**Python Statistics in EDA**

**Learning Objectives**

* Transfer statistical concepts into practical skills and learn how to implement statistical concepts in Python
* Take a deep dive into statistical inference, hypothesis testing, and statistical modeling in Python
* Incorporate learning from data visualization in Python

**Work to Complete**

In this unit, you'll:

* Complete a case study about frequentist inference
* Implement a hypothesis test via a permutation test in the Integrating Apps case study
* Work on a statistical modeling case study
* Have the opportunity to get more practice via a second, optional case study
* Complete the EDA step of your second capstone project

In this unit, you're going to learn about how to apply Python to the sound statistical knowledge you acquired in the previous unit. 'Excellence in data science,' by the best definitions of the term, consists of the programmatic application of a solid grounding in statistics to solve problems with data. In the first subunit, you'll look at statistical inference in Python. As the name suggests, frequentist inference is the process of deriving conclusions about an underlying distribution via the observation of data. You're already familiar with this concept — you actually carried out the Frequentist Inference Seeing Theory exercises after reading Chapter 7 of Professor Spiegelhalter's book.

The second subunit focuses on data visualization in Python. Data visualization and statistics have an intimate relation: statistics summarize numbers, and as Professor Spiegelhalter taught you, these numbers almost always need to be visualized for them to communicate facts about the world around us.

Subunit three is dedicated to learning more about how to implement statistical hypothesis testing in Python. One of the most important skills for hypothesis testing is bootstrapping, which involves producing confidence intervals and the distribution of test statistics through resampling the observed data with a replacement (rather than assuming a probability model for the underlying random variable).

Finally, you'll look at some slightly more complex applications of statistics in Python, such as the statistical modeling methods of linear regression and classification.

### Statistical Inference in Python

In this subunit, you'll take a look at **implementing statical inference with Python**. The subunit will culminate in a frequentist inference case study. Frequentist inference is the process of deriving conclusions about an underlying distribution via the observation of data.

You've already done some frequentist inference exercises on Seeing Theory. This subunit will introduce you to the basics of Pythonic statistics with a DataCamp course, reinforce those skills via a training session from LinkedIn Learning, and then give you hands-on practice with a case study.

**A picture containing clipart

Description automatically generatedWhen you get to the LinkedIn Learning video, click "Open in new tab." Choose "continue without credentials'' if prompted, and it will take you to the LinkedIn Learning material.**

Interactive Course

# Statistical Thinking in Python (Part 1)

* 3 hours
* 18 Videos
* 61 Exercises
* 170,186 Participants
* 4,550 XP

### Course Description



After all of the hard work of acquiring data and getting them into a form you can work with, you ultimately want to make clear, succinct conclusions from them. This crucial last step of a data analysis pipeline hinges on the principles of statistical inference. In this course, you will start building the foundation you need to think statistically, speak the language of your data, and understand what your data is telling you. The foundations of statistical thinking took decades to build, but can be grasped much faster today with the help of computers. With the power of Python-based tools, you will rapidly get up-to-speed and begin thinking statistically by the end of this course.

Read More

1. 1

#### Graphical Exploratory Data Analysis

0%

Before diving into sophisticated statistical inference techniques, you should first explore your data by plotting them and computing simple summary statistics. This process, called exploratory data analysis, is a crucial first step in statistical analysis of data.

##### Introduction to Exploratory Data Analysis

50 xp

##### Tukey's comments on EDA

50 xp

##### Advantages of graphical EDA

50 xp

##### Plotting a histogram

50 xp

##### Plotting a histogram of iris data

100 xp

##### Axis labels!

100 xp

##### Adjusting the number of bins in a histogram

100 xp

##### Plot all of your data: Bee swarm plots

50 xp

##### Bee swarm plot

100 xp

##### Interpreting a bee swarm plot

50 xp

##### Plot all of your data: ECDFs

50 xp

##### Computing the ECDF

100 xp

##### Plotting the ECDF

100 xp

##### Comparison of ECDFs

100 xp

##### Onward toward the whole story!

50 xp

[Hide Chapter Details](https://www.datacamp.com/courses/statistical-thinking-in-python-part-1?embedded=true)

  2

#### Quantitative Exploratory Data Analysis

0%

In this chapter, you will compute useful summary statistics, which serve to concisely describe salient features of a dataset with a few numbers.

##### Introduction to summary statistics: The sample mean and median

50 xp

##### Means and medians

50 xp

##### Computing means

100 xp

##### Percentiles, outliers, and box plots

50 xp

##### Computing percentiles

100 xp

##### Comparing percentiles to ECDF

100 xp

##### Box-and-whisker plot

100 xp

##### Variance and standard deviation

50 xp

##### Computing the variance

100 xp

##### The standard deviation and the variance

100 xp

##### Covariance and the Pearson correlation coefficient

50 xp

##### Scatter plots

100 xp

##### Variance and covariance by looking

50 xp

##### Computing the covariance

100 xp

##### Computing the Pearson correlation coefficient

100 xp

[Hide Chapter Details](https://www.datacamp.com/courses/statistical-thinking-in-python-part-1?embedded=true)

  3

#### Thinking Probabilistically-- Discrete Variables

0%

Statistical inference rests upon probability. Because we can very rarely say anything meaningful with absolute certainty from data, we use probabilistic language to make quantitative statements about data. In this chapter, you will learn how to think probabilistically about discrete quantities: those that can only take certain values, like integers.

##### Probabilistic logic and statistical inference

50 xp

##### What is the goal of statistical inference?

50 xp

##### Why do we use the language of probability?

50 xp

##### Random number generators and hacker statistics

50 xp

##### Generating random numbers using the np.random module

100 xp

##### The np.random module and Bernoulli trials

100 xp

##### How many defaults might we expect?

100 xp

##### Will the bank fail?

100 xp

##### Probability distributions and stories: The Binomial distribution

50 xp

##### Sampling out of the Binomial distribution

100 xp

##### Plotting the Binomial PMF

100 xp

##### Poisson processes and the Poisson distribution

50 xp

##### Relationship between Binomial and Poisson distributions

100 xp

##### How many no-hitters in a season?

50 xp

##### Was 2015 anomalous?

100 xp

[Hide Chapter Details](https://www.datacamp.com/courses/statistical-thinking-in-python-part-1?embedded=true)

  4

#### Thinking Probabilistically-- Continuous Variables

0%

It’s time to move onto continuous variables, such as those that can take on any fractional value. Many of the principles are the same, but there are some subtleties. At the end of this final chapter, you will be speaking the probabilistic language you need to launch into the inference techniques covered in the sequel to this course.

##### Probability density functions

50 xp

##### Interpreting PDFs

50 xp

##### Interpreting CDFs

50 xp

##### Introduction to the Normal distribution

50 xp

##### The Normal PDF

100 xp

##### The Normal CDF

100 xp

##### The Normal distribution: Properties and warnings

50 xp

##### Gauss and the 10 Deutschmark banknote

50 xp

##### Are the Belmont Stakes results Normally distributed?

100 xp

##### What are the chances of a horse matching or beating Secretariat's record?

100 xp

##### The Exponential distribution

50 xp

##### Matching a story and a distribution

50 xp

##### Waiting for the next Secretariat

50 xp

##### If you have a story, you can simulate it!

100 xp

##### Distribution of no-hitters and cycles

100 xp

##### Final thoughts

# xploratory Data Analysis

**50 XP**

## 1. Introduction to Exploratory Data Analysis

Yogi Berra said, "You can observe a lot by watching." The same is true with data. If you can appropriately display your data, you can already start to draw conclusions from it. I'll go even further. Exploring your data is a crucial step in your analysis.

## 2. Exploratory data analysis

When I say exploring your data, I mean organizing and plotting your data, and maybe computing a few numerical summaries about them. This idea is known as exploratory data analysis, or EDA, and was developed by one of the greatest statisticians of all time, John Tukey. He wrote a book entitled Exploratory Data Analysis in 1977 where he laid out the principles.

## 3. John Tukey

In that book, he said, "Exploratory data analysis can never be the whole story, but nothing else can serve as the foundation stone." I wholeheartedly agree with this, so we will begin our study of statistical thinking with EDA. Let's consider an example.

## 4. 2008 US swing state election results

Here, we have a data set I acquired from data dot gov containing the election results of 2008 at the county level in each of the three major swing states of Pennsylvania, Ohio, and Florida. Those are the ones that largely decide recent elections in the US. This is how they look when I open the file with my text editor. They are a little prettier if we look at them with in a Pandas DataFrame,

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 5. 2008 US swing state election results

in this case only looking at the columns of immediate interest, the state, county, and share of the vote that went to Democrat Barack Obama. We could stare the these numbers, but I think you'll agree that it is pretty hopeless to gain any sort of understanding from doing this. Alternatively, we could charge in headlong and start defining and computing parameters and their confidence intervals, and do hypothesis tests. You will learn how to do all of these things in this course and its sequel. But a good field commander does not just charge into battle without first getting a feel for the terrain and sizing up the opposing army. So, like the field commander, we should explore the data first. In this chapter, we will discuss graphical exploratory data analysis. This involves taking data from tabular form, like we have here in the DataFrame,

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 6. 2008 US swing state election results

and representing it graphically. You are presenting the same information, but it is in a more human-interpretable form. For example, we take the Democratic share of the vote in the counties of all of the three swing states and plot them as a histogram. The height of each bar is the number of counties that had the given level of support for Obama. For example, the tallest bar is the number of counties that had between 40% and 50% of its votes cast for Obama. Right away, because there is more area in the histogram to the left of 50%, we can see that more counties voted for Obama's opponent, John McCain, than voted for Obama. Look at that. Just by making one plot, we could already draw a conclusion about the data, which would have been extraordinarily tedious by hand counting in the DataFrame.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 7. Let's practice!

Now let's review some of the basic ideas behind EDA with a couple exercises.

# Tukey's comments on EDA

Even though you probably have not read Tukey's book, I suspect you already have a good idea about his viewpoint from the video introducing you to exploratory data analysis. Which of the following quotes is **not** directly from Tukey?

##### Answer the question

**50XP**

#### Possible Answers

* 

Exploratory data analysis is detective work.

* 

There is no excuse for failing to plot and look.

* 

The greatest value of a picture is that it forces us to notice what we never expected to see.

* 

It is important to understand what you can do before you learn how to measure how well you seem to have done it.

* 

**Often times EDA is too time consuming, so it is better to jump right in and do your hypothesis tests.**

press5

You're right. That statement is pretty absurd. If you don't have time to do EDA, you really don't have time to do hypothesis tests. And you should always do EDA first.

# Advantages of graphical EDA

Which of the following is not true of graphical EDA?

#### Possible Answers

* 

It often involves converting tabular data into graphical form.

* 

If done well, graphical representations can allow for more rapid interpretation of data.

* 

**A nice looking plot is always the end goal of a statistical analysis.**

* 

There is no excuse for neglecting to do graphical EDA

**Correct! While a good, informative plot can sometimes be the end point of an analysis, it is more like a beginning: it helps guide you in the quantitative statistical analyses that come next.**

**Daily XP150**

# Plotting a histogram

**50 XP**

## 1. Plotting a histogram

We saw in the last video that a histogram can be a useful plot to generate when exploring a data set. Let's go over how we can create one.

## 2. 2008 US swing state election results

We are interested in the fraction of the vote that went to Barack Obama in each county. We can plot this as a histogram using

## 3. Generating a histogram

the matplotlib dot pyplot module's hist function. We pass it the dem\_share column of the DataFrame. We could have also passed a NumPy array with the same data, and it works just fine. In fact, for this course and its sequel, you can use DataFrames and NumPy arrays interchangeably. Note that plt dot hist returns three arrays that I am not interested in; I only want the plot. I therefore assign a dummy variable called "underscore" to them, which is common practice in Python. After creating the histogram, we label the axes.

## 4. Always label your axes

Always label your axes, for histograms or any other kind of plot. Otherwise no one can know what it is you are plotting.

## 5. 2008 US swing state election results

You probably didn't notice, but this plot looks slightly different than the first plot I showed. You can see it if you look at them side-by-side.

## 6. Histograms with different binning

They are different because they have different binning. In the plot at left, we have ten bins that were automatically generated by the default settings of plt dot hist, and I set up the bins on the right myself.

## 7. Setting the bins of a histogram

I specified where the edges of the bars of the histogram are, the bin edges, and use the bins keyword argument to pass that to plt dot hist.

## 8. Setting the bins of a histogram

You can also specify a number of bins, say 20, using the bins keyword argument, and Matplotlib will automatically generate 20 evenly spaced bins. Now, the plots we've made so far are stylized with Matplotlib's default settings. I prefer to use the default settings

## 9. Seaborn

of Seaborn, an excellent matplotlib-based statistical data visualization package written primarily by Michael Waskom.

## 10. Setting Seaborn styling

We import it as sns, as is traditionally done. Upon import, we can set the style to be Seaborn's default, using the sns dot set function.

## 11. A Seaborn-styled histogram

This results in nicely formatted plots. Beyond this stylistic functionality, Seaborn offers useful plotting functions that we will explore in the next video.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 12. Let's practice!

Before we do that, let's practice making histograms with some exercises.

**Daily XP200**

##### Exercise

##### Exercise

# Plotting a histogram of iris data

For the exercises in this section, you will use a classic data set collected by botanist Edward Anderson and made famous by Ronald Fisher, one of the most prolific statisticians in history. Anderson carefully measured the anatomical properties of samples of three different species of iris, Iris setosa, Iris versicolor, and Iris virginica. The full data set is [available as part of scikit-learn](http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_iris.html). Here, you will work with his measurements of petal length.

