**Lab Sheet 3**

**Try the following with any dataset provided:**

# Working with Missing Data in Pandas

Missing Data can occur when no information is provided for one or more items or for a whole unit. Missing Data is a very big problem in real life scenario. Missing Data can also refer to as NA(Not Available) values in pandas. In DataFrame sometimes many datasets simply arrive with missing data, either because it exists and was not collected or it never existed. For Example, Suppose different user being surveyed may choose not to share their income, some user may choose not to share the address in this way many datasets went missing.

In Pandas missing data is represented by two value:

* None: None is a Python singleton object that is often used for missing data in Python code.
* NaN : NaN (an acronym for Not a Number), is a special floating-point value recognized by all systems that use the standard IEEE floating-point representation

Pandas treat None and NaN as essentially interchangeable for indicating missing or null values. To facilitate this convention, there are several useful functions for detecting, removing, and replacing null values in Pandas DataFrame :

* [isnull()](https://www.geeksforgeeks.org/python-pandas-isnull-and-notnull/)
* [notnull()](https://www.geeksforgeeks.org/python-pandas-isnull-and-notnull/)
* [dropna()](https://www.geeksforgeeks.org/python-pandas-dataframe-dropna/)
* [fillna()](https://www.geeksforgeeks.org/python-pandas-dataframe-fillna-to-replace-null-values-in-dataframe/)
* [replace()](https://www.geeksforgeeks.org/python-pandas-dataframe-replace/)
* [interpolate()](https://www.geeksforgeeks.org/python-pandas-dataframe-interpolate/)

### Checking for missing values using isnull() and notnull()

In order to check missing values in Pandas DataFrame, we use a function isnull() and notnull(). Both function help in checking whether a value is NaN or not. These function can also be used in Pandas Series in order to find null values in a series.

#### Checking for missing values using isnull()

In order to check null values in Pandas DataFrame, we use isnull() function this function return dataframe of Boolean values which are True for NaN values.

**Code #1:**

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, 90, np.nan, 95],          'Second Score': [30, 45, 56, np.nan],          'Third Score':[np.nan, 40, 80, 98]}    # creating a dataframe from list  df = pd.DataFrame(dict)    # using isnull() function  df.isnull() |

**Code #2:**

|  |
| --- |
| # importing pandas package  import pandas as pd    # making data frame from csv file  data = pd.read\_csv("employees.csv")    # creating bool series True for NaN values  bool\_series = pd.isnull(data["Gender"])    # filtering data  # displaying data only with Gender = NaN  data[bool\_series] |

#### Checking for missing values using notnull()

In order to check null values in Pandas Dataframe, we use notnull() function this function return dataframe of Boolean values which are False for NaN values.

**Code #3:**

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, 90, np.nan, 95],          'Second Score': [30, 45, 56, np.nan],          'Third Score':[np.nan, 40, 80, 98]}    # creating a dataframe using dictionary  df = pd.DataFrame(dict)    # using notnull() function  df.notnull() |

**Code #4:**

|  |
| --- |
| # importing pandas package  import pandas as pd    # making data frame from csv file  data = pd.read\_csv("employees.csv")    # creating bool series True for NaN values  bool\_series = pd.notnull(data["Gender"])    # filtering data  # displayind data only with Gender = Not NaN  data[bool\_series] |

### Filling missing values using fillna(), replace() and interpolate()

In order to fill null values in a datasets, we use fillna(), replace() and interpolate() function these function replace NaN values with some value of their own. All these function help in filling a null values in datasets of a DataFrame. Interpolate() function is basically used to fill NA values in the dataframe but it uses various interpolation technique to fill the missing values rather than hard-coding the value.

**Code #1:** Filling null values with a single value

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, 90, np.nan, 95],          'Second Score': [30, 45, 56, np.nan],          'Third Score':[np.nan, 40, 80, 98]}    # creating a dataframe from dictionary  df = pd.DataFrame(dict)    # filling missing value using fillna()  df.fillna(0) |

   
**Code #2:** Filling null values with the previous ones

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, 90, np.nan, 95],          'Second Score': [30, 45, 56, np.nan],          'Third Score':[np.nan, 40, 80, 98]}    # creating a dataframe from dictionary  df = pd.DataFrame(dict)    # filling a missing value with  # previous ones  df.fillna(method ='pad') |

**Code #3:** Filling null value with the next ones

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, 90, np.nan, 95],          'Second Score': [30, 45, 56, np.nan],          'Third Score':[np.nan, 40, 80, 98]}    # creating a dataframe from dictionary  df = pd.DataFrame(dict)    # filling  null value using fillna() function  df.fillna(method ='bfill')  #df.fillna(median,inplace=True) |

**Code #4:** Filling null values in CSV File

|  |
| --- |
| # importing pandas package  import pandas as pd    # making data frame from csv file  data = pd.read\_csv("employees.csv")    # Printing the first 10 to 24 rows of  # the data frame for visualization  data[10:25] |

Now we are going to fill all the null values in Gender column with “No Gender”

**Code #5:** Filling a null values using replace() method

|  |
| --- |
| # importing pandas package  import pandas as pd    # making data frame from csv file  data = pd.read\_csv("employees.csv")    # Printing the first 10 to 24 rows of  # the data frame for visualization  data[10:25] |

Now we are going to replace the all Nan value in the data frame with -99 value.

