**Course: Statistical Thinking in Python**

**Chapter 1: Graphical Exploratory Analysis**

**Tukey's comments on EDA**

* 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.

**Advantages of graphical EDA**

* It often involves converting tabular data into graphical form.
* If done well, graphical representations can allow for more rapid interpretation of data.
* There is no excuse for neglecting to do graphical EDA.

**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. 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.

* Import matplotlib.pyplot and seaborn as their usual aliases (plt and sns).

# Import plotting modules

import matplotlib.pyplot as plt

import seaborn as sns

* Use seaborn to set the plotting defaults.

# Set default Seaborn style

sns.set()

* Plot a histogram of the *Iris versicolor* petal lengths using plt.hist() and the provided NumPy array versicolor\_petal\_length.

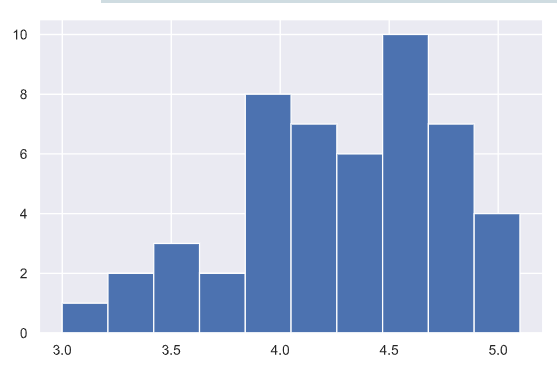
# Plot histogram of versicolor petal lengths

plt.hist(versicolor\_petal\_length)

* Show the histogram using plt.show().

# 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.

* Label the axes. Don't forget that you should always include units in your axis labels. Your yy-axis label is just 'count'. Your xx-axis label is 'petal length (cm)'. The units are essential!

# Plot histogram of versicolor petal lengths

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

# Label axes

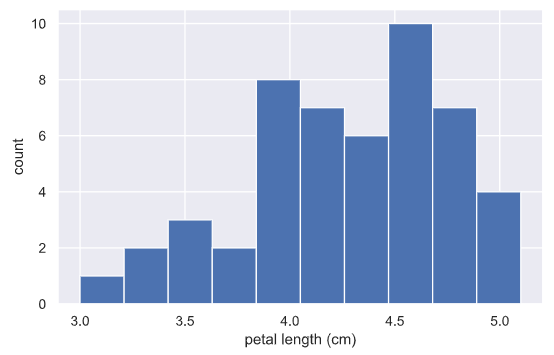
plt.xlabel('petal length (cm)')

plt.ylabel('count')

* Display the plot constructed in the above steps using plt.show().

# Show histogram

plt.show()



**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.

* Import numpy as np. This gives access to the square root function, np.sqrt().

# Import numpy

import numpy as np

* Determine how many data points you have using len().

# Compute number of data points: n\_data

n\_data = len(versicolor\_petal\_length)

* Compute the number of bins using the square root rule.

# Number of bins is the square root of number of data points: n\_bins

n\_bins = np.sqrt(n\_data)

* Convert the number of bins to an integer using the built in int() function.

# Convert number of bins to integer: n\_bins

n\_bins = int(n\_bins)

* Generate the histogram and make sure to use the bins keyword argument.

# Plot the histogram

\_ = plt.hist(versicolor\_petal\_length, bins = n\_bins)

* Hit 'Submit Answer' to plot the figure and see the fruit of your labors!

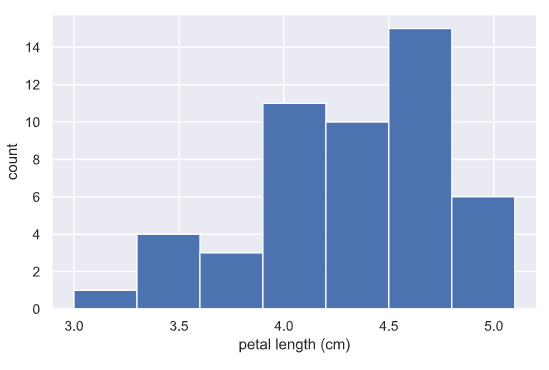
# Label axes

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

\_ = plt.ylabel('count')

# Show histogram

plt.show()



**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.

* 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().

# Create bee swarm plot with Seaborn's default settings

df.head()

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

* 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.

# Label the axes

\_ = plt.xlabel('species')

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

* Show your plot.

# Show the plot

plt.show()



**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.

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

**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)!

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

**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().

* Use ecdf() to compute the ECDF of versicolor\_petal\_length. Unpack the output into x\_vers and y\_vers.