Plot a histogram of the petal lengths of his 50 samples of Iris versicolor using matplotlib/seaborn's default settings. Recall that to specify the default seaborn style, you can use sns.set(), where sns is the alias that seaborn is imported as.

The subset of the data set containing the Iris versicolor petal lengths in units of centimeters (cm) is stored in the NumPy array versicolor\_petal\_length.

In the video, Justin plotted the histograms by using the pandas library and indexing the DataFrame to extract the desired column. Here, however, you only need to use the provided NumPy array. Also, Justin assigned his plotting statements (except for plt.show()) to the dummy variable \_. This is to prevent unnecessary output from being displayed. It is not required for your solutions to these exercises, however it is good practice to use it. Alternatively, if you are working in an interactive environment such as a Jupyter notebook, you could use a ; after your plotting statements to achieve the same effect. Justin prefers using \_. Therefore, you will see it used in the solution code.

##### Instructions

**100 XP**

* Import matplotlib.pyplot and seaborn as their usual aliases (plt and sns).
* Use seaborn to set the plotting defaults.
* Plot a histogram of the Iris versicolor petal lengths using plt.hist() and the provided NumPy array versicolor\_petal\_length.
* Show the histogram using plt.show().
* # Import plotting modules
* # Set default Seaborn style
* # Plot histogram of versicolor petal lengths
* # Show histogram
* # Import plotting modules
* import matplotlib.pyplot as plt
* import seaborn as sns
* # Set default Seaborn style
* sns.set()
* # Plot histogram of versicolor petal lengths
* plt.hist(versicolor\_petal\_length, bins=20)
* # Show histogram
* plt.show()

# Axis labels!

In the last exercise, you made a nice histogram of petal lengths of Iris versicolor, but **you didn't label the axes!** That's ok; it's not your fault since we didn't ask you to. Now, add axis labels to the plot using plt.xlabel() and plt.ylabel(). Don't forget to add units and assign both statements to \_. The packages matplotlib.pyplot and seaborn are already imported with their standard aliases. This will be the case in what follows, unless specified otherwise.

##### Instructions

**100 XP**

* Label the axes. Don't forget that you should always include units in your axis labels. Your

-axis label is just 'count'. Your

* -axis label is 'petal length (cm)'. The units are essential!
* Display the plot constructed in the above steps using plt.show().

# Plot histogram of versicolor petal lengths

\_ = plt.hist(versicolor\_petal\_length)

# Label axes

# Show histogram

# Plot histogram of versicolor petal lengths

\_ = plt.hist(versicolor\_petal\_length)

# Label axes

\_ = plt.xlabel('petal length (cm)')

\_ = plt.ylabel('count')

# Show histogram

plt.show()

**Daily XP400**

##### Exercise

##### Exercise

# Adjusting the number of bins in a histogram

The histogram you just made had ten bins. This is the default of matplotlib. The "square root rule" is a commonly-used rule of thumb for choosing number of bins: choose the number of bins to be the square root of the number of samples. Plot the histogram of Iris versicolor petal lengths again, this time using the square root rule for the number of bins. You specify the number of bins using the bins keyword argument of plt.hist().

The plotting utilities are already imported and the seaborn defaults already set. The variable you defined in the last exercise, versicolor\_petal\_length, is already in your namespace.

##### Instructions

**100 XP**

* Import numpy as np. This gives access to the square root function, np.sqrt().
* Determine how many data points you have using len().
* Compute the number of bins using the square root rule.
* Convert the number of bins to an integer using the built in int() function.
* Generate the histogram and make sure to use the bins keyword argument.
* Hit submit to plot the figure and see the fruit of your labors!
* # Import numpy
* # Compute number of data points: n\_data
* # Number of bins is the square root of number of data points: n\_bins
* # Convert number of bins to integer: n\_bins
* # Plot the histogram
* # Label axes
* \_ = plt.xlabel('petal length (cm)')
* \_ = plt.ylabel('count')
* # Show histogram
* plt.show()
* # Import numpy
* import numpy as np
* # Compute number of data points: n\_data
* n\_data = len(versicolor\_petal\_length)
* # Number of bins is the square root of number of data points: n\_bins
* n\_bins = np.sqrt(n\_data)
* # Convert number of bins to integer: n\_bins
* n\_bins = int(n\_bins)
* # Plot the histogram
* plt.hist(versicolor\_petal\_length, bins=n\_bins)
* # Label axes
* \_ = plt.xlabel('petal length (cm)')
* \_ = plt.ylabel('count')
* # Show histogram
* plt.show()

# Compute number of data points: n\_data n\_data = len(versicolor\_petal\_length) # Number of bins is the square root of number of data points: n\_bins n\_bins = np.sqrt(n\_data) # Convert number of bins to integer: n\_bins n\_bins = int(n\_bins) # Plot the histogram plt.hist(versicolor\_petal\_length, bins=n\_bins) # Label axes \_ = plt.xlabel('petal length (cm)') \_ = plt.ylabel('count') # Show histogram plt.show()

**Daily XP500**

# Plot all of your data: Bee swarm plots

**50 XP**

## 1. Plot all of your data: Bee swarm plots

The histogram of county-level election data was informative.

## 2. 2008 US swing state election results

We learned that more counties voted for McCain than for Obama. Since our goal is to learn from data, this is great! However, a major drawback of using histograms

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. 2008 US swing state election results

is that the same data set can look different depending on how the bins are chosen. And choice of bins is in many ways arbitrary. This leads to

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 4. Binning bias

binning bias; you might interpret your plot differently for two different choices of bin number. An additional problem with histograms is that we are not plotting all of the data. We are sweeping the data into bins, and losing their actual values.

## 5. Bee swarm plot

To remedy these problems we can make a bee swarm plot, also called a swarm plot. This is best shown by example. Here is a beeswarm plot of the vote totals in the three swing states. Each point in the plot represents the share of the vote Obama got in a single county. The position along the y-axis is the quantitative information. The data are spread in x to make them visible, but their precise location along the x-axis is unimportant. Notably, we no longer have any binning bias and all data are displayed. This plot may be conveniently generated using Seaborn.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 6. Organization of the data frame

A requirement is that your data are in a well-organized Pandas DataFrame

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 7. Organization of the data frame

where each column is a feature and

## 8. Organization of the data frame

each row an observation. In this case, an observation is a county, and the features are state and the Democratic share of the vote.

## 9. Generating a bee swarm plot

To make the plot, you need to specify which column gives the values for the y-axis, in this case the share of the vote that went to the Democrat Barack Obama, and the values for the x-axis, in this case the state. And of course, you need to tell it which DataFrame contains the data.

## 10. 2008 US swing state election results

From this plot, too, we can clearly see that Obama got less than 50% of the vote in the majority of counties in each of the three swing states. This time it is more detailed than a histogram, but without too much added visual complexity.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 11. Let's practice!

Now it's your turn to make some bee swarm plots!

**Daily XP550**

##### Exercise

##### Exercise

# Bee swarm plot

Make a bee swarm plot of the iris petal lengths. Your x-axis should contain each of the three species, and the y-axis the petal lengths. A data frame containing the data is in your namespace as df.

For your reference, the code Justin used to create the bee swarm plot in the video is provided below:

\_ = sns.swarmplot(x='state', y='dem\_share', data=df\_swing)

\_ = plt.xlabel('state')

\_ = plt.ylabel('percent of vote for Obama')

plt.show()

In the IPython Shell, you can use sns.swarmplot? or help(sns.swarmplot) for more details on how to make bee swarm plots using seaborn.

##### Instructions

**100 XP**

* In the IPython Shell, inspect the DataFrame df using df.head(). This will let you identify which column names you need to pass as the x and y keyword arguments in your call to sns.swarmplot().
* Use sns.swarmplot() to make a bee swarm plot from the DataFrame containing the Fisher iris data set, df. The x-axis should contain each of the three species, and the y-axis should contain the petal lengths.
* Label the axes.
* Show your plot.
* # Create bee swarm plot with Seaborn's default settings
* # Label the axes
* # Show the plot
* In [1]:
* help(sns.swarmplot)
* Help on function swarmplot in module seaborn.categorical:
* swarmplot(x=None, y=None, hue=None, data=None, order=None, hue\_order=None, dodge=False, orient=None, color=None, palette=None, size=5, edgecolor='gray', linewidth=0, ax=None, \*\*kwargs)
* Draw a categorical scatterplot with non-overlapping points.
* This function is similar to :func:`stripplot`, but the points are adjusted
* (only along the categorical axis) so that they don't overlap. This gives a
* better representation of the distribution of values, but it does not scale
* well to large numbers of observations. This style of plot is sometimes
* called a "beeswarm".
* A swarm plot can be drawn on its own, but it is also a good complement
* to a box or violin plot in cases where you want to show all observations
* along with some representation of the underlying distribution.
* Arranging the points properly requires an accurate transformation between
* data and point coordinates. This means that non-default axis limits must
* be set \*before\* drawing the plot.

* Input data can be passed in a variety of formats, including:
* - Vectors of data represented as lists, numpy arrays, or pandas Series
* objects passed directly to the ``x``, ``y``, and/or ``hue`` parameters.
* - A "long-form" DataFrame, in which case the ``x``, ``y``, and ``hue``
* variables will determine how the data are plotted.
* - A "wide-form" DataFrame, such that each numeric column will be plotted.
* - An array or list of vectors.
* In most cases, it is possible to use numpy or Python objects, but pandas
* objects are preferable because the associated names will be used to
* annotate the axes. Additionally, you can use Categorical types for the
* grouping variables to control the order of plot elements.
* This function always treats one of the variables as categorical and
* draws data at ordinal positions (0, 1, ... n) on the relevant axis, even
* when the data has a numeric or date type.
* See the :ref:`tutorial <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.
* dodge : bool, optional
* When using ``hue`` nesting, setting this to ``True`` will separate
* the strips for different hue levels along the categorical axis.
* Otherwise, the points for each level will be plotted in one swarm.
* orient : "v" | "h", optional
* Orientation of the plot (vertical or horizontal). This is usually
* inferred 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 :func:`color\_palette`, or a
* dictionary mapping hue levels to matplotlib colors.
* size : float, optional
* Diameter of the markers, in points. (Although ``plt.scatter`` is used
* to draw the points, the ``size`` argument here takes a "normal"
* markersize and not size^2 like ``plt.scatter``.
* edgecolor : matplotlib color, "gray" is special-cased, optional
* Color of the lines around each point. If you pass ``"gray"``, the
* brightness is determined by the color palette used for the body
* of the points.
* linewidth : float, optional
* Width of the gray lines that frame the plot elements.
* ax : matplotlib Axes, optional
* Axes object to draw the plot onto, otherwise uses the current Axes.
* Returns
* -------
* ax : matplotlib Axes
* Returns the Axes object with the plot drawn onto it.
* See Also
* --------
* boxplot : A traditional box-and-whisker plot with a similar API.
* violinplot : A combination of boxplot and kernel density estimation.
* stripplot : A scatterplot where one variable is categorical. Can be used
* in conjunction with other plots to show each observation.
* catplot : Combine a categorical plot with a class:`FacetGrid`.
* Examples
* --------
* Draw a single horizontal swarm plot:
* .. plot::
* :context: close-figs
* >>> import seaborn as sns
* >>> sns.set(style="whitegrid")
* >>> tips = sns.load\_dataset("tips")
* >>> ax = sns.swarmplot(x=tips["total\_bill"])
* Group the swarms by a categorical variable:
* .. plot::
* :context: close-figs
* >>> ax = sns.swarmplot(x="day", y="total\_bill", data=tips)
* Draw horizontal swarms:
* .. plot::
* :context: close-figs
* >>> ax = sns.swarmplot(x="total\_bill", y="day", data=tips)
* Color the points using a second categorical variable:
* .. plot::
* :context: close-figs
* >>> ax = sns.swarmplot(x="day", y="total\_bill", hue="sex", data=tips)
* Split each level of the ``hue`` variable along the categorical axis:
* .. plot::
* :context: close-figs
* >>> ax = sns.swarmplot(x="day", y="total\_bill", hue="smoker",
* ... data=tips, palette="Set2", dodge=True)
* Control swarm order by passing an explicit order:
* .. plot::
* :context: close-figs
* >>> ax = sns.swarmplot(x="time", y="tip", data=tips,
* ... order=["Dinner", "Lunch"])
* Plot using larger points:
* .. plot::
* :context: close-figs
* >>> ax = sns.swarmplot(x="time", y="tip", data=tips, size=6)
* Draw swarms of observations on top of a box plot:
* .. plot::
* :context: close-figs
* >>> ax = sns.boxplot(x="tip", y="day", data=tips, whis=np.inf)
* >>> ax = sns.swarmplot(x="tip", y="day", data=tips, color=".2")
* Draw swarms of observations on top of a violin plot:
* .. plot::
* :context: close-figs
* >>> ax = sns.violinplot(x="day", y="total\_bill", data=tips, inner=None)
* >>> ax = sns.swarmplot(x="day", y="total\_bill", data=tips,
* ... color="white", edgecolor="gray")
* Use :func:`catplot` to combine a :func:`swarmplot` and a
* :class:`FacetGrid`. This allows grouping within additional categorical
* variables. Using :func:`catplot` is safer than using :class:`FacetGrid`
* directly, as it ensures synchronization of variable order across facets:
* .. plot::
* :context: close-figs
* >>> g = sns.catplot(x="sex", y="total\_bill",
* ... hue="smoker", col="time",
* ... data=tips, kind="swarm",
* ... height=4, aspect=.7);

2]:

df.head()

Out[2]:

sepal length (cm) sepal width (cm) petal length (cm) petal width (cm) 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

n [3]:

df

Out[3]:

sepal length (cm) sepal width (cm) petal length (cm) petal width (cm) 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

.. ... ... ... ... ...