|  |
| --- |
| # importing pandas package  import pandas as pd    # making data frame from csv file  data = pd.read\_csv("employees.csv")    # will replace  Nan value in dataframe with value -99  data.replace(to\_replace = np.nan, value = -99) |

**Code #6:** Using interpolate() function to fill the missing values using linear method.

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # Creating the dataframe  df = pd.DataFrame({"A":[12, 4, 5, None, 1],                     "B":[None, 2, 54, 3, None],                     "C":[20, 16, None, 3, 8],                     "D":[14, 3, None, None, 6]})    # Print the dataframe  df |

Let’s interpolate the missing values using Linear method. Note that Linear method ignore the index and treat the values as equally spaced.

|  |
| --- |
| # to interpolate the missing values  df.interpolate(method ='linear', limit\_direction ='forward') |

### Dropping missing values using dropna()

In order to drop a null values from a dataframe, we used dropna() function this function drop Rows/Columns of datasets with Null values in different ways.

**Code #1:** Dropping rows with at least 1 null value.

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, 90, np.nan, 95],          'Second Score': [30, np.nan, 45, 56],          'Third Score':[52, 40, 80, 98],          'Fourth Score':[np.nan, np.nan, np.nan, 65]}    # creating a dataframe from dictionary  df = pd.DataFrame(dict)    df |

Now we drop rows with at least one Nan value (Null value)

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, 90, np.nan, 95],          'Second Score': [30, np.nan, 45, 56],          'Third Score':[52, 40, 80, 98],          'Fourth Score':[np.nan, np.nan, np.nan, 65]}    # creating a dataframe from dictionary  df = pd.DataFrame(dict)    # using dropna() function  df.dropna() |

**Code #2:** Dropping rows if all values in that row are missing.

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, np.nan, np.nan, 95],          'Second Score': [30, np.nan, 45, 56],          'Third Score':[52, np.nan, 80, 98],          'Fourth Score':[np.nan, np.nan, np.nan, 65]}    # creating a dataframe from dictionary  df = pd.DataFrame(dict)    df |

Now we drop a rows whose all data is missing or contain null values(NaN)

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, np.nan, np.nan, 95],          'Second Score': [30, np.nan, 45, 56],          'Third Score':[52, np.nan, 80, 98],          'Fourth Score':[np.nan, np.nan, np.nan, 65]}    df = pd.DataFrame(dict)    # using dropna() function  df.dropna(how = 'all') |

**Code #3:** Dropping columns with at least 1 null value.

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, np.nan, np.nan, 95],          'Second Score': [30, np.nan, 45, 56],          'Third Score':[52, np.nan, 80, 98],          'Fourth Score':[60, 67, 68, 65]}    # creating a dataframe from dictionary  df = pd.DataFrame(dict)    df |

Now we drop a columns which have at least 1 missing values

|  |
| --- |
| # importing pandas as pd  import pandas as pd    # importing numpy as np  import numpy as np    # dictionary of lists  dict = {'First Score':[100, np.nan, np.nan, 95],          'Second Score': [30, np.nan, 45, 56],          'Third Score':[52, np.nan, 80, 98],          'Fourth Score':[60, 67, 68, 65]}    # creating a dataframe from dictionary  df = pd.DataFrame(dict)    # using dropna() function  df.dropna(axis = 1) |

**Code #4:** Dropping Rows with at least 1 null value in CSV file

|  |
| --- |
| # importing pandas module  import pandas as pd    # making data frame from csv file  data = pd.read\_csv("employees.csv")    # making new data frame with dropped NA values  new\_data = data.dropna(axis = 0, how ='any')  new\_data |

Now we compare sizes of data frames so that we can come to know how many rows had at least 1 Null value

|  |
| --- |
| print("Old data frame length:", len(data))  print("New data frame length:", len(new\_data))  print("Number of rows with at least 1 NA value: ", (len(data)-len(new\_data))) |

df = pd.DataFrame({'value': [1, np.nan, np.nan, 2, 3, 1, 3, np.nan, 3], 'name': ['A','A', 'B','B','B','B', 'C','C','C']})

df["value"] = df.groupby("name").transform(lambda x: x.fillna(x.mean()))

**Binning of data**

[**https://pbpython.com/pandas-qcut-cut.html**](https://pbpython.com/pandas-qcut-cut.html)

**Seaborn**

Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics. This article deals with the distribution plots in seaborn which is used for examining univariate and bivariate distributions. In this article we will be discussing 4 types of distribution plots namely:

1. joinplot
2. distplot
3. pairplot
4. rugplot

Besides providing different kinds of visualization plots, seaborn also contains some built-in datasets. We will be using the tips dataset in this article. The “tips” dataset contains information about people who probably had food at a restaurant and whether or not they left a tip, their age, gender and so on. Lets have a look at it.

**Code :**

total\_bill Total bill (cost of the meal), including tax, in US dollars

tip Tip (gratuity) in US dollars

sex Sex of person paying for the meal (0=male, 1=female)

Smoker Smoker in party? (0=No, 1=Yes)

Day 3=Thur, 4=Fri, 5=Sat, 6=Sun

Time 0=Day, 1=Night

Size Size of the party

|  |
| --- |
| # import the necessary libraries  import seaborn as sns  import matplotlib.pyplot as plt % matplotlib inline    # to ignore the warnings  from warnings import filterwarnings    # load the dataset  df = sns.load\_dataset('tips')    # the first five entries of the dataset  df.head() |

## Distplot

It is used basically for univariant set of observations and visualizes it through a histogram i.e. only one observation and hence we choose one particular column of the dataset.  
**Syntax:**

distplot(a[, bins, hist, kde, rug, fit, ...])

