In The Name of God

Data Visualization with Python

By:

Hossein Haghbin

Department of Statistics,
Faculty of Intelligent Systems
Engineering and Data Science,
Persian Gulf University

October 8, 2025

Chapter 2

seaborn

Table of contents

Introduction

Seaborn is a library for making **statistical** graphics in Python. It builds on top of `matplotlib` and integrates closely with `pandas` data structures.

Seaborn helps you explore and understand your data. Its plotting functions operate on `dataframes` and `arrays` containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots. Most of your interactions with seaborn will happen through a set of plotting functions. Later chapters in the tutorial will explore the specific features offered by each function.

Figure-level vs. axes-level functions

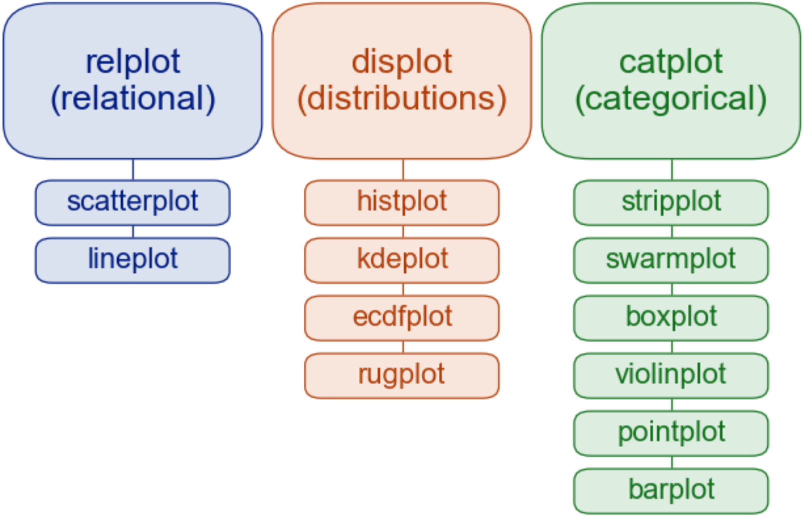
There is a cross-cutting classification of seaborn functions as **axes-level** or **figure-level**. The axes-level functions plot data onto a single `matplotlib.pyplot.Axes` object, which is the return value of the function.
In contrast, figure-level functions interface with matplotlib through a seaborn object, usually a FacetGrid, that manages the figure. Each module has a single figure-level function, which offers a unitary interface to its various axes-level functions.

The organization of Seaborn plots

Let’s get started! I have divided this implementation section into three categories:

- Visualizing statistical relationships
- Visualizing distributions of data
- Plotting categorical data

The organization looks a bit like this:


relplot
(relational)
scatterplot
lineplot
displot
(distributions)
histplot
kdeplot
ecdfplot
rugplot
catplot
(categorical)
stripplot
swarmplot
boxplot
violinplot
pointplot
barplot

1-Visualizing statistical relationships

1-Visualizing statistical relationships

We will discuss three seaborn functions. The one we will use most is **relplot()**. This is a figure-level function for visualizing statistical relationships using two common approaches: **scatter** plotsand **line** plots. relplot() combines a FacetGrid with one of two axes-level functions:

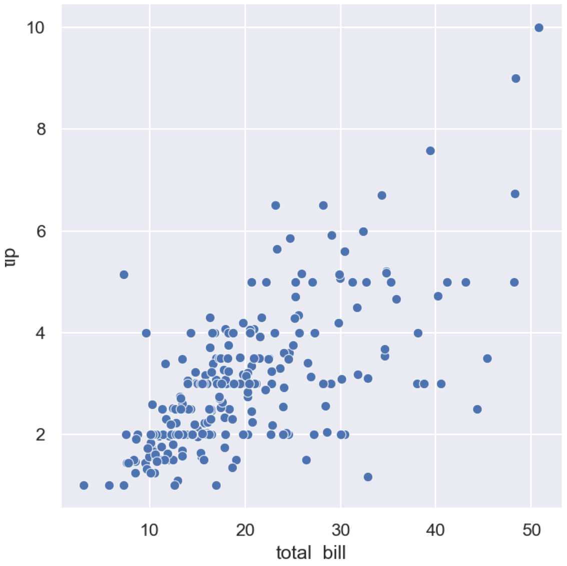
- scatterplot() (with kind=”scatter”; the default)
- lineplot() (with kind=”line”)

these functions can be quite illuminating because they use simple and easily-understood representations of data that can nevertheless represent complex dataset structures. They can do so because they plot two-dimensional graphics that can be enhanced by mapping up to three additional variables using the semantics of **hue**, **size**, and **style**.

Scatter plots

There are several ways to draw a scatter plot in seaborn. The most basic, which should be used when both variables are numeric, is the **scatterplot()** function. The **scatterplot()** is the default kind in **relplot()**

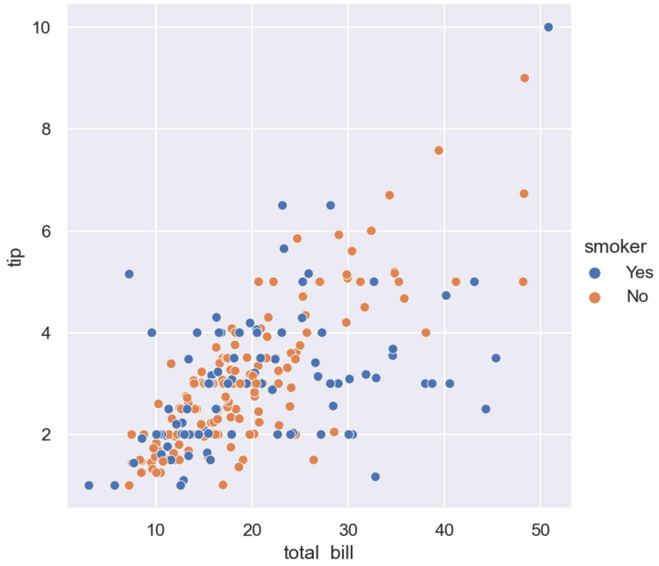
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
tips = sns.load_dataset("tips")
sns.relplot(x="total_bill", y="tip", data=tips);
plt.show()
```


10
8
6
4
2
tip
10
20
30
40
50
total_bill

hue semantic

While the points are plotted in two dimensions, another dimension can be added to the plot by coloring the points according to a third variable. In seaborn, this is referred to as using a **hue semantic**, because the color of the point gains meaning:

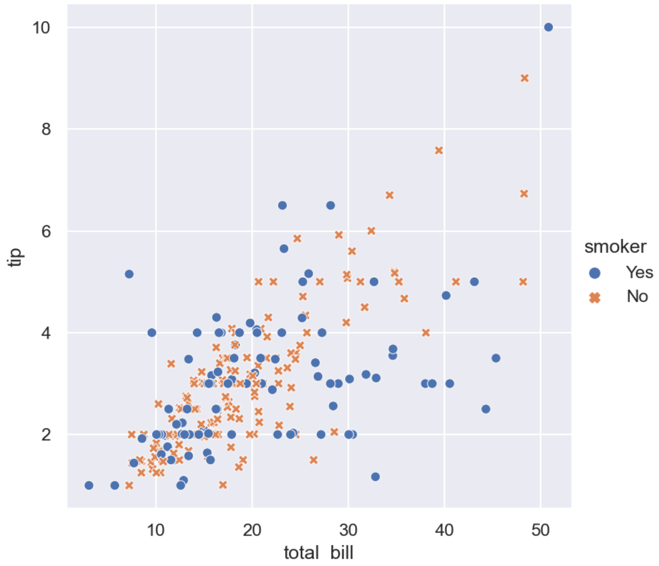
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
tips = sns.load_dataset("tips")
sns.relplot(x="total_bill", y="tip",
      hue="smoker", data=tips);
plt.show()
```