145 6.7 3.0 5.2 2.3 virginica

146 6.3 2.5 5.0 1.9 virginica

147 6.5 3.0 5.2 2.0 virginica

148 6.2 3.4 5.4 2.3 virginica

149 5.9 3.0 5.1 1.8 virginica

[150 rows x 5 columns]

# Create bee swarm plot with Seaborn's default settings

\_ = sns.swarmplot(x='species', y='petal length (cm)', data=df)

# Label the axes

\_ = plt.xlabel('species')

\_ = plt.ylabel('petal length (cm)')

# Show the plot

plt.show()

# Create bee swarm plot with Seaborn's default settings

\_ = sns.swarmplot(x='species', y='petal length (cm)', data=df)

# Label the axes

\_ = plt.xlabel('species')

\_ = plt.ylabel('petal length (cm)')

# Show the plot

plt.show()

**Daily XP650**

##### Exercise

##### Exercise

# Interpreting a bee swarm plot

Which of the following conclusions could you draw from the bee swarm plot of iris petal lengths you generated in the previous exercise? For your convenience, the bee swarm plot is regenerated and shown to the right.

##### Instructions

**50 XP**

##### Possible Answers

* 

All I. versicolor petals are shorter than I. virginica petals.

* 

I. setosa petals have a broader range of lengths than the other two species.

* 

**I. virginica petals tend to be the longest, and I. setosa petals tend to be the shortest of the three species.**

* 

I. versicolor is a hybrid of I. virginica and I. setosa.

Correct! Notice that we said "tend to be." Some individual I. virginica flowers may be shorter than individual I. versicolor flowers. It is also possible that an individual I. setosa flower may have longer petals than in individual I. versicolor flower, though this is highly unlikely, and was not observed by Anderson.

**Daily XP750**

##### Exercise

##### Exercise

# Computing the ECDF

In this exercise, you will write a function that takes as input a 1D array of data and then returns the x and y values of the ECDF. You will use this function over and over again throughout this course and its sequel. ECDFs are among the most important plots in statistical analysis. You can write your own function, foo(x,y) according to the following skeleton:

def foo(a,b):

"""State what function does here"""

# Computation performed here

return x, y

The function foo() above takes two arguments a and b and returns two values x and y. The function header def foo(a,b): contains the function signature foo(a,b), which consists of the function name, along with its parameters. For more on writing your own functions, see DataCamp's course [Python Data Science Toolbox (Part 1) here](https://www.datacamp.com/courses/python-data-science-toolbox-part-1)!

##### Instructions

**100 XP**

* Define a function with the signature ecdf(data). Within the function definition,
  + Compute the number of data points, n, using the len() function.
  + The x

 -values are the sorted data. Use the np.sort() function to perform the sorting.

 The y

* + data of the ECDF go from 1/n to 1 in equally spaced increments. You can construct this using np.arange(). Remember, however, that the end value in np.arange() is not inclusive. Therefore, np.arange() will need to go from 1 to n+1. Be sure to divide this by n.
  + The function returns the values x and y.
* def ecdf(data):
* """Compute ECDF for a one-dimensional array of measurements."""
* # Number of data points: n
* \_\_\_\_ = \_\_\_\_(\_\_\_\_)
* # x-data for the ECDF: x
* \_\_\_\_ = \_\_\_\_(\_\_\_\_)
* # y-data for the ECDF: y
* \_\_\_\_ = \_\_\_\_(\_\_\_\_, \_\_\_\_) / n
* return x, y

def ecdf(data): """Compute ECDF for a one-dimensional array of measurements.""" # Number of data points: n n = len(n) # x-data for the ECDF: x x = np.sort(n) # y-data for the ECDF: y y = np.arange(1, n+1) / n return x, y

def ecdf(data):

    """Compute ECDF for a one-dimensional array of measurements."""

    # Number of data points: n

    n = len(data)

    # x-data for the ECDF: x

    x = np.sort(data)

    # y-data for the ECDF: y

    y = np.arange(1, n+1) / n

    return x, y

def ecdf(data): """Compute ECDF for a one-dimensional array of measurements.""" # Number of data points: n n = len(data) # x-data for the ECDF: x x = np.sort(data) # y-data for the ECDF: y y = np.arange(1, n+1) / n return x, y

# A Product Manager’s Guide to Machine Learning: ECDFs

## Empirical cumulative distribution functions (ECDFs)

A picture containing blue, underpants

Description automatically generated

Photo by [Mark Basarab](https://unsplash.com/@ignitedit?utm_source=medium&utm_medium=referral) on [Unsplash](https://unsplash.com?utm_source=medium&utm_medium=referral)

Here is an another tool to add to Exploratory Data Analysis (EDA) toolkit. The ECDF maps every data point in the dataset to a quantile. Here is the code that you can use to create Histogram and ECDF:

import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
df = pd.read\_csv('ecdf.csv', names=["sales"])  
df.head()

Table

Description automatically generated

# Percentiles

data = df.to\_numpy()percentiles = np.array([2.5,25,50,75,100])# Compute percentiles: ptiles\_sales  
ptiles\_sales=np.percentile(data,percentiles)  
print(ptiles\_sales)[1.395500e+02 2.691800e+04 4.545300e+04 8.829950e+04 1.590546e+06]

# Histogram

# Import numpy  
import numpy as np# Compute number of data points: n\_data  
n\_data = len(data)# Number of bins is the square root of number of data points: n\_bins  
n\_bins = np.sqrt(n\_data)# Convert number of bins to integer: n\_bins  
n\_bins = int(n\_bins)# Plot the histogram  
plt.hist(data, bins=n\_bins)# Label axes  
\_ = plt.xlabel('Sales')  
\_ = plt.ylabel('count')# Show histogram  
plt.show()

A picture containing histogram

Description automatically generated

# ECDF

ECDFs stand for the “empirical cumulative distribution function”, and they map every data point in the dataset to a quantile, which is a number between 0 and 1 that indicates the cumulative fraction of data points smaller than that data point itself.

It takes as input a 1D array of data and then returns the x and y

def ecdf(a,b):  
 """State what function does here"""  
 # Computation performed here  
 return x, y

Your ecdf() function returns two arrays so you will need to unpack them. So unpacking is x, y = ecdf(data)

* The x-values are the sorted data. Use the np.sort() function to perform the sorting.
* The y data of the ECDF go from 1/n to 1 in equally spaced increments. You can construct this using np.arange(). Since np.arange() is not inclusive, will need to go from 1 to n+1

## ECDF Function

def ecdf(data):  
 ””” Compute ECDF for a one-dimensional array of measurements.”””  
 # Number of data points: n  
 n = len(data)

# x-data for the ECDF: x  
 x = np.sort(data)

# y-data for the ECDF: y  
 y = np.arange(1, n+1) / n

return x, y

## Plot Sales using ECDF

# Compute ECDF for sales data: x\_sales, y\_sales  
x\_sales, y\_sales = ecdf(data)  
# Generate plot  
plt.plot(x\_sales, y\_sales, marker = '.', linestyle = 'none')  
# Label the axes  
plt.xlabel('sales')  
plt.ylabel('ECDF')  
# Overlay percentiles as red diamonds.  
plt.plot(ptiles\_sales, percentiles/100, marker='D', color='red',  
 linestyle='none')  
# Display the plot  
plt.show()

Chart, scatter chart

Description automatically generated

Red dots are the percentiles.

# Conclusion

ECDFs provide forest landscape view of percentiles, values, outliers of a feature.

# Plotting the ECDF

You will now use your ecdf() function to compute the ECDF for the petal lengths of Anderson's Iris versicolor flowers. You will then plot the ECDF. Recall that your ecdf() function returns two arrays so you will need to unpack them. An example of such unpacking is x, y = foo(data), for some function foo().

##### Instructions

**100 XP**

* Use ecdf() to compute the ECDF of versicolor\_petal\_length. Unpack the output into x\_vers and y\_vers.
* Plot the ECDF as dots. Remember to include marker = '.' and linestyle = 'none' in addition to x\_vers and y\_vers as arguments inside plt.plot().
* Label the axes. You can label the y-axis 'ECDF'.
* Show your plot.

# Compute ECDF for versicolor data: x\_vers, y\_vers

\_\_\_\_, \_\_\_\_ = \_\_\_\_(\_\_\_\_)

# Generate plot

# Label the axes

# Display the plot

# Compute ECDF for versicolor data: x\_vers, y\_vers

x\_vers, y\_vers = ecdf(versicolor\_petal\_length)

# Generate plot

\_ = plt.plot(x\_vers, y\_vers, marker= '.', linestyle='none')

# Label the axes

\_ = plt.xlabel('Versicolor data')

\_ = plt.ylabel('ECDF')

# Display the plot

plt.show()

# Compute ECDF for versicolor data: x\_vers, y\_vers

x\_vers, y\_vers = ecdf(versicolor\_petal\_length)

# Generate plot \_ = plt.plot(x\_vers, y\_vers, marker= '.', linestyle='none')

# Label the axes \_ = plt.xlabel('Versicolor data') \_ = plt.ylabel('ECDF') # Display the plot plt.show()

# Comparison of ECDFs

ECDFs also allow you to compare two or more distributions (though plots get cluttered if you have too many). Here, you will plot ECDFs for the petal lengths of all three iris species. You already wrote a function to generate ECDFs so you can put it to good use!

To overlay all three ECDFs on the same plot, you can use plt.plot() three times, once for each ECDF. Remember to include marker='.' and linestyle='none' as arguments inside plt.plot().

##### Instructions

* Compute ECDFs for each of the three species using your ecdf() function. The variables setosa\_petal\_length, versicolor\_petal\_length, and virginica\_petal\_length are all in your namespace. Unpack the ECDFs into x\_set, y\_set, x\_vers, y\_vers and x\_virg, y\_virg, respectively.
* Plot all three ECDFs on the same plot as dots. To do this, you will need three plt.plot() commands. Assign the result of each to \_.
* A legend and axis labels have been added for you, so hit submit to see all the ECDFs!
* # Compute ECDFs
* # Plot all ECDFs on the same plot
* # Annotate the plot
* plt.legend(('setosa', 'versicolor', 'virginica'), loc='lower right')
* \_ = plt.xlabel('petal length (cm)')
* \_ = plt.ylabel('ECDF')
* # Display the plot
* plt.show()

# Compute ECDFs

x\_set, y\_set = ecdf(setosa\_petal\_length)

x\_vers, y\_vers = ecdf(versicolor\_petal\_length)

x\_virg, y\_virg = ecdf(virginica\_petal\_length)

# Plot all ECDFs on the same plot

\_ = plt.plot(x\_set, y\_set, marker='.', linestyle='none')

\_ = plt.plot(x\_vers, y\_vers, marker='.', linestyle='none')

\_ = plt.plot(x\_virg, y\_virg, marker='.', linestyle='none')

# Annotate the plot

plt.legend(('setosa', 'versicolor', 'virginica'), loc='lower right')

\_ = plt.xlabel('petal length (cm)')

\_ = plt.ylabel('ECDF')

# Display the plot

plt.show()

# Compute ECDFs x\_set, y\_set = ecdf(setosa\_petal\_length) x\_vers, y\_vers = ecdf(versicolor\_petal\_length) x\_virg, y\_virg = ecdf(virginica\_petal\_length) # Plot all ECDFs on the same plot \_ = plt.plot(x\_set, y\_set, marker='.', linestyle='none') \_ = plt.plot(x\_vers, y\_vers, marker='.', linestyle='none') \_ = plt.plot(x\_virg, y\_virg, marker='.', linestyle='none') # Annotate the plot plt.legend(('setosa', 'versicolor', 'virginica'), loc='lower right') \_ = plt.xlabel('petal length (cm)') \_ = plt.ylabel('ECDF') # Display the plot plt.show()

The ECDFs expose clear differences among the species. Setosa is much shorter, also with less absolute variability in petal length than versicolor and virginica.

# Onward toward the whole story!

**50 XP**

## 1. Onward toward the whole story!

You now have some great graphical EDA tools.

## 2. Insert title here...

You can quickly generate and investigate a histogram. You can immediately get a feel for your data by plotting all of them, with bee swarm plots or ECDFs. In almost every data set we encounter in this course, and in its sequel, and also in real life, we start with graphical EDA. Remember what Tukey said,

## 3. “Exploratory data analysis can never be the whole story, but nothing else can serve as the foundation stone.” — John Tukey

"Exploratory data analysis can never be the whole story, but nothing else can serve as the foundation stone."

## 4. Coming up…

In the next chapter, you will build upon graphical EDA with quantitative EDA, which allows you to compute useful summary statistics. With your foundation stone in place, you will spend the last half of this course learning to think probabilistically. You will learn about probability distributions for both discrete and continuous variables, which provide the mathematical foundation for you to draw meaningful conclusions from your data. We will not get mired in mathematical details, but rather will unleash the power of the NumPy random module to use hacker statistics in order to simulate the probabilistic stories and distributions that we encounter. You will find that by writing a few lines of Python code, you can perform even putatively complicated statistical analyses. As you work through this course and its sequel, you will grow ever closer to being able to tell what Tukey calls "the whole story."

## 5. Let's practice!

Now, let's get to work!

**Daily XP1100**

# Introduction to summary statistics: The sample mean and median

**50 XP**

## 1. Introduction to summary statistics: The sample mean and median

We have seen that histograms, bee swarm plots, and ECDFs provide effective summaries of data. But we often would like to summarize data even more succinctly, say in one or two numbers. These numerical summaries are not by any stretch a substitute for the graphical methods we have been employing, but they do take up a lot less real estate.

