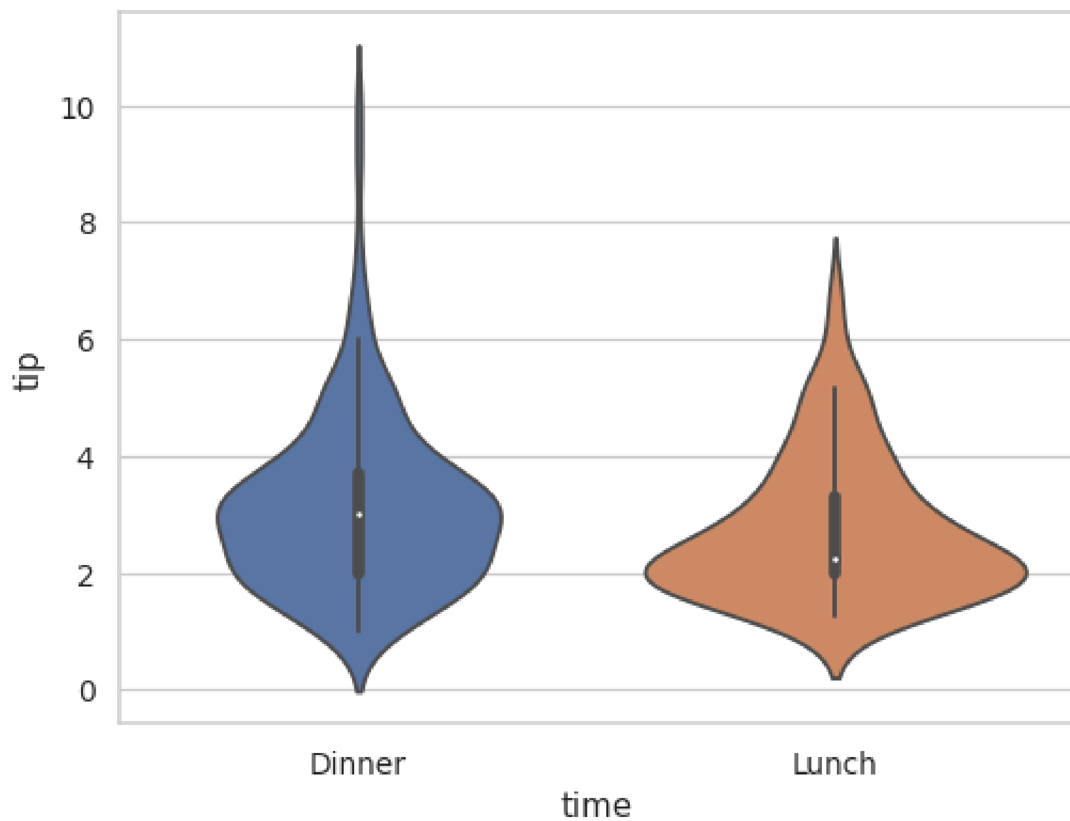
Scale the violin width by the number of observations in each bin:


0
10
20
30
40
50
total_bill


60
50
40
30
20
10
0
total_bill
Thur
Fri
Sat
Sun
day


smoker
Yes
No
60
50
40
30
20
10
0
total_bill
Thur
Fri
Sat
Sun
day


smoker
Yes
No
60
50
40
30
20
10
0
total_bill
Thur
Fri
Sat
Sun
day




```
>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                     data=tips, palette="Set2", split=True,
...                     scale="count")
```

Draw the quartiles as horizontal lines instead of a mini-box:

```
>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                     data=tips, palette="Set2", split=True,
...                     scale="count", inner="quartile")
```

Show each observation with a stick inside the violin:

```
>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                     data=tips, palette="Set2", split=True,
...                     scale="count", inner="stick")
```

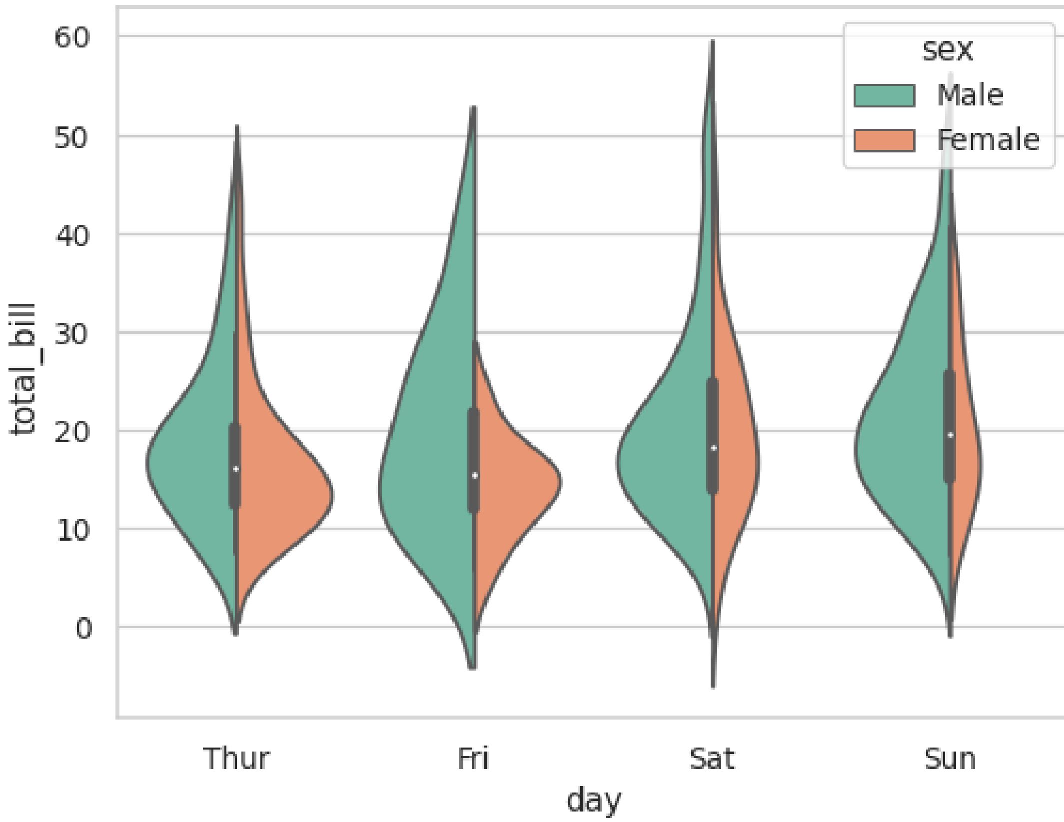
Scale the density relative to the counts across all bins:

```
>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                     data=tips, palette="Set2", split=True,
...                     scale="count", inner="stick", scale_hue=False)
```

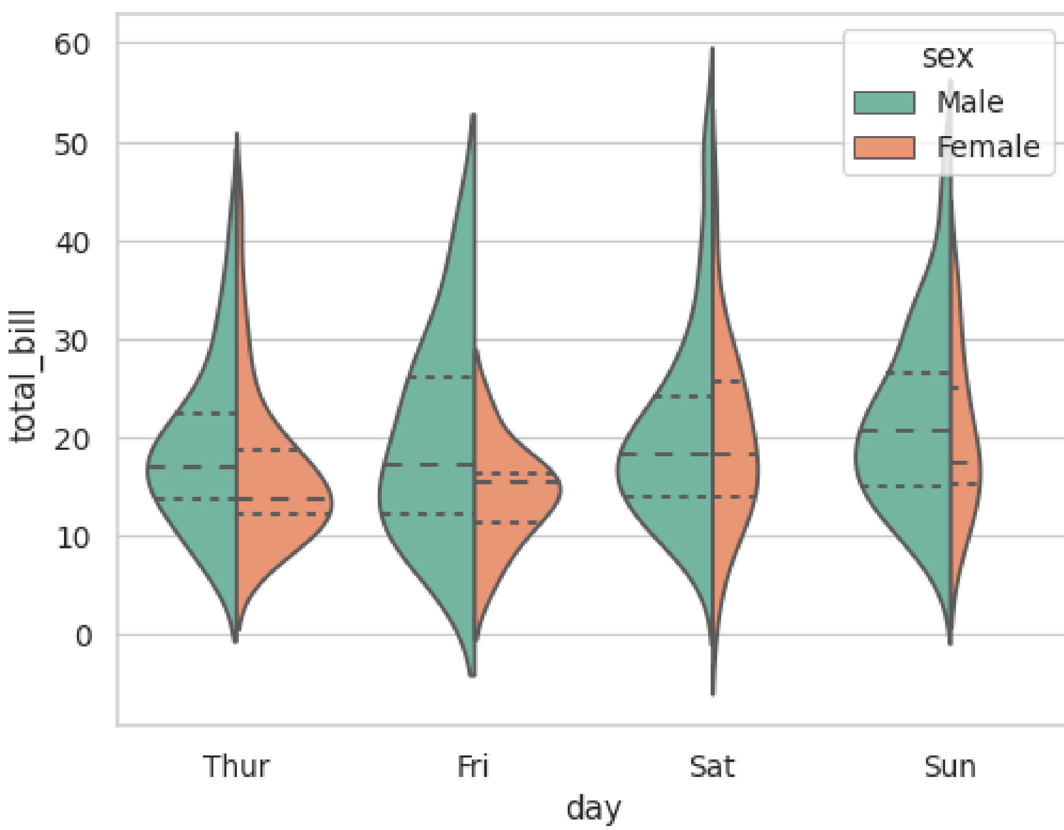
Use a narrow bandwidth to reduce the amount of smoothing:

```
>>> ax = sns.violinplot(x="day", y="total_bill", hue="sex",
...                     data=tips, palette="Set2", split=True,
```

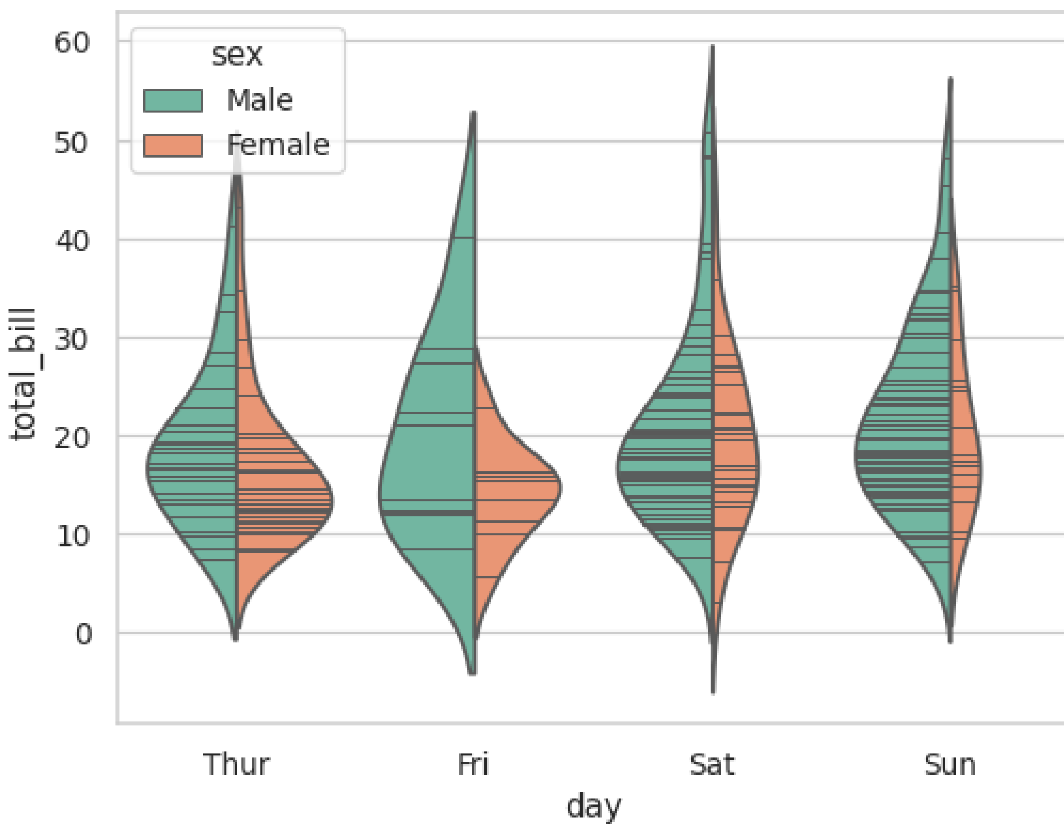
(continues on next page)

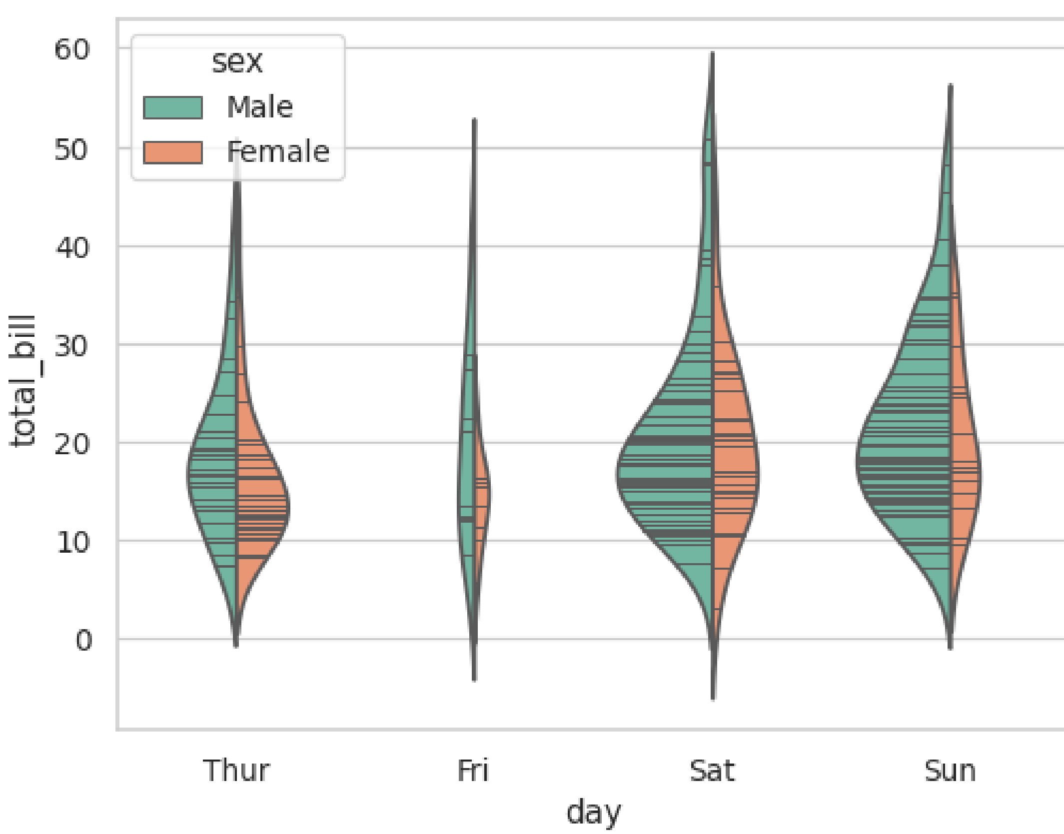


60
50
40
30
20
10
0
total_bill
sex
Male
Female
Thur
Fri
Sat
Sun
day

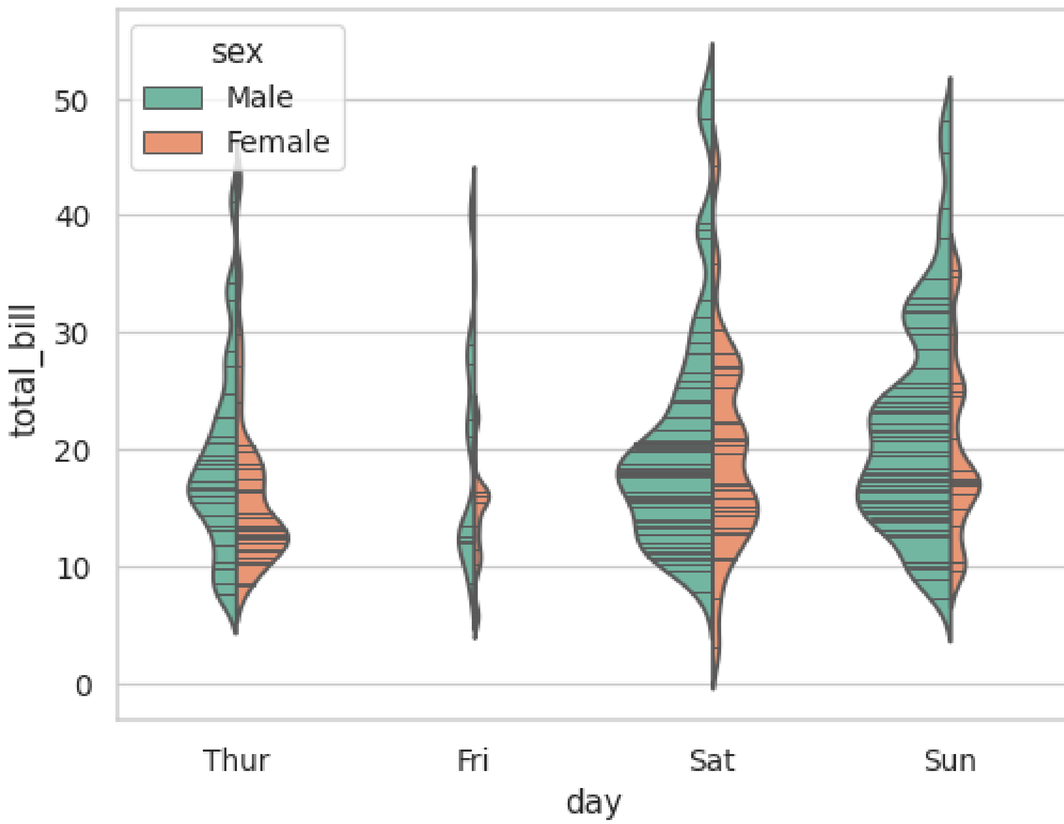
60
50
40
30
20
10
0
total_bill
sex
Male
Female
Thur
Fri
Sat
Sun
day


sex
Male
Female
total_bill
60
50
40
30
20
10
0
Thur
Fri
Sat
Sun
day


sex
Male
Female
total_bill
60
50
40
30
20
10
0
Thur
Fri
Sat
Sun
day

(continued from previous page)

```
...                         scale="count", inner="stick",
...                         scale_hue=False, bw=.2)
```




Draw horizontal violins:

```
>>> planets = sns.load_dataset("planets")
>>> ax = sns.violinplot(x="orbital_period", y="method",
...                     data=planets[planets.orbital_period < 1000],
...                     scale="width", palette="Set3")
```

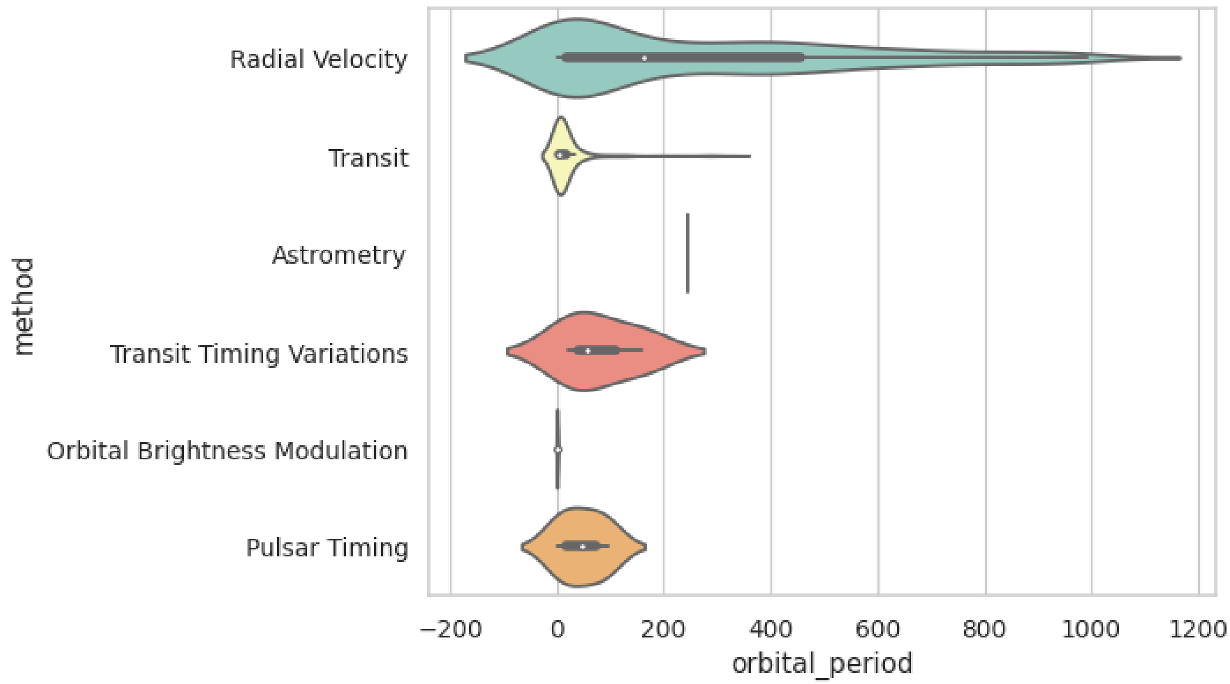
Don't let density extend past extreme values in the data:

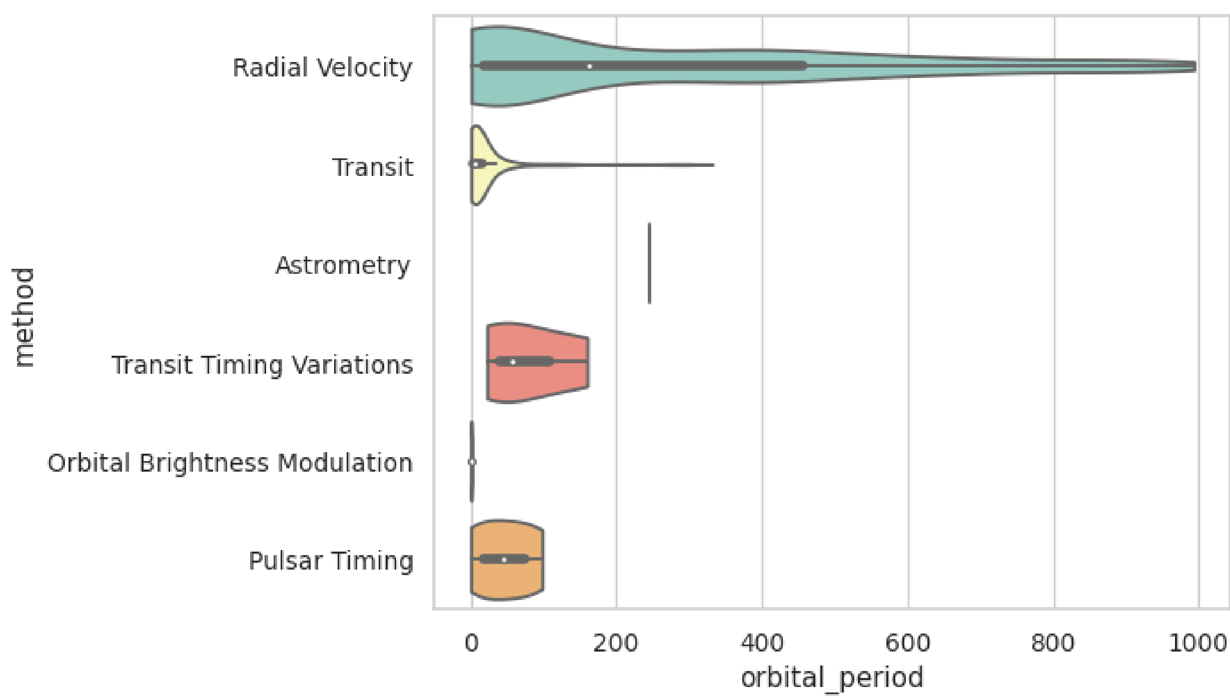
```
>>> ax = sns.violinplot(x="orbital_period", y="method",
...                     data=planets[planets.orbital_period < 1000],
...                     cut=0, scale="width", palette="Set3")
```

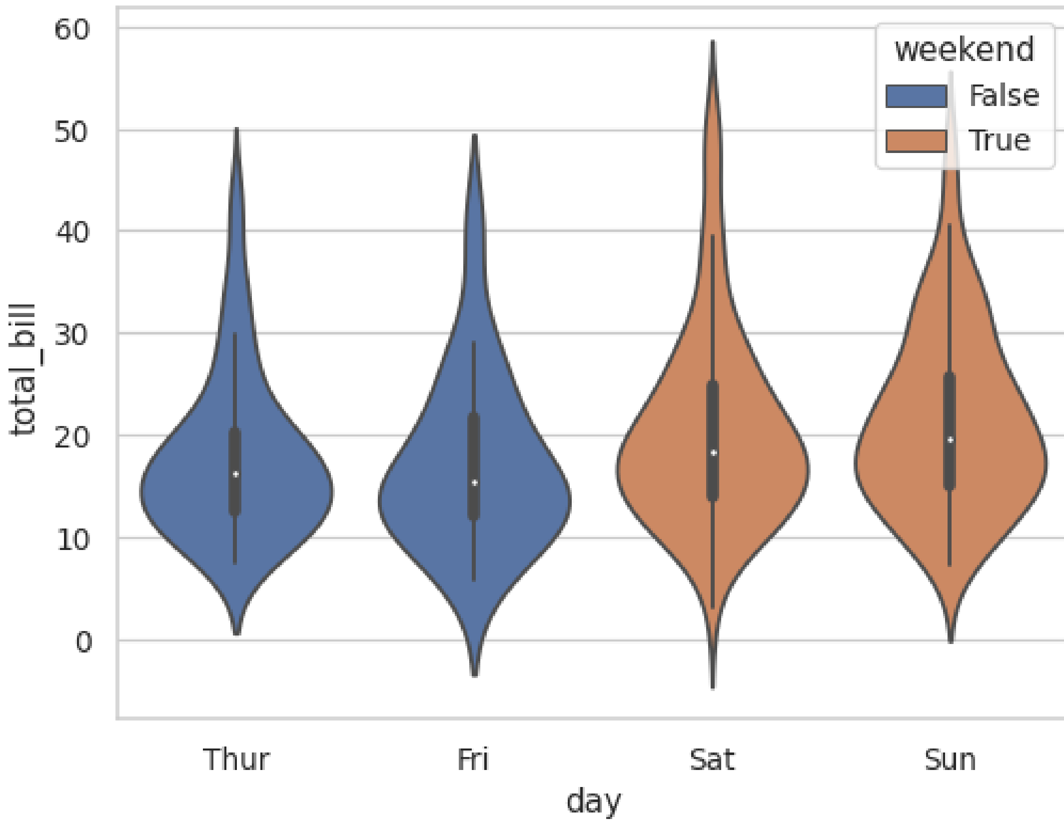
Use `hue` without changing violin position or width:

```
>>> tips["weekend"] = tips["day"].isin(["Sat", "Sun"])
>>> ax = sns.violinplot(x="day", y="total_bill", hue="weekend",
...                     data=tips, dodge=False)
```

Use *catplot()* to combine a *violinplot()* and a *FacetGrid*. This allows grouping within additional categorical variables. Using *catplot()* is safer than using *FacetGrid* directly, as it ensures synchronization of variable order across facets:


Radial Velocity
Transit
Astrometry
Transit Timing Variations
Orbital Brightness Modulation
Pulsar Timing
method
−200
0
200
400
600
800
1000
1200
orbital_period


Radial Velocity
Transit
Astrometry
Transit Timing Variations
Orbital Brightness Modulation
Pulsar Timing
method
0
200
400
600
800
1000
orbital_period




```
>>> g = sns.catplot(x="sex", y="total_bill",
...                 hue="smoker", col="time",
...                 data=tips, kind="violin", split=True,
...                 height=4, aspect=.7);
```

### 5.3.6 seaborn.boxenplot

`seaborn.boxenplot`(*, *x=None*, *y=None*, *hue=None*, *data=None*, *order=None*, *hue_order=None*, *orient=None*, *color=None*, *palette=None*, *saturation=0.75*, *width=0.8*, *dodge=True*, *k_depth='tukey'*, *linewidth=None*, *scale='exponential'*, *outlier_prop=0.007*, *trust_alpha=0.05*, *showfliers=True*, *ax=None*, ***kwargs*)

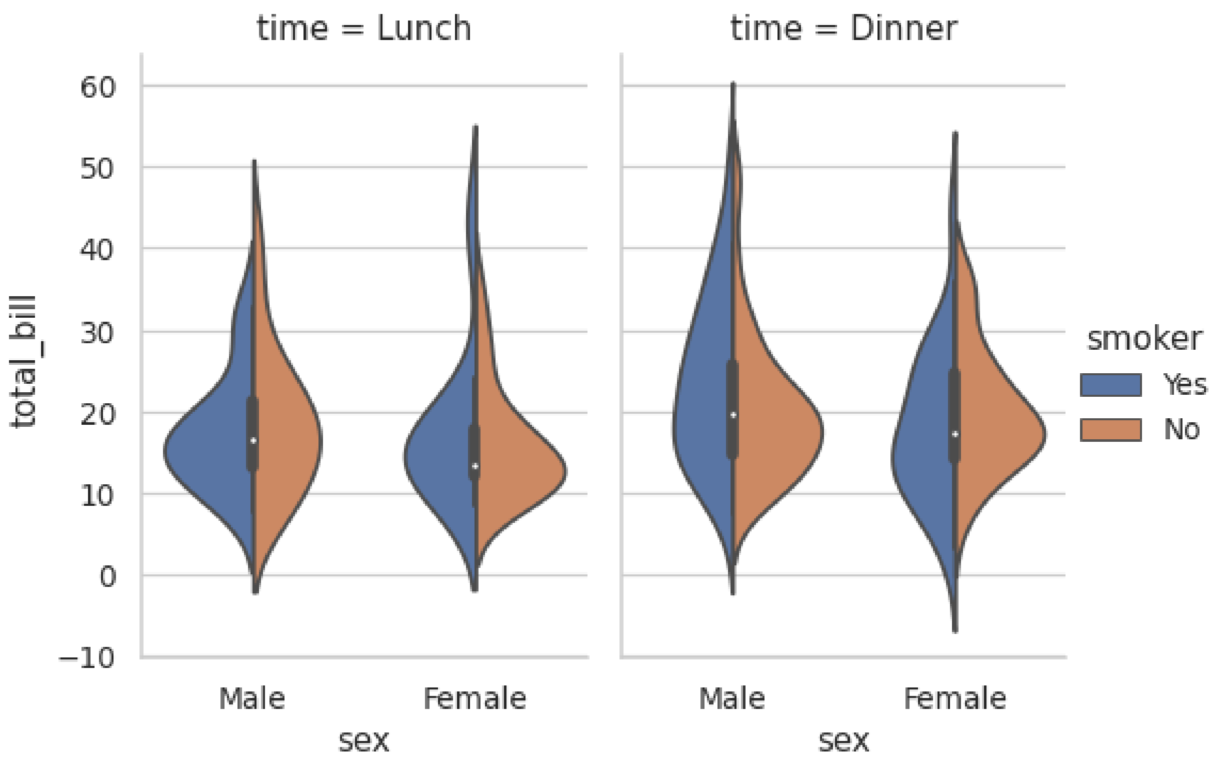
Draw an enhanced box plot for larger datasets.

This style of plot was originally named a "letter value" plot because it shows a large number of quantiles that are defined as "letter values". It is similar to a box plot in plotting a nonparametric representation of a distribution in which all features correspond to actual observations. By plotting more quantiles, it provides more information about the shape of the distribution, particularly in the tails. For a more extensive explanation, you can read the paper that introduced the plot:

https://vita.had.co.nz/papers/letter-value-plot.html

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`, and/or `hue` parameters.




- A "long-form" DataFrame, in which case the x, y, and hue variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, . . . n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

**Parameters**

**x, y, hue** [names of variables in `data` or vector data, optional] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame, array, or list of arrays, optional] Dataset for plotting. If `x` and `y` are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both `x` and `y` are numeric or when plotting wide-form data.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by `color_palette()`, or a dictionary mapping hue levels to matplotlib colors.

**saturation** [float, optional] Proportion of the original saturation to draw colors at. Large patches often look better with slightly desaturated colors, but set this to `1` if you want the plot colors to perfectly match the input color spec.

**width** [float, optional] Width of a full element when not using hue nesting, or width of all the elements for one level of the major grouping variable.

**dodge** [bool, optional] When hue nesting is used, whether elements should be shifted along the categorical axis.

**k_depth** [{"tukey", "proportion", "trustworthy", "full"} or scalar, optional] The number of boxes, and by extension number of percentiles, to draw. All methods are detailed in Wickham's paper. Each makes different assumptions about the number of outliers and leverages different statistical properties. If "proportion", draw no more than `outlier_prop` extreme observations. If "full", draw `log(n)+1` boxes.

**linewidth** [float, optional] Width of the gray lines that frame the plot elements.

**scale** [{"exponential", "linear", "area"}, optional] Method to use for the width of the letter value boxes. All give similar results visually. "linear" reduces the width by a constant linear factor, "exponential" uses the proportion of data not covered, "area" is proportional to the percentage of data covered.

**outlier_prop** [float, optional] Proportion of data believed to be outliers. Must be in the range (0, 1]. Used to determine the number of boxes to plot when `k_depth="proportion"`.

**trust_alpha** [float, optional] Confidence level for a box to be plotted. Used to determine the number of boxes to plot when `k_depth="trustworthy"`. Must be in the range (0, 1).

**showfliers** [bool, optional] If False, suppress the plotting of outliers.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**kwargs** [key, value mappings] Other keyword arguments are passed through to `matplotlib.axes.Axes.plot()` and `matplotlib.axes.Axes.scatter()`.

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot drawn onto it.

**See also:**

**`violinplot`** A combination of boxplot and kernel density estimation.

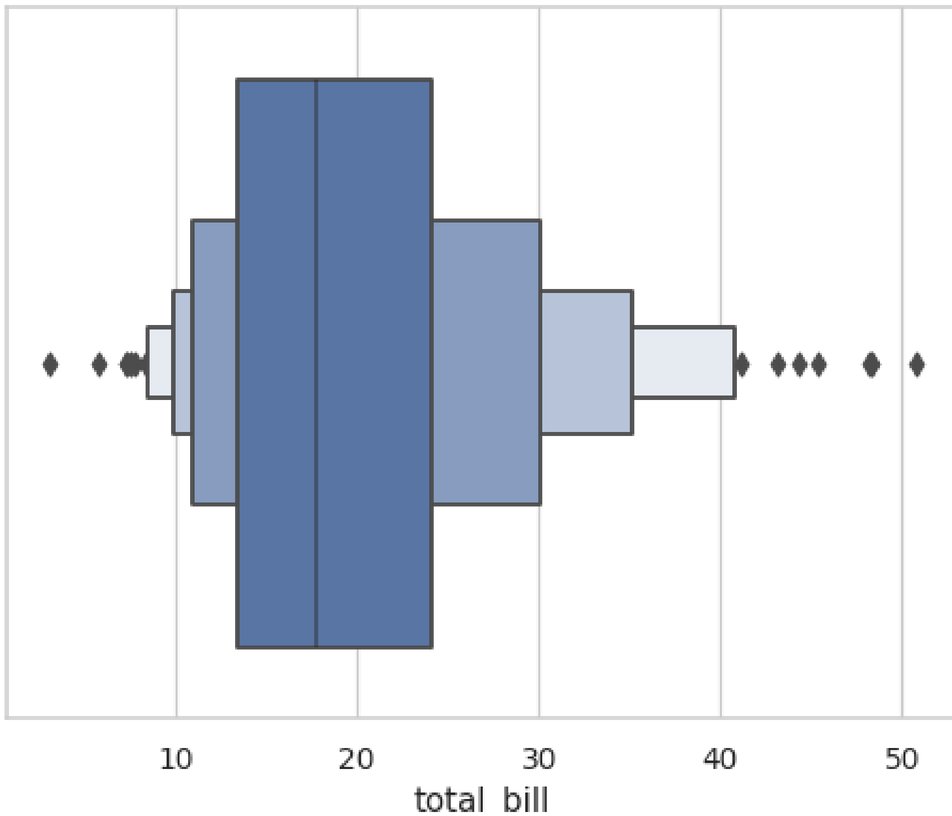
**`boxplot`** A traditional box-and-whisker plot with a similar API.

**`catplot`** Combine a categorical plot with a `FacetGrid`.

### Examples

Draw a single horizontal boxen plot:

```
>>> import seaborn as sns
>>> sns.set_theme(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.boxenplot(x=tips["total_bill"])
```




Draw a vertical boxen plot grouped by a categorical variable:

```
>>> ax = sns.boxenplot(x="day", y="total_bill", data=tips)
```

Draw a letter value plot with nested grouping by two categorical variables:

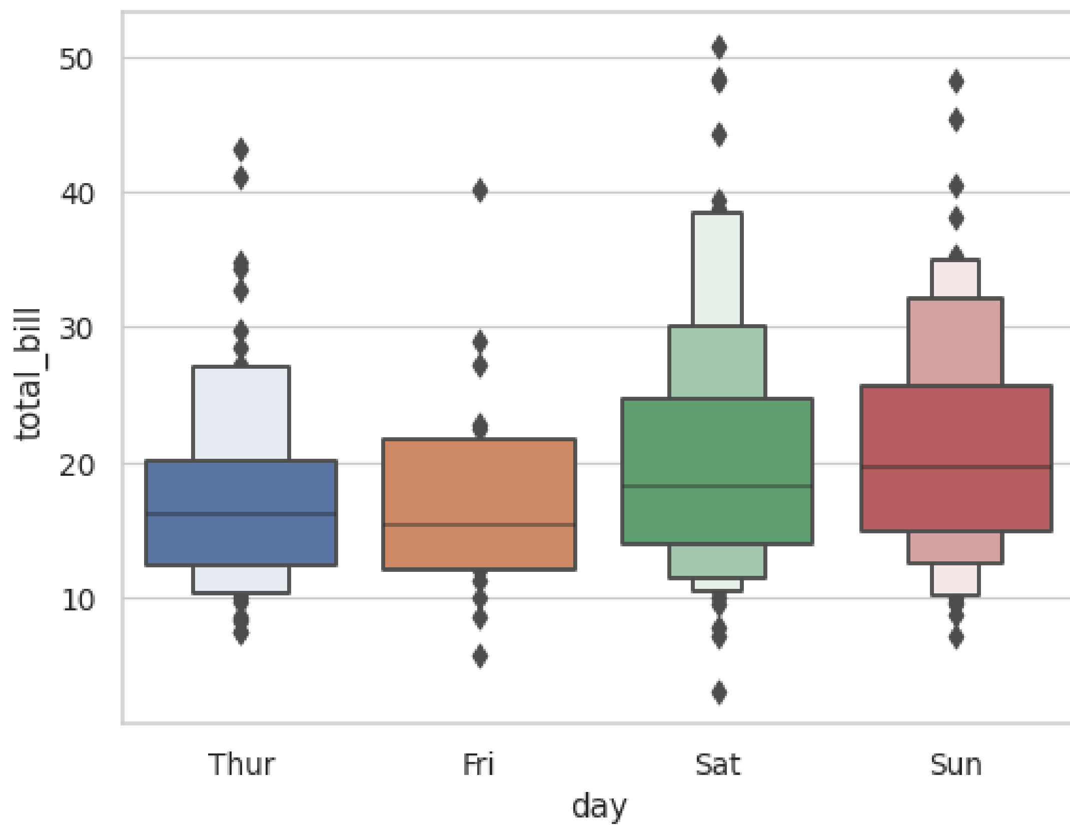
```
>>> ax = sns.boxenplot(x="day", y="total_bill", hue="smoker",
...                    data=tips, palette="Set3")
```

Draw a boxen plot with nested grouping when some bins are empty:

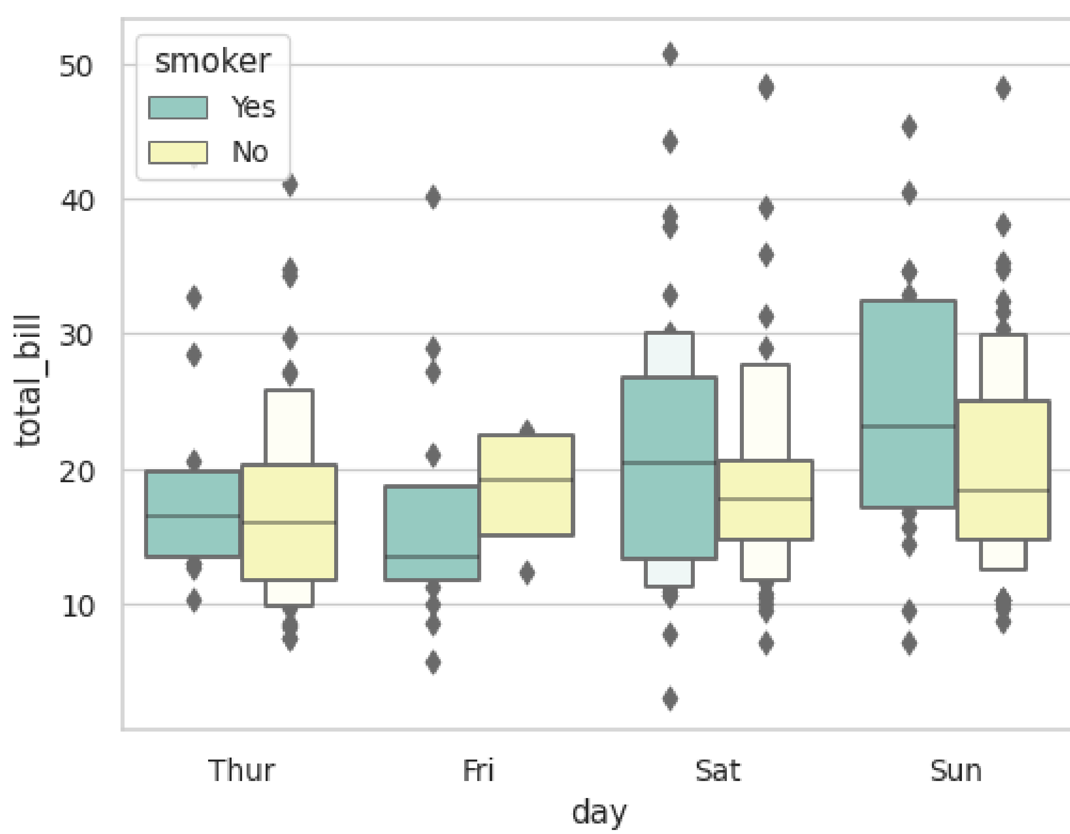
```
>>> ax = sns.boxenplot(x="day", y="total_bill", hue="time",
...                    data=tips, linewidth=2.5)
```

Control box order by passing an explicit order:

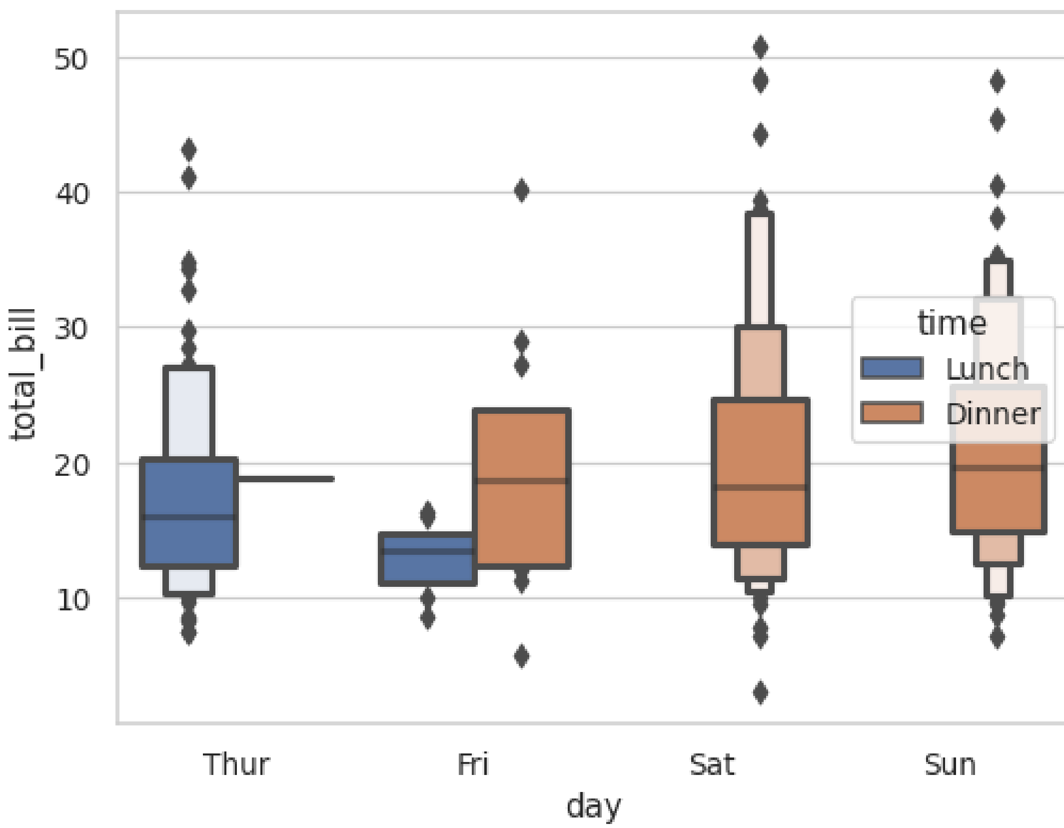
```
>>> ax = sns.boxenplot(x="time", y="tip", data=tips,
...                    order=["Dinner", "Lunch"])
```



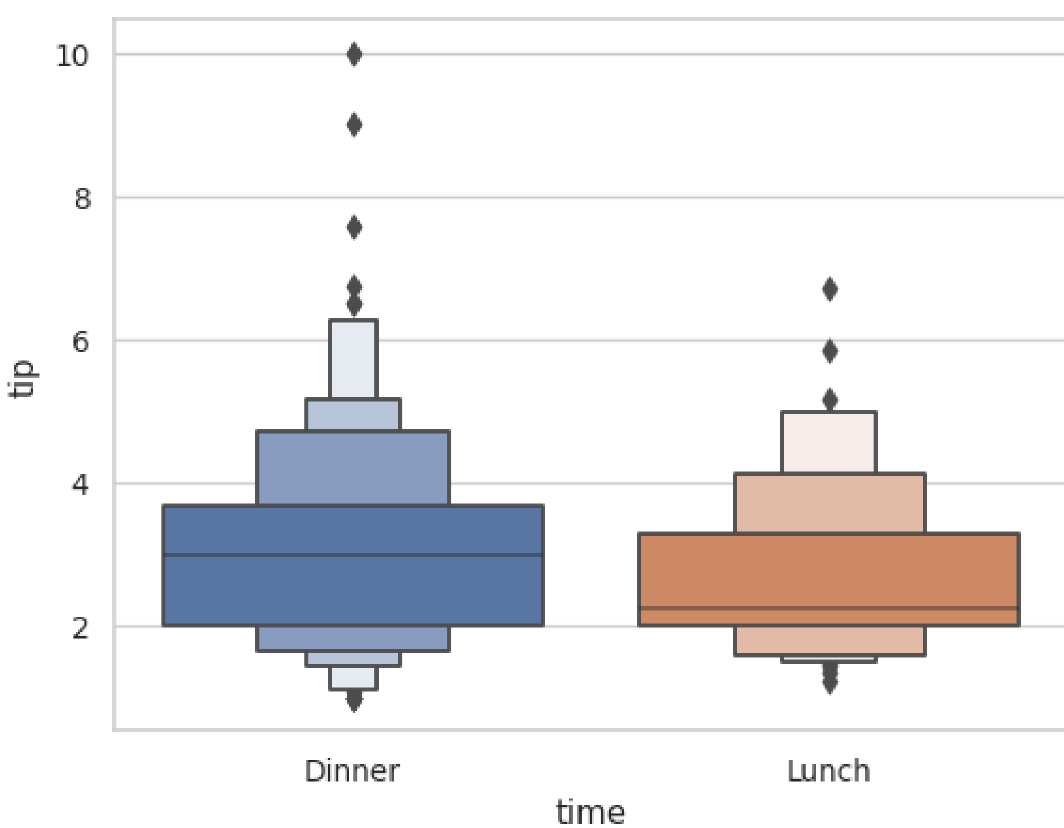
50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day

smoker
Yes
No
50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day



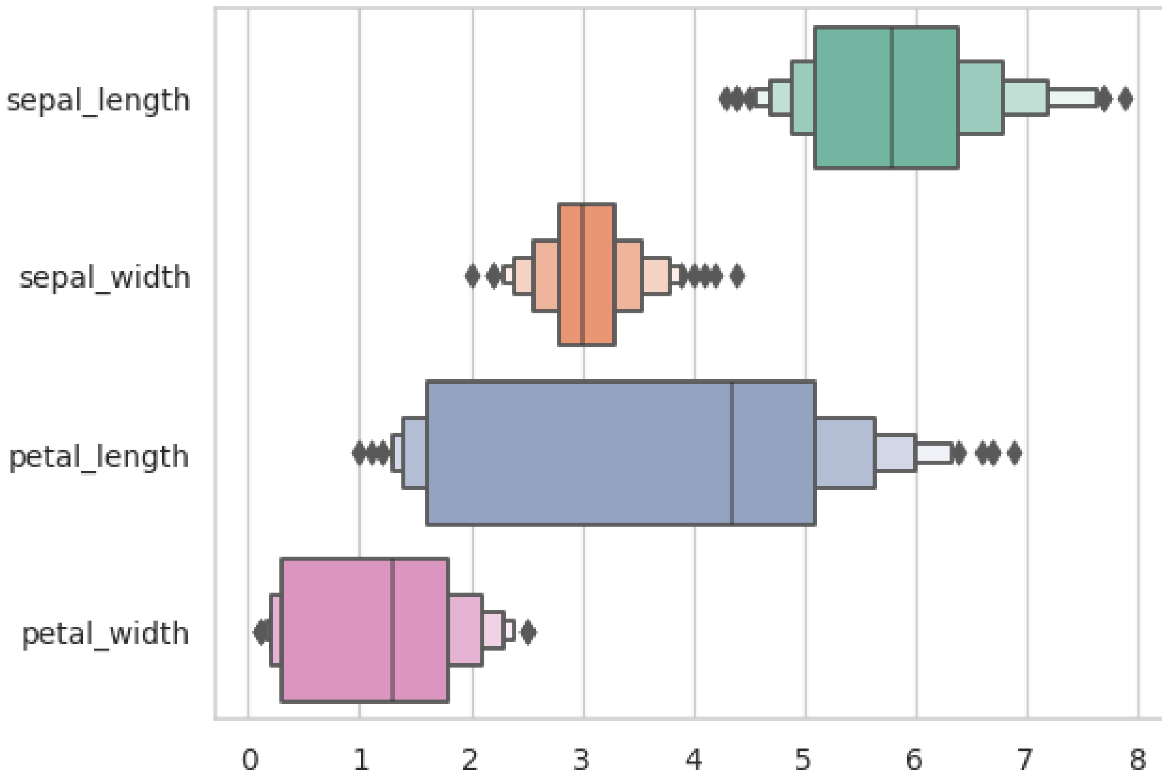
50
40
30
20
10
total_bill
time
Lunch
Dinner
Thur
Fri
Sat
Sun
day