10
8
6
4
2
tip
10
20
30
40
50
total_bill
smoker
Yes
No

marker style

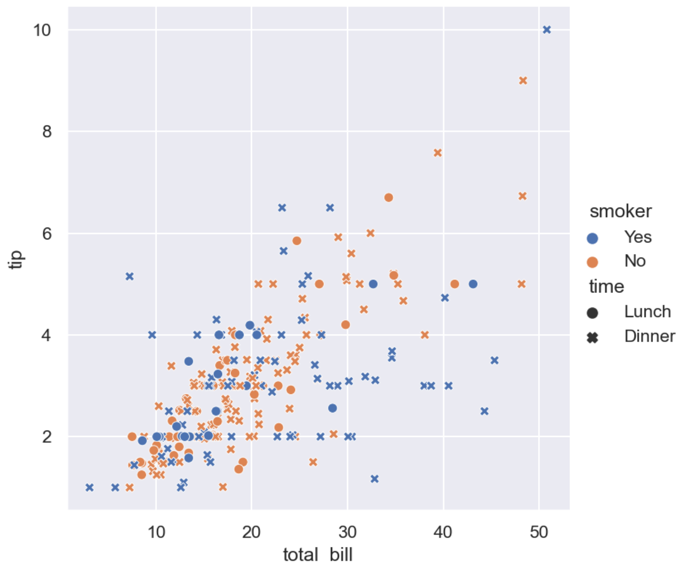
To emphasize the difference between the classes, and to improve accessibility, you can use a different marker style for each class:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
tips = sns.load_dataset("tips")
sns.relplot(x="total_bill", y="tip",
   hue="smoker", style="smoker",data=tips);
plt.show()
```


10
8
6
4
2
tip
10
20
30
40
50
total_bill
smoker
● Yes
✖ No

It's also possible to represent four variables by changing the hue and style of each point independently. But this should be done carefully, because the eye is much less sensitive to shape than to color:

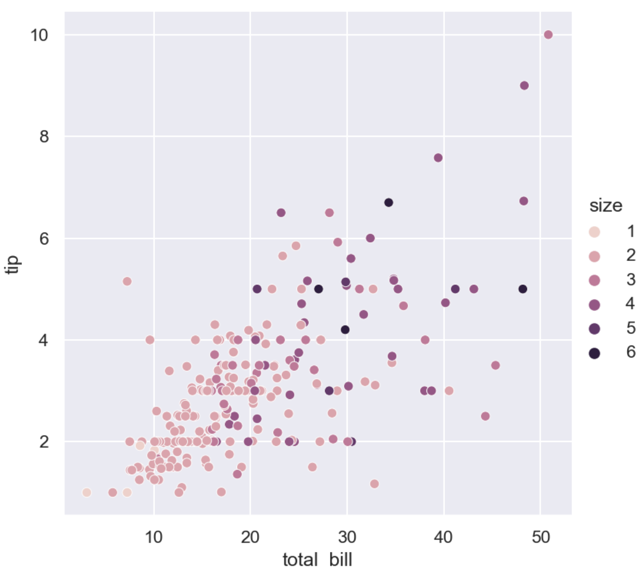
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
tips = sns.load_dataset("tips")
sns.relplot(x="total_bill", y="tip",
    hue="smoker", style="time", data=tips);
plt.show()
```


10
8
6
4
2
tip
10
20
30
40
50
total_bill
smoker
Yes
No
time
Lunch
Dinner

Numeric hue

In the examples above, the hue semantic was categorical, so the default qualitative palette was applied. If the hue semantic is numeric (specifically, if it can be cast to float), the default coloring switches to a sequential palette:

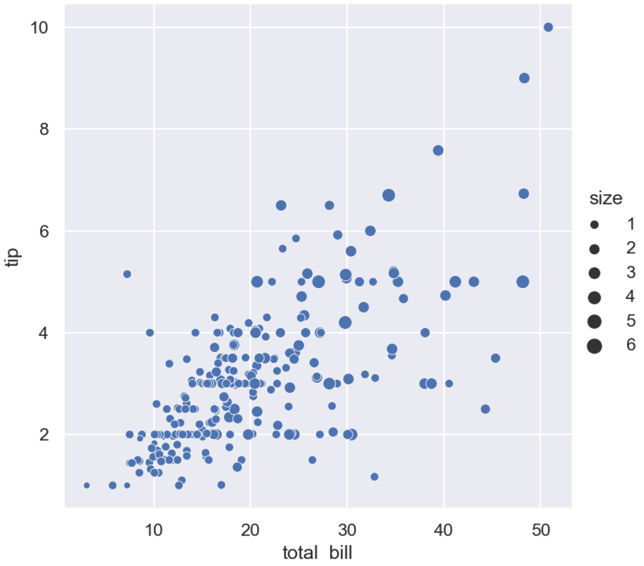
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
tips = sns.load_dataset("tips")
sns.relplot(x="total_bill", y="tip", hue="size",
    data=tips);
plt.show()
```


10
8
6
4
2
tip
10
20
30
40
50
total_bill
size
1
2
3
4
5
6

size semantic

The third kind of semantic variable changes the size of each point:

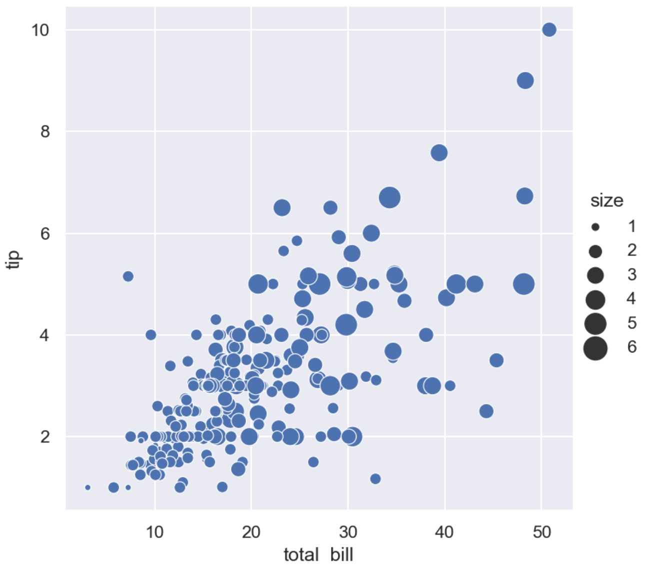
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
tips = sns.load_dataset("tips")
sns.relplot(x="total_bill", y="tip", size="size",
    data=tips);
plt.show()
```



10
8
6
4
2
tip
10
20
30
40
50
total_bill
size
1
2
3
4
5
6

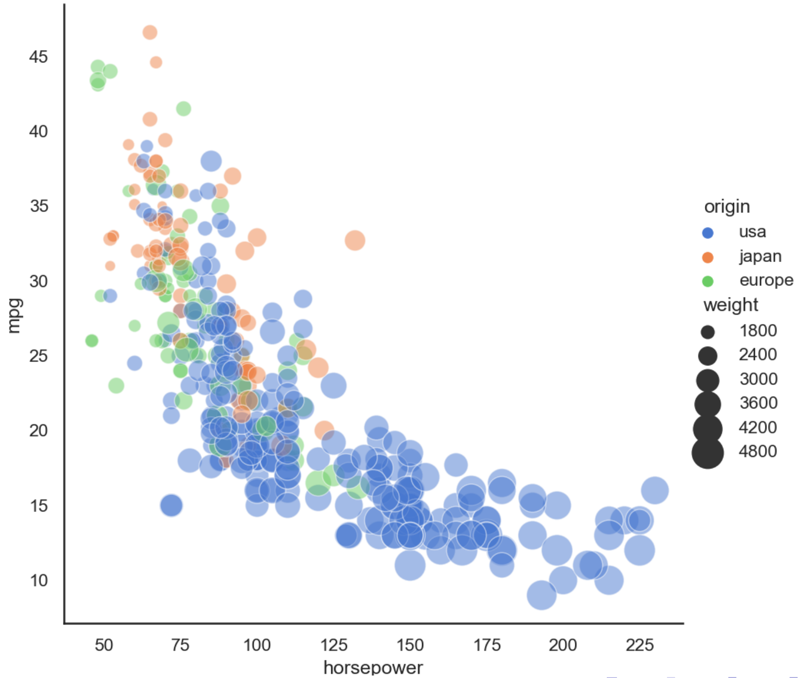
Unlike with **matplotlib.pyplot.scatter()**, the literal value of the variable is not used to pick the area of the point. Instead, the range of values in data units is normalized into a range in area units. This range can be customized:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
tips = sns.load_dataset("tips")
sns.relplot(x="total_bill", y="tip",
    size="size", sizes=(15, 200), data=tips);
plt.show()
```


size
1
2
3
4
5
6
tip
10
8
6
4
2
10
20
30
40
50
total_bill

Scatterplot with varying point sizes and hues

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="white")
mpg = sns.load_dataset("mpg")
sns.relplot(x="horsepower", y="mpg",
    hue="origin", size="weight",sizes=(40, 400),
    alpha=.5, palette="muted",height=6,data=mpg)
plt.show()
```

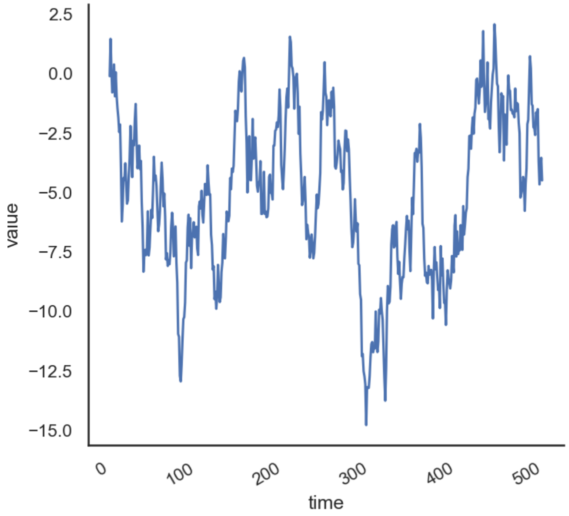


45
40
35
30
25
20
15
10
mpg
50
75
100
125
150
175
200
225
horsepower
origin
usa
japan
europe
weight
1800
2400
3000
3600
4200
4800