## 2. 2008 US swing state election results

Let's go back to the election data from the swing states again. If we could summarize the percentage of the votes for Obama at the county level in Pennsylvania in one number, what would we choose? The first number that pops into my mind is

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. 2008 US swing state election results

the mean. The mean for a given state is just the average percentage of votes over the counties. If we add the means as horizontal lines to the bee swarm plot, we see that they are a reasonable summary of the data.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 4. Mean vote percentage

To compute the mean of a set of data, we use the np dot mean function, here used to compute the mean county-level vote for Obama in Pennsylvania. To put it precisely, the mean, written here as x-bar, is the sum of all the data, divided by the number n of data points. Now, the mean is a useful statistic and easy to calculate, but a major problem is that it is heavily influenced

## 5. Outliers

by outliers, or data points whose value is far greater or less than most of the rest of the data. Consider the county-level

## 6. 2008 Utah election results

votes for Utah in the 2008 election. There are five counties that have high vote share for Obama, one of which has almost 60%. Even though the majority of the counties in Utah had less than 25% voting for Obama,

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 7. 2008 Utah election results

these anomalous counties pull the mean higher up. So, when we compute the mean, we get about 28%. We might like a summary statistic that is immune to extreme data.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 8. The median

The median provides exactly that. The median is the middle value of a data set. It is defined by how it is calculated: sort the the data and choosing the datum in the middle. Because it is derived from the ranking of sorted data, and not on the values of the data, the median is immune to data that take on extreme values.

## 9. 2008 Utah election results

Here it is displayed on the bee swarm plot. It is not tugged up by the counties with large fraction of votes for Obama.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 10. Computing the median

The median is computed by simply calling the np dot median function.

## 11. Let's practice!

Now let's practice using these two powerful and ubiquitous summary statistics!

# Means and medians

Which one of the following statements is true about means and medians?

#### Possible Answers

* 

An outlier can significantly affect the value of both the mean and the median.

* 

**An outlier can significantly affect the value of the mean, but not the median.**

* 

Means and medians are in general both robust to single outliers.

* 

The mean and median are equal if there is an odd number of data points.

# Computing means

The mean of all measurements gives an indication of the typical magnitude of a measurement. It is computed using np.mean().

##### Instructions

**100 XP**

* Compute the mean petal length of Iris versicolor from Anderson's classic data set. The variable versicolor\_petal\_length is provided in your namespace. Assign the mean to mean\_length\_vers.
* Hit submit to print the result.
* # Compute the mean: mean\_length\_vers
* # Print the result with some nice formatting
* print('I. versicolor:', mean\_length\_vers, 'cm')

# Compute the mean: mean\_length\_vers

mean\_length\_vers = np.mean(versicolor\_petal\_length)

# Print the result with some nice formatting

print('I. versicolor:', mean\_length\_vers, 'cm')

# Compute the mean: mean\_length\_vers

mean\_length\_vers = np.mean(versicolor\_petal\_length)

# Print the result with some nice formatting

print('I. versicolor:', mean\_length\_vers, 'cm')

I. versicolor: 4.26 cm

**Daily XP1300**

# Percentiles, outliers, and box plots

**50 XP**

## 1. Percentiles, outliers, and box plots

The median is a special name for the 50th percentile;

## 2. Percentiles on an ECDF

that is 50% of

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. Percentiles on an ECDF

the data are less than the median. Similarly, the 25th percentile

## 4. Percentiles on an ECDF

is the value of the data point that is greater than 25% of the sorted data, and so on for any

## 5. Percentiles on an ECDF

other percentile we want. Percentiles are useful summary statistics, and can be computed

## 6. Computing percentiles

using np dot percentile. We just pass a list of the percentiles we want (percentiles, not fractions), and it returns the data that match those percentiles. We can do this for all of the swing states. Let's compute the 25th, 50th, and 75th percentiles. We now have three summary statistics. Now the whole point of summary statistics was to keep things concise, but we're starting to get a lot of numbers here. Dealing with this issue is where quantitative EDA meets graphical EDA.

## 7. 2008 US election box plot

Box plots were invented by John Tukey himself to display some of the salient features of a data set based on percentiles. Here, we see a box plot showing Obama's vote share from states east and west of the Mississippi River. The center of box is the median,

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 8. 2008 US election box plot

which we know is the 50th percentile of the data. The edges of the boxes

## 9. 2008 US election box plot

are the 25th and 75th percentile. The total height of the box contains the middle 50% of the data, and is called

## 10. 2008 US election box plot

the interquartile range, or IQR. The whiskers extend a distance

## 11. 2008 US election box plot

of 1-point-5 times the IQR, or to the extent of the data, whichever is more extreme. Finally, any points outside of the whiskers are plotted

## 12. 2008 US election box plot

as individual points, which we often demarcate as outliers. While there is no single definition for an outlier, being more than 2 IQRs away from the median is a common criterion. It is important to remember that an outlier is not necessarily an erroneous data point. You should not assume an outlier is erroneous unless you have some known reason to. Since there is zero evidence of any substantial voter fraud in the United States, these outliers are not erroneous. They are just data points with extreme values. When the number of data are very large and bee swarm plots are too cluttered, box plots are a great alternative. It makes sense, then, that constructing a box plot

## 13. Generating a box plot

using Seaborn is exactly the same as making a bee swarm plot; we just use sns dot boxplot. And of course we never forget to label the axes.

## 14. Let's practice!

All right, let's go have some fun computing percentiles and making box plots!

# Computing percentiles

In this exercise, you will compute the percentiles of petal length of Iris versicolor.

##### Instructions

**100 XP**

* Create percentiles, a NumPy array of percentiles you want to compute. These are the 2.5th, 25th, 50th, 75th, and 97.5th. You can do so by creating a list containing these ints/floats and convert the list to a NumPy array using np.array(). For example, np.array([30, 50]) would create an array consisting of the 30th and 50th percentiles.
* Use np.percentile() to compute the percentiles of the petal lengths from the Iris versicolor samples. The variable versicolor\_petal\_length is in your namespace.
* Print the percentiles.
* # Specify array of percentiles: percentiles
* # Compute percentiles: ptiles\_vers
* # Print the result
* # Specify array of percentiles: percentiles
* percentiles = np.array([2.5, 25, 50, 75,97.5])
* # Compute percentiles: ptiles\_vers
* ptiles\_vers = np.percentile(versicolor\_petal\_length, percentiles)
* # Print the result
* print(ptiles\_vers)
* [3.3 4. 4.35 4.6 4.9775]
* <script.py> output:
* [3.3 4. 4.35 4.6 4.9775]
* # Specify array of percentiles: percentiles
* percentiles = np.array([2.5, 25, 50, 75,97.5])
* # Compute percentiles: ptiles\_vers
* ptiles\_vers = np.percentile(versicolor\_petal\_length, percentiles)
* # Print the result
* print(ptiles\_vers)
* [3.3    4.     4.35   4.6    4.9775]
* <script.py> output:
* [3.3    4.     4.35   4.6    4.9775]

# Comparing percentiles to ECDF

To see how the percentiles relate to the ECDF, you will plot the percentiles of Iris versicolor petal lengths you calculated in the last exercise on the ECDF plot you generated in chapter 1. The percentile variables from the previous exercise are available in the workspace as ptiles\_vers and percentiles.

Note that to ensure the Y-axis of the ECDF plot remains between 0 and 1, you will need to rescale the percentiles array accordingly - in this case, dividing it by 100.

##### Instructions

**100 XP**

* Plot the percentiles as red diamonds on the ECDF. Pass the x and y co-ordinates - ptiles\_vers and percentiles/100 - as positional arguments and specify the marker='D', color='red' and linestyle='none' keyword arguments. The argument for the y-axis - percentiles/100 has been specified for you.
* Display the plot.

# Plot the ECDF

\_ = plt.plot(x\_vers, y\_vers, '.')

\_ = plt.xlabel('petal length (cm)')

\_ = plt.ylabel('ECDF')

# Overlay percentiles as red diamonds.

\_ = plt.plot(\_\_\_\_, percentiles/100, \_\_\_\_='D', \_\_\_\_='red',

         \_\_\_\_=\_\_\_\_)

# Show the plot

# Plot the ECDF

\_ = plt.plot(x\_vers, y\_vers, '.')

\_ = plt.xlabel('petal length (cm)')

\_ = plt.ylabel('ECDF')

# Overlay percentiles as red diamonds.

\_ = plt.plot(ptiles\_vers, percentiles/100, marker='D', color='red',

         linestyle='none')

# Show the plot

plt.show()

# Plot the ECDF

\_ = plt.plot(x\_vers, y\_vers, '.')

\_ = plt.xlabel('petal length (cm)')

\_ = plt.ylabel('ECDF')

# Overlay percentiles as red diamonds.

\_ = plt.plot(ptiles\_vers, percentiles/100, marker='D', color='red',

linestyle='none')

# Show the plot

plt.show()

# Box-and-whisker plot

Making a box plot for the petal lengths is unnecessary because the iris data set is not too large and the bee swarm plot works fine. However, it is always good to get some practice. Make a box plot of the iris petal lengths. You have a pandas DataFrame, df, which contains the petal length data, in your namespace. Inspect the data frame df in the IPython Shell using df.head() to make sure you know what the pertinent columns are.

For your reference, the code used to produce the box plot in the video is provided below:

\_ = sns.boxplot(x='east\_west', y='dem\_share', data=df\_all\_states)

\_ = plt.xlabel('region')

\_ = plt.ylabel('percent of vote for Obama')

In the IPython Shell, you can use sns.boxplot? or help(sns.boxplot) for more details on how to make box plots using seaborn.

##### Instructions

**100 XP**

* The set-up is exactly the same as for the bee swarm plot; you just call sns.boxplot() with the same keyword arguments as you would sns.swarmplot(). The x-axis is 'species' and y-axis is 'petal length (cm)'.
* Don't forget to label your axes!
* Display the figure using the normal call.
* # Create box plot with Seaborn's default settings
* # Label the axes
* # Show the plot

In [3]:

sns.boxplot?

Signature: sns.boxplot(x=None, y=None, hue=None, data=None, order=None, hue\_order=None, orient=None, color=None, palette=None, saturation=0.75, width=0.8, dodge=True, fliersize=5, linewidth=None, whis=1.5, notch=False, ax=None, \*\*kwargs)

Docstring:

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

A box plot (or box-and-whisker plot) shows the distribution of quantitative

data in a way that facilitates comparisons between variables or across

levels of a categorical variable. The box shows the quartiles of the

dataset while the whiskers extend to show the rest of the distribution,

except for points that are determined to be "outliers" using a method

that is a function of the inter-quartile range.

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

- Vectors of data represented as lists, numpy arrays, or pandas Series

objects passed directly to the ``x``, ``y``, and/or ``hue`` parameters.

- A "long-form" DataFrame, in which case the ``x``, ``y``, and ``hue``

variables will determine how the data are plotted.

- A "wide-form" DataFrame, such that each numeric column will be plotted.

- An array or list of vectors.

In most cases, it is possible to use numpy or Python objects, but pandas

objects are preferable because the associated names will be used to

annotate the axes. Additionally, you can use Categorical types for the

grouping variables to control the order of plot elements.

This function always treats one of the variables as categorical and

draws data at ordinal positions (0, 1, ... n) on the relevant axis, even

when the data has a numeric or date type.

See the :ref:`tutorial <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 :func:`color\_palette`, or a

dictionary mapping hue levels to matplotlib colors.

saturation : float, optional

Proportion of the original saturation to draw colors at. Large patches

often look better with slightly desaturated colors, but set this to

``1`` if you want the plot colors to perfectly match the input color

spec.

width : float, optional

Width of a full element when not using hue nesting, or width of all the

elements for one level of the major grouping variable.

dodge : bool, optional

When hue nesting is used, whether elements should be shifted along the

categorical axis.

fliersize : float, optional

Size of the markers used to indicate outlier observations.

linewidth : float, optional

Width of the gray lines that frame the plot elements.

whis : float, optional

Proportion of the IQR past the low and high quartiles to extend the

plot whiskers. Points outside this range will be identified as

outliers.

notch : boolean, optional

Whether to "notch" the box to indicate a confidence interval for the

median. There are several other parameters that can control how the

notches are drawn; see the ``plt.boxplot`` help for more information

on them.

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 ``plt.boxplot`` at draw

time.

Returns

-------

ax : matplotlib Axes

Returns the Axes object with the plot drawn onto it.

See Also

--------

violinplot : A combination of boxplot and kernel density estimation.

stripplot : A scatterplot where one variable is categorical. Can be used

in conjunction with other plots to show each observation.

swarmplot : A categorical scatterplot where the points do not overlap. Can

be used with other plots to show each observation.