10
8
6
4
2
tip
Dinner
Lunch
time

Draw a boxen plot for each numeric variable in a DataFrame:

```
>>> iris = sns.load_dataset("iris")
>>> ax = sns.boxenplot(data=iris, orient="h", palette="Set2")
```

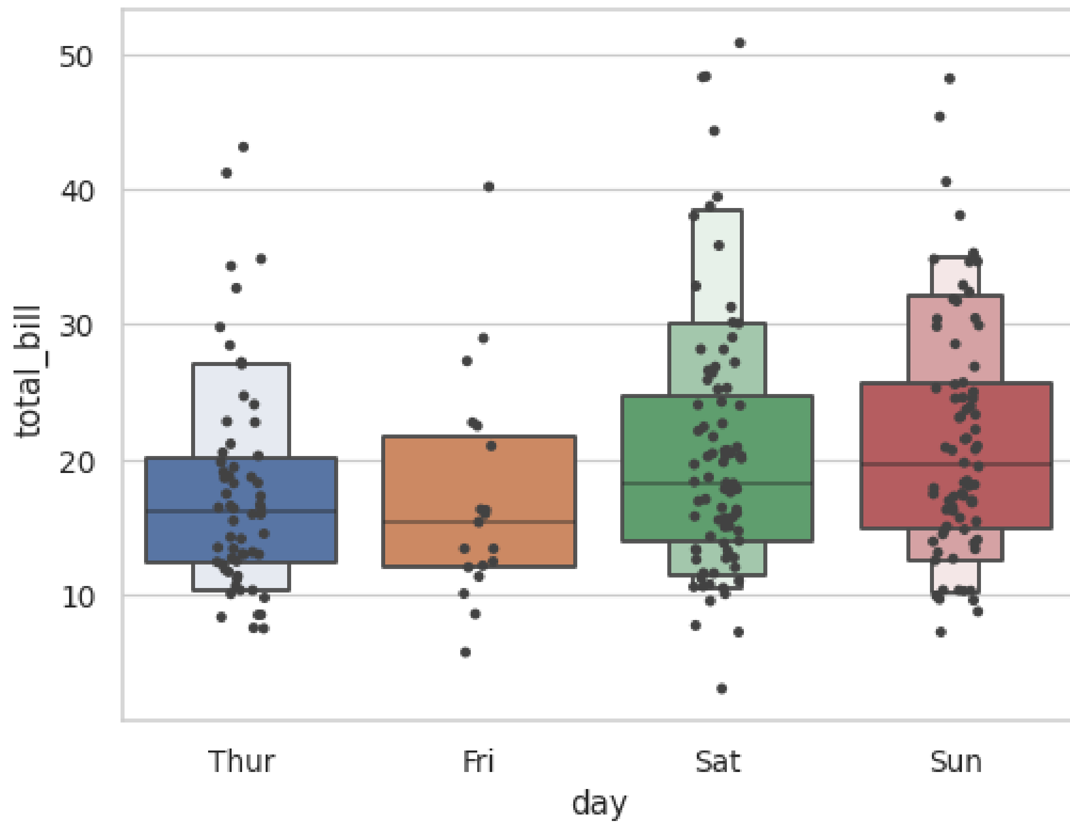


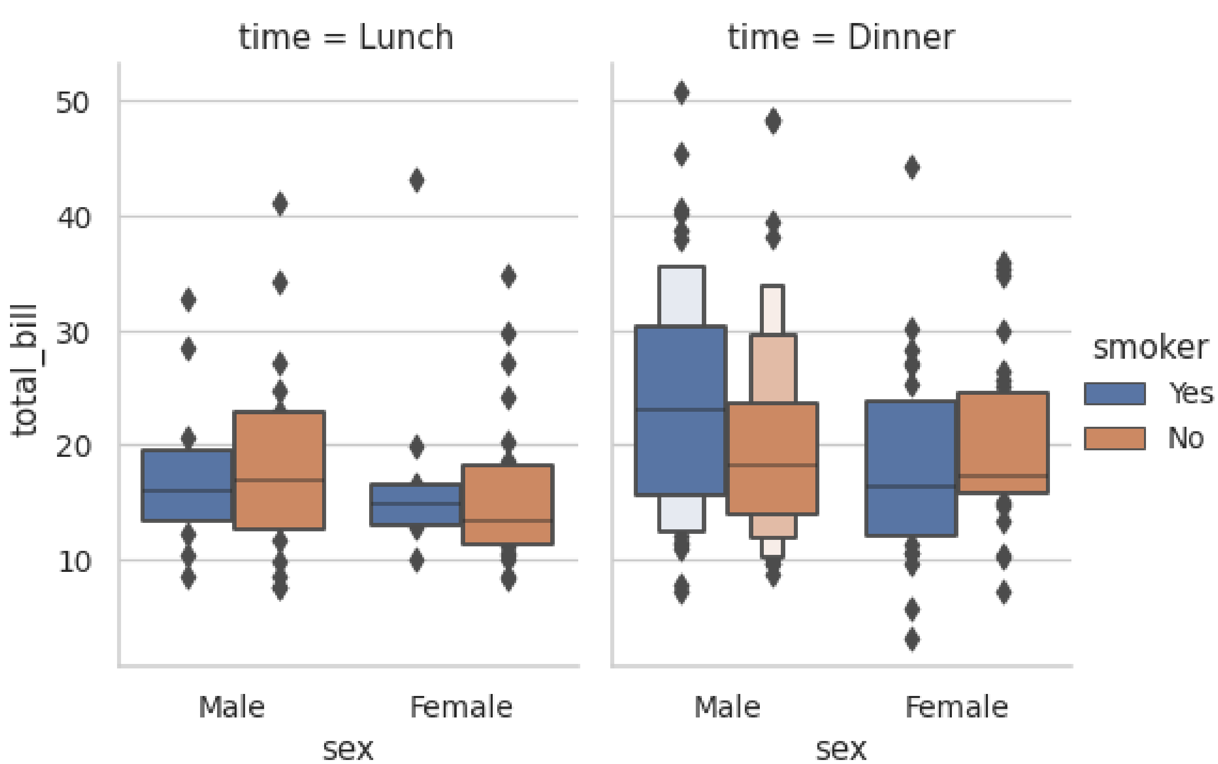

Use `stripplot()` to show the datapoints on top of the boxes:

```
>>> ax = sns.boxenplot(x="day", y="total_bill", data=tips,
...                    showfliers=False)
>>> ax = sns.stripplot(x="day", y="total_bill", data=tips,
...                    size=4, color=".26")
```

Use `catplot()` to combine `boxenplot()` and a `FacetGrid`. This allows grouping within additional categorical variables. Using `catplot()` is safer than using `FacetGrid` directly, as it ensures synchronization of variable order across facets:

```
>>> g = sns.catplot(x="sex", y="total_bill",
...                 hue="smoker", col="time",
...                 data=tips, kind="boxen",
...                 height=4, aspect=.7);
```


50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day


time = Lunch
time = Dinner
50
40
30
20
10
total_bill
Male
Female
sex
Male
Female
sex
smoker
Yes
No

### 5.3.7 seaborn.pointplot

`seaborn.pointplot(*, x=None, y=None, hue=None, data=None, order=None, hue_order=None, estimator=<function mean>, ci=95, n_boot=1000, units=None, seed=None, markers='o', linestyles='-', dodge=False, join=True, scale=1, orient=None, color=None, palette=None, errwidth=None, capsize=None, ax=None, **kwargs)`

Show point estimates and confidence intervals using scatter plot glyphs.

A point plot represents an estimate of central tendency for a numeric variable by the position of scatter plot points and provides some indication of the uncertainty around that estimate using error bars.

Point plots can be more useful than bar plots for focusing comparisons between different levels of one or more categorical variables. They are particularly adept at showing interactions: how the relationship between levels of one categorical variable changes across levels of a second categorical variable. The lines that join each point from the same `hue` level allow interactions to be judged by differences in slope, which is easier for the eyes than comparing the heights of several groups of points or bars.

It is important to keep in mind that a point plot shows only the mean (or other estimator) value, but in many cases it may be more informative to show the distribution of values at each level of the categorical variables. In that case, other approaches such as a box or violin plot may be more appropriate.

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`, and/or `hue` parameters.
- A "long-form" DataFrame, in which case the `x`, `y`, and `hue` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, … n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

**Parameters**

**x, y, hue** [names of variables in `data` or vector data, optional] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame, array, or list of arrays, optional] Dataset for plotting. If `x` and `y` are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**estimator** [callable that maps vector -> scalar, optional] Statistical function to estimate within each categorical bin.

**ci** [float or "sd" or None, optional] Size of confidence intervals to draw around estimated values. If "sd", skip bootstrapping and draw the standard deviation of the observations. If `None`, no bootstrapping will be performed, and error bars will not be drawn.

**n_boot** [int, optional] Number of bootstrap iterations to use when computing confidence intervals.

**units** [name of variable in data or vector data, optional] Identifier of sampling units, which will be used to perform a multilevel bootstrap and account for repeated measures design.

**seed** [int, numpy.random.Generator, or numpy.random.RandomState, optional] Seed or random number generator for reproducible bootstrapping.

**markers** [string or list of strings, optional] Markers to use for each of the hue levels.

**linestyles** [string or list of strings, optional] Line styles to use for each of the hue levels.

**dodge** [bool or float, optional] Amount to separate the points for each level of the hue variable along the categorical axis.

**join** [bool, optional] If True, lines will be drawn between point estimates at the same hue level.

**scale** [float, optional] Scale factor for the plot elements.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both x and y are numeric or when plotting wide-form data.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the hue variable. Should be something that can be interpreted by *color_palette()*, or a dictionary mapping hue levels to matplotlib colors.

**errwidth** [float, optional] Thickness of error bar lines (and caps).

**capsize** [float, optional] Width of the "caps" on error bars.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot drawn onto it.

**See also:**

***barplot*** Show point estimates and confidence intervals using bars.

***catplot*** Combine a categorical plot with a *FacetGrid*.

## Examples

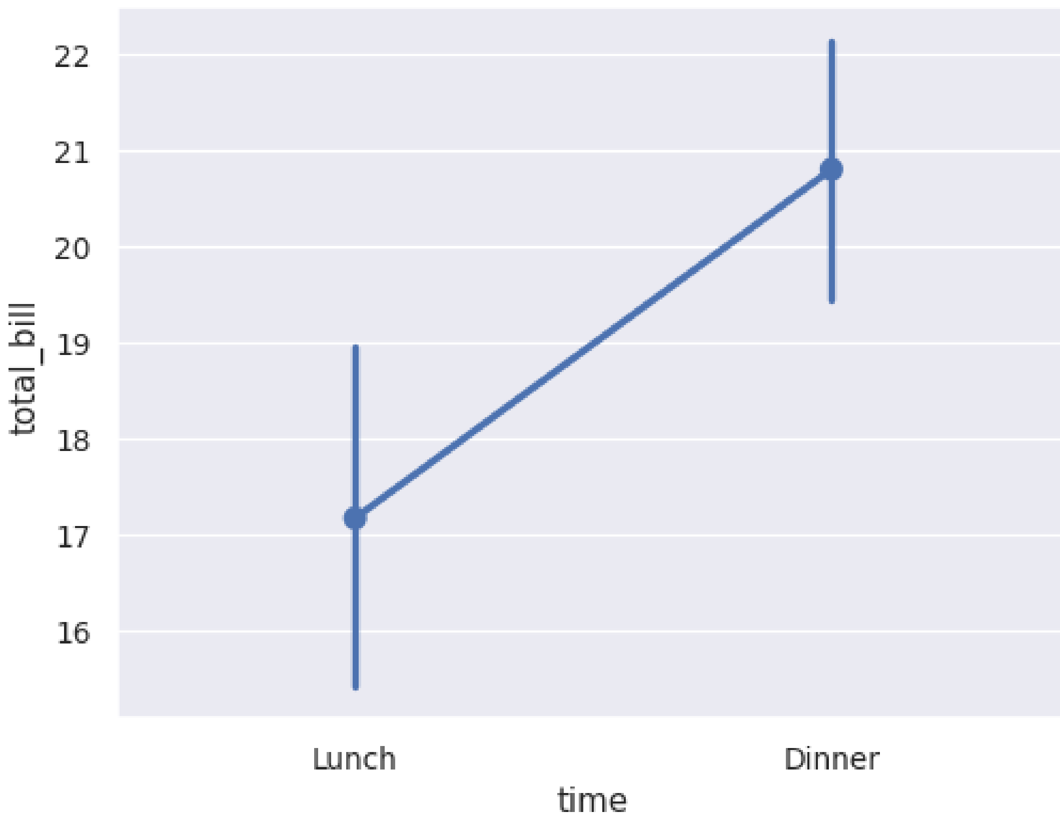
Draw a set of vertical point plots grouped by a categorical variable:

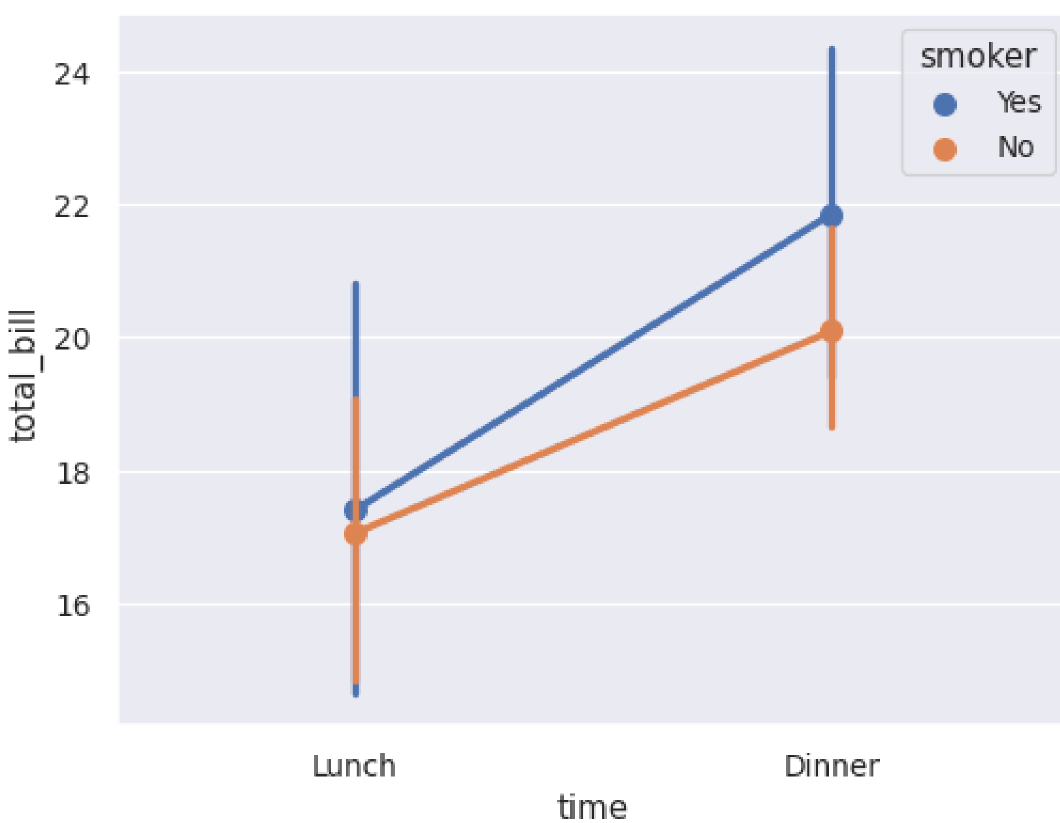
```
>>> import seaborn as sns
>>> sns.set_theme(style="darkgrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.pointplot(x="time", y="total_bill", data=tips)
```

Draw a set of vertical points with nested grouping by a two variables:

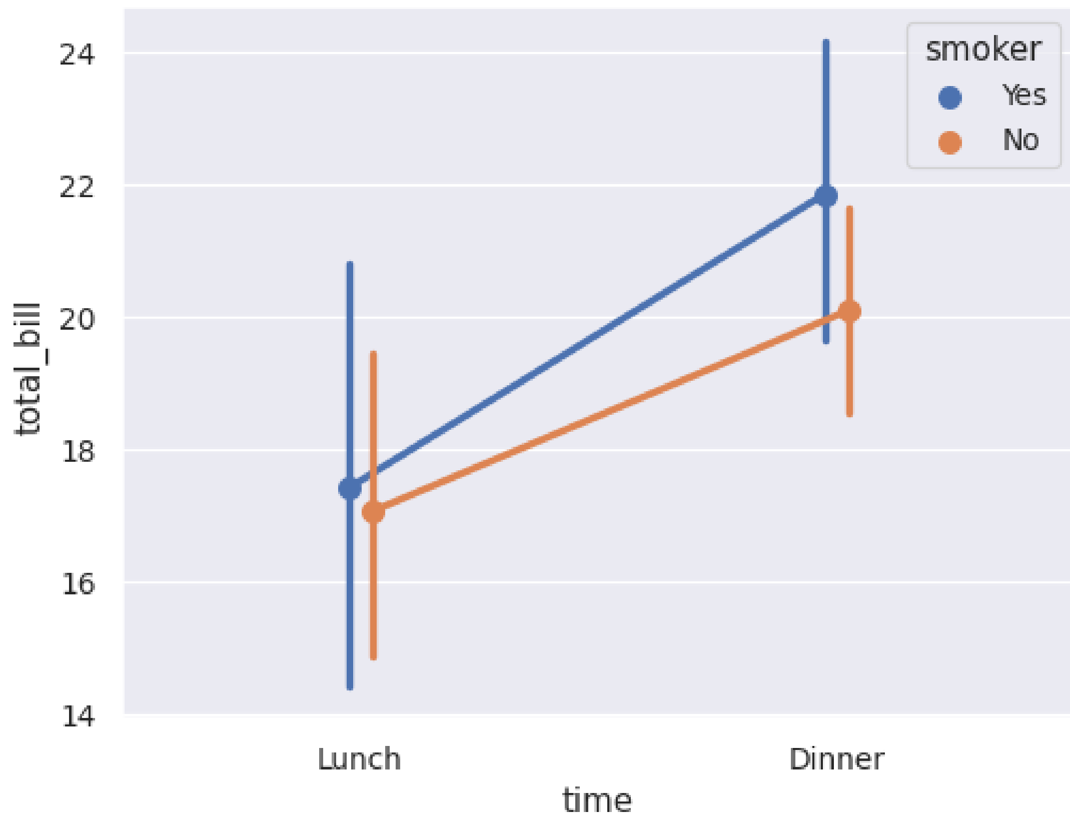
```
>>> ax = sns.pointplot(x="time", y="total_bill", hue="smoker",
...                    data=tips)
```

Separate the points for different hue levels along the categorical axis:


total_bill
22
21
20
19
18
17
16
Lunch
Dinner
time


total_bill
24
22
20
18
16
Lunch
Dinner
time
smoker
Yes
No

```
>>> ax = sns.pointplot(x="time", y="total_bill", hue="smoker",
...                    data=tips, dodge=True)
```




Use a different marker and line style for the hue levels:

```
>>> ax = sns.pointplot(x="time", y="total_bill", hue="smoker",
...                    data=tips,
...                    markers=["o", "x"],
...                    linestyles=["-", "--"])
```

Draw a set of horizontal points:

```
>>> ax = sns.pointplot(x="tip", y="day", data=tips)
```

Don't draw a line connecting each point:

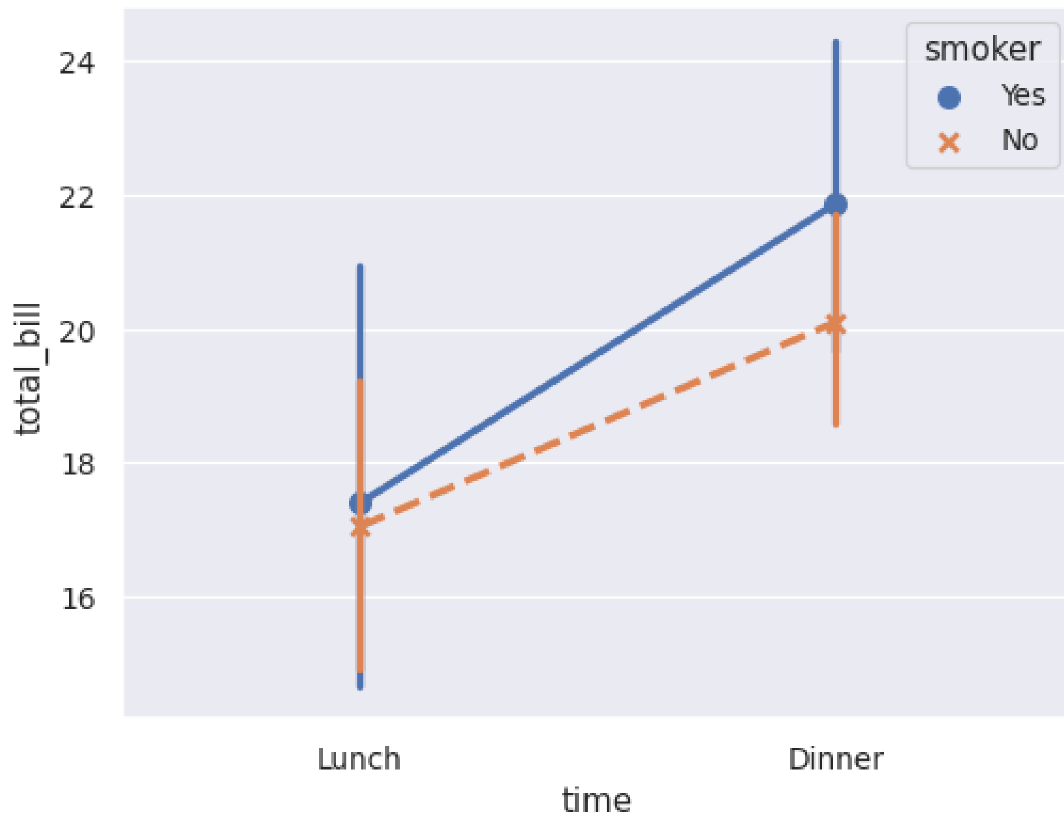
```
>>> ax = sns.pointplot(x="tip", y="day", data=tips, join=False)
```

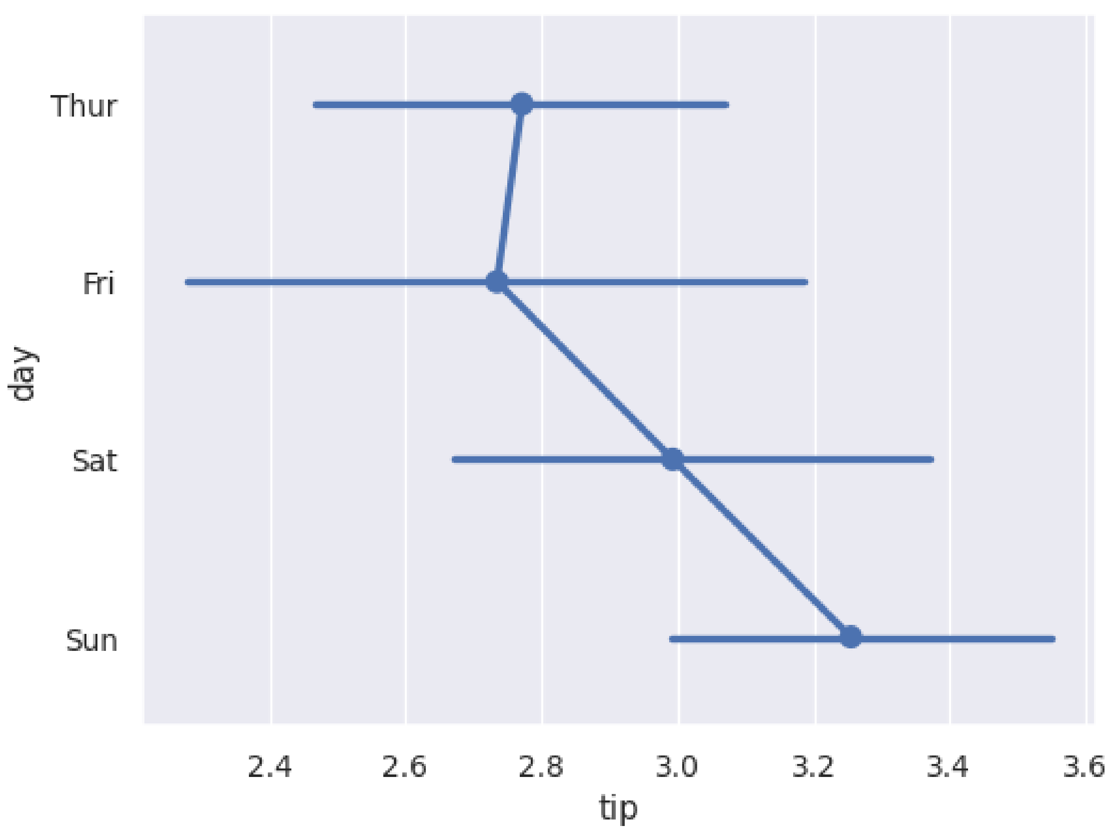
Use a different color for a single-layer plot:

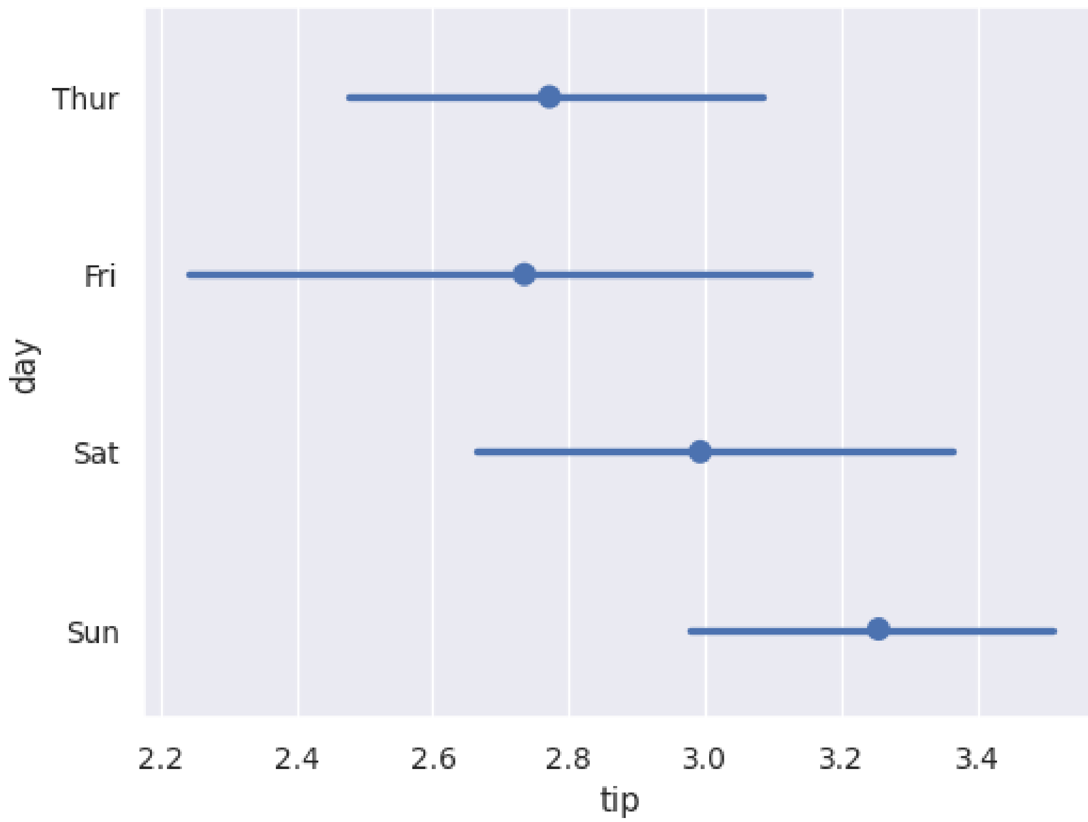
```
>>> ax = sns.pointplot(x="time", y="total_bill", data=tips,
...                    color="#bb3f3f")
```

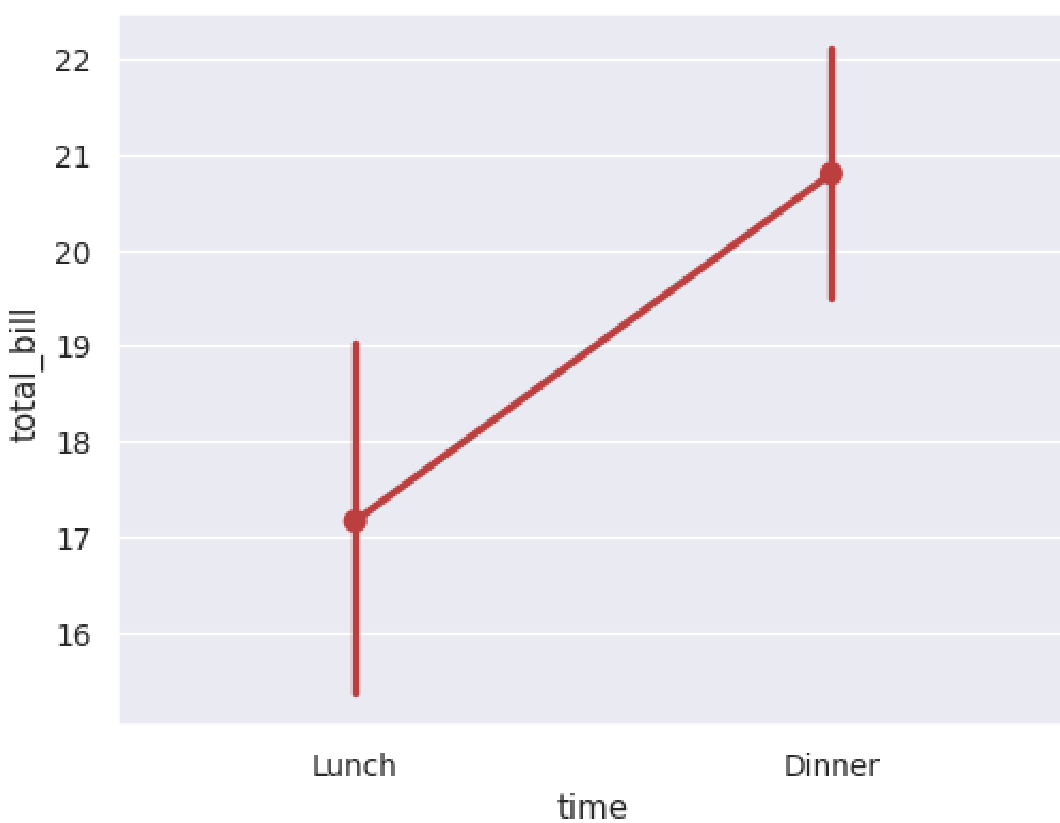
Use a different color palette for the points:

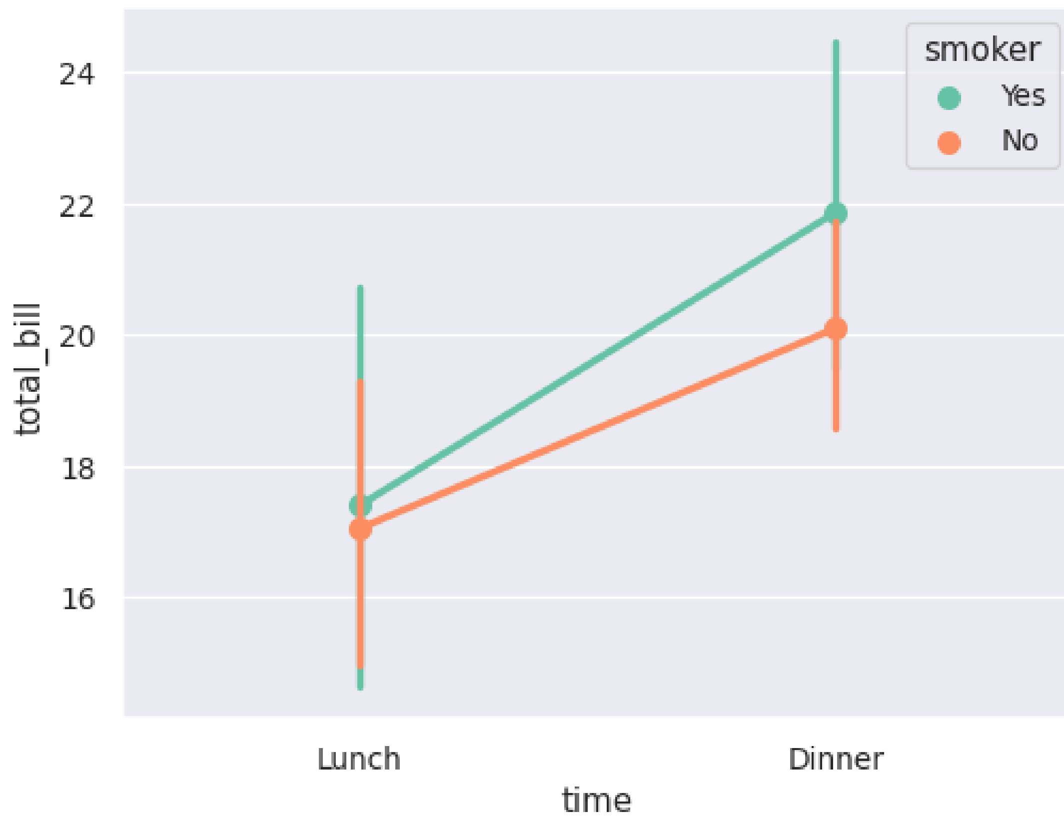
```
>>> ax = sns.pointplot(x="time", y="total_bill", hue="smoker",
...                    data=tips, palette="Set2")
```


smoker
Yes
No
24
22
20
18
16
total_bill
Lunch
Dinner
time


Thur
Fri
Sat
Sun
day
2.4
2.6
2.8
3.0
3.2
3.4
3.6
tip


Thur
Fri
Sat
Sun
day
2.2
2.4
2.6
2.8
3.0
3.2
3.4
tip


22
21
20
19
18
17
16
total_bill
Lunch
Dinner
time




Control point order by passing an explicit order:

```
>>> ax = sns.pointplot(x="time", y="tip", data=tips,
...                    order=["Dinner", "Lunch"])
```

Use median as the estimate of central tendency:

```
>>> from numpy import median
>>> ax = sns.pointplot(x="day", y="tip", data=tips, estimator=median)
```

Show the standard error of the mean with the error bars:

```
>>> ax = sns.pointplot(x="day", y="tip", data=tips, ci=68)
```

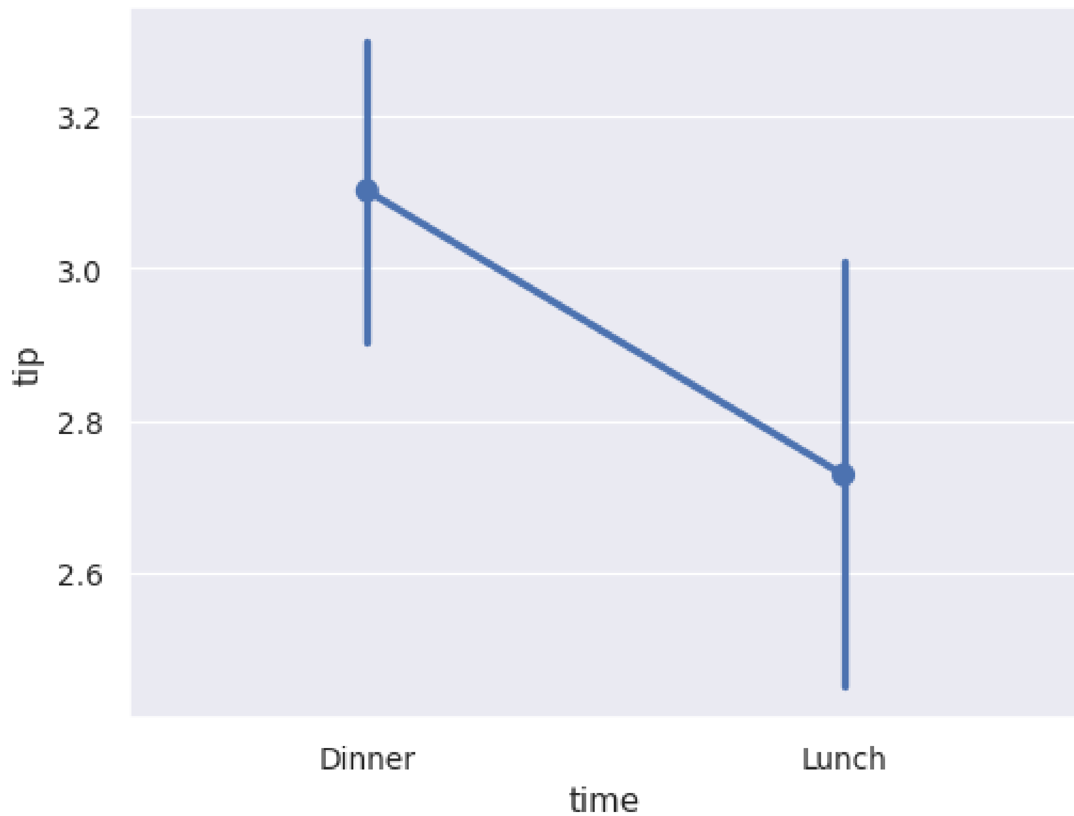
Show standard deviation of observations instead of a confidence interval:

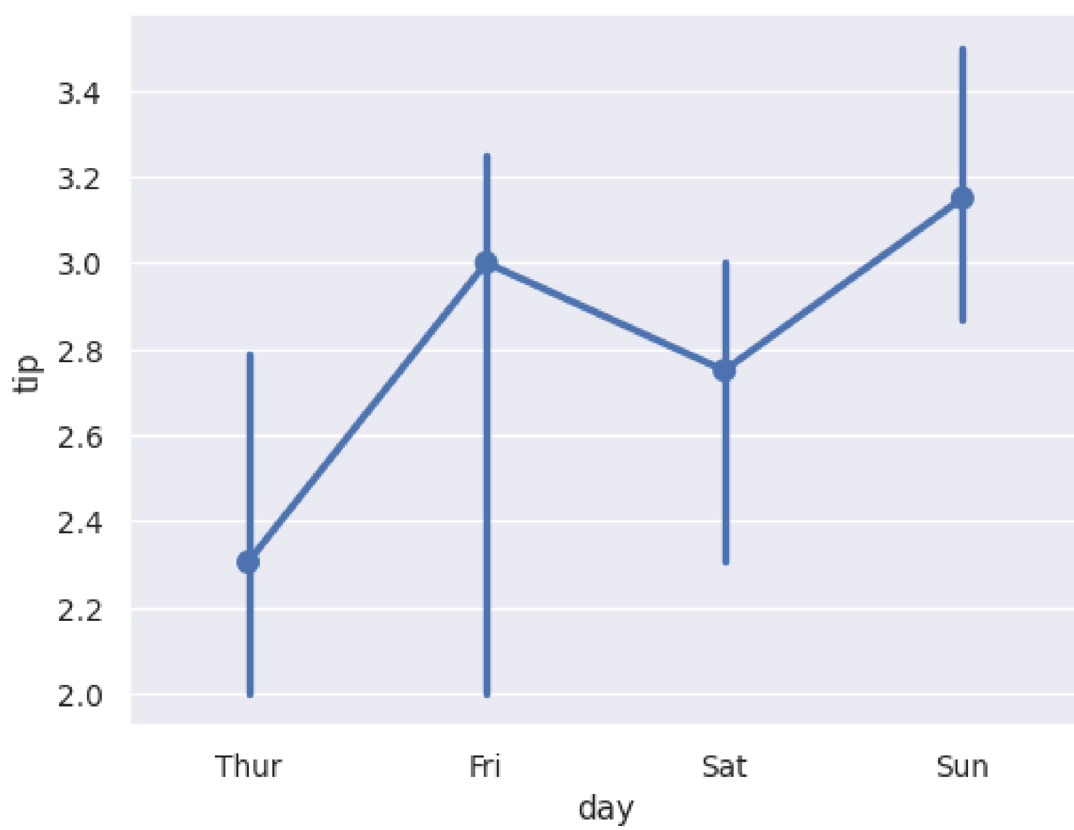
```
>>> ax = sns.pointplot(x="day", y="tip", data=tips, ci="sd")
```

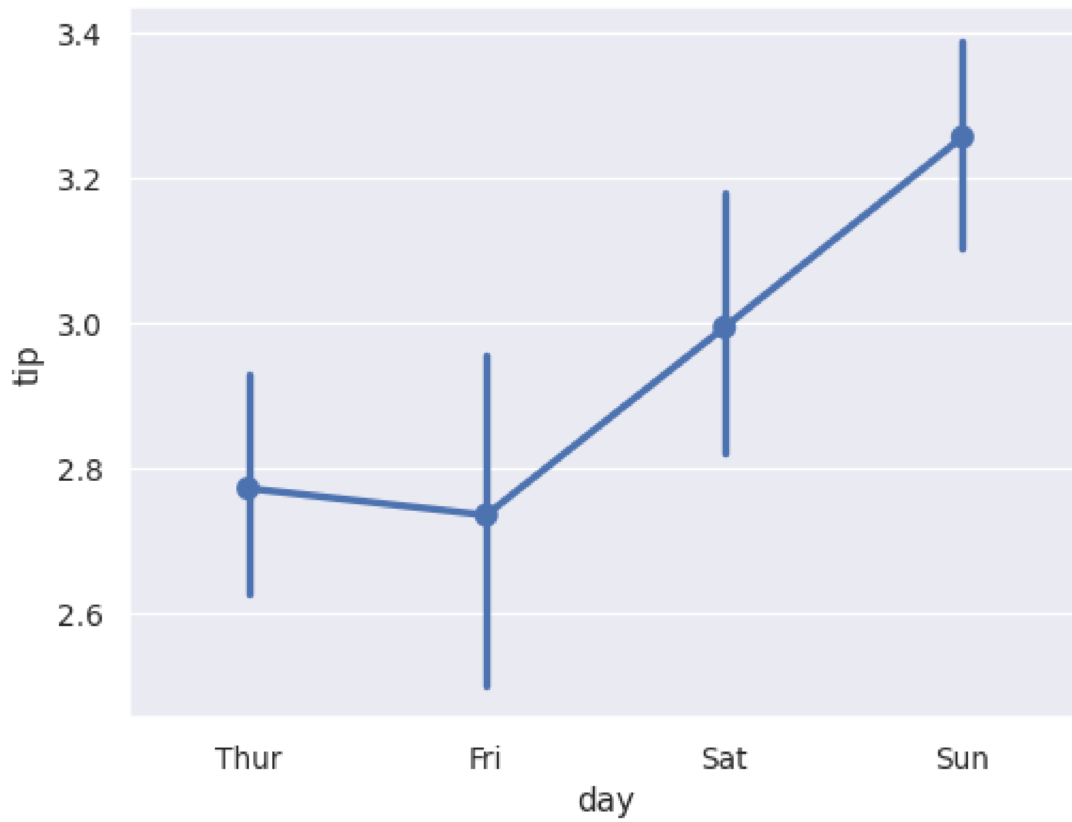
Add "caps" to the error bars:

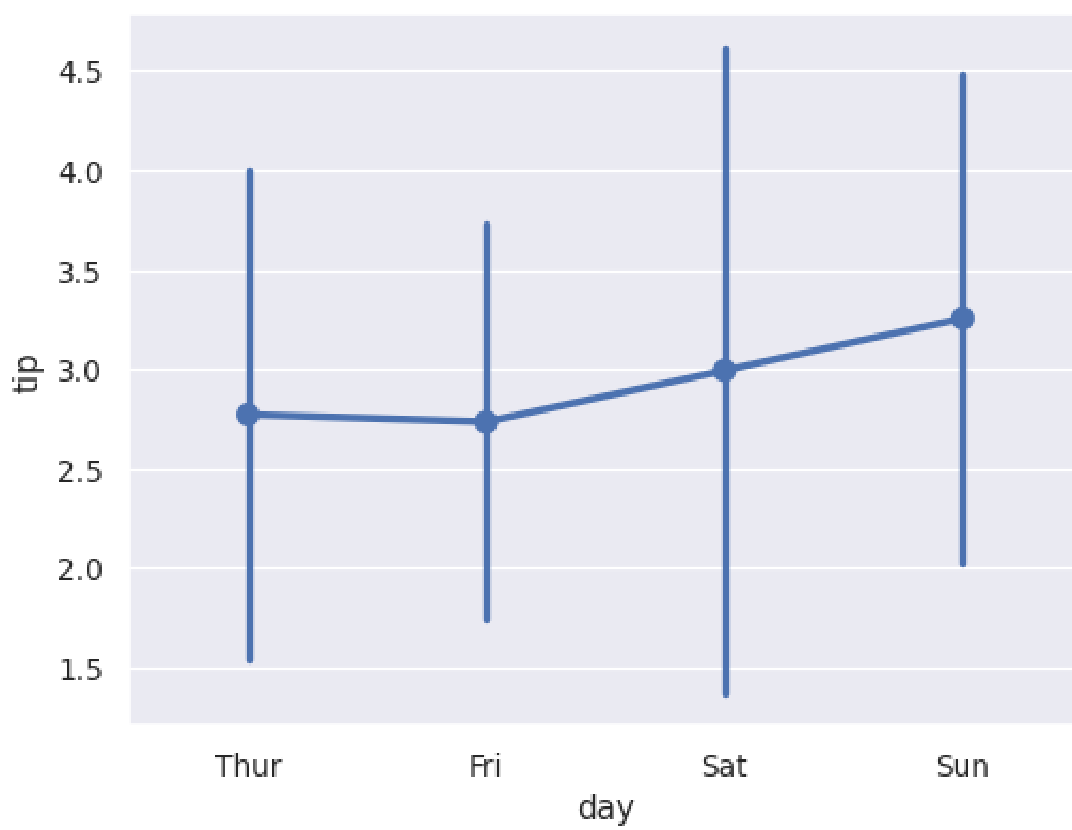
```
>>> ax = sns.pointplot(x="day", y="tip", data=tips, capsize=.2)
```

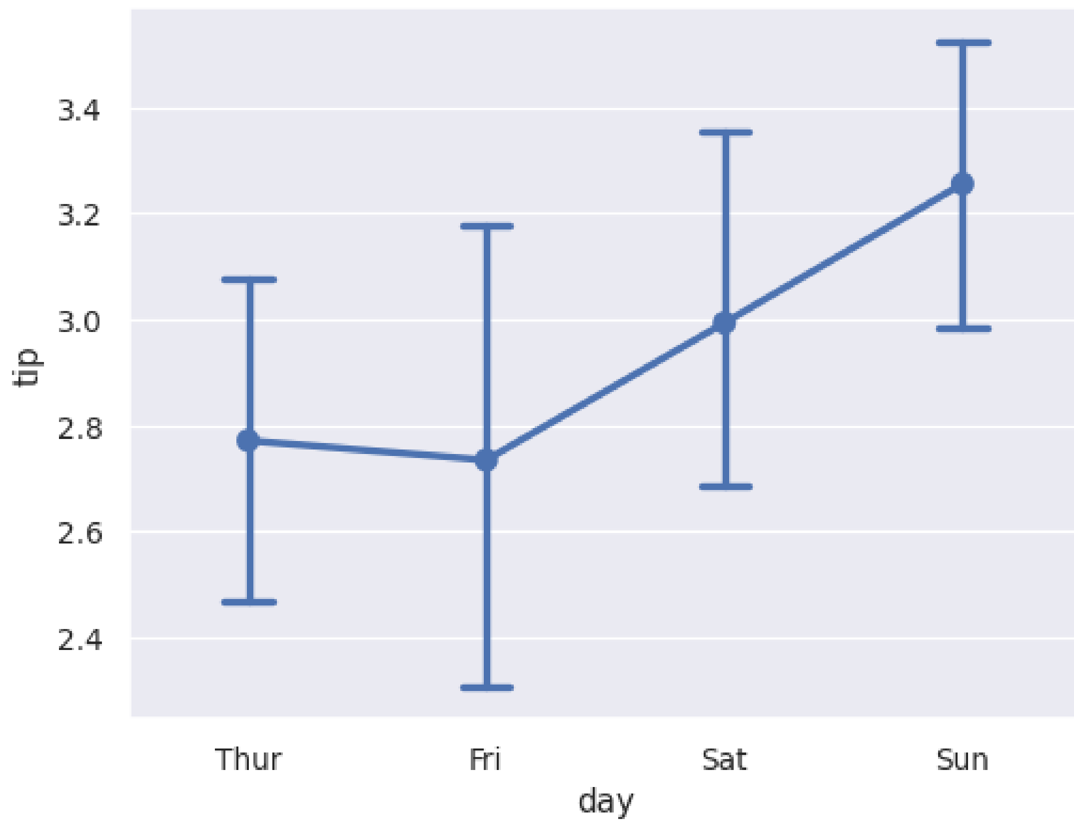
Use `catplot()` to combine a `pointplot()` and a `FacetGrid`. This allows grouping within additional categorical variables. Using `catplot()` is safer than using `FacetGrid` directly, as it ensures synchronization of variable order across facets:


3.2
3.0
2.8
2.6
tip
Dinner
Lunch
time


3.4
3.2
3.0
2.8
2.6
2.4
2.2
2.0
tip
Thur
Fri
Sat
Sun
day


3.4
3.2
3.0
2.8
2.6
tip
Thur
Fri
Sat
Sun
day


4.5
4.0
3.5
3.0
2.5
2.0
1.5
tip
Thur
Fri
Sat
Sun
day




```
>>> g = sns.catplot(x="sex", y="total_bill",
...                 hue="smoker", col="time",
...                 data=tips, kind="point",
...                 dodge=True,
...                 height=4, aspect=.7);
```

### 5.3.8 seaborn.barplot

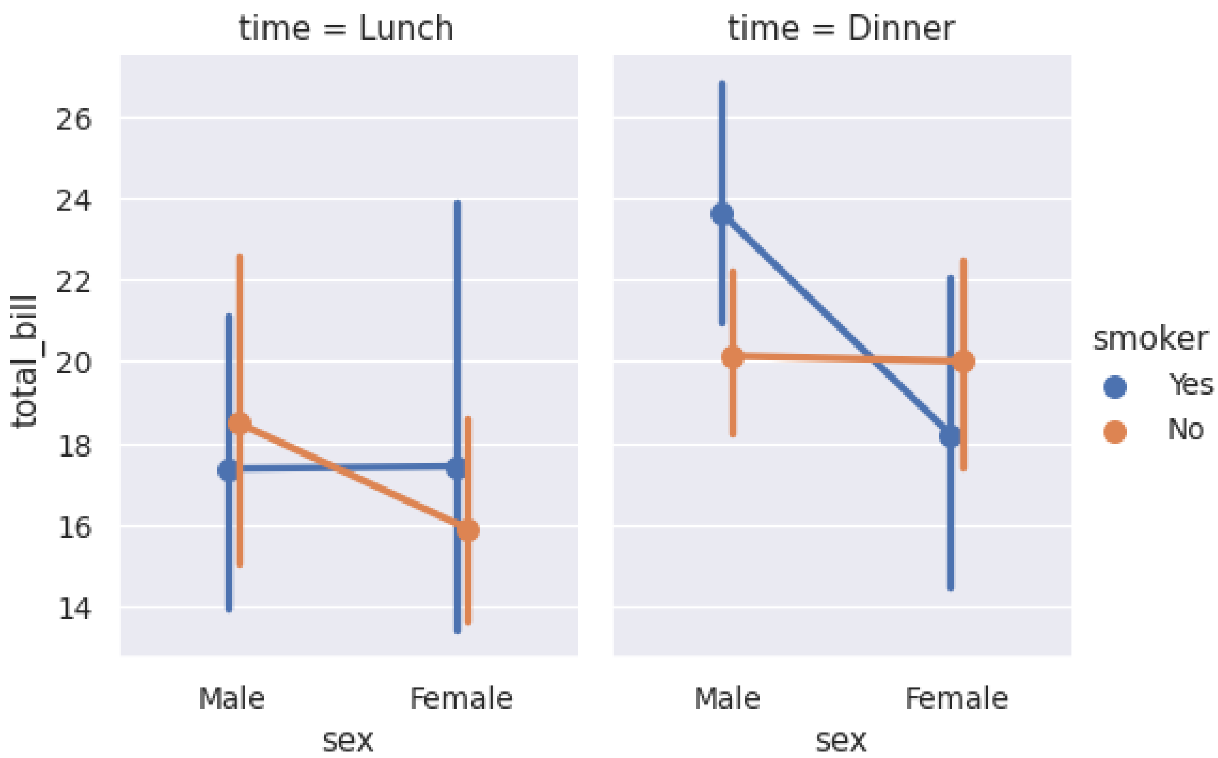
seaborn.**barplot**(**, x=None, y=None, hue=None, data=None, order=None, hue_order=None, estimator=<function mean>, ci=95, n_boot=1000, units=None, seed=None, orient=None, color=None, palette=None, saturation=0.75, errcolor='.26', errwidth=None, capsize=None, dodge=True, ax=None, **kwargs*)

Show point estimates and confidence intervals as rectangular bars.