Line plots

With some datasets, you may want to understand changes in one variable as a function of time, or a similarly continuous variable. In this situation, a good choice is to draw a line plot. In seaborn, this can be accomplished by the **lineplot()** function, either directly or with **relplot()** by setting kind=”line”:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="white")
df = pd.DataFrame(dict(time=np.arange(500),
    value=np.random.randn(500).cumsum()))
g = sns.relplot(x="time", y="value",
     kind="line", data=df)
g.fig.autofmt_xdate()
plt.show()
```

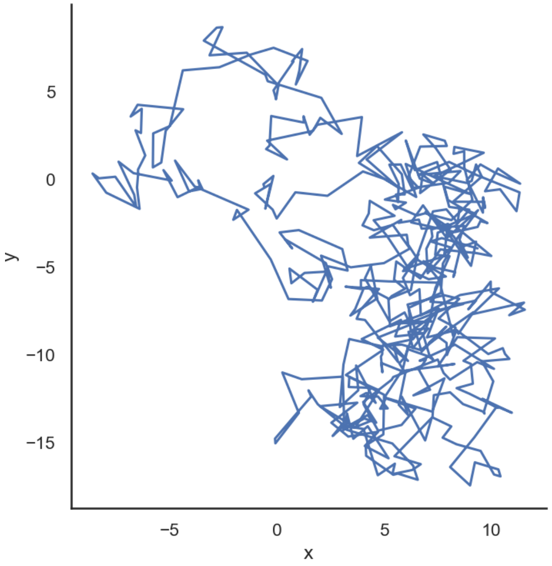


2.5
0.0
−2.5
−5.0
−7.5
−10.0
−12.5
−15.0
value
0
100
200
300
400
500
time

Disarrange data by the x values

Because **lineplot()** assumes that you are most often trying to draw y as a function of x, the default behavior is to sort the data by the x values before plotting. However, this can be disabled:

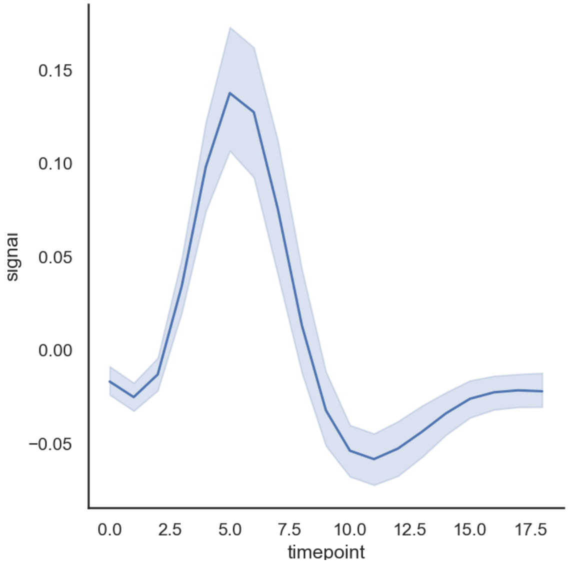
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="white")
df = pd.DataFrame(
     np.random.randn(500, 2).cumsum(axis=0),
     columns=["x", "y"])
sns.relplot(x="x", y="y",sort=False,kind="line",
      data=df);
plt.show()
```


5
0
−5
−10
−15
y
−5
0
5
10
x

Aggregation and representing uncertainty

More complex datasets will have multiple measurements for the same value of the x variable. The default behavior in seaborn is to aggregate the multiple measurements at each x value by plotting the mean and the 95% confidence interval around the mean. The confidence intervals are computed using bootstrapping, which can be time-intensive for larger datasets. It's therefore possible to disable them by **ci=None**.

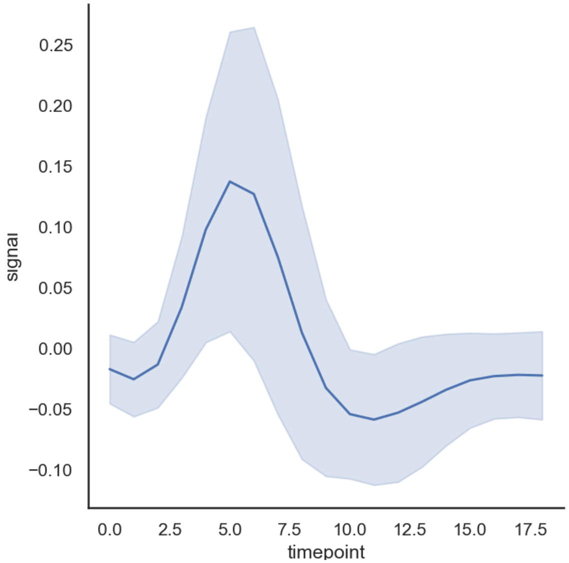
```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal",
      kind="line", data=fmri);
plt.show()
```


0.15
0.10
0.05
0.00
−0.05
signal
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
timepoint

A standard deviation CI

Another good option, especially with larger data, is to represent the spread of the distribution at each timepoint by plotting the standard deviation instead of a confidence interval:

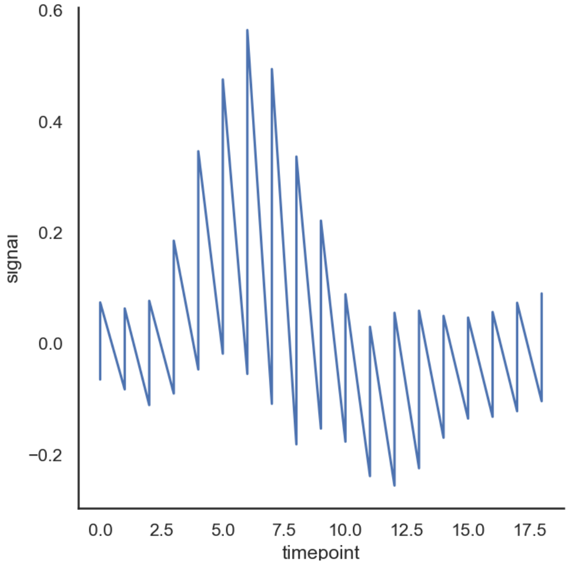
```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal", kind="line",
    ci="sd", data=fmri);
plt.show()
```


0.25
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
signal
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
timepoint

Turn aggregation off

To turn off aggregation altogether, set the estimator parameter to None This might produce a strange effect when the data have multiple observations at each point.

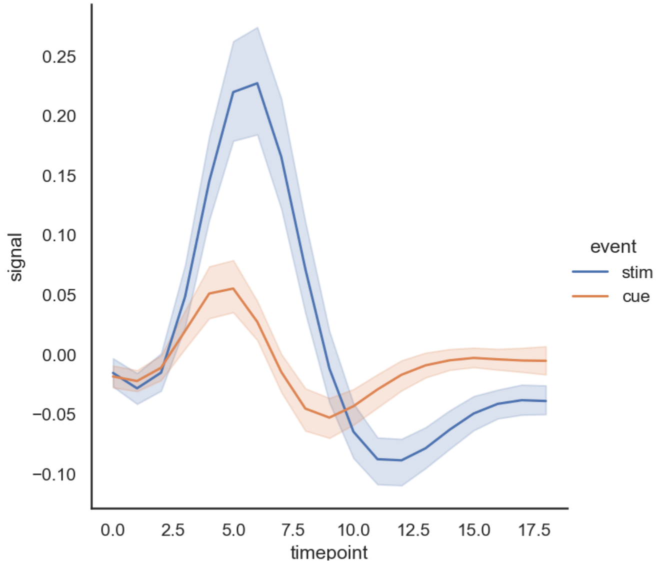
```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal",
    estimator=None,kind="line", data=fmri);
plt.show()
```


0.6
0.4
0.2
0.0
−0.2
signal
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
timepoint

Semantic mappings

The **lineplot()** function has the same flexibility as **scatterplot()**: it can show up to three additional variables by modifying the hue, size, and style of the plot elements. Using semantics in lineplot() will also determine how the data get aggregated.

```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal", hue="event",
    kind="line", data=fmri);
plt.show()
```

