Data Visualization with SEABORN

`Seaborn` is a statistical plotting library in python. It is built upon `Matplotlib` and has a lot of customization options. I'll try to cover all the things I know about it. You can also find a small introduction to `matplotlib` in this `repository`.

- LinkedIn
- YouTube
- gtihub
- Gmail
- discord

Installation

If you have `conda` installed, Just run the following command in your terminal:

```
conda install seaborn
```

Or if you dont't have `conda` installed, You can also run `pip install seaborn` in your terminal but you must need `matplotlib` already installed.

Introduction

`Seaborn` is my favorite library for `data visualization`. It is a very powerful tool that can create visually stunning plots in a single command and it's much faster to code in. You can go to the official website to learn more about it. There are a lot of modules built into `seaborn` and I'll try to go over those one by one.

Distribution Plot

A `Distribution Plot` is a plot that shows the distribution of a single variable. It is a very useful tool for understanding the distribution of a variable.

I think the best way would be to show you using real data set.

`Seaborn` has a built in plethora of `datasets` that we can use. We can see them by running the following command:

In [2]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

In [3]:
```
sns.get_dataset_names()
```

Out[3]:
```
['anagrams',
 'anscombe',
 'attention',
 'brain_networks',
 'car_crashes',
 'diamonds',
 'dots',
 'dowjones',
 'exercise',
 'flights',
 'fmri',
 'geyser',
 'glue',
 'healthexp',
 'iris',
 'mpg',
 'penguins',
 'planets',
 'seaice',
 'taxis',
 'tips',
 'titanic']
```

WOH! We have a lot of datasets. Each of thses datasets has different features and purposes.

So, let's load a data set.

In [4]:
```
data = sns.load_dataset('tips')
data.head()
```

Out[4]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

The `tips` dataset has data of a restaurant bills and tips.

Let's see some more infomation of this dataset.

In [5]:
```
data.describe()
```

Out[5]:

	total_bill	tip	size
count	244.000000	244.000000	244.000000
mean	19.785943	2.998279	2.569672
std	8.902412	1.383638	0.951100
min	3.070000	1.000000	1.000000
25%	13.347500	2.000000	2.000000
50%	17.795000	2.900000	2.000000
75%	24.127500	3.562500	3.000000
max	50.810000	10.000000	6.000000

In [6]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype   
---  ------      --------------  -----   
 0   total_bill  244 non-null    float64 
 1   tip         244 non-null    float64 
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64   
dtypes: category(4), float64(2), int64(1)
memory usage: 7.4 KB
```

As, you can see the data set has `244` rows and `7` columns like `total_bill`, `tip`, `sex`, `smoker`, `day`, `time`, `size` etc.

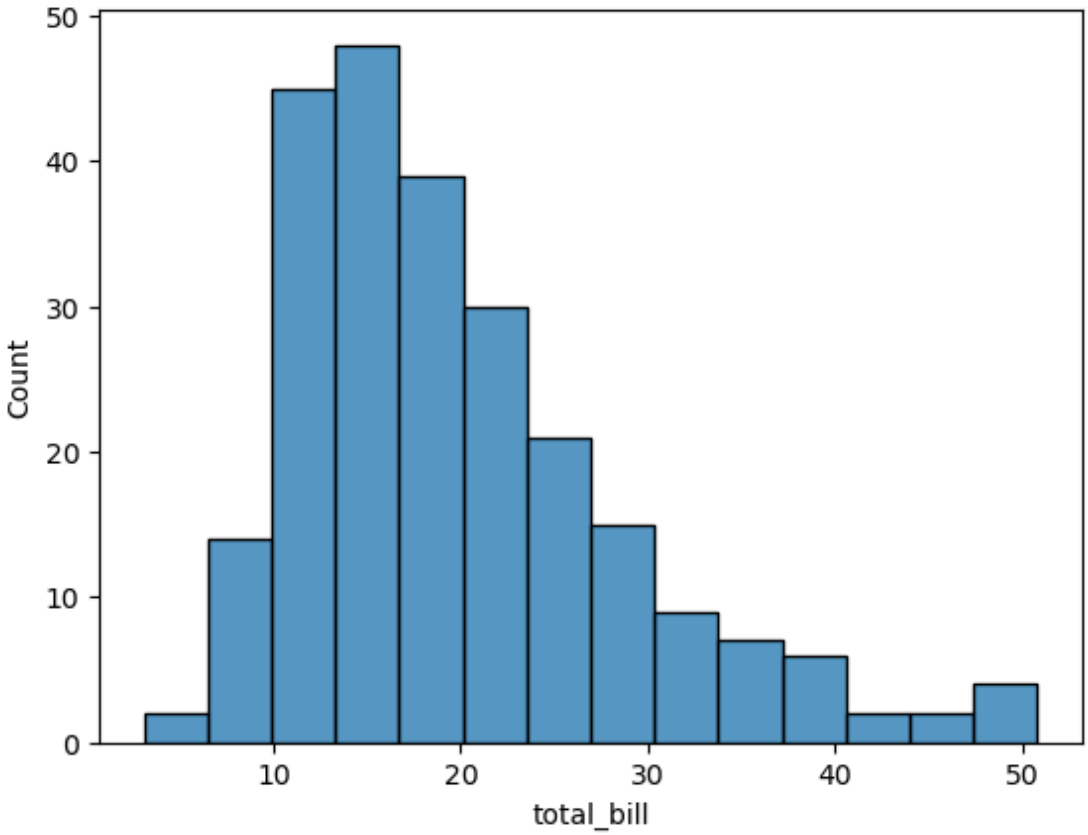
Now, we can use this dataset to visualize some data distributions.

Let's say I want to see the distribution of the `total_bill` column.

Histogram

In [7]:
```
sns.histplot(data=data, x='total_bill')
```

Out[7]:
```
<Axes: xlabel='total_bill', ylabel='Count'>
```



Above you can see the distribution of the `total_bill` column and if you look close, the `x axis` is the `total_bill` column and the `y axis` is the `count` of the `total_bill` column.

So, this histogram is explaining the `total_bill` count. We can see that the highest concentration is around `10` to `20`. This means that most of the `total_bill` are between `10` to `20`.

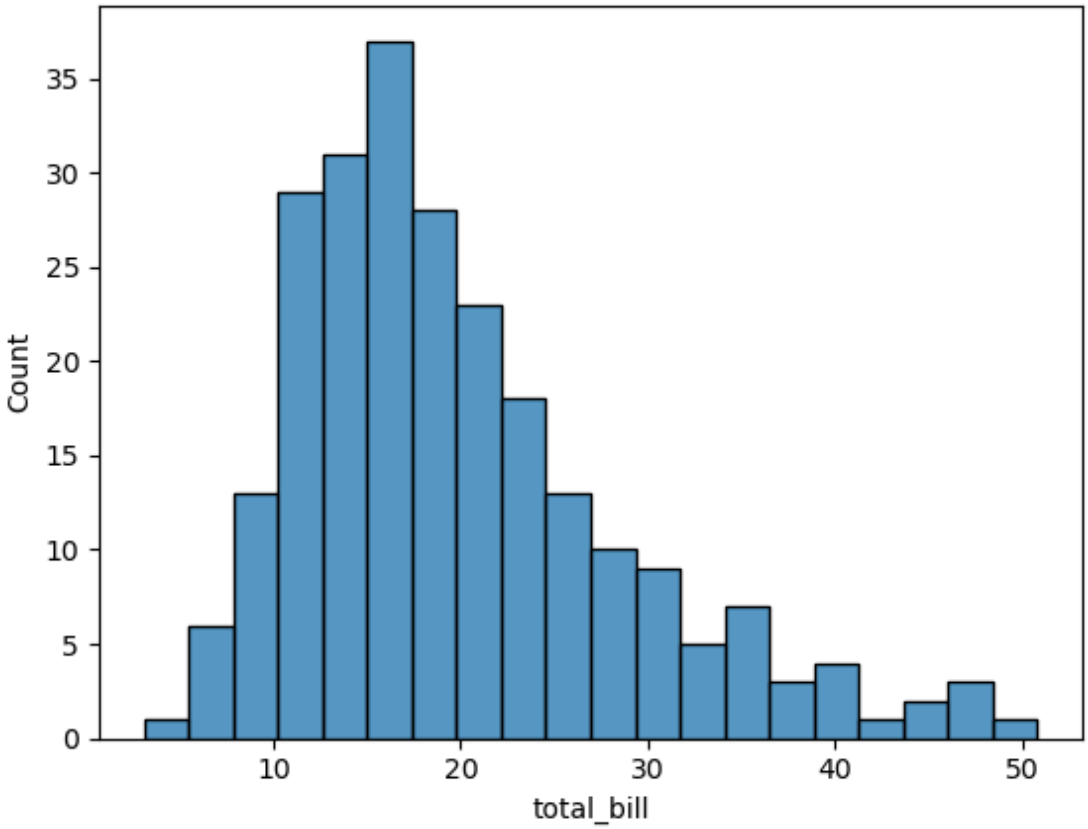
If you want to be more specific, you can specify the `bins` of the histogram.

I think showing it would be much more helpful.

> `Bins` refer to the number of bars in a histogram. By default it is set to `auto`. which means the number of bins will be determined internally.

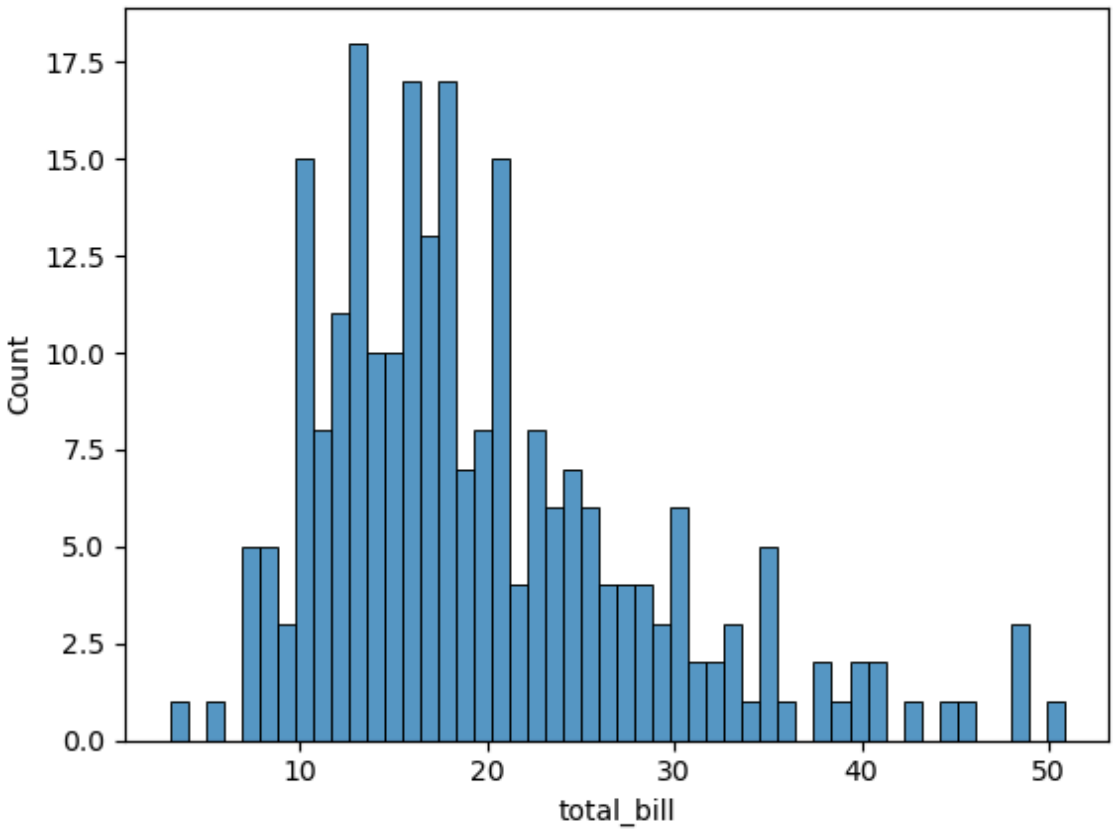
In [8]:
```
sns.histplot(data=data, x='total_bill', bins=20)
```

Out[8]:
```
<Axes: xlabel='total_bill', ylabel='Count'>
```



In [9]:
```
sns.histplot(data=data, x='total_bill', bins=50)
```

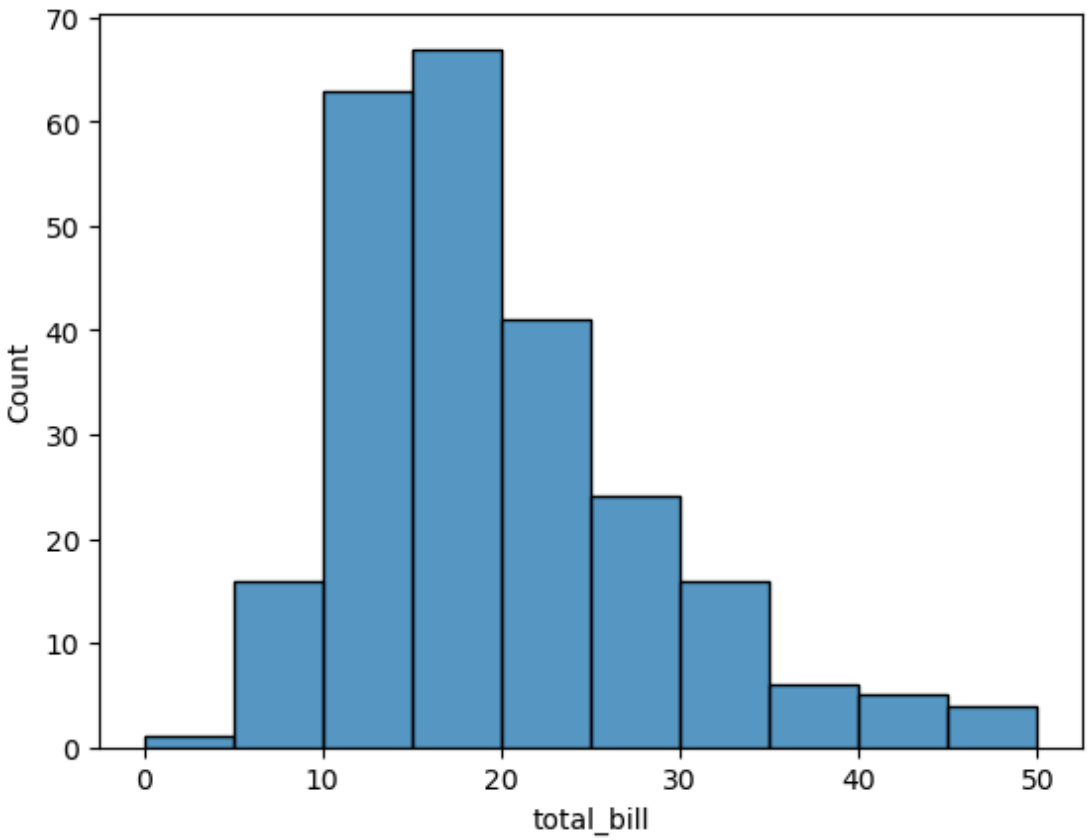
Out[9]:
```
<Axes: xlabel='total_bill', ylabel='Count'>
```



We can pass different types of arguments to the bins argument. For example, we can pass a `list` of values.

In [10]:
```
bins = [0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
sns.histplot(data=data, x='total_bill', bins=bins)
```

Out[10]:
```
<Axes: xlabel='total_bill', ylabel='Count'>
```



Now, as I set `bins` to a list the range of the bars for the histogram will be set to the values in the list.

And we have `arguments` like `binrange`, `binwidth`, `discrete` etc that allows for much more customization of the histogram.

We will talk more about histograms later on.

But One thing I want to show is a `jointplot`.

Joint Plot

`Joint Plot` is another distribution plot that shows the distribution of two variables.

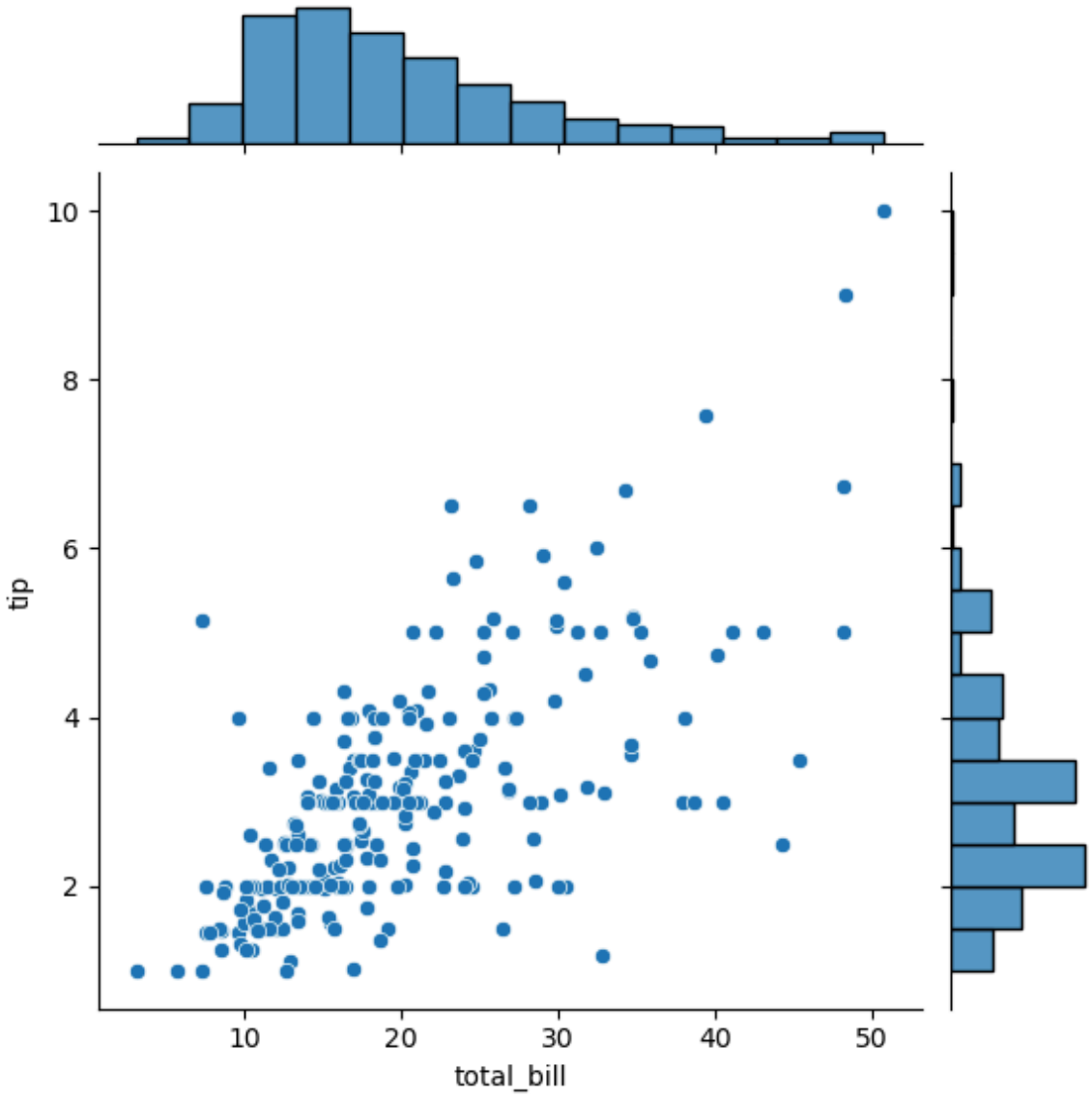
Let's say I want to see the distribution of the `total_bill` and `tip` columns.

In [11]:
```
data.columns
```

Out[11]:
```
Index(['total_bill', 'tip', 'sex', 'smoker', 'day', 'time', 'size'], dtyp
e='object')
```

In [12]:
```
sns.jointplot(data=data, x='total_bill', y='tip')
```

Out[12]:
```
<seaborn.axisgrid.JointGrid at 0x7ce687bfcb90>
```



And this looks like a `scatter plot`.

> A scatter plot show the relationship between two variables with dots.

If you look at the `jointplot`, you can see that the `x axis` is the `total_bill` column and the `y axis` is the `tip` column and on the top and the right side the `histogram` of the `total_bill` and `tip` columns are shown.

And this give us perspective of the relationship between the two variables.

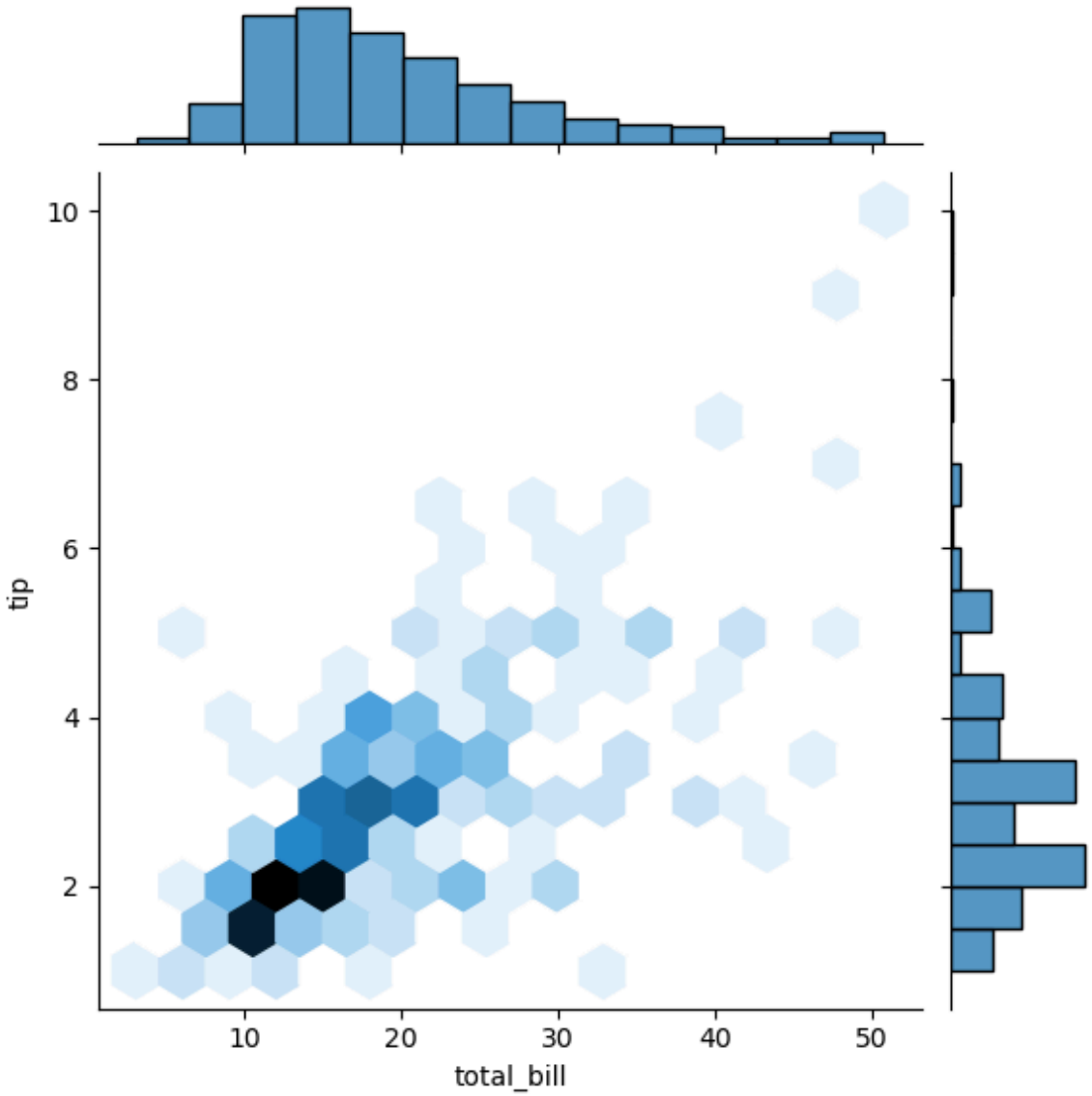
From, the `jointplot` we can somewhat come to a understanding that people tend to tip `2 to 4` dollers has a total bill in the range of `10 to 20` dollars.

This can become more clear if we do something called `hex plot`.

We can transform this jointplot into a `hex plot` by setting the `kind` argument to `hex`

In [13]:
```
sns.jointplot(data=data, x='total_bill', y='tip', kind='hex')
```

Out[13]:
```
<seaborn.axisgrid.JointGrid at 0x7ce687bdca50>
```



Now, we can see something like a `beehive structure` and every hex gon has a range that changes it color from `light` to `dark` with how many dots it has in its range.

So, now we can clearly see that most of the people who has a total bill in the range of `10 to 20` dollars tend to tip `2 to 4` dollars.

But doing this for every column can be a `hastle`. It is the suty of a data analyst to understand the data and relations between every column. But as the columns grows, Making a `jointplot` for every column can become a bit too much so we can use `pairplot` instead.

Pairplot

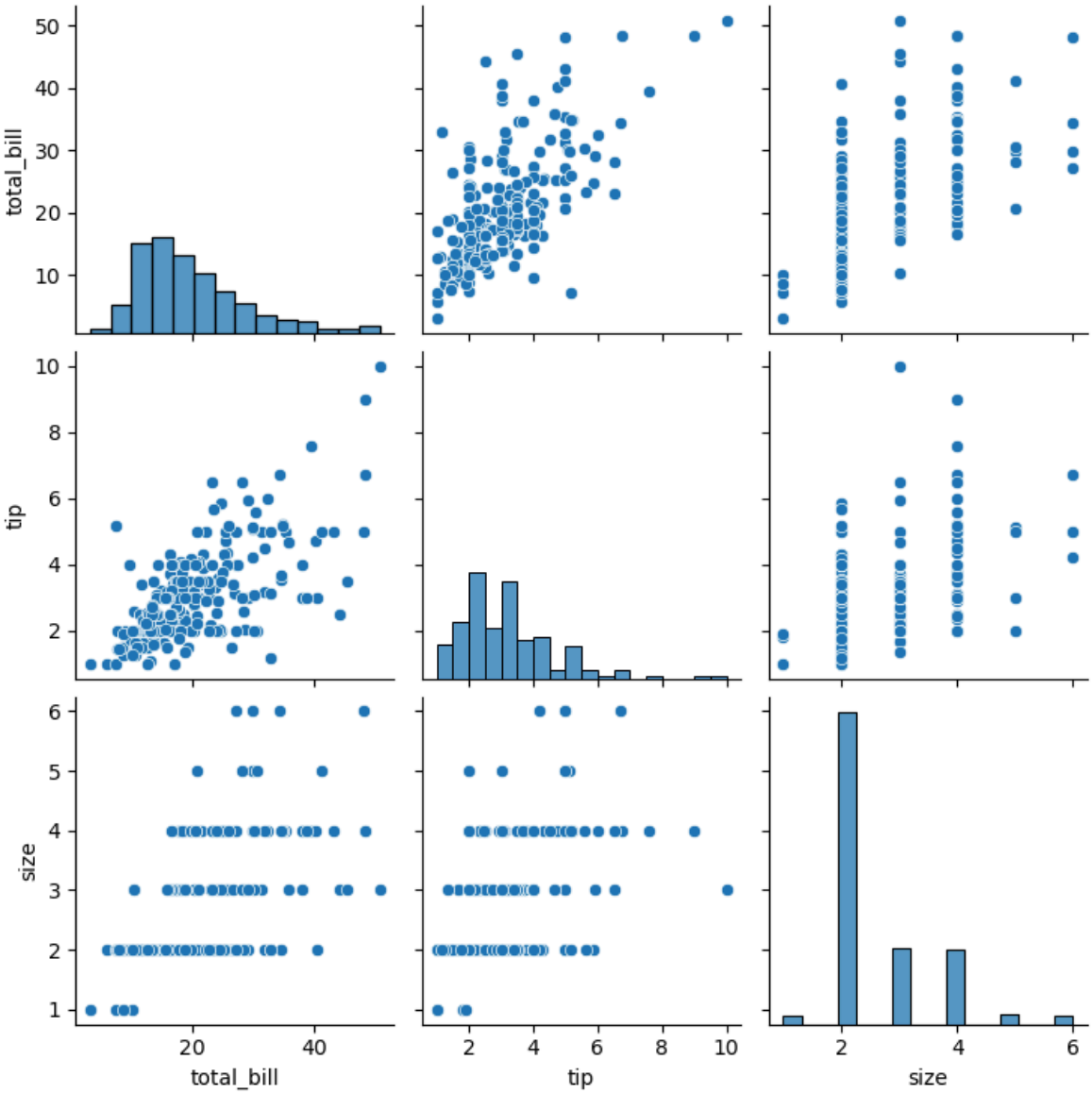
What is a `pairplot`?

It is not a plot but a `collection` of plots. It is a plot that shows the relationship between all the columns in the dataset.

> All the numeric columns at least.