Examples

--------

Draw a single horizontal boxplot:

.. plot::

:context: close-figs

>>> import seaborn as sns

>>> sns.set(style="whitegrid")

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

>>> ax = sns.boxplot(x=tips["total\_bill"])

Draw a vertical boxplot grouped by a categorical variable:

.. plot::

:context: close-figs

>>> ax = sns.boxplot(x="day", y="total\_bill", data=tips)

Draw a boxplot with nested grouping by two categorical variables:

.. plot::

:context: close-figs

>>> ax = sns.boxplot(x="day", y="total\_bill", hue="smoker",

... data=tips, palette="Set3")

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

.. plot::

:context: close-figs

>>> ax = sns.boxplot(x="day", y="total\_bill", hue="time",

... data=tips, linewidth=2.5)

Control box order by passing an explicit order:

.. plot::

:context: close-figs

>>> ax = sns.boxplot(x="time", y="tip", data=tips,

... order=["Dinner", "Lunch"])

Draw a boxplot for each numeric variable in a DataFrame:

.. plot::

:context: close-figs

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

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

Use ``hue`` without changing box position or width:

.. plot::

:context: close-figs

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

>>> ax = sns.boxplot(x="day", y="total\_bill", hue="weekend",

... data=tips, dodge=False)

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

.. plot::

:context: close-figs

>>> ax = sns.boxplot(x="day", y="total\_bill", data=tips)

>>> ax = sns.swarmplot(x="day", y="total\_bill", data=tips, color=".25")

Use :func:`catplot` to combine a :func:`pointplot` and a

:class:`FacetGrid`. This allows grouping within additional categorical

variables. Using :func:`catplot` is safer than using :class:`FacetGrid`

directly, as it ensures synchronization of variable order across facets:

.. plot::

:context: close-figs

>>> g = sns.catplot(x="sex", y="total\_bill",

... hue="smoker", col="time",

... data=tips, kind="box",

... height=4, aspect=.7);

File: /usr/local/lib/python3.6/dist-packages/seaborn/categorical.py

Type: function

# Create box plot with Seaborn's default settings

sns.boxplot(x ='species', y = 'petal length (cm)', data= df )

# Label the axes

\_ = plt.xlabel('species')

\_ = plt.ylabel('petal length (cm)')

# Show the plot

plt.show()

# Create box plot with Seaborn's default settings sns.boxplot(x ='species', y = 'petal length (cm)', data= df ) # Label the axes \_ = plt.xlabel('species') \_ = plt.ylabel('petal length (cm)') # Show the plot plt.show()

**Daily XP100**

# Variance and standard deviation

**50 XP**

## 1. Variance and standard deviation

Once again, let's look at the 2008

## 2. 2008 US swing state election results

swing state data on the county level and think about other summary statistics we can calculate. In this bee swarm plot, I also show the means of each state with a horizontal line. In looking at this plot, the mean seems to capture the magnitude of the data, but what about the variability, or the spread, of the data? Florida seems to have more county-to-county variability than Pennsylvania or Ohio.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. Variance

We can quantify this spread with the variance. The variance is the average of the squared distance from the mean. That definition was a mouthful. Let's parse that more carefully with a graphical example, looking specifically at Florida.

## 4. 2008 Florida election results

For each data point,

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 5. 2008 Florida election results

we square the distance from the mean, and then take the average of all of these values.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 6. Computing the variance

Calculation of the variance is implemented in the np dot var function. Now, because the calculation of the variance involves squared quantities, it does not have the same units of what we have measured, in this case the vote share for Obama. Therefore, we are interested in

## 7. Computing the standard deviation

the square root of the variance, which is called the standard deviation. This is calculated with the np dot std function, and the results are the same as taking the square root of the variance.

## 8. 2008 Florida election results

When we look at the swarm plot, it is clear that the standard deviation is a reasonable metric for the typical spread of the data.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 9. Let's practice!

Ok, let's practice computing some variances and standard deviations!

# Computing the variance

It is important to have some understanding of what commonly-used functions are doing under the hood. Though you may already know how to compute variances, this is a beginner course that does not assume so. In this exercise, we will explicitly compute the variance of the petal length of Iris veriscolor using the equations discussed in the videos. We will then use np.var() to compute it.

##### Instructions

**100 XP**

* Create an array called differences that is the difference between the petal lengths (versicolor\_petal\_length) and the mean petal length. The variable versicolor\_petal\_length is already in your namespace as a NumPy array so you can take advantage of NumPy's vectorized operations.
* Square each element in this array. For example, x\*\*2 squares each element in the array x. Store the result as diff\_sq.
* Compute the mean of the elements in diff\_sq using np.mean(). Store the result as variance\_explicit.
* Compute the variance of versicolor\_petal\_length using np.var(). Store the result as variance\_np.
* Print both variance\_explicit and variance\_np in one print call to make sure they are consistent.
* # Array of differences to mean: differences
* # Square the differences: diff\_sq
* # Compute the mean square difference: variance\_explicit
* # Compute the variance using NumPy: variance\_np
* # Print the results
* # Array of differences to mean: differences
* differences = versicolor\_petal\_length - np.mean(versicolor\_petal\_length)
* #print(differences)
* # Square the differences: diff\_sq
* diff\_sq = differences\*\*2
* #print(diff\_sq)
* # Compute the mean square difference: variance\_explicit
* variance\_explicit = np.mean(diff\_sq)
* #print(variance\_explicit)
* # Compute the variance using NumPy: variance\_np
* variance\_np = np.var(versicolor\_petal\_length)
* # Print the results
* print(variance\_explicit, variance\_np)

# Array of differences to mean: differences

differences = versicolor\_petal\_length - np.mean(versicolor\_petal\_length)

#print(differences)

# Square the differences: diff\_sq

diff\_sq = differences\*\*2

#print(diff\_sq)

# Compute the mean square difference: variance\_explicit

variance\_explicit = np.mean(diff\_sq)

#print(variance\_explicit)

# Compute the variance using NumPy: variance\_np

variance\_np = np.var(versicolor\_petal\_length)

# Print the results

print(variance\_explicit, variance\_np)

0.21640000000000004 0.21640000000000004

<script.py> output:

0.21640000000000004 0.21640000000000004

# The standard deviation and the variance

As mentioned in the video, the standard deviation is the square root of the variance. You will see this for yourself by computing the standard deviation using np.std() and comparing it to what you get by computing the variance with np.var() and then computing the square root.

##### Instructions

**100 XP**

* Compute the variance of the data in the versicolor\_petal\_length array using np.var() and store it in a variable called variance.
* Print the square root of this value.
* Print the standard deviation of the data in the versicolor\_petal\_length array using np.std().
* Compute the variance: variance
* # Print the square root of the variance
* # Print the standard deviation

# Compute the variance: variance

variance = np.var(versicolor\_petal\_length)

print(variance)

# Print the square root of the variance

print(np.sqrt(variance))

# Print the standard deviation

print(np.std(versicolor\_petal\_length))

# Compute the variance: variance

variance = np.var(versicolor\_petal\_length)

print(variance)

# Print the square root of the variance

print(np.sqrt(variance))

# Print the standard deviation

print(np.std(versicolor\_petal\_length))

0.21640000000000004

0.4651881339845203

0.4651881339845203

# Covariance and the Pearson correlation coefficient

**50 XP**

## 1. Covariance and the Pearson correlation coefficient

We have more data than just the vote share for Obama. We also know the total number of votes in each county. Let's look at how these two quantities vary together.

## 2. 2008 US swing state election results

We start by looking at a scatter plot of the county data for the three swing states, plotting the percent vote for Obama versus the total number of votes in each county. Immediately from the scatter plot, we see that

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 3. 2008 US swing state election results

the twelve most populous counties all voted for Obama, and that most of the counties

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 4. 2008 US swing state election results

with small populations voted for McCain.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 5. Generating a scatter plot

To generate a scatter plot, we plot the data as points by setting the marker and linestyle keyword arguments of plt dot plot. (And of course we label the axes!) So, we have exposed another graphical EDA technique: scatter plots! We would like to have a summary statistic to go along with the information we have just gleaned from the scatter plot. We want a number that summarizes how Obama's vote share varies with the total vote count.

## 6. Covariance

One such statistic is the covariance. To understand where it comes from,

## 7. Calculation of the covariance

let's annotate the scatter plot with the means of the two quantities we are interested in. Now let's look at

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 8. Calculation of the covariance

this data point, from Lucas County, Ohio. This data point

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 9. Calculation of the covariance

differs from the mean vote share for Obama, and

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 10. Calculation of the covariance

the mean total votes. We can compute these differences for each data point. The covariance is the mean of the product of these differences. If x and y both tend to be above, or both below their respective means together, as they are in this data set, then the covariance is positive. This means that they are positively correlated: when x is high, so is y; when the county is populous, it has more votes for Obama. Conversely, if x is high while y is low, the covariance is negative, and the data are negatively correlated, or anticorrelated, which is not the case for this data set. We can compute the covariance using built-in NumPy functions you will use in the exercises. However, if we want to have a more generally applicable measure of how two variables depend on each other, we want it to be dimensionless, that is to not have any units.

1. 1 Data retrieved from Data.gov (https://www.data.gov/)

## 11. Pearson correlation coefficient

So, we can divide the covariance by the standard deviations of the x and y variables. This is called the Pearson correlation coefficient, usually denoted by the Greek letter rho. It is a comparison of the variability in the data due to codependence (the covariance) to the variability inherent to each variable independently (their standard deviations). Conveniently, it is dimensionless and ranges from -1 (for complete anticorrelation) to 1 (for complete correlation).

## 12. Pearson correlation coefficient examples

A value of zero means that there is no correlation at all between the data, as shown in the plot on the upper left. Data with intermediate values are shown on the other plots. As you can see, the Pearson correlation coefficient is a good metric for correlation between two variables.

## 13. Let's practice!

Now that you know how what the Pearson correlation coefficient is and what it means, you can compute it in the exercises using Python. You will then have an added tool in your EDA summary statistics toolbox. Let's do it!

##### Exercise

# Scatter plots

When you made bee swarm plots, box plots, and ECDF plots in previous exercises, you compared the petal lengths of different species of iris. But what if you want to compare two properties of a single species? This is exactly what we will do in this exercise. We will make a **scatter plot** of the petal length and width measurements of Anderson's Iris versicolor flowers. If the flower scales (that is, it preserves its proportion as it grows), we would expect the length and width to be correlated.

For your reference, the code used to produce the scatter plot in the video is provided below:

\_ = plt.plot(total\_votes/1000, dem\_share, marker='.', linestyle='none')

\_ = plt.xlabel('total votes (thousands)')

\_ = plt.ylabel('percent of vote for Obama')

##### Instructions

**100 XP**

* Use plt.plot() with the appropriate keyword arguments to make a scatter plot of versicolor petal length (x-axis) versus petal width (y-axis). The variables versicolor\_petal\_length and versicolor\_petal\_width are already in your namespace. Do not forget to use the marker='.' and linestyle='none' keyword arguments.
* Label the axes.
* Display the plot.

##### Exercise

# Scatter plots

When you made bee swarm plots, box plots, and ECDF plots in previous exercises, you compared the petal lengths of different species of iris. But what if you want to compare two properties of a single species? This is exactly what we will do in this exercise. We will make a **scatter plot** of the petal length and width measurements of Anderson's Iris versicolor flowers. If the flower scales (that is, it preserves its proportion as it grows), we would expect the length and width to be correlated.

For your reference, the code used to produce the scatter plot in the video is provided below:

\_ = plt.plot(total\_votes/1000, dem\_share, marker='.', linestyle='none')

\_ = plt.xlabel('total votes (thousands)')

\_ = plt.ylabel('percent of vote for Obama')

##### Instructions

**100 XP**

* Use plt.plot() with the appropriate keyword arguments to make a scatter plot of versicolor petal length (x-axis) versus petal width (y-axis). The variables versicolor\_petal\_length and versicolor\_petal\_width are already in your namespace. Do not forget to use the marker='.' and linestyle='none' keyword arguments.
* Label the axes.
* Display the plot.

##### Exercise

# Scatter plots

When you made bee swarm plots, box plots, and ECDF plots in previous exercises, you compared the petal lengths of different species of iris. But what if you want to compare two properties of a single species? This is exactly what we will do in this exercise. We will make a **scatter plot** of the petal length and width measurements of Anderson's Iris versicolor flowers. If the flower scales (that is, it preserves its proportion as it grows), we would expect the length and width to be correlated.

For your reference, the code used to produce the scatter plot in the video is provided below:

\_ = plt.plot(total\_votes/1000, dem\_share, marker='.', linestyle='none')

\_ = plt.xlabel('total votes (thousands)')

\_ = plt.ylabel('percent of vote for Obama')

##### Instructions

**100 XP**

* Use plt.plot() with the appropriate keyword arguments to make a scatter plot of versicolor petal length (x-axis) versus petal width (y-axis). The variables versicolor\_petal\_length and versicolor\_petal\_width are already in your namespace. Do not forget to use the marker='.' and linestyle='none' keyword arguments.
* Label the axes.
* Display the plot.

##### Exercise

# Scatter plots

When you made bee swarm plots, box plots, and ECDF plots in previous exercises, you compared the petal lengths of different species of iris. But what if you want to compare two properties of a single species? This is exactly what we will do in this exercise. We will make a **scatter plot** of the petal length and width measurements of Anderson's Iris versicolor flowers. If the flower scales (that is, it preserves its proportion as it grows), we would expect the length and width to be correlated.

For your reference, the code used to produce the scatter plot in the video is provided below:

\_ = plt.plot(total\_votes/1000, dem\_share, marker='.', linestyle='none')

\_ = plt.xlabel('total votes (thousands)')

\_ = plt.ylabel('percent of vote for Obama')

##### Instructions

**100 XP**

* Use plt.plot() with the appropriate keyword arguments to make a scatter plot of versicolor petal length (x-axis) versus petal width (y-axis). The variables versicolor\_petal\_length and versicolor\_petal\_width are already in your namespace. Do not forget to use the marker='.' and linestyle='none' keyword arguments.
* Label the axes.
* Display the plot.
* # Make a scatter plot
* # Label the axes
* # Show the result
* # Make a scatter plot
* \_ = plt.plot(versicolor\_petal\_length, versicolor\_petal\_width, marker='.',linestyle='none')
* # Label the axes
* \_ = plt.xlabel('petal length')
* \_ = plt.ylabel('petal width')
* # Show the result
* plt.show()

# Make a scatter plot \_ = plt.plot(versicolor\_petal\_length, versicolor\_petal\_width, marker='.',linestyle='none') # Label the axes \_ = plt.xlabel('petal length') \_ = plt.ylabel('petal width') # Show the result plt.show()

Indeed, we see some correlation. Longer petals also tend to be wider.

