

Statistical Thinking in Python (Part 1)

Speak the statistical language of your data



[Black Raven \(James Ng\)](#)

[27 Jun 2020](#) · 27 min read

This is a tutorial to share what I have learnt in Statistical Thinking in Python (Part 1), capturing the learning objectives as well as my personal notes. The course is taught by Justin Bois from DataCamp, and it includes 4 chapters:

Chapter 1. Graphical exploratory data analysis

Chapter 2. Quantitative exploratory data analysis

Chapter 3. Thinking probabilistically – Discrete variables

Chapter 4. Thinking probabilistically – Continuous variables



Photo by [Chris Liverani](#) on [Unsplash](#)

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, summary 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.

Chapter 1 Graphical exploratory data analysis

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 (EDA)

EDA is the process of organising, plotting, and summarising a data set. It is developed by a great statistician John Tukey.

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

John 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.

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.

Plotting a histogram

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.

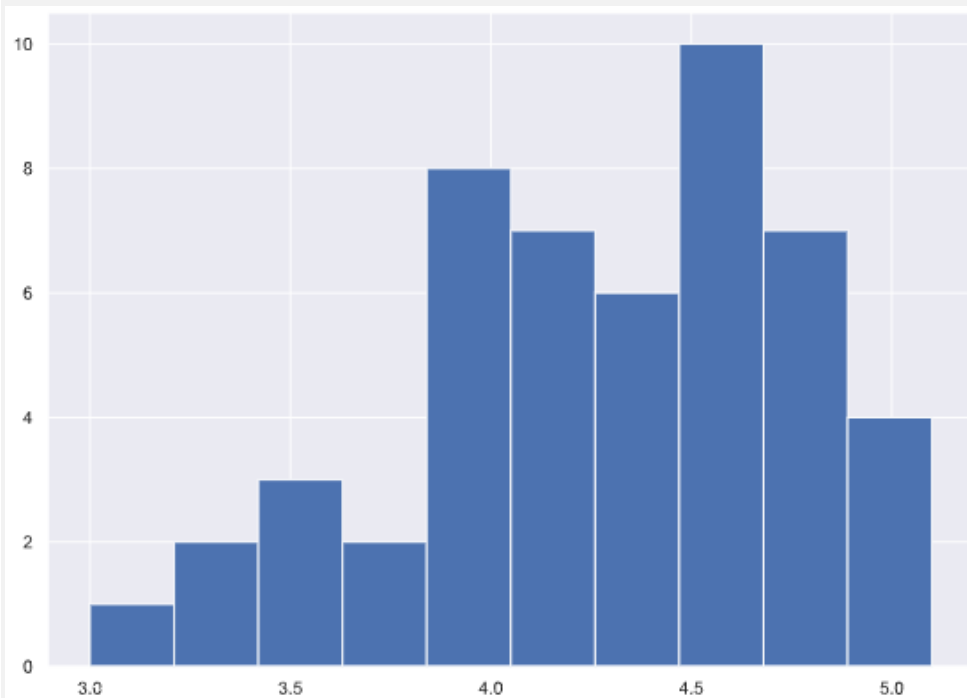
The subset of the data set containing the *Iris versicolor* petal lengths in units of centimeters (cm) is stored in the NumPy array `versicolor_petal_length`.

```
In [1]: versicolor_petal_length
Out[1]:
array([4.7, 4.5, 4.9, 4. , 4.6, 4.5, 4.7, 3.3, 4.6, 3.9, 3.5, 4.2, 4. , 4.7, 3.6, 4.4, 4.5, 4.1, 4.5, 3.9,
4.8, 4. , 4.9, 4.7, 4.3, 4.4, 4.8, 5. , 4.5, 3.5, 3.8, 3.7, 3.9, 5.1, 4.5, 4.5, 4.7, 4.4, 4.1, 4. , 4.4, 4.6,
4. , 3.3, 4.2, 4.2, 4.2, 4.3, 3. , 4.1])

# Set default Seaborn style
sns.set()

# Plot histogram of versicolor petal lengths
plt.hist(versicolor_petal_length)

# Show histogram
plt.show()
```