```
In [14]: sns.pairplot(data=data)
```

```
Out[14]: <seaborn.axisgrid.PairGrid at 0x7ce68783bc50>
```



Above you can see a `pairplot` of the `tips` dataset.

Here, we can see the relationship between all the columns in the dataset in the form of a scatter plot.

And also the count distribution of each column know as a `histogram` on the diagonal.

This is very usefull in case of exploring the dataset quickly. But sometimes it can get a little slow if the dataset is very large or has a lot of columns.

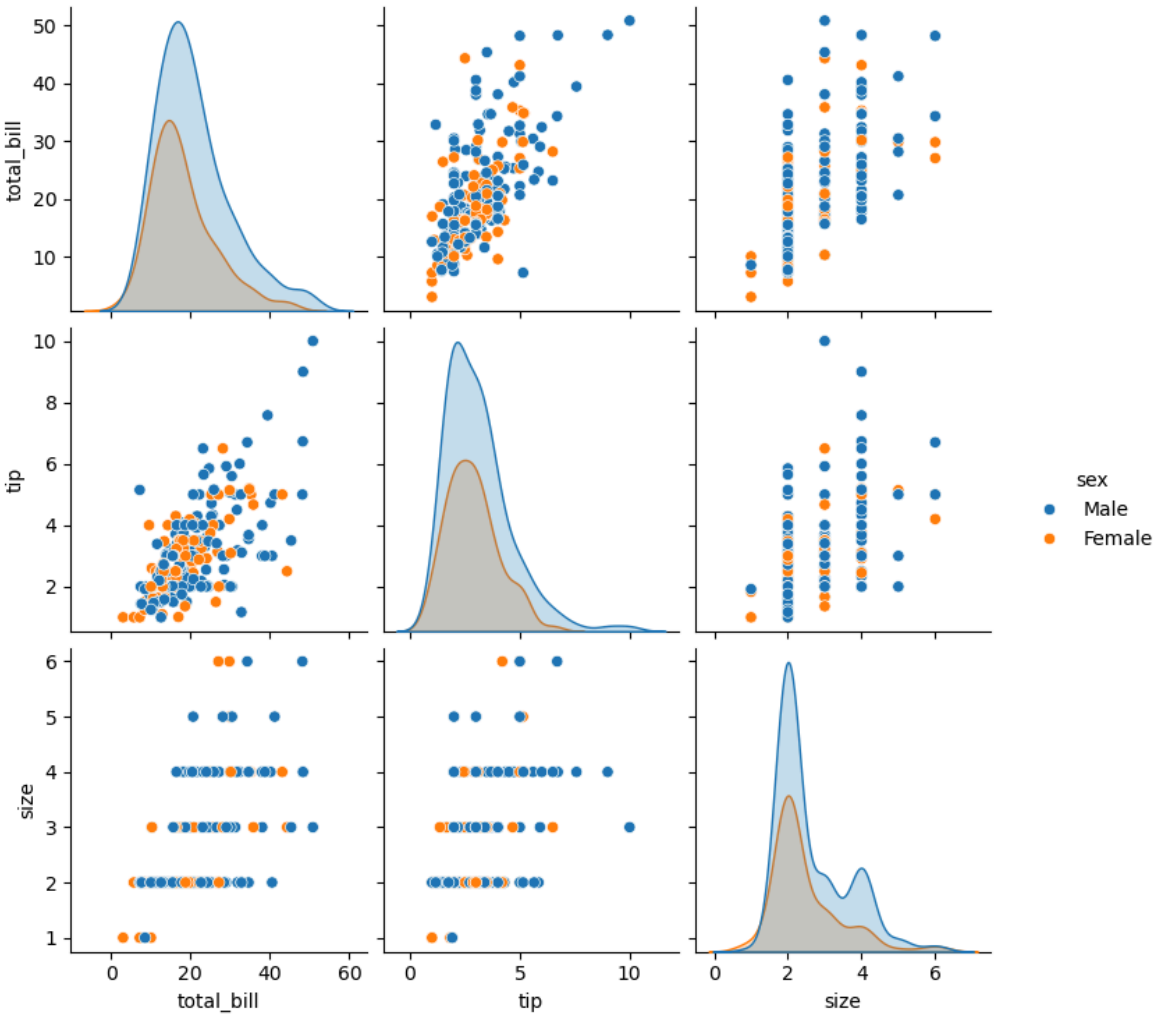
Not only plotting out the whole datasets column by column but also we can view the relations between the column from a `categorical` perspective.

Let's say you want to know the distribution of the `sex` column for all the numeric columns.

> You can set a hue argument where you can pass a column name that is categorical and then the `pairplot` will show the distribution of the numeric columns for each category in the `sex` column.

In [15]:
```
sns.pairplot(
    data=data,
    hue='sex'
)
```

Out[15]:
```
<seaborn.axisgrid.PairGrid at 0x7ce6872f2690>
```



And now, we can see the distribution of the numeric columns for each category in the `sex` column.

In the right side you can see the indicator for each category in the `sex` column.

> Blue for male and orange for female.

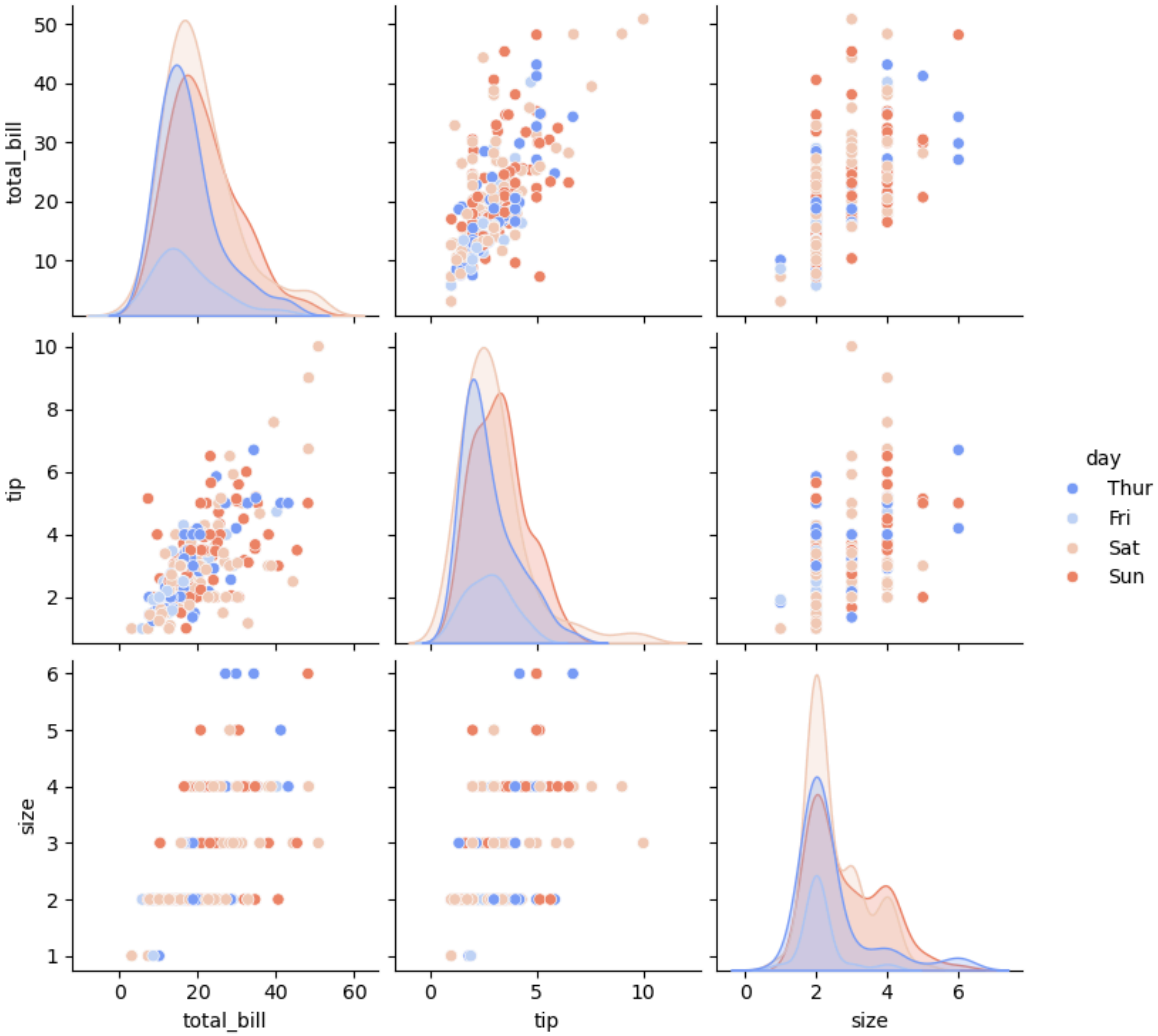
And it is all internally handled by `pairplot` so you don't have to worry about it.

Also, you can choose a color scheme for the indicators and the plot by passing `palette` argument.

In [16]:
```
sns.pairplot(
    data=data,
    hue='day',
```