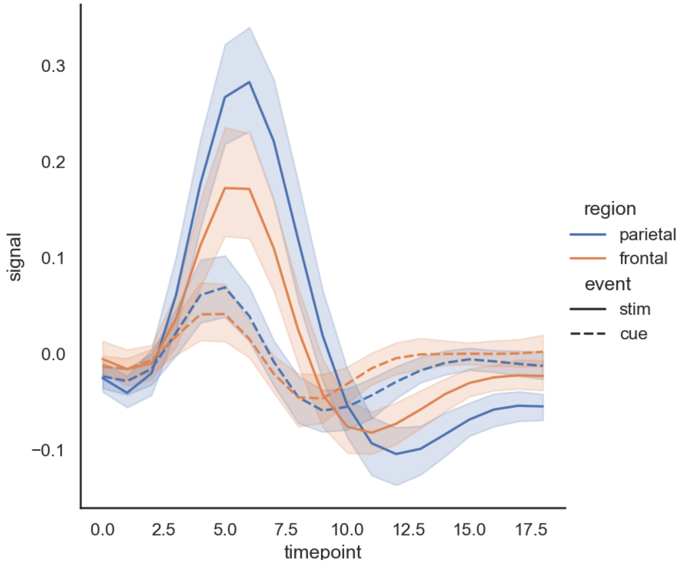


signal
0.25
0.20
0.15
0.10
0.05
0.00
−0.05
−0.10
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
timepoint
event
stim
cue

style semantic

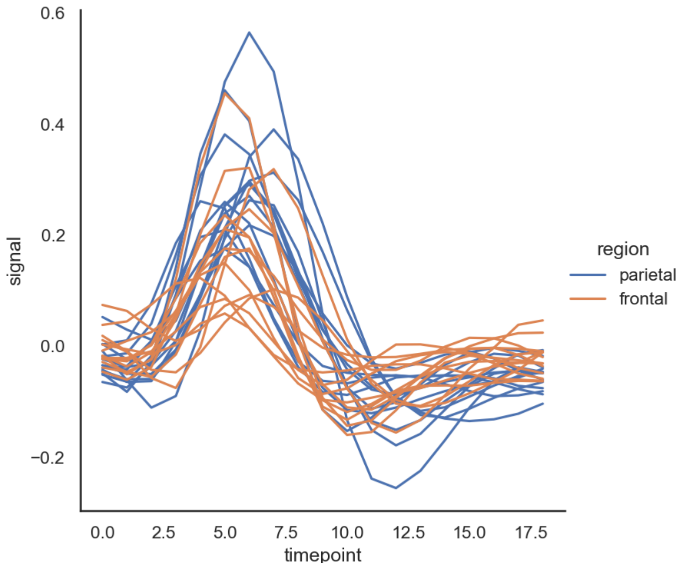
Adding a style semantic to a line plot changes the pattern of dashes in the line by default:

```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal",
    hue="region",style="event",kind="line",
    data=fmri);
plt.show()
```


signal
0.3
0.2
0.1
0.0
−0.1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
timepoint
region
parietal
frontal
event
stim
cue

When you are working with repeated measures data (that is, you have units that were sampled multiple times), you can also plot each sampling unit separately without distinguishing them through semantics. This avoids cluttering the legend:

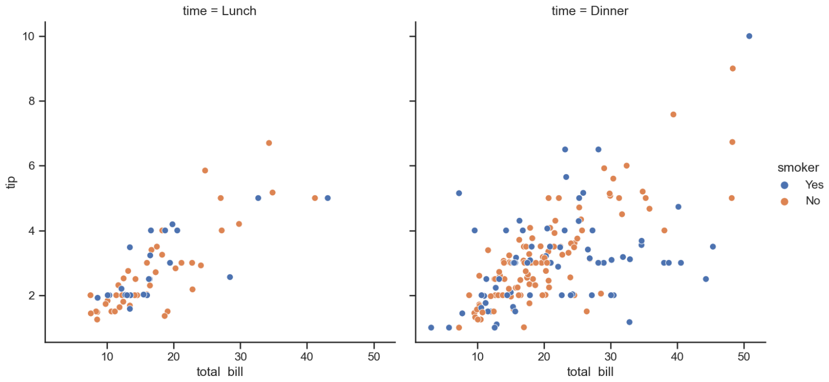
```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal",
    hue="region",units="subject",
    estimator=None,kind="line",
    data=fmri.query("event == 'stim'"));
plt.show()
```


0.6
0.4
0.2
0.0
−0.2
signal
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
timepoint
region
parietal
frontal

1-3 Showing multiple relationships with facets

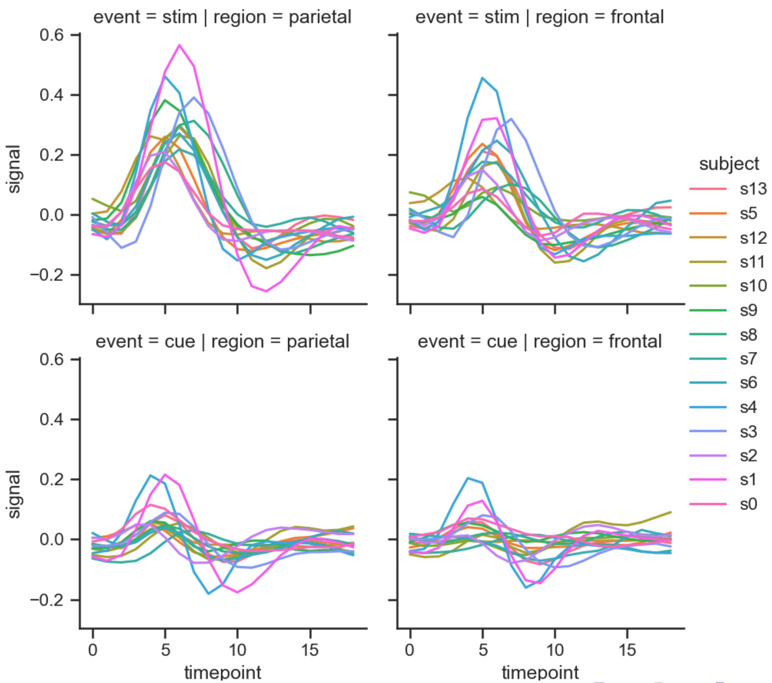
To show the influence of an additional variable, instead of assigning it to one of the semantic roles in the plot, use it to “facet” the visualization. This means that you make multiple axes and plot subsets of the data on each of them:

```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="total_bill", y="tip", hue="smoker",
            col="time", data=tips);
plt.show()
```


time = Lunch
time = Dinner
tip
total_bill
total_bill
10
8
6
4
2
10
20
30
40
50
smoker
Yes
No

You can also show the influence two variables this way: one by faceting on the columns and one by faceting on the rows. As you start adding more variables to the grid, you may want to decrease the figure size. Remember that the size FacetGrid is parameterized by the height and aspect ratio of each facet:

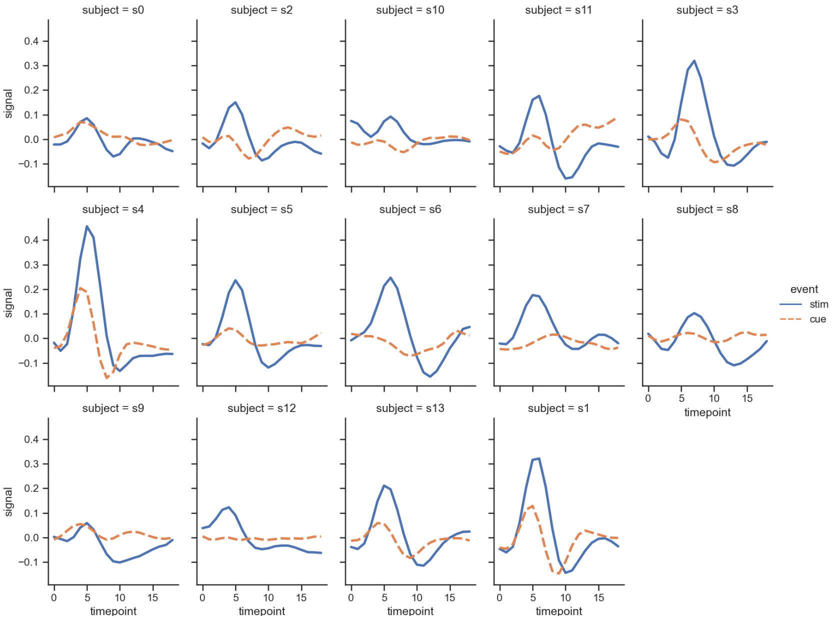
```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint",y="signal",hue="subject",
   col="region",row="event",height=3,
   kind="line",estimator=None,data=fmri);
plt.show()
```



event = stim | region = parietal
event = stim | region = frontal
event = cue | region = parietal
event = cue | region = frontal
signal
timepoint
0.6
0.4
0.2
0.0
−0.2
0
5
10
15
subject
s13
s5
s12
s11
s10
s9
s8
s7
s6
s4
s3
s2
s1
s0

When you want to examine effects across many levels of a variable, it can be a good idea to facet that variable on the columns and then “wrap” the facets into the rows:

```
import matplotlib.pyplot as plt
import seaborn as sns
fmri = sns.load_dataset("fmri")
sns.relplot(x="timepoint", y="signal",
  hue="event", style="event",
  col="subject", col_wrap=5,height=3,
  aspect=.75, linewidth=2.5,kind="line",
  data=fmri.query("region == 'frontal'"));
plt.show()
```



subject = s0
subject = s2
subject = s10
subject = s11
subject = s3
subject = s4
subject = s5
subject = s6
subject = s7
subject = s8
subject = s9
subject = s12
subject = s13
subject = s1
signal
timepoint
event
stim
cue

2-Visualizing distributions of data

2-Visualizing distributions of data

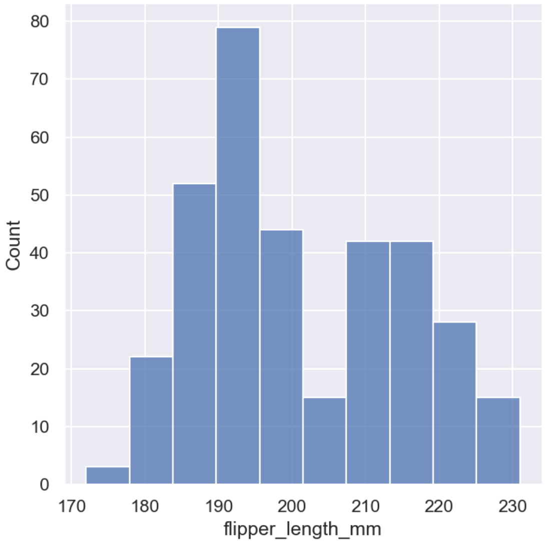
An early step in any effort to analyze or model data should be to understand how the variables are distributed. Techniques for distribution visualization can provide quick answers to many important questions. What range do the observations cover? What is their central tendency? Are they heavily skewed in one direction? Is there evidence for bimodality? Are there significant outliers? Do the answers to these questions vary across subsets defined by other variables?