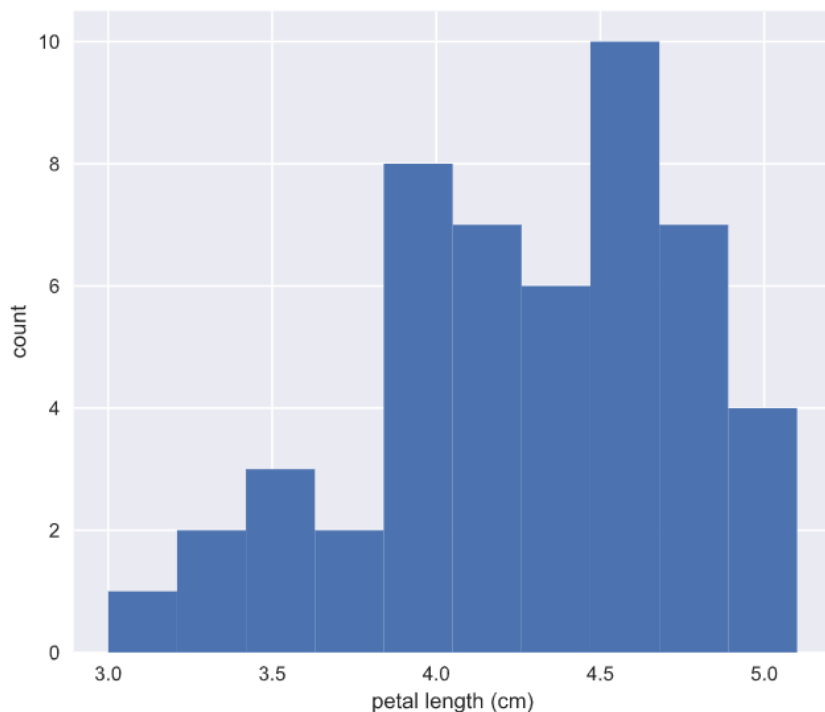
Axis labels!

In the last exercise, a nice histogram of petal lengths of *Iris versicolor* was made, but **the axes are not labelled!** Now, add axis labels to the plot using `plt.xlabel()` and `plt.ylabel()`. The packages `matplotlib.pyplot` and `seaborn` are already imported with their standard aliases.

```
# 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()
```



Adjusting the number of bins in a histogram

The previous histogram had 10 bins, which 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. 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 `versicolor_petal_length` has been defined in the last exercise.

```
# Import numpy
import numpy as np

# Compute number of data points: n_data
n_data = len(versicolor_petal_length)

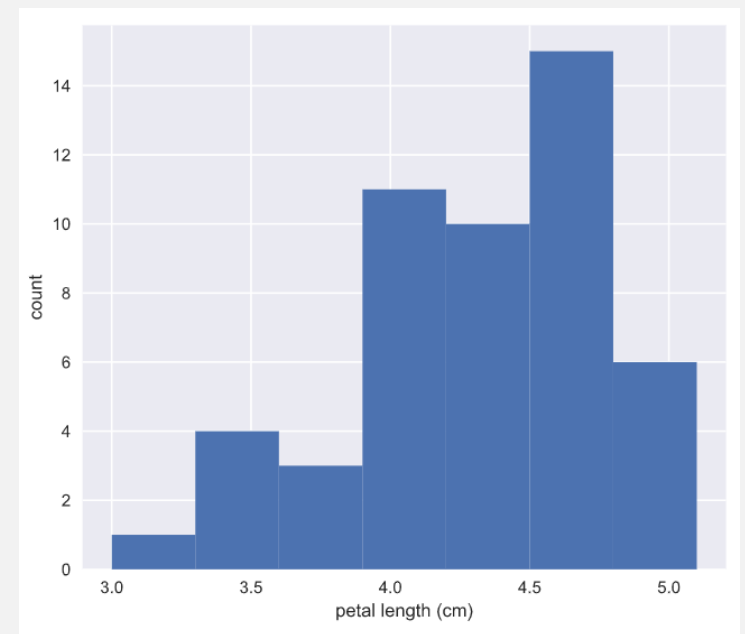
# Number of bins is the square root of number of data points: n_bins
n_bins = np.sqrt(n_data)

# 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()
```