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

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

* 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().

# Generate plot

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

* Label the axes. You can label the y-axis 'ECDF'.

# Label the axes

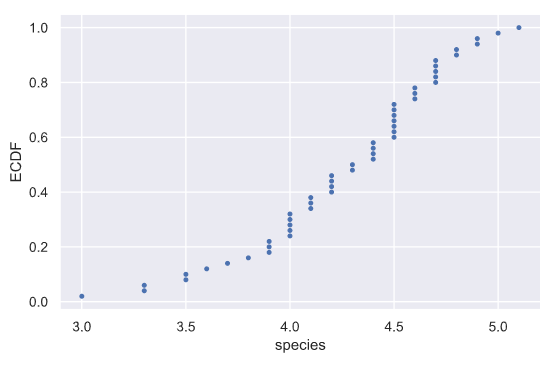
\_ = plt.xlabel('species')

\_ = plt.ylabel('ECDF')

* Show your plot.

# 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().

* 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.

# 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 three ECDFs on the same plot as dots. To do this, you will need three plt.plot() commands. Assign the result of each to \_.

# 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')

* A legend and axis labels have been added for you, so hit 'Submit Answer' to see all the ECDFs!

# Annotate the plot

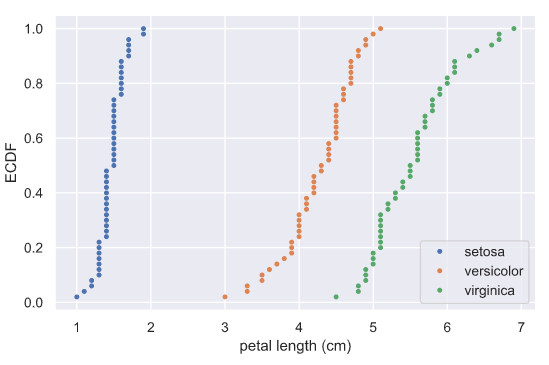
\_ = plt.legend(('setosa', 'versicolor', 'virginica'), loc='lower right')

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

\_ = plt.ylabel('ECDF')

# Display the plot

plt.show()



**Chapter 2: Quantitative Exploratory Data Analysis**

**Means and medians**

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

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

**Computing means**

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

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.

# 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')

**Computing percentiles**

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

* 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.

# Specify array of percentiles: percentiles

percentiles = np.array([2.5, 25, 50, 75, 97.5])

* 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.

# Compute percentiles: ptiles\_vers

ptiles\_vers = np.percentile(versicolor\_petal\_length, percentiles)

* Print the percentiles.

print(ptiles\_vers)

**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.

* 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.

# Plot the ECDF

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

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

\_ = plt.ylabel('ECDF')

# Overlay percentiles as red diamonds.

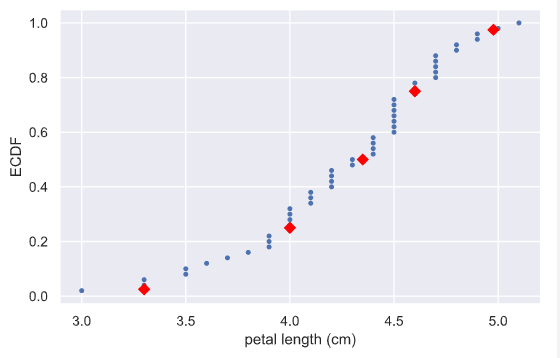
\_ = plt.plot(ptiles\_vers, percentiles/100, marker='D', color='red',

linestyle='none')

* Display the plot.

# 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.

* 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)'.

# Create box plot with Seaborn's default settings

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

* Don't forget to label your axes!

# Label the axes

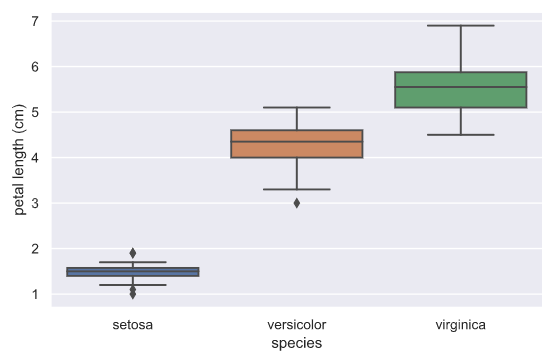
\_ = plt.xlabel('species')

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

* Display the figure using the normal call.

# Show the plot

plt.show()



**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.

* 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.

# Array of differences to mean: differences

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

* Square each element in this array. For example, x\*\*2 squares each element in the array x. Store the result as diff\_sq.