```
    palette='coolwarm'
)
```

Out[16]: <seaborn.axisgrid.PairGrid at 0x7ce6856a2f90>



Seaborn give you a nice collection of palettes that you can use.

- `coolwarm`.
- `viridis`.
- `Dark2`. (My favorite)
- `Plasma`. (Also my favorite)

You can find a list of almost all the color palettes here

So, check it out and find one that you like and use it in your `pairplot` or `jointplot`.

OOOOOOOOR,

You can just set the Palette of your whole `notebook` by using the following code:

In [17]:
```
sns.set_palette('Dark2')
```

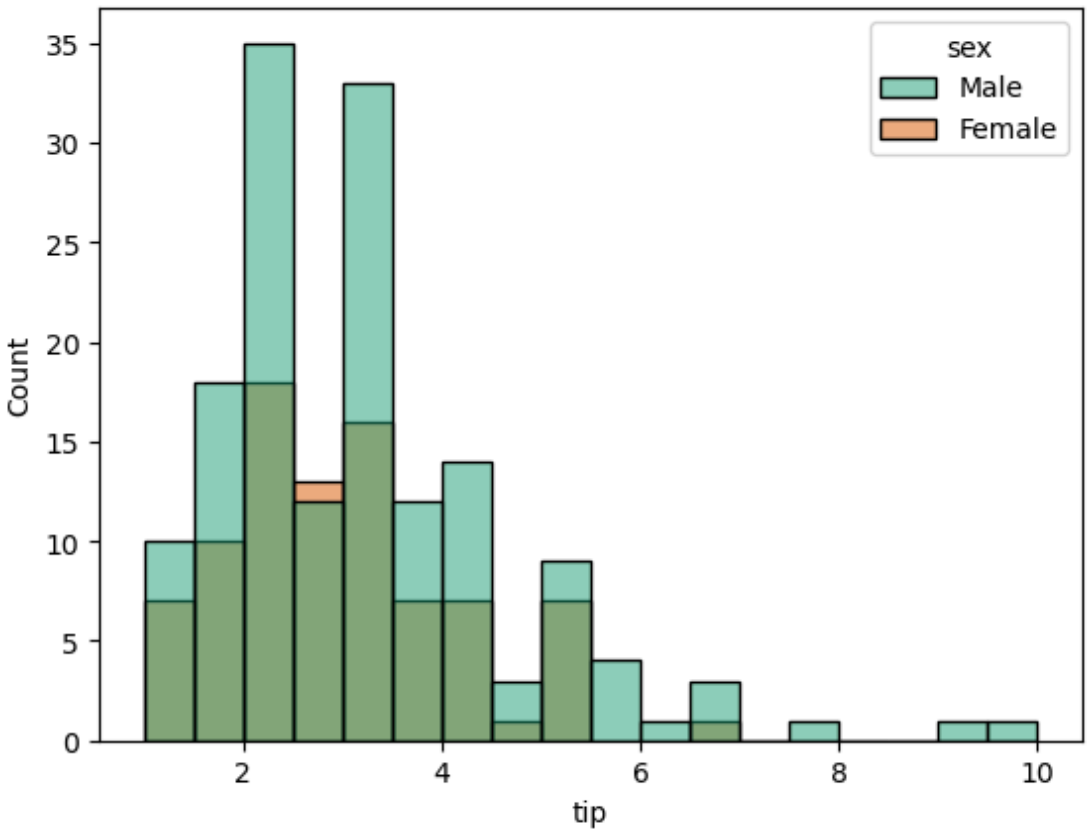
Now, you don't need to pass the `palette` argument in your `pairplot` or `jointplot` anymore.

By default the palette is set.

Now, we can do a `histplot` on the `Tip` column with the hue set to `sex` column.

In [18]:
```
sns.histplot(
    data=data,
    x='tip',
    hue='sex',
)
```

Out[18]:
```
<Axes: xlabel='tip', ylabel='Count'>
```



> You should set the palette at the very beginning of your `notebook` so that it is set for all the plots.

Now, let's move on to another plot called `kdeplot`.

KDE Plot

First, I want to tell you something.

Come close.

Closer.

Boo!

Scared ya!

`KDE` stands for `Kernel Density Estimation`.

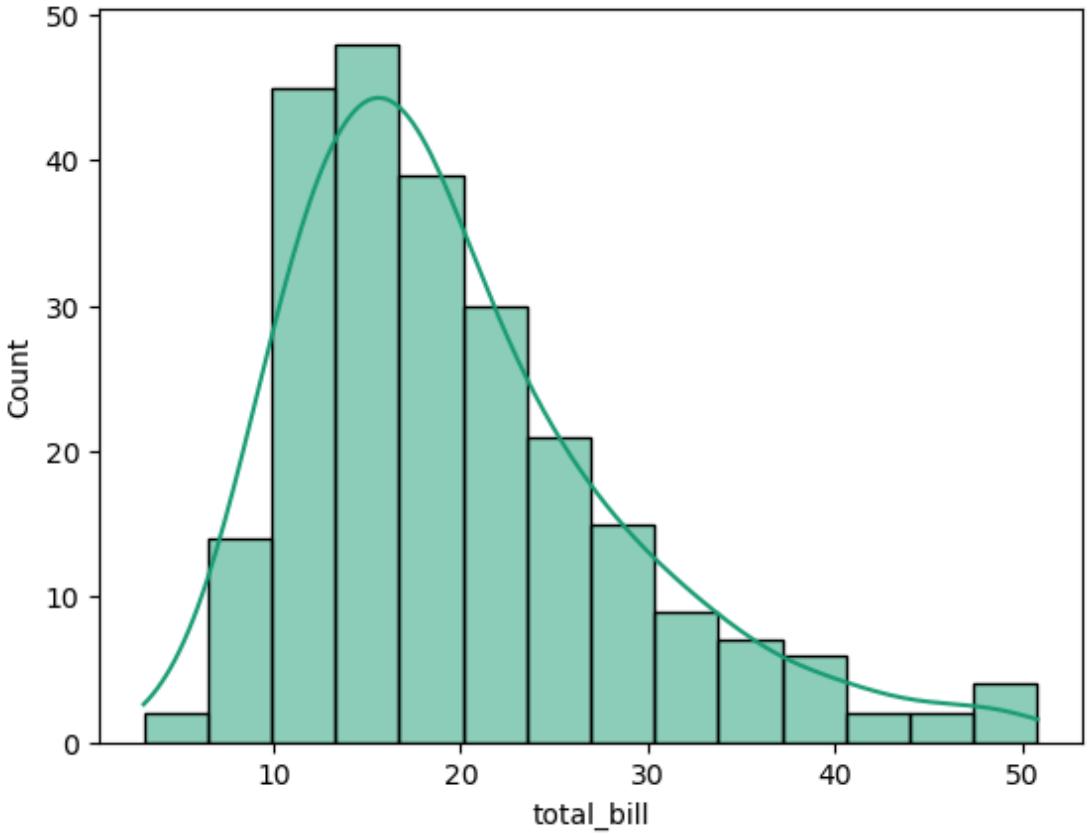
Some fancy mathematical curve to represent data density.

If we make a `histplot` of the `total_bill` column we can see that in someplaces there are a lot of values and in some places there are very few values.

And is the `histplot()` method there is a argument we can pass named `kde` that we can set to `True` to show the `KDE` curve.

In [19]:
```
sns.histplot(
    data=data,
    x='total_bill',
    kde=True,
)
```

Out[19]:
```
<Axes: xlabel='total_bill', ylabel='Count'>
```



Here you can see a line is formed. And this is the `KDE` curve.

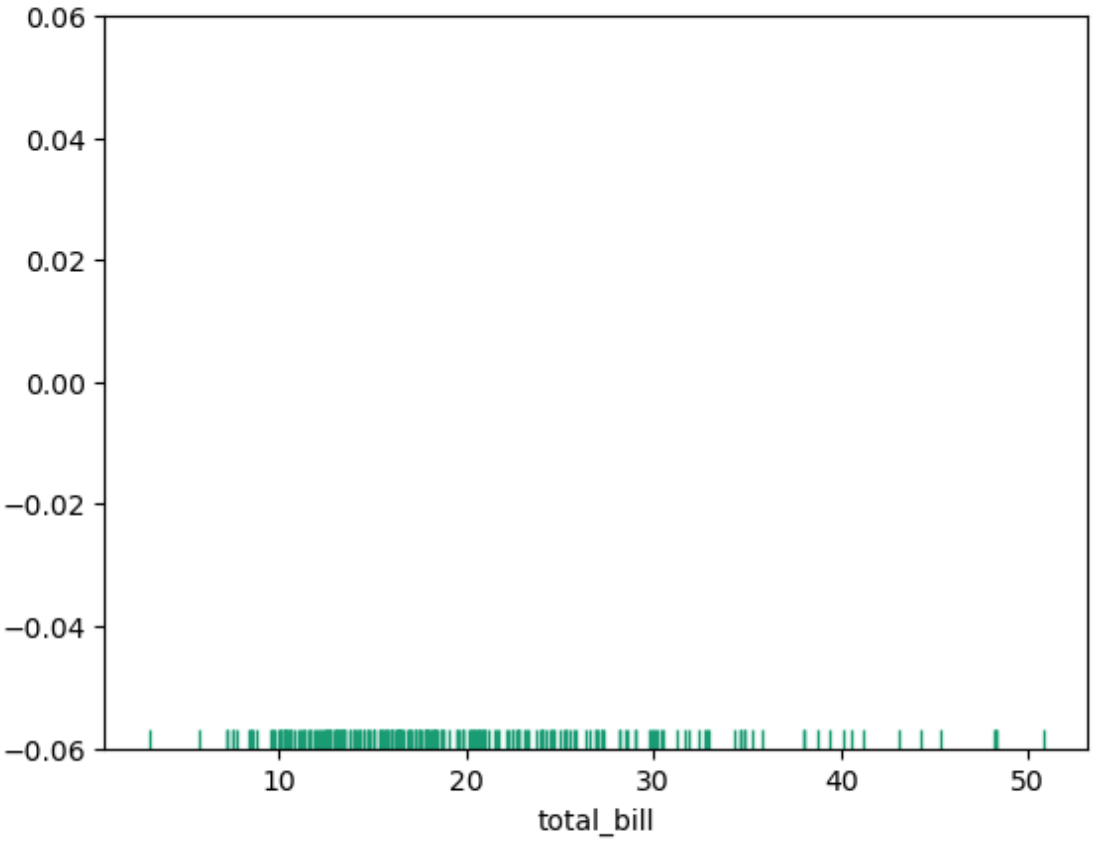
I won't go into the mathematical details here. Simply, `KDE` is a curve that represents the density of the datapoints. The Higher the density the higher the curve.

This helps visualize the data better in a more smooth way without the need to binning the data in a cirtain range.

To get understand the `KDE` curve better, we can use another plot named `rugplot` to show the distribution of the data points.

In [20]:
```
sns.rugplot(data=data, x='total_bill')
```

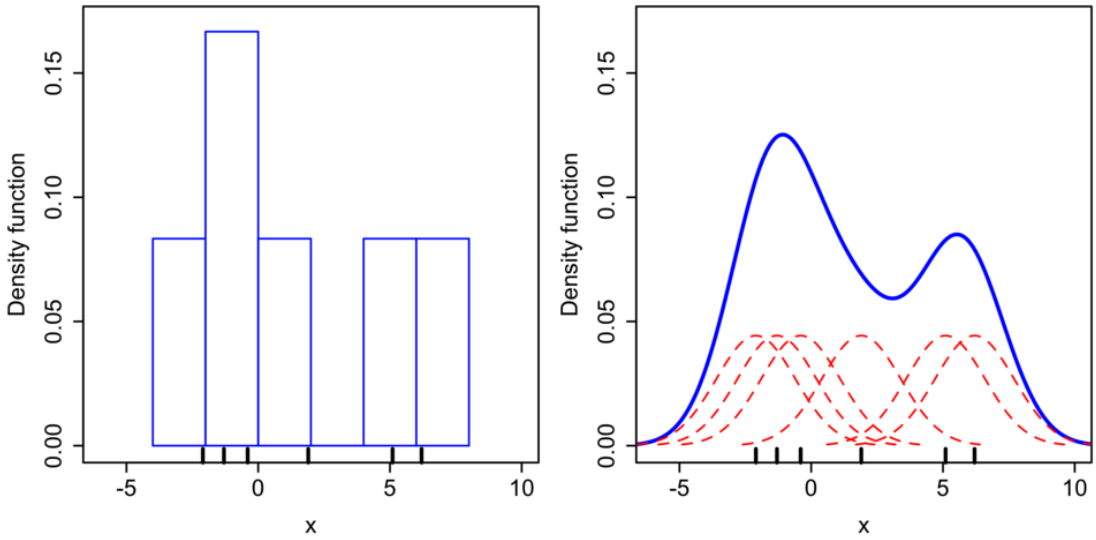
Out[20]:
```
<Axes: xlabel='total_bill'>
```



This is a plot that instead of showing the data points as dots, or in a bar form it shown as small bars that are aligned with the data points. It looks like a rug.

A histogram is actually representing the count of datapoint in a cirtain range.

And the `KDE` curve is representing the density of the datapoints.



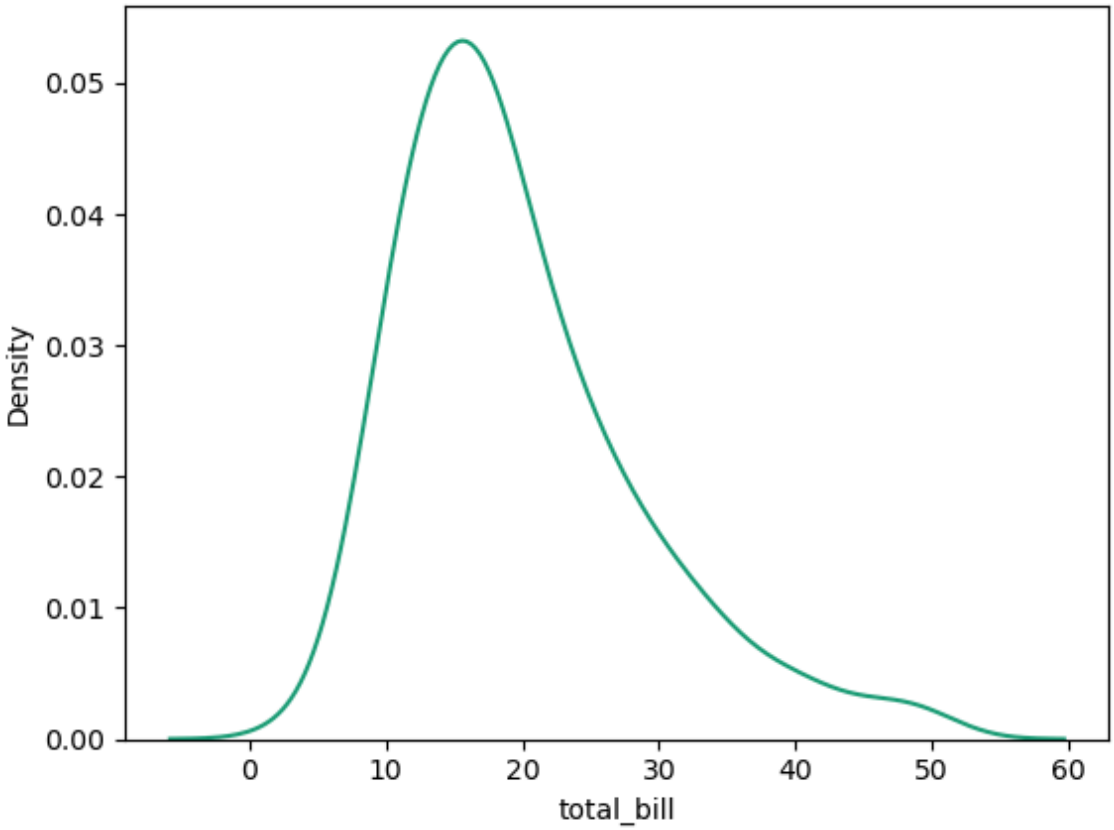
This image should make things more clear.

As you can in the image there are a gaussian curve drawn above all the data points and then the curves are summed up in a complecated mathematical way and we get a smooth curve representing the density of the datapoints.

Seaborn gives us a dedicated `kdeplot` method.

In [21]:
```
sns.kdeplot(data=data, x='total_bill')
```

Out[21]:
```
<Axes: xlabel='total_bill', ylabel='Density'>
```

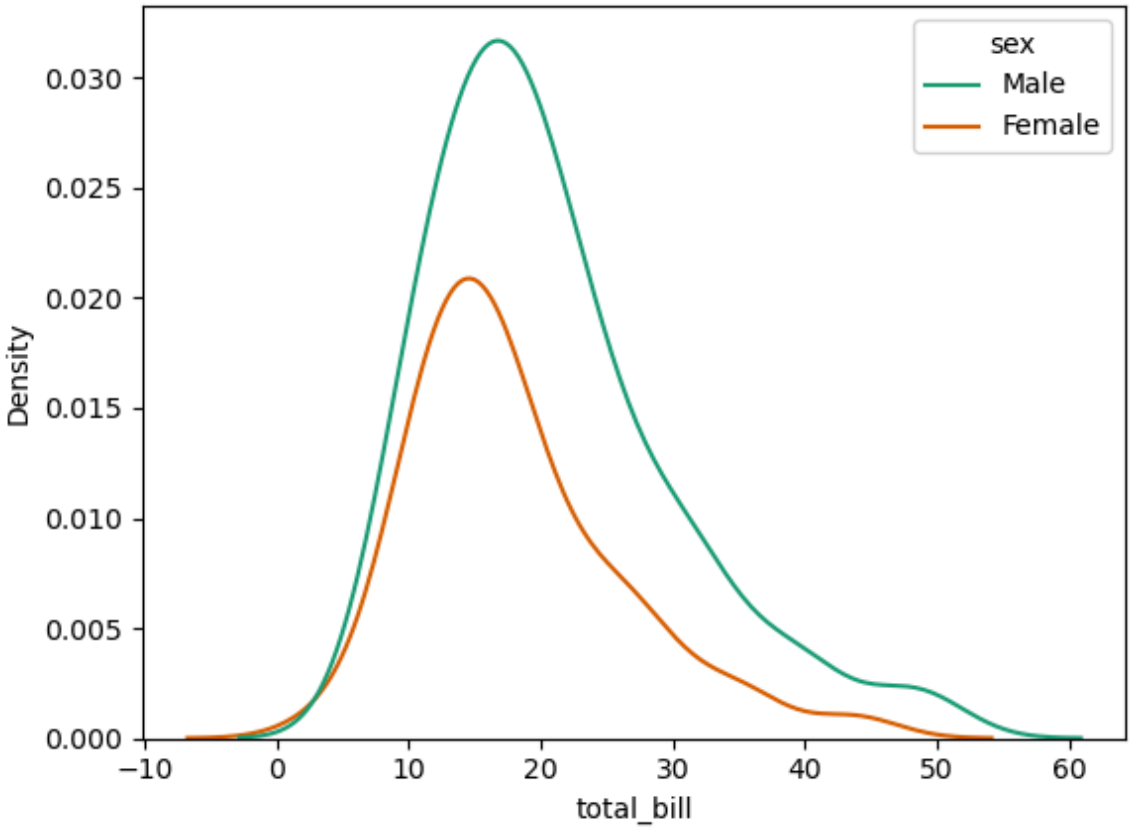


We can also set hue and palette in the `kdeplot` method too. But as I've already set the palette in this notebook I don't have to pass the `palette` argument.

I can just pass the `hue` argument.

In [22]:
```
sns.kdeplot(data=data, x='total_bill', hue='sex')
```

Out[22]:
```
<Axes: xlabel='total_bill', ylabel='Density'>
```



And that's it for the first part of the notebook and give you an small intro into the world of Seaborn and distribution plots.

Now let's talk about `categorical` plots.

Categorical plots

We learned about some interesting ways to visualize and monitor the distribution of the data, more specifically the numeric columns.

Now, we should talk about how we can get information about the `categorical` columns.

We have a bunch of categorical columns in the `tips` dataset.

In [23]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 7 columns):
 #   Column      Non-Null Count  Dtype   
---  ------      --------------  -----   
 0   total_bill  244 non-null    float64 
 1   tip         244 non-null    float64 
 2   sex         244 non-null    category
 3   smoker      244 non-null    category
 4   day         244 non-null    category
 5   time        244 non-null    category
 6   size        244 non-null    int64   
dtypes: category(4), float64(2), int64(1)
memory usage: 7.4 KB
```

We have columns like `sex`, `smoker`, `day` and `time`.

So, how do we get information about these columns?

First let's see the `histogram` equivalent for the `categorical` columns.

It's refered to as `countplot` in Seaborn.

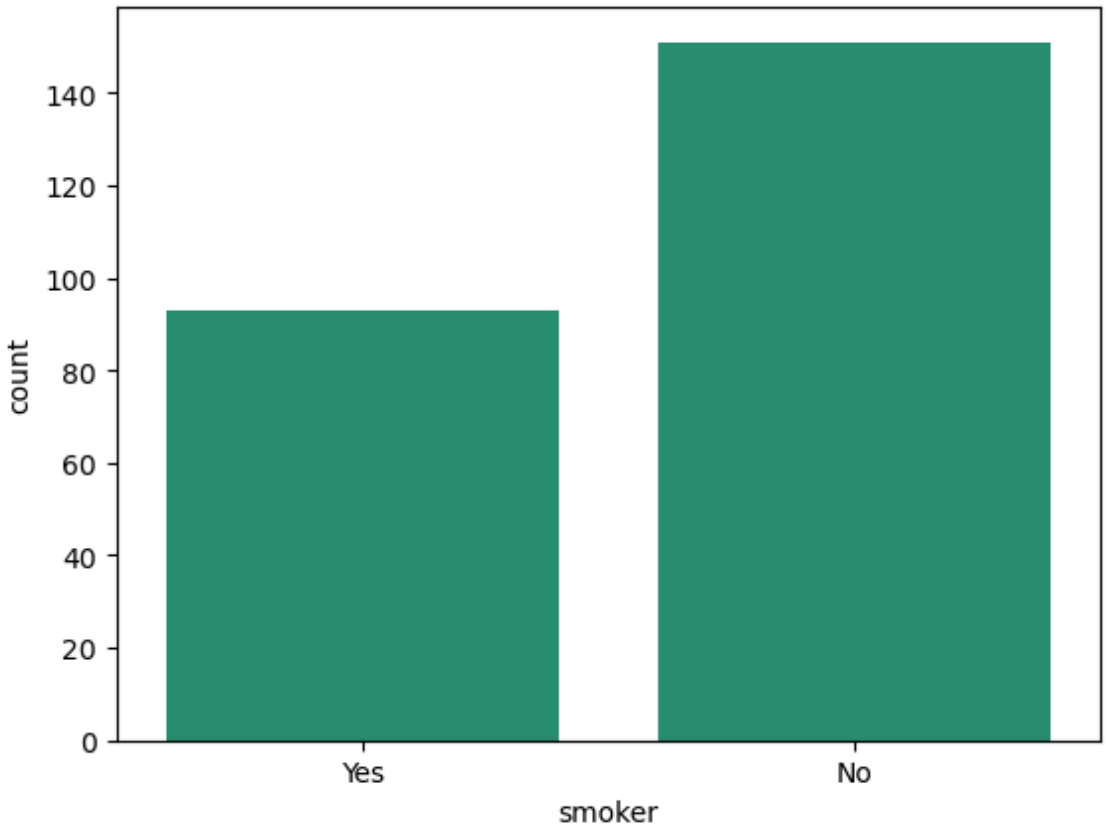
Countplot

Countplot is a plot that shows the count of the categorical columns.

Let's see the `countplot` for the `smoker` column.

In [24]:
```
sns.countplot(data=data, x='smoker')
```

Out[24]:
```
<Axes: xlabel='smoker', ylabel='count'>
```



Now, we can see how many smokers and non-smokers came to the restaurant.

This conveys the same information as the `histogram`, but for the `categorical` columns.

There is another plot that looks similar but gives us information on the `categorical` column and a `non-categorical` column.