A bar plot represents an estimate of central tendency for a numeric variable with the height of each rectangle and provides some indication of the uncertainty around that estimate using error bars. Bar plots include 0 in the quantitative axis range, and they are a good choice when 0 is a meaningful value for the quantitative variable, and you want to make comparisons against it.

For datasets where 0 is not a meaningful value, a point plot will allow you to focus on differences between levels of one or more categorical variables.

It is also important to keep in mind that a bar plot shows only the mean (or other estimator) value, but in many cases it may be more informative to show the distribution of values at each level of the categorical variables. In that case, other approaches such as a box or violin plot may be more appropriate.




Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`, and/or `hue` parameters.
- A "long-form" DataFrame, in which case the `x`, `y`, and `hue` variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, … n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

**Parameters**

**x, y, hue** [names of variables in `data` or vector data, optional] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame, array, or list of arrays, optional] Dataset for plotting. If `x` and `y` are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**estimator** [callable that maps vector -> scalar, optional] Statistical function to estimate within each categorical bin.

**ci** [float or "sd" or None, optional] Size of confidence intervals to draw around estimated values. If "sd", skip bootstrapping and draw the standard deviation of the observations. If `None`, no bootstrapping will be performed, and error bars will not be drawn.

**n_boot** [int, optional] Number of bootstrap iterations to use when computing confidence intervals.

**units** [name of variable in `data` or vector data, optional] Identifier of sampling units, which will be used to perform a multilevel bootstrap and account for repeated measures design.

**seed** [int, numpy.random.Generator, or numpy.random.RandomState, optional] Seed or random number generator for reproducible bootstrapping.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both `x` and `y` are numeric or when plotting wide-form data.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by *`color_palette()`*, or a dictionary mapping hue levels to matplotlib colors.

**saturation** [float, optional] Proportion of the original saturation to draw colors at. Large patches often look better with slightly desaturated colors, but set this to `1` if you want the plot colors to perfectly match the input color spec.

**errcolor** [matplotlib color] Color for the lines that represent the confidence interval.

**errwidth** [float, optional] Thickness of error bar lines (and caps).

**capsize** [float, optional] Width of the "caps" on error bars.

**dodge** [bool, optional] When hue nesting is used, whether elements should be shifted along the categorical axis.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**kwargs** [key, value mappings] Other keyword arguments are passed through to `matplotlib.axes.Axes.bar()`.

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot drawn onto it.

**See also:**

***`countplot`*** Show the counts of observations in each categorical bin.

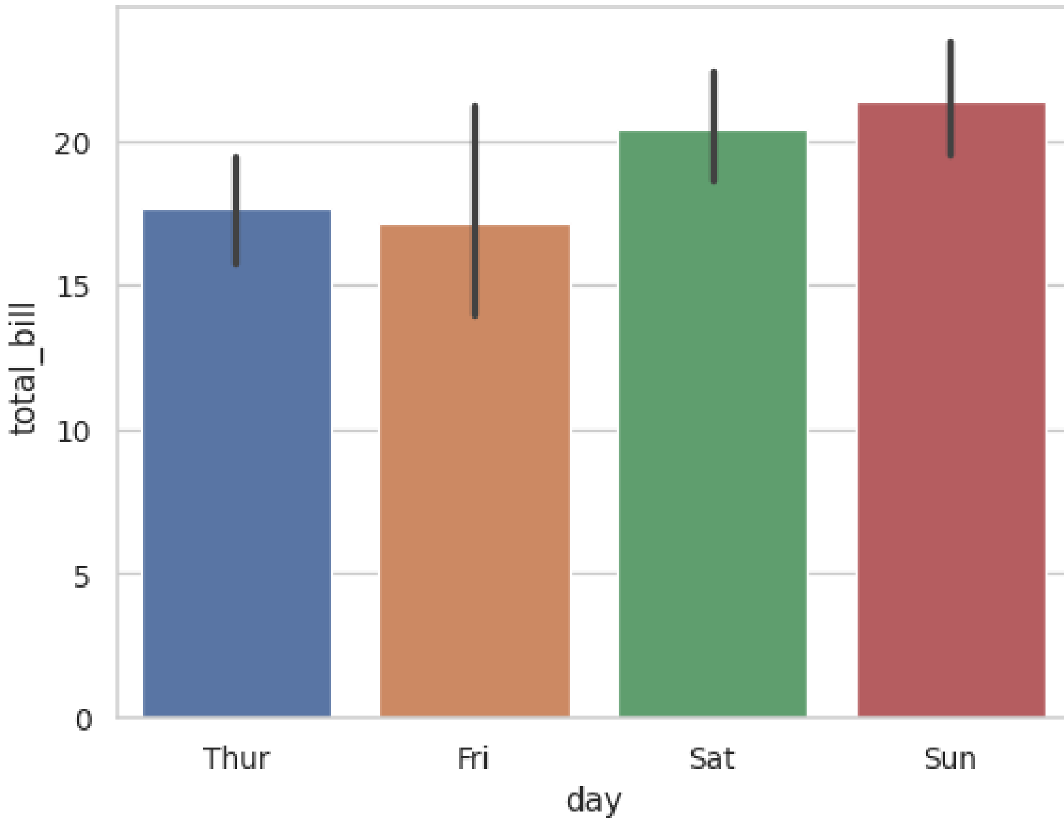
***`pointplot`*** Show point estimates and confidence intervals using scatterplot glyphs.

***`catplot`*** Combine a categorical plot with a *`FacetGrid`*.

### Examples

Draw a set of vertical bar plots grouped by a categorical variable:

```
>>> import seaborn as sns
>>> sns.set_theme(style="whitegrid")
>>> tips = sns.load_dataset("tips")
>>> ax = sns.barplot(x="day", y="total_bill", data=tips)
```




Draw a set of vertical bars with nested grouping by a two variables:

```
>>> ax = sns.barplot(x="day", y="total_bill", hue="sex", data=tips)
```

Draw a set of horizontal bars:

```
>>> ax = sns.barplot(x="tip", y="day", data=tips)
```

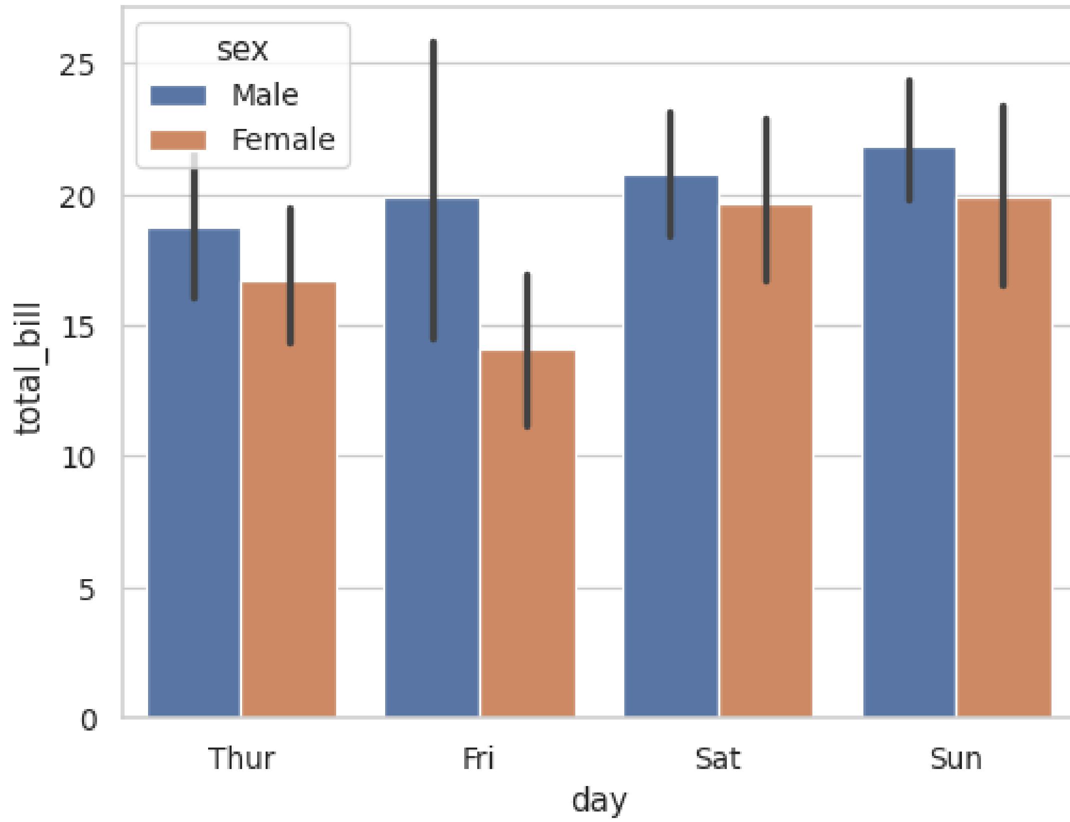
Control bar order by passing an explicit order:

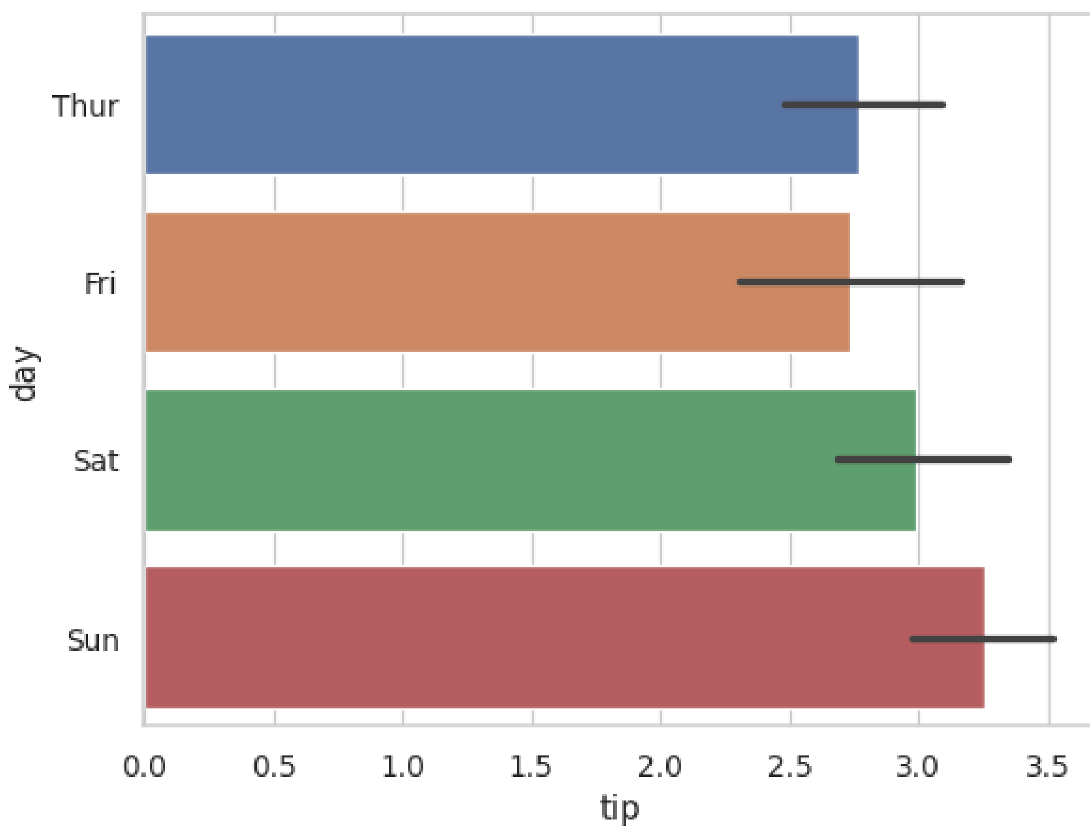
```
>>> ax = sns.barplot(x="time", y="tip", data=tips,
...                  order=["Dinner", "Lunch"])
```

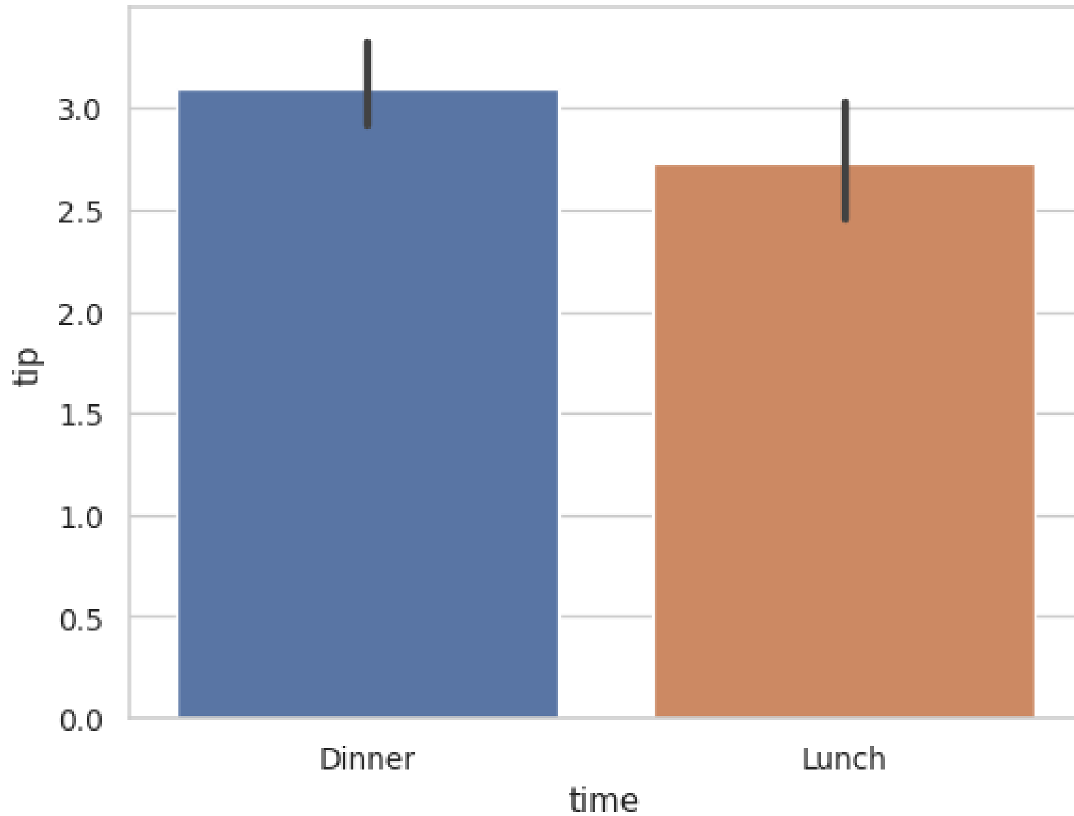
Use median as the estimate of central tendency:

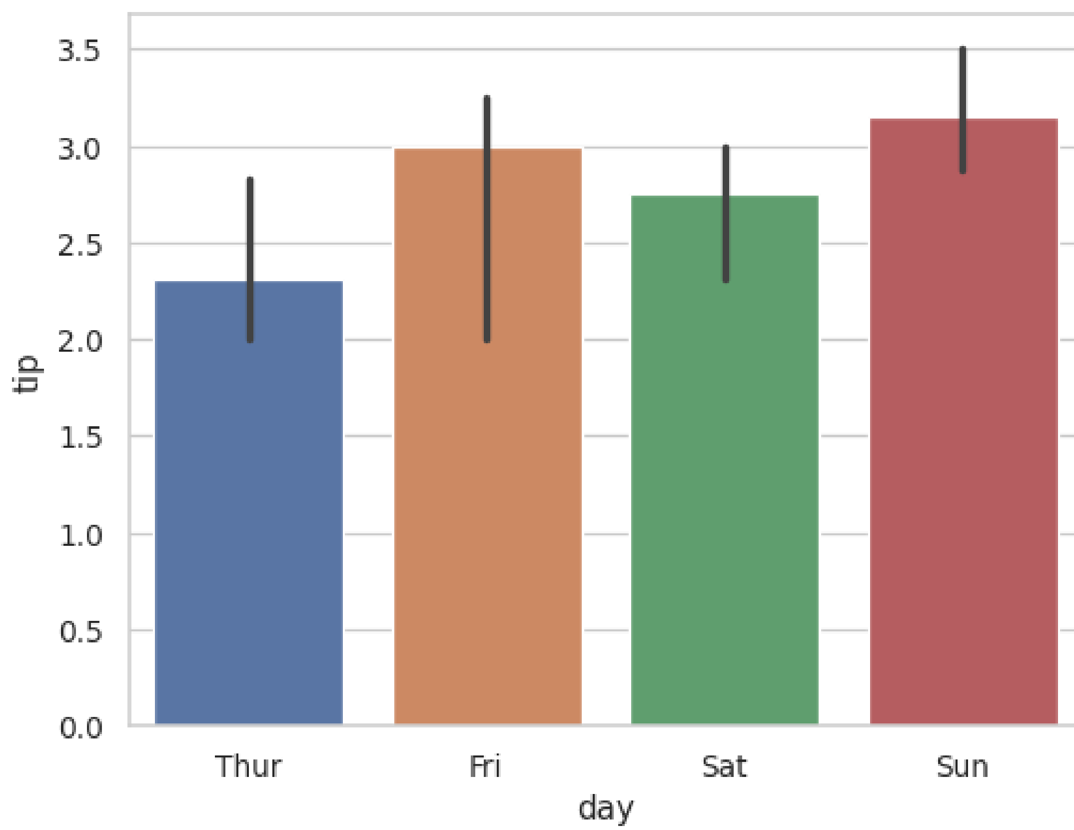
```
>>> from numpy import median
>>> ax = sns.barplot(x="day", y="tip", data=tips, estimator=median)
```

Show the standard error of the mean with the error bars:

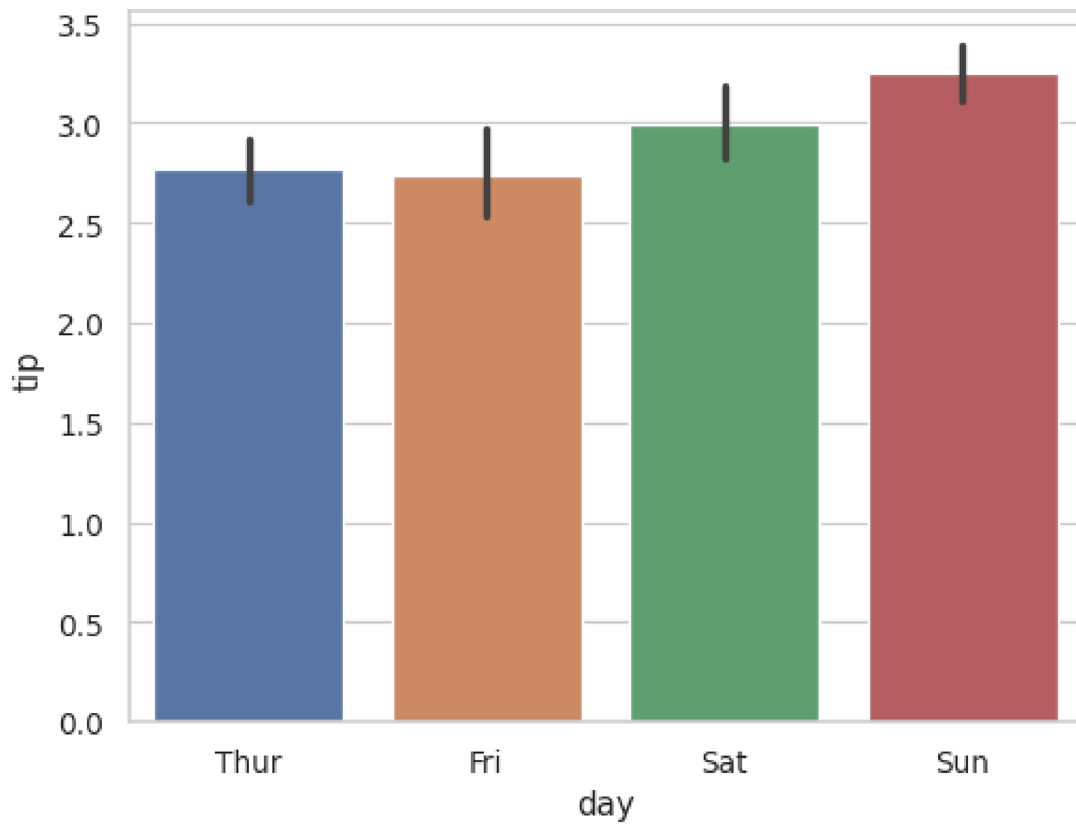

sex
Male
Female
0
5
10
15
20
25
total_bill
Thur
Fri
Sat
Sun
day


Thur
Fri
Sat
Sun
day
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
tip


3.0
2.5
2.0
1.5
1.0
0.5
0.0
tip
Dinner
Lunch
time


3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
tip
Thur
Fri
Sat
Sun
day

```
>>> ax = sns.barplot(x="day", y="tip", data=tips, ci=68)
```




Show standard deviation of observations instead of a confidence interval:

```
>>> ax = sns.barplot(x="day", y="tip", data=tips, ci="sd")
```

Add "caps" to the error bars:

```
>>> ax = sns.barplot(x="day", y="tip", data=tips, capsize=.2)
```

Use a different color palette for the bars:

```
>>> ax = sns.barplot(x="size", y="total_bill", data=tips,
...                  palette="Blues_d")
```

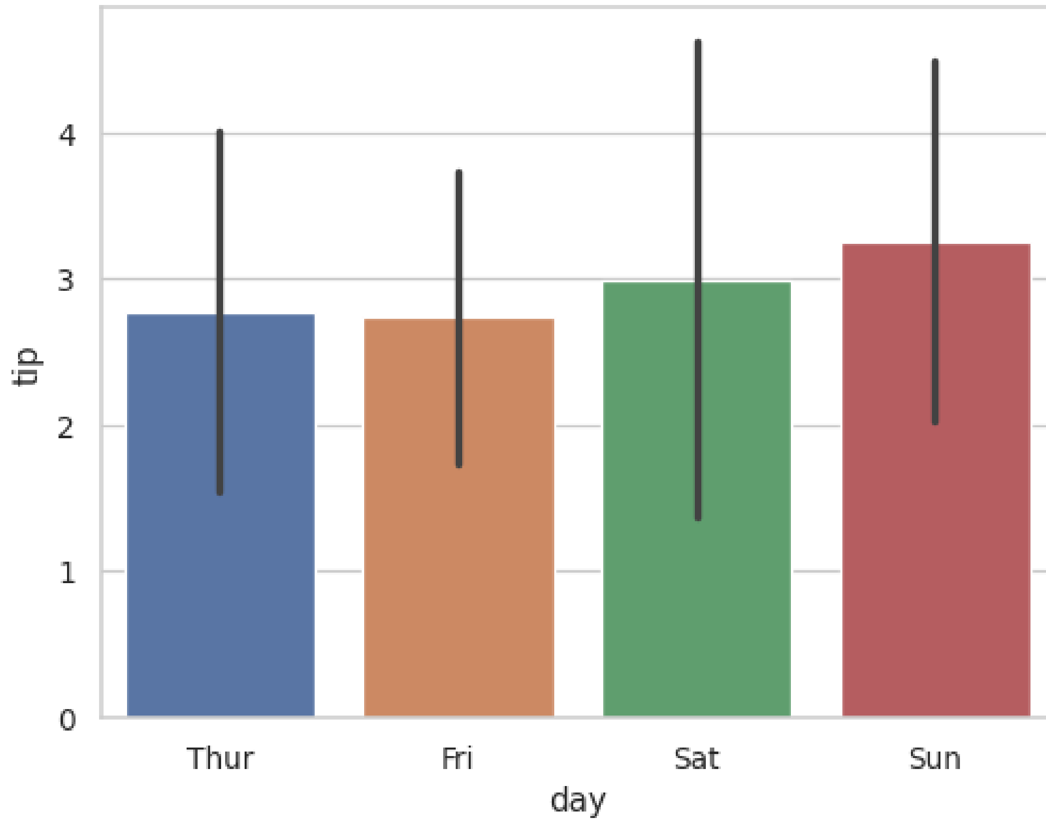
Use `hue` without changing bar position or width:

```
>>> tips["weekend"] = tips["day"].isin(["Sat", "Sun"])
>>> ax = sns.barplot(x="day", y="total_bill", hue="weekend",
...                  data=tips, dodge=False)
```

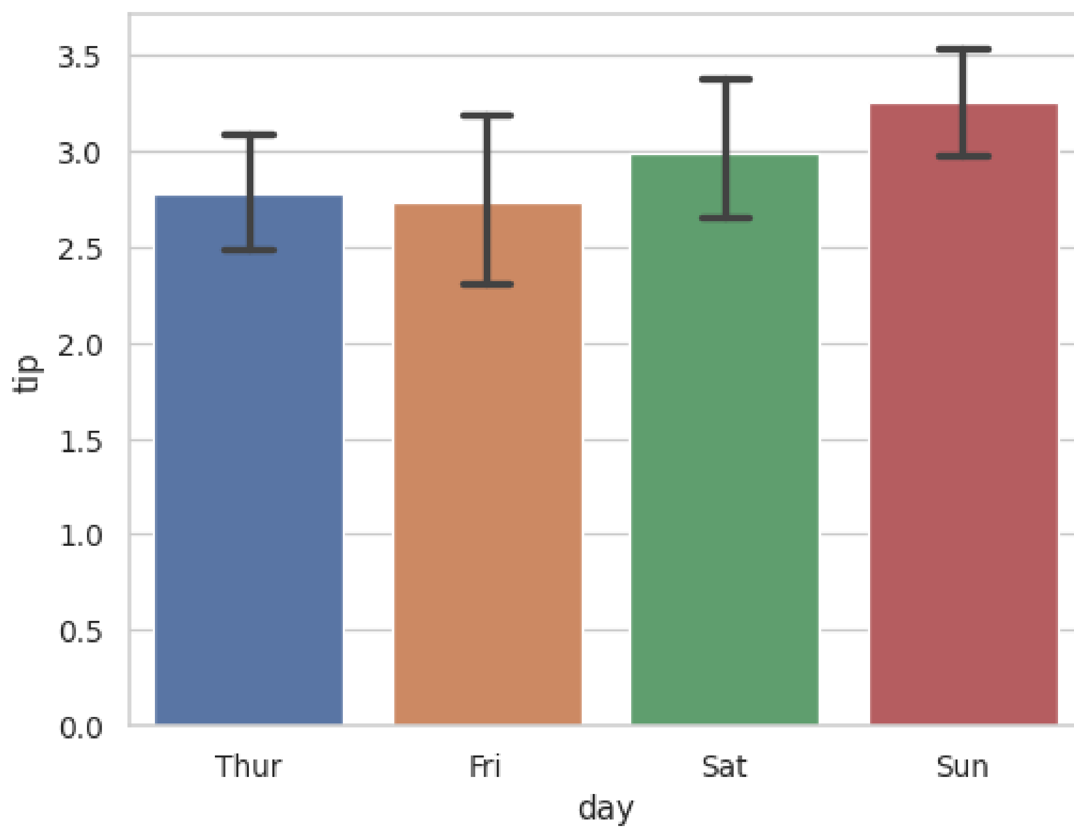
Plot all bars in a single color:

```
>>> ax = sns.barplot(x="size", y="total_bill", data=tips,
...                  color="salmon", saturation=.5)
```

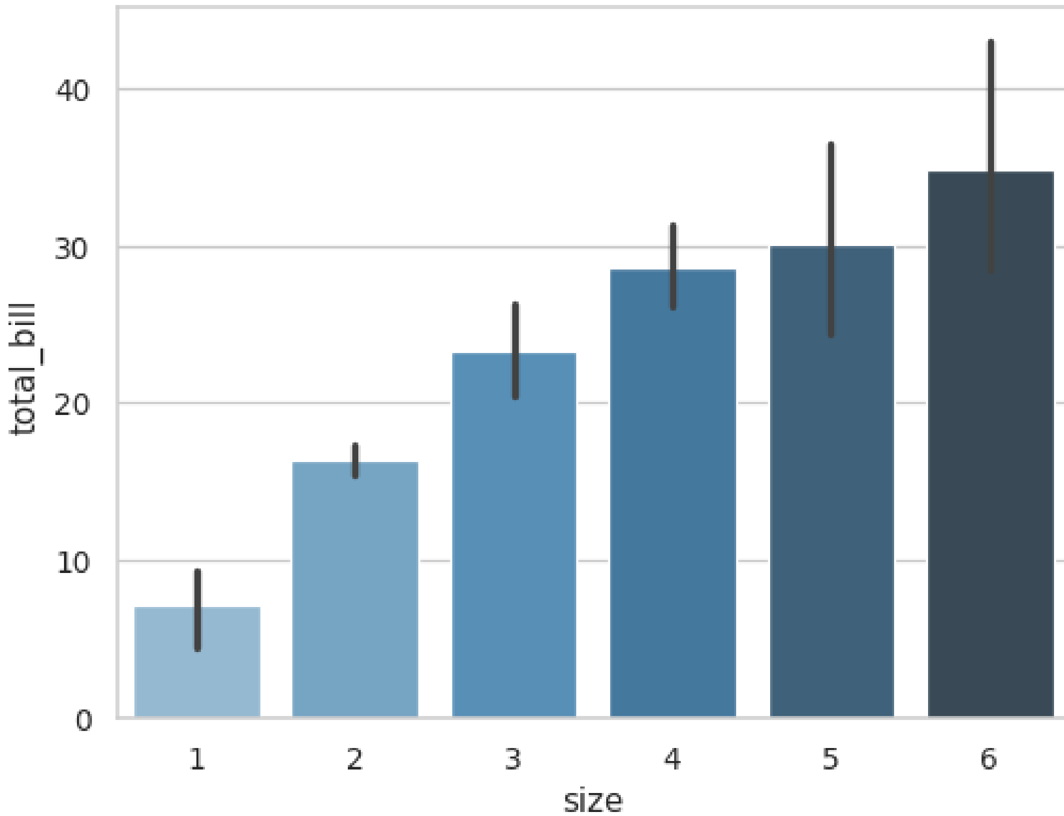
Use `matplotlib.axes.Axes.bar()` parameters to control the style.

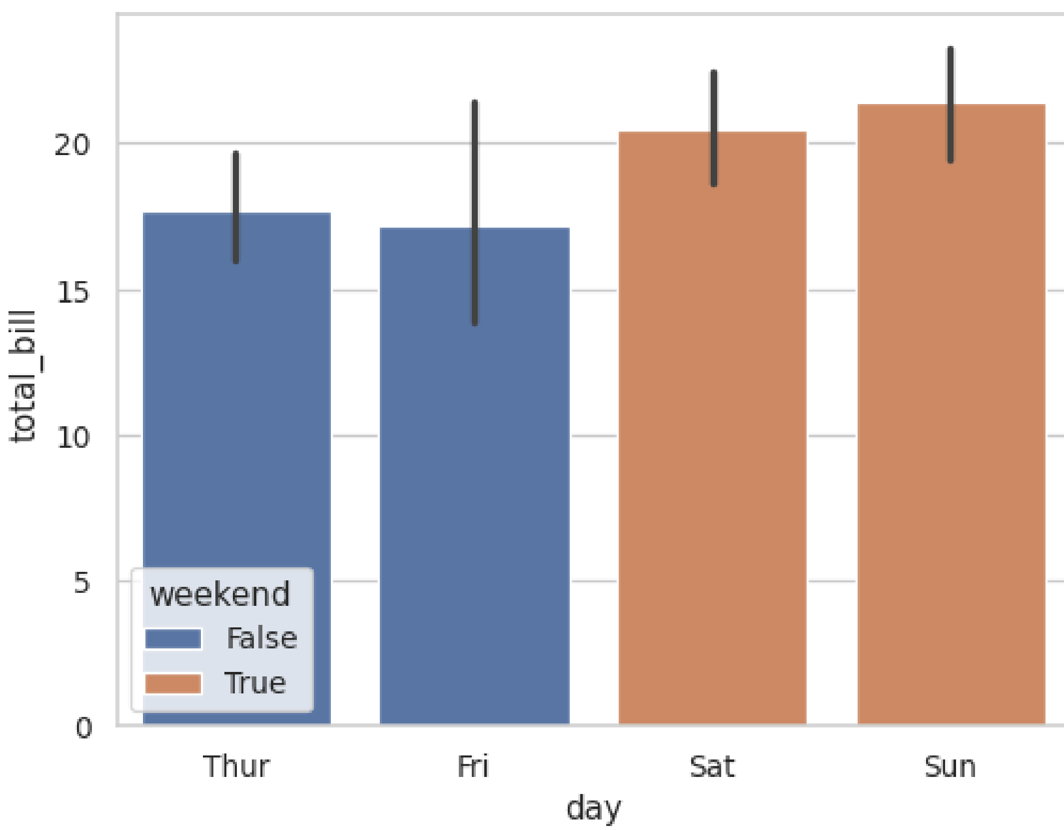


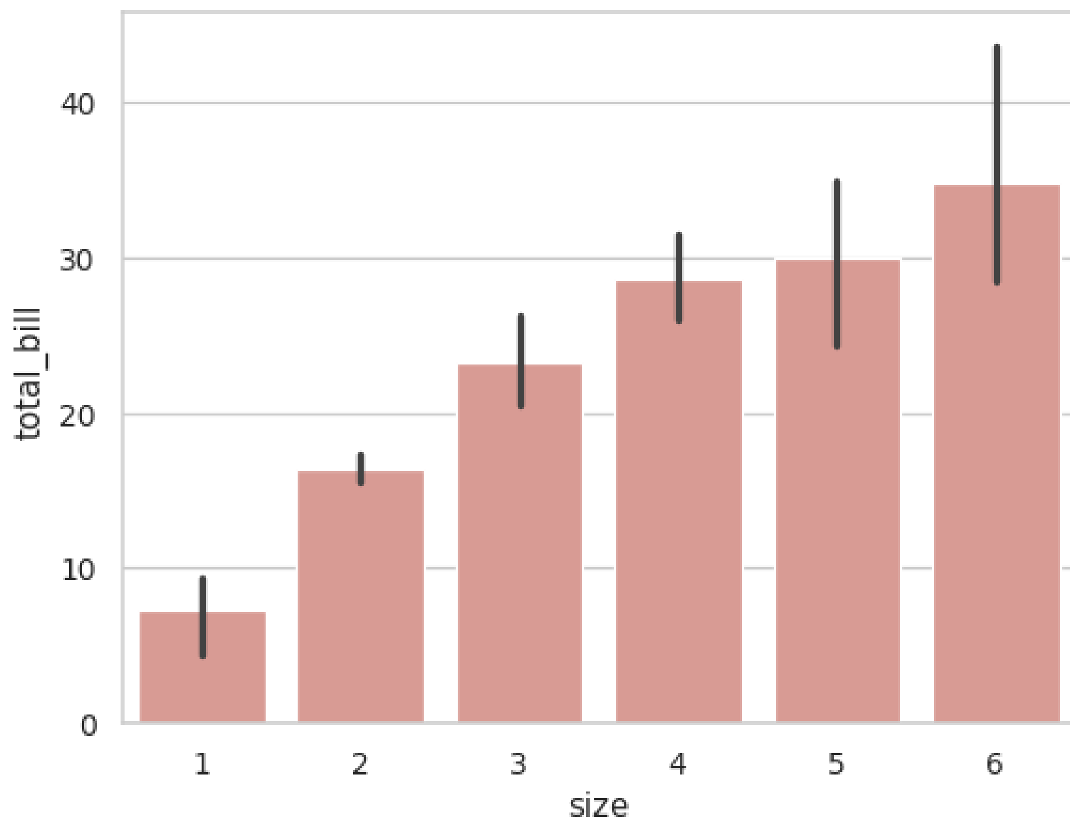
4
3
2
1
0
tip
Thur
Fri
Sat
Sun
day

3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
tip
Thur
Fri
Sat
Sun
day


total_bill
0
10
20
30
40
1
2
3
4
5
6
size


total_bill
0
5
10
15
20
weekend
False
True
Thur
Fri
Sat
Sun
day




```
>>> ax = sns.barplot(x="day", y="total_bill", data=tips,
...                  linewidth=2.5, facecolor=(1, 1, 1, 0),
...                  errcolor=".2", edgecolor=".2")
```

Use `catplot()` to combine a `barplot()` and a `FacetGrid`. This allows grouping within additional categorical variables. Using `catplot()` is safer than using `FacetGrid` directly, as it ensures synchronization of variable order across facets:

```
>>> g = sns.catplot(x="sex", y="total_bill",
...                 hue="smoker", col="time",
...                 data=tips, kind="bar",
...                 height=4, aspect=.7);
```

### 5.3.9 seaborn.countplot

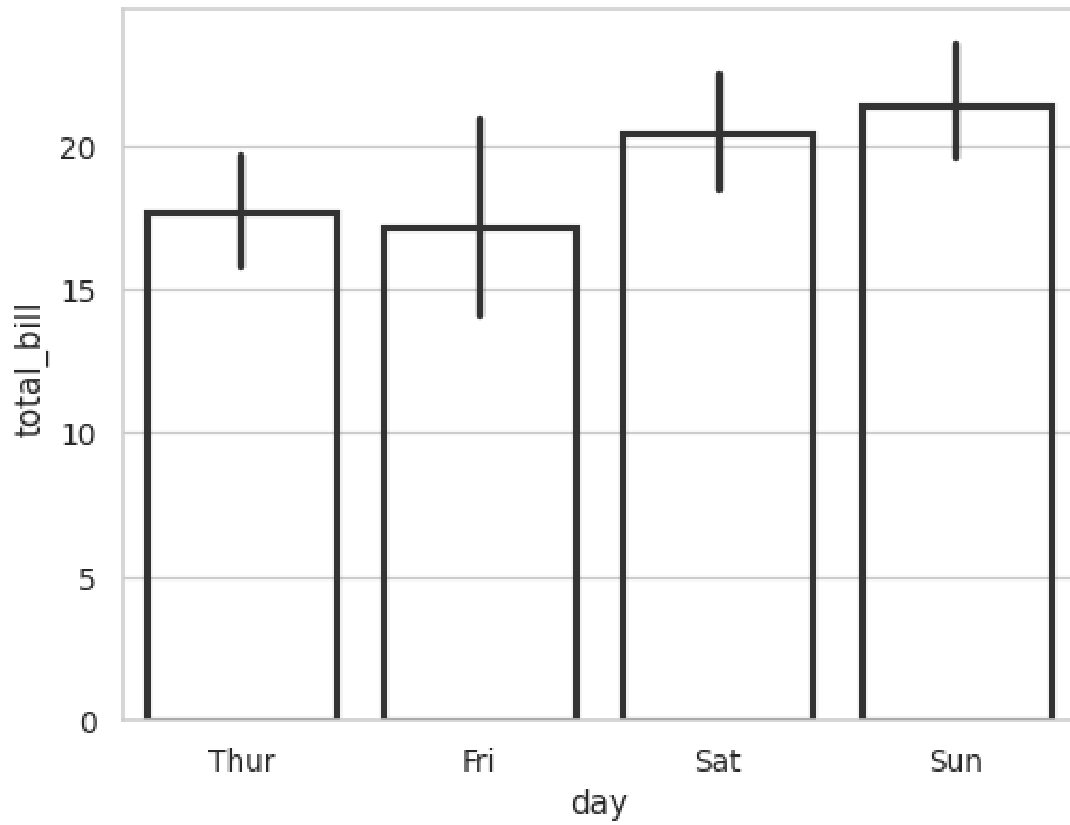
seaborn.**countplot**(*, *x=None*, *y=None*, *hue=None*, *data=None*, *order=None*, *hue_order=None*, *orient=None*, *color=None*, *palette=None*, *saturation=0.75*, *dodge=True*, *ax=None*, ***kwargs*)

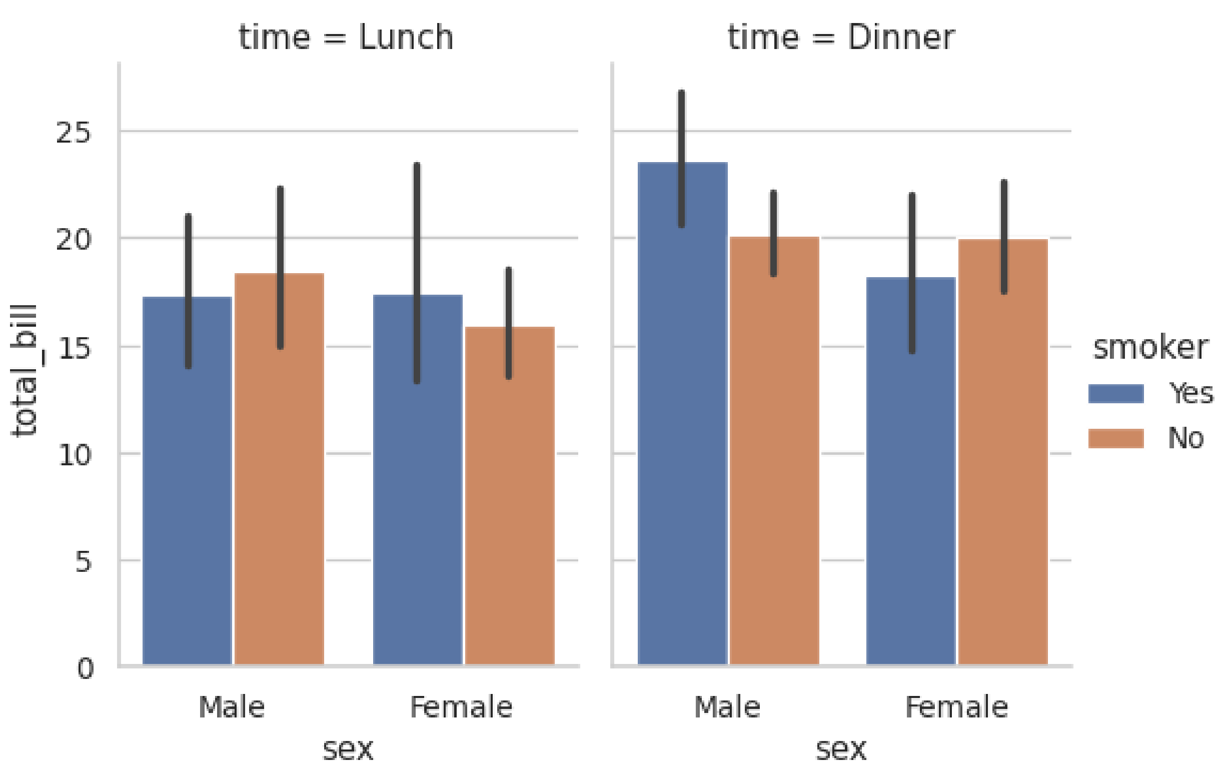
Show the counts of observations in each categorical bin using bars.

A count plot can be thought of as a histogram across a categorical, instead of quantitative, variable. The basic API and options are identical to those for `barplot()`, so you can compare counts across nested variables.

Input data can be passed in a variety of formats, including:

- Vectors of data represented as lists, numpy arrays, or pandas Series objects passed directly to the `x`, `y`,


20
15
10
5
0
total_bill
Thur
Fri
Sat
Sun
day


time = Lunch
time = Dinner
25
20
15
10
5
0
total_bill
Male
Female
sex
Male
Female
sex
smoker
Yes
No

and/or hue parameters.

- A "long-form" DataFrame, in which case the x, y, and hue variables will determine how the data are plotted.
- A "wide-form" DataFrame, such that each numeric column will be plotted.
- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas objects are preferable because the associated names will be used to annotate the axes. Additionally, you can use Categorical types for the grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and draws data at ordinal positions (0, 1, … n) on the relevant axis, even when the data has a numeric or date type.

See the tutorial for more information.

**Parameters**

**x, y, hue** [names of variables in data or vector data, optional] Inputs for plotting long-form data. See examples for interpretation.

**data** [DataFrame, array, or list of arrays, optional] Dataset for plotting. If x and y are absent, this is interpreted as wide-form. Otherwise it is expected to be long-form.

**order, hue_order** [lists of strings, optional] Order to plot the categorical levels in, otherwise the levels are inferred from the data objects.

**orient** ["v" | "h", optional] Orientation of the plot (vertical or horizontal). This is usually inferred based on the type of the input variables, but it can be used to resolve ambiguitiy when both x and y are numeric or when plotting wide-form data.

**color** [matplotlib color, optional] Color for all of the elements, or seed for a gradient palette.

**palette** [palette name, list, or dict] Colors to use for the different levels of the hue variable. Should be something that can be interpreted by *color_palette()*, or a dictionary mapping hue levels to matplotlib colors.

**saturation** [float, optional] Proportion of the original saturation to draw colors at. Large patches often look better with slightly desaturated colors, but set this to 1 if you want the plot colors to perfectly match the input color spec.

**dodge** [bool, optional] When hue nesting is used, whether elements should be shifted along the categorical axis.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**kwargs** [key, value mappings] Other keyword arguments are passed through to matplotlib.axes.Axes.bar().

**Returns**

**ax** [matplotlib Axes] Returns the Axes object with the plot drawn onto it.

**See also:**

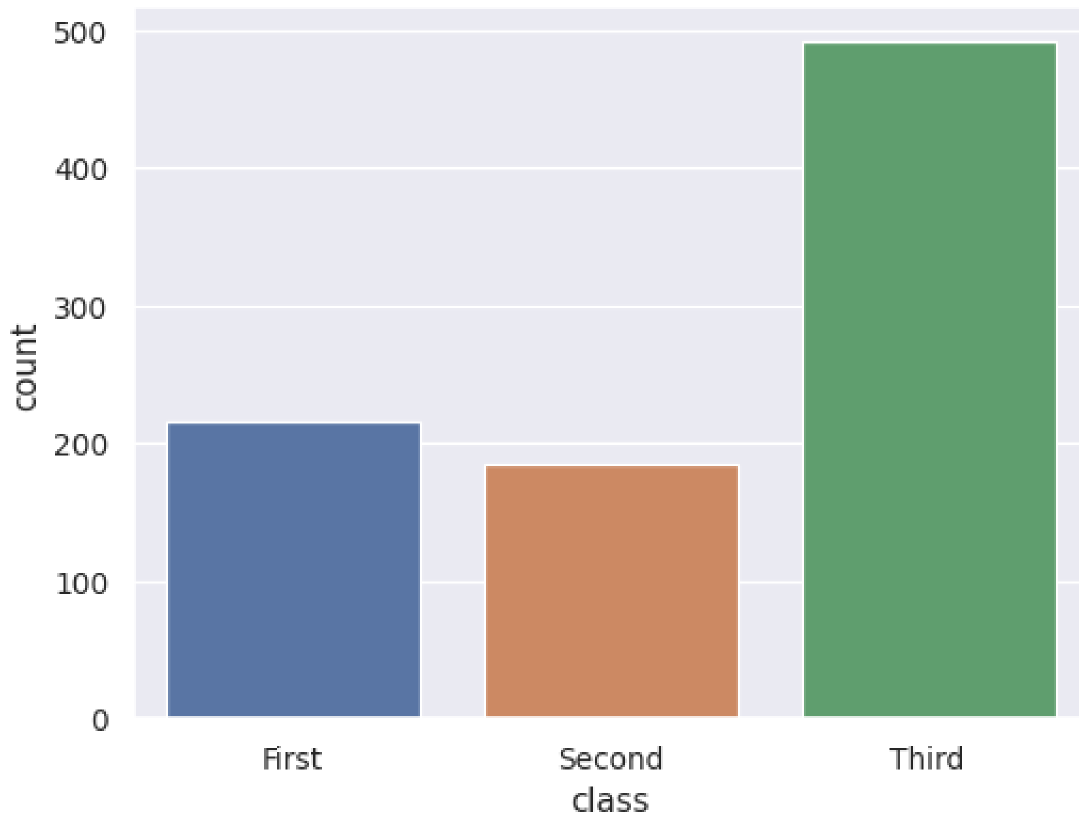
***barplot*** Show point estimates and confidence intervals using bars.