# Square the differences: diff\_sq

diff\_sq = differences \*\* 2

* Compute the mean of the elements in diff\_sq using np.mean(). Store the result as variance\_explicit.

# Compute the mean square difference: variance\_explicit

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

* Compute the variance of versicolor\_petal\_length using np.var(). Store the result as variance\_np.

# Compute the variance using NumPy: variance\_np

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

* Print both variance\_explicit and variance\_np in one print call to make sure they are consistent.

# Print the results

print(variance\_explicit, variance\_np)

**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.

* Compute the variance of the data in the versicolor\_petal\_length array using np.var() and store it in a variable called variance.

# Compute the variance: variance

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

* Print the square root of this value.

# Print the square root of the variance

print (np.sqrt(variance))

* Print the standard deviation of the data in the versicolor\_petal\_length array using np.std().

# Print the standard deviation

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

**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')

* 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.

# Make a scatter plot

\_ = plt.plot(versicolor\_petal\_length, versicolor\_petal\_width, marker = '.', linestyle = 'none')

* Label the axes.

# Label the axes

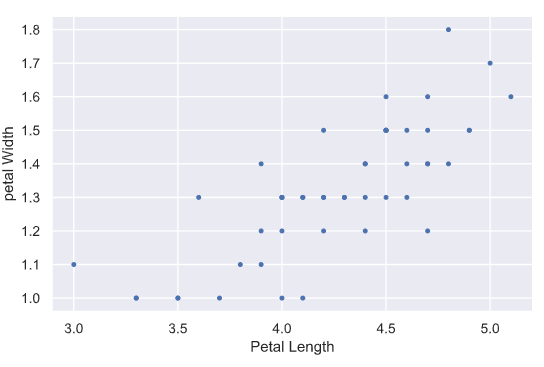
\_ = plt.xlabel('Petal Length')

\_ = plt.ylabel('petal Width')

* Display the plot.

# Show the result

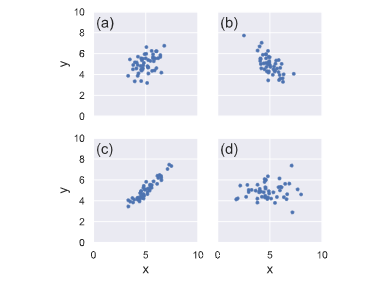
plt.show()



**Variance and covariance by looking**

Consider four scatter plots of xx-yy data, appearing to the right. Which has, respectively,

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

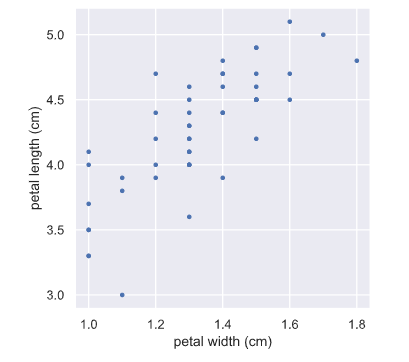


**Answer:** d, c, b

**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.



* Use np.cov() to compute the covariance matrix for the petal length (versicolor\_petal\_length) and width (versicolor\_petal\_width) of *I. versicolor*.

# Compute the covariance matrix: covariance\_matrix

covariance\_matrix = np.cov(versicolor\_petal\_length, versicolor\_petal\_width)

* Print the covariance matrix.

# Print covariance matrix

print(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].

# Extract covariance of length and width of petals: petal\_cov

petal\_cov = covariance\_matrix[0, 1]

* Print the covariance.

# Print the length/width covariance

print(petal\_cov)

**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.

* 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.

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 the Pearson correlation between the data in the arrays versicolor\_petal\_length and versicolor\_petal\_width. Assign the result to r.

# Compute Pearson correlation coefficient for I. versicolor: r

r = pearson\_r(versicolor\_petal\_length, versicolor\_petal\_width)

* Print the result.

# Print the result

print(r)

**Chapter 3: Thinking Probabilistically – Discrete Variables**

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

Why do we do statistical inference?

* 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.

Why do we use the language of probability?

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

**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.

* Seed the random number generator using the seed 42.

# Seed the random number generator

np.random.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.

# Initialize random numbers: random\_numbers

random\_numbers = np.empty(100000)

* 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).