**Example:**

|  |
| --- |
| # set the background style of the plot  sns.set\_style('whitegrid')  sns.distplot(df['total\_bill'], kde = False, color ='red', bins = 30) |

**Explanation:**

* KDE stands for **Kernel Density Estimation** and that is another kind of the plot in seaborn.
* bins is used to set the number of bins you want in your plot and it actually depends on your dataset.
* color is used to specify the color of the plot

Now looking at this we can say that most of the total bill given lies between 10 and 20.

## Jointplot

It is used to draw a plot of two variables with bivariate and univariate graphs. It basically combines two different plots.  
**Syntax:**

jointplot(x, y[, data, kind, stat\_func, ...])

**Example:**

|  |
| --- |
| sns.jointplot(x ='total\_bill', y ='tip', data = df) |
| sns.jointplot(x ='total\_bill', y ='tip', data = df, kind ='kde')  # KDE shows the density where the points match up the most  **Explanation:**   * kind is a variable that helps us play around with the fact as to how do you want to visualise the data.It helps to see whats going inside the joinplot. The default is scatter and can be hex, reg(regression) or kde. * x and y are two strings that are the column names and the data that column contains is used by specifying the data parameter. * here we can see tips on the y axis and total bill on the x axis as well as a linear relationship between the two that suggests that the total bill increases with the tips.    Pairplot It represents pairwise relation across the entire dataframe and supports an additional argument called **hue** for categorical separation. What it does basically is create a jointplot between every possible numerical column and takes a while if the dataframe is really huge.  **Syntax:**  pairplot(data[, hue, hue\_order, palette, …])  **Example:**   |  | | --- | | sns.pairplot(df, hue ="sex", palette ='coolwarm') | |

**Explanation:**

* hue sets up the categorical separation between the entries if the dataset.
* palette is used for designing the plots.

## Rugplot

It plots datapoints in an array as sticks on an axis.Just like a distplot it takes a single column. Instead of drawing a histogram it creates dashes all across the plot. If you compare it with the joinplot you can see that what a jointplot does is that it counts the dashes and shows it as bins.

**Syntax:**

rugplot(a[, height, axis, ax])

**Example:**

|  |
| --- |
| sns.rugplot(df['total\_bill']) |

**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()](https://seaborn.pydata.org/generated/seaborn.barplot.html#seaborn.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, 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](https://seaborn.pydata.org/tutorial/categorical.html#categorical-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 from the dtype of the input variables, but can be used to specify when the “categorical” variable is a 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, optional

Colors to use for the different levels of the hue variable. Should be something that can be interpreted by [color\_palette()](https://seaborn.pydata.org/generated/seaborn.color_palette.html#seaborn.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()](https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.bar.html#matplotlib.axes.Axes.bar).

Returns

**ax**matplotlib Axes

Returns the Axes object with the plot drawn onto it.

Examples

Show value counts for a single categorical variable:

import seaborn as sns

sns.set(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()](https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.bar.html#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))

**seaborn.catplot**

seaborn.catplot(*x=None*, *y=None*, *hue=None*, *data=None*, *row=None*, *col=None*, *col\_wrap=None*, *estimator=<function mean at 0x10a2a03b0>*, *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*)

#Catplot

g = sns.catplot(x="class", hue="who", col="survived",

data=titanic, kind="count",

height=4, aspect=.7);

Figure-level interface for drawing categorical plots onto a [FacetGrid](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.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()](https://seaborn.pydata.org/generated/seaborn.stripplot.html#seaborn.stripplot) (with kind="strip"; the default)
* [swarmplot()](https://seaborn.pydata.org/generated/seaborn.swarmplot.html#seaborn.swarmplot) (with kind="swarm")

Categorical distribution plots:

* [boxplot()](https://seaborn.pydata.org/generated/seaborn.boxplot.html#seaborn.boxplot) (with kind="box")
* [violinplot()](https://seaborn.pydata.org/generated/seaborn.violinplot.html#seaborn.violinplot) (with kind="violin")
* [boxenplot()](https://seaborn.pydata.org/generated/seaborn.boxenplot.html#seaborn.boxenplot) (with kind="boxen")

Categorical estimate plots:

* [pointplot()](https://seaborn.pydata.org/generated/seaborn.pointplot.html#seaborn.pointplot) (with kind="point")
* [barplot()](https://seaborn.pydata.org/generated/seaborn.barplot.html#seaborn.barplot) (with kind="bar")
* [countplot()](https://seaborn.pydata.org/generated/seaborn.countplot.html#seaborn.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 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.

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, optional

“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**string, optional

The kind of plot to draw (corresponds to the name of a categorical plotting function. Options are: “point”, “bar”, “strip”, “swarm”, “box”, “violin”, or “boxen”.

**height**scalar, optional

Height (in inches) of each facet. See also: aspect.

**aspect**scalar, optional

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 from the dtype of the input variables, but can be used to specify when the “categorical” variable is a 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, optional

Colors to use for the different levels of the hue variable. Should be something that can be interpreted by [color\_palette()](https://seaborn.pydata.org/generated/seaborn.color_palette.html#seaborn.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, optional

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, optional

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](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid).