***catplot*** Combine a categorical plot with a *FacetGrid*.

### Examples

Show value counts for a single categorical variable:

```
>>> import seaborn as sns
>>> sns.set_theme(style="darkgrid")
>>> titanic = sns.load_dataset("titanic")
>>> ax = sns.countplot(x="class", data=titanic)
```




Show value counts for two categorical variables:

```
>>> ax = sns.countplot(x="class", hue="who", data=titanic)
```

Plot the bars horizontally:

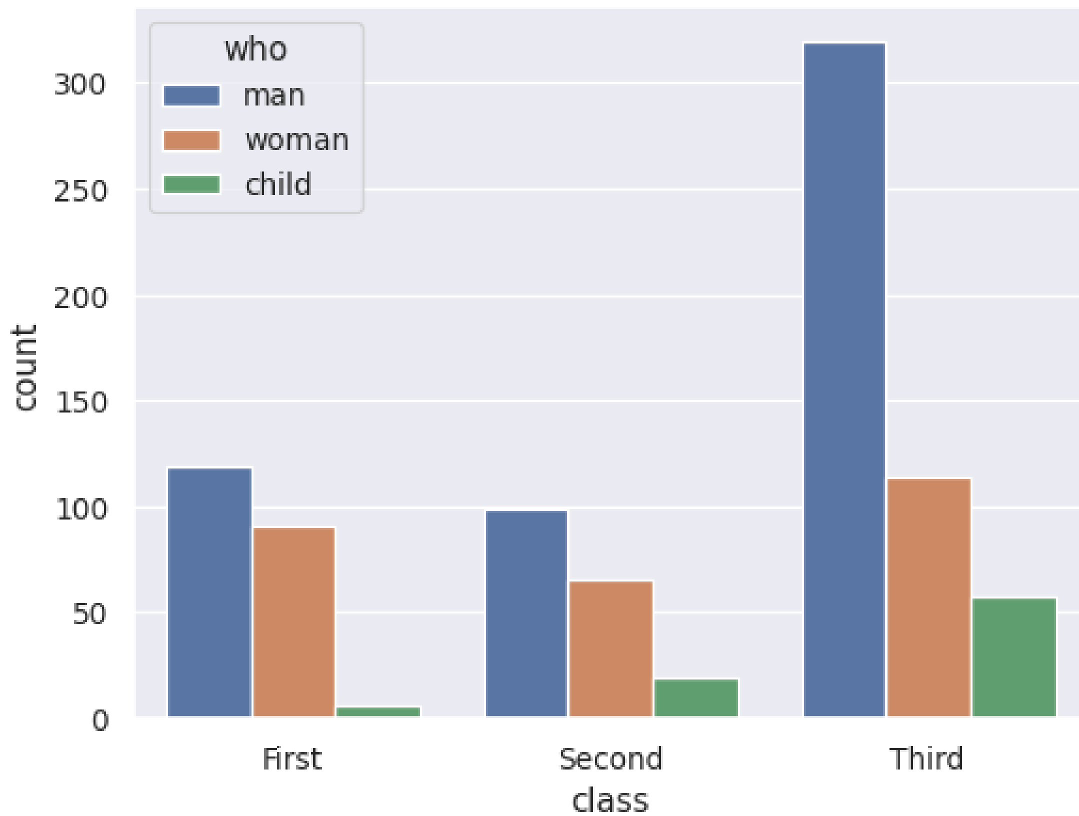
```
>>> ax = sns.countplot(y="class", hue="who", data=titanic)
```

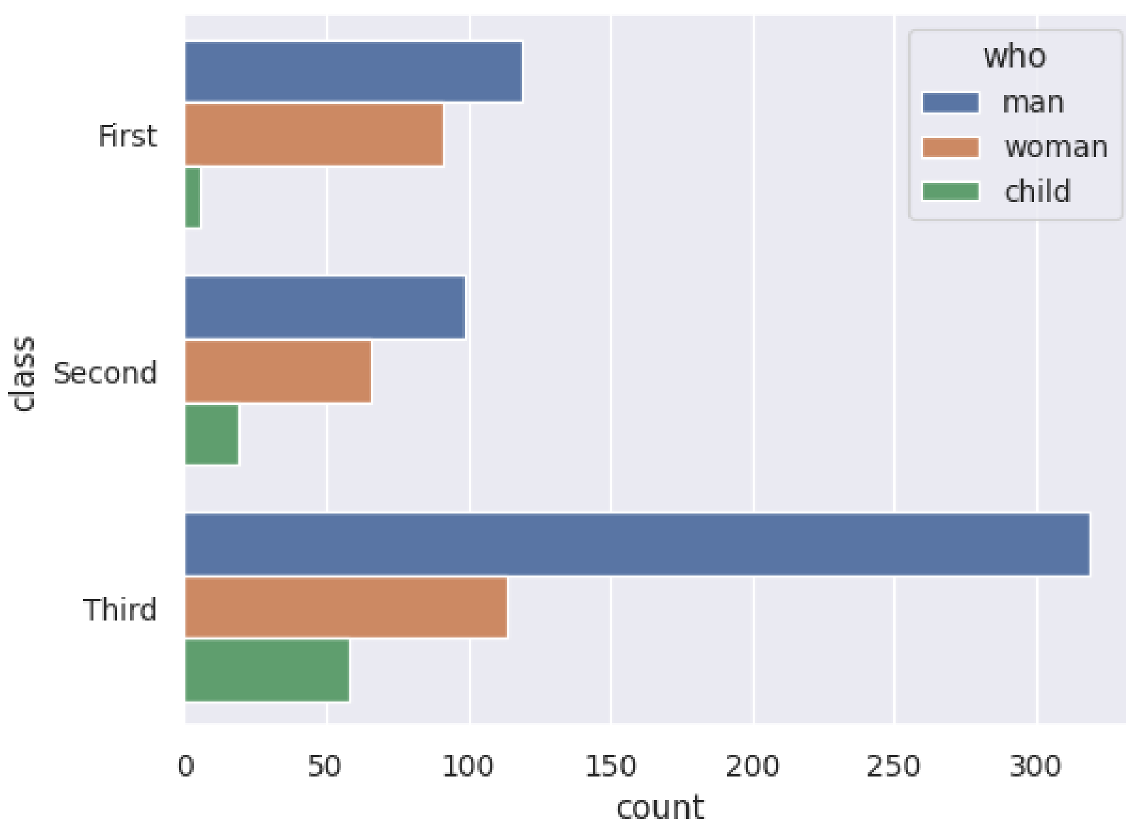
Use a different color palette:

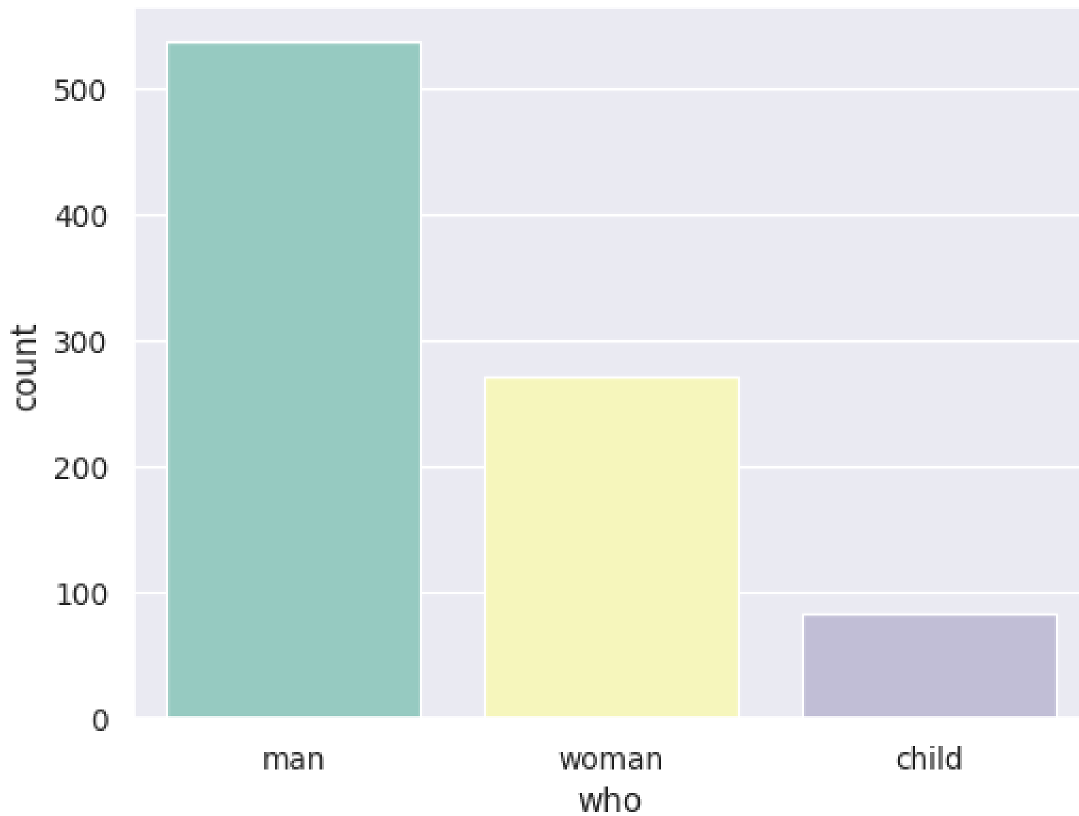
```
>>> ax = sns.countplot(x="who", data=titanic, palette="Set3")
```

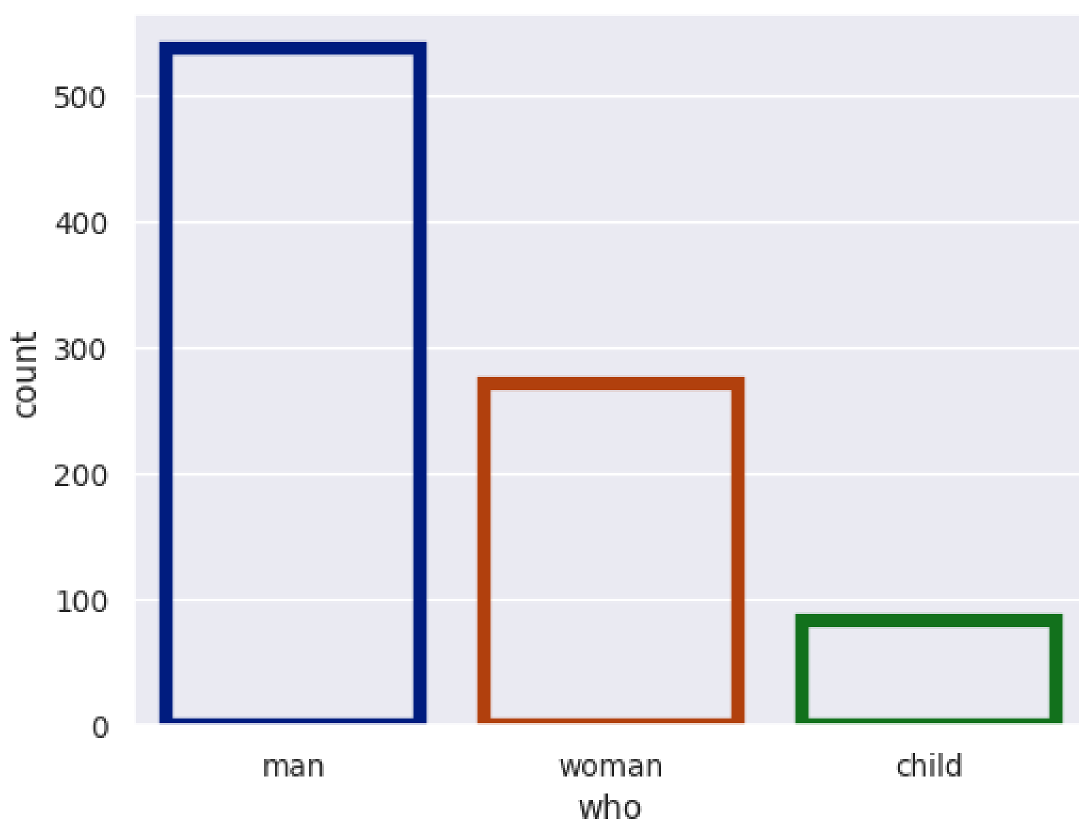
Use `matplotlib.axes.Axes.bar()` parameters to control the style.

```
>>> ax = sns.countplot(x="who", data=titanic,
...                    facecolor=(0, 0, 0, 0),
...                    linewidth=5,
...                    edgecolor=sns.color_palette("dark", 3))
```


who
man
woman
child
count
0
50
100
150
200
250
300
First
Second
Third
class

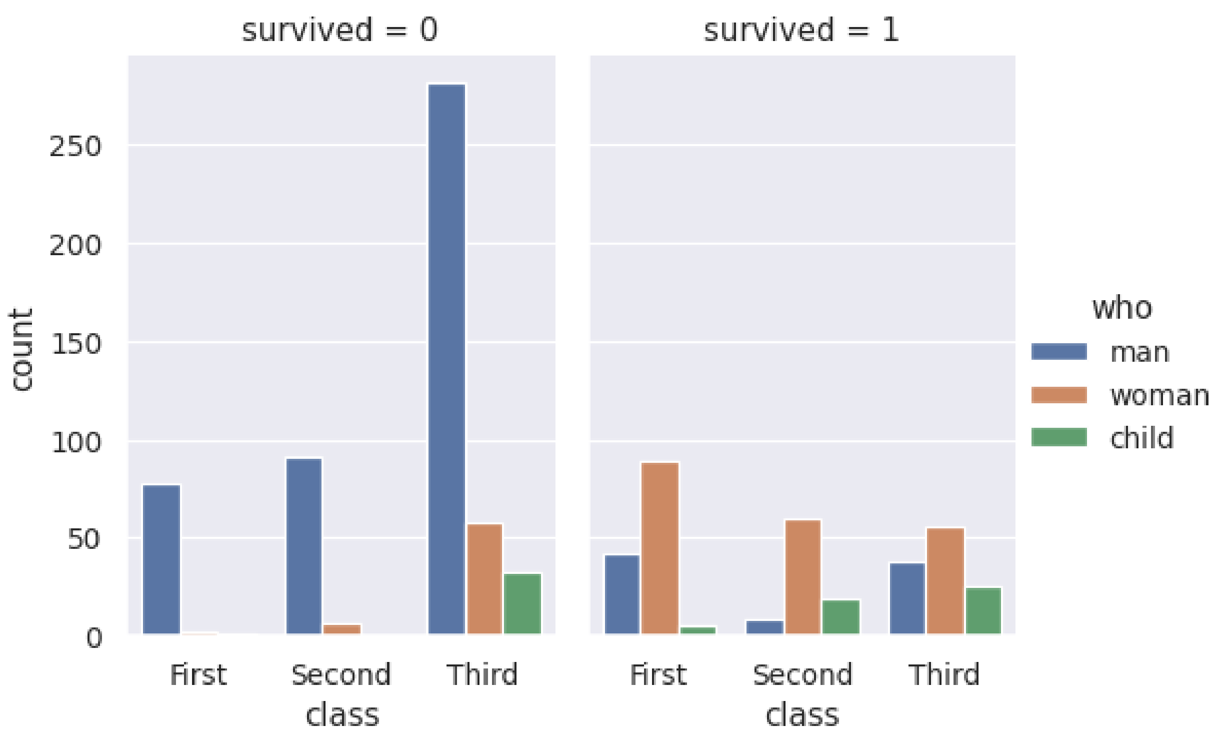

who
man
woman
child
class
First
Second
Third
0
50
100
150
200
250
300
count


count
500
400
300
200
100
0
man
woman
child
who


count
500
400
300
200
100
0
man
woman
child
who

Use *catplot()* to combine a *countplot()* and a *FacetGrid*. This allows grouping within additional categorical variables. Using *catplot()* is safer than using *FacetGrid* directly, as it ensures synchronization of variable order across facets:

```
>>> g = sns.catplot(x="class", hue="who", col="survived",
...                 data=titanic, kind="count",
...                 height=4, aspect=.7);
```




## 5.4 Regression plots

| | |
|---|---|
| *lmplot* | Plot data and regression model fits across a FacetGrid. |
| *regplot* | Plot data and a linear regression model fit. |
| *residplot* | Plot the residuals of a linear regression. |

### 5.4.1 seaborn.lmplot

seaborn.**lmplot**(**, x=None, y=None, data=None, hue=None, col=None, row=None, palette=None, col_wrap=None, height=5, aspect=1, markers='o', sharex=True, sharey=True, hue_order=None, col_order=None, row_order=None, legend=True, legend_out=True, x_estimator=None, x_bins=None, x_ci='ci', scatter=True, fit_reg=True, ci=95, n_boot=1000, units=None, seed=None, order=1, logistic=False, lowess=False, robust=False, logx=False, x_partial=None, y_partial=None, truncate=True, x_jitter=None, y_jitter=None, scatter_kws=None, line_kws=None, size=None*)

Plot data and regression model fits across a FacetGrid.

This function combines *regplot()* and *FacetGrid*. It is intended as a convenient interface to fit regression models across conditional subsets of a dataset.

When thinking about how to assign variables to different facets, a general rule is that it makes sense to use `hue` for the most important comparison, followed by `col` and `row`. However, always think about your particular dataset and the goals of the visualization you are creating.

There are a number of mutually exclusive options for estimating the regression model. See the tutorial for more information.

The parameters to this function span most of the options in *FacetGrid*, although there may be occasional cases where you will want to use that class and *regplot()* directly.

**Parameters**

**x, y** [strings, optional] Input variables; these should be column names in `data`.

**data** [DataFrame] Tidy ("long-form") dataframe where each column is a variable and each row is an observation.

**hue, col, row** [strings] Variables that define subsets of the data, which will be drawn on separate facets in the grid. See the `*_order` parameters to control the order of levels of this variable.

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by *color_palette()*, or a dictionary mapping hue levels to matplotlib colors.

**col_wrap** [int] "Wrap" the column variable at this width, so that the column facets span multiple rows. Incompatible with a `row` facet.

**height** [scalar] Height (in inches) of each facet. See also: `aspect`.

**aspect** [scalar] Aspect ratio of each facet, so that `aspect * height` gives the width of each facet in inches.

**markers** [matplotlib marker code or list of marker codes, optional] Markers for the scatterplot. If a list, each marker in the list will be used for each level of the `hue` variable.

**share{x,y}** [bool, 'col', or 'row' optional] If true, the facets will share y axes across columns and/or x axes across rows.

**{hue,col,row}_order** [lists, optional] Order for the levels of the faceting variables. By default, this will be the order that the levels appear in `data` or, if the variables are pandas categoricals, the category order.

**legend** [bool, optional] If `True` and there is a `hue` variable, add a legend.

**legend_out** [bool] If `True`, the figure size will be extended, and the legend will be drawn outside the plot on the center right.

**x_estimator** [callable that maps vector -> scalar, optional] Apply this function to each unique value of `x` and plot the resulting estimate. This is useful when `x` is a discrete variable. If `x_ci` is given, this estimate will be bootstrapped and a confidence interval will be drawn.

**x_bins** [int or vector, optional] Bin the `x` variable into discrete bins and then estimate the central tendency and a confidence interval. This binning only influences how the scatterplot is drawn; the regression is still fit to the original data. This parameter is interpreted either as the number of evenly-sized (not necessary spaced) bins or the positions of the bin centers. When this parameter is used, it implies that the default of `x_estimator` is `numpy.mean`.

**x_ci** ["ci", "sd", int in [0, 100] or None, optional] Size of the confidence interval used when plotting a central tendency for discrete values of `x`. If `"ci"`, defer to the value of the `ci` parameter. If `"sd"`, skip bootstrapping and show the standard deviation of the observations in each bin.

**scatter** [bool, optional] If True, draw a scatterplot with the underlying observations (or the x_estimator values).

**fit_reg** [bool, optional] If True, estimate and plot a regression model relating the x and y variables.

**ci** [int in [0, 100] or None, optional] Size of the confidence interval for the regression estimate. This will be drawn using translucent bands around the regression line. The confidence interval is estimated using a bootstrap; for large datasets, it may be advisable to avoid that computation by setting this parameter to None.

**n_boot** [int, optional] Number of bootstrap resamples used to estimate the ci. The default value attempts to balance time and stability; you may want to increase this value for "final" versions of plots.

**units** [variable name in data, optional] If the x and y observations are nested within sampling units, those can be specified here. This will be taken into account when computing the confidence intervals by performing a multilevel bootstrap that resamples both units and observations (within unit). This does not otherwise influence how the regression is estimated or drawn.

**seed** [int, numpy.random.Generator, or numpy.random.RandomState, optional] Seed or random number generator for reproducible bootstrapping.

**order** [int, optional] If order is greater than 1, use numpy.polyfit to estimate a polynomial regression.

**logistic** [bool, optional] If True, assume that y is a binary variable and use statsmodels to estimate a logistic regression model. Note that this is substantially more computationally intensive than linear regression, so you may wish to decrease the number of bootstrap resamples (n_boot) or set ci to None.

**lowess** [bool, optional] If True, use statsmodels to estimate a nonparametric lowess model (locally weighted linear regression). Note that confidence intervals cannot currently be drawn for this kind of model.

**robust** [bool, optional] If True, use statsmodels to estimate a robust regression. This will de-weight outliers. Note that this is substantially more computationally intensive than standard linear regression, so you may wish to decrease the number of bootstrap resamples (n_boot) or set ci to None.

**logx** [bool, optional] If True, estimate a linear regression of the form y ~ log(x), but plot the scatterplot and regression model in the input space. Note that x must be positive for this to work.

**{x,y}_partial** [strings in data or matrices] Confounding variables to regress out of the x or y variables before plotting.

**truncate** [bool, optional] If True, the regression line is bounded by the data limits. If False, it extends to the x axis limits.

**{x,y}_jitter** [floats, optional] Add uniform random noise of this size to either the x or y variables. The noise is added to a copy of the data after fitting the regression, and only influences the look of the scatterplot. This can be helpful when plotting variables that take discrete values.

**{scatter,line}_kws** [dictionaries] Additional keyword arguments to pass to plt.scatter and plt.plot.

**See also:**

**regplot** Plot data and a conditional model fit.

**FacetGrid** Subplot grid for plotting conditional relationships.

**pairplot** Combine *regplot()* and *PairGrid* (when used with kind="reg").

### Notes

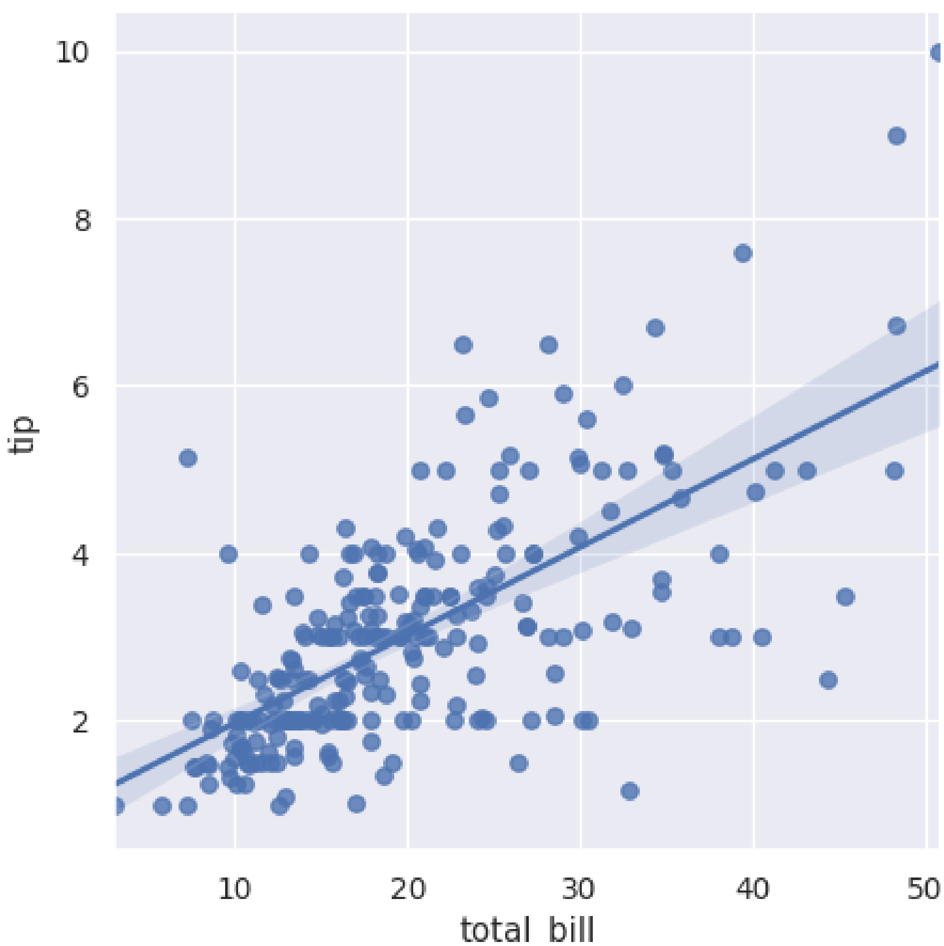
The *regplot()* and *lmplot()* functions are closely related, but the former is an axes-level function while the latter is a figure-level function that combines *regplot()* and *FacetGrid*.

### Examples

These examples focus on basic regression model plots to exhibit the various faceting options; see the *regplot()* docs for demonstrations of the other options for plotting the data and models. There are also other examples for how to manipulate plot using the returned object on the *FacetGrid* docs.

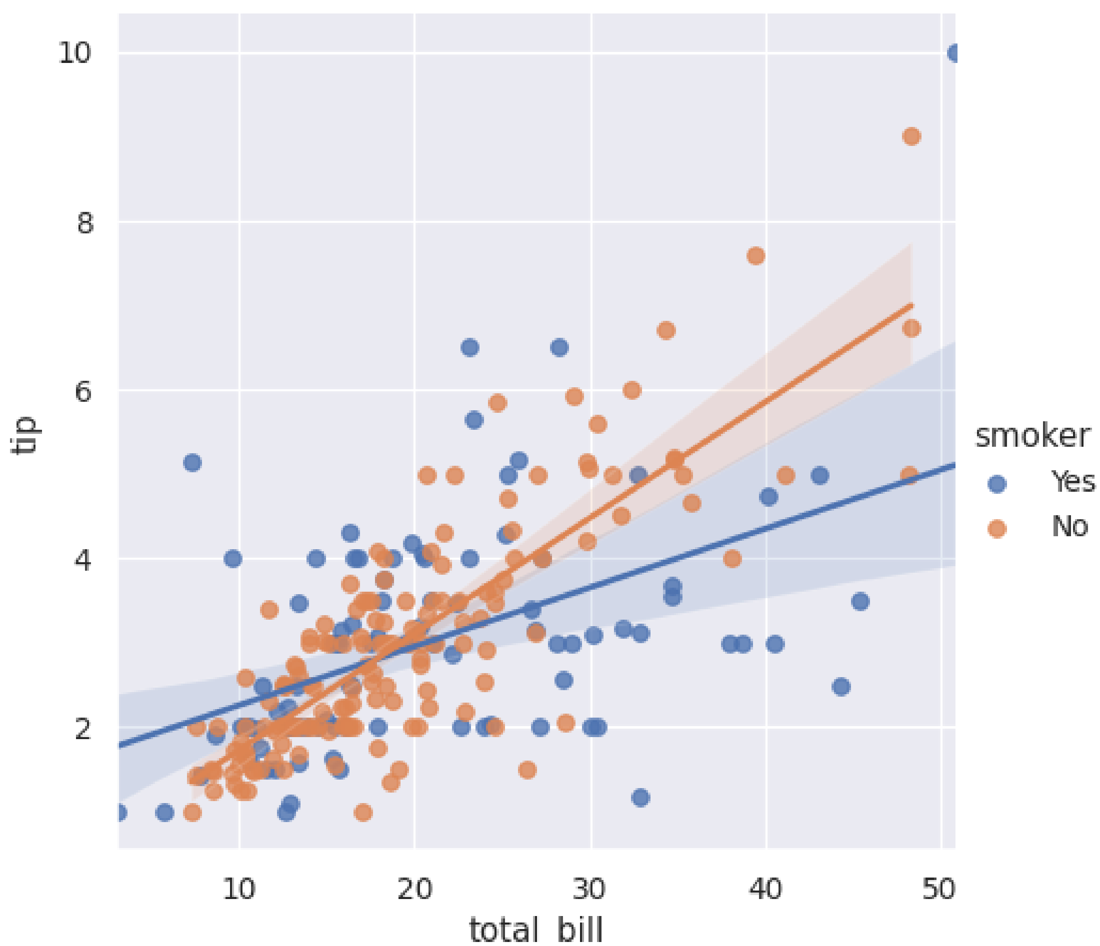
Plot a simple linear relationship between two variables:

```
>>> import seaborn as sns; sns.set_theme(color_codes=True)
>>> tips = sns.load_dataset("tips")
>>> g = sns.lmplot(x="total_bill", y="tip", data=tips)
```



Condition on a third variable and plot the levels in different colors:

```
>>> g = sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips)
```




Use different markers as well as colors so the plot will reproduce to black-and-white more easily:

```
>>> g = sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips,
...                markers=["o", "x"])
```

Use a different color palette:

```
>>> g = sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips,
...                palette="Set1")
```

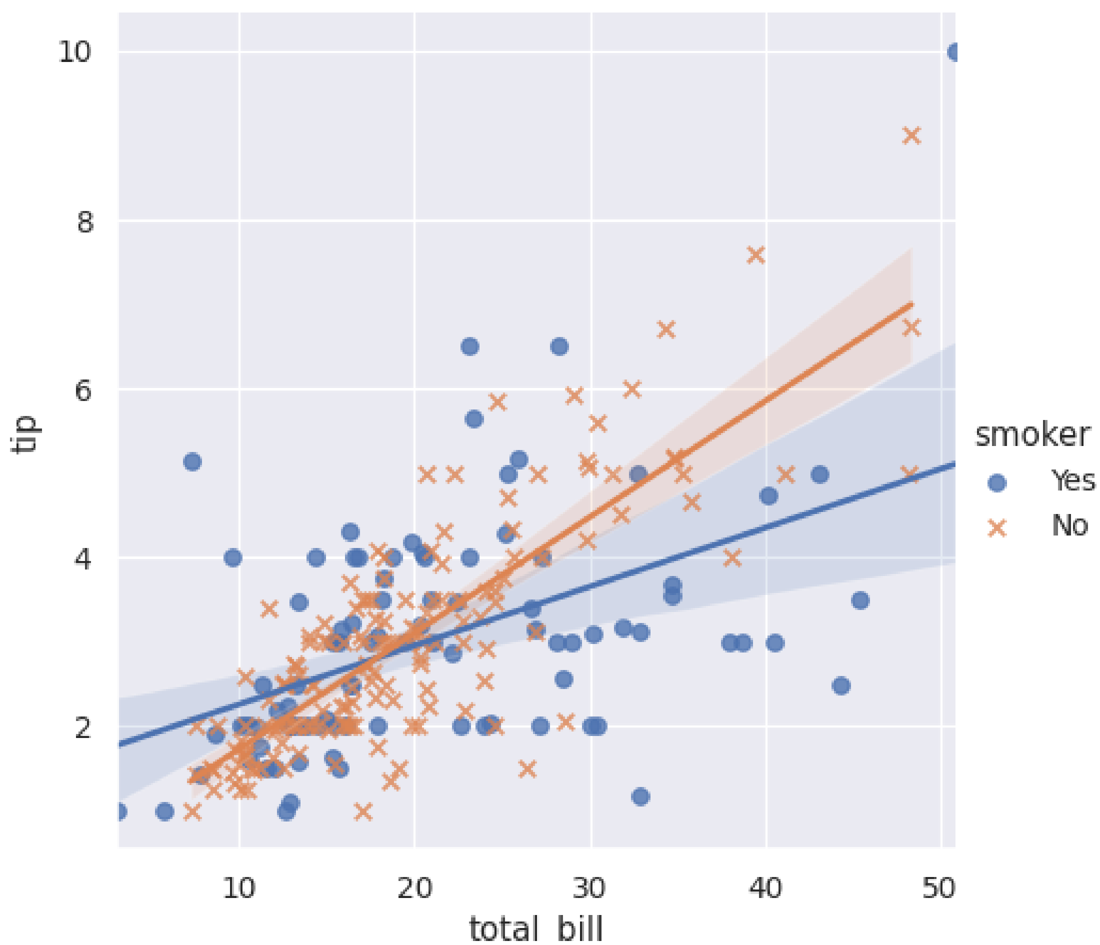
Map `hue` levels to colors with a dictionary:

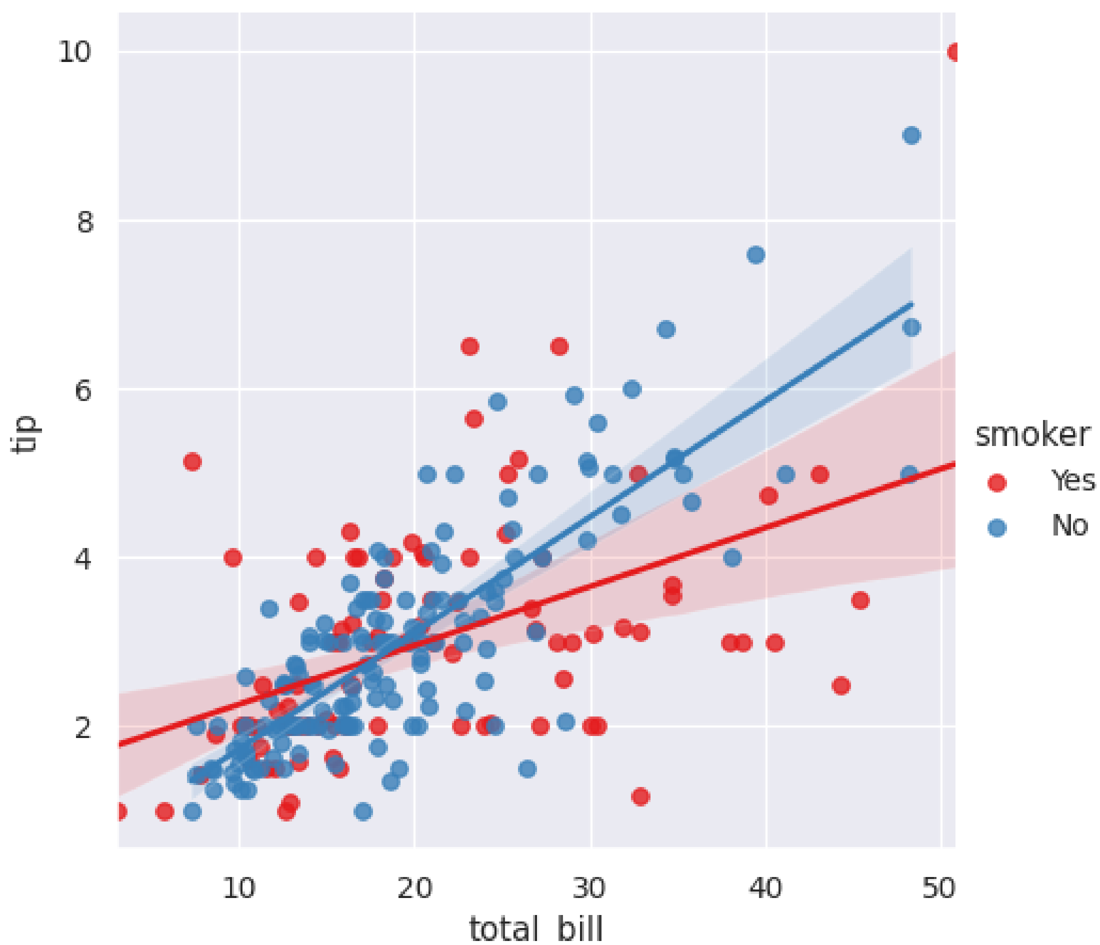
```
>>> g = sns.lmplot(x="total_bill", y="tip", hue="smoker", data=tips,
...                palette=dict(Yes="g", No="m"))
```

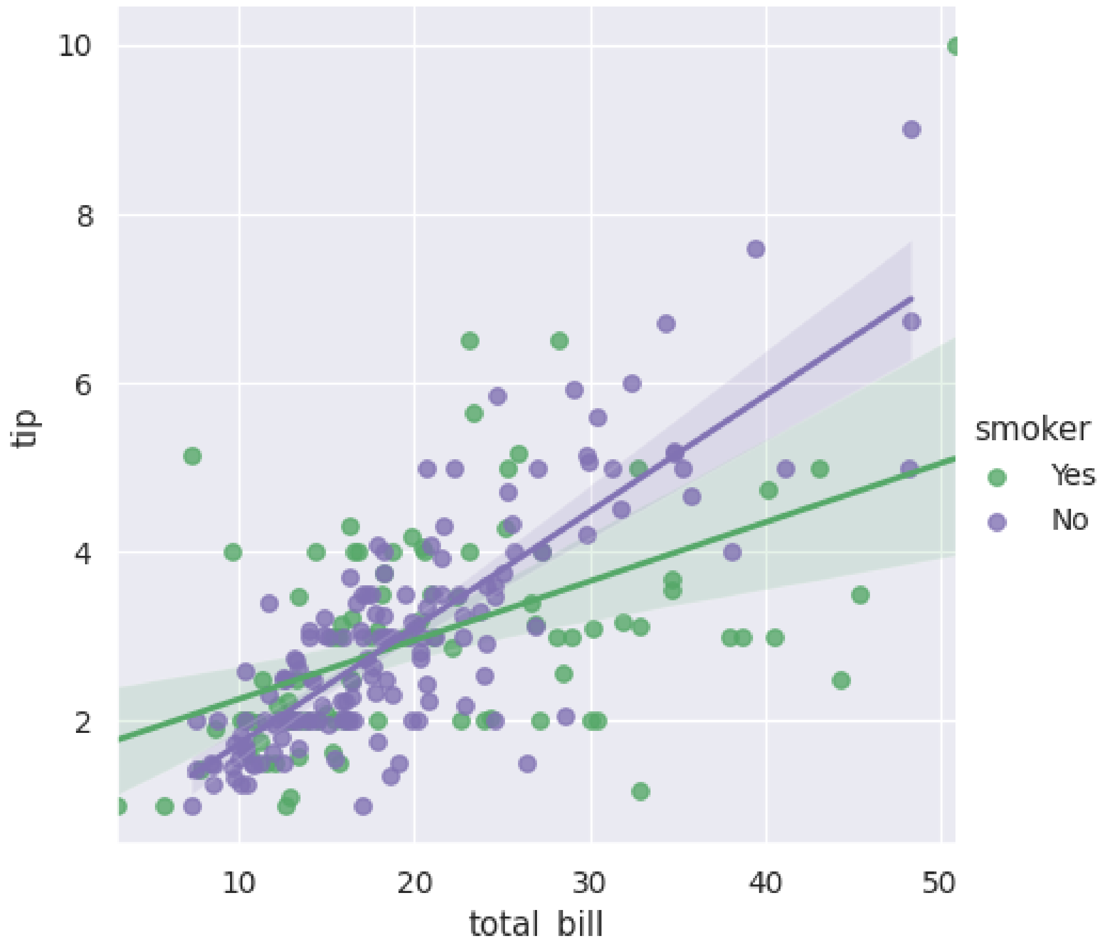
Plot the levels of the third variable across different columns:

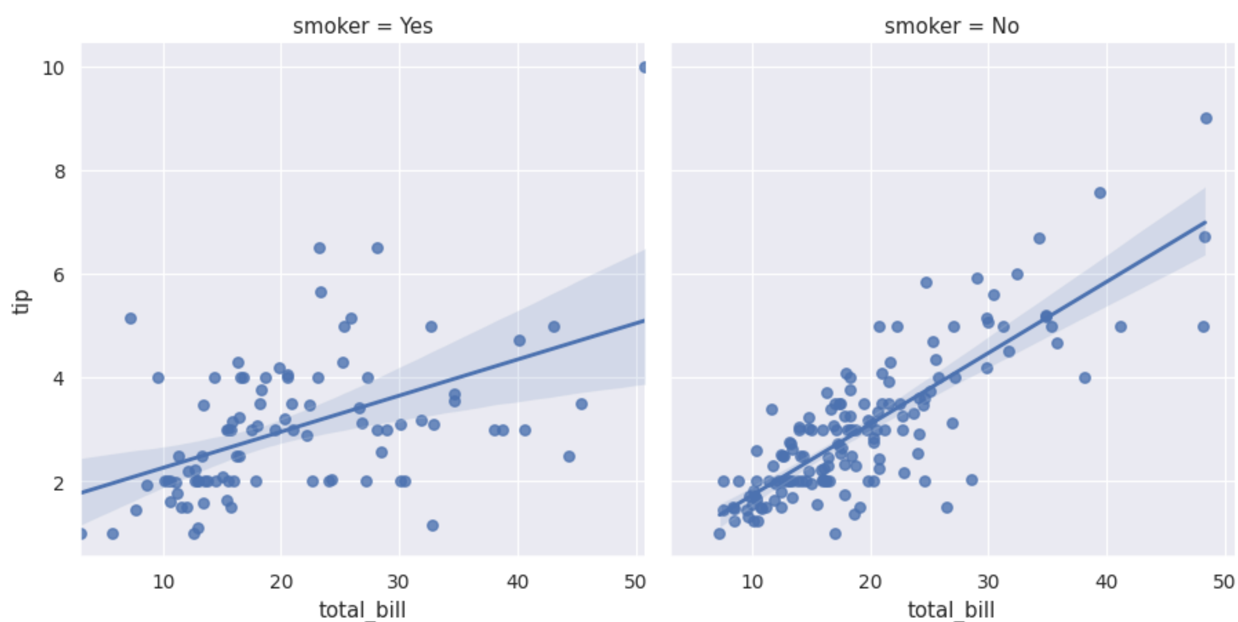
```
>>> g = sns.lmplot(x="total_bill", y="tip", col="smoker", data=tips)
```

Change the height and aspect ratio of the facets:

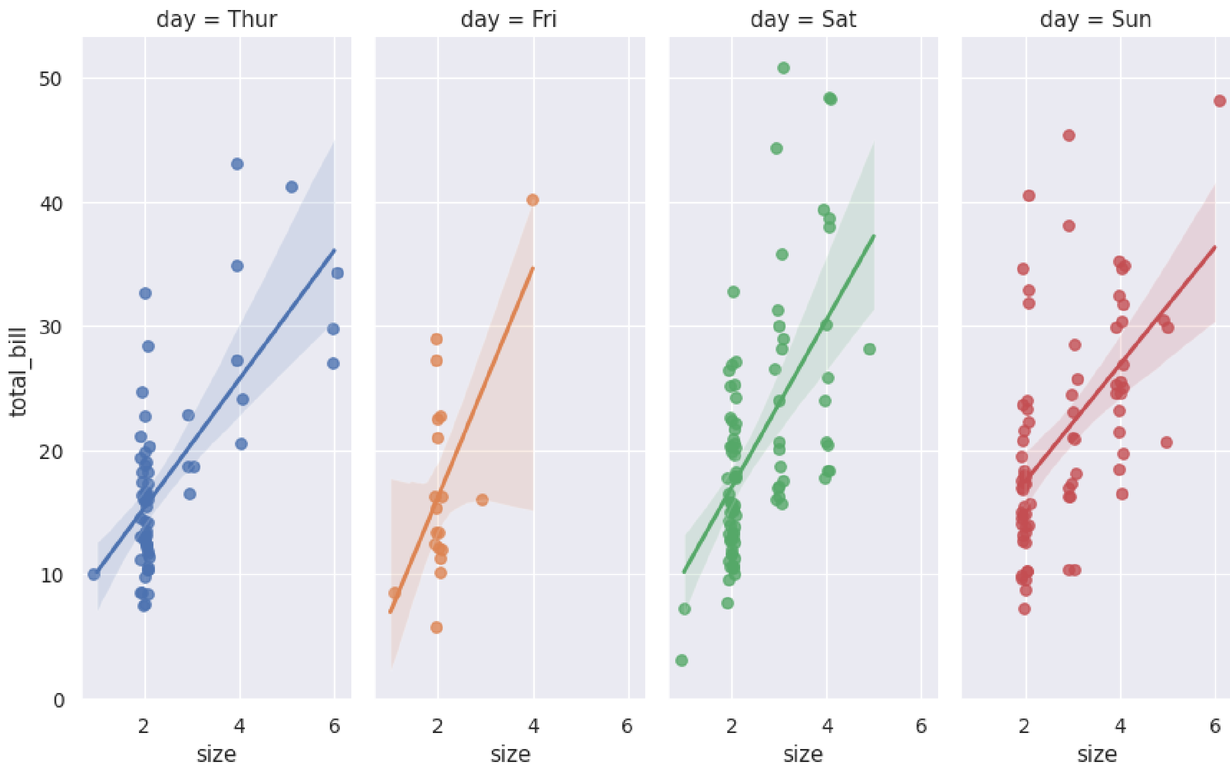

10
8
6
4
2
tip
10
20
30
40
50
total_bill
smoker
Yes
No


10
8
6
4
2
tip
10
20
30
40
50
total_bill
smoker
Yes
No


tip
total_bill
smoker
Yes
No
10
8
6
4
2
10
20
30
40
50


smoker = Yes
smoker = No
tip
total_bill
total_bill

```
>>> g = sns.lmplot(x="size", y="total_bill", hue="day", col="day",
...                data=tips, height=6, aspect=.4, x_jitter=.1)
```




Wrap the levels of the column variable into multiple rows:

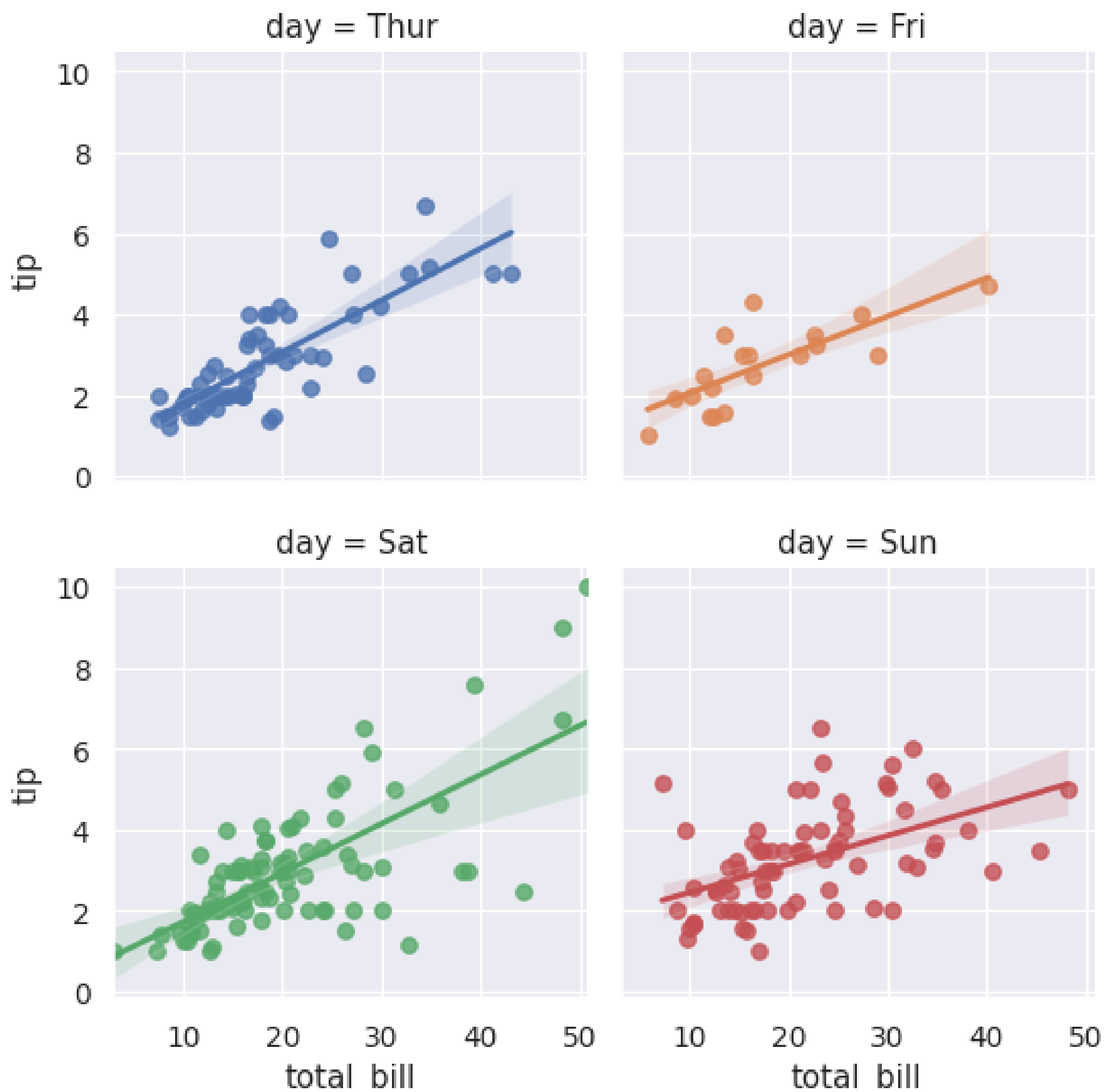
```
>>> g = sns.lmplot(x="total_bill", y="tip", col="day", hue="day",
...                data=tips, col_wrap=2, height=3)
```

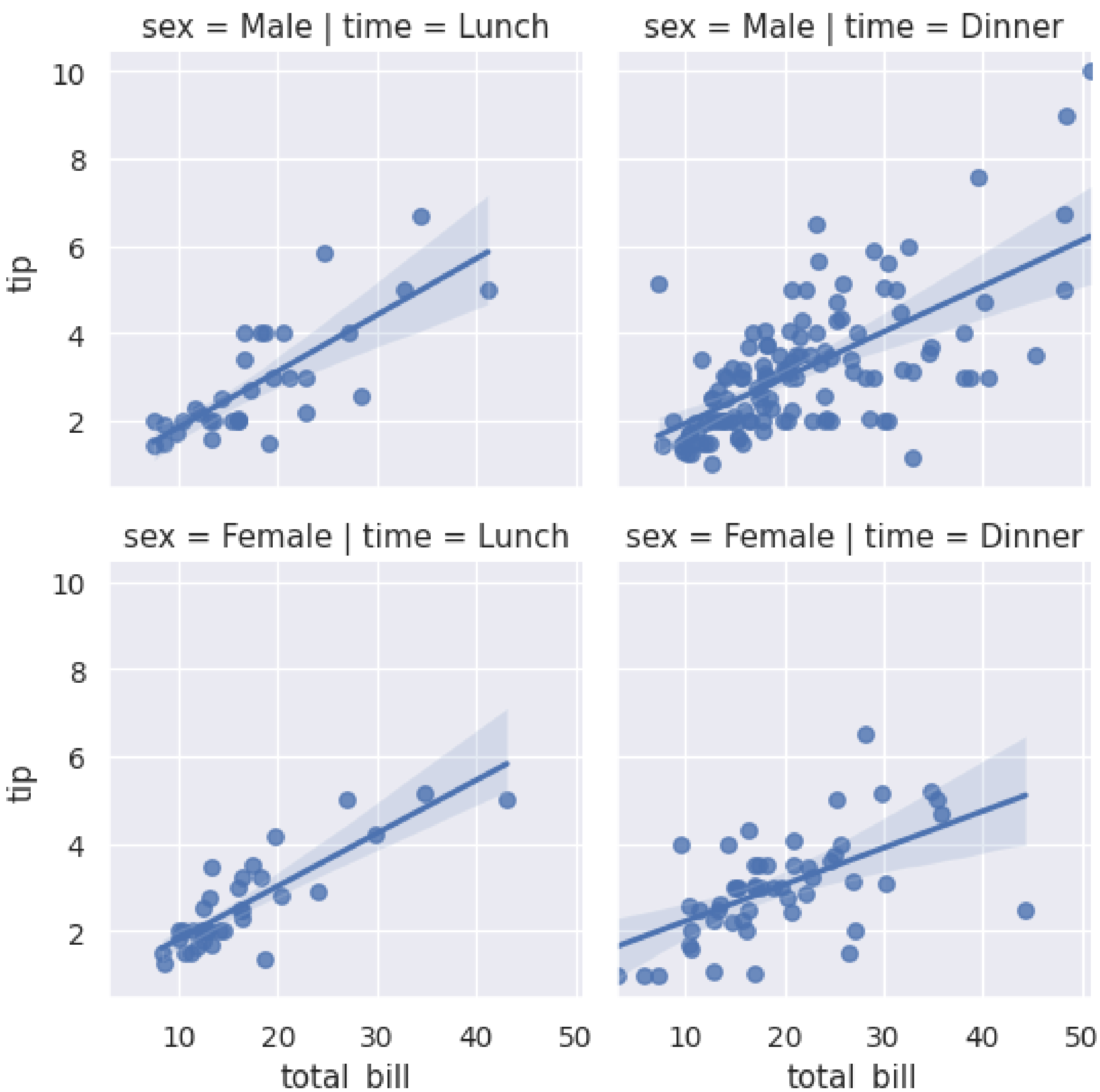
Condition on two variables to make a full grid:

```
>>> g = sns.lmplot(x="total_bill", y="tip", row="sex", col="time",
...                data=tips, height=3)
```

Use methods on the returned *FacetGrid* instance to further tweak the plot:

```
>>> g = sns.lmplot(x="total_bill", y="tip", row="sex", col="time",
...                data=tips, height=3)
>>> g = (g.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=.02))
```


day = Thur
day = Fri
day = Sat
day = Sun
tip
tip
total_bill
total_bill
0
2
4
6
8
10
10
20
30
40
50


sex = Male | time = Lunch
sex = Male | time = Dinner
sex = Female | time = Lunch
sex = Female | time = Dinner
tip
tip
total_bill
total_bill
10
8
6
4
2
10
20
30
40
50


sex = Male | time = Lunch
sex = Male | time = Dinner
sex = Female | time = Lunch
sex = Female | time = Dinner
Tip
10
6
2
Tip
10
6
2
10
30
50
Total bill (US Dollars)
10
30
50
Total bill (US Dollars)

### 5.4.2 seaborn.regplot

`seaborn.regplot(*, x=None, y=None, data=None, x_estimator=None, x_bins=None, x_ci='ci', scatter=True, fit_reg=True, ci=95, n_boot=1000, units=None, seed=None, order=1, logistic=False, lowess=False, robust=False, logx=False, x_partial=None, y_partial=None, truncate=True, dropna=True, x_jitter=None, y_jitter=None, label=None, color=None, marker='o', scatter_kws=None, line_kws=None, ax=None)`

Plot data and a linear regression model fit.