# Generate random numbers by looping over range(100000)

for i in range(100000):

random\_numbers[i] = np.random.random()

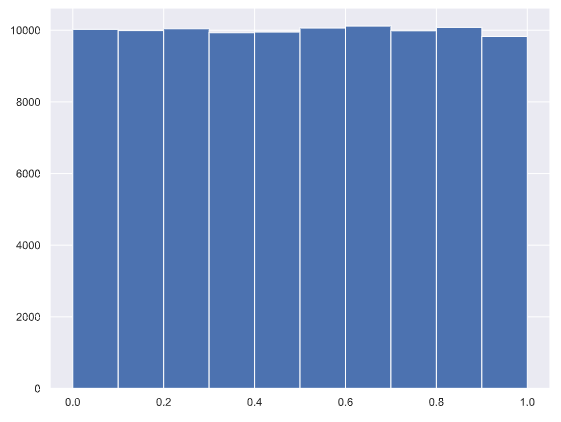
* Plot a histogram of random\_numbers. It is not necessary to label the axes in this case because we are just checking the random number generator. Hit 'Submit Answer' to show your plot.

# Plot a histogram

\_ = plt.hist(random\_numbers)

# Show the plot

plt.show()



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 pp of landing heads (success) and probability 1−p1−p 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.

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 = 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

**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.

* Seed the random number generator to 42.

# Seed random number generator

np.random.seed(42)

* Initialize n\_defaults, an empty array, using np.empty(). It should contain 1000 entries, since we are doing 1000 simulations.

# Initialize the number of defaults: n\_defaults

n\_defaults = np.empty(1000)

* 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.

# Compute the number of defaults

for i in range(1000):

n\_defaults[i] = perform\_bernoulli\_trials(100, 0.05)

* 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.

# Plot the histogram with default number of bins; label your axes

\_ = plt.hist(n\_defaults, normed=True)

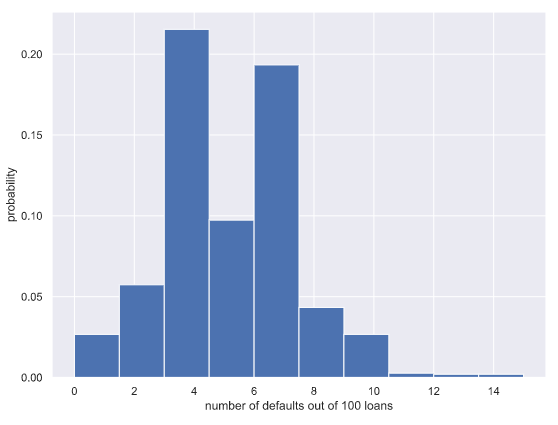
\_ = plt.xlabel('number of defaults out of 100 loans')

\_ = plt.ylabel('probability')

* Show your plot.

# Show the plot

plt.show()



**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?

* Compute the x and y values for the ECDF of n\_defaults.

# Compute ECDF: x, y

x, y = ecdf(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().

# Plot the ECDF with labeled axes

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

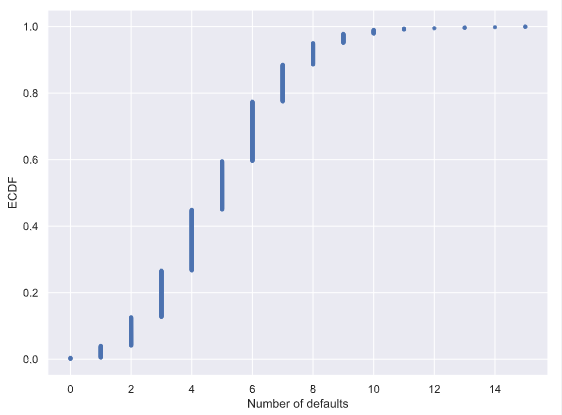
\_ = plt.xlabel('Number of defaults')

\_ = plt.ylabel('ECDF')

* Show the plot.

# Show the plot

plt.show()



* 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.

# Compute the number of 100-loan simulations with 10 or more defaults: n\_lose\_money

n\_lose\_money = np.sum(n\_defaults >= 10)

* 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 Answer'!

# Compute and print probability of losing money

print('Probability of losing money =', n\_lose\_money / len(n\_defaults))

**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.

* 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.

# Take 10,000 samples out of the binomial distribution: n\_defaults

n\_defaults = np.random.binomial(n=100, p=0.05, size = 10000)

* Compute the CDF using your previously-written ecdf() function.

# Compute CDF: x, y

x, y = ecdf(n\_defaults)

* 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.