**kwargs**key, value pairings

Other keyword arguments are passed through to the underlying plotting function.

Examples

import seaborn as sns

sns.set(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("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:

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](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.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...>

# Plotting with categorical data

In the [relational plot tutorial](https://seaborn.pydata.org/tutorial/relational.html#relational-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.

In seaborn, there are several different ways to visualize a relationship involving categorical data. Similar to the relationship between [relplot()](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot) and either [scatterplot()](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot) or [lineplot()](https://seaborn.pydata.org/generated/seaborn.lineplot.html#seaborn.lineplot), there are two ways to make these plots. There are a number of axes-level functions for plotting categorical data in different ways and a figure-level interface, [catplot()](https://seaborn.pydata.org/generated/seaborn.catplot.html#seaborn.catplot), that gives unified higher-level access to them.

It’s helpful to think of the different categorical plot kinds as belonging to three different families, which we’ll discuss in detail below. They are:

Categorical scatterplots:

* [stripplot()](https://seaborn.pydata.org/generated/seaborn.stripplot.html#seaborn.stripplot) (with kind="strip"; the default)
* [swarmplot()](https://seaborn.pydata.org/generated/seaborn.swarmplot.html#seaborn.swarmplot) (with kind="swarm")

Categorical distribution plots:

* [boxplot()](https://seaborn.pydata.org/generated/seaborn.boxplot.html#seaborn.boxplot) (with kind="box")
* [violinplot()](https://seaborn.pydata.org/generated/seaborn.violinplot.html#seaborn.violinplot) (with kind="violin")
* [boxenplot()](https://seaborn.pydata.org/generated/seaborn.boxenplot.html#seaborn.boxenplot) (with kind="boxen")

Categorical estimate plots:

* [pointplot()](https://seaborn.pydata.org/generated/seaborn.pointplot.html#seaborn.pointplot) (with kind="point")
* [barplot()](https://seaborn.pydata.org/generated/seaborn.barplot.html#seaborn.barplot) (with kind="bar")
* [countplot()](https://seaborn.pydata.org/generated/seaborn.countplot.html#seaborn.countplot) (with kind="count")

These families represent the data using different levels of granularity. When deciding which to use, you’ll have to think about the question that you want to answer. The unified API makes it easy to switch between different kinds and see your data from several perspectives.

import seaborn as sns

import matplotlib.pyplot as plt

sns.set(style="ticks", color\_codes=True)

## Categorical scatterplots

The default representation of the data in [catplot()](https://seaborn.pydata.org/generated/seaborn.catplot.html#seaborn.catplot) uses a scatterplot. There are actually two different categorical scatter plots in seaborn. They take different approaches to resolving the main challenge in representing categorical data with a scatter plot, which is that all of the points belonging to one category would fall on the same position along the axis corresponding to the categorical variable. The approach used by [stripplot()](https://seaborn.pydata.org/generated/seaborn.stripplot.html#seaborn.stripplot), which is the default “kind” in [catplot()](https://seaborn.pydata.org/generated/seaborn.catplot.html#seaborn.catplot) is to adjust the positions of points on the categorical axis with a small amount of random “jitter”:

tips = sns.load\_dataset("tips")

sns.catplot(x="day", y="total\_bill", data=tips);

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

sns.catplot(x="day", y="total\_bill", jitter=False, data=tips);

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()](https://seaborn.pydata.org/generated/seaborn.swarmplot.html#seaborn.swarmplot), which is activated by setting kind="swarm" in [catplot()](https://seaborn.pydata.org/generated/seaborn.catplot.html#seaborn.catplot):

sns.catplot(x="day", y="total\_bill", kind="swarm", data=tips);

Similar to the relational plots, it’s possible to add another dimension to a categorical plot by using a hue semantic. (The categorical plots do not currently support size or style semantics). Each different categorical plotting function handles the hue semantic differently. For the scatter plots, it is only necessary to change the color of the points:

sns.catplot(x="day", y="total\_bill", hue="sex", kind="swarm", data=tips);

Unlike with numerical data, it is not always obvious how to order the levels of the categorical variable along its axis. In general, the seaborn categorical plotting functions try to infer the order of categories from the data. If your data have a pandas Categorical datatype, then the default order of the categories can be set there. If the variable passed to the categorical axis looks numerical, the levels will be sorted. But the data are still treated as categorical and drawn at ordinal positions on the categorical axes (specifically, at 0, 1, …) even when numbers are used to label them:

sns.catplot(x="size", y="total\_bill", kind="swarm",

data=tips.query("size != 3"));

The other option for choosing a default ordering is to take the levels of the category as they appear in the dataset. The ordering can also be controlled on a plot-specific basis using the order parameter. This can be important when drawing multiple categorical plots in the same figure, which we’ll see more of below:

sns.catplot(x="smoker", y="tip", order=["No", "Yes"], data=tips);

We’ve referred to the idea of “categorical axis”. In these examples, that’s always corresponded to the horizontal axis. But it’s often helpful to put the categorical variable on the vertical axis (particularly when the category names are relatively long or there are many categories). To do this, swap the assignment of variables to axes:

sns.catplot(x="total\_bill", y="day", hue="time", kind="swarm", data=tips);