The distributions module contains several functions designed to answer questions such as these. The axes-level functions are histplot(), kdeplot(), ecdfplot(), and rugplot(). They are grouped together within the figure-level displot(), jointplot(), and pairplot() functions.

Histograms

A histogram is a bar plot where the axis representing the data variable is divided into a set of discrete bins and the count of observations falling within each bin is shown using the height of the corresponding bar:

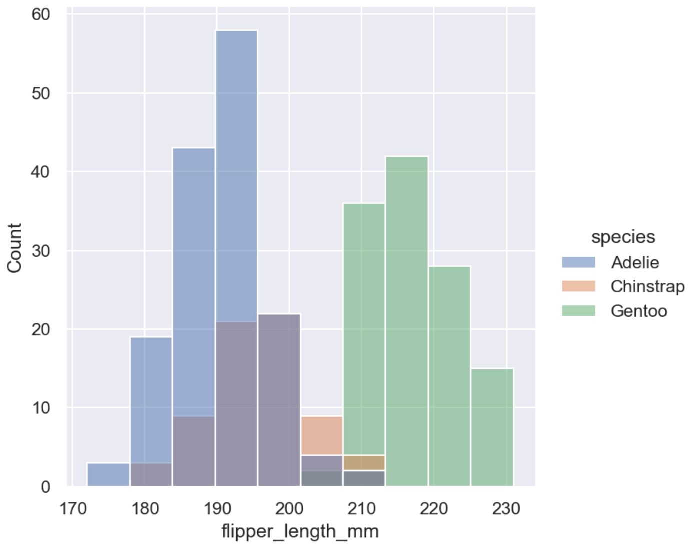
```python
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.displot(penguins, x="flipper_length_mm")
plt.show()
```


Count
80
70
60
50
40
30
20
10
0
170
180
190
200
210
220
230
flipper_length_mm

Conditioning on other variables

Once you understand the distribution of a variable, the next step is often to ask whether features of that distribution differ across other variables in the dataset. For example, what accounts for the bimodal distribution of flipper lengths that we saw above? displot() and histplot() provide support for conditional subsetting via the `hue` semantic. Assigning a variable to `hue` will draw a separate histogram for each of its unique values and distinguish them by color:

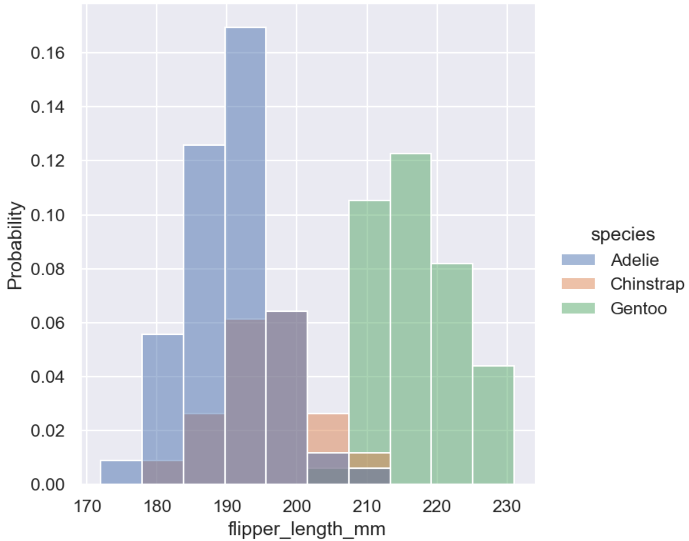
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.displot(penguins, x="flipper_length_mm",
      hue="species")
plt.show()
```


60
50
40
30
20
10
0
Count
170
180
190
200
210
220
230
flipper_length_mm
species
Adelie
Chinstrap
Gentoo

Normalized histogram statistics

Before we do, another point to note is that, when the subsets have unequal numbers of observations, comparing their distributions in terms of counts may not be ideal. One solution is to normalize the counts using the stat parameter:

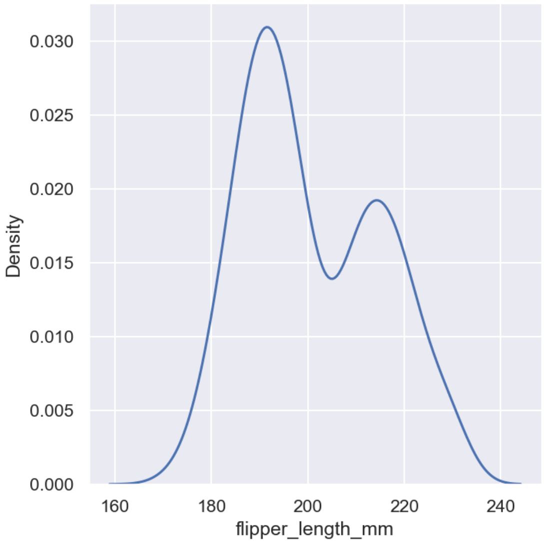
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.displot(penguins, x="flipper_length_mm",
   hue="species", stat="probability")
plt.show()
```


Probability
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
170
180
190
200
210
220
230
flipper_length_mm
species
Adelie
Chinstrap
Gentoo

Kernel density estimation

A histogram aims to approximate the underlying probability density function that generated the data by binning and counting observations. Kernel density estimation (KDE) presents a different solution to the same problem. Rather than using discrete bins, a KDE plot smooths the observations with a Gaussian kernel, producing a continuous density estimate:

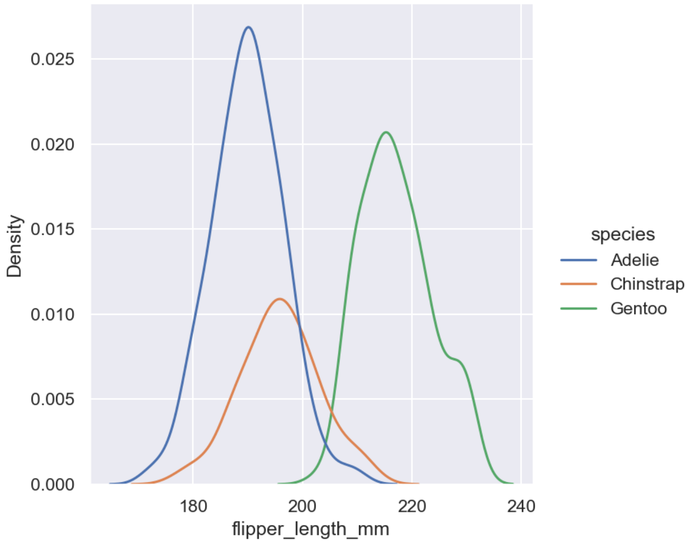
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.displot(penguins, x="flipper_length_mm",
     kind="kde")
plt.show()
```



0.030
0.025
0.020
0.015
0.010
0.005
0.000
Density
160
180
200
220
240
flipper_length_mm

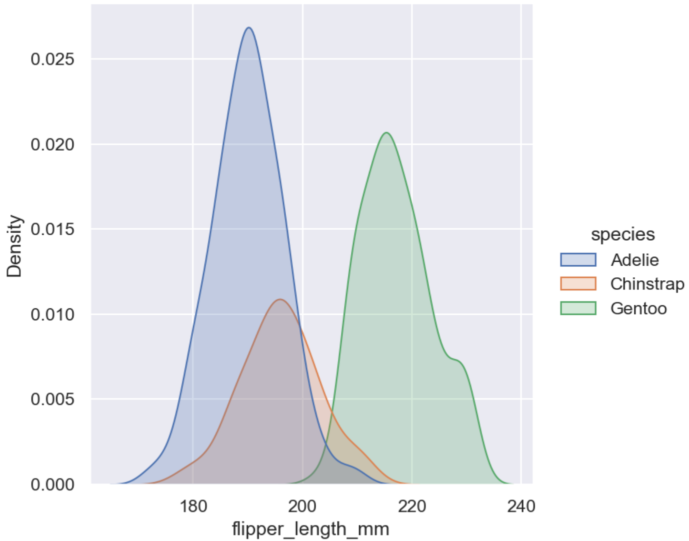
Conditioning on other variables

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.displot(penguins, x="flipper_length_mm",
    hue="species", kind="kde")
plt.show()
```


0.025
0.020
0.015
0.010
0.005
0.000
Density
180
200
220
240
flipper_length_mm
species
Adelie
Chinstrap
Gentoo