There are a number of mutually exclusive options for estimating the regression model. See the tutorial for more information.

**Parameters**

**x, y: string, series, or vector array** Input variables. If strings, these should correspond with column names in `data`. When pandas objects are used, axes will be labeled with the series name.

**data** [DataFrame] Tidy ("long-form") dataframe where each column is a variable and each row is an observation.

**x_estimator** [callable that maps vector -> scalar, optional] Apply this function to each unique value of `x` and plot the resulting estimate. This is useful when `x` is a discrete variable. If `x_ci` is given, this estimate will be bootstrapped and a confidence interval will be drawn.

**x_bins** [int or vector, optional] Bin the `x` variable into discrete bins and then estimate the central tendency and a confidence interval. This binning only influences how the scatterplot is drawn; the regression is still fit to the original data. This parameter is interpreted either as the number of evenly-sized (not necessary spaced) bins or the positions of the bin centers. When this parameter is used, it implies that the default of `x_estimator` is `numpy.mean`.

**x_ci** ["ci", "sd", int in [0, 100] or None, optional] Size of the confidence interval used when plotting a central tendency for discrete values of `x`. If `"ci"`, defer to the value of the `ci` parameter. If `"sd"`, skip bootstrapping and show the standard deviation of the observations in each bin.

**scatter** [bool, optional] If `True`, draw a scatterplot with the underlying observations (or the `x_estimator` values).

**fit_reg** [bool, optional] If `True`, estimate and plot a regression model relating the `x` and `y` variables.

**ci** [int in [0, 100] or None, optional] Size of the confidence interval for the regression estimate. This will be drawn using translucent bands around the regression line. The confidence interval is estimated using a bootstrap; for large datasets, it may be advisable to avoid that computation by setting this parameter to None.

**n_boot** [int, optional] Number of bootstrap resamples used to estimate the `ci`. The default value attempts to balance time and stability; you may want to increase this value for "final" versions of plots.

**units** [variable name in `data`, optional] If the `x` and `y` observations are nested within sampling units, those can be specified here. This will be taken into account when computing the confidence intervals by performing a multilevel bootstrap that resamples both units and observations (within unit). This does not otherwise influence how the regression is estimated or drawn.

**seed** [int, numpy.random.Generator, or numpy.random.RandomState, optional] Seed or random number generator for reproducible bootstrapping.

**order** [int, optional] If `order` is greater than 1, use `numpy.polyfit` to estimate a polynomial regression.

**logistic** [bool, optional] If `True`, assume that `y` is a binary variable and use `statsmodels` to estimate a logistic regression model. Note that this is substantially more computationally intensive than linear regression, so you may wish to decrease the number of bootstrap resamples (`n_boot`) or set `ci` to None.

**lowess** [bool, optional] If `True`, use `statsmodels` to estimate a nonparametric lowess model (locally weighted linear regression). Note that confidence intervals cannot currently be drawn for this kind of model.

**robust** [bool, optional] If `True`, use `statsmodels` to estimate a robust regression. This will de-weight outliers. Note that this is substantially more computationally intensive than standard linear regression, so you may wish to decrease the number of bootstrap resamples (`n_boot`) or set `ci` to None.

**logx** [bool, optional] If `True`, estimate a linear regression of the form y ~ log(x), but plot the scatterplot and regression model in the input space. Note that `x` must be positive for this to work.

**{x,y}_partial** [strings in `data` or matrices] Confounding variables to regress out of the `x` or `y` variables before plotting.

**truncate** [bool, optional] If `True`, the regression line is bounded by the data limits. If `False`, it extends to the `x` axis limits.

**{x,y}_jitter** [floats, optional] Add uniform random noise of this size to either the `x` or `y` variables. The noise is added to a copy of the data after fitting the regression, and only influences the look of the scatterplot. This can be helpful when plotting variables that take discrete values.

**label** [string] Label to apply to either the scatterplot or regression line (if `scatter` is `False`) for use in a legend.

**color** [matplotlib color] Color to apply to all plot elements; will be superseded by colors passed in `scatter_kws` or `line_kws`.

**marker** [matplotlib marker code] Marker to use for the scatterplot glyphs.

**{scatter,line}_kws** [dictionaries] Additional keyword arguments to pass to `plt.scatter` and `plt.plot`.

**ax** [matplotlib Axes, optional] Axes object to draw the plot onto, otherwise uses the current Axes.

**Returns**

**ax** [matplotlib Axes] The Axes object containing the plot.

**See also:**

***lmplot*** Combine *regplot()* and *FacetGrid* to plot multiple linear relationships in a dataset.

***jointplot*** Combine *regplot()* and *JointGrid* (when used with `kind="reg"`).

***pairplot*** Combine *regplot()* and *PairGrid* (when used with `kind="reg"`).

***residplot*** Plot the residuals of a linear regression model.

### Notes

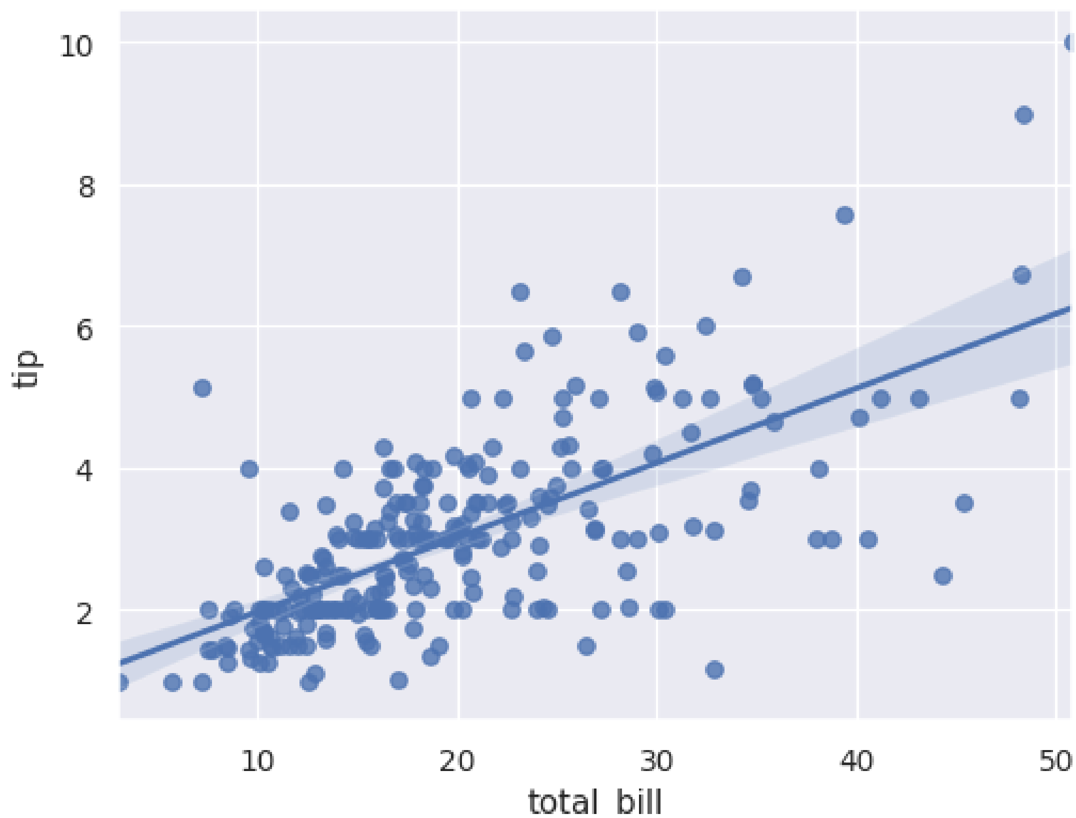
The `regplot()` and `lmplot()` functions are closely related, but the former is an axes-level function while the latter is a figure-level function that combines `regplot()` and `FacetGrid`.

It's also easy to combine combine `regplot()` and `JointGrid` or `PairGrid` through the `jointplot()` and `pairplot()` functions, although these do not directly accept all of `regplot()`'s parameters.

### Examples

Plot the relationship between two variables in a DataFrame:

```
>>> import seaborn as sns; sns.set_theme(color_codes=True)
>>> tips = sns.load_dataset("tips")
>>> ax = sns.regplot(x="total_bill", y="tip", data=tips)
```




Plot with two variables defined as numpy arrays; use a different color:

```
>>> import numpy as np; np.random.seed(8)
>>> mean, cov = [4, 6], [(1.5, .7), (.7, 1)]
>>> x, y = np.random.multivariate_normal(mean, cov, 80).T
>>> ax = sns.regplot(x=x, y=y, color="g")
```

Plot with two variables defined as pandas Series; use a different marker:


```
>>> import pandas as pd
>>> x, y = pd.Series(x, name="x_var"), pd.Series(y, name="y_var")
>>> ax = sns.regplot(x=x, y=y, marker="+")
```

Use a 68% confidence interval, which corresponds with the standard error of the estimate, and extend the regression line to the axis limits:

```
>>> ax = sns.regplot(x=x, y=y, ci=68, truncate=False)
```

Plot with a discrete `x` variable and add some jitter:

```
>>> ax = sns.regplot(x="size", y="total_bill", data=tips, x_jitter=.1)
```

Plot with a discrete `x` variable showing means and confidence intervals for unique values:

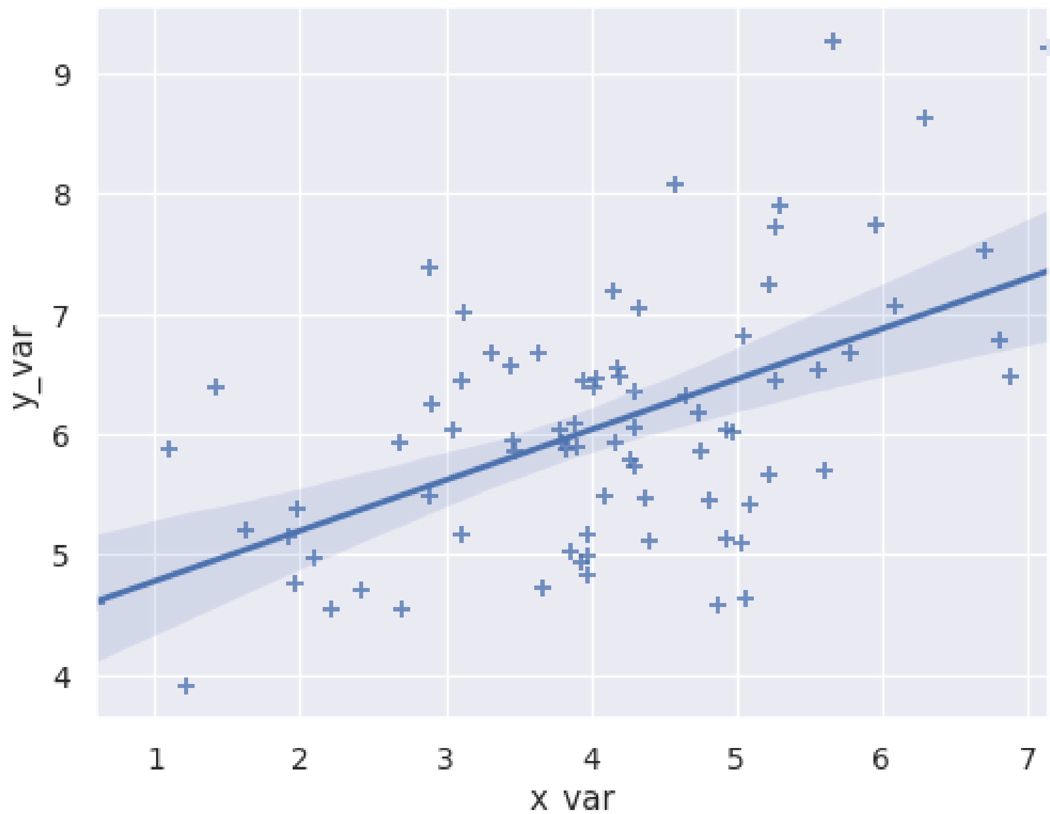
```
>>> ax = sns.regplot(x="size", y="total_bill", data=tips,
...                  x_estimator=np.mean)
```

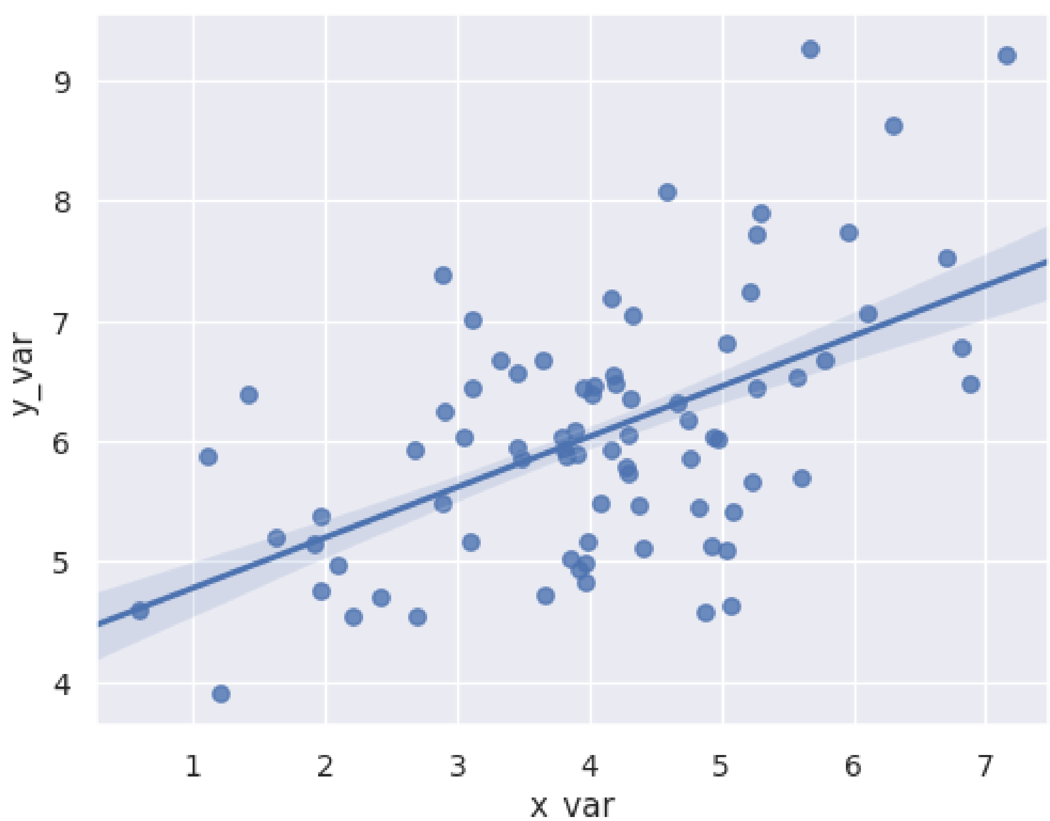
Plot with a continuous variable divided into discrete bins:

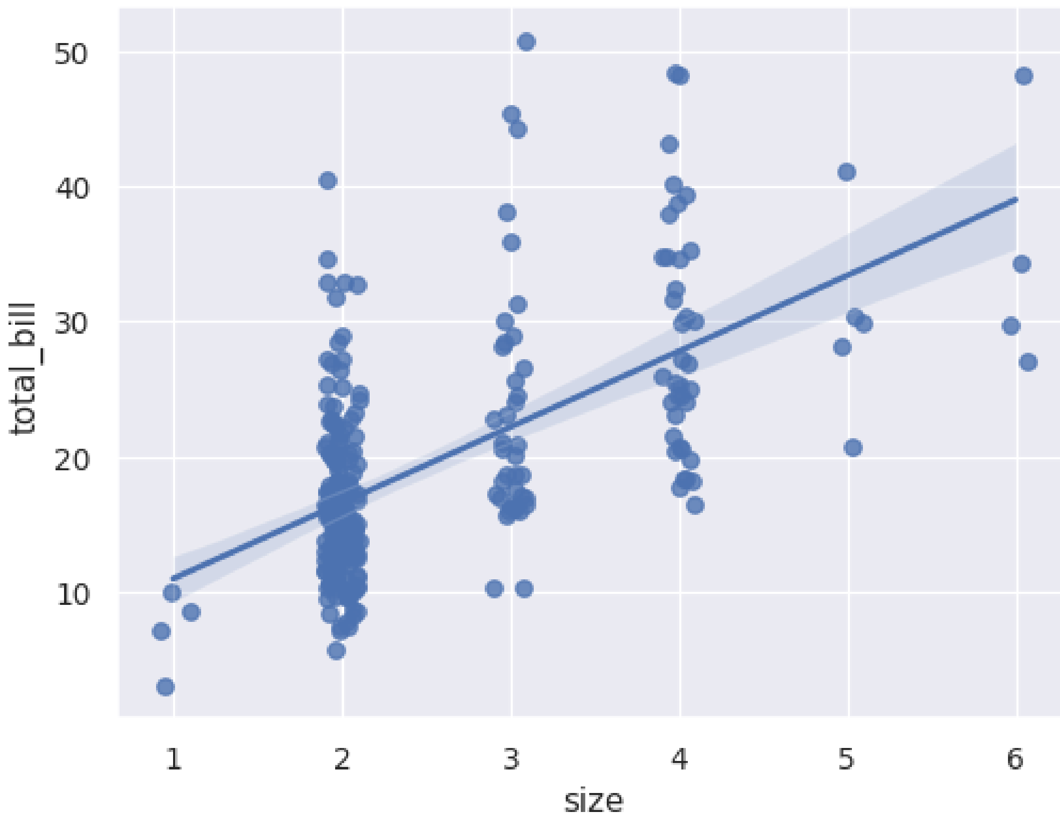
```
>>> ax = sns.regplot(x=x, y=y, x_bins=4)
```

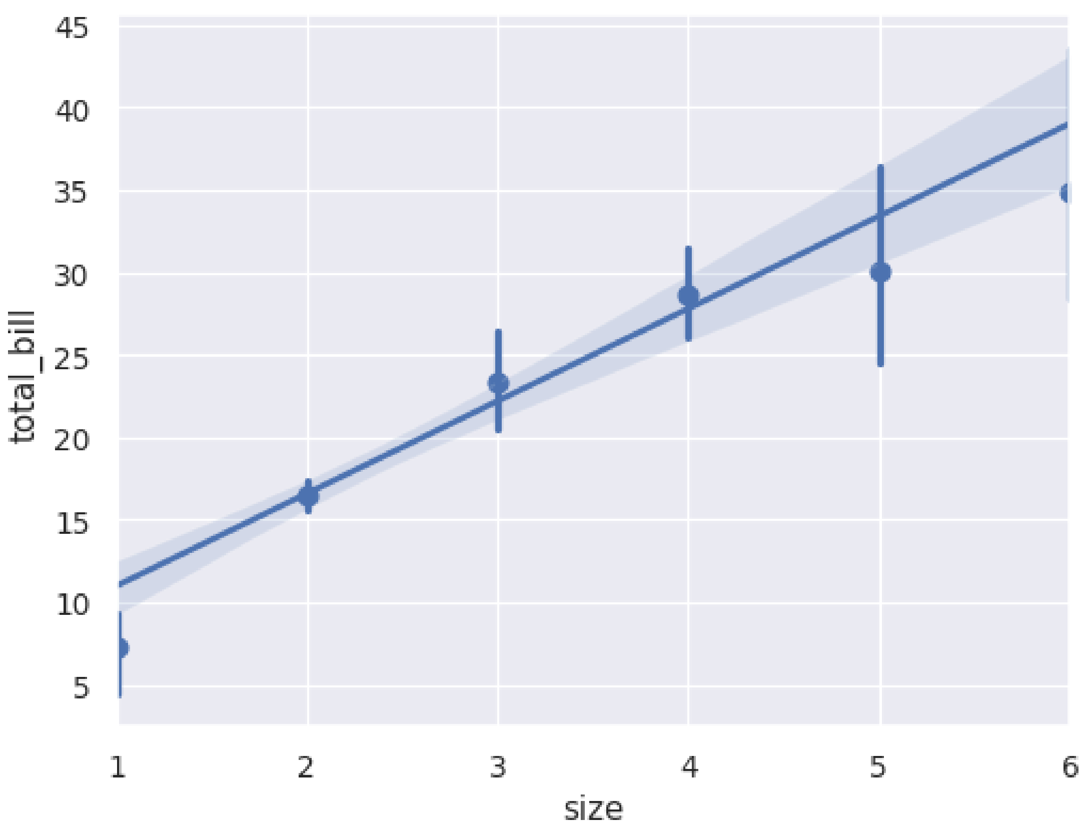
Fit a higher-order polynomial regression:

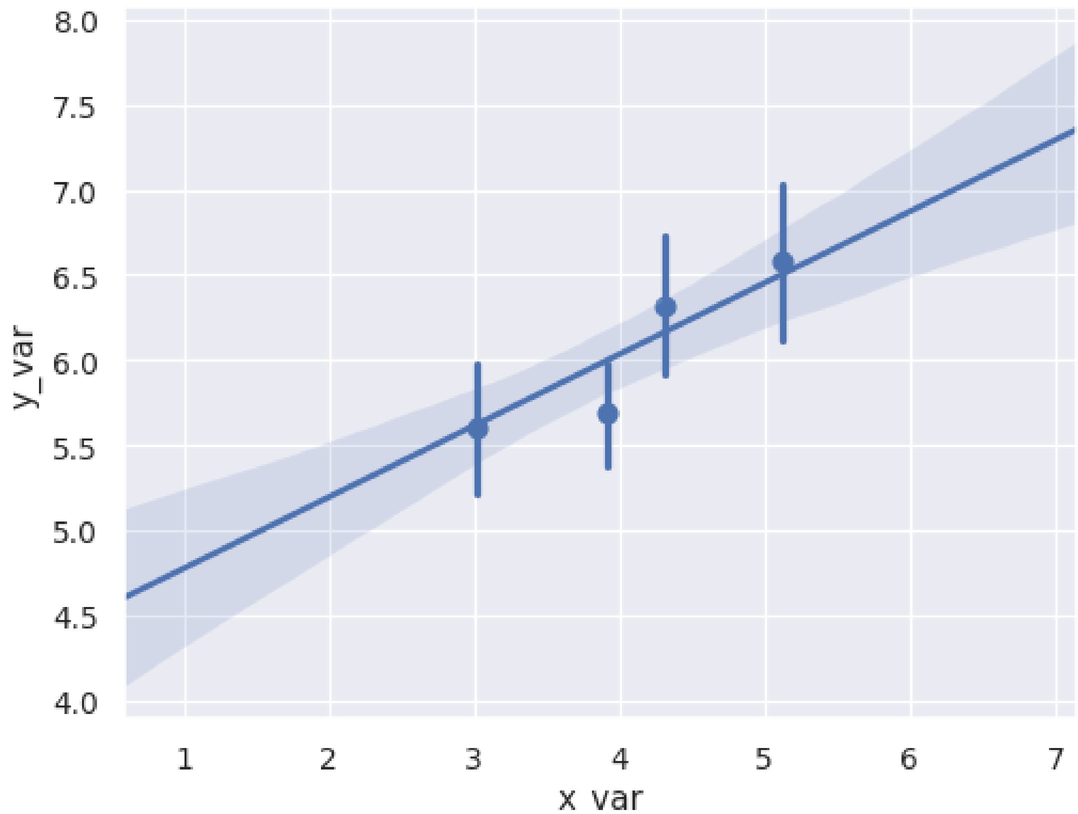
```
>>> ans = sns.load_dataset("anscombe")
>>> ax = sns.regplot(x="x", y="y", data=ans.loc[ans.dataset == "II"],
...                  scatter_kws={"s": 80},
...                  order=2, ci=None)
```

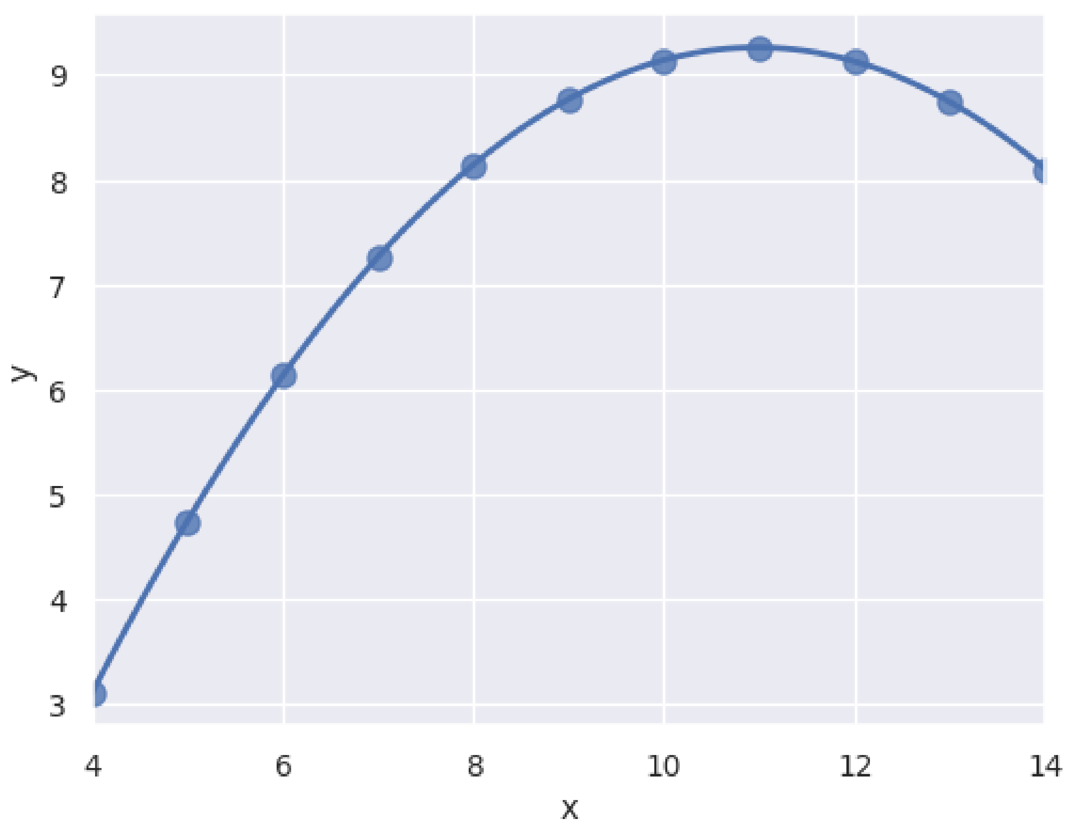

9
8
7
6
5
4
y_var
1
2
3
4
5
6
7
x_var


9
8
7
6
5
4
y_var
1
2
3
4
5
6
7
x_var


total_bill
50
40
30
20
10
1
2
3
4
5
6
size

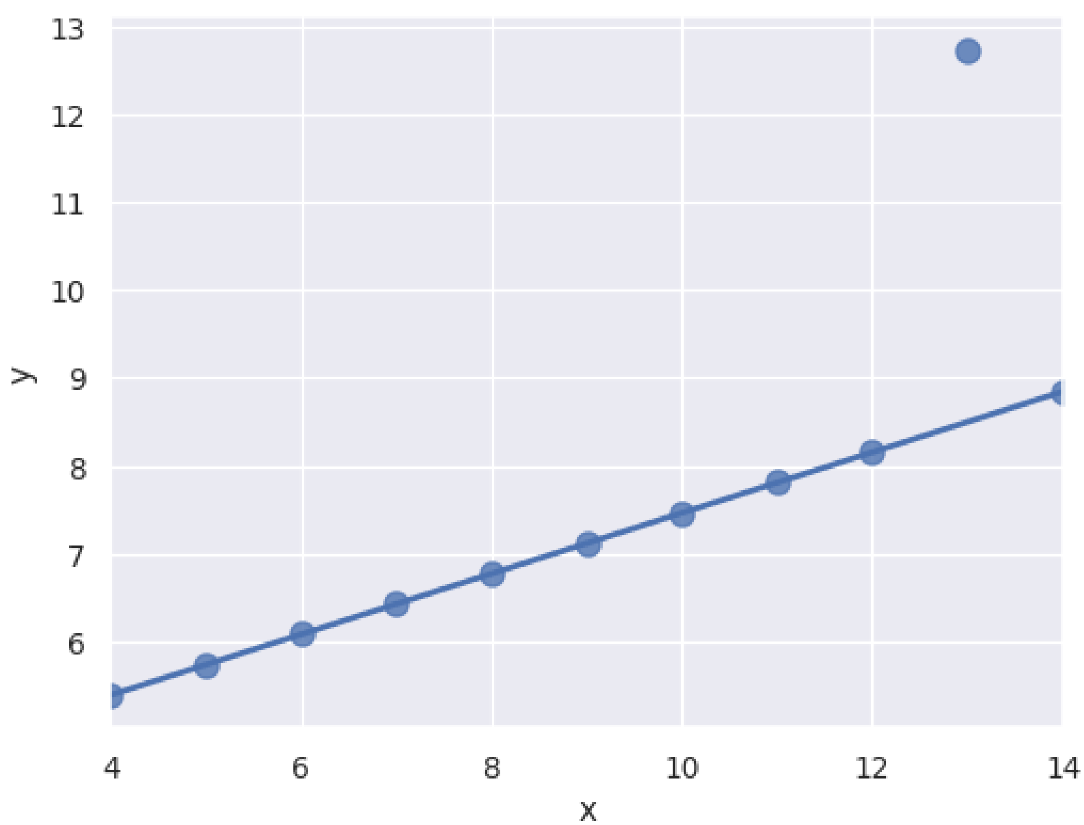

total_bill
45
40
35
30
25
20
15
10
5
1
2
3
4
5
6
size


8.0
7.5
7.0
6.5
6.0
5.5
5.0
4.5
4.0
y_var
1
2
3
4
5
6
7
x_var


9
8
7
6
5
4
3
y
4
6
8
10
12
14
x

Fit a robust regression and don't plot a confidence interval:

```
>>> ax = sns.regplot(x="x", y="y", data=ans.loc[ans.dataset == "III"],
...                  scatter_kws={"s": 80},
...                  robust=True, ci=None)
```

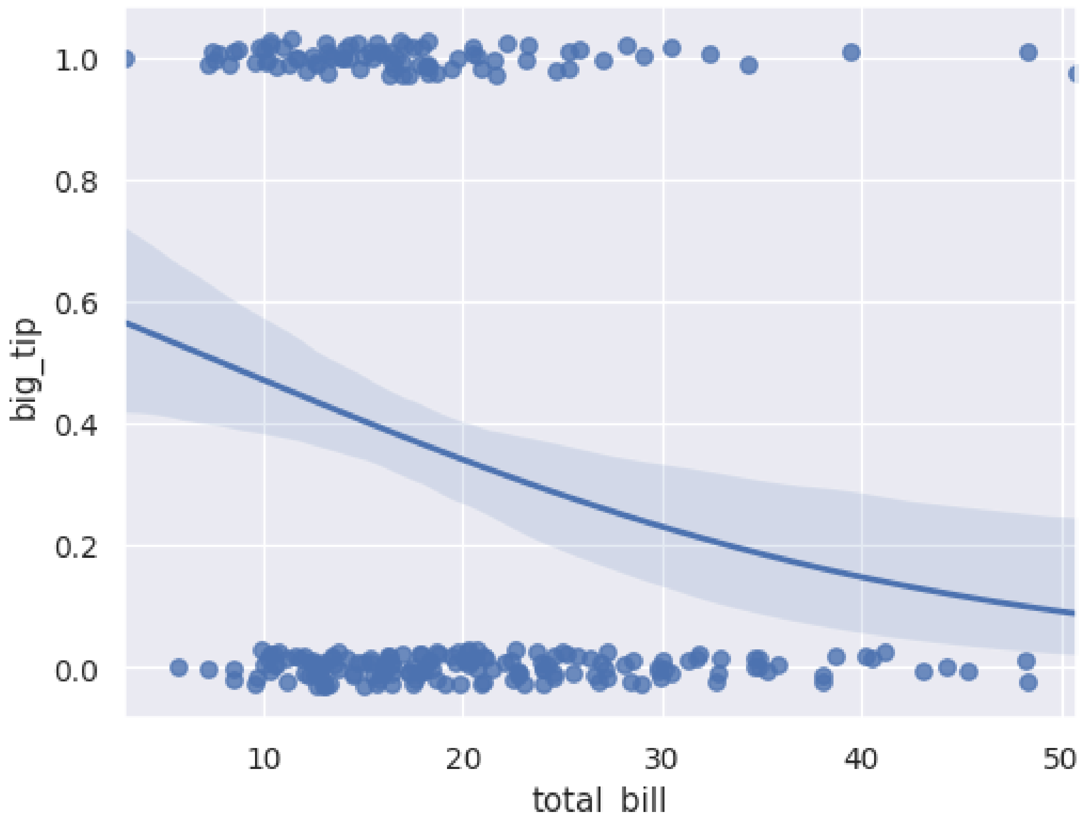


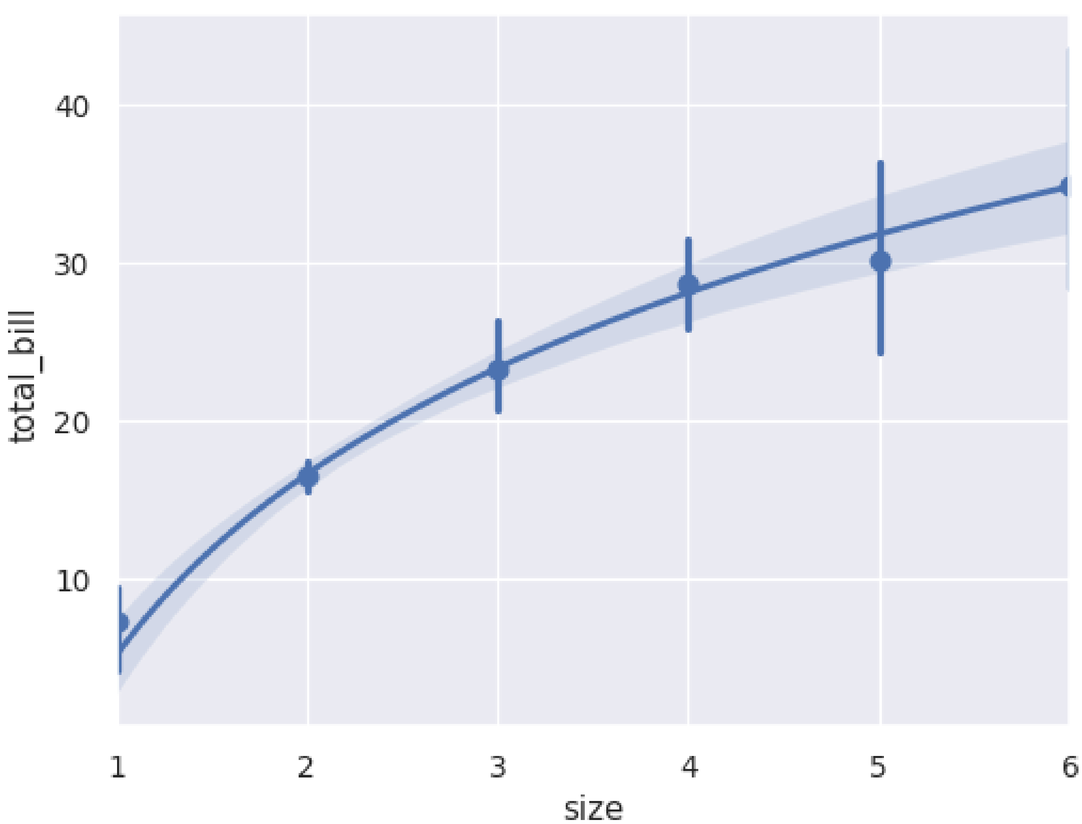

Fit a logistic regression; jitter the y variable and use fewer bootstrap iterations:

```
>>> tips["big_tip"] = (tips.tip / tips.total_bill) > .175
>>> ax = sns.regplot(x="total_bill", y="big_tip", data=tips,
...                  logistic=True, n_boot=500, y_jitter=.03)
```

Fit the regression model using log(x):

```
>>> ax = sns.regplot(x="size", y="total_bill", data=tips,
...                  x_estimator=np.mean, logx=True)
```


1.0
0.8
0.6
0.4
0.2
0.0
big_tip
10
20
30
40
50
total_bill


40
30
20
10
total_bill
1
2
3
4
5
6
size

### 5.4.3 seaborn.residplot

`seaborn.residplot(*, x=None, y=None, data=None, lowess=False, x_partial=None, y_partial=None, order=1, robust=False, dropna=True, label=None, color=None, scatter_kws=None, line_kws=None, ax=None)`

Plot the residuals of a linear regression.

This function will regress y on x (possibly as a robust or polynomial regression) and then draw a scatterplot of the residuals. You can optionally fit a lowess smoother to the residual plot, which can help in determining if there is structure to the residuals.

**Parameters**

**x** [vector or string] Data or column name in `data` for the predictor variable.

**y** [vector or string] Data or column name in `data` for the response variable.

**data** [DataFrame, optional] DataFrame to use if `x` and `y` are column names.

**lowess** [boolean, optional] Fit a lowess smoother to the residual scatterplot.

**{x, y}_partial** [matrix or string(s) , optional] Matrix with same first dimension as `x`, or column name(s) in `data`. These variables are treated as confounding and are removed from the `x` or `y` variables before plotting.

**order** [int, optional] Order of the polynomial to fit when calculating the residuals.

**robust** [boolean, optional] Fit a robust linear regression when calculating the residuals.

**dropna** [boolean, optional] If True, ignore observations with missing data when fitting and plotting.

**label** [string, optional] Label that will be used in any plot legends.

**color** [matplotlib color, optional] Color to use for all elements of the plot.

**{scatter, line}_kws** [dictionaries, optional] Additional keyword arguments passed to scatter() and plot() for drawing the components of the plot.

**ax** [matplotlib axis, optional] Plot into this axis, otherwise grab the current axis or make a new one if not existing.

**Returns**

**ax: matplotlib axes** Axes with the regression plot.

**See also:**

***regplot*** Plot a simple linear regression model.

***jointplot*** Draw a *residplot()* with univariate marginal distributions (when used with `kind="resid"`).

## 5.5 Matrix plots

| | |
|---|---|
| *heatmap* | Plot rectangular data as a color-encoded matrix. |
| *clustermap* | Plot a matrix dataset as a hierarchically-clustered heatmap. |

### 5.5.1 seaborn.heatmap

seaborn.**heatmap**(*data*, ***, *vmin=None*, *vmax=None*, *cmap=None*, *center=None*, *robust=False*, *annot=None*, *fmt='.2g'*, *annot_kws=None*, *linewidths=0*, *linecolor='white'*, *cbar=True*, *cbar_kws=None*, *cbar_ax=None*, *square=False*, *xticklabels='auto'*, *yticklabels='auto'*, *mask=None*, *ax=None*, ***kwargs*)

Plot rectangular data as a color-encoded matrix.

This is an Axes-level function and will draw the heatmap into the currently-active Axes if none is provided to the `ax` argument. Part of this Axes space will be taken and used to plot a colormap, unless `cbar` is False or a separate Axes is provided to `cbar_ax`.

**Parameters**

**data** [rectangular dataset] 2D dataset that can be coerced into an ndarray. If a Pandas DataFrame is provided, the index/column information will be used to label the columns and rows.

**vmin, vmax** [floats, optional] Values to anchor the colormap, otherwise they are inferred from the data and other keyword arguments.

**cmap** [matplotlib colormap name or object, or list of colors, optional] The mapping from data values to color space. If not provided, the default will depend on whether `center` is set.

**center** [float, optional] The value at which to center the colormap when plotting divergant data. Using this parameter will change the default `cmap` if none is specified.

**robust** [bool, optional] If True and `vmin` or `vmax` are absent, the colormap range is computed with robust quantiles instead of the extreme values.

**annot** [bool or rectangular dataset, optional] If True, write the data value in each cell. If an array-like with the same shape as `data`, then use this to annotate the heatmap instead of the data. Note that DataFrames will match on position, not index.

**fmt** [str, optional] String formatting code to use when adding annotations.

**annot_kws** [dict of key, value mappings, optional] Keyword arguments for `matplotlib.axes.Axes.text()` when `annot` is True.

**linewidths** [float, optional] Width of the lines that will divide each cell.

**linecolor** [color, optional] Color of the lines that will divide each cell.

**cbar** [bool, optional] Whether to draw a colorbar.

**cbar_kws** [dict of key, value mappings, optional] Keyword arguments for `matplotlib.figure.Figure.colorbar()`.

**cbar_ax** [matplotlib Axes, optional] Axes in which to draw the colorbar, otherwise take space from the main Axes.

**square** [bool, optional] If True, set the Axes aspect to "equal" so each cell will be square-shaped.

**xticklabels, yticklabels** [“auto”, bool, list-like, or int, optional] If True, plot the column names of the dataframe. If False, don’t plot the column names. If list-like, plot these alternate labels as the xticklabels. If an integer, use the column names but plot only every n label. If “auto”, try to densely plot non-overlapping labels.

**mask** [bool array or DataFrame, optional] If passed, data will not be shown in cells where `mask` is True. Cells with missing values are automatically masked.