# Plot the CDF with axis labels

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

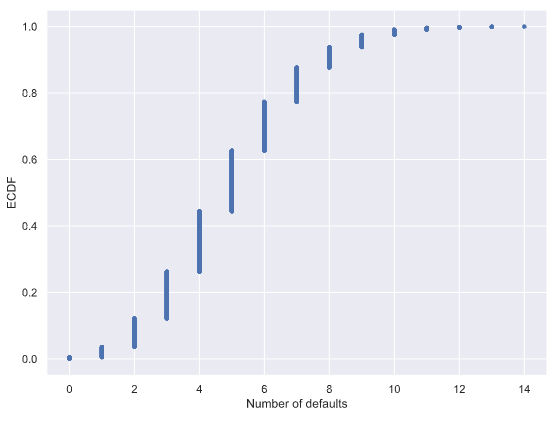
\_ = plt.xlabel('Number of defaults')

\_ = plt.ylabel('ECDF')

* Show the plot.

# Show the plot

plt.show()



**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.

* Using np.arange(), compute the bin edges such that the bins are centered on the integers. Store the resulting array in the variable bins.

# Compute bin edges: bins

bins = np.arange(0, max(n\_defaults) + 1.5) - 0.5

* Use plt.hist() to plot the histogram of n\_defaults with the normed=True and bins=bins keyword arguments.

# Generate histogram

\_ = plt.hist(n\_defaults, normed=True, bins=bins)

# Label axes

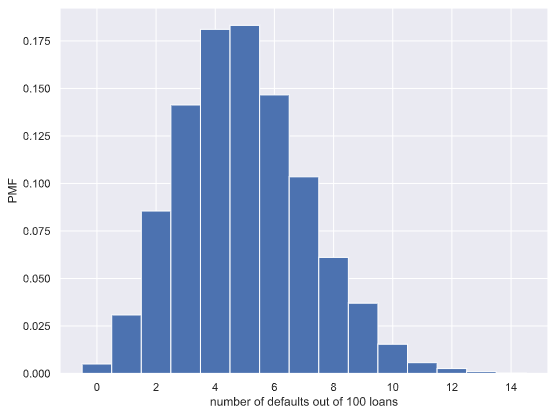
\_ = plt.xlabel('number of defaults out of 100 loans')

\_ = plt.ylabel('PMF')

* Show the plot.

# Show the plot

plt.show()



**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 npnp approximates a Binomial distribution for nn Bernoulli trials with probability pp of success (with nn large and pp 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 nn and pp such that np=10np=10.

* Using the np.random.poisson() function, draw 10000 samples from a Poisson distribution with a mean of 10.

# 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))

* 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 npnp is always 10.

# Specify values of n and p to consider for Binomial: n, p

n = [20, 100, 1000]

p = [0.5, 0.1, 0.01]

* 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 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))

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?

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

**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?

* Draw 10000 samples from a Poisson distribution with a mean of 251/115 and assign to n\_nohitters.

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

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

* Determine how many of your samples had a result greater than or equal to 7 and assign to n\_large.

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

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

* Compute the probability, p\_large, of having 7 or more no-hitters by dividing n\_large by the total number of samples (10000).

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

p\_large = n\_large/len(n\_nohitters)

* Hit 'Submit Answer' to print the probability that you calculated.

# Print the result

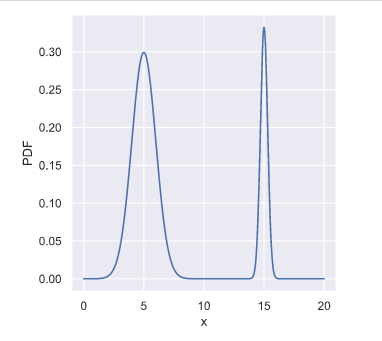
print('Probability of seven or more no-hitters:', p\_large)

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.

**Chapter 4: Thinking Probabilistically – Continuous Variables**

**Interpreting PDFs**

Consider the PDF shown to the right (it may take a second to load!). Which of the following is true?

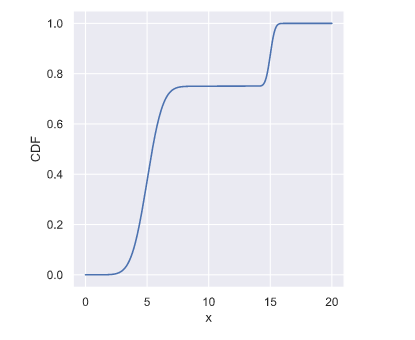


**Answer:** x is more likely to be **less than 10** than to be greater than 10.

The probability is given by the *area under the PDF*, and there is more area to the left of 10 than to the right.

**Interpreting CDFs**

At right is the CDF corresponding to the PDF you considered in the last exercise. Using the CDF, what is the probability that xx is greater than 10?



**Answer:** 0.25

The value of the CDF at x = 10 is 0.75, so the probability that x < 10 is 0.75. Thus, the probability that x > 10 is 0.25.