**Daily XP500**

##### Exercise

##### Exercise

# Variance and covariance by looking

Consider four scatter plots of

-

data, appearing to the right. Which has, respectively,

* the highest variance in the variable
* ,
* the highest covariance,
* negative covariance?

##### Instructions

**50 XP**

##### Possible Answers

* 

a, c, b

* 

d, c, a

* 

**d, c, b**

* 

d, d, b

**Daily XP550**

##### Exercise

##### Exercise

# Computing the covariance

The covariance may be computed using the Numpy function np.cov(). For example, we have two sets of data x and y, np.cov(x, y) returns a 2D array where entries [0,1] and [1,0] are the covariances. Entry [0,0] is the variance of the data in x, and entry [1,1] is the variance of the data in y. This 2D output array is called the **covariance matrix**, since it organizes the self- and covariance.

To remind you how the I. versicolor petal length and width are related, we include the scatter plot you generated in a previous exercise.

##### Instructions

**100 XP**

* Use np.cov() to compute the covariance matrix for the petal length (versicolor\_petal\_length) and width (versicolor\_petal\_width) of I. versicolor.
* Print the covariance matrix.
* Extract the covariance from entry [0,1] of the covariance matrix. Note that by symmetry, entry [1,0] is the same as entry [0,1].
* Print the covariance.
* # Compute the covariance matrix: covariance\_matrix
* # Print covariance matrix
* # Extract covariance of length and width of petals: petal\_cov
* # Print the length/width covariance
* # Compute the covariance matrix: covariance\_matrix
* covariance\_matrix = np.cov(versicolor\_petal\_length,versicolor\_petal\_width)
* # Print covariance matrix
* print(covariance\_matrix)
* # Extract covariance of length and width of petals: petal\_cov
* petal\_cov = covariance\_matrix[0,1]
* # Print the length/width covariance
* print(petal\_cov)
* # Compute the covariance matrix: covariance\_matrix
* covariance\_matrix = np.cov(versicolor\_petal\_length,versicolor\_petal\_width)
* # Print covariance matrix
* print(covariance\_matrix)
* # Extract covariance of length and width of petals: petal\_cov
* petal\_cov = covariance\_matrix[0,1]
* # Print the length/width covariance
* print(petal\_cov)
* [[0.22081633 0.07310204]
* [0.07310204 0.03910612]]
* 0.07310204081632653
* <script.py> output:
* [[0.22081633 0.07310204]
* [0.07310204 0.03910612]]
* 0.07310204081632653

# Computing the Pearson correlation coefficient

As mentioned in the video, the Pearson correlation coefficient, also called the Pearson r, is often easier to interpret than the covariance. It is computed using the np.corrcoef() function. Like np.cov(), it takes two arrays as arguments and returns a 2D array. Entries [0,0] and [1,1] are necessarily equal to 1 (can you think about why?), and the value we are after is entry [0,1].

In this exercise, you will write a function, pearson\_r(x, y) that takes in two arrays and returns the Pearson correlation coefficient. You will then use this function to compute it for the petal lengths and widths of I. versicolor.

Again, we include the scatter plot you generated in a previous exercise to remind you how the petal width and length are related.

##### Instructions

**100 XP**

* Define a function with signature pearson\_r(x, y).
  + Use np.corrcoef() to compute the correlation matrix of x and y (pass them to np.corrcoef() in that order).
  + The function returns entry [0,1] of the correlation matrix.
* Compute the Pearson correlation between the data in the arrays versicolor\_petal\_length and versicolor\_petal\_width. Assign the result to r.
* Print the result.
* def \_\_\_\_(\_\_\_\_, \_\_\_\_):
* """Compute Pearson correlation coefficient between two arrays."""
* # Compute correlation matrix: corr\_mat
* # Return entry [0,1]
* return corr\_mat[0,1]
* # Compute Pearson correlation coefficient for I. versicolor: r
* # Print the result
* def pearson\_r(x, y):
* """Compute Pearson correlation coefficient between two arrays."""
* # Compute correlation matrix: corr\_mat
* corr\_mat = np.corrcoef(x,y)
* # Return entry [0,1]
* return corr\_mat[0,1]
* # Compute Pearson correlation coefficient for I. versicolor: r
* r = pearson\_r(versicolor\_petal\_width,versicolor\_petal\_length)
* # Print the result
* print(r)
* def pearson\_r(x, y):
* """Compute Pearson correlation coefficient between two arrays."""
* # Compute correlation matrix: corr\_mat
* corr\_mat = np.corrcoef(x,y)
* # Return entry [0,1]
* return corr\_mat[0,1]
* # Compute Pearson correlation coefficient for I. versicolor: r
* r = pearson\_r(versicolor\_petal\_width,versicolor\_petal\_length)
* # Print the result
* print(r)
* 0.7866680885228168
* <script.py> output:
* 0.7866680885228168

# Probabilistic logic and statistical inference

**50 XP**

## 1. Probabilistic logic and statistical inference

Imagine you measured the petal lengths of

## 2. 50 measurements of petal length

50 flowers of a certain species. Here is the ECDF of those measurements. From what you have just learned, you can compute

## 3. 50 measurements of petal length

the mean of those 50 measurements, and I'll annotate it on the ECDF with a vertical line. That is useful, but there are millions of these flowers on the planet. Can you tell me the mean petal length of all of the flowers of that species?

## 4. 50 measurements of petal length

If I measure another 50 flowers, I get a similar, but quantitatively different set of measurements. Can you tell me what value I would get for the mean petal length if I measured yet another 50 flowers? We just don't have the language to do that, without probability. Probabilistic reasoning allows us to describe uncertainty. Though you can't tell me exactly what the mean of the next 50 petal lengths you measure will be, you could say that it is more probable to be close to what you got in the first 50 measurements that it is to be much greater.

## 5. 50 measurements of petal length

We can go ahead and repeat the measurements

## 6. 50 measurements of petal length

over and over again.

## 7. Repeats of 50 measurements of petal length

We see from the vertical lines that we expect the mean to be somewhere between 4-point-5 and 5 cm. This is what probabilistic thinking is all about. Given a set of data, you describe probabilistically what you might expect if those data were acquired again and again and again. This is the heart of statistical inference. It is the process by which we go from measured data to probabilistic conclusions about what we might expect if we collected the same data again. Your data speak in the language of probability.

## 8. Let's practice!

Let's do a few exercises exploring these idea, and then we'll come back to learn how to start speaking this probabilistic language.

**Booking:  Thursday, April 20, 2023  1:30 PM - 2:15 PM  Time Zone: US/Central**

**Thanks, Charrie!**

**Your DSC Group Elevator Pitch and LinkedIn Call has been scheduled for Thursday, April 20, 2023, 1:30 PM (US/Central).**

**You’ll get an email confirmation shortly.**

# What is the goal of statistical inference?

Why do we do statistical inference

#### Possible Answers

* To draw probabilistic conclusions about what we might expect if we collected the same data again.
* To draw actionable conclusions from data.
* To draw more general conclusions from relatively few data or observations.
* **All of these.**
* **Correct! Statistical inference involves taking your data to probabilistic conclusions about what you would expect if you took even more data, and you can make decisions based on these conclusions.**

# Why do we use the language of probability?

Which of the following is not a reason why we use probabilistic language in statistical inference?

#### Possible Answers

* 

Probability provides a measure of uncertainty.

* 

**Probabilistic language is not very precise.**

* Data are almost never exactly the same when acquired again, and probability allows us to say how much we expect them to vary.

Correct. Probabilistic language is in fact very precise. It precisely describes uncertainty.

# Random number generators and hacker statistics

**50 XP**

## 1. Random number generators and hacker statistics

In practice, we are going to think probabilistically using hacker statistics. You had an introduction to hacker statistics in previous DataCamp courses, and in this course we will greatly extend your expertise.

## 2. Hacker statistics

The basic idea is that instead of literally repeating the data acquisition over and over again, we can simulate those repeated measurements using Python. For our first simulation, we will take a cue from our forebears. The concepts of probability originated from studies of games of chance

## 3. Blaise Pascal

by Pascal and others in the 17th century, so we will simulate

1. 1 Image: artist unknown

## 4. Coins

coin flips. Specifically, we will simulate the outcome of 4 successive coin flips. Our goal is to compute the probability that we will get four heads out of four flips.

1. 1 Image: Heritage Auction

## 5. The np.random module

Numpy's random module, a suite of functions based on pseudorandom number generation, will be your main engine for doing this. To simulate a coin flip, we will use the function np dot random dot random, which draws a number between zero and one such that all numbers in this interval are equally likely to occur.

## 6. The np.random module

If the number we draw is less than point-5, which has a 50% chance of happening, we say we got heads, and we get tails otherwise. This type of experiment, where the result is either True (heads) or False (tails) is referred to as

## 7. Bernoulli trial

a Bernoulli trial, and we will work with these more as we go through the course. The pseudorandom number generator works by starting with an integer, called a seed,

## 8. Random number seed

and then generates random numbers in succession. The same seed gives the same sequence of random numbers, hence the name, "psuedorandom number generation". So, if you want to have reproducible code, it is a good idea to seed the random number generator using the np dot random dot seed function.

## 9. Simulating 4 coin flips

Now, to do our coin flips, we import NumPy, seed the random number generator, and then draw four random numbers. Conveniently, we can specify how many random numbers we want with the size keyword argument. The first number we get is less than one half, so it is a heads, but the remaining three are tails. We can show that explicitly using the less than operation, which gives us an array with the Boolean value True for heads and False for tails. We can compute the number of heads by summing the array of Booleans because in numerical contexts, Python treats True at one and False as zero. We want to know the probability of getting four heads if we were to repeat the four flips over and over again. We can do this with a for loop.

## 10. Simulating 4 coin flips

We first initialize the count to zero. We then do 10,000 repeats of the four-flip trials. If a given trial had four heads, we increase the count. So, what is the probability of getting all four heads? It's the number of times we got all heads, divided by the total number of trials we did. The result is about point-06. Pascal and his friends did not have computers and worked out problems like these with pen and paper. While this particular problem is tractable, pen-and-paper statistics can get hard fast.

## 11. Hacker stats probabilities

With hacker statistics, you pretty much do this same procedure every time. Figure out how to simulate your data, simulate it many many times, and then compute the fraction of trials that had the outcome you're interested in.

## 12. Let's practice!

Now, let's use hacker statistics to simulate some more data!

**Daily XP950**

##### Exercise

##### Exercise

# Generating random numbers using the np.random module

We will be hammering the np.random module for the rest of this course and its sequel. Actually, you will probably call functions from this module more than any other while wearing your hacker statistician hat. Let's start by taking its simplest function, np.random.random() for a test spin. The function returns a random number between zero and one. Call np.random.random() a few times in the IPython Shell. You should see numbers jumping around between zero and one.

In this exercise, we'll generate lots of random numbers between zero and one, and then plot a histogram of the results. If the numbers are truly random, all bars in the histogram should be of (close to) equal height.

You may have noticed that, in the video, Justin generated 4 random numbers by passing the keyword argument size=4 to np.random.random(). Such an approach is more efficient than a for loop: in this exercise, however, you will write a for loop to experience hacker statistics as the practice of repeating an experiment over and over again.

##### Instructions

**100 XP**

* Seed the random number generator using the seed 42.
* Initialize an empty array, random\_numbers, of 100,000 entries to store the random numbers. Make sure you use np.empty(100000) to do this.
* Write a for loop to draw 100,000 random numbers using np.random.random(), storing them in the random\_numbers array. To do so, loop over range(100000).
* Plot a histogram of random\_numbers. It is not necessary to label the axes in this case because we are just checking the random number generator. Hit submit to show your plot.
* Seed the random number generator
* # Initialize random numbers: random\_numbers
* random\_numbers = \_\_\_\_
* # Generate random numbers by looping over range(100000)
* for i in \_\_\_\_:
* random\_numbers[i] = \_\_\_\_
* # Plot a histogram
* \_ = plt.hist(\_\_\_\_)
* # Show the plot
* plt.show()
* # Seed the random number generator
* np.random.seed(42)
* # Initialize random numbers: random\_numbers
* random\_numbers = np.empty(100000)
* # Generate random numbers by looping over range(100000)
* for i in range(100000):
* random\_numbers[i] = np.random.random()
* # Plot a histogram
* \_ = plt.hist(random\_numbers)
* # Show the plot
* plt.show()

# Seed the random number generator np.random.seed(42) # Initialize random numbers: random\_numbers random\_numbers = np.empty(100000) # Generate random numbers by looping over range(100000) for i in range(100000): random\_numbers[i] = np.random.random() # Plot a histogram \_ = plt.hist(random\_numbers) # Show the plot plt.show()

Good work! The histogram is almost exactly flat across the top, indicating that there is equal chance that a randomly-generated number is in any of the bins of the histogram.

# The np.random module and Bernoulli trials

You can think of a Bernoulli trial as a flip of a possibly biased coin. Specifically, each coin flip has a probability

of landing heads (success) and probability

of landing tails (failure). In this exercise, you will write a function to perform n Bernoulli trials, perform\_bernoulli\_trials(n, p), which returns the number of successes out of n Bernoulli trials, each of which has probability p of success. To perform each Bernoulli trial, use the np.random.random() function, which returns a random number between zero and one.