Plot all of your data: Bee swarm plots

Bee swarm plot

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

```
In [1]: df
Out[1]:
```

	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]

Here is the code to create the bee swarm plot:

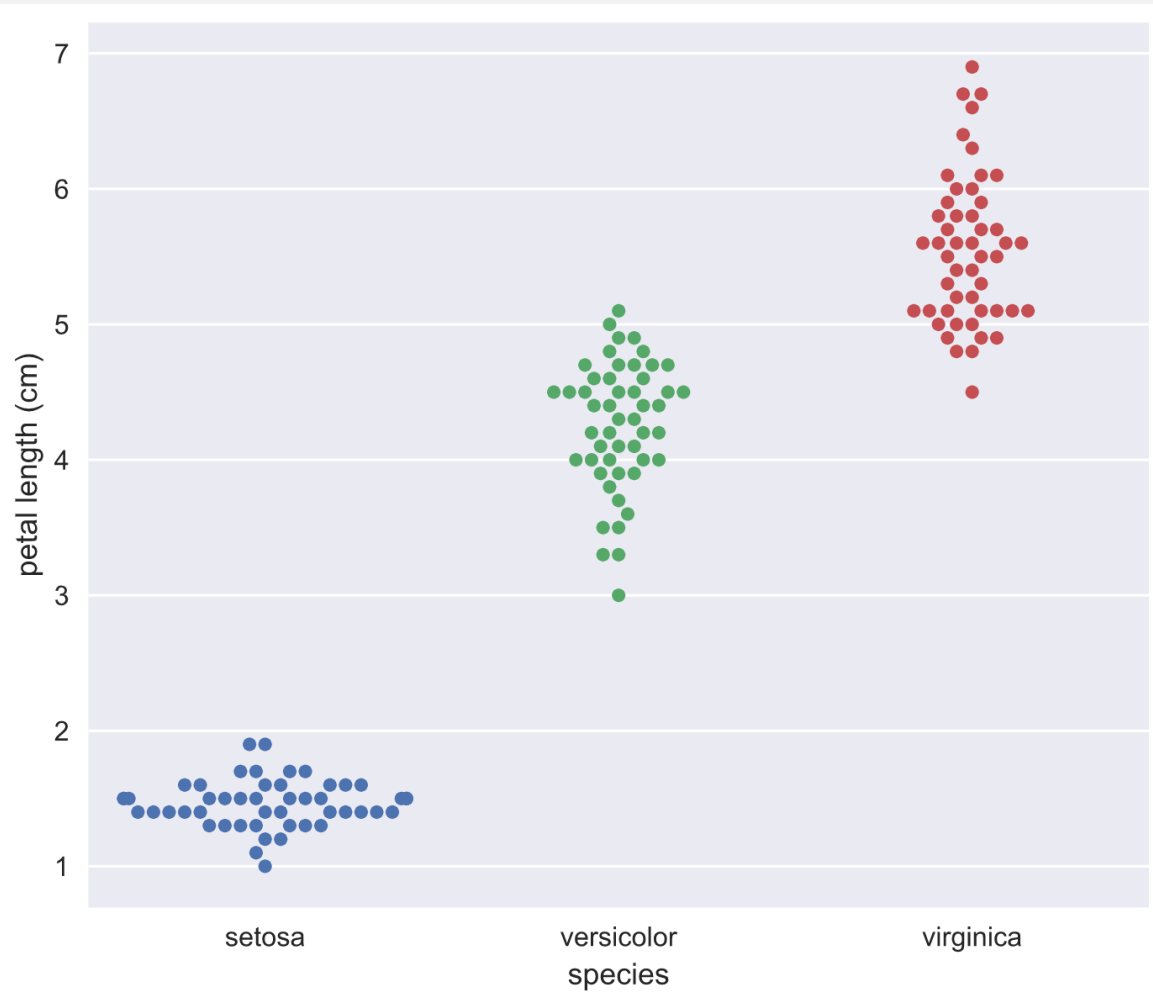
```
sns.swarmplot(x='state', y='dem_share', data=df_swing)
plt.xlabel('state')
plt.ylabel('percent of vote for Obama')
plt.show()
```