**The Normal PDF**

In this exercise, you will explore the Normal PDF and also learn a way to plot a PDF of a known distribution using hacker statistics. Specifically, you will plot a Normal PDF for various values of the variance.

* Draw 100,000 samples from a Normal distribution that has a mean of 20 and a standard deviation of 1. Do the same for Normal distributions with standard deviations of 3 and 10, each still with a mean of 20. Assign the results to samples\_std1, samples\_std3 and samples\_std10, respectively.

# Draw 100000 samples from Normal distribution with stds of interest: samples\_std1, samples\_std3, samples\_std10

samples\_std1 = np.random.normal(20, 1, size=100000)

samples\_std3 = np.random.normal(20, 3, size=100000)

samples\_std10 = np.random.normal(20, 10, size=100000)

* Plot a histograms of each of the samples; for each, use 100 bins, also using the keyword arguments normed=True and histtype='step'. The latter keyword argument makes the plot look much like the smooth theoretical PDF. You will need to make 3 plt.hist() calls.

# Make histograms

\_ = plt.hist(samples\_std1, bins=100, normed=True, histtype='step')

\_ = plt.hist(samples\_std3, bins=100, normed=True, histtype='step')

\_ = plt.hist(samples\_std10, bins=100, normed=True, histtype='step')

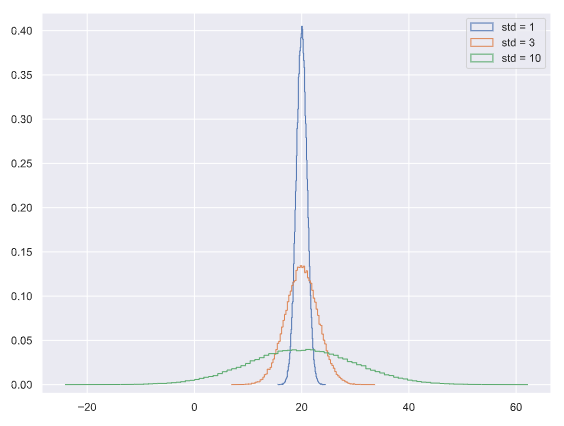
* Hit 'Submit Answer' to make a legend, showing which standard deviations you used, and show your plot! There is no need to label the axes because we have not defined what is being described by the Normal distribution; we are just looking at shapes of PDFs.

# Make a legend, set limits and show plot

\_ = plt.legend(('std = 1', 'std = 3', 'std = 10'))

plt.ylim(-0.01, 0.42)

plt.show()



You can see how the different standard deviations result in PDFs of different widths. The peaks are all centered at the mean of 20.

**The Normal CDF**

Now that you have a feel for how the Normal PDF looks, let's consider its CDF. Using the samples you generated in the last exercise (in your namespace as samples\_std1, samples\_std3, and samples\_std10), generate and plot the CDFs.

* Use your ecdf() function to generate x and y values for CDFs: x\_std1, y\_std1, x\_std3, y\_std3 and x\_std10, y\_std10, respectively.

# Generate CDFs

x\_std1, y\_std1 = ecdf(samples\_std1)

x\_std3, y\_std3 = ecdf(samples\_std3)

x\_std10, y\_std10 = ecdf(samples\_std10)

* Plot all three CDFs as dots (do not forget the marker and linestyle keyword arguments!).

# Plot CDFs

\_ = plt.plot(x\_std1, y\_std1, marker = '.', linestyle = 'none')

\_ = plt.plot(x\_std3, y\_std3, marker = '.', linestyle = 'none')

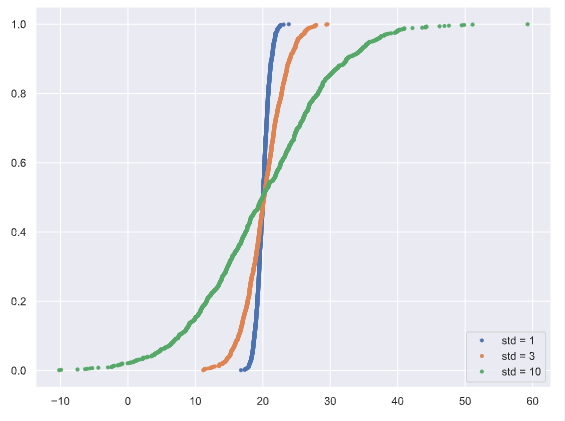
\_ = plt.plot(x\_std10, y\_std10, marker = '.', linestyle = 'none')

* Hit submit to make a legend, showing which standard deviations you used, and to show your plot. There is no need to label the axes because we have not defined what is being described by the Normal distribution; we are just looking at shapes of CDFs.