## 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

The first is the familiar [boxplot()](https://seaborn.pydata.org/generated/seaborn.boxplot.html#seaborn.boxplot). This kind of plot shows the three quartile values of the distribution along with extreme values. The “whiskers” extend to points that lie within 1.5 IQRs of the lower and upper quartile, and then observations that fall outside this range are displayed independently. This means that each value in the boxplot corresponds to an actual observation in the data.

sns.catplot(x="day", y="total\_bill", kind="box", data=tips);

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:

sns.catplot(x="day", y="total\_bill", hue="smoker", kind="box", data=tips);

This behavior is called “dodging” and is turned on by default because it is assumed that the semantic variable is nested within the main categorical variable. If that’s not the case, you can disable the dodging:

tips["weekend"] = tips["day"].isin(["Sat", "Sun"])

sns.catplot(x="day", y="total\_bill", hue="weekend",

kind="box", dodge=False, data=tips);

A related function, [boxenplot()](https://seaborn.pydata.org/generated/seaborn.boxenplot.html#seaborn.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:

diamonds = sns.load\_dataset("diamonds")

sns.catplot(x="color", y="price", kind="boxen",

data=diamonds.sort\_values("color"));

### Violinplots

A different approach is a [violinplot()](https://seaborn.pydata.org/generated/seaborn.violinplot.html#seaborn.violinplot), which combines a boxplot with the kernel density estimation procedure described in the [distributions](https://seaborn.pydata.org/tutorial/distributions.html#distribution-tutorial) tutorial:

sns.catplot(x="total\_bill", y="day", hue="sex",

kind="violin", data=tips);

This approach uses the kernel density estimate to provide a richer description of the distribution of values. Additionally, the quartile and whisker values from the boxplot are shown inside the violin. The downside is that, because the violinplot uses a KDE, there are some other parameters that may need tweaking, adding some complexity relative to the straightforward boxplot:

sns.catplot(x="total\_bill", y="day", hue="sex",

kind="violin", bw=.15, cut=0,

data=tips);

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:

sns.catplot(x="day", y="total\_bill", hue="sex",

kind="violin", split=True, data=tips);

Finally, there are several options for the plot that is drawn on the interior of the violins, including ways to show each individual observation instead of the summary boxplot values:

sns.catplot(x="day", y="total\_bill", hue="sex",

kind="violin", inner="stick", split=True,

palette="pastel", data=tips);

It can also be useful to combine [swarmplot()](https://seaborn.pydata.org/generated/seaborn.swarmplot.html#seaborn.swarmplot) or striplot() with a box plot or violin plot to show each observation along with a summary of the distribution:

g = sns.catplot(x="day", y="total\_bill", kind="violin", inner=None, data=tips)

sns.swarmplot(x="day", y="total\_bill", color="k", size=3, data=tips, ax=g.ax);

## Statistical estimation within categories

For other applications, rather than showing the distribution within each category, you might want to show an estimate of the central tendency of the values. Seaborn has two main ways to show this information. Importantly, the basic API for these functions is identical to that for the ones discussed above.

### Bar plots

A familiar style of plot that accomplishes this goal is a bar plot. In seaborn, the [barplot()](https://seaborn.pydata.org/generated/seaborn.barplot.html#seaborn.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:

titanic = sns.load\_dataset("titanic")

sns.catplot(x="sex", y="survived", hue="class", kind="bar", data=titanic);

A special case for the bar plot is when you want to show the number of observations in each category rather than computing a statistic for a second variable. This is similar to a histogram over a categorical, rather than quantitative, variable. In seaborn, it’s easy to do so with the [countplot()](https://seaborn.pydata.org/generated/seaborn.countplot.html#seaborn.countplot) function:

sns.catplot(x="deck", kind="count", palette="ch:.25", data=titanic);

Both [barplot()](https://seaborn.pydata.org/generated/seaborn.barplot.html#seaborn.barplot) and [countplot()](https://seaborn.pydata.org/generated/seaborn.countplot.html#seaborn.countplot) can be invoked with all of the options discussed above, along with others that are demonstrated in the detailed documentation for each function:

sns.catplot(y="deck", hue="class", kind="count",

palette="pastel", edgecolor=".6",

data=titanic);

### Point plots

An alternative style for visualizing the same information is offered by the [pointplot()](https://seaborn.pydata.org/generated/seaborn.pointplot.html#seaborn.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()](https://seaborn.pydata.org/generated/seaborn.pointplot.html#seaborn.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:

sns.catplot(x="sex", y="survived", hue="class", kind="point", data=titanic);

While the categorical functions lack the style semantic of the relational functions, it can still be a good idea to vary the marker and/or linestyle along with the hue to make figures that are maximally accessible and reproduce well in black and white:

sns.catplot(x="class", y="survived", hue="sex",

palette={"male": "g", "female": "m"},

markers=["^", "o"], linestyles=["-", "--"],

kind="point", data=titanic);

## Plotting “wide-form” data

While using “long-form” or “tidy” data is preferred, these functions can also by applied to “wide-form” data in a variety of formats, including pandas DataFrames or two-dimensional numpy arrays. These objects should be passed directly to the data parameter:

iris = sns.load\_dataset("iris")

sns.catplot(data=iris, orient="h", kind="box");

Additionally, the axes-level functions accept vectors of Pandas or numpy objects rather than variables in a DataFrame:

sns.violinplot(x=iris.species, y=iris.sepal\_length);

To control the size and shape of plots made by the functions discussed above, you must set up the figure yourself using matplotlib commands:

f, ax = plt.subplots(figsize=(7, 3))

sns.countplot(y="deck", data=titanic, color="c");

This is the approach you should take when you need a categorical figure to happily coexist in a more complex figure with other kinds of plots.