Conditioning on other variables

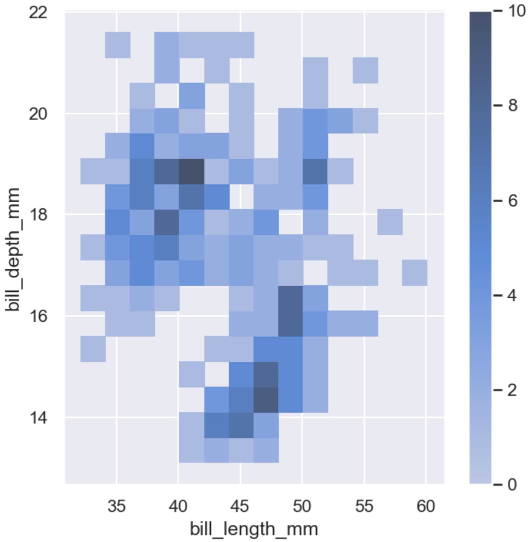
```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.displot(penguins, x="flipper_length_mm",
     hue="species", kind="kde", fill=True)
plt.show()
```


0.025
0.020
0.015
0.010
0.005
0.000
Density
180
200
220
240
flipper_length_mm
species
Adelie
Chinstrap
Gentoo

Visualizing bivariate distributions

All of the examples so far have considered univariate distributions: distributions of a single variable, perhaps conditional on a second variable assigned to hue. Assigning a second variable to y, however, will plot a bivariate distribution:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.displot(penguins, x="bill_length_mm",
 y="bill_depth_mm", binwidth=(2, .5), cbar=True)
plt.show()
```

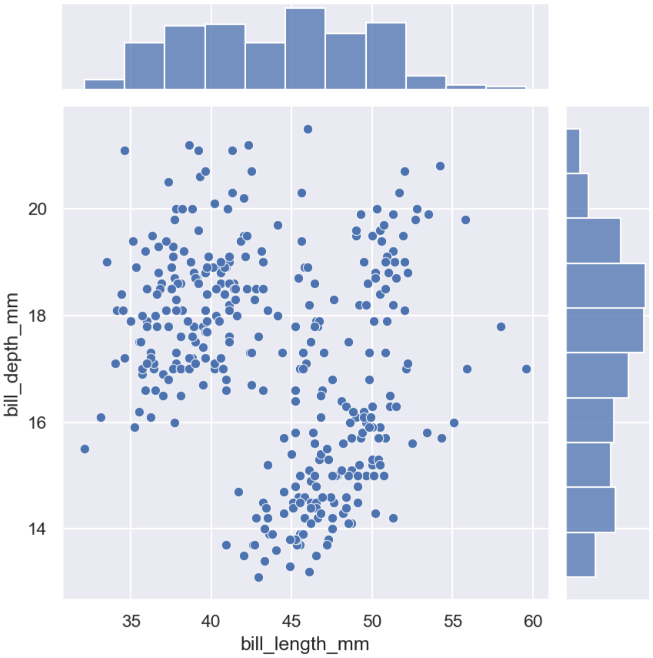


22
20
18
16
14
bill_depth_mm
35
40
45
50
55
60
bill_length_mm
10
8
6
4
2
0

Plotting joint and marginal distributions

The first is jointplot(), which augments a bivariate relatonal or distribution plot with the marginal distributions of the two variables. By default, jointplot() represents the bivariate distribution using scatterplot() and the marginal distributions using histplot():

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.jointplot(data=penguins, x="bill_length_mm",
    y="bill_depth_mm")
plt.show()
```

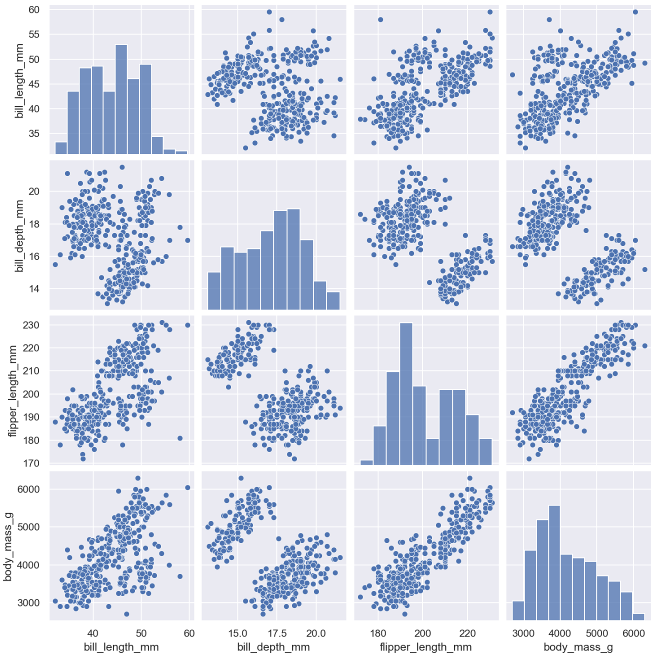


20
18
16
14
bill_depth_mm
35
40
45
50
55
60
bill_length_mm

Plotting many distributions

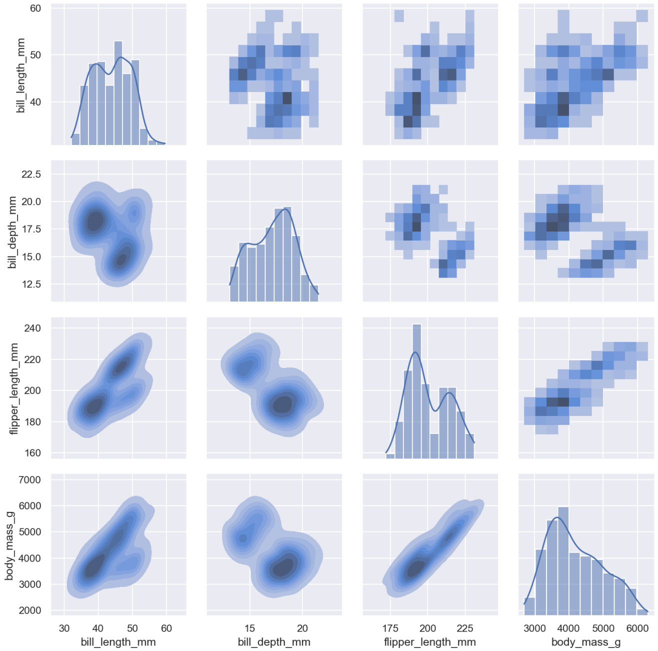
The pairplot() function offers a similar blend of joint and marginal distributions. Rather than focusing on a single relationship, however, pairplot() uses a “small-multiple” approach to visualize the univariate distribution of all variables in a dataset along with all of their pairwise relationships:

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
sns.pairplot(penguins)
plt.show()
```


bill_length_mm
bill_depth_mm
flipper_length_mm
body_mass_g
60
55
50
45
40
35
20
18
16
14
230
220
210
200
190
180
170
6000
5000
4000
3000
40
50
60
15.0
17.5
20.0
180
200
220
3000
4000
5000
6000

Plotting many distributions

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_theme(style="darkgrid")
penguins = sns.load_dataset("penguins")
g = sns.PairGrid(penguins)
g.map_upper(sns.histplot)
g.map_lower(sns.kdeplot, fill=True)
g.map_diag(sns.histplot, kde=True)
plt.show()
```



bill_length_mm
60
50
40
bill_depth_mm
22.5
20.0
17.5
15.0
12.5
flipper_length_mm
240
220
200
180
160
body_mass_g
7000
6000
5000
4000
3000
2000
30
40
50
60
bill_length_mm
15
20
bill_depth_mm
175
200
225
flipper_length_mm
3000
4000
5000
6000
body_mass_g

3-Plotting with categorical data

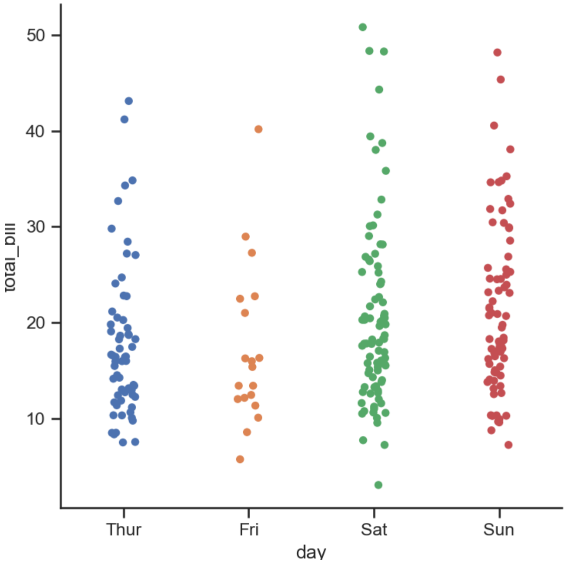
Plotting with categorical data

In the relational plot tutorial we saw how to use different visual representations to show the relationship between multiple variables in a dataset. In the examples, we focused on cases where the main relationship was between two numerical variables. If one of the main variables is **categorical** (divided into discrete groups) it may be helpful to use a more specialized approach to visualization.
There are a number of axes-level functions for plotting categorical data in different ways and a figure-level interface, **catplot()**, that gives unified higher-level access to them.