# Make a legend and show the plot

\_ = plt.legend(('std = 1', 'std = 3', 'std = 10'), loc='lower right')

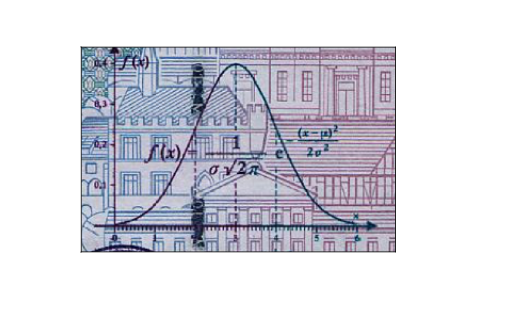
plt.show()



 The CDFs all pass through the mean at the 50th percentile; the mean and median of a Normal distribution are equal. The width of the CDF varies with the standard deviation.

**Gauss and the 10 Deutschmark banknote**

What are the mean and standard deviation, respectively, of the Normal distribution that was on the 10 Deutschmark banknote, shown to the right?



**Answer:** mean = 3, std = 1

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

Since 1926, the Belmont Stakes is a 1.5 mile-long race of 3-year old thoroughbred horses. [**Secretariat**](https://en.wikipedia.org/wiki/Secretariat_(horse)) ran the fastest Belmont Stakes in history in 1973. While that was the fastest year, 1970 was the slowest because of unusually wet and sloppy conditions. With these two outliers removed from the data set, compute the mean and standard deviation of the Belmont winners' times. Sample out of a Normal distribution with this mean and standard deviation using the np.random.normal() function and plot a CDF. Overlay the ECDF from the winning Belmont times. Are these close to Normally distributed?

*Note*: Justin scraped the data concerning the Belmont Stakes from the [**Belmont Wikipedia page**](https://en.wikipedia.org/wiki/Belmont_Stakes).

* Compute mean and standard deviation of Belmont winners' times with the two outliers removed. The NumPy array belmont\_no\_outliers has these data.

# Compute mean and standard deviation: mu, sigma

mu = np.mean(belmont\_no\_outliers)

sigma = np.std(belmont\_no\_outliers)

* Take 10,000 samples out of a normal distribution with this mean and standard deviation using np.random.normal().

# Sample out of a normal distribution with this mu and sigma: samples

samples = np.random.normal(mu, sigma, size = 10000)

* Compute the CDF of the theoretical samples and the ECDF of the Belmont winners' data, assigning the results to x\_theor, y\_theor and x, y, respectively.

# Get the CDF of the samples and of the data

x\_theor, y\_theor = ecdf(samples)

x, y = ecdf(belmont\_no\_outliers)

* Hit submit to plot the CDF of your samples with the ECDF, label your axes and show the plot.

# Plot the CDFs and show the plot

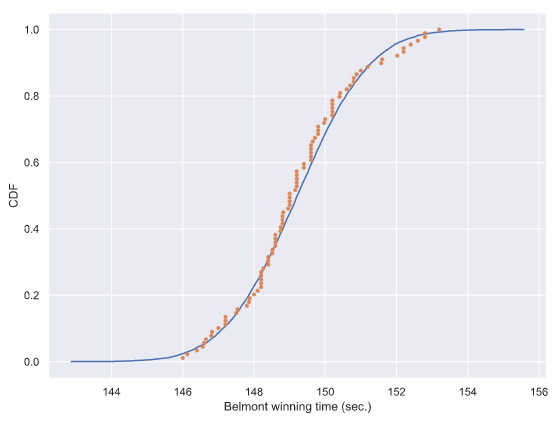
\_ = plt.plot(x\_theor, y\_theor)

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

\_ = plt.xlabel('Belmont winning time (sec.)')

\_ = plt.ylabel('CDF')

plt.show()



The theoretical CDF and the ECDF of the data suggest that the winning Belmont times are, indeed, Normally distributed. This also suggests that in the last 100 years or so, there have not been major technological or training advances that have significantly affected the speed at which horses can run this race.

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

Assume that the Belmont winners' times are Normally distributed (with the 1970 and 1973 years removed), what is the probability that the winner of a given Belmont Stakes will run it as fast or faster than Secretariat?

* Take 1,000,000 samples from the normal distribution using the np.random.normal() function. The mean mu and standard deviation sigma are already loaded into the namespace of your IPython instance.

# Take a million samples out of the Normal distribution: samples

samples = np.random.normal(mu, sigma, size = 1000000)

* Compute the fraction of samples that have a time less than or equal to Secretariat's time of 144 seconds.

# Compute the fraction that are faster than 144 seconds: prob

prob = np.sum(samples <= 144)/len(samples)

* Print the result.