**ax** [matplotlib Axes, optional] Axes in which to draw the plot, otherwise use the currently-active Axes.

**kwargs** [other keyword arguments] All other keyword arguments are passed to `matplotlib.axes.Axes.pcolormesh()`.

**Returns**

**ax** [matplotlib Axes] Axes object with the heatmap.

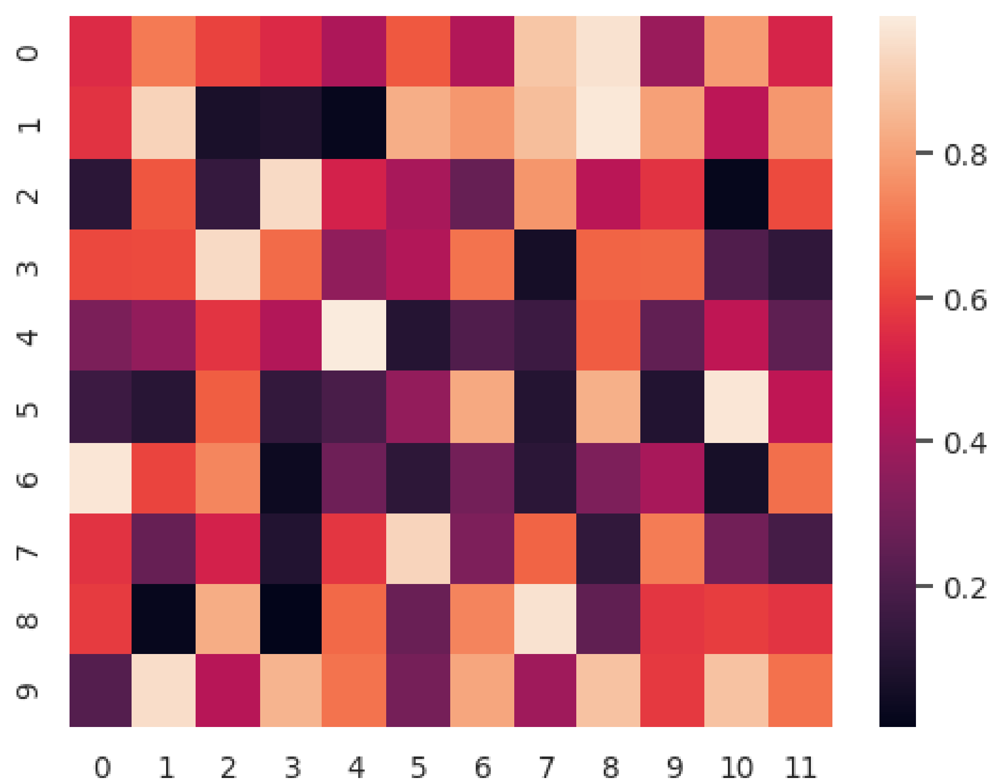
**See also:**

***clustermap*** Plot a matrix using hierachical clustering to arrange the rows and columns.

## Examples

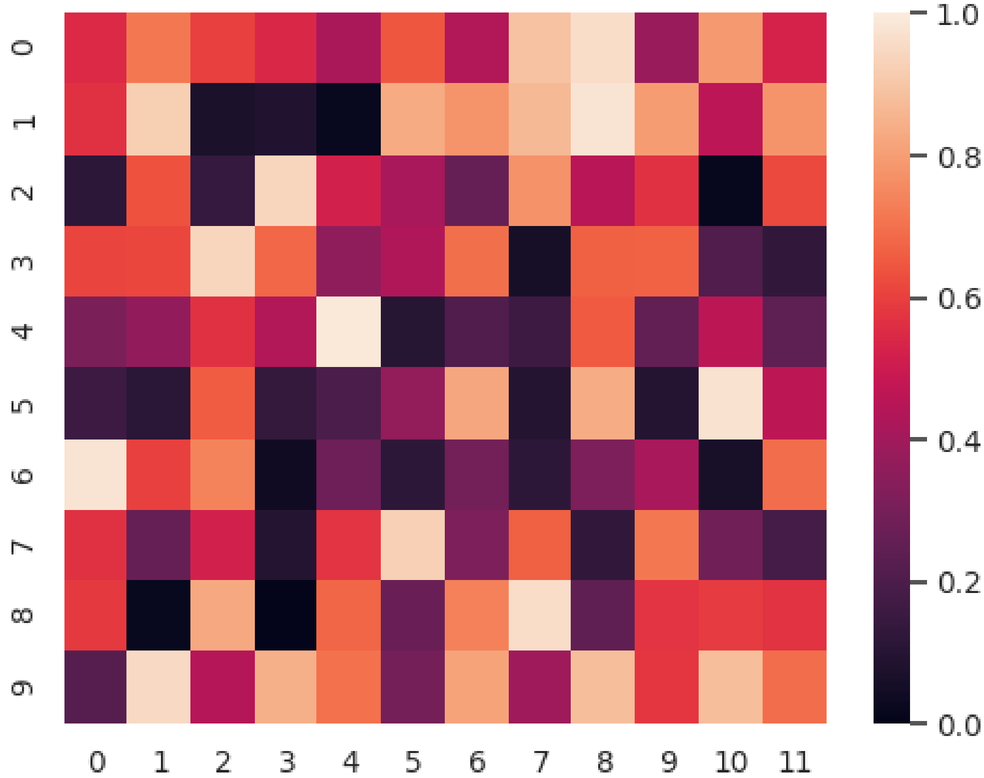
Plot a heatmap for a numpy array:

```
>>> import numpy as np; np.random.seed(0)
>>> import seaborn as sns; sns.set_theme()
>>> uniform_data = np.random.rand(10, 12)
>>> ax = sns.heatmap(uniform_data)
```



Change the limits of the colormap:

```
>>> ax = sns.heatmap(uniform_data, vmin=0, vmax=1)
```




Plot a heatmap for data centered on 0 with a diverging colormap:

```
>>> normal_data = np.random.randn(10, 12)
>>> ax = sns.heatmap(normal_data, center=0)
```

Plot a dataframe with meaningful row and column labels:

```
>>> flights = sns.load_dataset("flights")
>>> flights = flights.pivot("month", "year", "passengers")
>>> ax = sns.heatmap(flights)
```

Annotate each cell with the numeric value using integer formatting:

```
>>> ax = sns.heatmap(flights, annot=True, fmt="d")
```

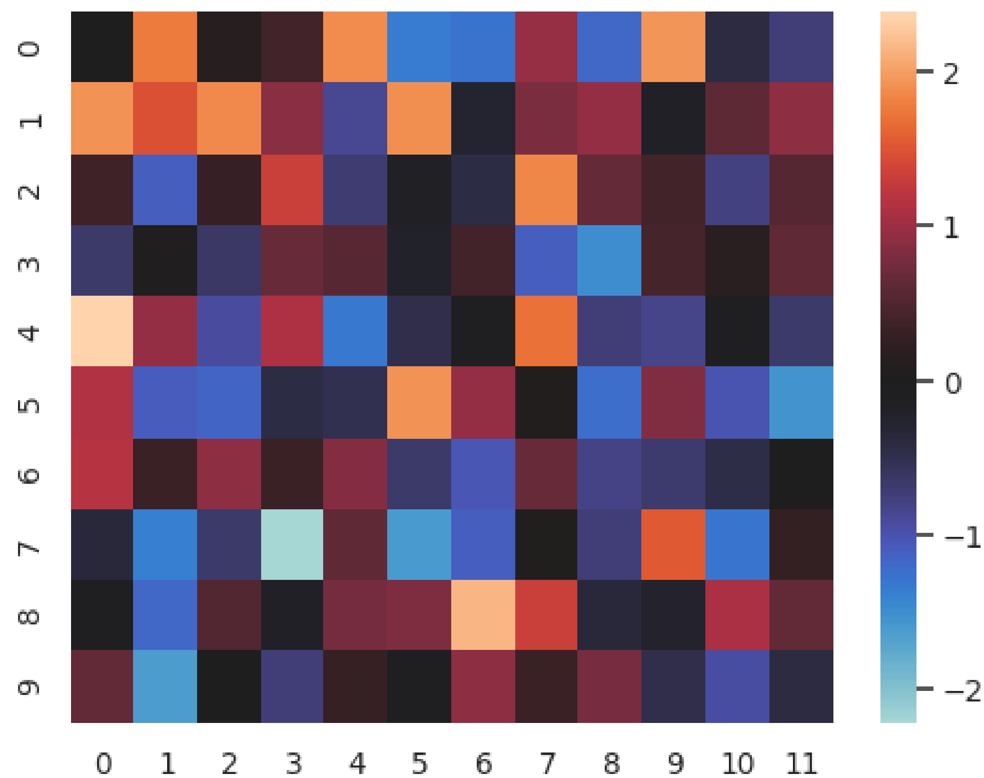
Add lines between each cell:

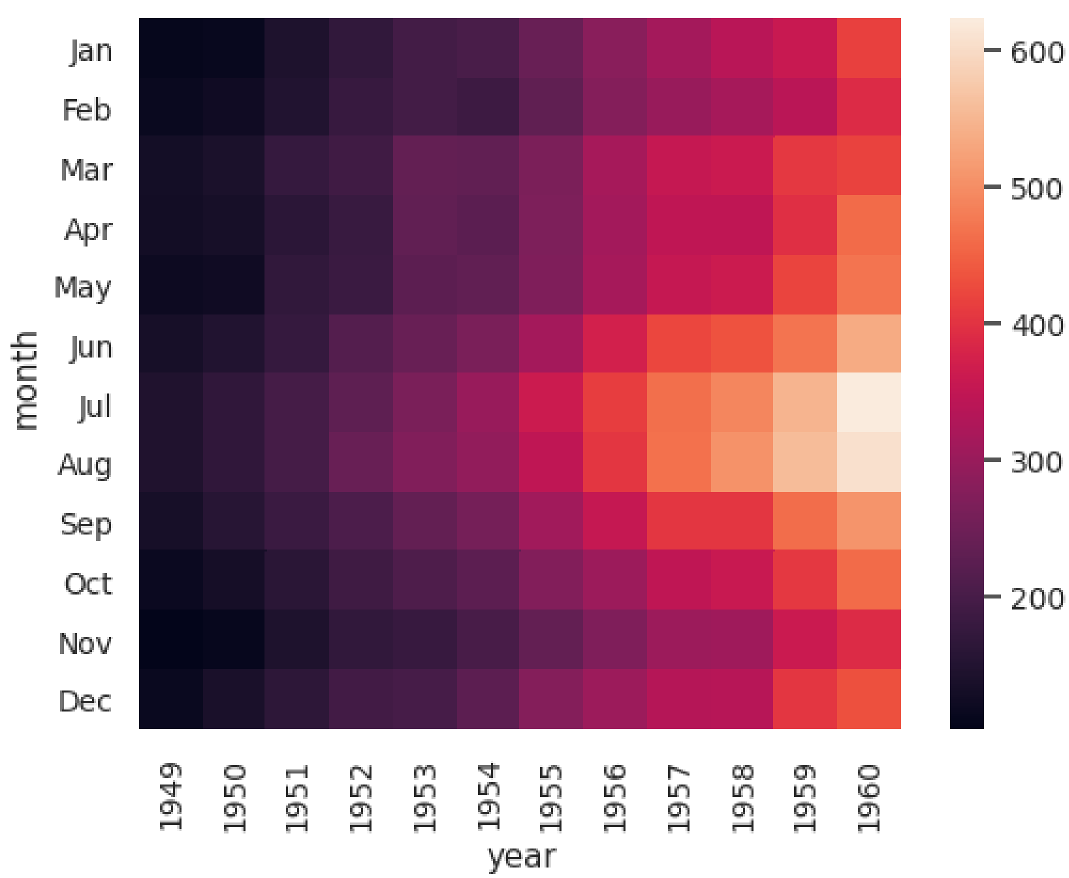
```
>>> ax = sns.heatmap(flights, linewidths=.5)
```

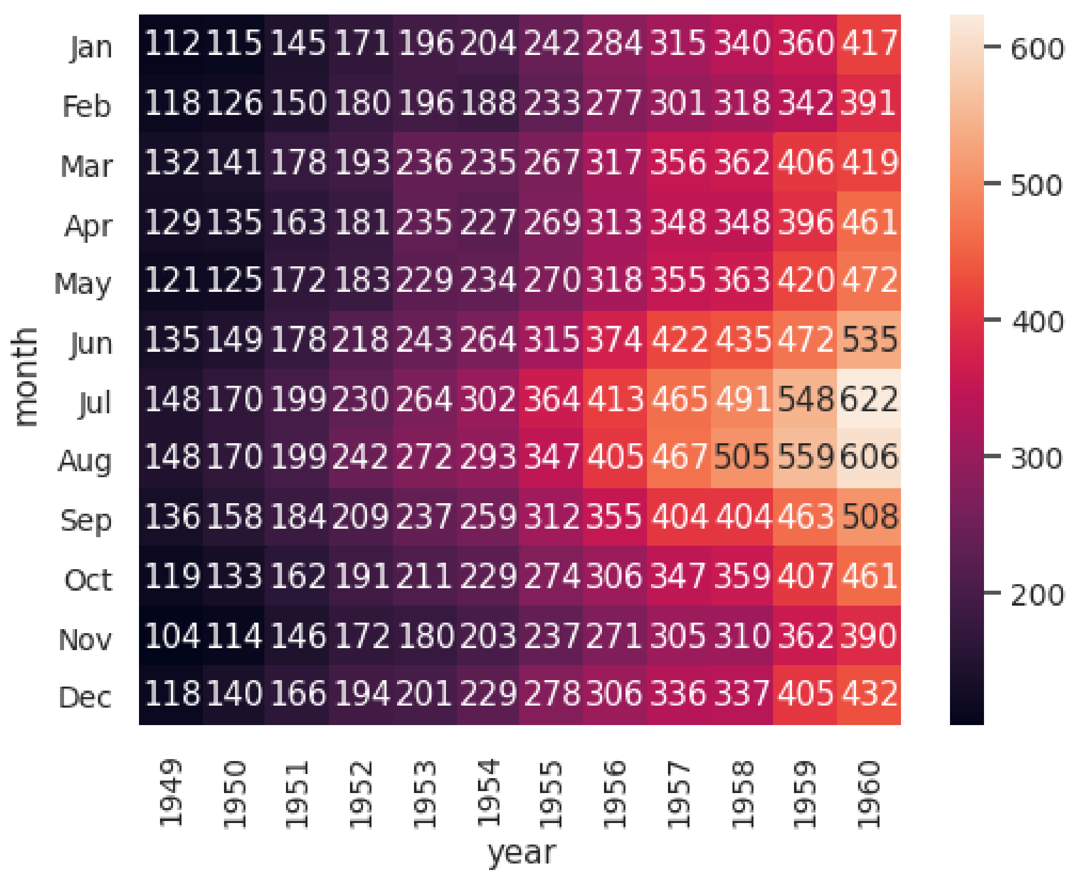
Use a different colormap:

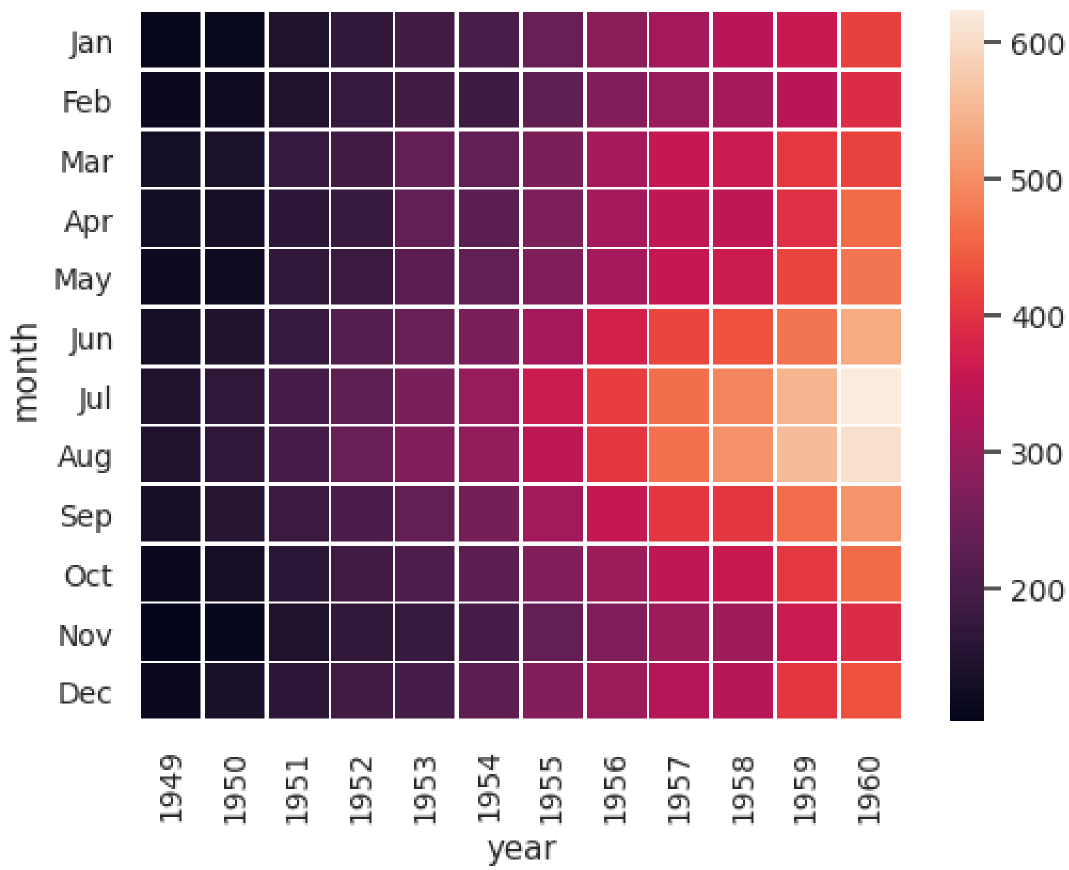
```
>>> ax = sns.heatmap(flights, cmap="YlGnBu")
```

Center the colormap at a specific value:

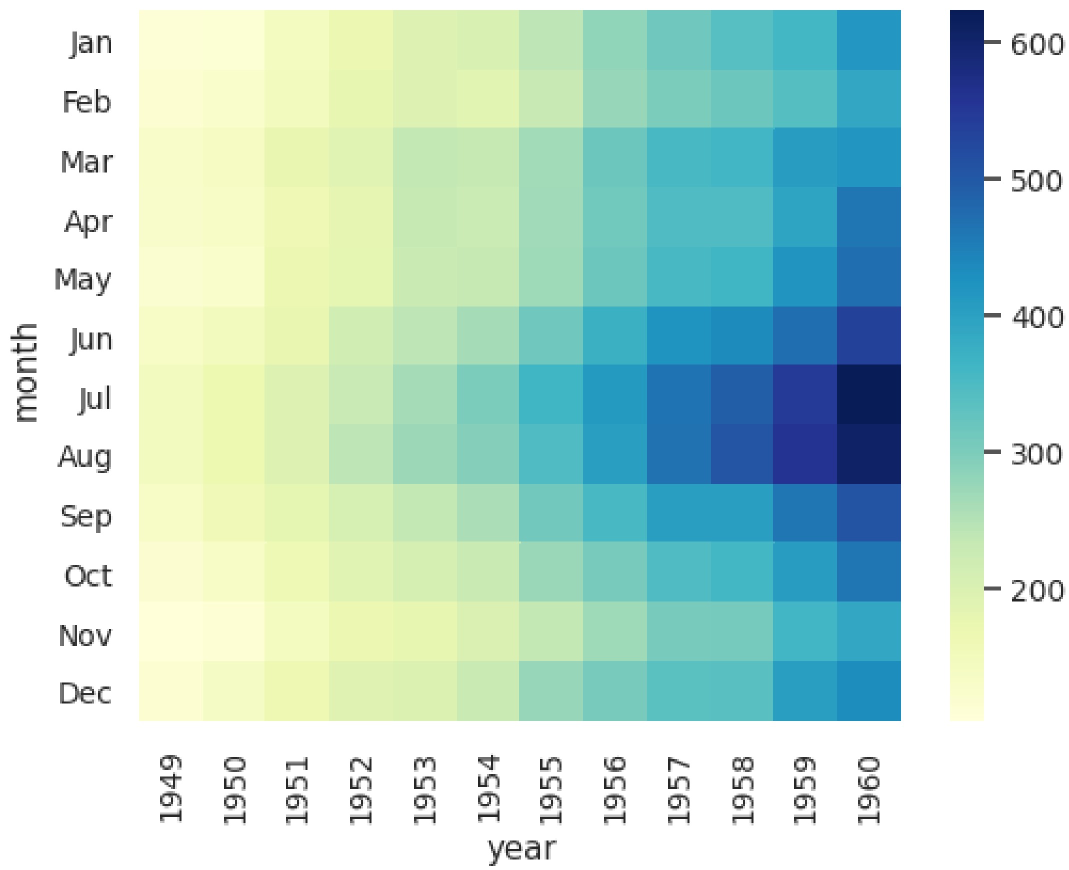

0
1
2
3
4
5
6
7
8
9
10
11
2
1
0
−1
−2


Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
month
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
year
600
500
400
300
200


Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
month
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
year
600
500
400
300
200
112 115 145 171 196 204 242 284 315 340 360 417
118 126 150 180 196 188 233 277 301 318 342 391
132 141 178 193 236 235 267 317 356 362 406 419
129 135 163 181 235 227 269 313 348 348 396 461
121 125 172 183 229 234 270 318 355 363 420 472
135 149 178 218 243 264 315 374 422 435 472 535
148 170 199 230 264 302 364 413 465 491 548 622
148 170 199 242 272 293 347 405 467 505 559 606
136 158 184 209 237 259 312 355 404 404 463 508
119 133 162 191 211 229 274 306 347 359 407 461
104 114 146 172 180 203 237 271 305 310 362 390
118 140 166 194 201 229 278 306 336 337 405 432

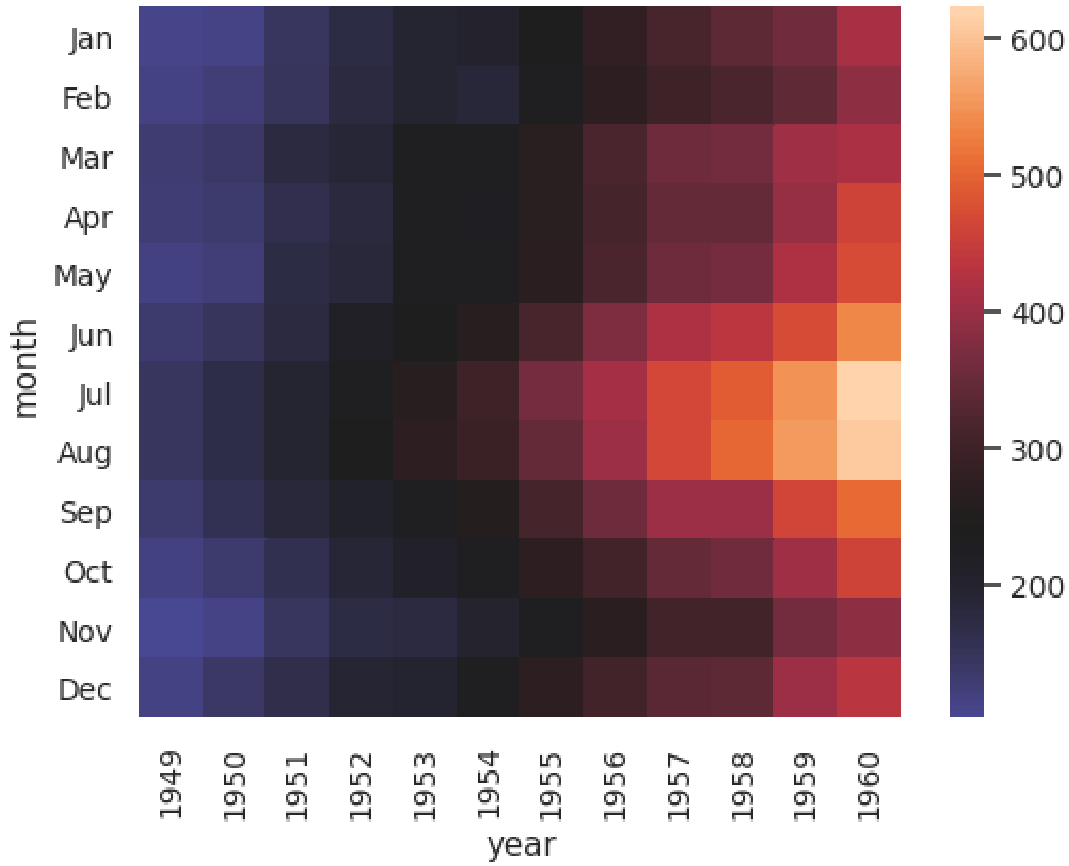


Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
month
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
year
600
500
400
300
200



Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
month
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
year
600
500
400
300
200

```
>>> ax = sns.heatmap(flights, center=flights.loc["Jan", 1955])
```




Plot every other column label and don't plot row labels:

```
>>> data = np.random.randn(50, 20)
>>> ax = sns.heatmap(data, xticklabels=2, yticklabels=False)
```

Don't draw a colorbar:

```
>>> ax = sns.heatmap(flights, cbar=False)
```

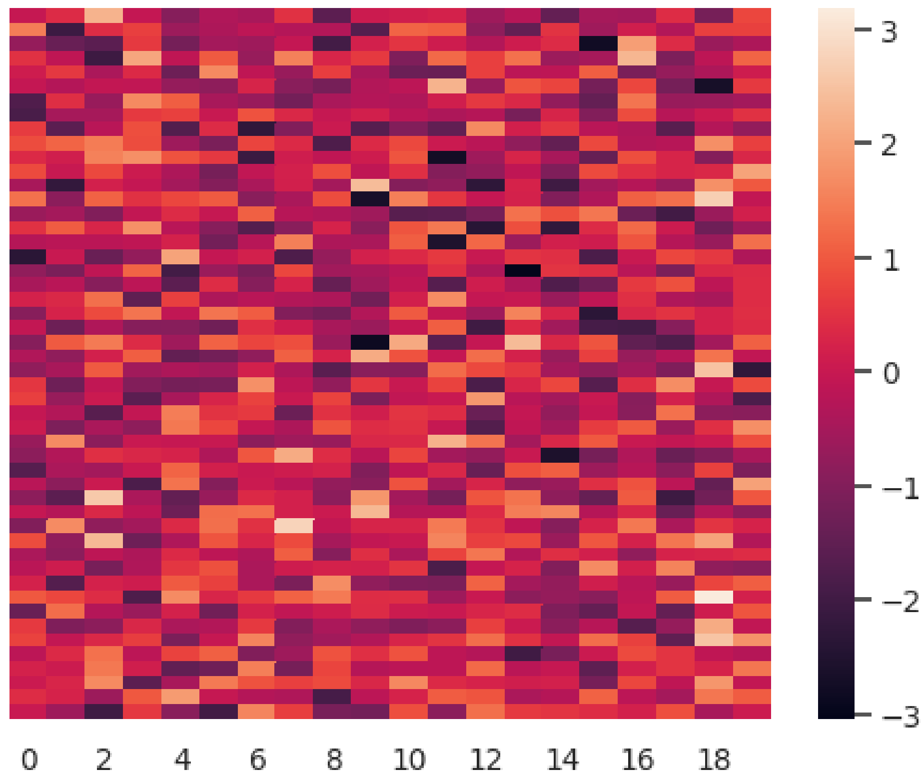
Use different axes for the colorbar:

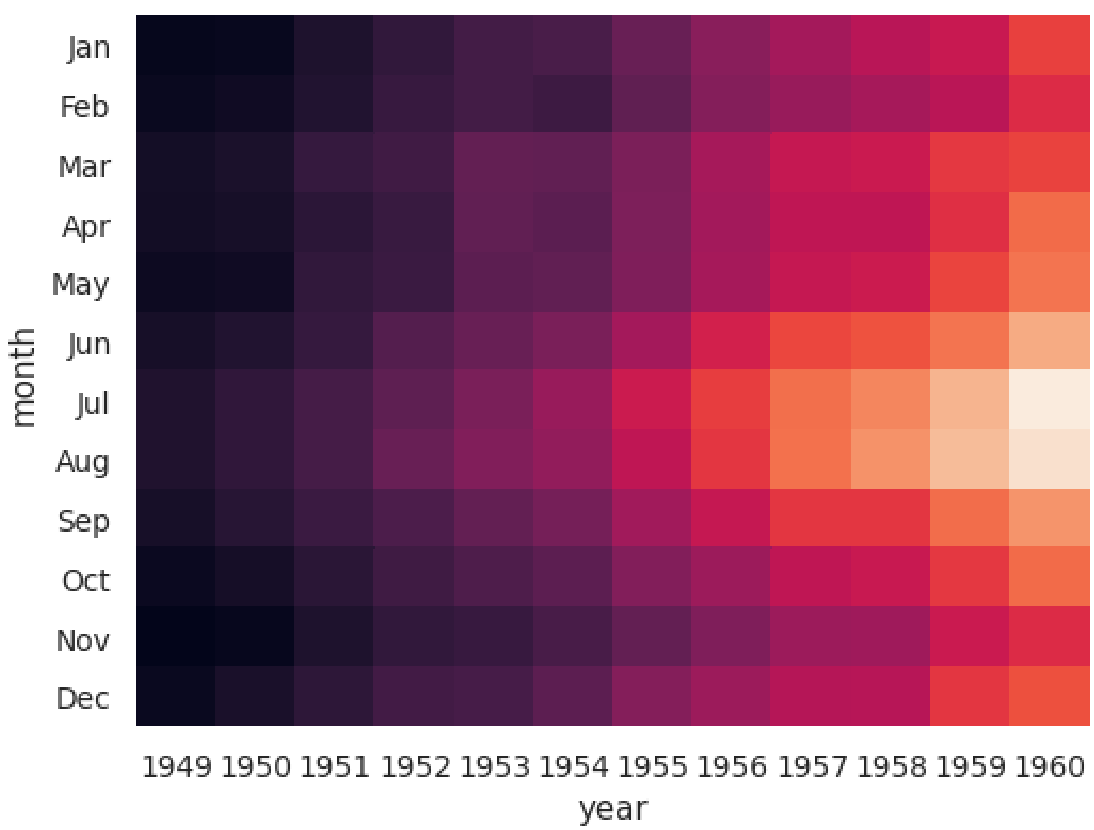
```
>>> grid_kws = {"height_ratios": (.9, .05), "hspace": .3}
>>> f, (ax, cbar_ax) = plt.subplots(2, gridspec_kw=grid_kws)
>>> ax = sns.heatmap(flights, ax=ax,
...                  cbar_ax=cbar_ax,
...                  cbar_kws={"orientation": "horizontal"})
```

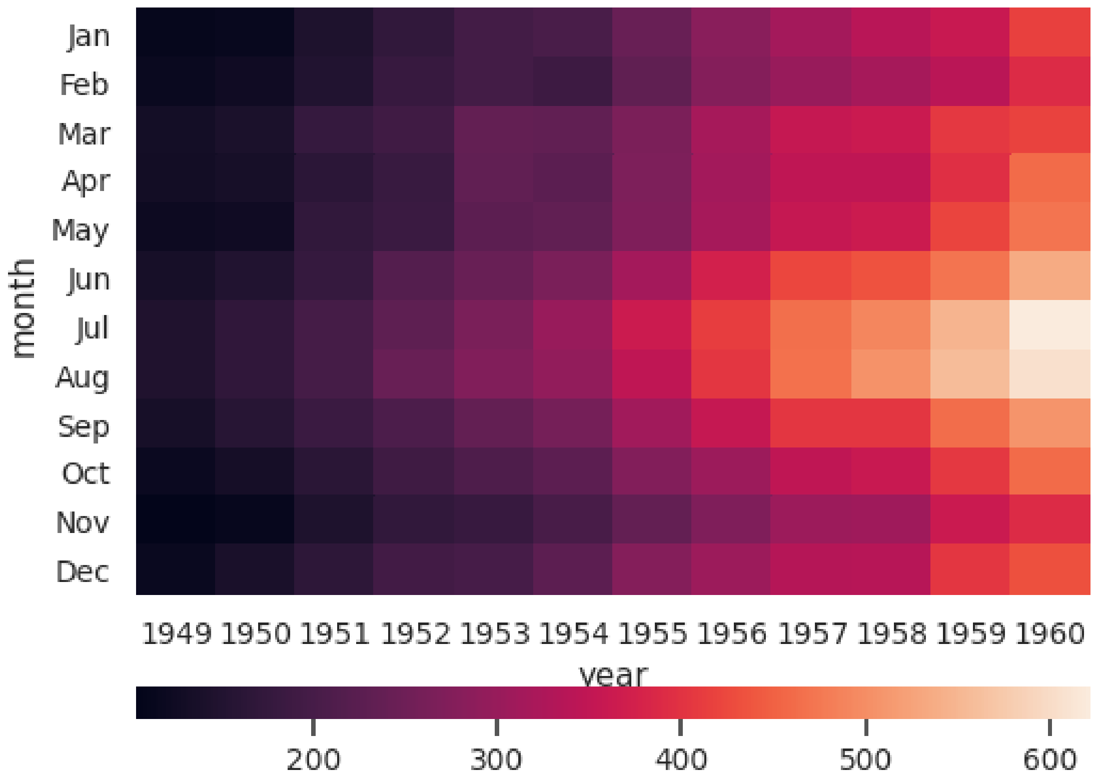
Use a mask to plot only part of a matrix

```
>>> corr = np.corrcoef(np.random.randn(10, 200))
>>> mask = np.zeros_like(corr)
>>> mask[np.triu_indices_from(mask)] = True
>>> with sns.axes_style("white"):
```

(continues on next page)


3
2
1
0
−1
−2
−3
0
2
4
6
8
10
12
14
16
18


Jan
Feb
Mar
Apr
May
Jun
Jul
Aug
Sep
Oct
Nov
Dec
month
1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960
year




(continued from previous page)

```
...     f, ax = plt.subplots(figsize=(7, 5))
...     ax = sns.heatmap(corr, mask=mask, vmax=.3, square=True)
```

### 5.5.2 seaborn.clustermap

seaborn.**clustermap**(*data*, ***, *pivot_kws=None*, *method='average'*, *metric='euclidean'*, *z_score=None*, *standard_scale=None*, *figsize=(10, 10)*, *cbar_kws=None*, *row_cluster=True*, *col_cluster=True*, *row_linkage=None*, *col_linkage=None*, *row_colors=None*, *col_colors=None*, *mask=None*, *dendrogram_ratio=0.2*, *colors_ratio=0.03*, *cbar_pos=(0.02, 0.8, 0.05, 0.18)*, *tree_kws=None*, ***kwargs*)

Plot a matrix dataset as a hierarchically-clustered heatmap.

This function requires scipy to be available.

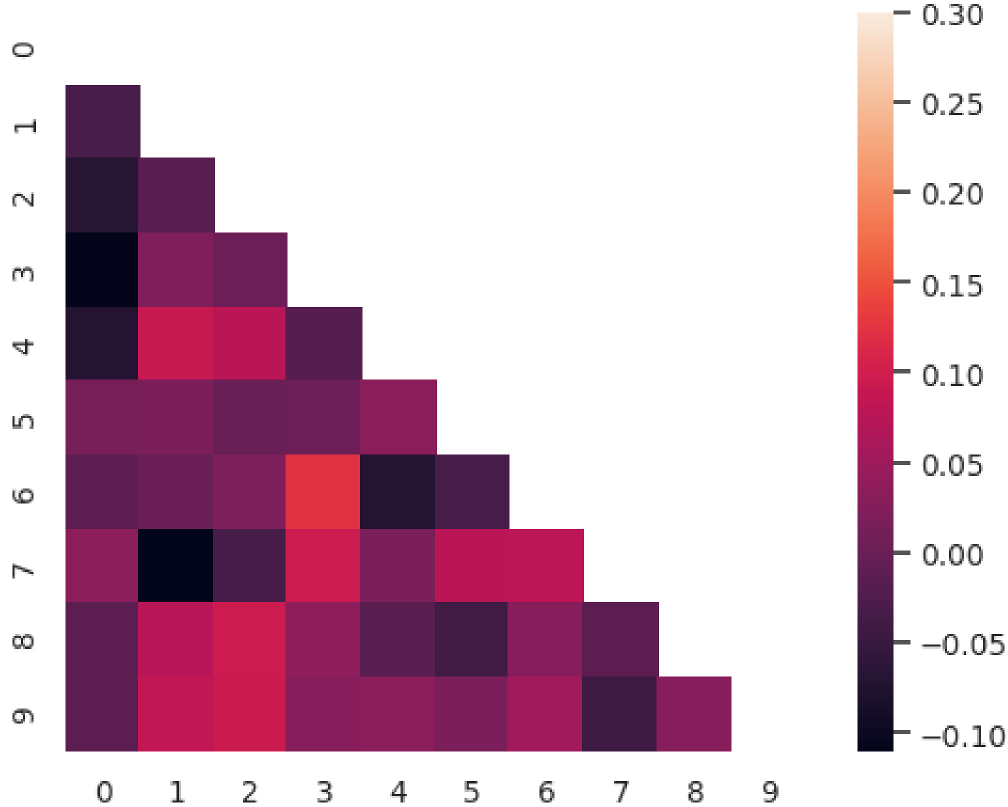
**Parameters**

**data** [2D array-like] Rectangular data for clustering. Cannot contain NAs.

**pivot_kws** [dict, optional] If `data` is a tidy dataframe, can provide keyword arguments for pivot to create a rectangular dataframe.

**method** [str, optional] Linkage method to use for calculating clusters. See `scipy.cluster.hierarchy.linkage()` documentation for more information.

**metric** [str, optional] Distance metric to use for the data. See `scipy.spatial.distance.pdist()` documentation for more options. To use different metrics (or methods) for rows and columns, you may construct each linkage matrix yourself and provide them as `{row,col}_linkage`.




**z_score** [int or None, optional] Either 0 (rows) or 1 (columns). Whether or not to calculate z-scores for the rows or the columns. Z scores are: z = (x - mean)/std, so values in each row (column) will get the mean of the row (column) subtracted, then divided by the standard deviation of the row (column). This ensures that each row (column) has mean of 0 and variance of 1.

**standard_scale** [int or None, optional] Either 0 (rows) or 1 (columns). Whether or not to standardize that dimension, meaning for each row or column, subtract the minimum and divide each by its maximum.

**figsize** [tuple of (width, height), optional] Overall size of the figure.

**cbar_kws** [dict, optional] Keyword arguments to pass to `cbar_kws` in *`heatmap()`*, e.g. to add a label to the colorbar.

**{row,col}_cluster** [bool, optional] If `True`, cluster the {rows, columns}.

**{row,col}_linkage** [`numpy.ndarray`, optional] Precomputed linkage matrix for the rows or columns. See `scipy.cluster.hierarchy.linkage()` for specific formats.

**{row,col}_colors** [list-like or pandas DataFrame/Series, optional] List of colors to label for either the rows or columns. Useful to evaluate whether samples within a group are clustered together. Can use nested lists or DataFrame for multiple color levels of labeling. If given as a `pandas.DataFrame` or `pandas.Series`, labels for the colors are extracted from the DataFrames column names or from the name of the Series. DataFrame/Series colors are also matched to the data by their index, ensuring colors are drawn in the correct order.

**mask** [bool array or DataFrame, optional] If passed, data will not be shown in cells where `mask` is True. Cells with missing values are automatically masked. Only used for visualizing, not for calculating.

**{dendrogram,colors}_ratio** [float, or pair of floats, optional] Proportion of the figure size devoted to the two marginal elements. If a pair is given, they correspond to (row, col) ratios.

**cbar_pos** [tuple of (left, bottom, width, height), optional] Position of the colorbar axes in the figure. Setting to `None` will disable the colorbar.

**tree_kws** [dict, optional] Parameters for the `matplotlib.collections.LineCollection` that is used to plot the lines of the dendrogram tree.

**kwargs** [other keyword arguments] All other keyword arguments are passed to *`heatmap()`*.

**Returns**

**`ClusterGrid`** A `ClusterGrid` instance.

**See also:**

***`heatmap`*** Plot rectangular data as a color-encoded matrix.

### Notes

The returned object has a `savefig` method that should be used if you want to save the figure object without clipping the dendrograms.

To access the reordered row indices, use: `clustergrid.dendrogram_row.reordered_ind`

Column indices, use: `clustergrid.dendrogram_col.reordered_ind`

### Examples

Plot a clustered heatmap:

```
>>> import seaborn as sns; sns.set_theme(color_codes=True)
>>> iris = sns.load_dataset("iris")
>>> species = iris.pop("species")
>>> g = sns.clustermap(iris)
```

Change the size and layout of the figure:

```
>>> g = sns.clustermap(iris,
...                    figsize=(7, 5),
...                    row_cluster=False,
...                    dendrogram_ratio=(.1, .2),
...                    cbar_pos=(0, .2, .03, .4))
```

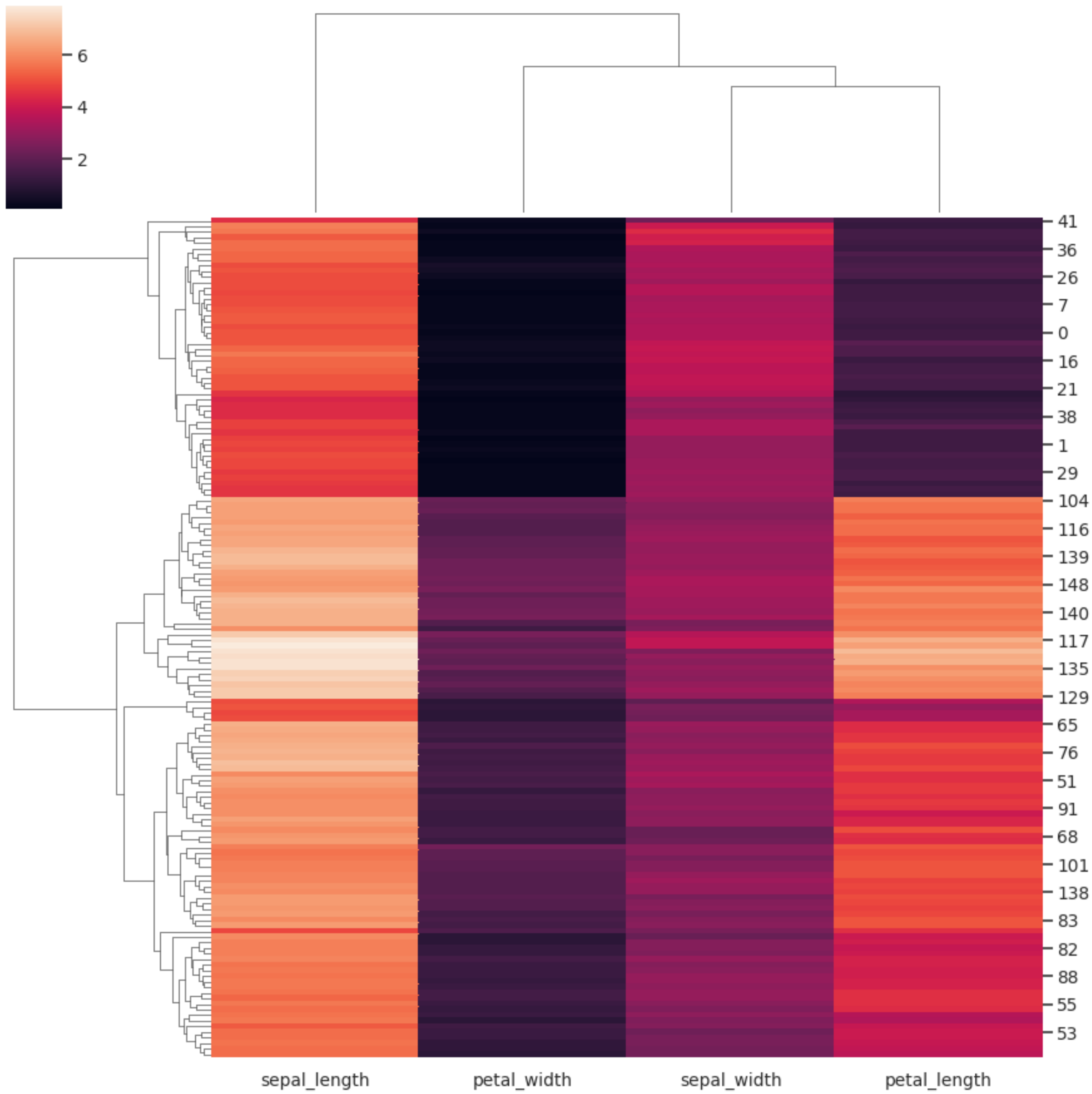
Add colored labels to identify observations:

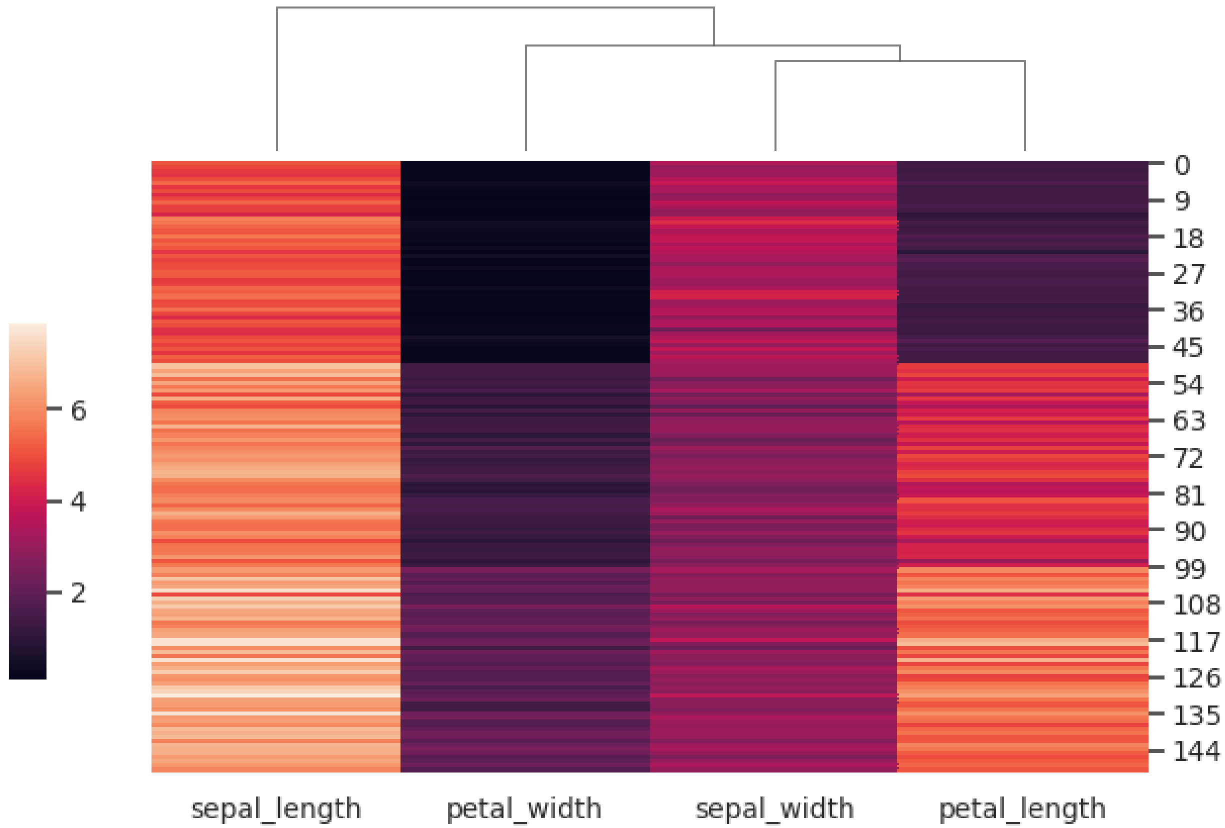
```
>>> lut = dict(zip(species.unique(), "rbg"))
>>> row_colors = species.map(lut)
>>> g = sns.clustermap(iris, row_colors=row_colors)
```

Use a different colormap and adjust the limits of the color range:

```
>>> g = sns.clustermap(iris, cmap="mako", vmin=0, vmax=10)
```

Use a different similarity metric:


6
4
2
41
36
26
7
0
16
21
38
1
29
104
116
139
148
140
117
135
129
65
76
51
91
68
101
138
83
82
88
55
53
sepal_length
petal_width
sepal_width
petal_length




```
>>> g = sns.clustermap(iris, metric="correlation")
```

Use a different clustering method:

```
>>> g = sns.clustermap(iris, method="single")
```

Standardize the data within the columns:

```
>>> g = sns.clustermap(iris, standard_scale=1)
```

Normalize the data within the rows:

```
>>> g = sns.clustermap(iris, z_score=0, cmap="vlag")
```

## 5.6 Multi-plot grids

### 5.6.1 Facet grids

| | |
|---|---|
| *FacetGrid* | Multi-plot grid for plotting conditional relationships. |
| *FacetGrid.map* | Apply a plotting function to each facet's subset of the data. |

continues on next page

Table 6 – continued from previous page

| | |
|---|---|
| *FacetGrid.map_dataframe* | Like `.map` but passes args as strings and inserts data in kwargs. |

### seaborn.FacetGrid

**class** seaborn.**FacetGrid**(***kwargs*)

Multi-plot grid for plotting conditional relationships.

**__init__**(*data*, ***, *row=None*, *col=None*, *hue=None*, *col_wrap=None*, *sharex=True*, *sharey=True*, *height=3*, *aspect=1*, *palette=None*, *row_order=None*, *col_order=None*, *hue_order=None*, *hue_kws=None*, *dropna=False*, *legend_out=True*, *despine=True*, *margin_titles=False*, *xlim=None*, *ylim=None*, *subplot_kws=None*, *gridspec_kws=None*, *size=None*)

Initialize the matplotlib figure and FacetGrid object.

This class maps a dataset onto multiple axes arrayed in a grid of rows and columns that correspond to *levels* of variables in the dataset. The plots it produces are often called "lattice", "trellis", or "small-multiple" graphics.

It can also represent levels of a third variable with the `hue` parameter, which plots different subsets of data in different colors. This uses color to resolve elements on a third dimension, but only draws subsets on top of each other and will not tailor the `hue` parameter for the specific visualization the way that axes-level functions that accept `hue` will.

The basic workflow is to initialize the *FacetGrid* object with the dataset and the variables that are used to structure the grid. Then one or more plotting functions can be applied to each subset by calling *FacetGrid.map()* or *FacetGrid.map_dataframe()*. Finally, the plot can be tweaked with other methods to do things like change the axis labels, use different ticks, or add a legend. See the detailed code examples below for more information.

> **Warning:** When using seaborn functions that infer semantic mappings from a dataset, care must be taken to synchronize those mappings across facets (e.g., by defing the `hue` mapping with a palette dict or setting the data type of the variables to `category`). In most cases, it will be better to use a figure-level function (e.g. *relplot()* or *catplot()*) than to use *FacetGrid* directly.

See the tutorial for more information.

**Parameters**

**data** [DataFrame] Tidy ("long-form") dataframe where each column is a variable and each row is an observation.

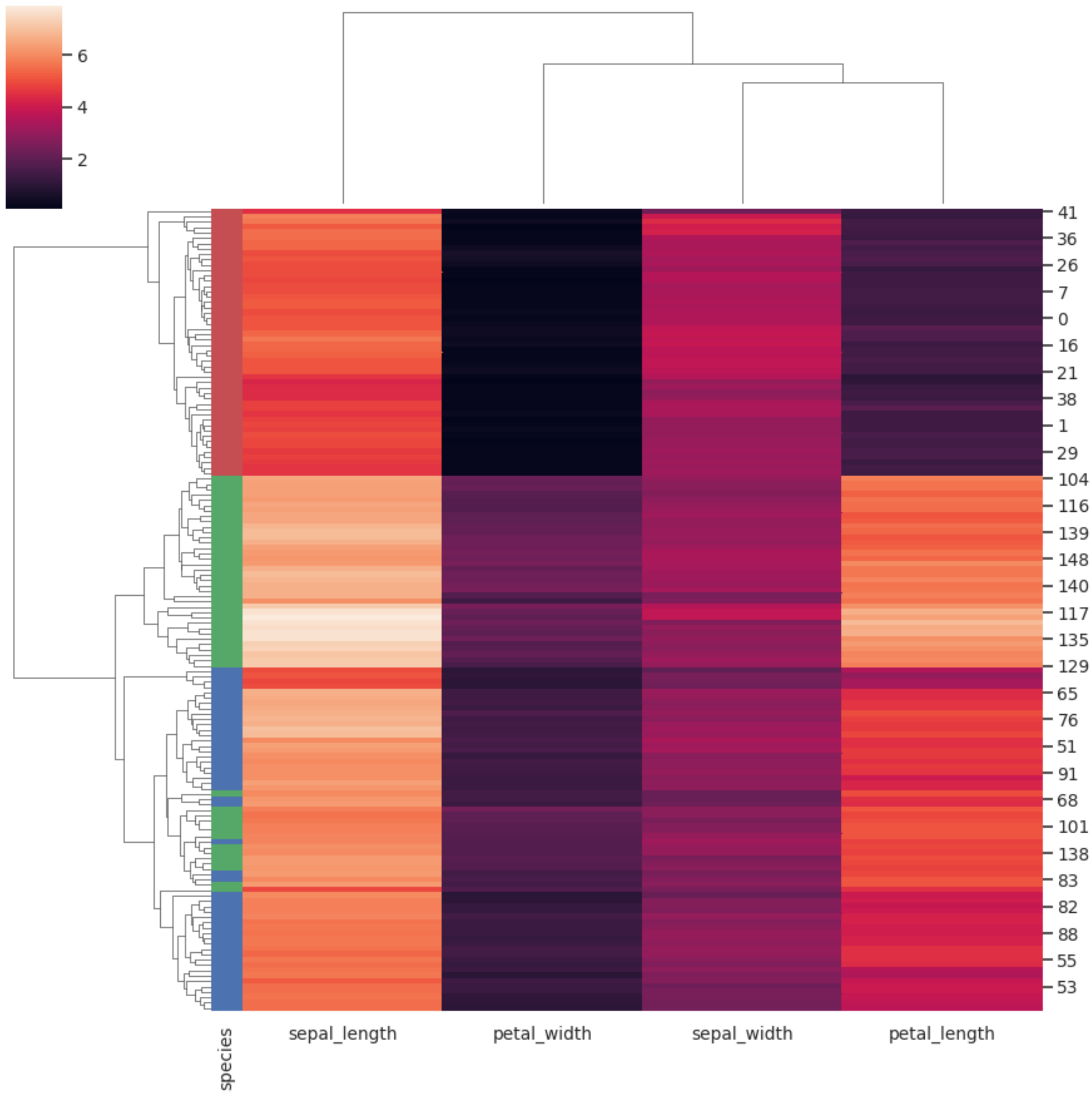
**row, col, hue** [strings] Variables that define subsets of the data, which will be drawn on separate facets in the grid. See the `{var}_order` parameters to control the order of levels of this variable.