## Showing multiple relationships with facets

Just like [relplot()](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot), the fact that [catplot()](https://seaborn.pydata.org/generated/seaborn.catplot.html#seaborn.catplot) is built on a [FacetGrid](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid) means that it is easy to add faceting variables to visualize higher-dimensional relationships:

sns.catplot(x="day", y="total\_bill", hue="smoker",

col="time", aspect=.6,

kind="swarm", data=tips);

For further customization of the plot, you can use the methods on the [FacetGrid](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid) object that it returns:

g = sns.catplot(x="fare", y="survived", row="class",

kind="box", orient="h", height=1.5, aspect=4,

data=titanic.query("fare > 0"))

g.set(xscale="log");

# Visualizing statistical relationships

Statistical analysis is a process of understanding how variables in a dataset relate to each other and how those relationships depend on other variables. Visualization can be a core component of this process because, when data are visualized properly, the human visual system can see trends and patterns that indicate a relationship.

We will discuss three seaborn functions in this tutorial. The one we will use most is [relplot()](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot). This is a [figure-level function](https://seaborn.pydata.org/introduction.html#intro-func-types) for visualizing statistical relationships using two common approaches: scatter plots and line plots. [relplot()](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot) combines a [FacetGrid](https://seaborn.pydata.org/generated/seaborn.FacetGrid.html#seaborn.FacetGrid) with one of two axes-level functions:

* [scatterplot()](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot) (with kind="scatter"; the default)
* [lineplot()](https://seaborn.pydata.org/generated/seaborn.lineplot.html#seaborn.lineplot) (with kind="line")

As we will see, 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.

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

sns.set(style="darkgrid")

## Relating variables with scatter plots

The scatter plot is a mainstay of statistical visualization. It depicts the joint distribution of two variables using a cloud of points, where each point represents an observation in the dataset. This depiction allows the eye to infer a substantial amount of information about whether there is any meaningful relationship between them.

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()](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot) function. In the [categorical visualization tutorial](https://seaborn.pydata.org/tutorial/categorical.html#categorical-tutorial), we will see specialized tools for using scatterplots to visualize categorical data. The [scatterplot()](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot) is the default kind in [relplot()](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot) (it can also be forced by setting kind="scatter"):

tips = sns.load\_dataset("tips")

sns.relplot(x="total\_bill", y="tip", data=tips);

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:

sns.relplot(x="total\_bill", y="tip", hue="smoker", data=tips);

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

sns.relplot(x="total\_bill", y="tip", hue="smoker", style="smoker",

data=tips);

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:

sns.relplot(x="total\_bill", y="tip", hue="smoker", style="time", data=tips);

In the examples above, the hue semantic was categorical, so the default [qualitative palette](https://seaborn.pydata.org/tutorial/color_palettes.html#palette-tutorial) was applied. If the hue semantic is numeric (specifically, if it can be cast to float), the default coloring switches to a sequential palette:

sns.relplot(x="total\_bill", y="tip", hue="size", data=tips);

In both cases, you can customize the color palette. There are many options for doing so. Here, we customize a sequential palette using the string interface to [cubehelix\_palette()](https://seaborn.pydata.org/generated/seaborn.cubehelix_palette.html#seaborn.cubehelix_palette):

sns.relplot(x="total\_bill", y="tip", hue="size", palette="ch:r=-.5,l=.75", data=tips);

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

sns.relplot(x="total\_bill", y="tip", size="size", data=tips);

Unlike with [matplotlib.pyplot.scatter()](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html#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:

sns.relplot(x="total\_bill", y="tip", size="size", sizes=(15, 200), data=tips);

More examples for customizing how the different semantics are used to show statistical relationships are shown in the [scatterplot()](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot) API examples.

## Emphasizing continuity with line plots

Scatter plots are highly effective, but there is no universally optimal type of visualisation. Instead, the visual representation should be adapted for the specifics of the dataset and to the question you are trying to answer with the plot.

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()](https://seaborn.pydata.org/generated/seaborn.lineplot.html#seaborn.lineplot) function, either directly or with [relplot()](https://seaborn.pydata.org/generated/seaborn.relplot.html#seaborn.relplot) by setting kind="line":

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()

Because [lineplot()](https://seaborn.pydata.org/generated/seaborn.lineplot.html#seaborn.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:

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);

### 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:

fmri = sns.load\_dataset("fmri")

sns.relplot(x="timepoint", y="signal", kind="line", data=fmri);

The confidence intervals are computed using bootstrapping, which can be time-intensive for larger datasets. It’s therefore possible to disable them:

sns.relplot(x="timepoint", y="signal", ci=None, kind="line", data=fmri);

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:

sns.relplot(x="timepoint", y="signal", kind="line", ci="sd", data=fmri);

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.

sns.relplot(x="timepoint", y="signal", estimator=None, kind="line", data=fmri);

### Plotting subsets of data with semantic mappings

The [lineplot()](https://seaborn.pydata.org/generated/seaborn.lineplot.html#seaborn.lineplot) function has the same flexibility as [scatterplot()](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot): it can show up to three additional variables by modifying the hue, size, and style of the plot elements. It does so using the same API as [scatterplot()](https://seaborn.pydata.org/generated/seaborn.scatterplot.html#seaborn.scatterplot), meaning that we don’t need to stop and think about the parameters that control the look of lines vs. points in matplotlib.