3-1 Categorical scatterplots

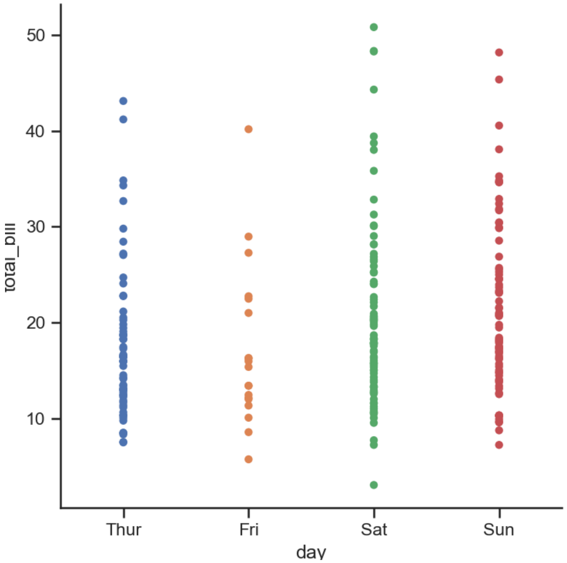
The default representation of the data in **catplot()** uses a scatterplot. There are actually two different categorical scatter plots in seaborn. The approach used by **stripplot()**, which is the default **kind** in **catplot()** is to adjust the positions of points on the categorical axis with a small amount of random **jitter**:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
sns.catplot(x="day", y="total_bill", data=tips)
plt.show()
```


total_bill
50
40
30
20
10
Thur
Fri
Sat
Sun
day

The **jitter** parameter controls the magnitude of jitter or disables it altogether:

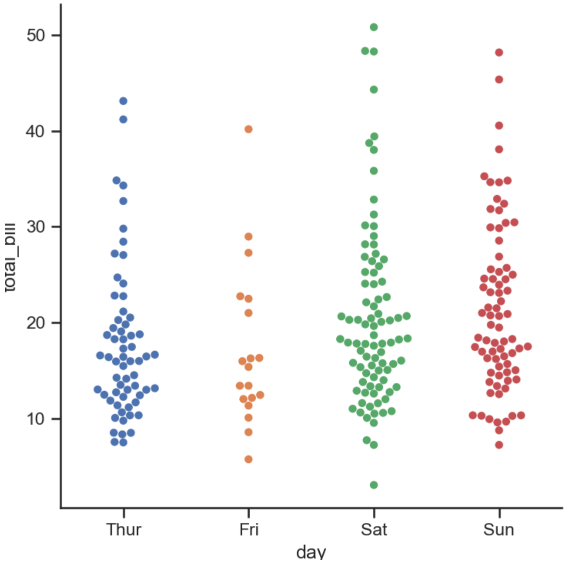
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
sns.catplot(x="day", y="total_bill",
   jitter=False, data=tips)
plt.show()
```


50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day

swarmplot()

The second approach adjusts the points along the categorical axis using an algorithm that prevents them from overlapping. It can give a better representation of the distribution of observations, although it only works well for relatively small datasets. This kind of plot is sometimes called a **beeswarm** and is drawn in seaborn by **swarmplot()**, which is activated by setting **kind=”swarm** in **catplot()**:

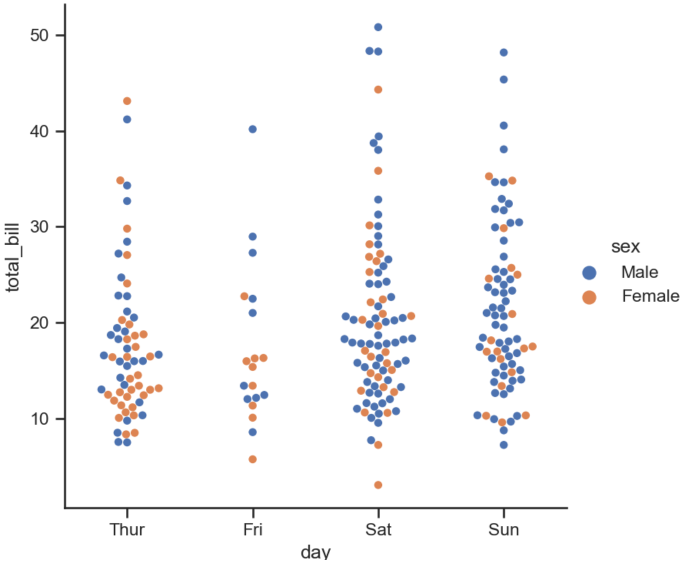
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
sns.catplot(x="day", y="total_bill",
    kind="swarm", data=tips)
plt.show()
```



50
40
30
20
10
total_bill
Thur
Fri
Sat
Sun
day

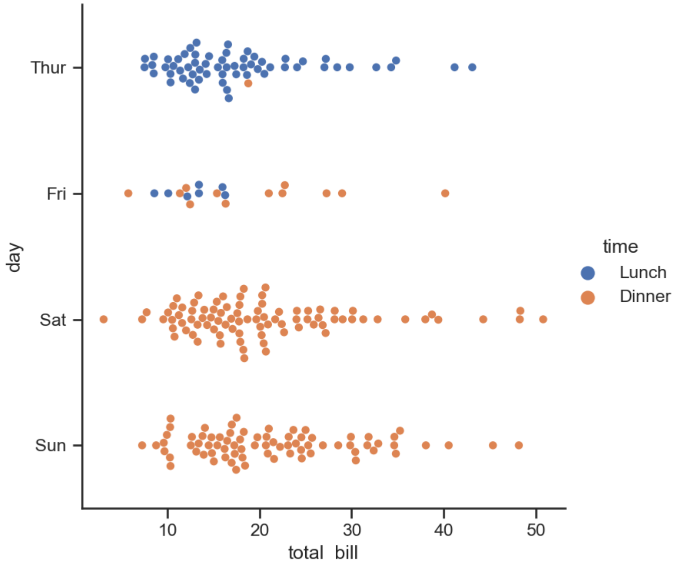
Using a hue semantic

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
sns.catplot(x="day", y="total_bill",
    kind="swarm", data=tips)
plt.show()
```


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

Swap the assignment of variables to axes

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
sns.catplot(x="total_bill", y="day", hue="time",
    kind="swarm", data=tips)
plt.show()
```

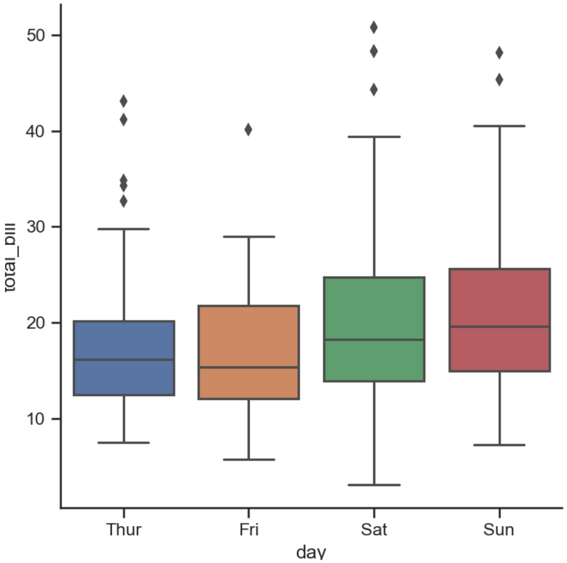

Thur
Fri
Sat
Sun
day
10
20
30
40
50
total bill
time
Lunch
Dinner

3-2 Distributions of observations within categories

As the size of the dataset grows, categorical scatter plots become limited in the information they can provide about the distribution of values within each category. When this happens, there are several approaches for summarizing the distributional information in ways that facilitate easy comparisons across the category levels.

Boxplots

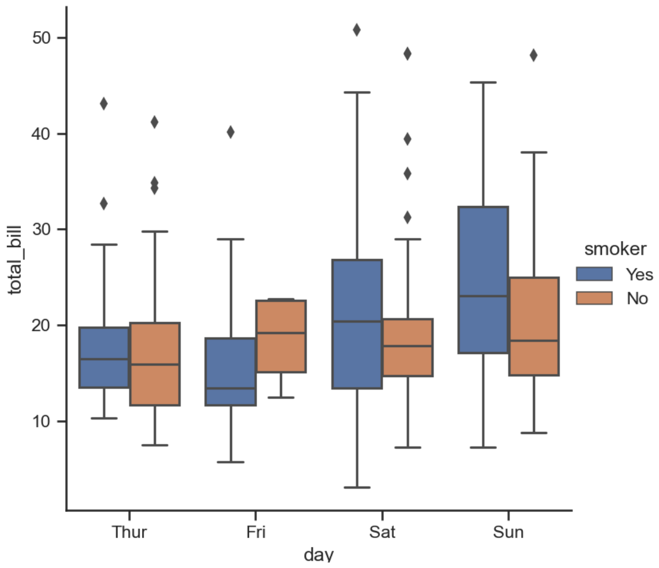
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
sns.catplot(x="day", y="total_bill",
    kind="box", data=tips)
plt.show()
```