##### Instructions

**100 XP**

* Define a function with signature perform\_bernoulli\_trials(n, p).
  + Initialize to zero a variable n\_success the counter of Trues, which are Bernoulli trial successes.
  + Write a for loop where you perform a Bernoulli trial in each iteration and increment the number of success if the result is True. Perform n iterations by looping over range(n).
    - To perform a Bernoulli trial, choose a random number between zero and one using np.random.random(). If the number you chose is less than p, increment n\_success (use the += 1 operator to achieve this).
  + The function returns the number of successes n\_success.
* def perform\_bernoulli\_trials(n, p):
* """Perform n Bernoulli trials with success probability p
* and return number of successes."""
* # Initialize number of successes: n\_success
* n\_success = \_\_\_\_
* # Perform trials
* for i in \_\_\_\_:
* # Choose random number between zero and one: random\_number
* # If less than p, it's a success so add one to n\_success
* if \_\_\_\_:
* \_\_\_\_
* return n\_success
* def perform\_bernoulli\_trials(n, p):
* """Perform n Bernoulli trials with success probability p
* and return number of successes."""
* # Initialize number of successes: n\_success
* n\_success = 0
* # Perform trials
* for i in range(n):
* # Choose random number between zero and one: random\_number
* random\_number = np.random.random()
* # If less than p, it's a success so add one to n\_success
* if random\_number < p:
* n\_success += 1
* return n\_success

def perform\_bernoulli\_trials(n, p): """Perform n Bernoulli trials with success probability p and return number of successes.""" # Initialize number of successes: n\_success n\_success = 0 # Perform trials for i in range(n): # Choose random number between zero and one: random\_number random\_number = np.random.random() # If less than p, it's a success so add one to n\_success if random\_number < p: n\_success += 1 return n\_success

##### Exercise

# How many defaults might we expect?

Let's say a bank made 100 mortgage loans. It is possible that anywhere between 0 and 100 of the loans will be defaulted upon. You would like to know the probability of getting a given number of defaults, given that the probability of a default is p = 0.05. To investigate this, you will do a simulation. You will perform 100 Bernoulli trials using the perform\_bernoulli\_trials() function you wrote in the previous exercise and record how many defaults we get. Here, a success is a default. (Remember that the word "success" just means that the Bernoulli trial evaluates to True, i.e., did the loan recipient default?) You will do this for another 100 Bernoulli trials. And again and again until we have tried it 1000 times. Then, you will plot a histogram describing the probability of the number of defaults.

##### Instructions

* Seed the random number generator to 42.
* Initialize n\_defaults, an empty array, using np.empty(). It should contain 1000 entries, since we are doing 1000 simulations.
* Write a for loop with 1000 iterations to compute the number of defaults per 100 loans using the perform\_bernoulli\_trials() function. It accepts two arguments: the number of trials n - in this case 100 - and the probability of success p - in this case the probability of a default, which is 0.05. On each iteration of the loop store the result in an entry of n\_defaults.
* Plot a histogram of n\_defaults. Include the normed=True keyword argument so that the height of the bars of the histogram indicate the probability.
* Show your plot.
* # Seed random number generator
* # Initialize the number of defaults: n\_defaults
* # Compute the number of defaults
* for i in \_\_\_\_:
* n\_defaults[i] = \_\_\_\_
* # Plot the histogram with default number of bins; label your axes
* \_ = plt.hist(\_\_\_\_, \_\_\_\_)
* \_ = plt.xlabel('number of defaults out of 100 loans')
* \_ = plt.ylabel('probability')
* # Show the plot
* # Seed random number generator
* np.random.seed(42)
* # Initialize the number of defaults: n\_defaults
* n\_defaults = np.empty(1000)
* # Compute the number of defaults
* for i in range(1000):
* n\_defaults[i] = perform\_bernoulli\_trials(100, 0.05)
* # Plot the histogram with default number of bins; label your axes
* \_ = plt.hist(n\_defaults, normed=True)
* \_ = plt.xlabel('number of defaults out of 100 loans')
* \_ = plt.ylabel('probability')
* # Show the plot
* plt.show()

# Seed random number generator np.random.seed(42) # Initialize the number of defaults: n\_defaults n\_defaults = np.empty(1000) # Compute the number of defaults for i in range(1000): n\_defaults[i] = perform\_bernoulli\_trials(100, 0.05) # Plot the histogram with default number of bins; label your axes \_ = plt.hist(n\_defaults, normed=True) \_ = plt.xlabel('number of defaults out of 100 loans') \_ = plt.ylabel('probability') # Show the plot plt.show()

Nice work! This is actually not an optimal way to plot a histogram when the results are known to be integers. We will revisit this in forthcoming exercises.

# Will the bank fail?

Plot the number of defaults you got from the previous exercise, in your namespace as n\_defaults, as a CDF. The ecdf() function you wrote in the first chapter is available.

If interest rates are such that the bank will lose money if 10 or more of its loans are defaulted upon, what is the probability that the bank will lose money?

##### Instructions

**100 XP**

* Compute the x and y values for the ECDF of n\_defaults.
* Plot the ECDF, making sure to label the axes. Remember to include marker = '.' and linestyle = 'none' in addition to x and y in your call plt.plot().
* Show the plot.
* Compute the total number of entries in your n\_defaults array that were greater than or equal to 10. To do so, compute a boolean array that tells you whether a given entry of n\_defaults is >= 10. Then sum all the entries in this array using np.sum(). For example, np.sum(n\_defaults <= 5) would compute the number of defaults with 5 or fewer defaults.
* The probability that the bank loses money is the fraction of n\_defaults that are greater than or equal to 10. Print this result by hitting submit!
* # Compute ECDF: x, y
* # Plot the ECDF with labeled axes
* # Show the plot
* # Compute the number of 100-loan simulations with 10 or more defaults: n\_lose\_money
* # Compute and print probability of losing money
* print('Probability of losing money =', n\_lose\_money / len(n\_defaults))
* # Compute ECDF: x, y
* x, y = ecdf(n\_defaults)
* # Plot the ECDF with labeled axes
* \_ = plt.plot(x,y, marker='.', linestyle='none')
* \_ = plt.xlabel('number of defaults')
* \_ = plt.ylabel('ECDF')
* # Show the plot
* plt.show()
* # Compute the number of 100-loan simulations with 10 or more defaults: n\_lose\_money
* n\_lose\_money = np.sum(n\_defaults >= 10)
* # Compute and print probability of losing money
* print('Probability of losing money =', n\_lose\_money / len(n\_defaults))
* # Plot the ECDF with labeled axes
* \_ = plt.plot(x,y, marker='.', linestyle='none')
* \_ = plt.xlabel('number of defaults')
* \_ = plt.ylabel('ECDF')
* # Show the plot
* plt.show()
* # Compute the number of 100-loan simulations with 10 or more defaults: n\_lose\_money
* n\_lose\_money = np.sum(n\_defaults >= 10)
* # Compute and print probability of losing money
* print('Probability of losing money =', n\_lose\_money / len(n\_defaults))
* Probability of losing money = 0.022
* <script.py> output:
* Probability of losing money = 0.022

As we might expect, we most likely get 5/100 defaults. But we still have about a 2% chance of getting 10 or more defaults out of 100 loans.

**Daily XP1350**

# Probability distributions and stories: The Binomial distribution

**50 XP**

## 1. Probability distributions and stories: The Binomial distribution

In the last video, we simulated a story about a person flipping a coin. We did this to get the probability for each possible outcome of the story. That set of probabilities is called

## 2. Probability mass function (PMF)

a probability mass function (PMF). A PMF is defined as the set of probabilities of discrete outcomes. To understand how this works, consider a simpler story,

## 3. Discrete Uniform PMF

a person rolling a die once. The outcomes are discrete because only certain values may be attained; you cannot roll a 3-point-7 with a die. Each result has the same, or uniform probability, 1/6. For this reason, the PMF associated with this story is called the Discrete Uniform PMF. Now the PMF is a property of a discrete probability distribution.

## 4. Probability distribution

A distribution is just a mathematical description of outcomes. We can match a story to a mathematical description of probabilities, as we have just seen with the Discrete Uniform distribution.

## 5. Discrete Uniform distribution: the story

The story we simulated in the last video corresponds to the Binomial distribution. Its story is as follows:

## 6. Binomial distribution: the story

the number r of successes in n Bernoulli trials with probability p of success is Binomially distributed. The number of heads in four coin flips matches this story, since a coin flip is a Bernoulli trial with p = point-5.

## 7. Sampling from the Binomial distribution

We call the function np dot random dot binomial with two arguments, the number of Bernoulli trials (coin flips) and the probability of success (heads). We get 2 heads out of four. We want repeat the four-flip experiment over and over again. Again, we can specify the size keyword argument, which tells the function how many random numbers to sample out of the Binomial distribution.

## 8. The Binomial PMF

To be able to plot the Binomial PMF, we'll draw 10,000 samples from a Binomial distribution where we do 60 Bernoulli trials with a probability of success of point-1. If we do this over and over and tally the results, we can plot the PMF. As expected the most likely number of successes is 6 out of 60, but we may expect to get as many as 11 or as few as 1. Unfortunately, while this is a proper way to display a PMF, it is not the easiest to implement with Matplotlib. You'll plot a PMF as a histogram in the exercises.

## 9. The Binomial CDF

The CDF is just as informative and easier to plot just as we have done before,

## 10. The Binomial CDF

and here it is. Now that you know the story of the binomial distribution, have a feel for how it looks, and know how to sample out of it,

## 11. Let's practice!

let's do some hacker statistics with it!

# Sampling out of the Binomial distribution

Compute the probability mass function for the number of defaults we would expect for 100 loans as in the last section, but instead of simulating all of the Bernoulli trials, perform the sampling using np.random.binomial(). This is identical to the calculation you did in the last set of exercises using your custom-written perform\_bernoulli\_trials() function, but far more computationally efficient. Given this extra efficiency, we will take 10,000 samples instead of 1000. After taking the samples, plot the CDF as last time. This CDF that you are plotting is that of the Binomial distribution.

Note: For this exercise and all going forward, the random number generator is pre-seeded for you (with np.random.seed(42)) to save you typing that each time.

##### Instructions

**100 XP**

* Draw samples out of the Binomial distribution using np.random.binomial(). You should use parameters n = 100 and p = 0.05, and set the size keyword argument to 10000.
* Compute the CDF using your previously-written ecdf() function.
* Plot the CDF with axis labels. The x-axis here is the number of defaults out of 100 loans, while the y-axis is the CDF.
* Show the plot.
* # Take 10,000 samples out of the binomial distribution: n\_defaults
* # Compute CDF: x, y
* # Plot the CDF with axis labels
* # Show the plot
* # Take 10,000 samples out of the binomial distribution: n\_defaults
* n\_defaults = np.random.binomial(100, 0.05, size=10000)
* # Compute CDF: x, y
* x,y = ecdf(n\_defaults)
* # Plot the CDF with axis labels
* \_ = plt.plot(x, y, marker='.', linestyle='none')
* \_ = plt.xlabel('number of defaults')
* \_ = plt.ylabel('CDF')
* # Show the plot
* plt.show()

# Take 10,000 samples out of the binomial distribution: n\_defaults n\_defaults = np.random.binomial(100, 0.05, size=10000) # Compute CDF: x, y x,y = ecdf(n\_defaults) # Plot the CDF with axis labels \_ = plt.plot(x, y, marker='.', linestyle='none') \_ = plt.xlabel('number of defaults') \_ = plt.ylabel('CDF') # Show the plot plt.show()

Great work! If you know the story, using built-in algorithms to directly sample out of the distribution is much faster.

# Plotting the Binomial PMF

As mentioned in the video, plotting a nice looking PMF requires a bit of matplotlib trickery that we will not go into here. Instead, we will plot the PMF of the Binomial distribution as a histogram with skills you have already learned. The trick is setting up the edges of the bins to pass to plt.hist() via the bins keyword argument. We want the bins centered on the integers. So, the edges of the bins should be -0.5, 0.5, 1.5, 2.5, ... up to max(n\_defaults) + 1.5. You can generate an array like this using np.arange() and then subtracting 0.5 from the array.

You have already sampled out of the Binomial distribution during your exercises on loan defaults, and the resulting samples are in the NumPy array n\_defaults.

##### Instructions

**100 XP**

* Using np.arange(), compute the bin edges such that the bins are centered on the integers. Store the resulting array in the variable bins.
* Use plt.hist() to plot the histogram of n\_defaults with the normed=True and bins=bins keyword arguments.
* Show the plot.
* # Compute bin edges: bins
* bins = np.arange(\_\_\_\_, \_\_\_\_ + \_\_\_\_) - 0.5
* # Generate histogram
* # Label axes
* # Show the plot
* # Compute bin edges: bins
* bins = np.arange(0, max(n\_defaults) + 1.5) - 0.5
* # Generate histogram
* plt.hist(n\_defaults, normed=True, bins=bins)
* # Label axes
* \_ = plt.xlabel('number of defaults')
* \_ = plt.ylabel('PMF')
* # Show the plot
* plt.show()

# Compute bin edges: bins bins = np.arange(0, max(n\_defaults) + 1.5) - 0.5 # Generate histogram plt.hist(n\_defaults, normed=True, bins=bins) # Label axes \_ = plt.xlabel('number of defaults') \_ = plt.ylabel('PMF') # Show the plot plt.show()

**Daily XP1600**

# Poisson processes and the Poisson distribution

**50 XP**

## 1. Poisson processes and the Poisson distribution

In his great book on information theory, statistical inference, and machine learning, David MacKay described a town called Poissonville where the buses have a very erratic schedule. If you stand at a bus stop in Poissonville, the amount of time you have to wait for a bus is completely independent of when the previous bus arrived. In other words, you would watch a bus drive off and another one will arrive almost instantly, or you could be waiting for hours. Arrival of buses in Poissonville is what we call

## 2. Poisson process

a Poisson process. The timing of the next event is completely independent of when the previous event happened. Many real-life processes behave in this way.