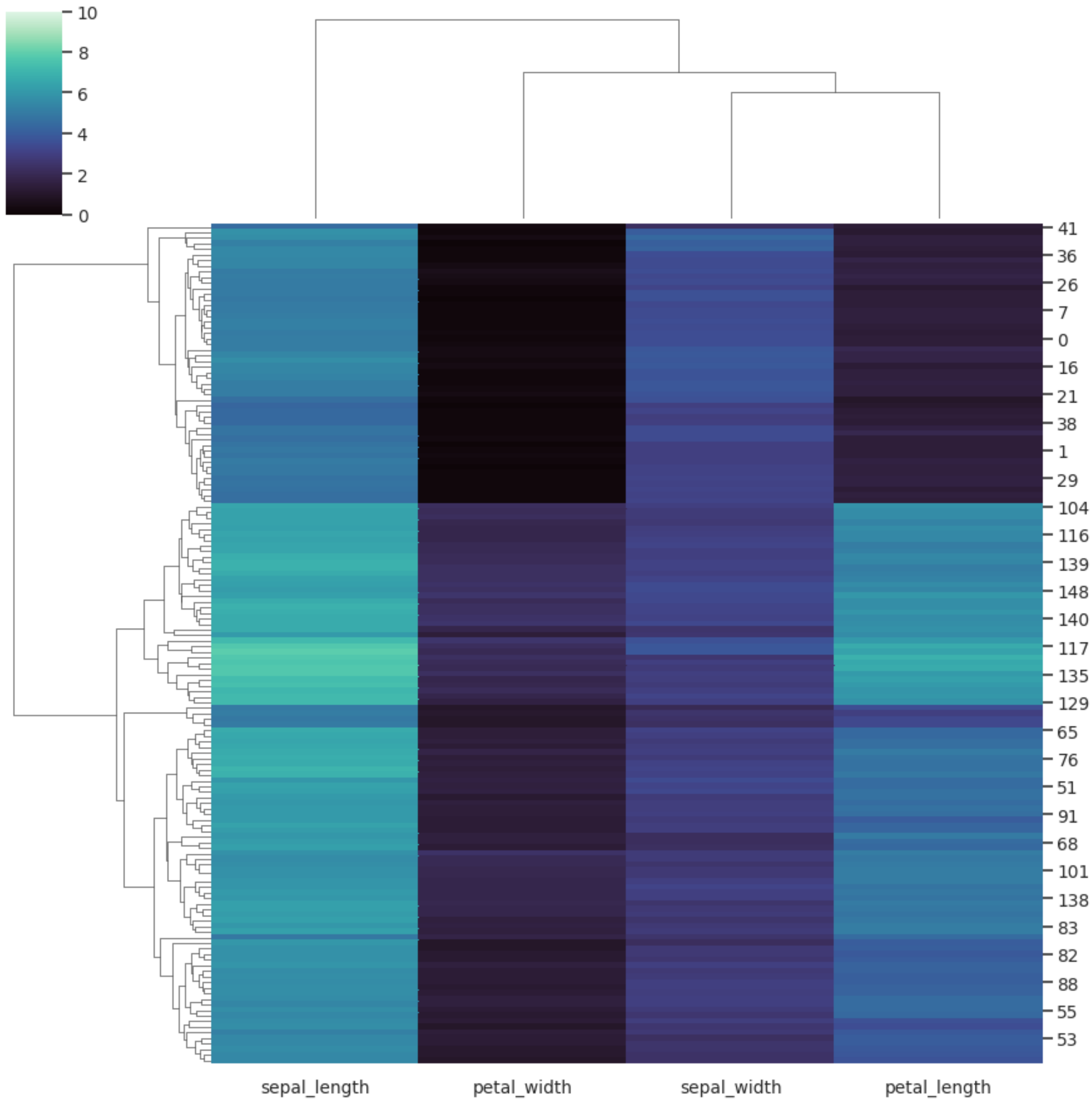
**col_wrap** [int] "Wrap" the column variable at this width, so that the column facets span multiple rows. Incompatible with a `row` facet.

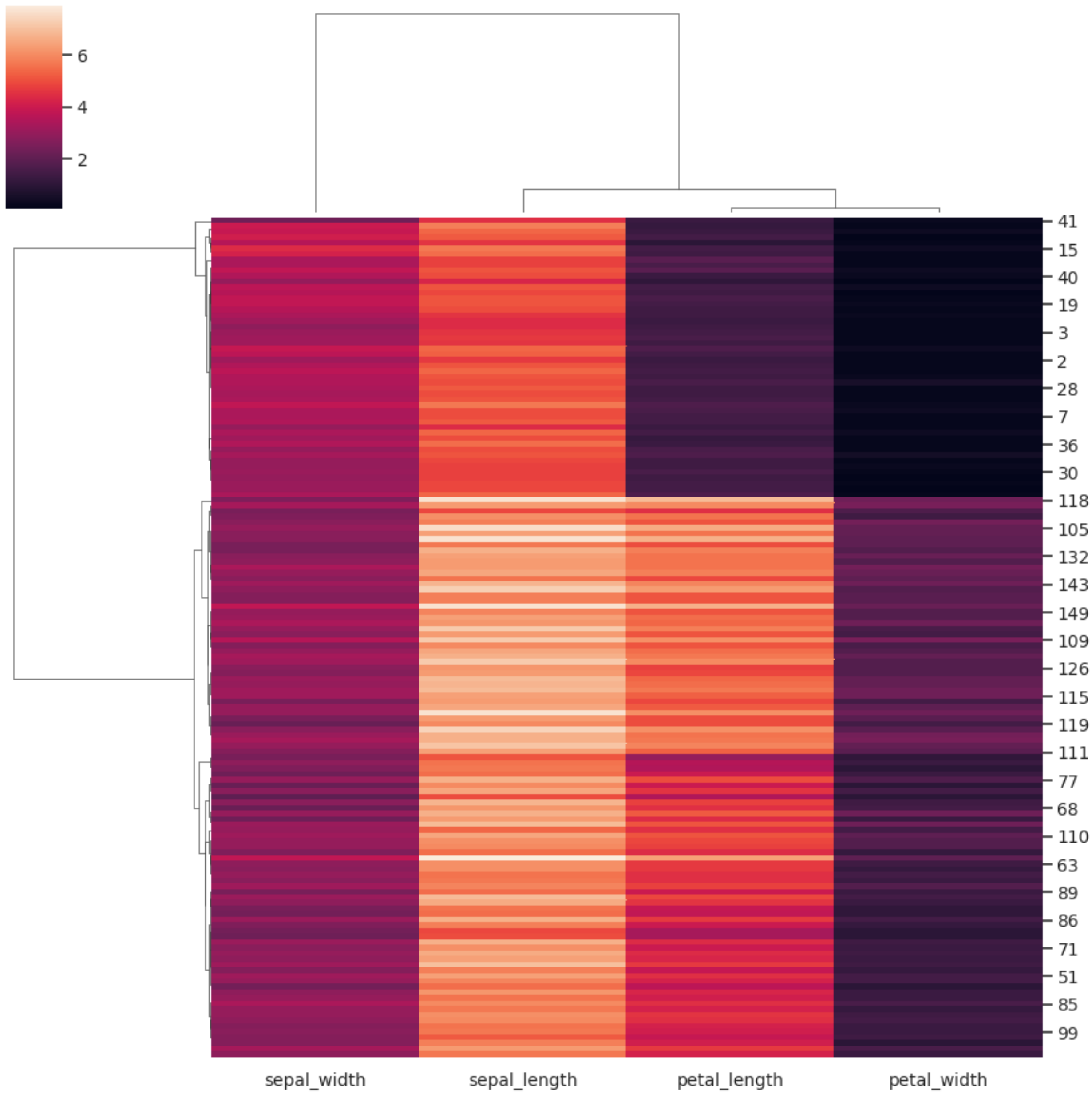
**share{x,y}** [bool, 'col', or 'row' optional] If true, the facets will share y axes across columns and/or x axes across rows.

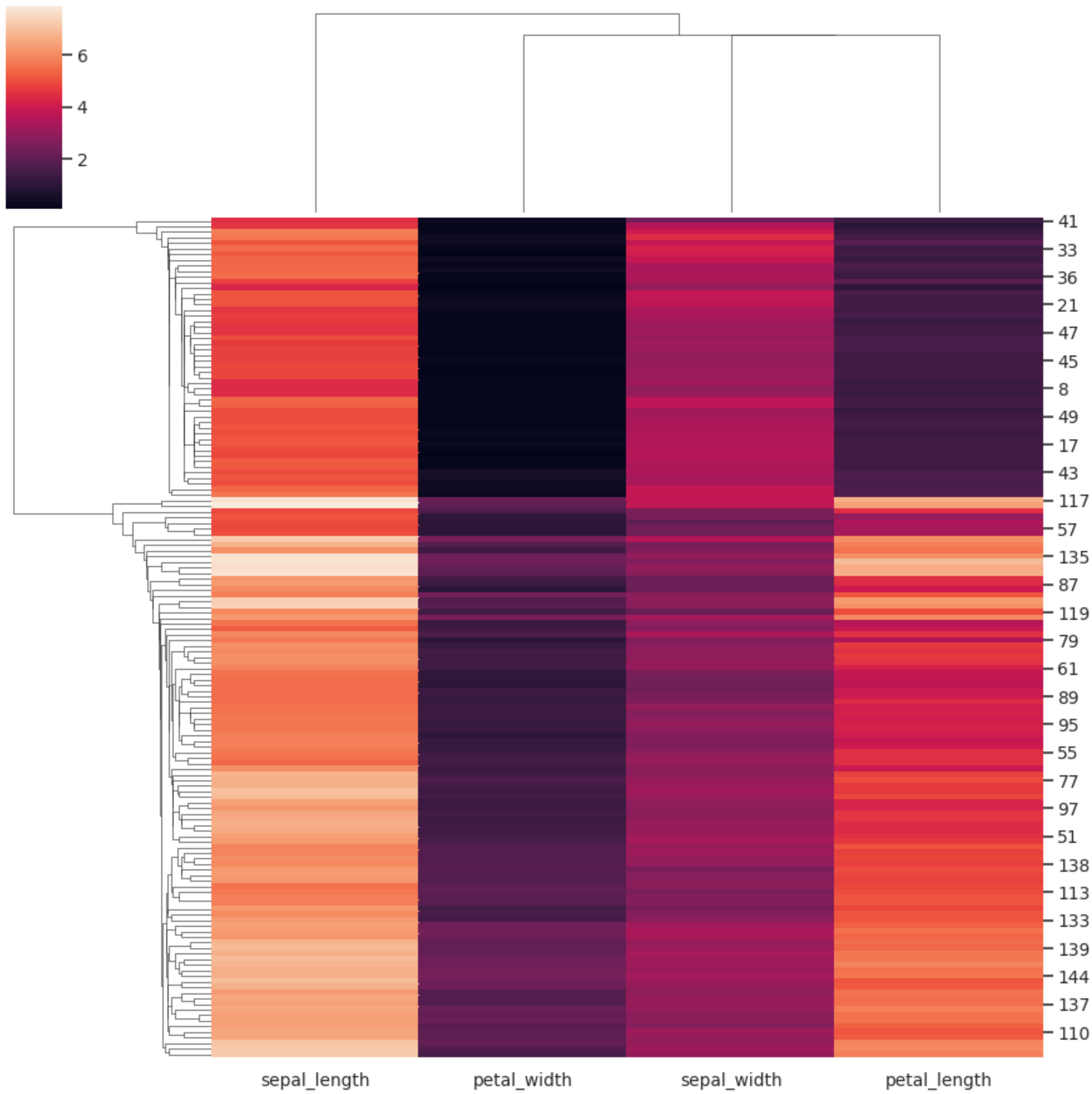
**height** [scalar] Height (in inches) of each facet. See also: `aspect`.

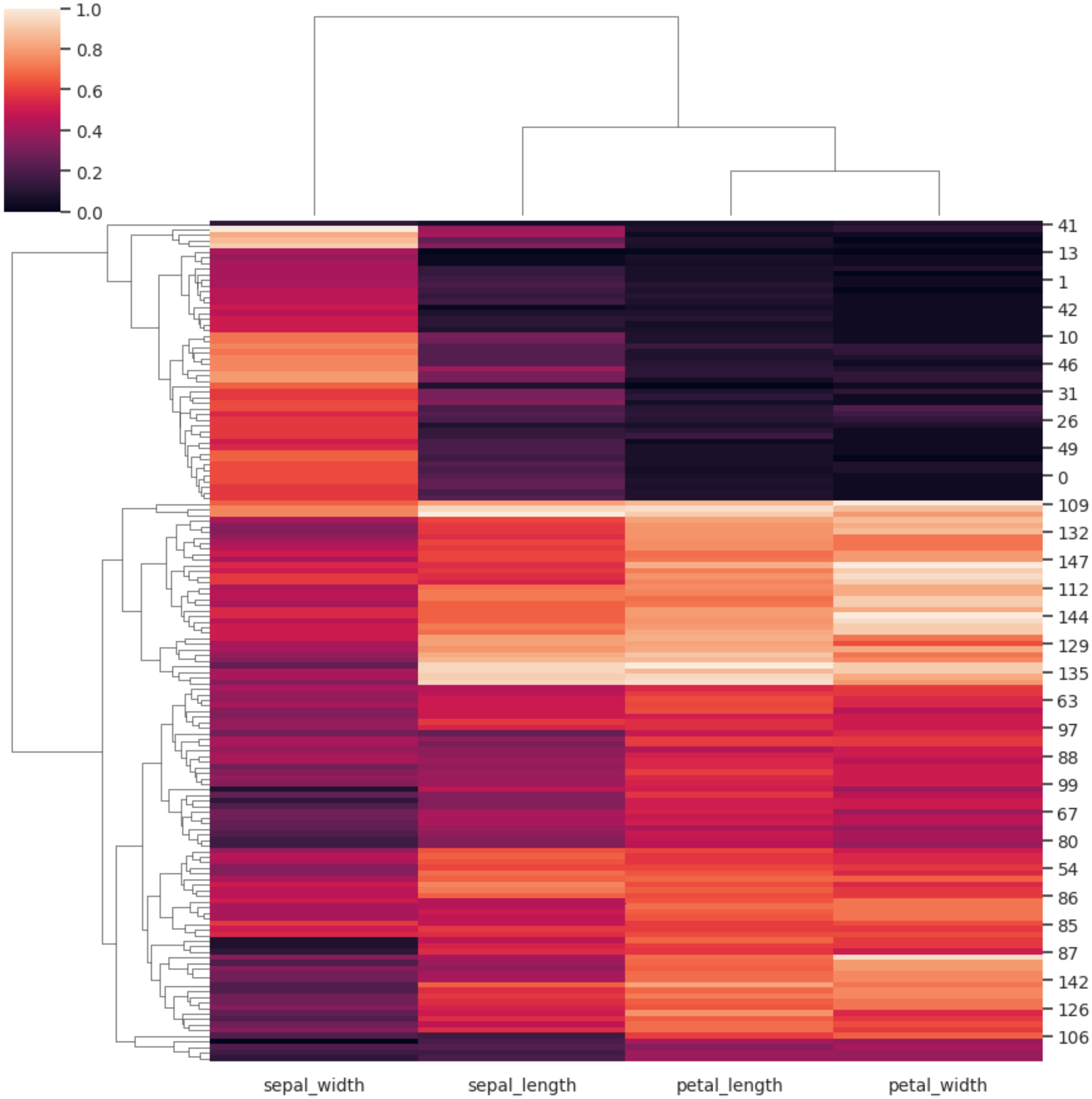
**aspect** [scalar] Aspect ratio of each facet, so that `aspect * height` gives the width of each facet in inches.

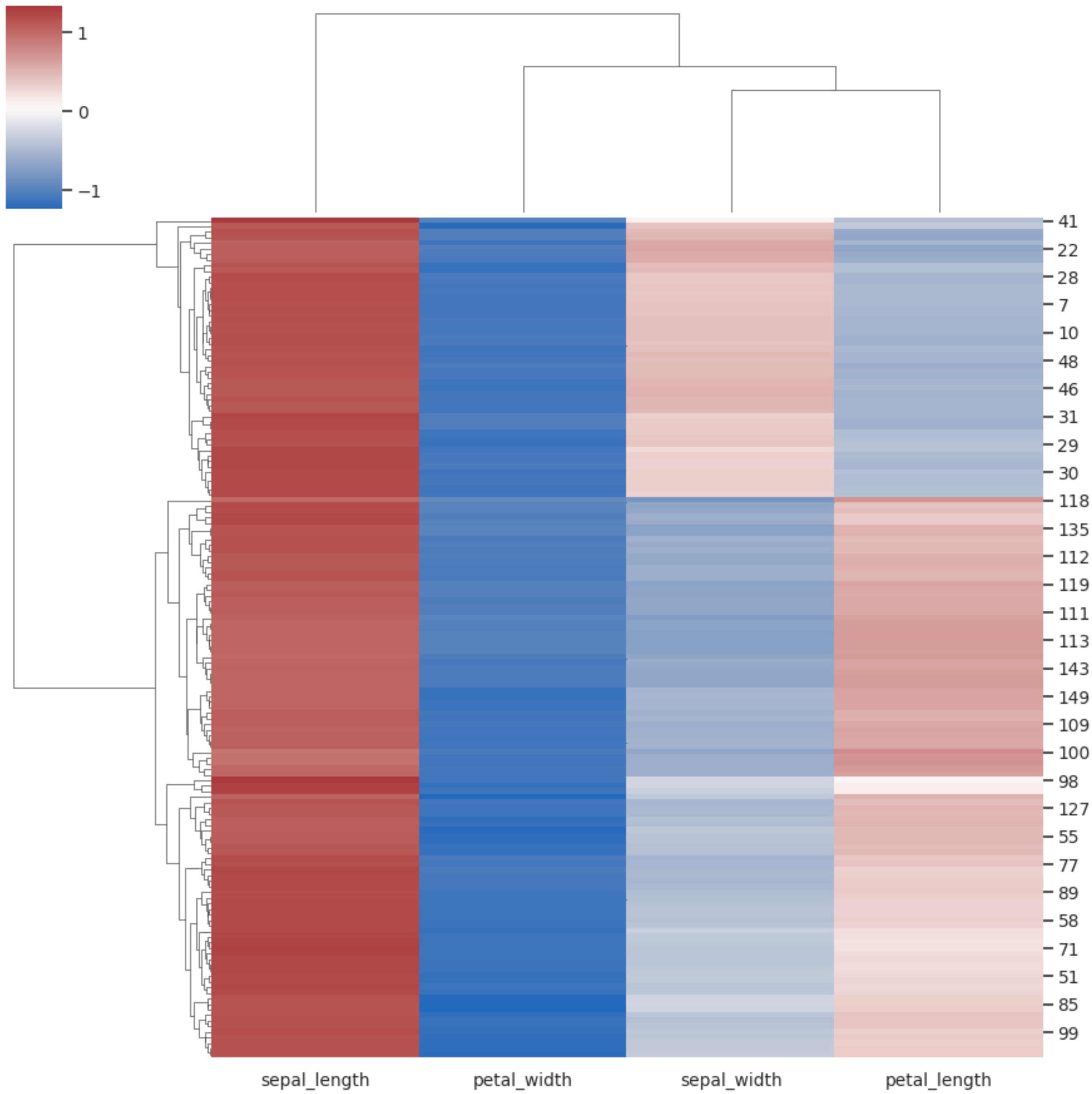

6
4
2
41
36
26
7
0
16
21
38
1
29
104
116
139
148
140
117
135
129
65
76
51
91
68
101
138
83
82
88
55
53
species
sepal_length
petal_width
sepal_width
petal_length


10
8
6
4
2
0
41
36
26
7
0
16
21
38
1
29
104
116
139
148
140
117
135
129
65
76
51
91
68
101
138
83
82
88
55
53
sepal_length
petal_width
sepal_width
petal_length


6
4
2
41
15
40
19
3
2
28
7
36
30
118
105
132
143
149
109
126
115
119
111
77
68
110
63
89
86
71
51
85
99
sepal_width
sepal_length
petal_length
petal_width


6
4
2
41
33
36
21
47
45
8
49
17
43
117
57
135
87
119
79
61
89
95
55
77
97
51
138
113
133
139
144
137
110
sepal_length
petal_width
sepal_width
petal_length


1.0
0.8
0.6
0.4
0.2
0.0
41
13
1
42
10
46
31
26
49
0
109
132
147
112
144
129
135
63
97
88
99
67
80
54
86
85
87
142
126
106
sepal_width
sepal_length
petal_length
petal_width


1
0
−1
41
22
28
7
10
48
46
31
29
30
118
135
112
119
111
113
143
149
109
100
98
127
55
77
89
58
71
51
85
99
sepal_length
petal_width
sepal_width
petal_length

**palette** [palette name, list, or dict] Colors to use for the different levels of the `hue` variable. Should be something that can be interpreted by *color_palette()*, or a dictionary mapping hue levels to matplotlib colors.

**{row,col,hue}_order** [lists] Order for the levels of the faceting variables. By default, this will be the order that the levels appear in `data` or, if the variables are pandas categoricals, the category order.

**hue_kws** [dictionary of param -> list of values mapping] Other keyword arguments to insert into the plotting call to let other plot attributes vary across levels of the hue variable (e.g. the markers in a scatterplot).

**legend_out** [bool] If `True`, the figure size will be extended, and the legend will be drawn outside the plot on the center right.

**despine** [boolean] Remove the top and right spines from the plots.

**margin_titles** [bool] If `True`, the titles for the row variable are drawn to the right of the last column. This option is experimental and may not work in all cases.

**{x, y}lim: tuples** Limits for each of the axes on each facet (only relevant when share{x, y} is True).

**subplot_kws** [dict] Dictionary of keyword arguments passed to matplotlib subplot(s) methods.

**gridspec_kws** [dict] Dictionary of keyword arguments passed to `matplotlib.gridspec.GridSpec` (via `matplotlib.pyplot.subplots()`). Ignored if `col_wrap` is not `None`.

**See also:**

***PairGrid*** Subplot grid for plotting pairwise relationships

***relplot*** Combine a relational plot and a *FacetGrid*

***displot*** Combine a distribution plot and a *FacetGrid*

***catplot*** Combine a categorical plot and a *FacetGrid*

***lmplot*** Combine a regression plot and a *FacetGrid*

### Examples

**Note:** These examples use seaborn functions to demonstrate some of the advanced features of the class, but in most cases you will want to use figue-level functions (e.g. *displot()*, *relplot()*) to make the plots shown here.

### Methods

| | |
|---|---|
| `__init__`(data, *[, row, col, hue, col_wrap, …]) | Initialize the matplotlib figure and FacetGrid object. |
| `add_legend`([legend_data, title, …]) | Draw a legend, maybe placing it outside axes and resizing the figure. |
| `despine`(**kwargs) | Remove axis spines from the facets. |
| `facet_axis`(row_i, col_j[, modify_state]) | Make the axis identified by these indices active and return it. |
| `facet_data`() | Generator for name indices and data subsets for each facet. |
| `map`(func, *args, **kwargs) | Apply a plotting function to each facet's subset of the data. |
| `map_dataframe`(func, *args, **kwargs) | Like `.map` but passes args as strings and inserts data in kwargs. |
| `savefig`(*args, **kwargs) | Save the figure. |
| `set`(**kwargs) | Set attributes on each subplot Axes. |
| `set_axis_labels`([x_var, y_var, clear_inner]) | Set axis labels on the left column and bottom row of the grid. |
| `set_titles`([template, row_template, …]) | Draw titles either above each facet or on the grid margins. |
| `set_xlabels`([label, clear_inner]) | Label the x axis on the bottom row of the grid. |
| `set_xticklabels`([labels, step]) | Set x axis tick labels of the grid. |
| `set_ylabels`([label, clear_inner]) | Label the y axis on the left column of the grid. |
| `set_yticklabels`([labels]) | Set y axis tick labels on the left column of the grid. |
| `tight_layout`(*args, **kwargs) | Call fig.tight_layout within rect that exclude the legend. |

### Attributes

| | |
|---|---|
| `ax` | The `matplotlib.axes.Axes` when no faceting variables are assigned. |
| `axes` | An array of the `matplotlib.axes.Axes` objects in the grid. |
| `axes_dict` | A mapping of facet names to corresponding `matplotlib.axes.Axes`. |
| `fig` | The `matplotlib.figure.Figure` with the plot. |
| `legend` | The `matplotlib.legend.Legend` object, if present. |

### seaborn.FacetGrid.map

`FacetGrid.map(func, *args, **kwargs)`

Apply a plotting function to each facet's subset of the data.

**Parameters**

**func** [callable] A plotting function that takes data and keyword arguments. It must plot to the currently active matplotlib Axes and take a `color` keyword argument. If faceting on the `hue` dimension, it must also take a `label` keyword argument.

**args** [strings] Column names in self.data that identify variables with data to plot. The data for each variable is passed to `func` in the order the variables are specified in the call.

**kwargs** [keyword arguments] All keyword arguments are passed to the plotting function.

**Returns**

**self** [object] Returns self.

### seaborn.FacetGrid.map_dataframe

`FacetGrid.map_dataframe(func, *args, **kwargs)`

Like `.map` but passes args as strings and inserts data in kwargs.

This method is suitable for plotting with functions that accept a long-form DataFrame as a `data` keyword argument and access the data in that DataFrame using string variable names.

**Parameters**

**func** [callable] A plotting function that takes data and keyword arguments. Unlike the `map` method, a function used here must "understand" Pandas objects. It also must plot to the currently active matplotlib Axes and take a `color` keyword argument. If faceting on the `hue` dimension, it must also take a `label` keyword argument.

**args** [strings] Column names in self.data that identify variables with data to plot. The data for each variable is passed to `func` in the order the variables are specified in the call.

**kwargs** [keyword arguments] All keyword arguments are passed to the plotting function.

**Returns**

**self** [object] Returns self.

## 5.6.2 Pair grids

| | |
|---|---|
| *pairplot* | Plot pairwise relationships in a dataset. |
| *PairGrid* | Subplot grid for plotting pairwise relationships in a dataset. |
| *PairGrid.map* | Plot with the same function in every subplot. |
| *PairGrid.map_diag* | Plot with a univariate function on each diagonal subplot. |
| *PairGrid.map_offdiag* | Plot with a bivariate function on the off-diagonal subplots. |
| *PairGrid.map_lower* | Plot with a bivariate function on the lower diagonal subplots. |
| *PairGrid.map_upper* | Plot with a bivariate function on the upper diagonal subplots. |

### 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.

**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.

**See also:**

**`PairGrid`** Subplot grid for more flexible plotting of pairwise relationships.

`JointGrid` Grid for plotting joint and marginal distributions of two variables.

### Examples

## seaborn.PairGrid

**class** seaborn.**PairGrid**(***kwargs*)

Subplot grid for plotting pairwise relationships in a dataset.

This object maps each variable in a dataset onto a column and row in a grid of multiple axes. Different axes-level plotting functions can be used to draw bivariate plots in the upper and lower triangles, and the the marginal distribution of each variable can be shown on the diagonal.

Several different common plots can be generated in a single line using `pairplot()`. Use `PairGrid` when you need more flexibility.

See the tutorial for more information.

**__init__**(*data*, ***, *hue=None*, *hue_order=None*, *palette=None*, *hue_kws=None*, *vars=None*, *x_vars=None*, *y_vars=None*, *corner=False*, *diag_sharey=True*, *height=2.5*, *aspect=1*, *layout_pad=0.5*, *despine=True*, *dropna=False*, *size=None*)

Initialize the plot figure and PairGrid object.

**Parameters**

**data** [DataFrame] Tidy (long-form) dataframe where each column is a variable and each row is an observation.

**hue** [string (variable name)] Variable in `data` to map plot aspects to different colors. This variable will be excluded from the default x and y variables.

**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.

**hue_kws** [dictionary of param -> list of values mapping] Other keyword arguments to insert into the plotting call to let other plot attributes vary across levels of the hue variable (e.g. the markers in a scatterplot).

**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.

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

**height** [scalar] Height (in inches) of each facet.

**aspect** [scalar] Aspect * height gives the width (in inches) of each facet.

**layout_pad** [scalar] Padding between axes; passed to `fig.tight_layout`.

**despine** [boolean] Remove the top and right spines from the plots.

**dropna** [boolean] Drop missing values from the data before plotting.

**See also:**

`pairplot` Easily drawing common uses of `PairGrid`.

*FacetGrid* Subplot grid for plotting conditional relationships.

#### Examples

### Methods

| | |
|---|---|
| `__init__`(data, *[, hue, hue_order, palette, . . . ]) | Initialize the plot figure and PairGrid object. |
| `add_legend`([legend_data, title, . . . ]) | Draw a legend, maybe placing it outside axes and resizing the figure. |
| `map`(func, **kwargs) | Plot with the same function in every subplot. |
| `map_diag`(func, **kwargs) | Plot with a univariate function on each diagonal subplot. |
| `map_lower`(func, **kwargs) | Plot with a bivariate function on the lower diagonal subplots. |
| `map_offdiag`(func, **kwargs) | Plot with a bivariate function on the off-diagonal subplots. |
| `map_upper`(func, **kwargs) | Plot with a bivariate function on the upper diagonal subplots. |
| `savefig`(*args, **kwargs) | Save the figure. |
| `set`(**kwargs) | Set attributes on each subplot Axes. |
| `tight_layout`(*args, **kwargs) | Call fig.tight_layout within rect that exclude the legend. |

### Attributes

| | |
|---|---|
| `legend` | The `matplotlib.legend.Legend` object, if present. |

## seaborn.PairGrid.map

`PairGrid.`**map** (*func*, ***kwargs*)

Plot with the same function in every subplot.

**Parameters**

**func** [callable plotting function] Must take x, y arrays as positional arguments and draw onto the "currently active" matplotlib Axes. Also needs to accept kwargs called `color` and `label`.

## seaborn.PairGrid.map_diag

`PairGrid.`**map_diag** (*func*, ***kwargs*)

Plot with a univariate function on each diagonal subplot.

**Parameters**

**func** [callable plotting function] Must take an x array as a positional argument and draw onto the "currently active" matplotlib Axes. Also needs to accept kwargs called `color` and `label`.

#### seaborn.PairGrid.map_offdiag

PairGrid.**map_offdiag**(*func*, ***kwargs*)

Plot with a bivariate function on the off-diagonal subplots.

**Parameters**

**func** [callable plotting function] Must take x, y arrays as positional arguments and draw onto the "currently active" matplotlib Axes. Also needs to accept kwargs called `color` and `label`.

#### seaborn.PairGrid.map_lower

PairGrid.**map_lower**(*func*, ***kwargs*)

Plot with a bivariate function on the lower diagonal subplots.

**Parameters**

**func** [callable plotting function] Must take x, y arrays as positional arguments and draw onto the "currently active" matplotlib Axes. Also needs to accept kwargs called `color` and `label`.

#### seaborn.PairGrid.map_upper

PairGrid.**map_upper**(*func*, ***kwargs*)

Plot with a bivariate function on the upper diagonal subplots.

**Parameters**

**func** [callable plotting function] Must take x, y arrays as positional arguments and draw onto the "currently active" matplotlib Axes. Also needs to accept kwargs called `color` and `label`.

### 5.6.3 Joint grids

| | |
|---|---|
| `jointplot` | Draw a plot of two variables with bivariate and univariate graphs. |
| `JointGrid` | Grid for drawing a bivariate plot with marginal univariate plots. |
| `JointGrid.plot` | Draw the plot by passing functions for joint and marginal axes. |
| `JointGrid.plot_joint` | Draw a bivariate plot on the joint axes of the grid. |
| `JointGrid.plot_marginals` | Draw univariate plots on each marginal axes. |

#### seaborn.jointplot

seaborn.**jointplot**(**, x=None, y=None, data=None, kind='scatter', color=None, height=6, ratio=5, space=0.2, dropna=False, xlim=None, ylim=None, marginal_ticks=False, joint_kws=None, marginal_kws=None, hue=None, palette=None, hue_order=None, hue_norm=None, **kwargs*)

Draw a plot of two variables with bivariate and univariate graphs.

This function provides a convenient interface to the `JointGrid` class, with several canned plot kinds. This is intended to be a fairly lightweight wrapper; if you need more flexibility, you should use `JointGrid` directly.

**Parameters**

**x, y** [vectors or keys in `data`] Variables that specify positions on the x and y axes.

**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.

**kind** [{ "scatter" | "kde" | "hist" | "hex" | "reg" | "resid" }] Kind of plot to draw. See the examples for references to the underlying functions.

**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.

**height** [numeric] Size of the figure (it will be square).

**ratio** [numeric] Ratio of joint axes height to marginal axes height.

**space** [numeric] Space between the joint and marginal axes

**dropna** [bool] If True, remove observations that are missing from `x` and `y`.

**{x, y}lim** [pairs of numbers] Axis limits to set before plotting.

**marginal_ticks** [bool] If False, suppress ticks on the count/density axis of the marginal plots.

**{joint, marginal}_kws** [dicts] Additional keyword arguments for the plot components.

**hue** [vector or key in `data`] Semantic variable that is mapped to determine the color of plot elements. Semantic variable that is mapped to determine the color of plot elements.

**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.

**kwargs** Additional keyword arguments are passed to the function used to draw the plot on the joint Axes, superseding items in the `joint_kws` dictionary.

**Returns**

***`JointGrid`*** An object managing multiple subplots that correspond to joint and marginal axes for plotting a bivariate relationship or distribution.

**See also:**

***`JointGrid`*** Set up a figure with joint and marginal views on bivariate data.

***`PairGrid`*** Set up a figure with joint and marginal views on multiple variables.

***`jointplot`*** Draw multiple bivariate plots with univariate marginal distributions.

### Examples

## seaborn.JointGrid

**class** seaborn.**JointGrid**(***kwargs*)

Grid for drawing a bivariate plot with marginal univariate plots.

Many plots can be drawn by using the figure-level interface `jointplot()`. Use this class directly when you need more flexibility.

**__init__**(**, x=None, y=None, data=None, height=6, ratio=5, space=0.2, dropna=False, xlim=None, ylim=None, size=None, marginal_ticks=False, hue=None, palette=None, hue_order=None, hue_norm=None*)

Set up the grid of subplots and store data internally for easy plotting.

**Parameters**

**x, y** [vectors or keys in `data`] Variables that specify positions on the x and y axes.

**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.

**height** [number] Size of each side of the figure in inches (it will be square).

**ratio** [number] Ratio of joint axes height to marginal axes height.

**space** [number] Space between the joint and marginal axes

**dropna** [bool] If True, remove missing observations before plotting.

**{x, y}lim** [pairs of numbers] Set axis limits to these values before plotting.

**marginal_ticks** [bool] If False, suppress ticks on the count/density axis of the marginal plots.

**hue** [vector or key in `data`] Semantic variable that is mapped to determine the color of plot elements. Note: unlike in `FacetGrid` or `PairGrid`, the axes-level functions must support `hue` to use it in `JointGrid`.

**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.

**See also:**

**`jointplot`** Draw a bivariate plot with univariate marginal distributions.

**`PairGrid`** Set up a figure with joint and marginal views on multiple variables.

**`jointplot`** Draw multiple bivariate plots with univariate marginal distributions.

### Examples

### Methods

| | |
|---|---|
| `__init__`(*[, x, y, data, height, ratio, ...]) | Set up the grid of subplots and store data internally for easy plotting. |
| `plot`(joint_func, marginal_func, **kwargs) | Draw the plot by passing functions for joint and marginal axes. |
| `plot_joint`(func, **kwargs) | Draw a bivariate plot on the joint axes of the grid. |
| `plot_marginals`(func, **kwargs) | Draw univariate plots on each marginal axes. |
| `savefig`(*args, **kwargs) | Save the figure using a "tight" bounding box by default. |
| `set_axis_labels`([xlabel, ylabel]) | Set axis labels on the bivariate axes. |

## seaborn.JointGrid.plot

`JointGrid.`**`plot`** (*joint_func*, *marginal_func*, ***kwargs*)

Draw the plot by passing functions for joint and marginal axes.

This method passes the `kwargs` dictionary to both functions. If you need more control, call `JointGrid.plot_joint()` and `JointGrid.plot_marginals()` directly with specific parameters.

**Parameters**

**joint_func, marginal_func: callables** Functions to draw the bivariate and univariate plots. See methods referenced above for information about the required characteristics of these functions.

**kwargs** Additional keyword arguments are passed to both functions.

**Returns**

**`JointGrid` instance** Returns `self` for easy method chaining.

## seaborn.JointGrid.plot_joint

`JointGrid.`**`plot_joint`** (*func*, ***kwargs*)

Draw a bivariate plot on the joint axes of the grid.

**Parameters**

**func** [plotting callable] If a seaborn function, it should accept `x` and `y`. Otherwise, it must accept `x` and `y` vectors of data as the first two positional arguments, and it must plot on the "current" axes. If `hue` was defined in the class constructor, the function must accept `hue` as a parameter.

**kwargs** Keyword argument are passed to the plotting function.

**Returns**

**`JointGrid` instance** Returns `self` for easy method chaining.

#### seaborn.JointGrid.plot_marginals

`JointGrid.plot_marginals(func, **kwargs)`

Draw univariate plots on each marginal axes.

**Parameters**

**func** [plotting callable] If a seaborn function, it should accept `x` and `y` and plot when only one of them is defined. Otherwise, it must accept a vector of data as the first positional argument and determine its orientation using the `vertical` parameter, and it must plot on the "current" axes. If `hue` was defined in the class constructor, it must accept `hue` as a parameter.

**kwargs** Keyword argument are passed to the plotting function.

**Returns**

***JointGrid* instance** Returns `self` for easy method chaining.

## 5.7 Themeing

| | |
|---|---|
| *set_theme* | Set multiple theme parameters in one step. |
| *axes_style* | Return a parameter dict for the aesthetic style of the plots. |
| *set_style* | Set the aesthetic style of the plots. |
| *plotting_context* | Return a parameter dict to scale elements of the figure. |
| *set_context* | Set the plotting context parameters. |
| *set_color_codes* | Change how matplotlib color shorthands are interpreted. |
| *reset_defaults* | Restore all RC params to default settings. |
| *reset_orig* | Restore all RC params to original settings (respects custom rc). |
| *set* | Alias for *set_theme()*, which is the preferred interface. |

### 5.7.1 seaborn.set_theme

`seaborn.set_theme(context='notebook', style='darkgrid', palette='deep', font='sans-serif', font_scale=1, color_codes=True, rc=None)`

Set multiple theme parameters in one step.

Each set of parameters can be set directly or temporarily, see the referenced functions below for more information.

**Parameters**

**context** [string or dict] Plotting context parameters, see *plotting_context()*.

**style** [string or dict] Axes style parameters, see *axes_style()*.

**palette** [string or sequence] Color palette, see *color_palette()*.

**font** [string] Font family, see matplotlib font manager.

**font_scale** [float, optional] Separate scaling factor to independently scale the size of the font elements.

**color_codes** [bool] If `True` and `palette` is a seaborn palette, remap the shorthand color codes (e.g. "b", "g", "r", etc.) to the colors from this palette.

**rc** [dict or None] Dictionary of rc parameter mappings to override the above.

### 5.7.2 seaborn.axes_style

`seaborn.`**`axes_style`**(*style=None*, *rc=None*)

Return a parameter dict for the aesthetic style of the plots.

This affects things like the color of the axes, whether a grid is enabled by default, and other aesthetic elements.

This function returns an object that can be used in a `with` statement to temporarily change the style parameters.

**Parameters**

**style** [dict, None, or one of {darkgrid, whitegrid, dark, white, ticks}] A dictionary of parameters or the name of a preconfigured set.

**rc** [dict, optional] Parameter mappings to override the values in the preset seaborn style dictionaries. This only updates parameters that are considered part of the style definition.

**See also:**

***set_style*** set the matplotlib parameters for a seaborn theme

***plotting_context*** return a parameter dict to to scale plot elements

***color_palette*** define the color palette for a plot

#### Examples

```
>>> st = axes_style("whitegrid")
```

```
>>> set_style("ticks", {"xtick.major.size": 8, "ytick.major.size": 8})
```

```
>>> import matplotlib.pyplot as plt
>>> with axes_style("white"):
...     f, ax = plt.subplots()
...     ax.plot(x, y)
```

### 5.7.3 seaborn.set_style

`seaborn.`**`set_style`**(*style=None*, *rc=None*)

Set the aesthetic style of the plots.

This affects things like the color of the axes, whether a grid is enabled by default, and other aesthetic elements.

**Parameters**

**style** [dict, None, or one of {darkgrid, whitegrid, dark, white, ticks}] A dictionary of parameters or the name of a preconfigured set.

**rc** [dict, optional] Parameter mappings to override the values in the preset seaborn style dictionaries. This only updates parameters that are considered part of the style definition.

**See also:**

***axes_style*** return a dict of parameters or use in a `with` statement to temporarily set the style.

***set_context*** set parameters to scale plot elements

***set_palette*** set the default color palette for figures

### Examples

```
>>> set_style("whitegrid")
```

```
>>> set_style("ticks", {"xtick.major.size": 8, "ytick.major.size": 8})
```

## 5.7.4 seaborn.plotting_context

seaborn.**plotting_context**(*context=None*, *font_scale=1*, *rc=None*)

Return a parameter dict to scale elements of the figure.

This affects things like the size of the labels, lines, and other elements of the plot, but not the overall style. The base context is "notebook", and the other contexts are "paper", "talk", and "poster", which are version of the notebook parameters scaled by .8, 1.3, and 1.6, respectively.

This function returns an object that can be used in a `with` statement to temporarily change the context parameters.

**Parameters**

**context** [dict, None, or one of {paper, notebook, talk, poster}] A dictionary of parameters or the name of a preconfigured set.

**font_scale** [float, optional] Separate scaling factor to independently scale the size of the font elements.

**rc** [dict, optional] Parameter mappings to override the values in the preset seaborn context dictionaries. This only updates parameters that are considered part of the context definition.

**See also:**

***set_context*** set the matplotlib parameters to scale plot elements

***axes_style*** return a dict of parameters defining a figure style

***color_palette*** define the color palette for a plot

### Examples

```
>>> c = plotting_context("poster")
```

```
>>> c = plotting_context("notebook", font_scale=1.5)
```

```
>>> c = plotting_context("talk", rc={"lines.linewidth": 2})
```

```
>>> import matplotlib.pyplot as plt
>>> with plotting_context("paper"):
...     f, ax = plt.subplots()
...     ax.plot(x, y)
```

### 5.7.5 seaborn.set_context

`seaborn.set_context(context=None, font_scale=1, rc=None)`

Set the plotting context parameters.

This affects things like the size of the labels, lines, and other elements of the plot, but not the overall style. The base context is "notebook", and the other contexts are "paper", "talk", and "poster", which are version of the notebook parameters scaled by .8, 1.3, and 1.6, respectively.

**Parameters**

**context** [dict, None, or one of {paper, notebook, talk, poster}] A dictionary of parameters or the name of a preconfigured set.

**font_scale** [float, optional] Separate scaling factor to independently scale the size of the font elements.

**rc** [dict, optional] Parameter mappings to override the values in the preset seaborn context dictionaries. This only updates parameters that are considered part of the context definition.

**See also:**

***plotting_context*** return a dictionary of rc parameters, or use in a `with` statement to temporarily set the context.

***set_style*** set the default parameters for figure style

***set_palette*** set the default color palette for figures

#### Examples

```
>>> set_context("paper")
```

```
>>> set_context("talk", font_scale=1.4)
```

```
>>> set_context("talk", rc={"lines.linewidth": 2})
```

### 5.7.6 seaborn.set_color_codes

`seaborn.set_color_codes(palette='deep')`

Change how matplotlib color shorthands are interpreted.

Calling this will change how shorthand codes like "b" or "g" are interpreted by matplotlib in subsequent plots.

**Parameters**

**palette** [{deep, muted, pastel, dark, bright, colorblind}] Named seaborn palette to use as the source of colors.

**See also:**

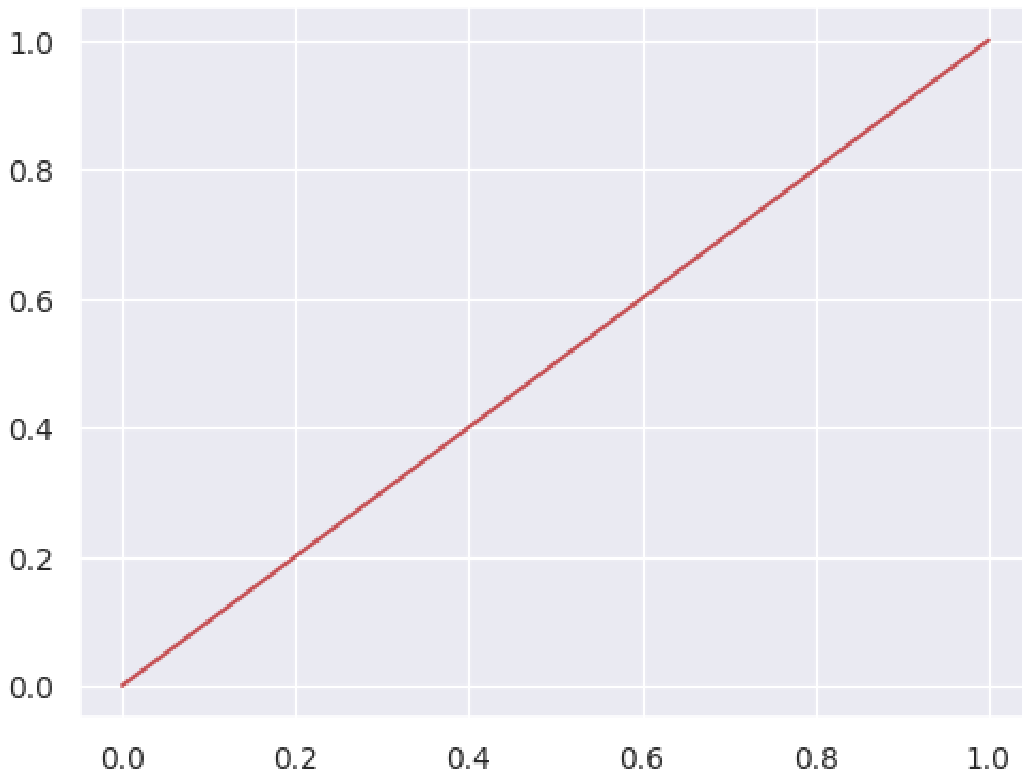
***set*** Color codes can be set through the high-level seaborn style manager.

***set_palette*** Color codes can also be set through the function that sets the matplotlib color cycle.

### Examples

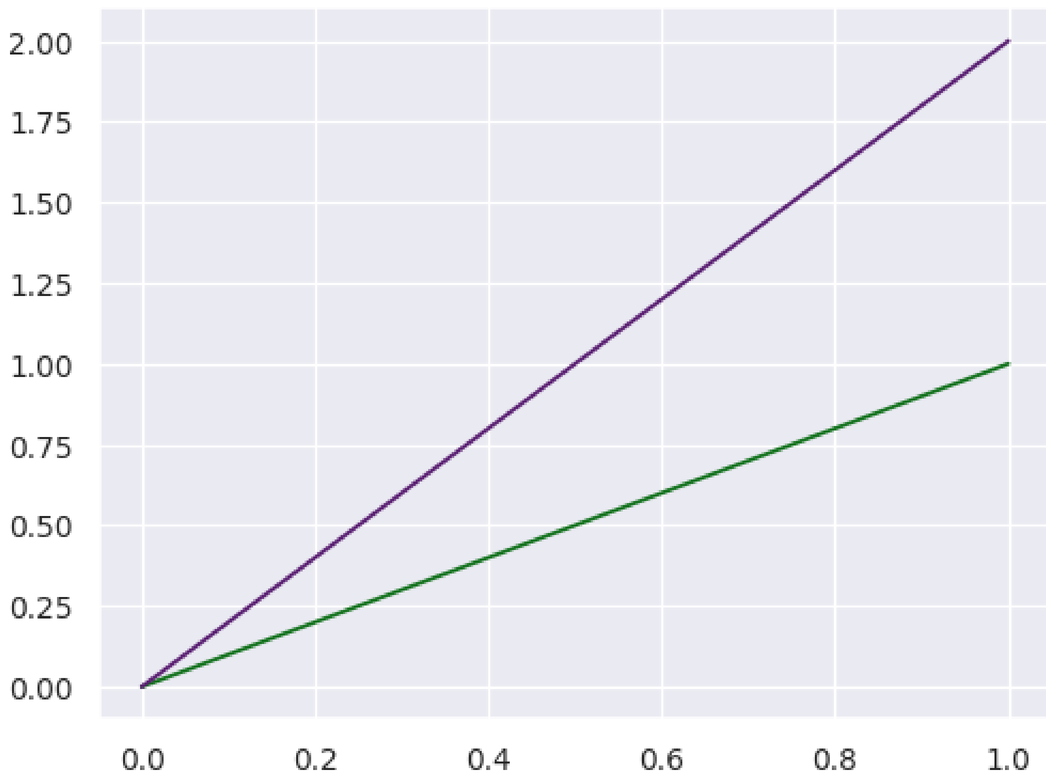
Map matplotlib color codes to the default seaborn palette.

```
>>> import matplotlib.pyplot as plt
>>> import seaborn as sns; sns.set_theme()
>>> sns.set_color_codes()
>>> _ = plt.plot([0, 1], color="r")
```




Use a different seaborn palette.

```
>>> sns.set_color_codes("dark")
>>> _ = plt.plot([0, 1], color="g")
>>> _ = plt.plot([0, 2], color="m")
```




### 5.7.7 seaborn.reset_defaults

`seaborn.reset_defaults()`

Restore all RC params to default settings.

### 5.7.8 seaborn.reset_orig

`seaborn.reset_orig()`

Restore all RC params to original settings (respects custom rc).

### 5.7.9 seaborn.set

`seaborn.set(*args, **kwargs)`

Alias for `set_theme()`, which is the preferred interface.

## 5.8 Color palettes

| | |
|---|---|
| `set_palette` | Set the matplotlib color cycle using a seaborn palette. |
| `color_palette` | Return a list of colors or continuous colormap defining a palette. |
| `husl_palette` | Get a set of evenly spaced colors in HUSL hue space. |
| `hls_palette` | Get a set of evenly spaced colors in HLS hue space. |
| `cubehelix_palette` | Make a sequential palette from the cubehelix system. |
| `dark_palette` | Make a sequential palette that blends from dark to `color`. |
| `light_palette` | Make a sequential palette that blends from light to `color`. |
| `diverging_palette` | Make a diverging palette between two HUSL colors. |
| `blend_palette` | Make a palette that blends between a list of colors. |

**See also:**

*color_palette* build a color palette or set the color cycle temporarily in a `with` statement.