Using semantics in [lineplot()](https://seaborn.pydata.org/generated/seaborn.lineplot.html#seaborn.lineplot) will also determine how the data get aggregated. For example, adding a hue semantic with two levels splits the plot into two lines and error bands, coloring each to indicate which subset of the data they correspond to.

sns.relplot(x="timepoint", y="signal", hue="event", kind="line", data=fmri);

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

sns.relplot(x="timepoint", y="signal", hue="region", style="event",

kind="line", data=fmri);

But you can identify subsets by the markers used at each observation, either together with the dashes or instead of them:

sns.relplot(x="timepoint", y="signal", hue="region", style="event",

dashes=False, markers=True, kind="line", data=fmri);

As with scatter plots, be cautious about making line plots using multiple semantics. While sometimes informative, they can also be difficult to parse and interpret. But even when you are only examining changes across one additional variable, it can be useful to alter both the color and style of the lines. This can make the plot more accessible when printed to black-and-white or viewed by someone with color blindness:

sns.relplot(x="timepoint", y="signal", hue="event", style="event",

kind="line", data=fmri);

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:

sns.relplot(x="timepoint", y="signal", hue="region",

units="subject", estimator=None,

kind="line", data=fmri.query("event == 'stim'"));

The default colormap and handling of the legend in [lineplot()](https://seaborn.pydata.org/generated/seaborn.lineplot.html#seaborn.lineplot) also depends on whether the hue semantic is categorical or numeric:

dots = sns.load\_dataset("dots").query("align == 'dots'")

sns.relplot(x="time", y="firing\_rate",

hue="coherence", style="choice",

kind="line", data=dots);

It may happen that, even though the hue variable is numeric, it is poorly represented by a linear color scale. That’s the case here, where the levels of the hue variable are logarithmically scaled. You can provide specific color values for each line by passing a list or dictionary:

palette = sns.cubehelix\_palette(light=.8, n\_colors=6)

sns.relplot(x="time", y="firing\_rate",

hue="coherence", style="choice",

palette=palette,

kind="line", data=dots);

Or you can alter how the colormap is normalized:

from matplotlib.colors import LogNorm

palette = sns.cubehelix\_palette(light=.7, n\_colors=6)

sns.relplot(x="time", y="firing\_rate",

hue="coherence", style="choice",

hue\_norm=LogNorm(),

kind="line", data=dots);

The third semantic, size, changes the width of the lines:

sns.relplot(x="time", y="firing\_rate",

size="coherence", style="choice",

kind="line", data=dots);

While the size variable will typically be numeric, it’s also possible to map a categorical variable with the width of the lines. Be cautious when doing so, because it will be difficult to distinguish much more than “thick” vs “thin” lines. However, dashes can be hard to perceive when lines have high-frequency variability, so using different widths may be more effective in that case:

sns.relplot(x="time", y="firing\_rate",

hue="coherence", size="choice",

palette=palette,

kind="line", data=dots);

### Plotting with date data

Line plots are often used to visualize data associated with real dates and times. These functions pass the data down in their original format to the underlying matplotlib functions, and so they can take advantage of matplotlib’s ability to format dates in tick labels. But all of that formatting will have to take place at the matplotlib layer, and you should refer to the matplotlib documentation to see how it works:

df = pd.DataFrame(dict(time=pd.date\_range("2017-1-1", periods=500),

value=np.random.randn(500).cumsum()))

g = sns.relplot(x="time", y="value", kind="line", data=df)

g.fig.autofmt\_xdate()

/Users/mwaskom/miniconda3/envs/seaborn-py37-latest/lib/python3.7/site-packages/pandas/plotting/\_matplotlib/converter.py:103: FutureWarning: Using an implicitly registered datetime converter for a matplotlib plotting method. The converter was registered by pandas on import. Future versions of pandas will require you to explicitly register matplotlib converters.

To register the converters:

>>> from pandas.plotting import register\_matplotlib\_converters

>>> register\_matplotlib\_converters()

warnings.warn(msg, FutureWarning)

## Showing multiple relationships with facets

We’ve emphasized in this tutorial that, while these functions can show several semantic variables at once, it’s not always effective to do so. But what about when you do want to understand how a relationship between two variables depends on more than one other variable?

sns.relplot(x="total\_bill", y="tip", hue="smoker",

col="time", data=tips);

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.

sns.relplot(x="timepoint", y="signal", hue="subject",

col="region", row="event", height=3,

kind="line", estimator=None, data=fmri);

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:

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'"));

**Heatmaps**   
Heatmap is a way to show some sort of matrix plot. To use a heatmap the data should be in a matrix form. By matrix we mean that the index name and the column name must match in some way so that the data that we fill inside the cells are relevant. Lets look at an example to understand this better.