## 3. Examples of Poisson processes

For example, natural births in a given hospital are a Poisson process. There is a well-defined average number of natural births per year, and the timing of one birth is independent of the timing of the previous one. Another example is hits on a website. The timing of the next hit is independent of the timing of the last. There are countless other examples. Any process that matches the buses in Poissonville story is a Poisson process. The number of arrivals of a Poisson process in a given amount of time is Poisson distributed.

## 4. Poisson distribution

The Poisson distribution has one parameter, the average number of arrivals in a given length of time. So, to match the story, we could consider the number of hits on a website in an hour with an average of six hits per hour. This is Poisson distributed.

## 5. Poisson PMF

Let's take a look at the PMF for this example. For a given hour, we are most likely to get 6 hits, which is the average, but we may get more than ten, or possibly none. You might notice that this looks an awful lot like the Binomial PMF we looked at in the last video. This is because the Poisson distribution

## 6. Poisson Distribution

is a limit of the Binomial distribution for low probability of success and large number of trials, ie for rare events. You'll explore this relationship further in the following interactive exercises.

## 7. The Poisson CDF

To sample from the Poisson distribution, we use np dot random dot poisson. It also has the size keyword argument to allow multiple samples. Let's use this function to generate the Poisson CDF. After that it is the usual procedure of computing the ECDF, plotting it, and labeling axes.

## 8. The Poisson CDF

The result, predictably, looks like the Binomial CDF we saw in the last video.

## 9. Let's practice!

Now let's get some experience with the Poisson distribution in the exercises!

# Relationship between Binomial and Poisson distributions

You just heard that the Poisson distribution is a limit of the Binomial distribution for rare events. This makes sense if you think about the stories. Say we do a Bernoulli trial every minute for an hour, each with a success probability of 0.1. We would do 60 trials, and the number of successes is Binomially distributed, and we would expect to get about 6 successes. This is just like the Poisson story we discussed in the video, where we get on average 6 hits on a website per hour. So, the Poisson distribution with arrival rate equal to

approximates a Binomial distribution for Bernoulli trials with probability of success (with large and

small). Importantly, the Poisson distribution is often simpler to work with because it has only one parameter instead of two for the Binomial distribution.

Let's explore these two distributions computationally. You will compute the mean and standard deviation of samples from a Poisson distribution with an arrival rate of 10. Then, you will compute the mean and standard deviation of samples from a Binomial distribution with parameters

and such that

.

##### Instructions

**100 XP**

* Using the np.random.poisson() function, draw 10000 samples from a Poisson distribution with a mean of 10.
* Make a list of the n and p values to consider for the Binomial distribution. Choose n = [20, 100, 1000] and p = [0.5, 0.1, 0.01] so that
* is always 10.
* Using np.random.binomial() inside the provided for loop, draw 10000 samples from a Binomial distribution with each n, p pair and print the mean and standard deviation of the samples. There are 3 n, p pairs: 20, 0.5, 100, 0.1, and 1000, 0.01. These can be accessed inside the loop as n[i], p[i].
* # Draw 10,000 samples out of Poisson distribution: samples\_poisson
* # Print the mean and standard deviation
* print('Poisson:     ', np.mean(samples\_poisson),
* np.std(samples\_poisson))
* # Specify values of n and p to consider for Binomial: n, p
* # Draw 10,000 samples for each n,p pair: samples\_binomial
* for i in range(3):
* samples\_binomial = \_\_\_\_
* # Print results
* print('n =', n[i], 'Binom:', np.mean(samples\_binomial),
* np.std(samples\_binomial))
* # Draw 10,000 samples out of Poisson distribution: samples\_poisson
* samples\_poisson = np.random.poisson(10, size=10000)
* # Print the mean and standard deviation
* print('Poisson:     ', np.mean(samples\_poisson),
* np.std(samples\_poisson))
* # Specify values of n and p to consider for Binomial: n, p
* n = [20, 100, 1000]
* p = [0.5, 0.1, 0.01]
* # Draw 10,000 samples for each n,p pair: samples\_binomial
* for i in range(3):
* samples\_binomial = np.random.binomial(n[i], p[i], size=10000)
* # Print results
* print('n =', n[i], 'Binom:', np.mean(samples\_binomial),
* np.std(samples\_binomial))
* # Draw 10,000 samples out of Poisson distribution: samples\_poisson
* samples\_poisson = np.random.poisson(10, size=10000)
* # Print the mean and standard deviation
* print('Poisson: ', np.mean(samples\_poisson),
* np.std(samples\_poisson))
* # Specify values of n and p to consider for Binomial: n, p
* n = [20, 100, 1000]
* p = [0.5, 0.1, 0.01]
* # Draw 10,000 samples for each n,p pair: samples\_binomial
* for i in range(3):
* samples\_binomial = np.random.binomial(n[i], p[i], size=10000)
* # Print results
* print('n =', n[i], 'Binom:', np.mean(samples\_binomial),
* np.std(samples\_binomial))
* Poisson: 10.0303 3.1562290648810647
* n = 20 Binom: 10.0021 2.249554531457284
* n = 100 Binom: 9.9586 2.9906664875910187
* n = 1000 Binom: 9.9952 3.1254402825841994
* <script.py> output:
* Poisson: 10.0186 3.144813832327758
* n = 20 Binom: 9.9637 2.2163443572694206
* n = 100 Binom: 9.9947 3.0135812433050484
* n = 1000 Binom: 9.9985 3.139378561116833

The means are all about the same, which can be shown to be true by doing some pen-and-paper work. The standard deviation of the Binomial distribution gets closer and closer to that of the Poisson distribution as the probability p gets lower and lower.

# How many no-hitters in a season?

In baseball, a no-hitter is a game in which a pitcher does not allow the other team to get a hit. This is a rare event, and since the beginning of the so-called modern era of baseball (starting in 1901), there have only been 251 of them through the 2015 season in over 200,000 games. The ECDF of the number of no-hitters in a season is shown to the right. Which probability distribution would be appropriate to describe the number of no-hitters we would expect in a given season?

Note: The no-hitter data set was scraped and calculated from the data sets available at [retrosheet.org](http://www.retrosheet.org) ([license](http://www.retrosheet.org/notice.txt)).

##### Possible Answers

* 

Discrete uniform

* 

Binomial

* 

Poisson

* 

**Both Binomial and Poisson, though Poisson is easier to model and compute.**

* 

Both Binomial and Poisson, though Binomial is easier to model and compute.

Correct! When we have rare events (low p, high n), the Binomial distribution is Poisson. This has a single parameter, the mean number of successes per time interval, in our case the mean number of no-hitters per season.

**Daily XP200**

##### Exercise

##### Exercise

# Was 2015 anomalous?

1990 and 2015 featured the most no-hitters of any season of baseball (there were seven). Given that there are on average 251/115 no-hitters per season, what is the probability of having seven or more in a season?

##### Instructions

**100 XP**

* Draw 10000 samples from a Poisson distribution with a mean of 251/115 and assign to n\_nohitters.
* Determine how many of your samples had a result greater than or equal to 7 and assign to n\_large.
* Compute the probability, p\_large, of having 7 or more no-hitters by dividing n\_large by the total number of samples (10000).
* Hit submit to print the probability that you calculated.
* # Draw 10,000 samples out of Poisson distribution: n\_nohitters
* # Compute number of samples that are seven or greater: n\_large
* n\_large = np.sum(\_\_\_\_)
* # Compute probability of getting seven or more: p\_large
* # Print the result
* print('Probability of seven or more no-hitters:', p\_large)
* # Draw 10,000 samples out of Poisson distribution: n\_nohitters
* n\_nohitters = np.random.poisson(251/115, size=10000)
* # Compute number of samples that are seven or greater: n\_large
* n\_large = np.sum(n\_nohitters >= 7)
* # Compute probability of getting seven or more: p\_large
* p\_large = n\_large/10000
* # Print the result
* print('Probability of seven or more no-hitters:', p\_large)

# Draw 10,000 samples out of Poisson distribution: n\_nohitters

n\_nohitters = np.random.poisson(251/115, size=10000)

# Compute number of samples that are seven or greater: n\_large

n\_large = np.sum(n\_nohitters >= 7)

# Compute probability of getting seven or more: p\_large

p\_large = n\_large/10000

# Print the result

print('Probability of seven or more no-hitters:', p\_large)

Probability of seven or more no-hitters: 0.0067

The result is about 0.007. This means that it is not that improbable to see a 7-or-more no-hitter season in a century. We have seen two in a century and a half, so it is not unreasonable.

# Probability density functions

**50 XP**

## 1. Probability density functions

So far, we have talked about probabilities of discrete quantities, such as die rolls and number of bus arrivals, but what about continuous quantities?

## 2. Continuous variables

A continuous quantity can take on any value, not just discrete ones. For example, the speed of a train can be 45-point-76 km/h.

## 3. Michelson's speed of light experiment

Continuous variables also have probability distributions. Let's consider an example. In 1879, Albert Michelson performed 100 measurements of the speed of light in air.

## 4. Michelson's speed of light experiment

Each measurement has some error in it; conditions, such as temperature, humidity, alignment of his optics, et cetera, change from measurement to measurement. As a result, any fractional value of the measured speed of light is possible, so it is apt to describe the results with a continuous probability distribution. In looking at Michelson's numbers, shown here in units of megameters/s, or 1000s of kilometers/s, we see this is indeed the case. What probability distribution describes these data? I posit that these data follow the famous Normal distribution. To understand what the Normal distribution is,

## 5. Probability density function (PDF)

let's consider its probability density function, or PDF. This is the continuous analog to the probability mass function, the PMF. It describes the chances of observing a value of a continuous variable. The probability of observing a single value of the speed of light does not make sense, because there is an infinity of numbers,

## 6. Normal PDF

say between 299-point-6 and 300-point-0 megameters per second. Instead, areas under the PDF give probabilities. So, the probability of measuring that the speed of light is greater than

## 7. Normal PDF

300,000 km/s is an area under the normal curve. Parametrizing the PDF based on Michelson's experiments, this is about a 3% chance, since the pink region is about 3% of the total area under the PDF. To do this calculation, we were really just looking at the cumulative distribution function,

## 8. Normal CDF

or CDF, of the Normal distribution. Here is the CDF of the Normal distribution. Remember that the CDF gives the probability the measured speed of light will be less than the value on the x-axis. So, reading off the value at 300,000 km/s,

## 9. Normal CDF

we see that there is a 97% chance that a speed of light measurement is less than that. So, there is about a 3% change it is greater. We will study the Normal distribution in more depth in the coming exercises, but for right now,

## 10. Let's practice!

let's review some of the concepts we've learned about continuous distribution functions.

## Question 5 of 8

You have a file called Planets.csv. Assuming you already have the line import pandas as pd, how can you import this file into a DataFrame called planets using pandas?

* 

**planets = pd.read\_csv('Pandas.csv')**

* 

planets = pd.read\_csv('Pandas.csv',usecols=[0,1,2,3])

* 

import pandas as pd

## Question 6 of 8

Which set\_index() keyword should be set to TRUE to modify a given DataFrame as opposed to creating a copy?

* 

DataFrame

* 

append

* 

**Inplace**

## Question 7 of 8

How can you determine the length of a DataFrame called planets in pandas?

* 

**len(planets)**

* 

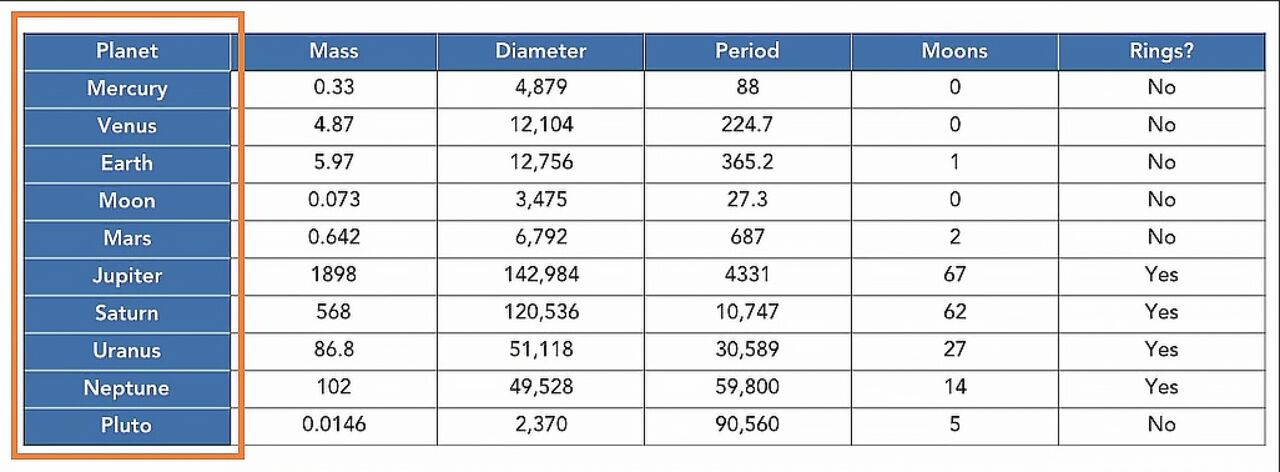
info(planets)

* 

planets.loc()

## Question 8 of 8

What should you consider the column of planets in the table shown?



* 

**Cases Correct**

**Each case will have a list of attributes called variables.**

Variables

categorical data