total_bill
50
40
30
20
10
Thur
Fri
Sat
Sun
day

hue semantic

When adding a hue semantic, the box for each level of the semantic variable is moved along the categorical axis so they don't overlap:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
tips["weekend"] = tips["day"].isin(["Sat", "Sun"])
sns.catplot(x="day", y="total_bill", hue="weekend",
            kind="box", dodge=False, data=tips)
plt.show()
```

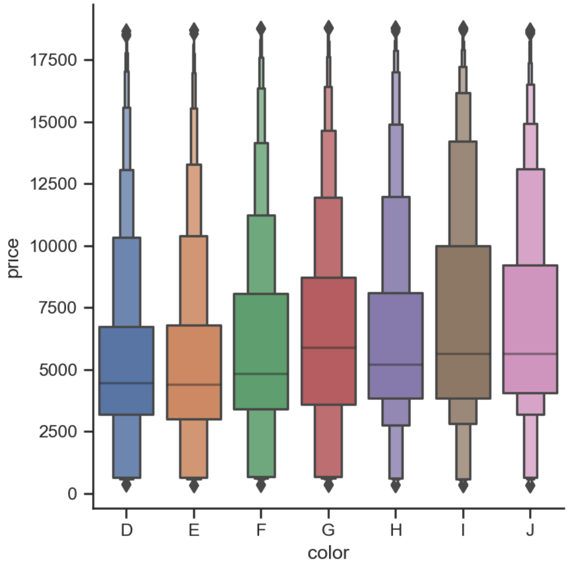


total_bill
50
40
30
20
10
Thur
Fri
Sat
Sun
day
smoker
Yes
No

boxenplot()

A related function, **boxenplot()**, draws a plot that is similar to a box plot but optimized for showing more information about the shape of the distribution. It is best suited for larger datasets:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
diamonds = sns.load_dataset("diamonds")
sns.catplot(x="color", y="price", kind="boxen",
            data=diamonds.sort_values("color"))
plt.show()
```

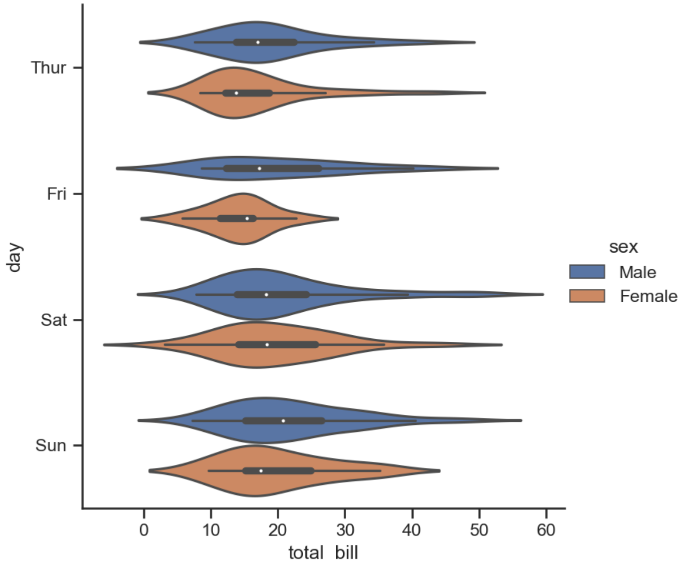


price
17500
15000
12500
10000
7500
5000
2500
0
D
E
F
G
H
I
J
color

Violinplots

A different approach is a violinplot(), which combines a boxplot with the kernel density estimation procedure described in the distributions tutorial. This approach uses the kernel density estimate to provide a richer description of the distribution of values:

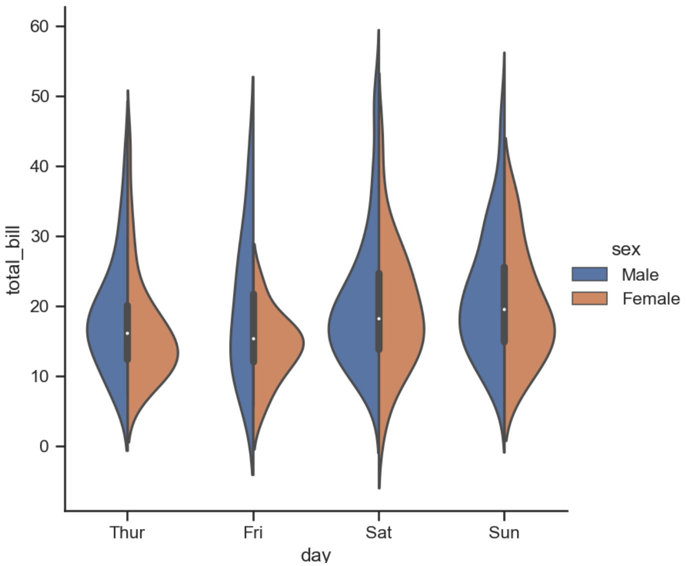
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
sns.catplot(x="total_bill", y="day", hue="sex",
            kind="violin", data=tips)
plt.show()
```


Thur
Fri
Sat
Sun
day
0
10
20
30
40
50
60
total bill
sex
Male
Female

split the violins

It's also possible to "split" the violins when the hue parameter has only two levels, which can allow for a more efficient use of space:

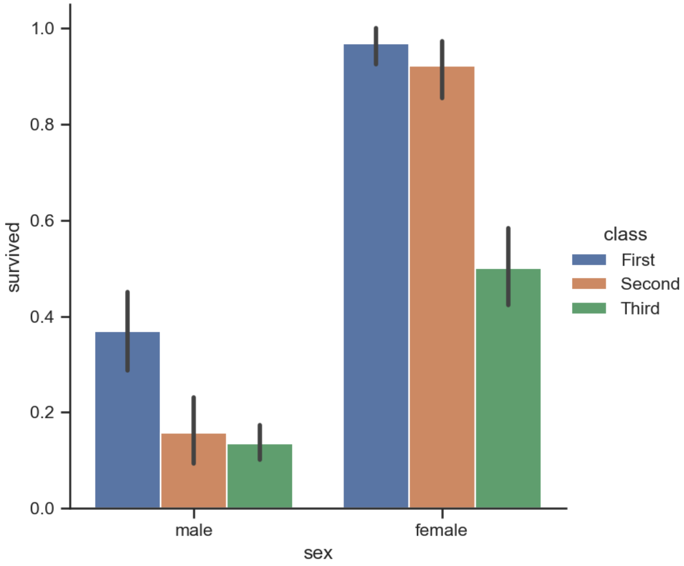
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
tips = sns.load_dataset("tips")
sns.catplot(x="total_bill", y="day", hue="sex",
            kind="violin", data=tips)
plt.show()
```


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

Bar plots

In seaborn, the barplot() function operates on a full dataset and applies a function to obtain the estimate (taking the mean by default). When there are multiple observations in each category, it also uses bootstrapping to compute a confidence interval around the estimate, which is plotted using error bars:

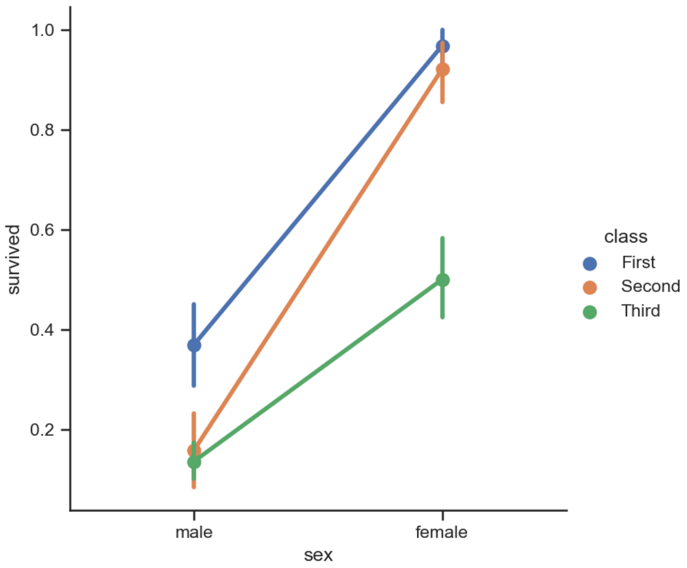
```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
titanic = sns.load_dataset("titanic")
sns.catplot(x="sex", y="survived", hue="class",
    kind="bar", data=titanic)
plt.show()
```


1.0
0.8
0.6
0.4
0.2
0.0
survived
male
female
sex
class
First
Second
Third

Point plots

An alternative style for visualizing the same information is offered by the **pointplot()** function. This function also encodes the value of the estimate with height on the other axis, but rather than showing a full bar, it plots the point estimate and confidence interval. Additionally, **pointplot()** connects points from the same hue category. This makes it easy to see how the main relationship is changing as a function of the **hue** semantic, because your eyes are quite good at picking up on differences of slopes:

```
import seaborn as sns
import matplotlib.pyplot as plt
sns.set_theme(style="ticks", color_codes=True)
titanic = sns.load_dataset("titanic")
sns.catplot(x="sex", y="survived", hue="class",
    kind="point", data=titanic)
plt.show()
```


1.0
0.8
0.6
0.4
0.2
survived
male
female
sex
class
First
Second
Third