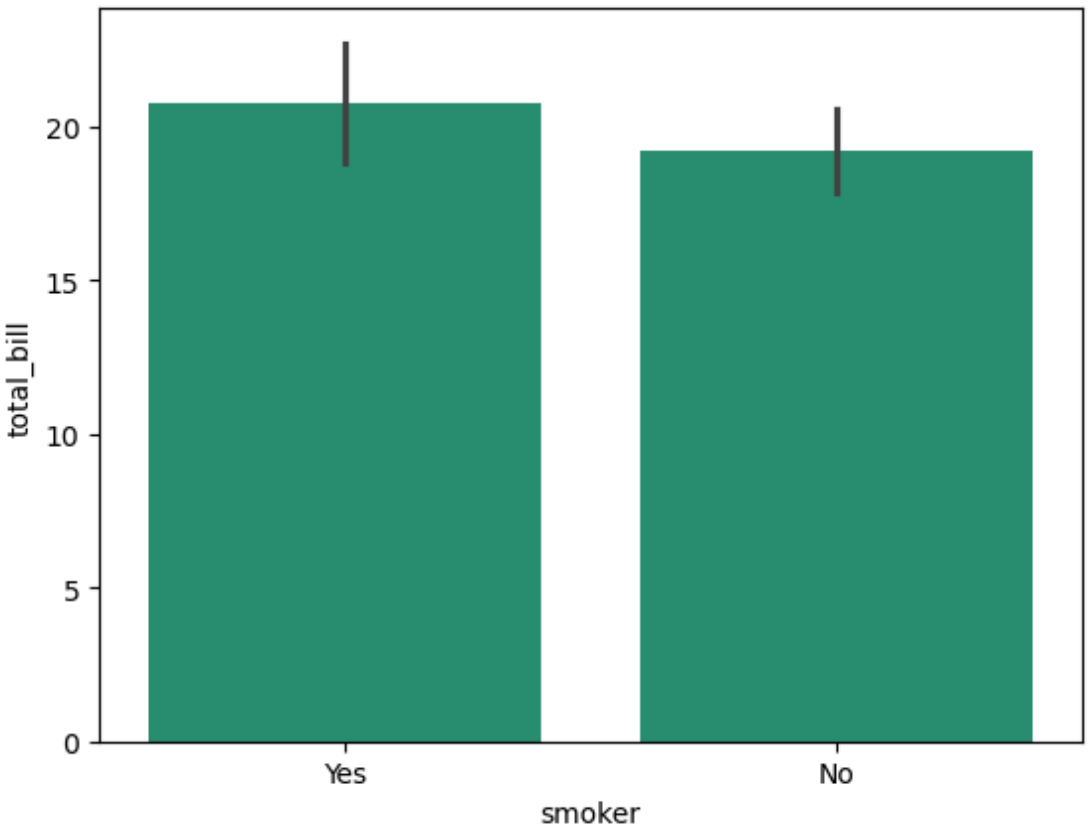
Let's say I want to see the relation between the `smoker` column and the `total_bill` column.

Barplot

Barplots are almost identical to `countplots` except that instead of showing the count of the categorical column it shows the `mean` of the `non-categorical` column.

In [25]:
```
sns.barplot(data=data, x='smoker', y='total_bill')
```

Out[25]:
```
<Axes: xlabel='smoker', ylabel='total_bill'>
```



And now, we have a barplot that shows the mean of the `total_bill` for `smokers` is higher than the mean of the `total_bill` for `non-smokers`.

In the previous countplot we saw the count of the `smokers` is higher than the `non-smokers`. But this barplot shows the mean of the `total_bill` for the `smokers` is higher than the mean of the `total_bill` for the `non-smokers`.

This the `smokers` are more likely to pay more.

And there is another thing I want to talk about is a `estimator` argument.

By default, the barplot method will show the mean of the `non-categorical` column. But if you want you can pass an `estimator` argument where you can tell Seaborn to show a different value.

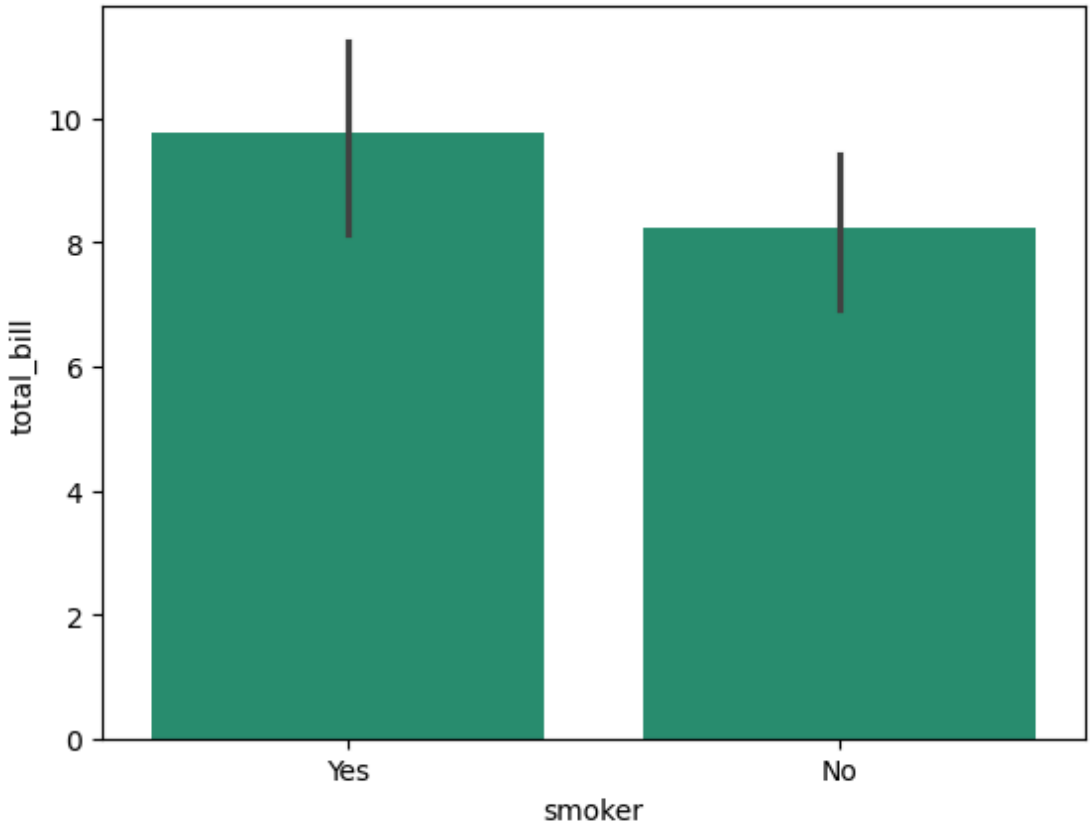
> You have to pass a `function` to the `estimator` argument.

Let's see the sum of the `total_bill` column.

In [26]:
```
import numpy as np

sns.barplot(data=data, x='smoker', y='total_bill', estimator=np.std)
```

Out[26]:
```
<Axes: xlabel='smoker', ylabel='total_bill'>
```



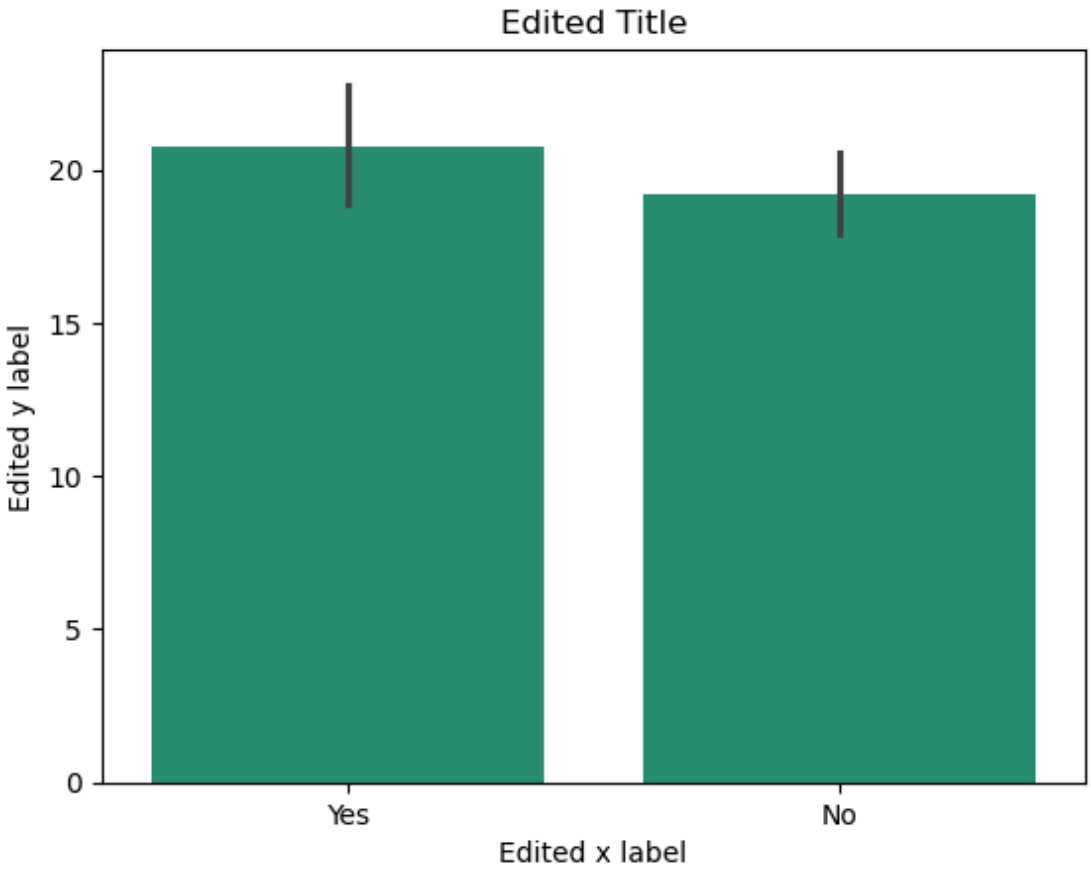
> In the above code, I have passed the `np.std` function to the `estimator` argument and in the barplot I can see the standard deviation of the `total_bill` for the `smokers` is higher than the standard deviation of the `total_bill` for the `non-smokers`

One thing I haven't talked about in this article is how you can use `matplotlib.pyplot` adjust of edit the plot.

As seaborn is actually a wrapper around `matplotlib.pyplot` you can use `matplotlib.pyplot` to adjust the plot directly like this.

In [27]:
```
sns.barplot(data=data, x='smoker', y='total_bill')
```

```
plt.xlabel('Edited x label')
plt.ylabel('Edited y label')
plt.title('Edited Title')
plt.show()
```



And it works seemlessly.

Also, when you are using seaborn methods, most of them returns a `matplotlib` axes object.

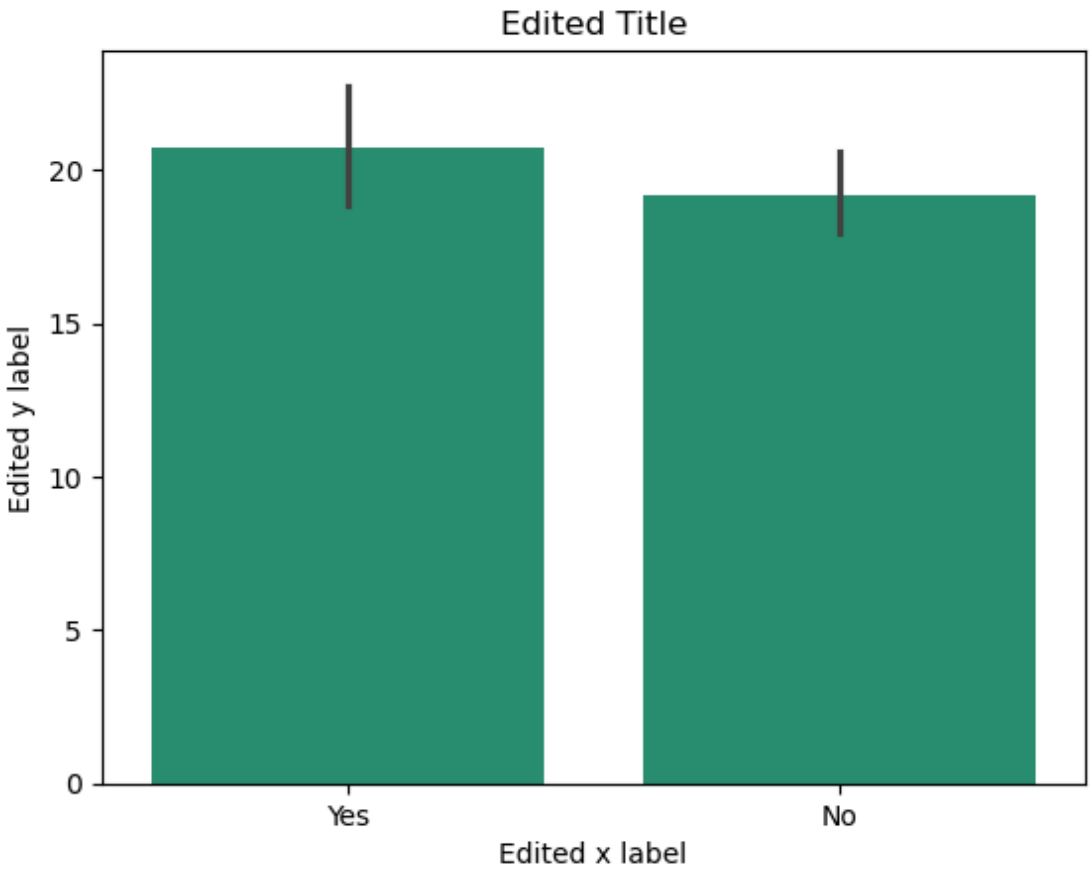
So, we can use that too to modify the plot.

In [28]:
```
from matplotlib.axes import Axes

ax: Axes = sns.barplot(data=data, x='smoker', y='total_bill')

ax.set_xlabel('Edited x label')
ax.set_ylabel('Edited y label')
ax.set_title('Edited Title')
```

Out[28]:
```
Text(0.5, 1.0, 'Edited Title')
```



That's how we can actually modify the plot directly.

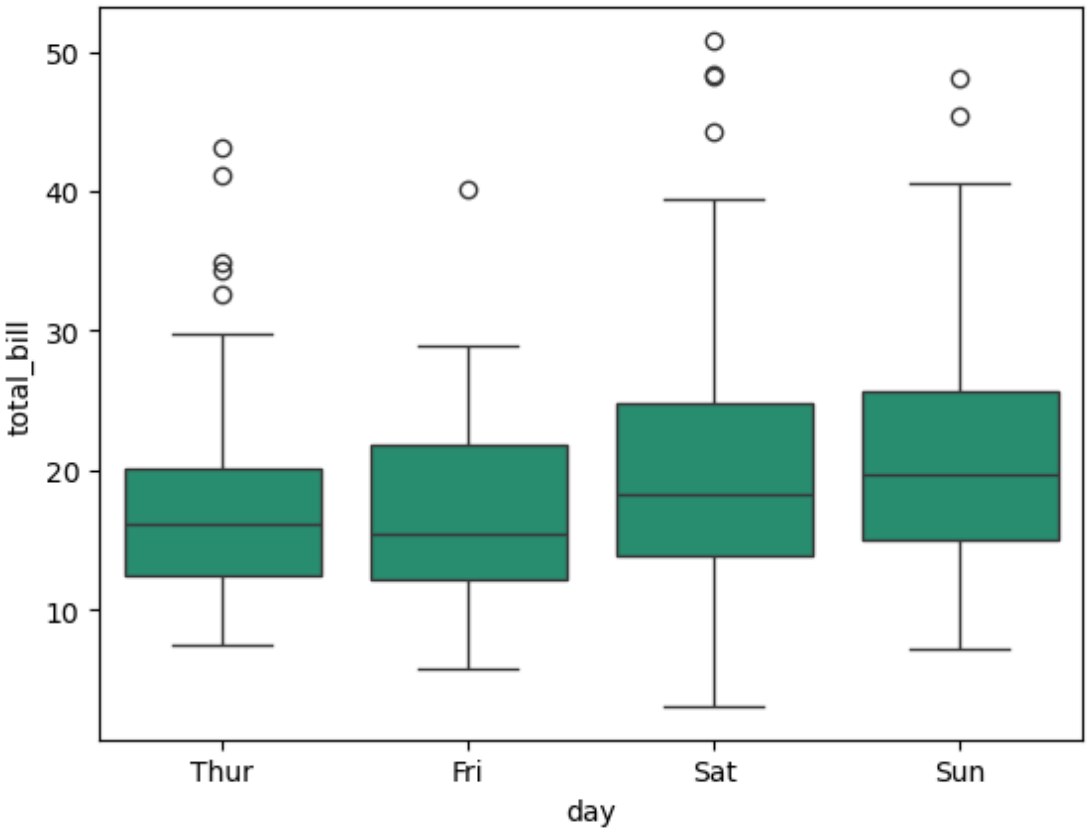
Now, another type of plot I want to show you is a `boxplot`.

Boxplot

A `boxplot` is a plot that shows the distribution of the `categorical vs non-categorical` columns.

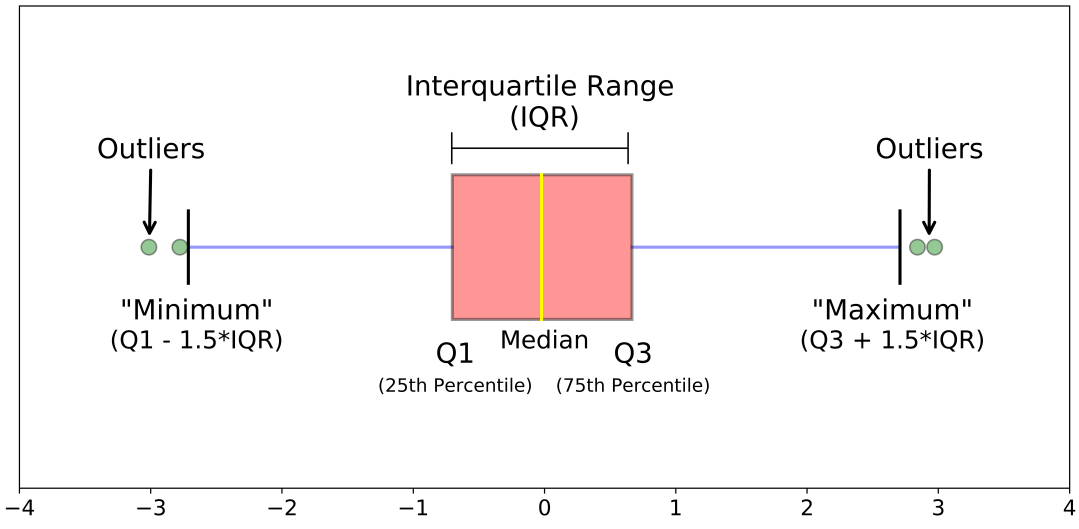
In [29]:
```
sns.boxplot(data=data, x='day', y='total_bill')
```

Out[29]:
```
<Axes: xlabel='day', ylabel='total_bill'>
```



Box plot show informations like `min` , `max` , `median` etc.

Heres how you can understand the box plot.



The dots are called `ouliers` . These outliers show the range of the `non-categorical` column, more precisely the `min` and `max` values.

Inside the box is the `median` value.

> Median is the middle value of the `non-categorical` column. It's also refered as the `2nd quartile` .

So, Median represents the middle value that mean 50% of the data is less than the median if the data is sorted.

`1st quartile` is the `25%` value of the data and `3rd quartile` is the `75%` value of the data.

Now, you might be curious about what does the Line at the both end of the box represent.

The line at the both end of the box is called the `whisker` and it shows the range of nearest 1st and 3rd quartile.

More, specically $Q1 \pm 1.5 * IQR$. This defines the range of the box any thing outside is called an `outlier`.

> IQR stands for Interquartile Range.

So, this box plot shows us a lot of information about the distribution of the `non-categorical` column.

SO, for example, from the graph above we can interpret that on `Thursday` `total_bill` is more likely to be in the range of `10` to `20` dollars with maximum range could be `30` dollars.

It's a usefull plot that can be used to understand the distribution of the `non-categorical` column like no other plot.