# Print the result

print('Probability of besting Secretariat:', prob)

We had to take a million samples because the probability of a fast time is very low and we had to be sure to sample enough. We get that there is only a 0.06% chance of a horse running the Belmont as fast as Secretariat.

**Matching a story and a distribution**

How might we expect the time between Major League no-hitters to be distributed? Be careful here: a few exercises ago, we considered the probability distribution for the number of no-hitters in a season. Now, we are looking at the probability distribution of the *time between* no hitters.

**Answer:** Exponential

**Waiting for the next Secretariat**

Unfortunately, Justin was not alive when Secretariat ran the Belmont in 1973. Do you think he will get to see a performance like that? To answer this, you are interested in how many years you would expect to wait until you see another performance like Secretariat's. How is the waiting time until the next performance as good or better than Secretariat's distributed? Choose the best answer.

Exponential: A horse as fast as Secretariat is a rare event, which can be modeled as a Poisson process, and the waiting time between arrivals of a Poisson process is Exponentially distributed.

The Exponential distribution describes the waiting times between rare events

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

Sometimes, the story describing our probability distribution does not have a named distribution to go along with it. In these cases, fear not! You can always simulate it. We'll do that in this and the next exercise.

In earlier exercises, we looked at the rare event of no-hitters in Major League Baseball. *Hitting the cycle* is another rare baseball event. When a batter hits the cycle, he gets all four kinds of hits, a single, double, triple, and home run, in a single game. Like no-hitters, this can be modeled as a Poisson process, so the time between hits of the cycle are also Exponentially distributed.

How long must we wait to see both a no-hitter *and then* a batter hit the cycle? The idea is that we have to wait some time for the no-hitter, and then after the no-hitter, we have to wait for hitting the cycle. Stated another way, what is the total waiting time for the arrival of two different Poisson processes? The total waiting time is the time waited for the no-hitter, plus the time waited for the hitting the cycle.

Now, you will write a function to sample out of the distribution described by this story.

* Define a function with call signature successive\_poisson(tau1, tau2, size=1) that samples the waiting time for a no-hitter and a hit of the cycle.
  + Draw waiting times tau1 (size number of samples) for the no-hitter out of an exponential distribution and assign to t1.
  + Draw waiting times tau2 (size number of samples) for hitting the cycle out of an exponential distribution and assign to t2.
  + The function returns the sum of the waiting times for the two events.

def successive\_poisson(tau1, tau2, size=1):

"""Compute time for arrival of 2 successive Poisson processes."""

# Draw samples out of first exponential distribution: t1

t1 = np.random.exponential(tau1, size)

# Draw samples out of second exponential distribution: t2

t2 = np.random.exponential(tau2, size)

return t1 + t2

**Distribution of no-hitters and cycles**

Now, you'll use your sampling function to compute the waiting time to observe a no-hitter and hitting of the cycle. The mean waiting time for a no-hitter is 764 games, and the mean waiting time for hitting the cycle is 715 games.

* Use your successive\_poisson() function to draw 100,000 out of the distribution of waiting times for observing a no-hitter and a hitting of the cycle.

# Draw samples of waiting times: waiting\_times

waiting\_times = successive\_poisson(764, 715, 100000)

* Plot the PDF of the waiting times using the step histogram technique of a previous exercise. Don't forget the necessary keyword arguments. You should use bins=100, normed=True, and histtype='step'.

# Make the histogram

\_ = plt.hist(waiting\_times, normed = True, bins = 100, histtype = 'step')

* Label the axes.

# Label axes

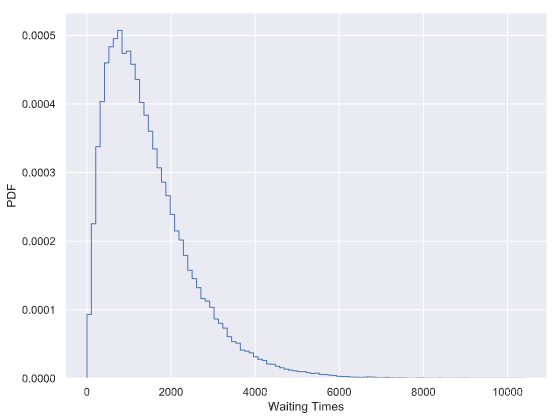
\_ = plt.xlabel('Waiting Times')

\_ = plt.ylabel('PDF')

* Show your plot.

# Show the plot

plt.show()



 Notice that the PDF is peaked, unlike the waiting time for a single Poisson process. For fun (and enlightenment), I encourage you to also plot the CDF.