You can use `sns.swarmplot?` or `help(sns.swarmplot)` for more details on how to make bee swarm plots using seaborn.


```
# 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()
```



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 (see plot above)?

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

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’s data.

Plot all of your data: Empirical Cumulative Distribution Functions (ECDF)

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`.

```
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

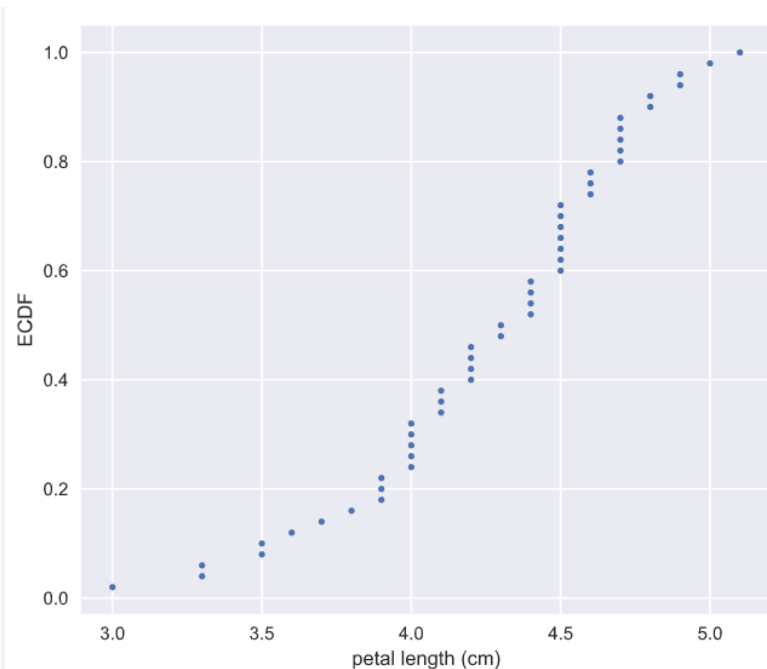
You will now use the above `ecdf()` function to compute the ECDF for the petal lengths of Anderson's *Iris versicolor* flowers. You will then plot the ECDF. Recall that the `ecdf()` function returns two arrays so you will need to unpack them. An example of such unpacking is `x, y = foo(data)`, for some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

```
# 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_petal_length')
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.