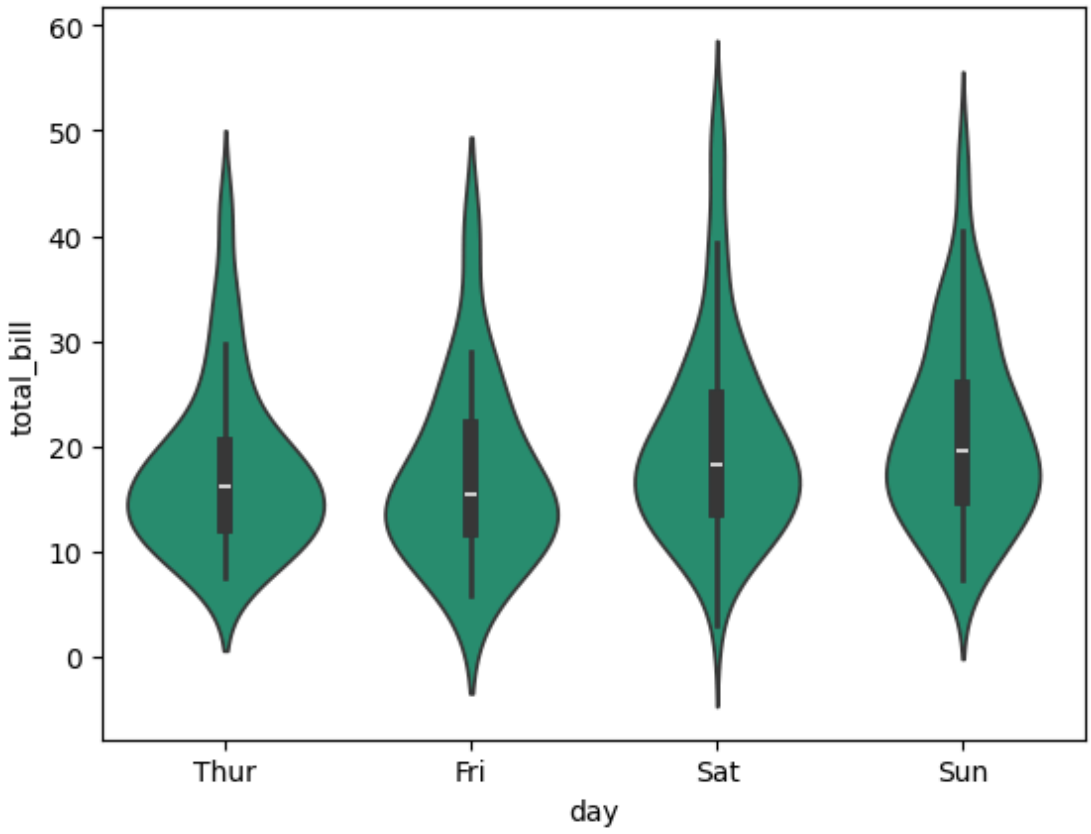
Another plot that does almost the same is a `violinplot`.

Violin Plot

A Violin Plot is almost the same as `boxplot` But it shows the distribution in a more `smooth` way. You don't have to worry about the maths behind it.

In [30]:
```
sns.violinplot(data=data, x='day', y='total_bill')
```

Out[30]:
```
<Axes: xlabel='day', ylabel='total_bill'>
```



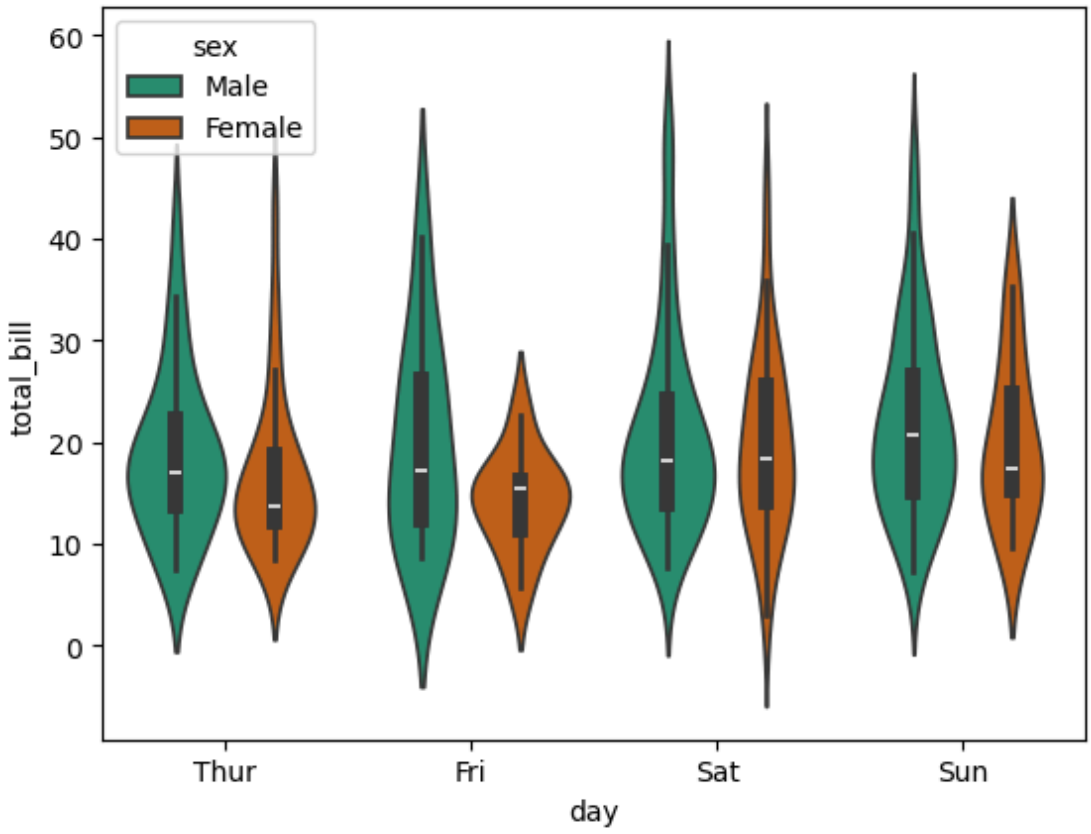
Looks good.

It also gives us the same information as the `boxplot` but in a `smooth` way. Inside each violin is a small box that shows the range of the `non-categorical` column and the violin is representing the distribution of the `non-categorical` column.

We can also set the `hue` argument to show the distribution of the `non-categorical` column for each category in the `categorical` column.

In [31]:
```
sns.violinplot(data=data, x='day', y='total_bill', hue='sex')
```

Out[31]:
```
<Axes: xlabel='day', ylabel='total_bill'>
```

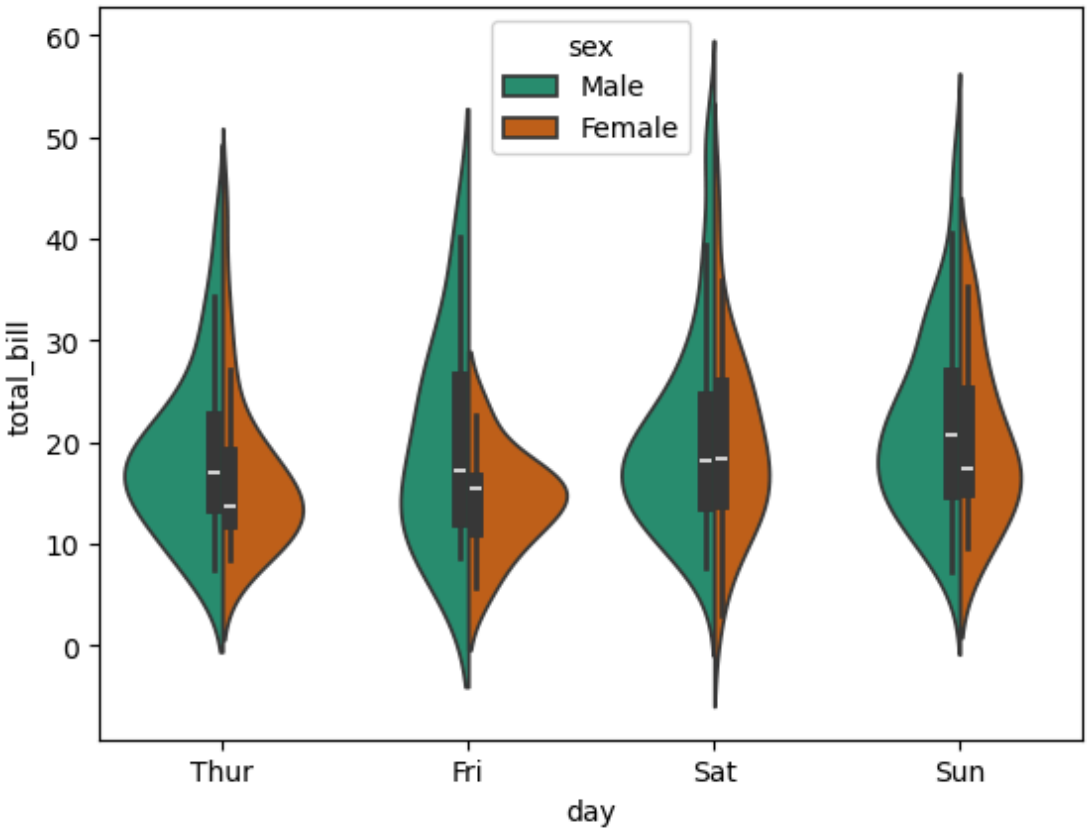


Now, this doesn't look so good right?

That's why we can pass another argument to see the hue side by side instead of two violin plots.

In [32]:
```
sns.violinplot(data=data, x='day', y='total_bill', hue='sex', split=True)
```

Out[32]:
```
<Axes: xlabel='day', ylabel='total_bill'>
```



Now, we can see how diffent the distribution of the `total bills` on different days for `male and female` is.

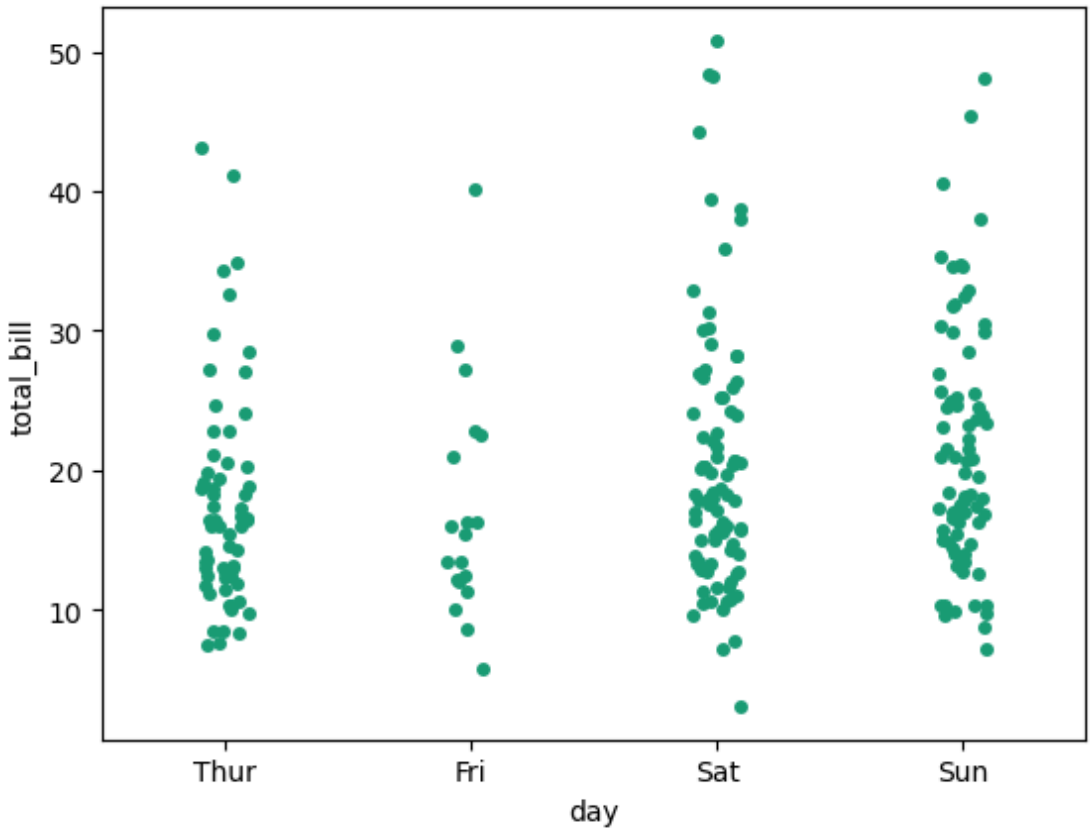
Now, there some plots that are closely related to `boxplot` and `violinplot` are `stripplot` and `swarmplot`.

Stripplot and Swarmplot

A `stripplot` is a scatter plot that shows the distribution of the `non-categorical` column for each category in the `categorical` column.

In [33]:
```
sns.stripplot(data=data, x='day', y='total_bill')
```

Out[33]:
```
<Axes: xlabel='day', ylabel='total_bill'>
```

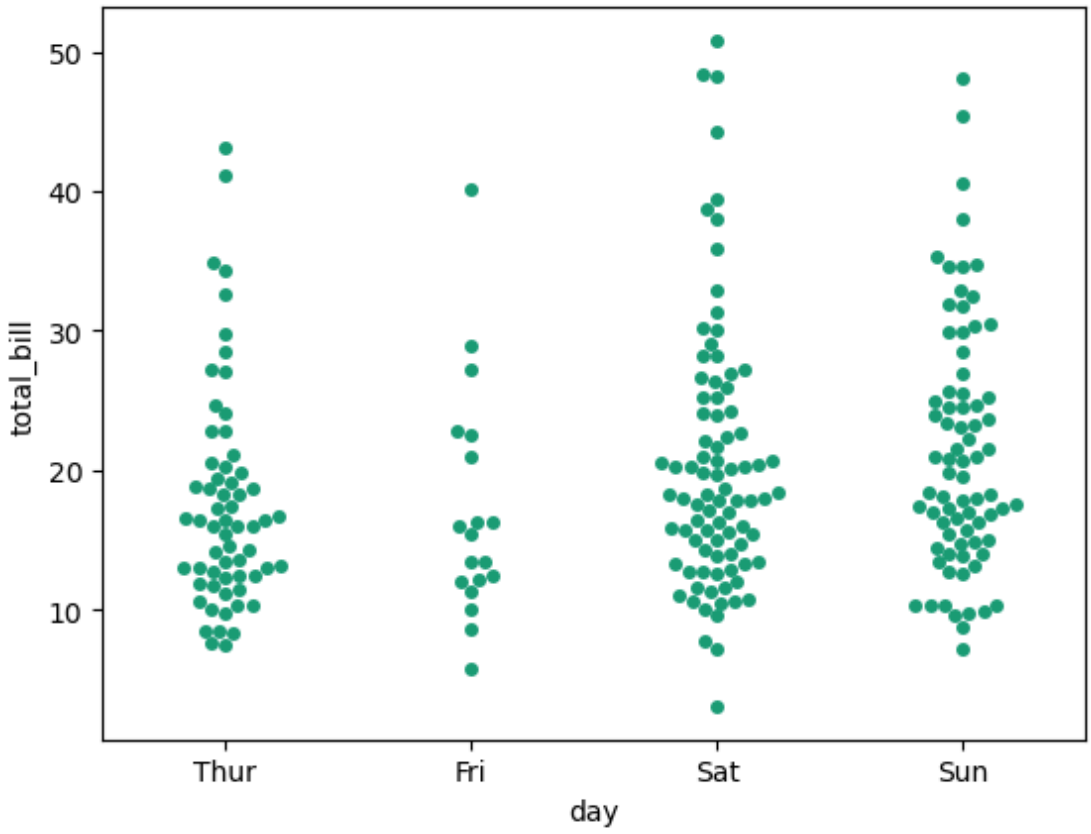


This shows the concentration of the `non-categorical` column for each category in the `categorical` column. Which can be interpreted from the box plot. But if you are a dummy like me and you want to see the distribution of the `non-categorical vs categorical` columns you can use a `stripplot`.

Swarmplot is also a scatter plot that does the same this but instead of showing and overlapping points it shows the scatterplot in a pattern that you might remember from a `violinplot`.

In [34]:
```
sns.swarmplot(data=data, x='day', y='total_bill')
```

Out[34]:
```
<Axes: xlabel='day', ylabel='total_bill'>
```

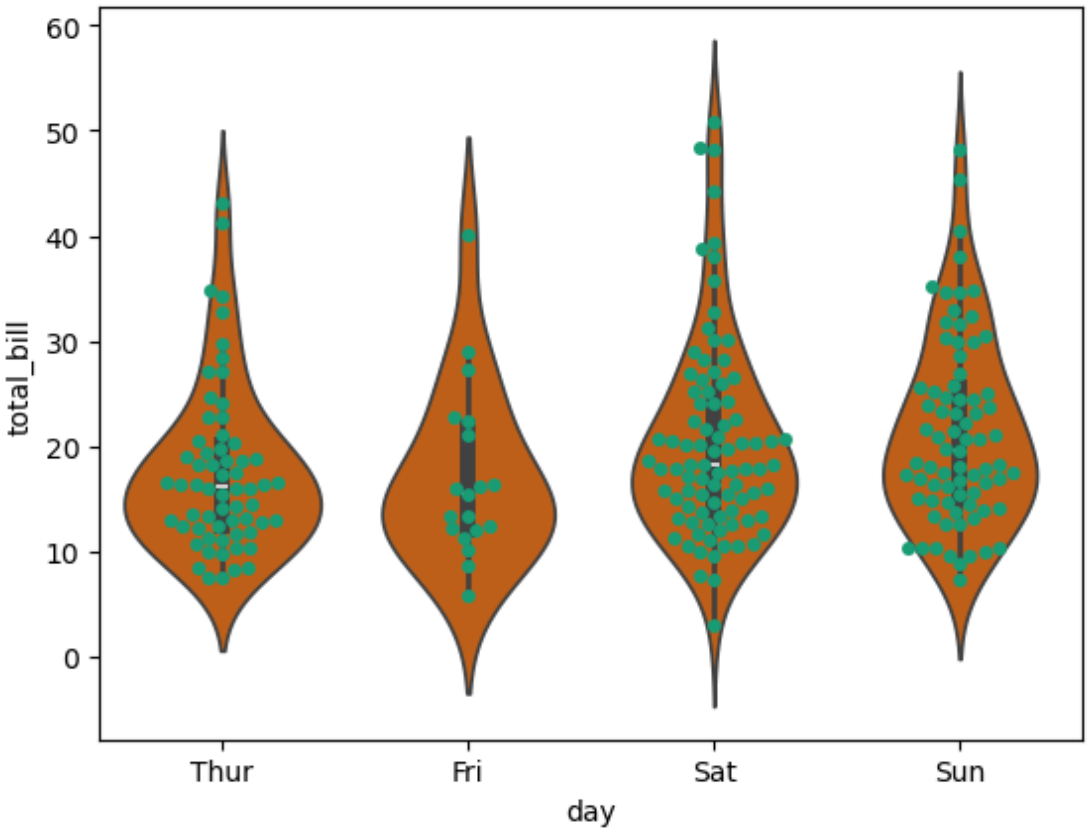


Looks kinda simillar to a `violinplot`.

Exactly, ans we can confirm this by doing a `violinplot` and a `swarmplot` on top of each other.

In [35]:
```
sns.swarmplot(data=data, x='day', y='total_bill')
sns.violinplot(data=data, x='day', y='total_bill')
```

Out[35]:
```
<Axes: xlabel='day', ylabel='total_bill'>
```



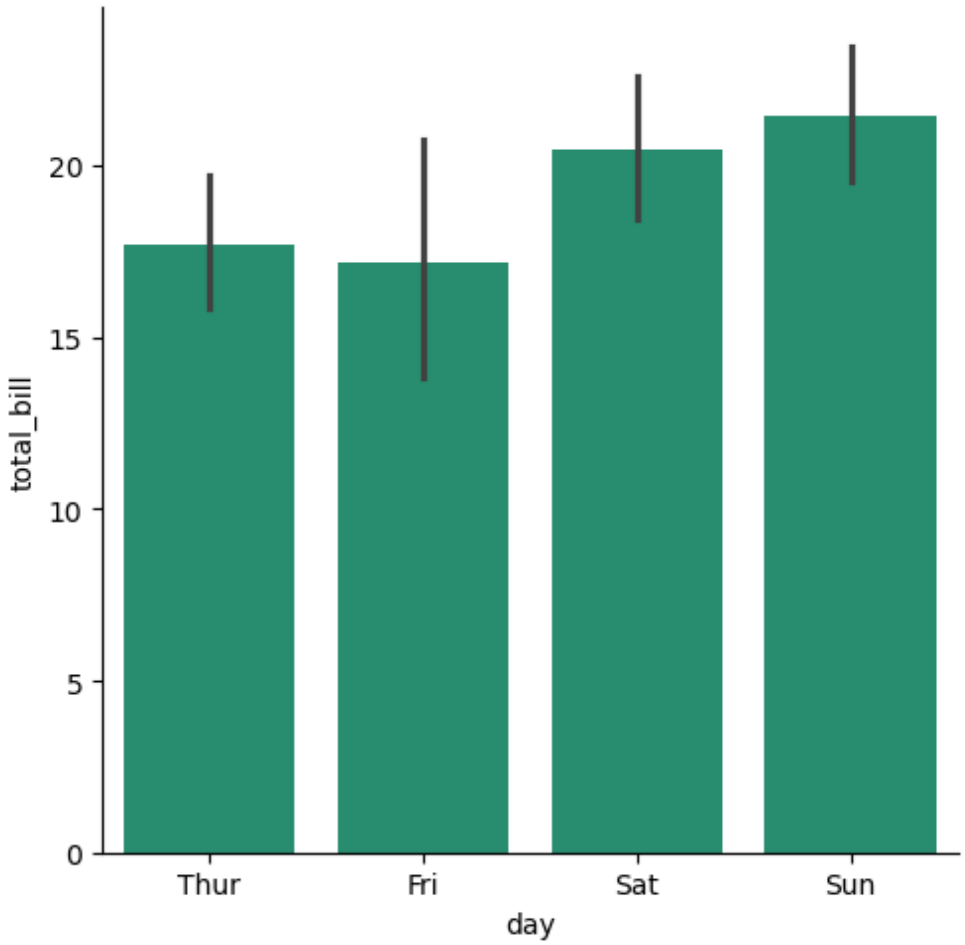
As you can see, they are exactly the same adn this brings me to another point that a violin plot is a `KDE plot` of the `stripplot` or `swarmplot`.