**Code : Python program**

|  |
| --- |
| # import the necessary libraries  import seaborn as sns  import matplotlib.pyplot as plt % matplotlib inline    # load the tips dataset  dataset = sns.load\_dataset('tips')    # first five entries of the tips dataset  dataset.head()    # correlation between the different parameters  tc = dataset.corr()    # plot a heatmap of the correlated data  sns.heatmap(tc) |

**Heatmap of the correlated matrix**  
Inorder to obatin a better visualisation with the heatmap, we can add the parameters such as annot, linewidth and line colour.

|  |
| --- |
| # import the necessary libraries  import seaborn as sns  import matplotlib.pyplot as plt % matplotlib inline    # load the tips dataset  dataset = sns.load\_dataset('tips')    # first five entries of the tips dataset  dataset.head()    # correlation between the different parameters  tc = dataset.corr()  sns.heatmap(tc, annot = True, cmap ='plasma',linecolor ='black', linewidths = 1) |

**Explanation**

* annot is used to annotate the actual value that belongs to these cells
* cmap is used for the colour mapping you want like coolwarm, plasma, magma etc.
* linewidth is used to set the width of the lines separating the cells.
* linecolor is used to set the colour of the lines separating the cells.

**Cluster maps**   
Cluster maps use hierarchical clustering. It performs the clustering based on the similarity of the rows and columns.

|  |
| --- |
| # import the necessary libraries  import pandas as pd  import seaborn as sns  import matplotlib.pyplot as plt % matplotlib inline    # load the flights dataset  fd = sns.load\_dataset('flights')    # make a dataframe of the data  df = pd.pivot\_table(values ='passengers', index ='month',                      columns ='year', data = fd)    # first five entries of the dataset  df.head()    # make a clustermap from the dataset  sns.clustermap(df, cmap ='plasma') |

**Clustermap from the given data**  
We can also change the scale of the color bar by using the standard\_scale parameter.

|  |
| --- |
| # import the necessary libraries  import pandas as pd  import seaborn as sns  import matplotlib.pyplot as plt % matplotlib inline    # load the flights dataset  fd = sns.load\_dataset('flights')    # make a dataframe of the data  df = pd.pivot\_table(values ='passengers',                      index ='month', columns ='year', data = fd)    # first five entries of the dataset  df.head()    # make a clustermap from the dataset  sns.clustermap(df, cmap ='plasma', standard\_scale = 1) |

**Clustermap after using standard scaling**  
**standard\_scale = 1** normalises the data from 0 to 1 range. We can see that the months as well as years are no longer in order as they are clustered according to the similarity in case of clustermaps.

**3-D plotting**

We can easily plot 3-D figures in matplotlib. Now, we discuss some important and commonly used 3-D plots.

|  |
| --- |
| from mpl\_toolkits.mplot3d import axes3d  import matplotlib.pyplot as plt  from matplotlib import style  import numpy as np    # setting a custom style to use  style.use('ggplot')    # create a new figure for plotting  fig = plt.figure()    # create a new subplot on our figure  # and set projection as 3d  ax1 = fig.add\_subplot(111, projection='3d')    # defining x, y, z co-ordinates  x = np.random.randint(0, 10, size = 20)  y = np.random.randint(0, 10, size = 20)  z = np.random.randint(0, 10, size = 20)    # plotting the points on subplot      # setting labels for the axes  ax1.set\_xlabel('x-axis')  ax1.set\_ylabel('y-axis')  ax1.set\_zlabel('z-axis') |

# function to show the plot

plt.show()

**Plotting lines**

|  |
| --- |
| # importing required modules  from mpl\_toolkits.mplot3d import axes3d  import matplotlib.pyplot as plt  from matplotlib import style  import numpy as np    # setting a custom style to use  style.use('ggplot')    # create a new figure for plotting  fig = plt.figure()    # create a new subplot on our figure  ax1 = fig.add\_subplot(111, projection='3d')    # defining x, y, z co-ordinates  x = np.random.randint(0, 10, size = 5)  y = np.random.randint(0, 10, size = 5)  z = np.random.randint(0, 10, size = 5)    # plotting the points on subplot  ax1.plot\_wireframe(x,y,z)    # setting the labels  ax1.set\_xlabel('x-axis')  ax1.set\_ylabel('y-axis')  ax1.set\_zlabel('z-axis')    plt.show() |

**Plotting Bars**

|  |
| --- |
| # importing required modules  from mpl\_toolkits.mplot3d import axes3d  import matplotlib.pyplot as plt  from matplotlib import style  import numpy as np    # setting a custom style to use  style.use('ggplot')    # create a new figure for plotting  fig = plt.figure()    # create a new subplot on our figure  ax1 = fig.add\_subplot(111, projection='3d')    # defining x, y, z co-ordinates for bar position  x = [1,2,3,4,5,6,7,8,9,10]  y = [4,3,1,6,5,3,7,5,3,7]  z = np.zeros(10)    # size of bars  dx = np.ones(10)              # length along x-axis  dy = np.ones(10)              # length along y-axs  dz = [1,3,4,2,6,7,5,5,10,9]   # height of bar    # setting color scheme  color = []  for h in dz:      if h > 5:          color.append('r')      else:          color.append('b')    # plotting the bars  ax1.bar3d(x, y, z, dx, dy, dz, color = color)    # setting axes labels  ax1.set\_xlabel('x-axis')  ax1.set\_ylabel('y-axis')  ax1.set\_zlabel('z-axis')    plt.show() |