*set_context* set parameters to scale plot elements

*set_style* set the default parameters for figure style

### Examples

```
>>> set_palette("Reds")
```

```
>>> set_palette("Set1", 8, .75)
```

## 5.8.2 seaborn.color_palette

seaborn.**color_palette**(*palette=None, n_colors=None, desat=None, as_cmap=False*)

Return a list of colors or continuous colormap defining a palette.

**Possible `palette` values include:**

- Name of a seaborn palette (deep, muted, bright, pastel, dark, colorblind)
- Name of matplotlib colormap
- 'husl' or 'hls'
- 'ch:<cubehelix arguments>'
- 'light:<color>', 'dark:<color>', 'blend:<color>,<color>',
- A sequence of colors in any format matplotlib accepts

Calling this function with `palette=None` will return the current matplotlib color cycle.

This function can also be used in a `with` statement to temporarily set the color cycle for a plot or set of plots.

See the tutorial for more information.

**Parameters**

**palette: None, string, or sequence, optional** Name of palette or None to return current palette. If a sequence, input colors are used but possibly cycled and desaturated.

**n_colors** [int, optional] Number of colors in the palette. If `None`, the default will depend on how `palette` is specified. Named palettes default to 6 colors, but grabbing the current palette or passing in a list of colors will not change the number of colors unless this is specified. Asking for more colors than exist in the palette will cause it to cycle. Ignored when `as_cmap` is True.

**desat** [float, optional] Proportion to desaturate each color by.

**as_cmap** [bool] If True, return a `matplotlib.colors.Colormap`.

**Returns**

**list of RGB tuples or `matplotlib.colors.Colormap`**

**See also:**

*set_palette* Set the default color cycle for all plots.

*set_color_codes* Reassign color codes like "b", "g", etc. to colors from one of the seaborn palettes.

#### Examples

### 5.8.3 seaborn.husl_palette

seaborn.**husl_palette**(*n_colors=6*, *h=0.01*, *s=0.9*, *l=0.65*, *as_cmap=False*)

Get a set of evenly spaced colors in HUSL hue space.

h, s, and l should be between 0 and 1

**Parameters**

**n_colors** [int] number of colors in the palette

**h** [float] first hue

**s** [float] saturation

**l** [float] lightness

**Returns**

**list of RGB tuples or** matplotlib.colors.Colormap

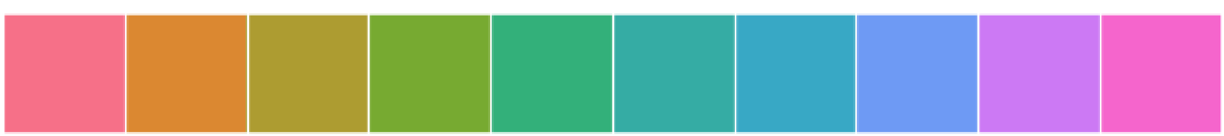
**See also:**

*hls_palette* Make a palette using evently spaced circular hues in the HSL system.

#### Examples

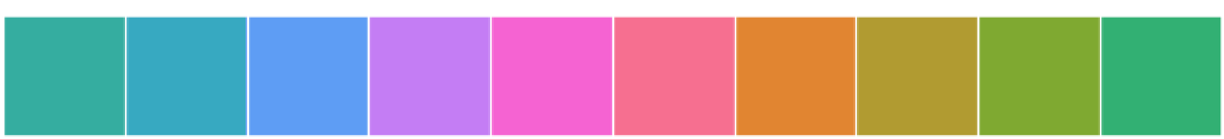
Create a palette of 10 colors with the default parameters:

```
>>> import seaborn as sns; sns.set_theme()
>>> sns.palplot(sns.husl_palette(10))
```



Create a palette of 10 colors that begins at a different hue value:

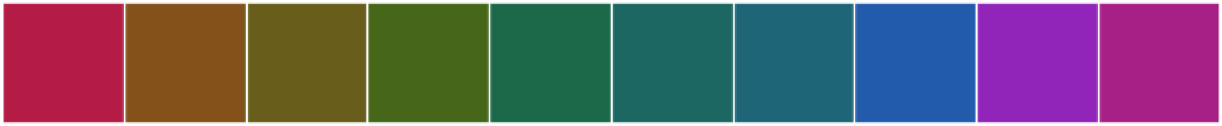
```
>>> sns.palplot(sns.husl_palette(10, h=.5))
```



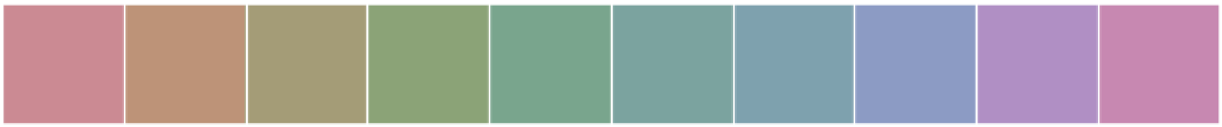
Create a palette of 10 colors that are darker than the default:

```
>>> sns.palplot(sns.husl_palette(10, l=.4))
```

Create a palette of 10 colors that are less saturated than the default:



```
>>> sns.palplot(sns.husl_palette(10, s=.4))
```



### 5.8.4 seaborn.hls_palette

seaborn.**hls_palette**(*n_colors=6*, *h=0.01*, *l=0.6*, *s=0.65*, *as_cmap=False*)

Get a set of evenly spaced colors in HLS hue space.

h, l, and s should be between 0 and 1

**Parameters**

**n_colors** [int] number of colors in the palette

**h** [float] first hue

**l** [float] lightness

**s** [float] saturation

**Returns**

**list of RGB tuples or** `matplotlib.colors.Colormap`

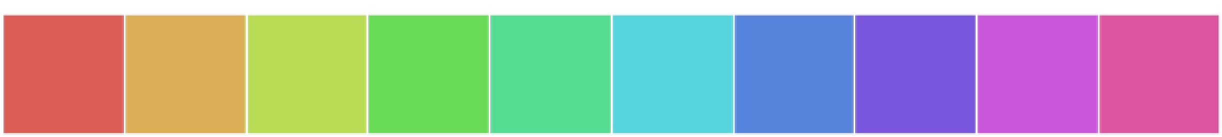
**See also:**

*husl_palette* Make a palette using evenly spaced hues in the HUSL system.

#### Examples

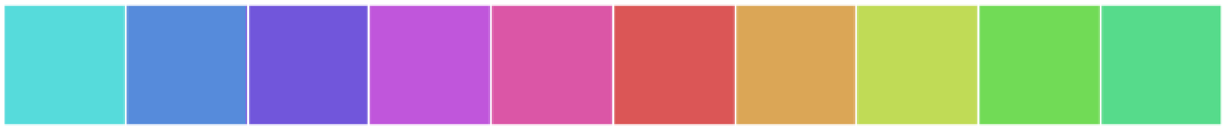
Create a palette of 10 colors with the default parameters:

```
>>> import seaborn as sns; sns.set_theme()
>>> sns.palplot(sns.hls_palette(10))
```



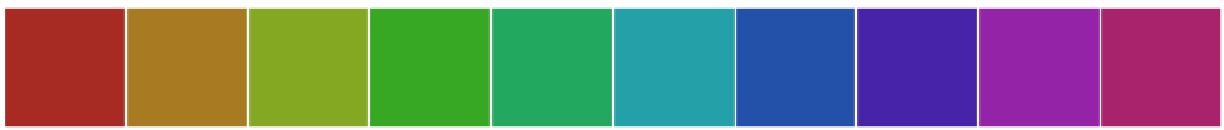
Create a palette of 10 colors that begins at a different hue value:

```
>>> sns.palplot(sns.hls_palette(10, h=.5))
```



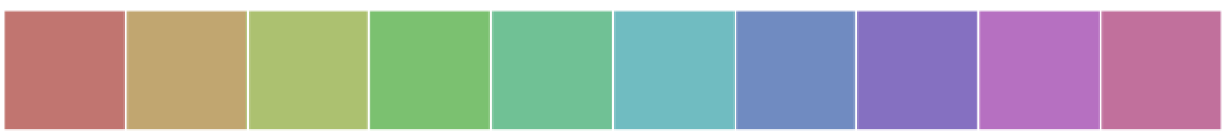
Create a palette of 10 colors that are darker than the default:

```
>>> sns.palplot(sns.hls_palette(10, l=.4))
```



Create a palette of 10 colors that are less saturated than the default:

```
>>> sns.palplot(sns.hls_palette(10, s=.4))
```



### 5.8.5 seaborn.cubehelix_palette

`seaborn.`**`cubehelix_palette`**(*n_colors=6*, *start=0*, *rot=0.4*, *gamma=1.0*, *hue=0.8*, *light=0.85*, *dark=0.15*, *reverse=False*, *as_cmap=False*)

Make a sequential palette from the cubehelix system.

This produces a colormap with linearly-decreasing (or increasing) brightness. That means that information will be preserved if printed to black and white or viewed by someone who is colorblind. "cubehelix" is also available as a matplotlib-based palette, but this function gives the user more control over the look of the palette and has a different set of defaults.

In addition to using this function, it is also possible to generate a cubehelix palette generally in seaborn using a string-shorthand; see the example below.

**Parameters**

**n_colors** [int] Number of colors in the palette.

**start** [float, 0 <= start <= 3] The hue at the start of the helix.

**rot** [float] Rotations around the hue wheel over the range of the palette.

**gamma** [float 0 <= gamma] Gamma factor to emphasize darker (gamma < 1) or lighter (gamma > 1) colors.

**hue** [float, 0 <= hue <= 1] Saturation of the colors.

**dark** [float 0 <= dark <= 1] Intensity of the darkest color in the palette.

**light** [float 0 <= light <= 1] Intensity of the lightest color in the palette.

**reverse** [bool] If True, the palette will go from dark to light.

**as_cmap** [bool] If True, return a `matplotlib.colors.Colormap`.

**Returns**

**list of RGB tuples or `matplotlib.colors.Colormap`**

**See also:**

*`choose_cubehelix_palette`* Launch an interactive widget to select cubehelix palette parameters.

*`dark_palette`* Create a sequential palette with dark low values.

*`light_palette`* Create a sequential palette with bright low values.

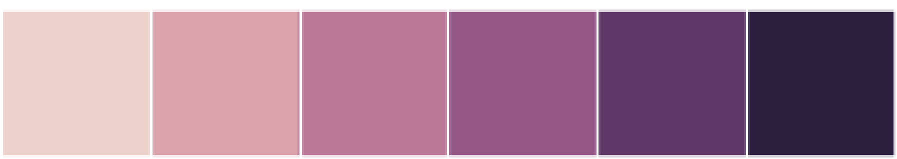
### References

Green, D. A. (2011). "A colour scheme for the display of astronomical intensity images". Bulletin of the Astromical Society of India, Vol. 39, p. 289-295.

### Examples

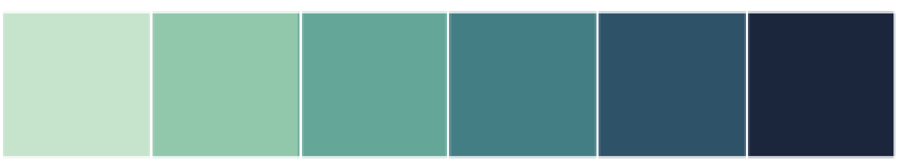
Generate the default palette:

```
>>> import seaborn as sns; sns.set_theme()
>>> sns.palplot(sns.cubehelix_palette())
```



Rotate backwards from the same starting location:

```
>>> sns.palplot(sns.cubehelix_palette(rot=-.4))
```



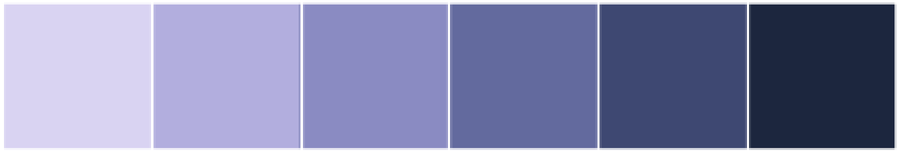
Use a different starting point and shorter rotation:

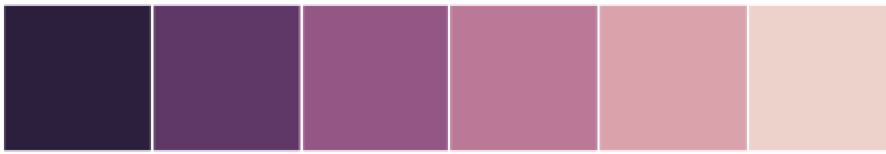
```
>>> sns.palplot(sns.cubehelix_palette(start=2.8, rot=.1))
```

Reverse the direction of the lightness ramp:

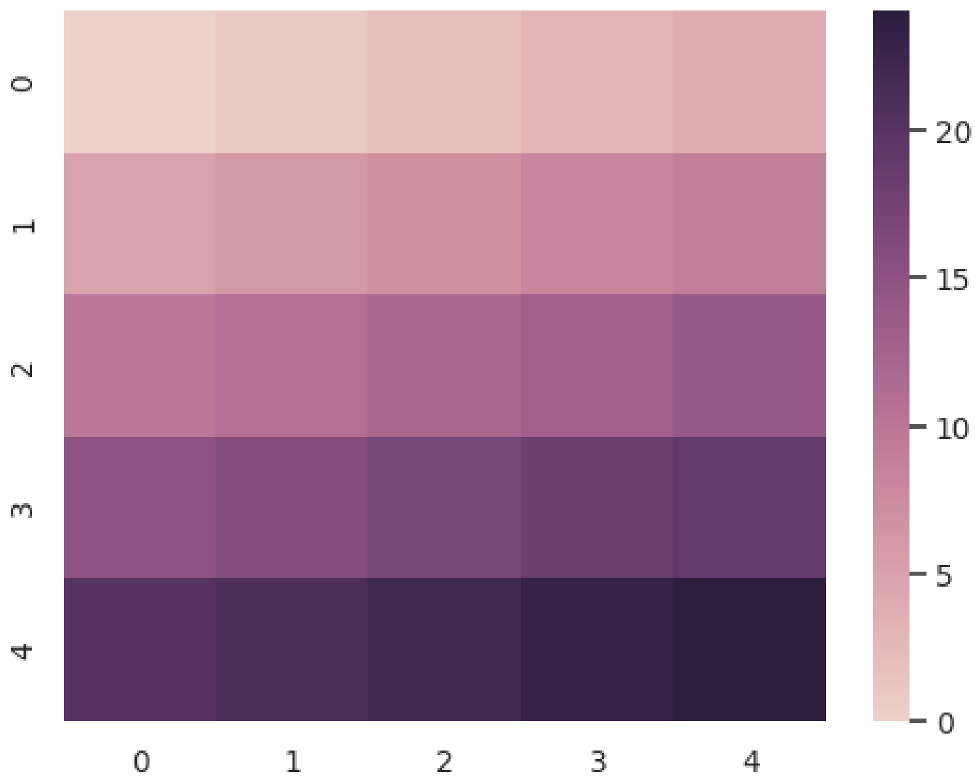
```
>>> sns.palplot(sns.cubehelix_palette(reverse=True))
```

Generate a colormap object:





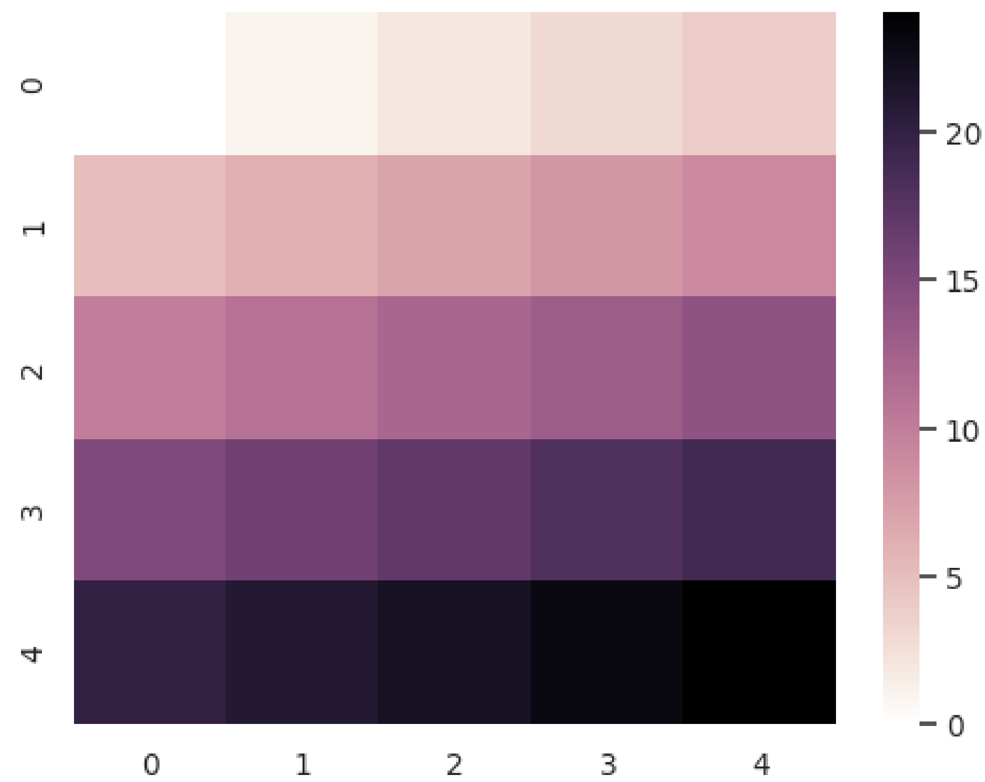
```
>>> from numpy import arange
>>> x = arange(25).reshape(5, 5)
>>> cmap = sns.cubehelix_palette(as_cmap=True)
>>> ax = sns.heatmap(x, cmap=cmap)
```



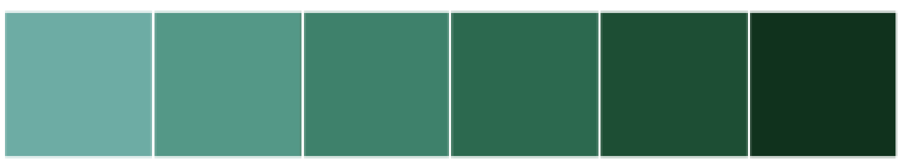

Use the full lightness range:

```
>>> cmap = sns.cubehelix_palette(dark=0, light=1, as_cmap=True)
>>> ax = sns.heatmap(x, cmap=cmap)
```

Use through the `color_palette()` interface:




```
>>> sns.palplot(sns.color_palette("ch:2,r=.2,l=.6"))
```



### 5.8.6 seaborn.dark_palette

seaborn.**dark_palette**(*color*, *n_colors=6*, *reverse=False*, *as_cmap=False*, *input='rgb'*)

Make a sequential palette that blends from dark to `color`.

This kind of palette is good for data that range between relatively uninteresting low values and interesting high values.

The `color` parameter can be specified in a number of ways, including all options for defining a color in matplotlib and several additional color spaces that are handled by seaborn. You can also use the database of named colors from the XKCD color survey.

If you are using the IPython notebook, you can also choose this palette interactively with the `choose_dark_palette()` function.

**Parameters**

**color** [base color for high values] hex, rgb-tuple, or html color name

**n_colors** [int, optional] number of colors in the palette

**reverse** [bool, optional] if True, reverse the direction of the blend

**as_cmap** [bool, optional] If True, return a `matplotlib.colors.Colormap`.

**input** [{'rgb', 'hls', 'husl', xkcd'}] Color space to interpret the input color. The first three options apply to tuple inputs and the latter applies to string inputs.

**Returns**

**list of RGB tuples or `matplotlib.colors.Colormap`**

**See also:**

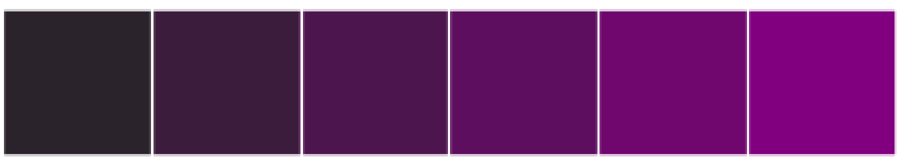
*`light_palette`* Create a sequential palette with bright low values.

*`diverging_palette`* Create a diverging palette with two colors.

## Examples

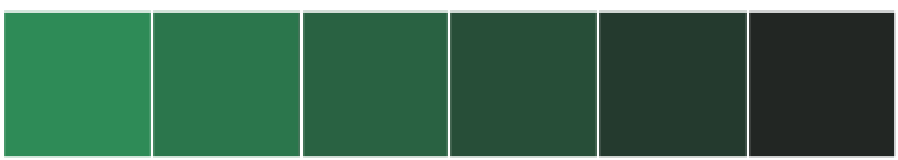
Generate a palette from an HTML color:

```
>>> import seaborn as sns; sns.set_theme()
>>> sns.palplot(sns.dark_palette("purple"))
```



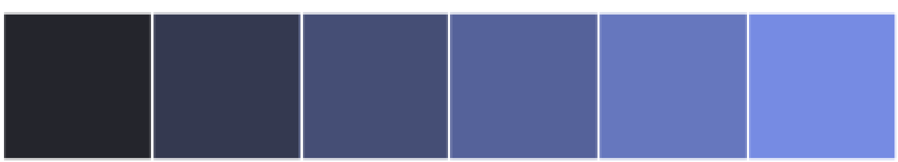
Generate a palette that decreases in lightness:

```
>>> sns.palplot(sns.dark_palette("seagreen", reverse=True))
```



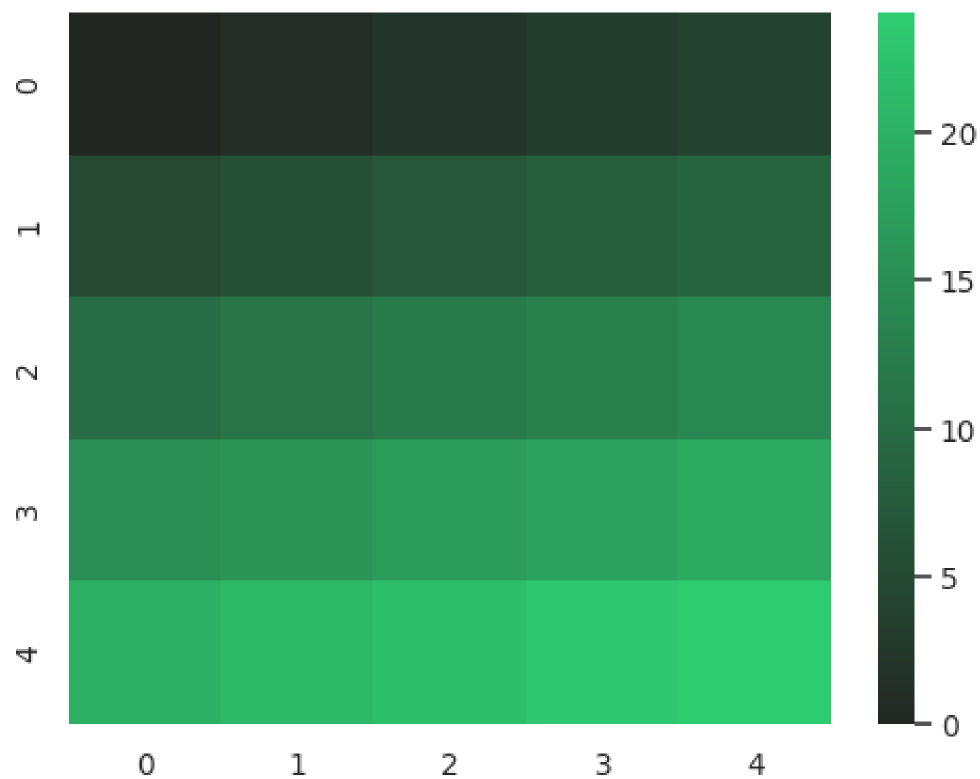
Generate a palette from an HUSL-space seed:

```
>>> sns.palplot(sns.dark_palette((260, 75, 60), input="husl"))
```



Generate a colormap object:

```
>>> from numpy import arange
>>> x = arange(25).reshape(5, 5)
>>> cmap = sns.dark_palette("#2ecc71", as_cmap=True)
>>> ax = sns.heatmap(x, cmap=cmap)
```




### 5.8.7 seaborn.light_palette

seaborn.**light_palette**(*color*, *n_colors=6*, *reverse=False*, *as_cmap=False*, *input='rgb'*)

Make a sequential palette that blends from light to `color`.

This kind of palette is good for data that range between relatively uninteresting low values and interesting high values.

The `color` parameter can be specified in a number of ways, including all options for defining a color in matplotlib and several additional color spaces that are handled by seaborn. You can also use the database of named colors from the XKCD color survey.

If you are using the IPython notebook, you can also choose this palette interactively with the *choose_light_palette()* function.

**Parameters**

**color** [base color for high values] hex code, html color name, or tuple in `input` space.

**n_colors** [int, optional] number of colors in the palette

**reverse** [bool, optional] if True, reverse the direction of the blend

**as_cmap** [bool, optional] If True, return a `matplotlib.colors.Colormap`.

**input** [{'rgb', 'hls', 'husl', xkcd'}] Color space to interpret the input color. The first three options apply to tuple inputs and the latter applies to string inputs.

**Returns**

**list of RGB tuples or** `matplotlib.colors.Colormap`

**See also:**

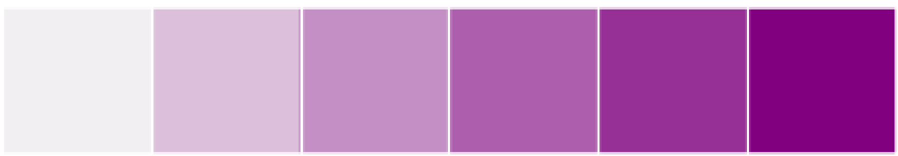
*dark_palette* Create a sequential palette with dark low values.

*diverging_palette* Create a diverging palette with two colors.

## Examples

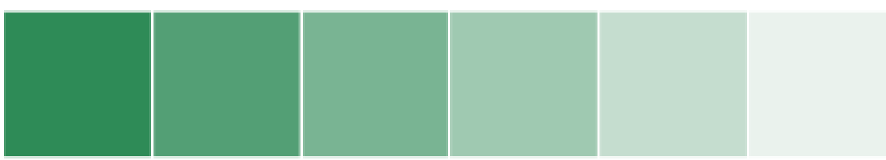
Generate a palette from an HTML color:

```
>>> import seaborn as sns; sns.set_theme()
>>> sns.palplot(sns.light_palette("purple"))
```



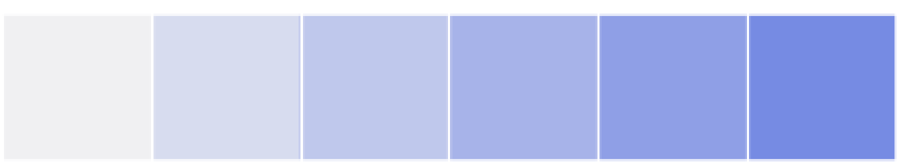
Generate a palette that increases in lightness:

```
>>> sns.palplot(sns.light_palette("seagreen", reverse=True))
```



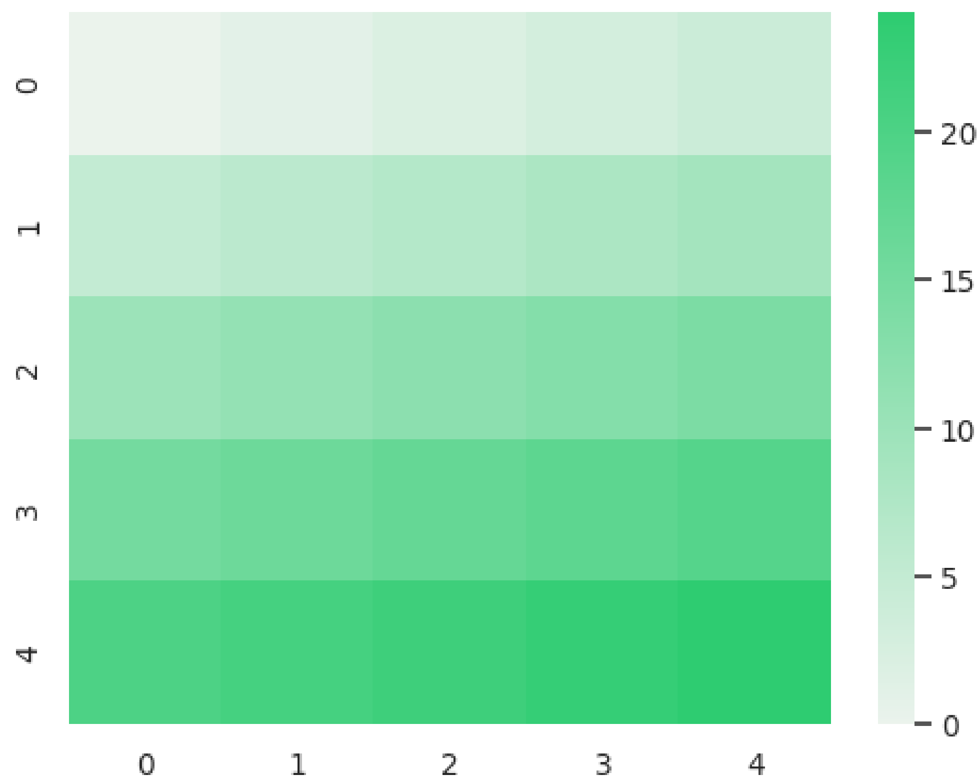
Generate a palette from an HUSL-space seed:

```
>>> sns.palplot(sns.light_palette((260, 75, 60), input="husl"))
```



Generate a colormap object:

```
>>> from numpy import arange
>>> x = arange(25).reshape(5, 5)
>>> cmap = sns.light_palette("#2ecc71", as_cmap=True)
>>> ax = sns.heatmap(x, cmap=cmap)
```




## 5.8.8 seaborn.diverging_palette

seaborn.**diverging_palette**(*h_neg*, *h_pos*, *s=75*, *l=50*, *sep=1*, *n=6*, *center='light'*, *as_cmap=False*)

Make a diverging palette between two HUSL colors.

If you are using the IPython notebook, you can also choose this palette interactively with the *choose_diverging_palette()* function.

**Parameters**

**h_neg, h_pos** [float in [0, 359]] Anchor hues for negative and positive extents of the map.

**s** [float in [0, 100], optional] Anchor saturation for both extents of the map.

**l** [float in [0, 100], optional] Anchor lightness for both extents of the map.

**sep** [int, optional] Size of the intermediate region.

**n** [int, optional] Number of colors in the palette (if not returning a cmap)

**center** [{"light", "dark"}, optional] Whether the center of the palette is light or dark

**as_cmap** [bool, optional] If True, return a matplotlib.colors.Colormap.

**Returns**

**list of RGB tuples or matplotlib.colors.Colormap**

**See also:**

***dark_palette*** Create a sequential palette with dark values.

***light_palette*** Create a sequential palette with light values.

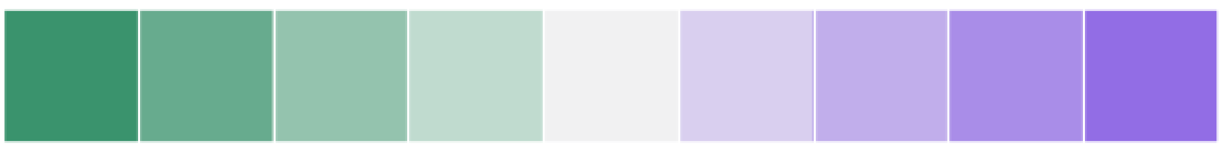
### Examples

Generate a blue-white-red palette:

```
>>> import seaborn as sns; sns.set_theme()
>>> sns.palplot(sns.diverging_palette(240, 10, n=9))
```

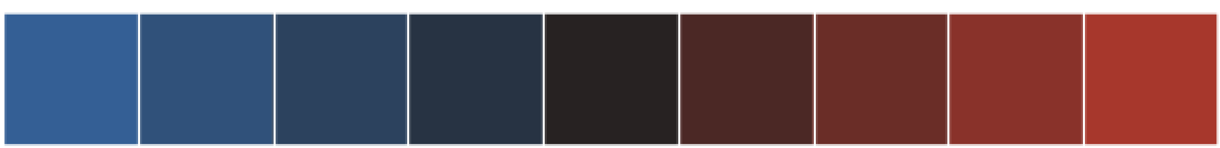
Generate a brighter green-white-purple palette:

```
>>> sns.palplot(sns.diverging_palette(150, 275, s=80, l=55, n=9))
```



Generate a blue-black-red palette:

```
>>> sns.palplot(sns.diverging_palette(250, 15, s=75, l=40,
...                                   n=9, center="dark"))
```



Generate a colormap object:

```
>>> from numpy import arange
>>> x = arange(25).reshape(5, 5)
>>> cmap = sns.diverging_palette(220, 20, as_cmap=True)
>>> ax = sns.heatmap(x, cmap=cmap)
```

## 5.8.9 seaborn.blend_palette

`seaborn.`**`blend_palette`**(*colors*, *n_colors=6*, *as_cmap=False*, *input='rgb'*)

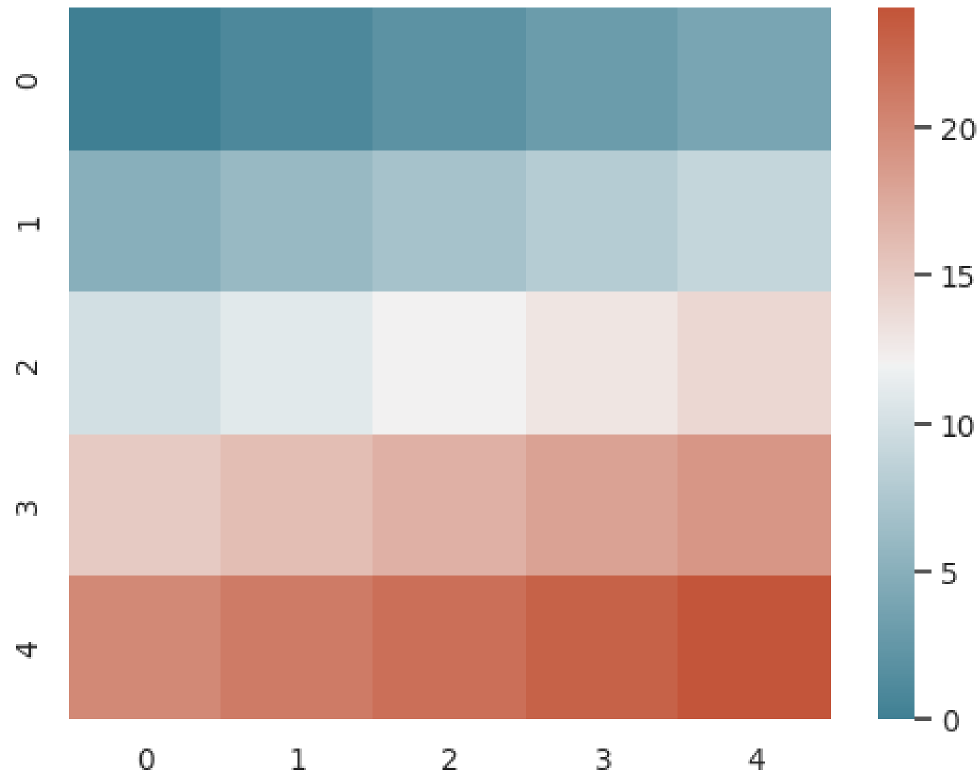
Make a palette that blends between a list of colors.

**Parameters**

**colors** [sequence of colors in various formats interpreted by `input`] hex code, html color name, or tuple in `input` space.

**n_colors** [int, optional] Number of colors in the palette.

**as_cmap** [bool, optional] If True, return a `matplotlib.colors.Colormap`.




**Returns**

**list of RGB tuples or** `matplotlib.colors.Colormap`

### 5.8.10 seaborn.xkcd_palette

`seaborn.xkcd_palette(colors)`

Make a palette with color names from the xkcd color survey.

See xkcd for the full list of colors: https://xkcd.com/color/rgb/

This is just a simple wrapper around the `seaborn.xkcd_rgb` dictionary.

**Parameters**

**colors** [list of strings] List of keys in the `seaborn.xkcd_rgb` dictionary.

**Returns**

**palette** [seaborn color palette] Returns the list of colors as RGB tuples in an object that behaves like other seaborn color palettes.

**See also:**

*crayon_palette* Make a palette with Crayola crayon colors.

### 5.8.11 seaborn.crayon_palette

`seaborn.crayon_palette(colors)`

Make a palette with color names from Crayola crayons.

Colors are taken from here: https://en.wikipedia.org/wiki/List_of_Crayola_crayon_colors

This is just a simple wrapper around the `seaborn.crayons` dictionary.

**Parameters**

**colors** [list of strings] List of keys in the `seaborn.crayons` dictionary.

**Returns**

**palette** [seaborn color palette] Returns the list of colors as rgb tuples in an object that behaves like other seaborn color palettes.

**See also:**

***xkcd_palette*** Make a palette with named colors from the XKCD color survey.

### 5.8.12 seaborn.mpl_palette

`seaborn.mpl_palette(name, n_colors=6, as_cmap=False)`

Return discrete colors from a matplotlib palette.

Note that this handles the qualitative colorbrewer palettes properly, although if you ask for more colors than a particular qualitative palette can provide you will get fewer than you are expecting. In contrast, asking for qualitative color brewer palettes using `color_palette()` will return the expected number of colors, but they will cycle.

If you are using the IPython notebook, you can also use the function `choose_colorbrewer_palette()` to interactively select palettes.

**Parameters**

**name** [string] Name of the palette. This should be a named matplotlib colormap.

**n_colors** [int] Number of discrete colors in the palette.

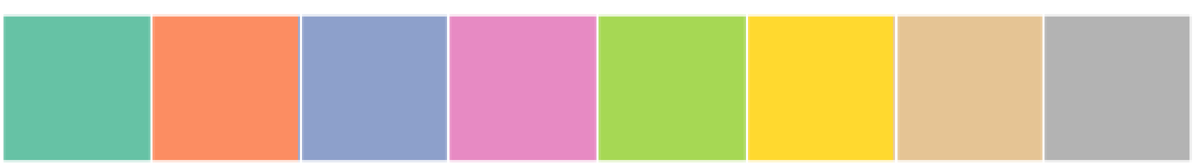
**Returns**

**list of RGB tuples or** **`matplotlib.colors.Colormap`**

#### Examples

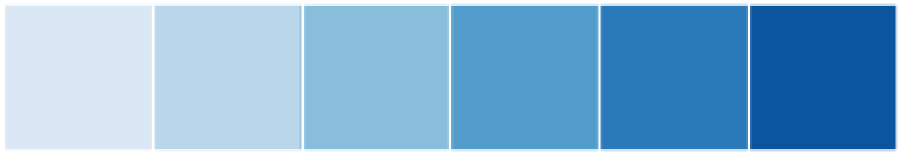
Create a qualitative colorbrewer palette with 8 colors:

```
>>> import seaborn as sns; sns.set_theme()
>>> sns.palplot(sns.mpl_palette("Set2", 8))
```



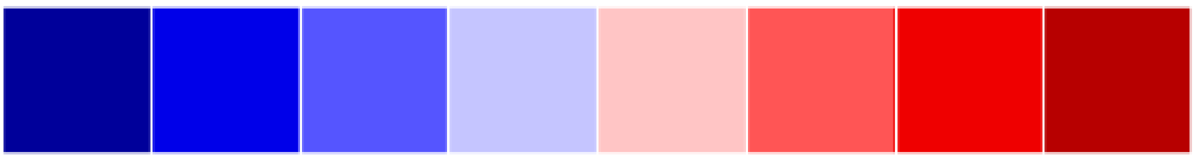
Create a sequential colorbrewer palette:

```
>>> sns.palplot(sns.mpl_palette("Blues"))
```



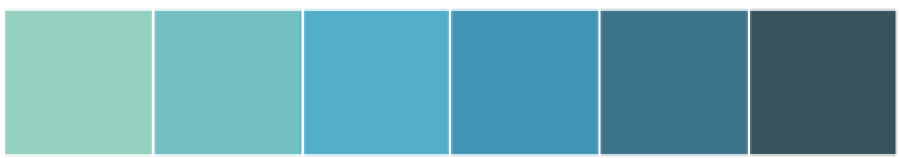
Create a diverging palette:

```
>>> sns.palplot(sns.mpl_palette("seismic", 8))
```



Create a "dark" sequential palette:

```
>>> sns.palplot(sns.mpl_palette("GnBu_d"))
```



## 5.9 Palette widgets

| | |
|---|---|
| *choose_colorbrewer_palette* | Select a palette from the ColorBrewer set. |
| *choose_cubehelix_palette* | Launch an interactive widget to create a sequential cubehelix palette. |
| *choose_light_palette* | Launch an interactive widget to create a light sequential palette. |
| *choose_dark_palette* | Launch an interactive widget to create a dark sequential palette. |
| *choose_diverging_palette* | Launch an interactive widget to choose a diverging color palette. |

### 5.9.1 seaborn.choose_colorbrewer_palette

seaborn.**choose_colorbrewer_palette**(*data_type*, *as_cmap=False*)

Select a palette from the ColorBrewer set.

These palettes are built into matplotlib and can be used by name in many seaborn functions, or by passing the object returned by this function.

**Parameters**

**data_type** [{'sequential', 'diverging', 'qualitative'}] This describes the kind of data you want to visualize. See the seaborn color palette docs for more information about how to choose this value. Note that you can pass substrings (e.g. 'q' for 'qualitative.

**as_cmap** [bool] If True, the return value is a matplotlib colormap rather than a list of discrete colors.

**Returns**

**pal or cmap** [list of colors or matplotlib colormap] Object that can be passed to plotting functions.

**See also:**

***dark_palette*** Create a sequential palette with dark low values.

***light_palette*** Create a sequential palette with bright low values.

***diverging_palette*** Create a diverging palette from selected colors.

***cubehelix_palette*** Create a sequential palette or colormap using the cubehelix system.

### 5.9.2 seaborn.choose_cubehelix_palette

seaborn.**choose_cubehelix_palette**(*as_cmap=False*)

Launch an interactive widget to create a sequential cubehelix palette.

This corresponds with the *cubehelix_palette()* function. This kind of palette is good for data that range between relatively uninteresting low values and interesting high values. The cubehelix system allows the palette to have more hue variance across the range, which can be helpful for distinguishing a wider range of values.

Requires IPython 2+ and must be used in the notebook.

**Parameters**

**as_cmap** [bool] If True, the return value is a matplotlib colormap rather than a list of discrete colors.

**Returns**

**pal or cmap** [list of colors or matplotlib colormap] Object that can be passed to plotting functions.

**See also:**

***cubehelix_palette*** Create a sequential palette or colormap using the cubehelix system.

### 5.9.3 seaborn.choose_light_palette

seaborn.**choose_light_palette**(*input='husl'*, *as_cmap=False*)

Launch an interactive widget to create a light sequential palette.

This corresponds with the `light_palette()` function. This kind of palette is good for data that range between relatively uninteresting low values and interesting high values.

Requires IPython 2+ and must be used in the notebook.

**Parameters**

**input** [{'husl', 'hls', 'rgb'}] Color space for defining the seed value. Note that the default is different than the default input for `light_palette()`.

**as_cmap** [bool] If True, the return value is a matplotlib colormap rather than a list of discrete colors.

**Returns**

**pal or cmap** [list of colors or matplotlib colormap] Object that can be passed to plotting functions.

**See also:**

**`light_palette`** Create a sequential palette with bright low values.

**`dark_palette`** Create a sequential palette with dark low values.

**`cubehelix_palette`** Create a sequential palette or colormap using the cubehelix system.

### 5.9.4 seaborn.choose_dark_palette

seaborn.**choose_dark_palette**(*input='husl'*, *as_cmap=False*)

Launch an interactive widget to create a dark sequential palette.

This corresponds with the `dark_palette()` function. This kind of palette is good for data that range between relatively uninteresting low values and interesting high values.

Requires IPython 2+ and must be used in the notebook.

**Parameters**

**input** [{'husl', 'hls', 'rgb'}] Color space for defining the seed value. Note that the default is different than the default input for `dark_palette()`.

**as_cmap** [bool] If True, the return value is a matplotlib colormap rather than a list of discrete colors.

**Returns**

**pal or cmap** [list of colors or matplotlib colormap] Object that can be passed to plotting functions.

**See also:**

**`dark_palette`** Create a sequential palette with dark low values.

**`light_palette`** Create a sequential palette with bright low values.

**`cubehelix_palette`** Create a sequential palette or colormap using the cubehelix system.

### 5.9.5 seaborn.choose_diverging_palette

seaborn.**choose_diverging_palette**(*as_cmap=False*)

Launch an interactive widget to choose a diverging color palette.

This corresponds with the `diverging_palette()` function. This kind of palette is good for data that range between interesting low values and interesting high values with a meaningful midpoint. (For example, change scores relative to some baseline value).

Requires IPython 2+ and must be used in the notebook.

**Parameters**

**as_cmap** [bool] If True, the return value is a matplotlib colormap rather than a list of discrete colors.

**Returns**

**pal or cmap** [list of colors or matplotlib colormap] Object that can be passed to plotting functions.

**See also:**

**`diverging_palette`** Create a diverging color palette or colormap.

**`choose_colorbrewer_palette`** Interactively choose palettes from the colorbrewer set, including diverging palettes.

## 5.10 Utility functions

| | |
|---|---|
| `load_dataset` | Load an example dataset from the online repository (requires internet). |
| `get_dataset_names` | Report available example datasets, useful for reporting issues. |
| `get_data_home` | Return a path to the cache directory for example datasets. |
| `despine` | Remove the top and right spines from plot(s). |
| `desaturate` | Decrease the saturation channel of a color by some percent. |
| `saturate` | Return a fully saturated color with the same hue. |
| `set_hls_values` | Independently manipulate the h, l, or s channels of a color. |

### 5.10.1 seaborn.load_dataset

seaborn.**load_dataset**(*name*, *cache=True*, *data_home=None*, ***kws*)

Load an example dataset from the online repository (requires internet).

This function provides quick access to a small number of example datasets that are useful for documenting seaborn or generating reproducible examples for bug reports. It is not necessary for normal usage.

Note that some of the datasets have a small amount of preprocessing applied to define a proper ordering for categorical variables.

Use `get_dataset_names()` to see a list of available datasets.

**Parameters**

**name** [str] Name of the dataset ({name}.csv on https://github.com/mwaskom/seaborn-data).

**cache** [boolean, optional] If True, try to load from the local cache first, and save to the cache if a download is required.

**data_home** [string, optional] The directory in which to cache data; see *get_data_home()*.

**kws** [keys and values, optional] Additional keyword arguments are passed to passed through to pandas.read_csv().

**Returns**

**df** [pandas.DataFrame] Tabular data, possibly with some preprocessing applied.

### 5.10.2 seaborn.get_dataset_names

seaborn.**get_dataset_names**()

Report available example datasets, useful for reporting issues.

Requires an internet connection.

### 5.10.3 seaborn.get_data_home

seaborn.**get_data_home**(*data_home=None*)

Return a path to the cache directory for example datasets.

This directory is then used by *load_dataset()*.

If the data_home argument is not specified, it tries to read from the SEABORN_DATA environment variable and defaults to ~/seaborn-data.

### 5.10.4 seaborn.despine

seaborn.**despine**(*fig=None*, *ax=None*, *top=True*, *right=True*, *left=False*, *bottom=False*, *offset=None*, *trim=False*)

Remove the top and right spines from plot(s).

**fig** [matplotlib figure, optional] Figure to despine all axes of, defaults to the current figure.

**ax** [matplotlib axes, optional] Specific axes object to despine. Ignored if fig is provided.

**top, right, left, bottom** [boolean, optional] If True, remove that spine.

**offset** [int or dict, optional] Absolute distance, in points, spines should be moved away from the axes (negative values move spines inward). A single value applies to all spines; a dict can be used to set offset values per side.

**trim** [bool, optional] If True, limit spines to the smallest and largest major tick on each non-despined axis.

**Returns**

**None**

### 5.10.5 seaborn.desaturate

`seaborn.`**`desaturate`**(*color*, *prop*)

Decrease the saturation channel of a color by some percent.

**Parameters**

**color** [matplotlib color] hex, rgb-tuple, or html color name

**prop** [float] saturation channel of color will be multiplied by this value

**Returns**

**new_color** [rgb tuple] desaturated color code in RGB tuple representation

### 5.10.6 seaborn.saturate

`seaborn.`**`saturate`**(*color*)

Return a fully saturated color with the same hue.

**Parameters**

**color** [matplotlib color] hex, rgb-tuple, or html color name

**Returns**

**new_color** [rgb tuple] saturated color code in RGB tuple representation

### 5.10.7 seaborn.set_hls_values

`seaborn.`**`set_hls_values`**(*color*, *h=None*, *l=None*, *s=None*)

Independently manipulate the h, l, or s channels of a color.

**Parameters**

**color** [matplotlib color] hex, rgb-tuple, or html color name

**h, l, s** [floats between 0 and 1, or None] new values for each channel in hls space

**Returns**

**new_color** [rgb tuple] new color code in RGB tuple representation

# CITING AND LOGO

## 6.1 Citing seaborn

If seaborn is integral to a scientific publication, we would appreciate a citation. While there is not currently a paper describing seaborn, the library can be cited using the following DOI:

```
@software{waskom2020seaborn,
    author       = {Michael Waskom and the seaborn development team},
    title        = {mwaskom/seaborn},
    month        = sep,
    year         = 2020,
    publisher    = {Zenodo},
    version      = {latest},
    doi          = {10.5281/zenodo.592845},
    url          = {https://doi.org/10.5281/zenodo.592845},
}
```

Visit the Zenodo page for version-specific DOIs and contributor lists.

## 6.2 Logo files

Additional logo files, including hi-res PNGs and images suitable for use over a dark background, are available on GitHub.

### 6.2.1 Wide logo

### 6.2.2 Tall logo

### 6.2.3 Logo mark

Credit to Matthias Bussonnier for the initial design and implementation of the logo.

CHAPTER

SEVEN

# DOCUMENTATION ARCHIVE

- Version 0.10
- Version 0.9
- Relational: *API* | Tutorial
- Distribution: *API* | Tutorial
- Categorical: *API* | Tutorial
- Regression: *API* | Tutorial
- Multiples: *API* | Tutorial
- Style: *API* | Tutorial
- Color: *API* | Tutorial

# INDEX

## R



## S



## V



## X