And lastly I want show you a plot that can do all these things at the same time is a `catplot`.

Catplot

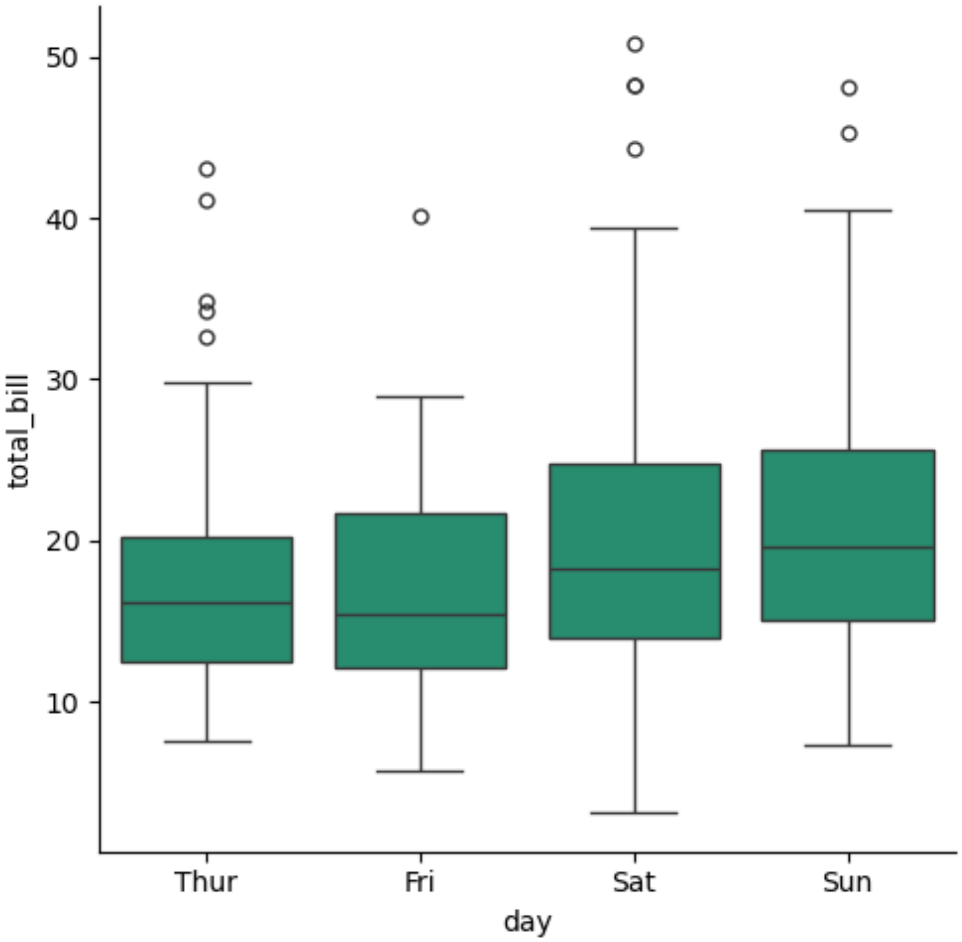
In [36]:
```
sns.catplot(data=data, x='day', y='total_bill', kind='bar')
```

Out[36]:
```
<seaborn.axisgrid.FacetGrid at 0x7ce67dadb7d0>
```



In [37]:
```
sns.catplot(data=data, x='day', y='total_bill', kind='box')
```

Out[37]:
```
<seaborn.axisgrid.FacetGrid at 0x7ce67d892950>
```



And almost every categorical plot can be made with this one method.

Even though I Like dedicated methods for each plot because it keeps the code clean and readable, catplot is a very useful method that can do all these plots at the same time with a single line of code.

So, I would heavily recommend going to the official documentation for more information about catplot.

Well, now that we have a lot of plots for `categorical` and `numeric` columns, There are some special types of plots that can be used to visualize the relationship of multiple `columns` like the pair plot.

One of them is a `heatmap`.

Matrix plots

Heatmap

A `heatmap` is plot that show the `correlation` between the `numeric` columns.

Now, what is `correlation`?

Correlation is a statistical way to measure how strongly two variables are related to each other.

If the correlation is `1` then the two variables are perfectly correlated, if the correlation is `-1` then the two variables are perfectly anti-correlated and if the correlation is `0` then the two variables are not correlated at all.

So, let's see an example.

In [38]:
```
data.head()
```

Out[38]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

The `tips` dataset has data of a restaurant bills and tips and has both `numeric` and `categorical` columns. Correlation can be use to see the relationship between the `numeric` columns only.

For that, pandas dataframes have a method called `corr` that can be used to calculate the correlation between the `numeric` columns.

> We have to specify `numeric_only=True` to only calculate the correlation between the `numeric` columns.

In [39]:
```
corr = data.corr(numeric_only=True)
corr
```

Out[39]:

	total_bill	tip	size
total_bill	1.000000	0.675734	0.598315
tip	0.675734	1.000000	0.489299
size	0.598315	0.489299	1.000000

This is called a correlation table. And this shows the `correlation coefficients` between the `numeric` columns and If the row and the column has the same `column name` then it is `1` because a column is perfectly correlated with itself.

- `1` is perfectly correlated
- `.5 - .75` is moderately correlated
- `.25 - .5` is slightly correlated
- `0` is not correlated at all

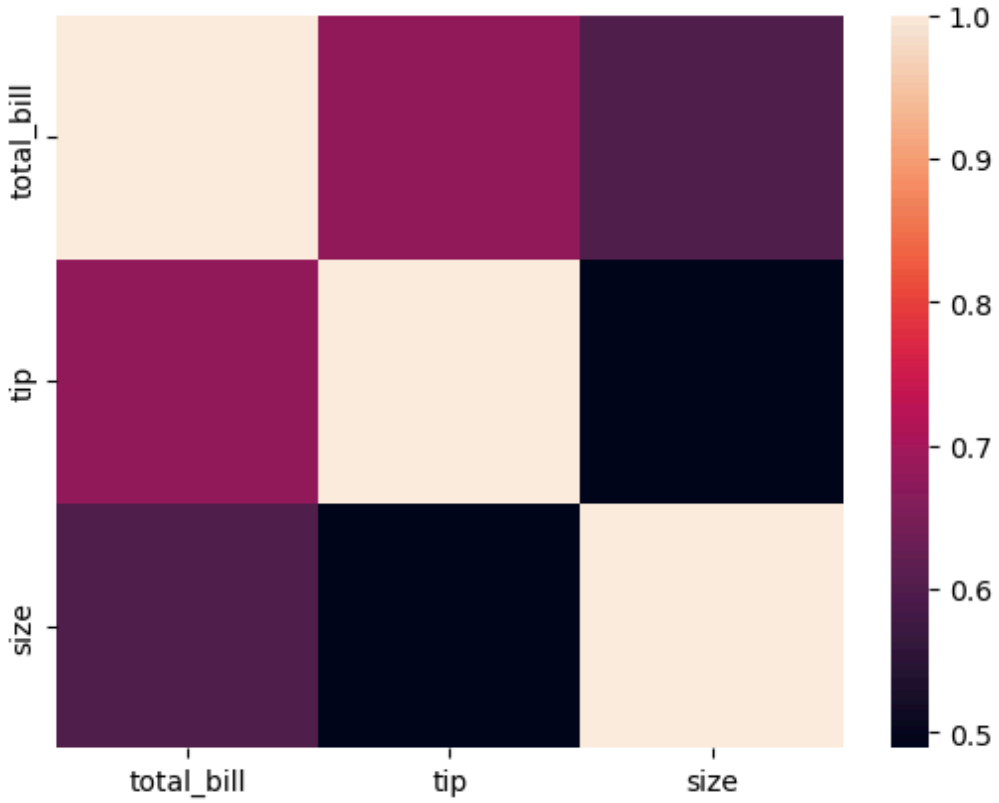
And if the correlation is `negetive` then the two variables are anti-correlated.

Now, seeing this table might be confusing and overwhelming. Sometimes the table gets too big and hard to read.

So, we can visualize this using a `heatmap`.

In [40]:
```
sns.heatmap(corr)
```

Out[40]: <Axes: >



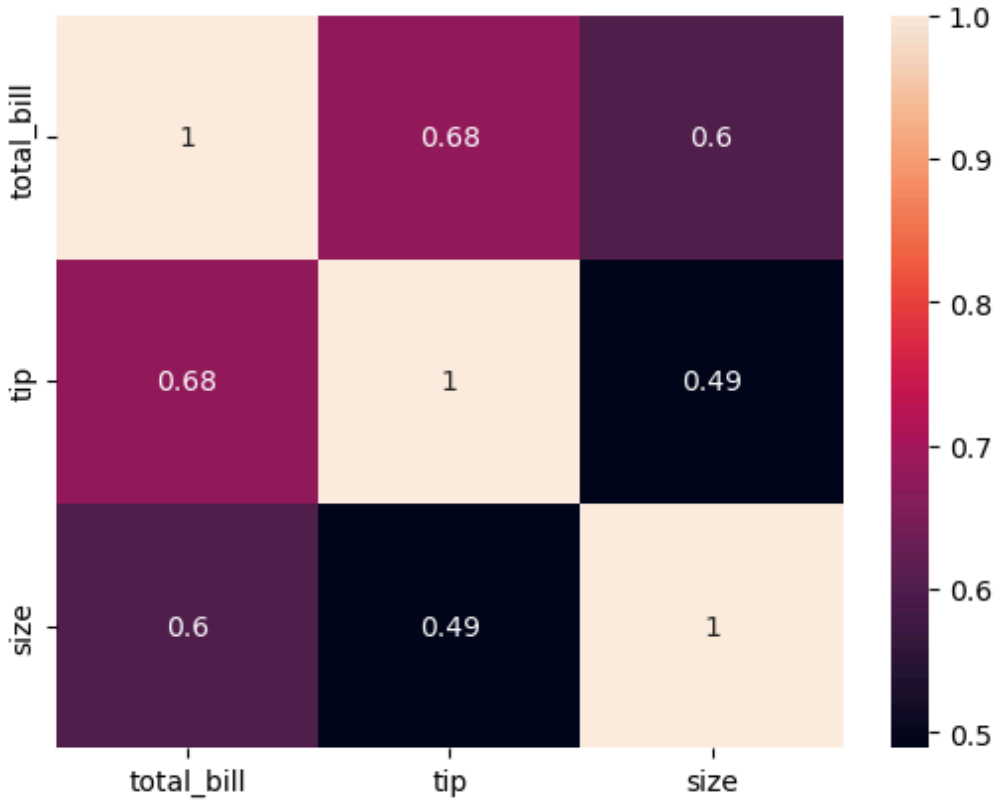
And now we can visualize the correlation between the `numeric` columns.

Darker shades are less correlated and lighter shades are more correlated.

And If you want to see the values of the correlation you can use the `annot` argument.

In [41]:
```
sns.heatmap(corr, annot=True)
```

Out[41]: <Axes: >



Now, the values of the correlation are shown in the heatmap.

Heatmaps are mainly used for visualizing the correlation between the `numeric` columns. But sometimes you might want to visualize numrical value change of a dataset...

For example, I'll load another dataset from seaborn named `flights` that contains data of flights from New York to Los Angeles in 2013.

In [42]:
```
flights = sns.load_dataset('flights')
flights.head()
```

Out[42]:

	year	month	passengers
0	1949	Jan	112
1	1949	Feb	118
2	1949	Mar	132
3	1949	Apr	129
4	1949	May	121

In [43]:
```
flights.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 144 entries, 0 to 143
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   year        144 non-null    int64
 1   month       144 non-null    category
 2   passengers  144 non-null    int64
dtypes: category(1), int64(2)
memory usage: 2.9 KB
```

As you can see that we have a data set with `3` columns and `144` rows. This dataset contains the passenger count from...

In [44]:
```
flights['year'].min(), flights['year'].max()
```

Out[44]:
```
(1949, 1960)
```

`1949` and `1960` on each month.

It's a very straight forward dataset.

But to get more information like let's say You want to see passenger count from `1949` to `1960` for each month, is hard.

So, what we can do is we can pivot the dataset to get the passenger count from `1949` to `1960` for each month.

> A pivot table is a table where the rows and columns are interchanged.

In this case, We have limited `year` and `months` in the month column, So, we can pivot the dataset to get the passenger count from `1949` to `1960` for each month as follows:

In [45]:
```
pivot = flights.pivot_table(
    index='month',
    columns='year',
    values='passengers'
)

pivot
```

```
/tmp/ipykernel_195674/3287994124.py:1: FutureWarning: The default value of
observed=False is deprecated and will change to observed=True in a future v
ersion of pandas. Specify observed=False to silence this warning and retain
the current behavior
  pivot = flights.pivot_table(
```

Out[45]:

year month	1949	1950	1951	1952	1953	1954	1955	1956	1957	1958	1959	196
Jan	112.0	115.0	145.0	171.0	196.0	204.0	242.0	284.0	315.0	340.0	360.0	417
Feb	118.0	126.0	150.0	180.0	196.0	188.0	233.0	277.0	301.0	318.0	342.0	391
Mar	132.0	141.0	178.0	193.0	236.0	235.0	267.0	317.0	356.0	362.0	406.0	419
Apr	129.0	135.0	163.0	181.0	235.0	227.0	269.0	313.0	348.0	348.0	396.0	461
May	121.0	125.0	172.0	183.0	229.0	234.0	270.0	318.0	355.0	363.0	420.0	472
Jun	135.0	149.0	178.0	218.0	243.0	264.0	315.0	374.0	422.0	435.0	472.0	535
Jul	148.0	170.0	199.0	230.0	264.0	302.0	364.0	413.0	465.0	491.0	548.0	622
Aug	148.0	170.0	199.0	242.0	272.0	293.0	347.0	405.0	467.0	505.0	559.0	606
Sep	136.0	158.0	184.0	209.0	237.0	259.0	312.0	355.0	404.0	404.0	463.0	508
Oct	119.0	133.0	162.0	191.0	211.0	229.0	274.0	306.0	347.0	359.0	407.0	461
Nov	104.0	114.0	146.0	172.0	180.0	203.0	237.0	271.0	305.0	310.0	362.0	390
Dec	118.0	140.0	166.0	194.0	201.0	229.0	278.0	306.0	336.0	337.0	405.0	432

Here, I'm using a dataframe method called `pivot_table` to pivot the dataset and set columns as years from the `year` column and months as `rows` from the `month` column.

And we have a dataframe called `pivot` that contains the passenger count from `1949` to `1960` for each month.

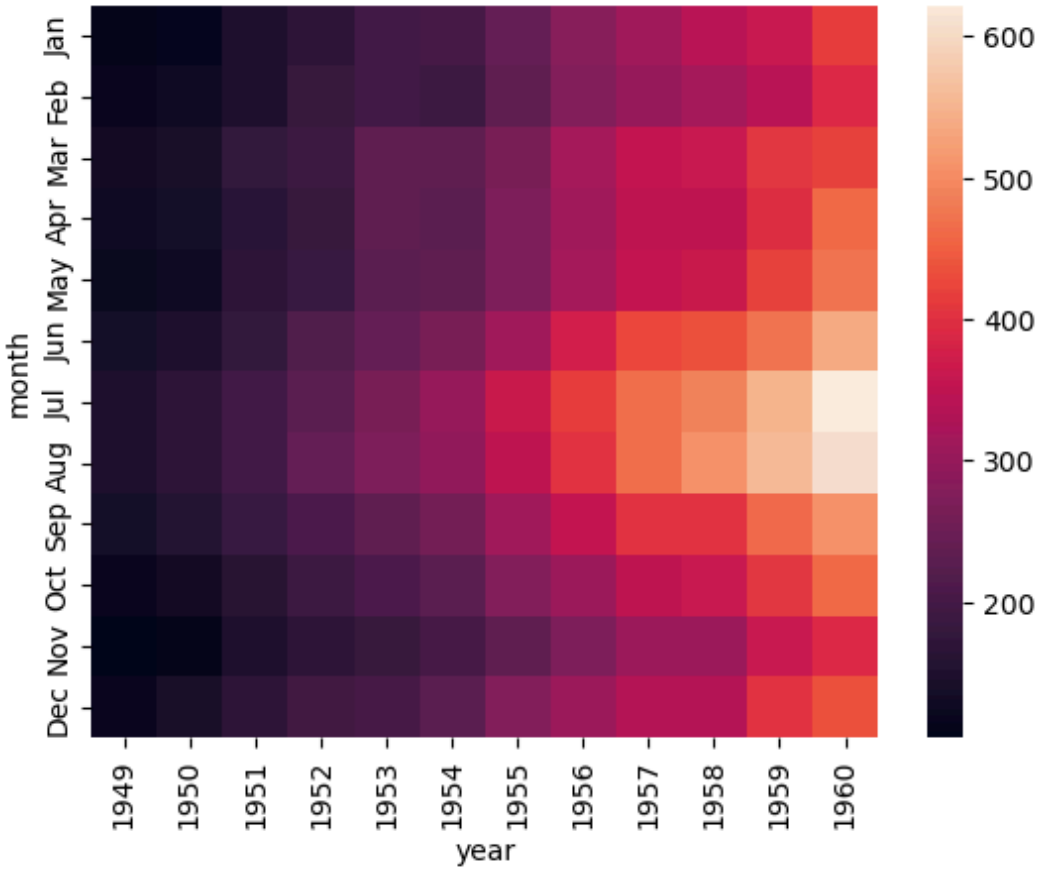
We can now easily find how many passengers boarded from `1949` to `1960` for each month.

But it's hard to see the change in the passenger count.

So, what we can do is we can use a `heatmap` to visualize the change in the passenger count.

In [46]:
```
sns.heatmap(pivot)
```

Out[46]:
```
<Axes: xlabel='year', ylabel='month'>
```



Now, we can clearly see that, as the years pass, the passenger count has increased and most increase was in `july` and `august` months maybe because that is the holiday season.

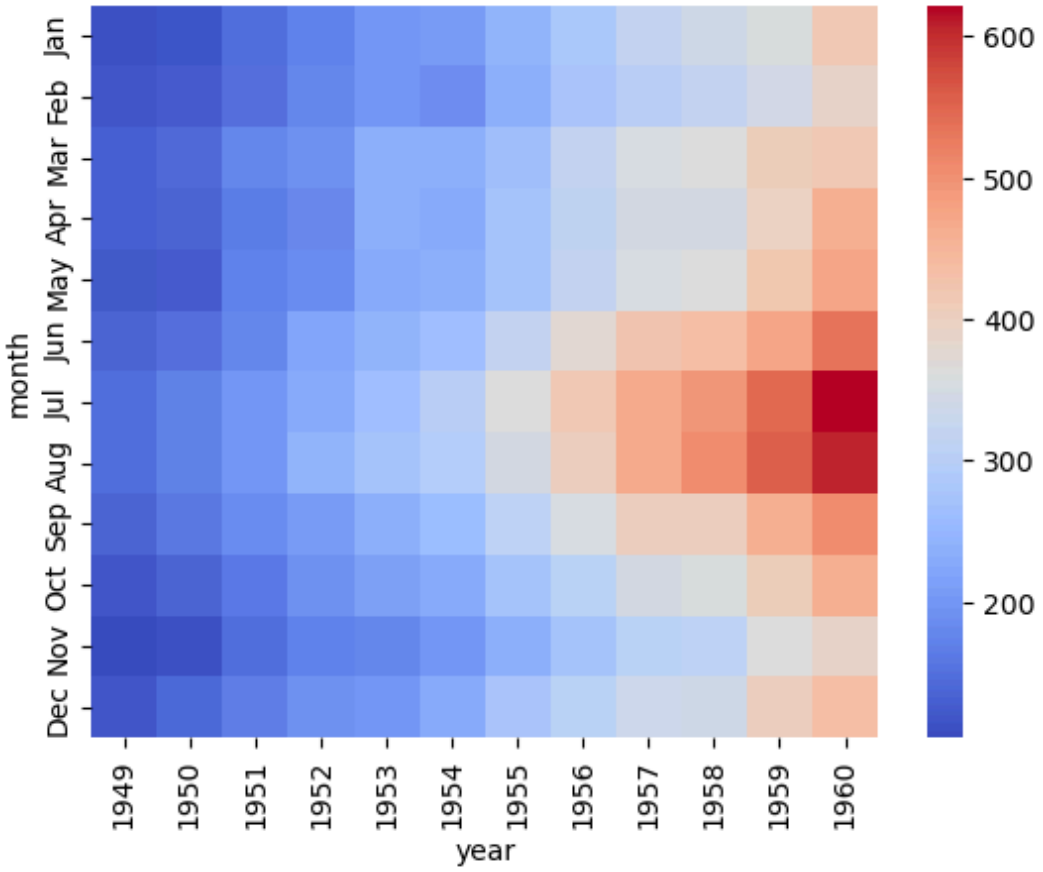
Now, lastly I want to show you some stylling of heatmaps.

One thing that you can do is you can change the color of the heatmap. But be sure to pick something smooth, not like my palette(Dark2).

Unlick other plots the `heatmap` does not have a `palette` argument, it has a `cmap` argument.

In [47]:
```
sns.heatmap(pivot, cmap='coolwarm')
```

Out[47]:
```
<Axes: xlabel='year', ylabel='month'>
```



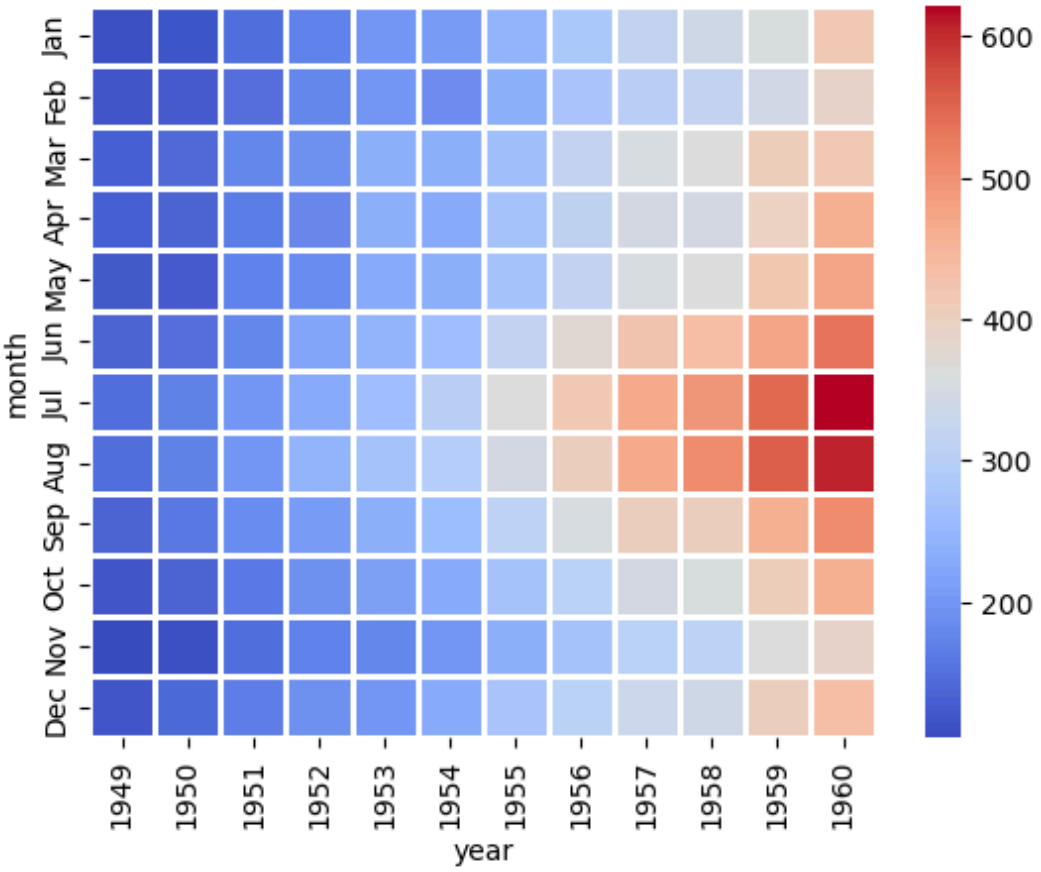
Just pass the name of the `palette` you like in the `cmap` argument.

Another modification you might need is line color and line width.

Right now the colors are touching each other and it look cool but sometime you might want to change that.

In [48]:
```
sns.heatmap(pivot, cmap='coolwarm', linecolor='white', linewidth=1)
```

Out[48]:
```
<Axes: xlabel='year', ylabel='month'>
```



And now you have a heat map with seperated shades of the palette of your choice.

Clustermaps

`Clustermaps` are very similar to `heatmaps` but instead of showing the correlation between the columns, they cluster the columns and rows based of how similar or close they are to each other.

> Too bad, for cluster maps to work you need to have `scipy` installed.

So, let's install it.

```
conda install scipy
```

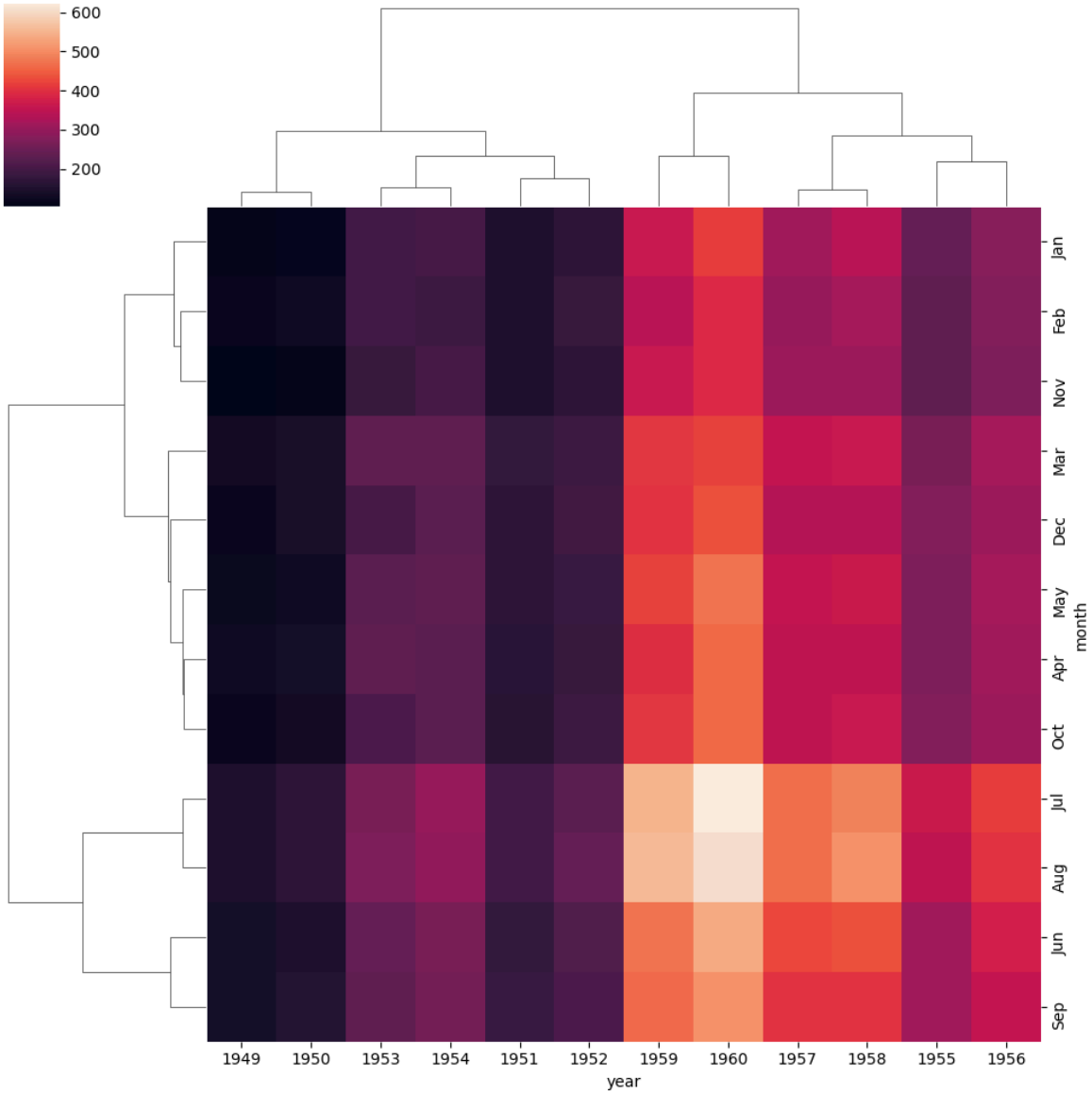
> Don't forget to activate your environment.

Almost all the libraries we use in machine learning stuff are `scipy` based even the OG `numpy` library. There's a interesting story of how numpy got separated from scipy to become it's own library. You can easily find it on the internet.

So, now let's see what this clustermap looks like.

In [49]:
```
sns.clustermap(pivot)
```

Out[49]:
```
<seaborn.matrix.ClusterGrid at 0x7ce67d47b510>
```



And here you can see some interesting things.

First, the columns are grouped based on how similar they are to each other as well as the rows.

And a tree like structure is formed for the columns and the pivot table is shown in the middle.

Now, we vey clearly specify which columns are similar to each other.

As you can see `july` and `august` are in the same cluster and `1959` and `1960` are in the same cluster Meaning the most passengers boarded in `july` and `august` were in `1959` and `1960` respectively.

I hope you understood the heatmap and clustermap well.

Now, let's talk about `grid` plots.

Grid plots

Pair Grid

You remember the `pair plot` ?

It automatically makes a grid plot of all the numeric columns in the dataset.

Pair frid exactly that but instead of making all the plot you have the ability customize it.

In [50]:
```
iris = sns.load_dataset('iris')
iris.head()
```

Out[50]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

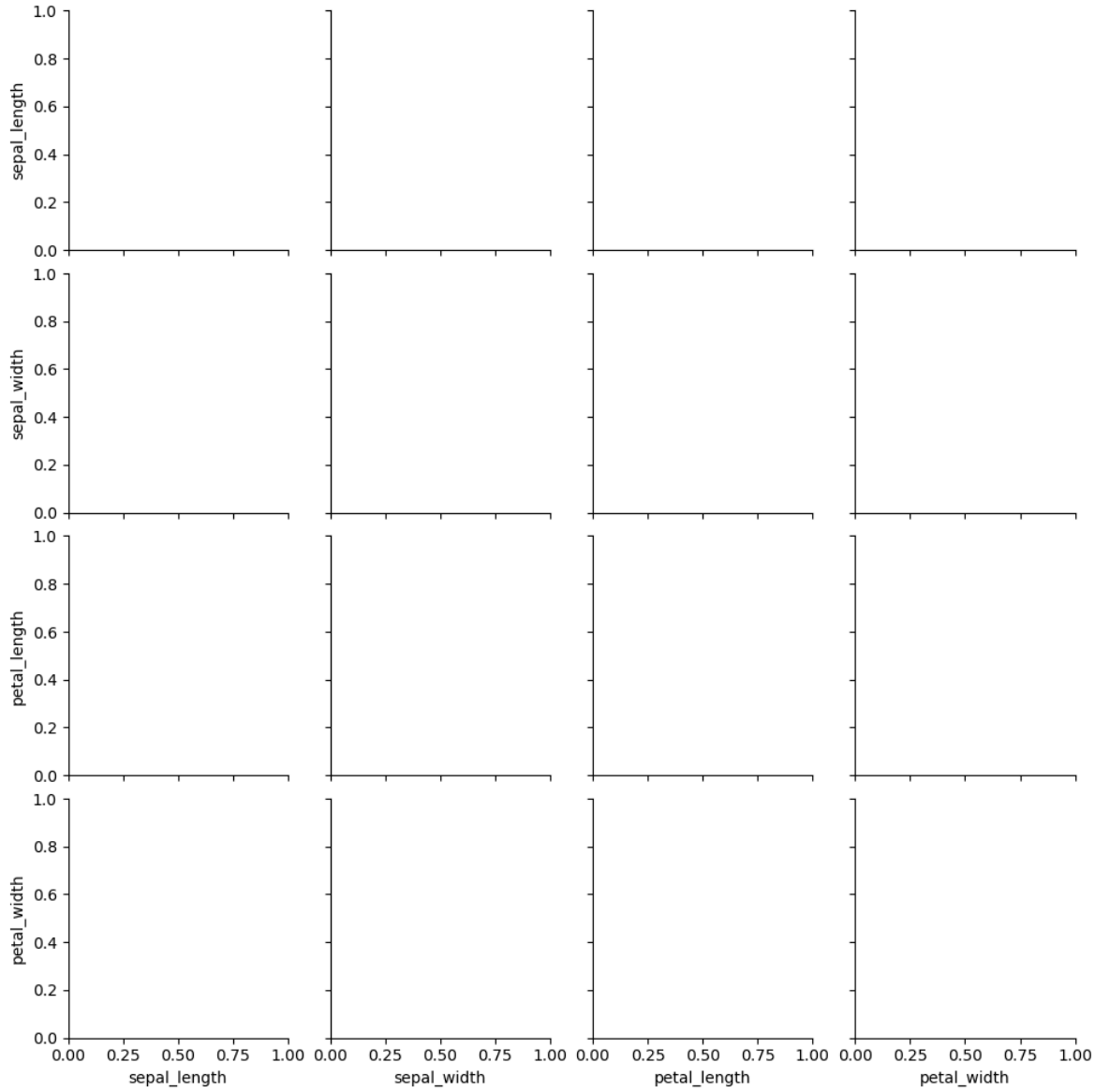
> I'm using the `iris` dataset to demostrate this.

> Iris dataset is a dataset of details about different species of iris flowers.

So, now let's make a `pair grid` plot.

In [51]:
```
sns.PairGrid(
    data=iris,
)
```

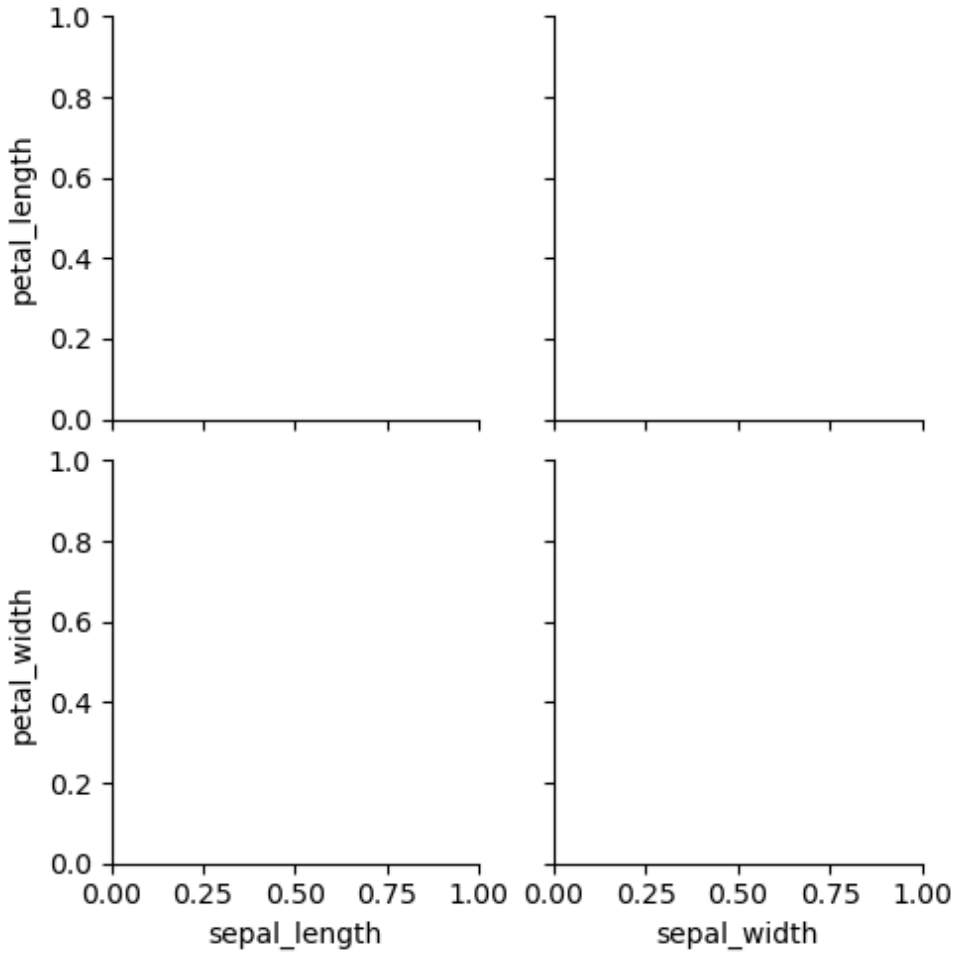
Out[51]:
```
<seaborn.axisgrid.PairGrid at 0x7ce67ce85110>
```



It automatically makes a grid plot of all the numeric columns in the dataset. But all the plots are empty. Before going into how we can customize it, I want to show you how we can set a grid for the columns we want to plot.

In [52]:
```
sns.PairGrid(
    data=iris,
    x_vars=['sepal_length', 'sepal_width'],
    y_vars=['petal_length', 'petal_width'],
    hue='species'
)
```

Out[52]: <seaborn.axisgrid.PairGrid at 0x7ce67cea8790>



> We can pass two arguments to `x_vars` and `y_vars` to specify the columns we want to plot in the x and y axes respectively.

This way we can get the specific plots we want.

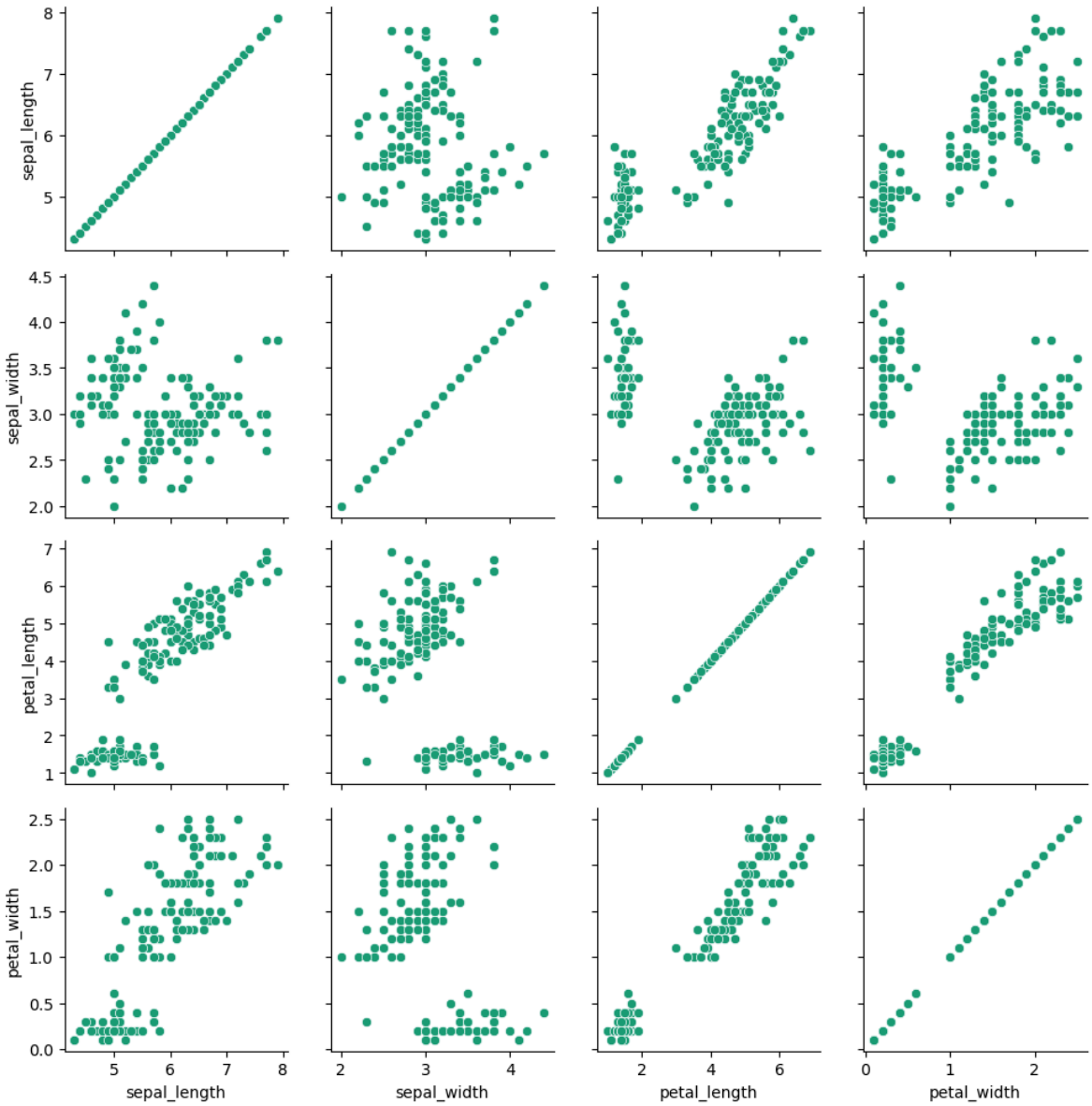
Now, how do we plot?

A parigrid returns a `PairGrid` object and that's object can be used to plot with mathods like `map` , `map_diag` and `map_upper` and `map_lower` methods.

In [53]:
```
grid = sns.PairGrid(
    data=iris
)

grid.map(sns.scatterplot)
```

Out[53]:
```
<seaborn.axisgrid.PairGrid at 0x7ce67dc07590>
```



Inside the map method we can pass any plot function we want and that will be applied to all the plots.

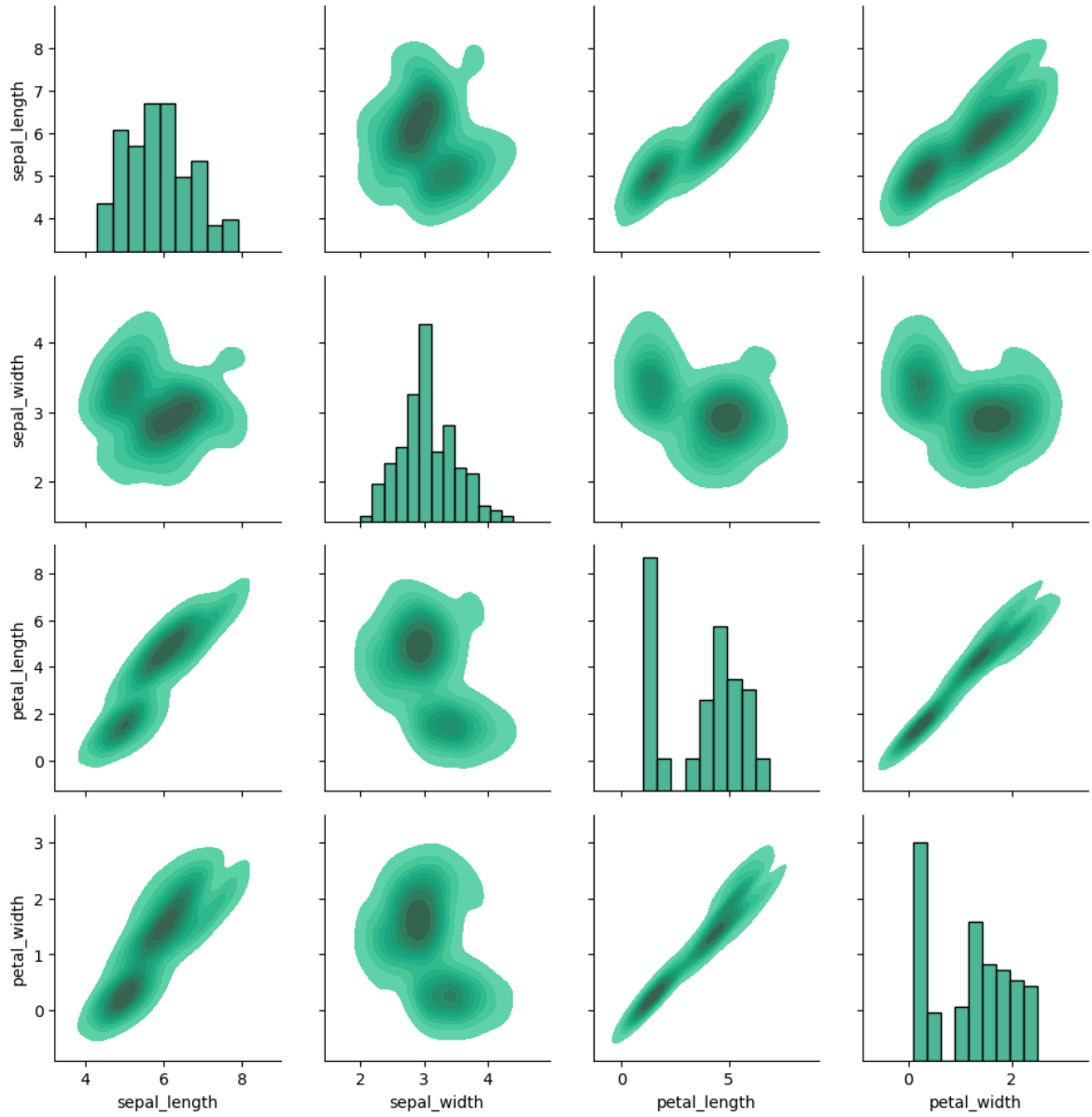
As you can see, I passed `sns.scatterplot` to the `map` method and the pair grid plot looks like this.

We can specify how the plots on the diagonal should look like and well as how the plots above and below the diagonal should look like.

In [54]:
```
grid = sns.PairGrid(
    data=iris
)

grid.map_diag(sns.histplot)
grid.map_lower(sns.kdeplot, fill=True)
grid.map_upper(sns.kdeplot, fill=True)
```

Out[54]:
```
<seaborn.axisgrid.PairGrid at 0x7ce67c0cbe50>
```



Looks nice right?

That's the beauty of grid plots.

There's another type of grid plot named `FacetGrid`.

FacetGrid

`Facet grid` is a grid plot where we can specify the ploting of a certain categorical variable on the x and y axes.

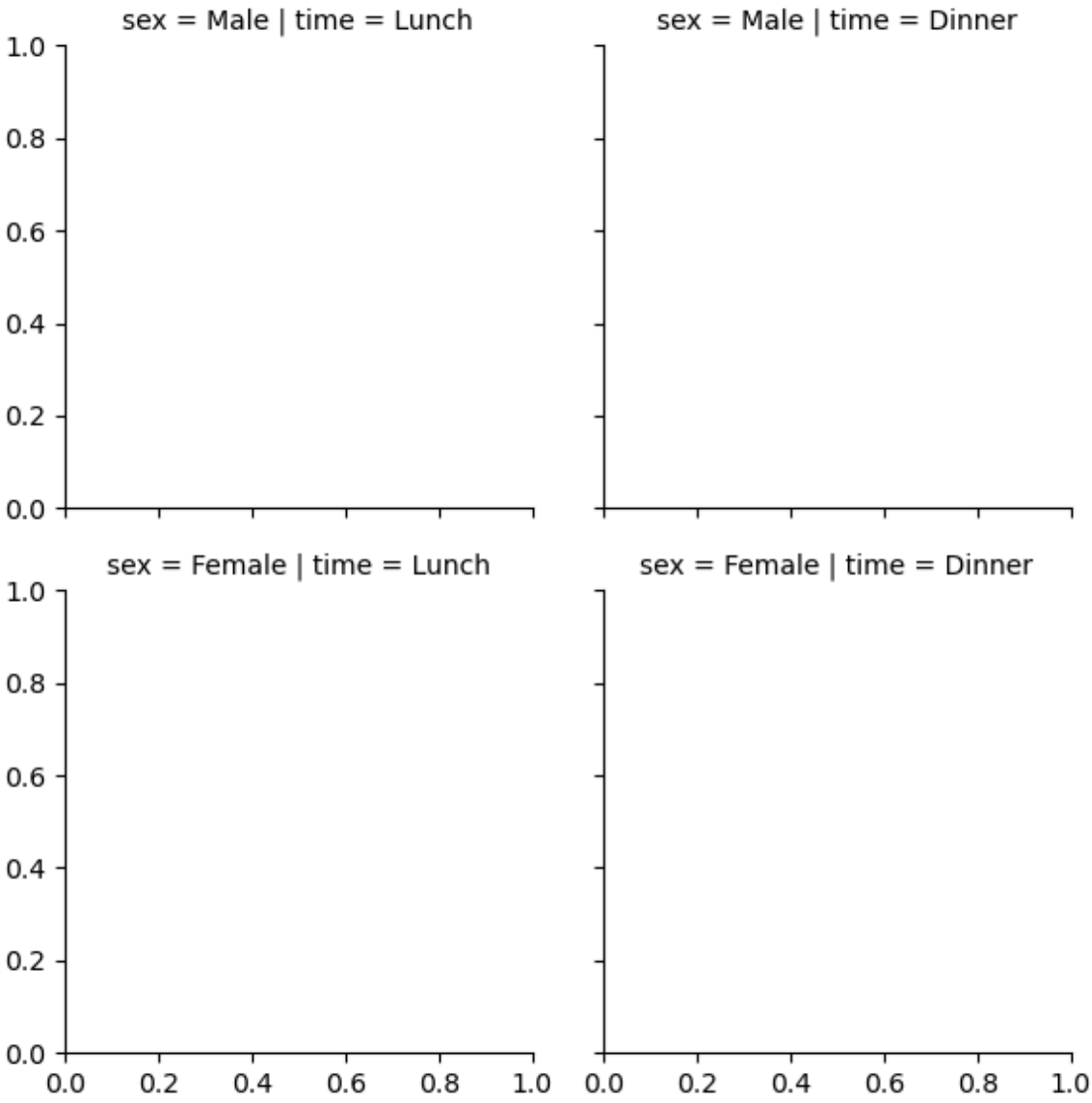
The grid will be configured in a way that the plots are grouped based on the value of the categorical variable.

And you can plot the numeric columns on the plots.

In [57]:
```
sns.FacetGrid(
    data=data, #using the tips dataset
    col='time',
```

```
    row='sex',
)
```

Out[57]: <seaborn.axisgrid.FacetGrid at 0x7ce66b48dd10>

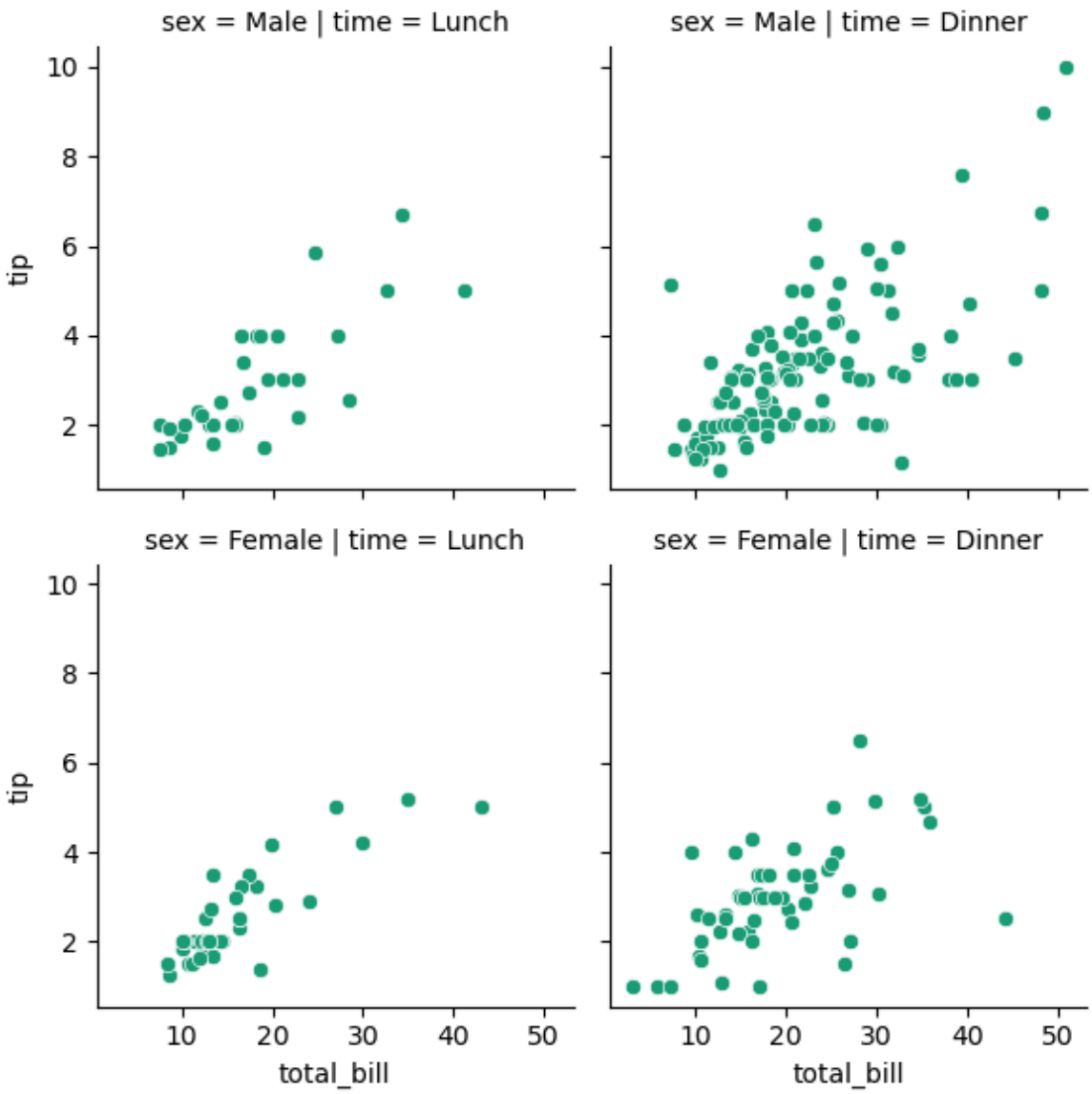


Each plot will is grouped and the numeric columns can be plotted like the pair grid plot.

In [58]:
```
g = sns.FacetGrid(
    data=data, #using the tips dataset
    col='time',
    row='sex',
)

g.map(sns.scatterplot, 'total_bill', 'tip')
```

Out[58]: <seaborn.axisgrid.FacetGrid at 0x7ce67d773a10>



This gives us a whole lot of information about the `total bill vs tip` of the people in the `tips` dataset.

And that's it! I think I've covered almost all the basic visualization stuff with `Seaborn` (exept `lmplot`, I'll get into it later on).

And Time to end this article with some styling tips.

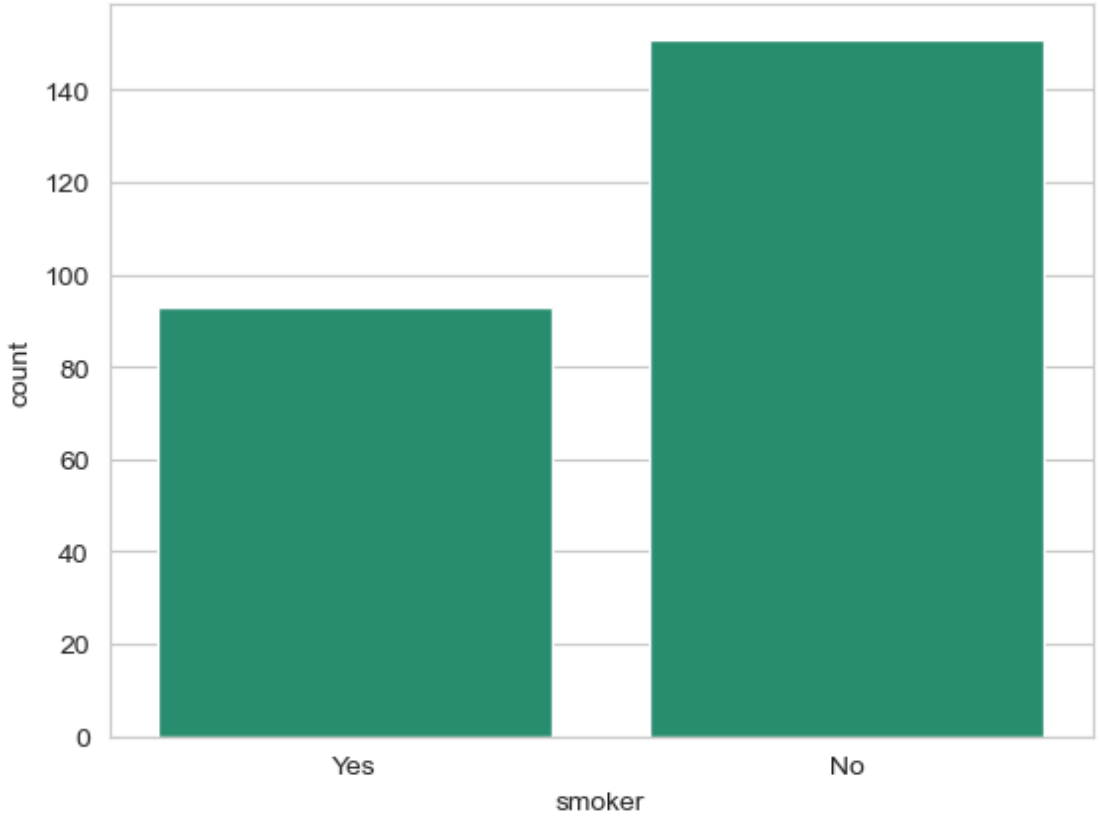
Styling tips

I talked about how we can set color palettes and how you can use matplotlib along with seaborn. There are some specific mathods like `set_style` and `set_context` that can be used to set the style of the plots.

So, try to use these methods before a plot method and see how they affect the plot.

In [60]:
```
sns.set_style('whitegrid')
sns.countplot(data=data, x='smoker')
```

Out[60]:
```
<Axes: xlabel='smoker', ylabel='count'>
```



You can also pass `white` to the `set_style` method. It'll make the background white.

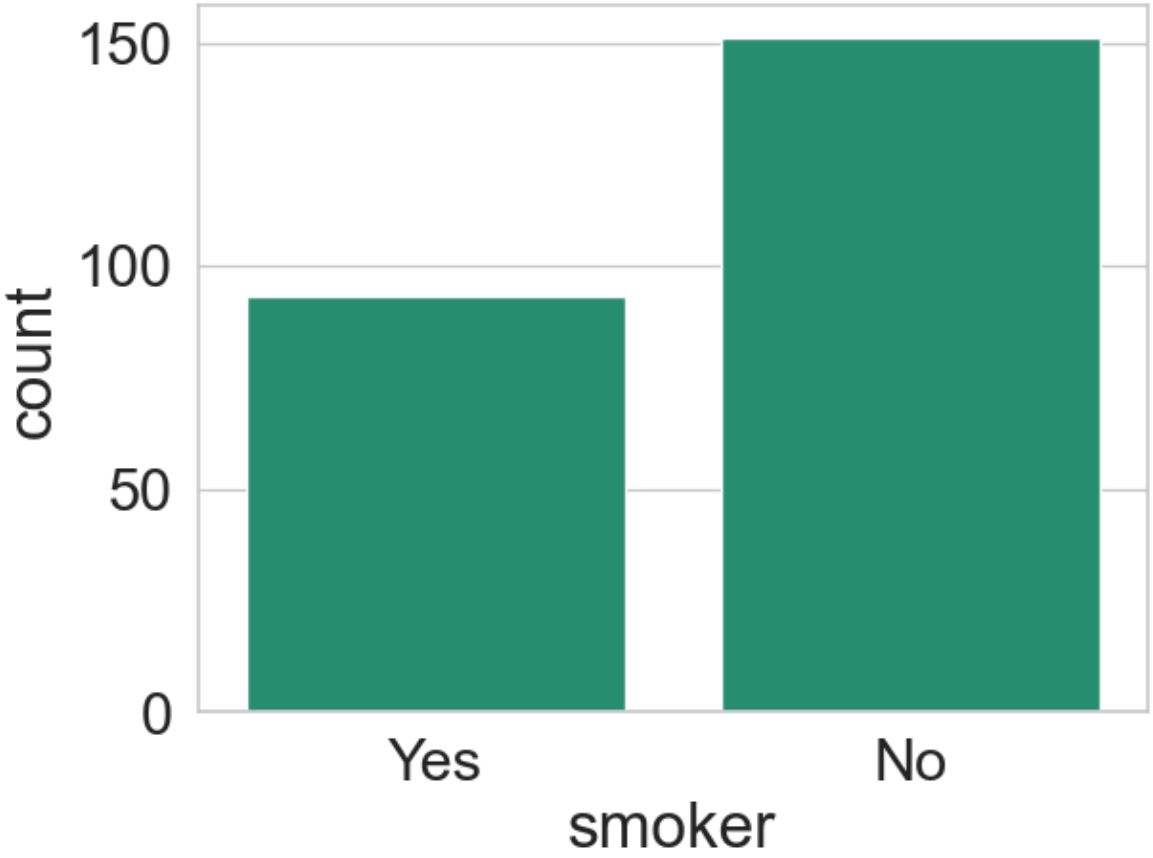
`dark` to make the background dark. `ticks` to make the ticks visible, `darkgrid` to make the grid dark and `whitegrid` to make the grid white.

Try them out.

Also, there is another method called `set_context` that can be used to set the size and contents of the plot.

In [63]:
```
sns.set_context('notebook', font_scale=2)
sns.countplot(data=data, x='smoker')
```

Out[63]:
```
<Axes: xlabel='smoker', ylabel='count'>
```



You can also pass `paper`, `talk` and `poster` to the `set_context` method for making the plot bigger or smaller. And font scale to make the font bigger or smaller.

Try it out and explore the documentations.

Final words

This was pain re-incarnated. I hope you enjoyed this article though.