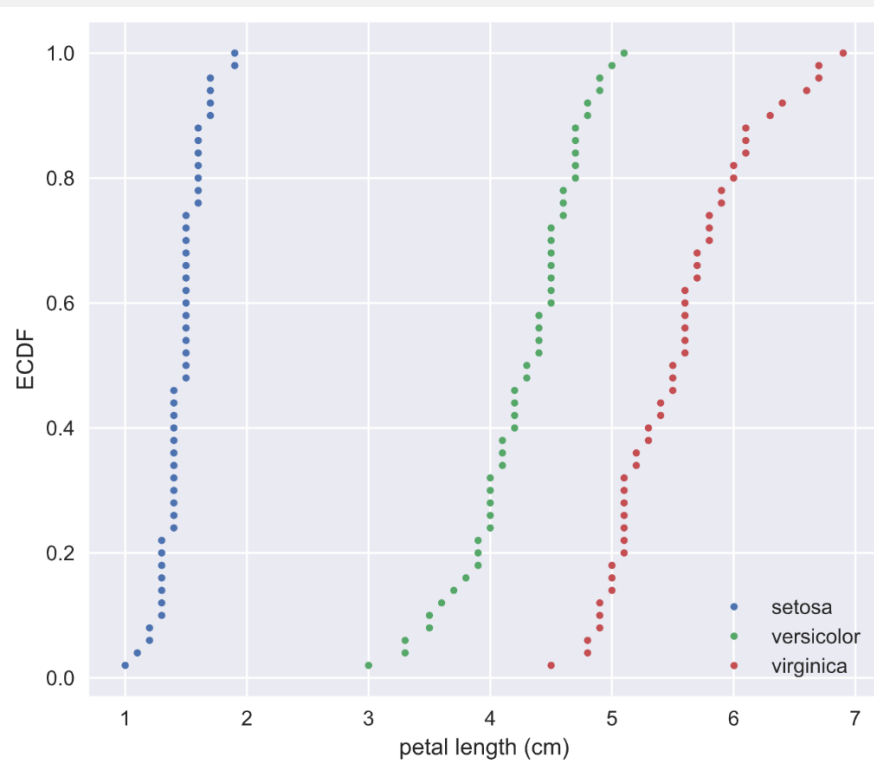
Remember to include `marker='.'` and `linestyle='none'` as arguments inside `plt.plot()`.

```
# 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')

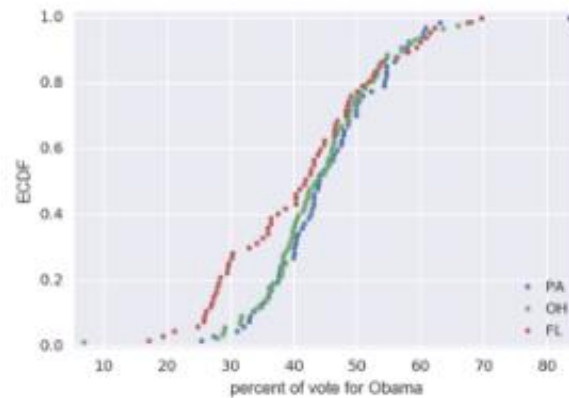
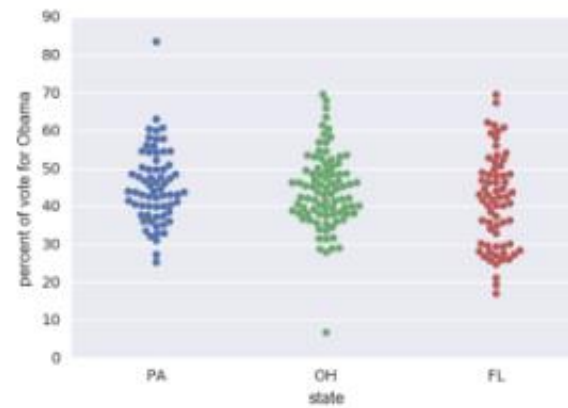
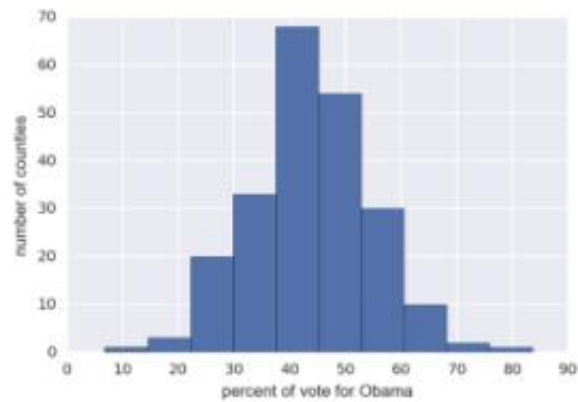
# Display the plot
plt.show()
```



Observation: 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!

We have learnt 3 types of plots to summarise data — histogram, bee swarm plot, and ECDF.



Chapter 2. Quantitative exploratory data analysis

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

Means and medians

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

- ☐ 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.

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

Computing means

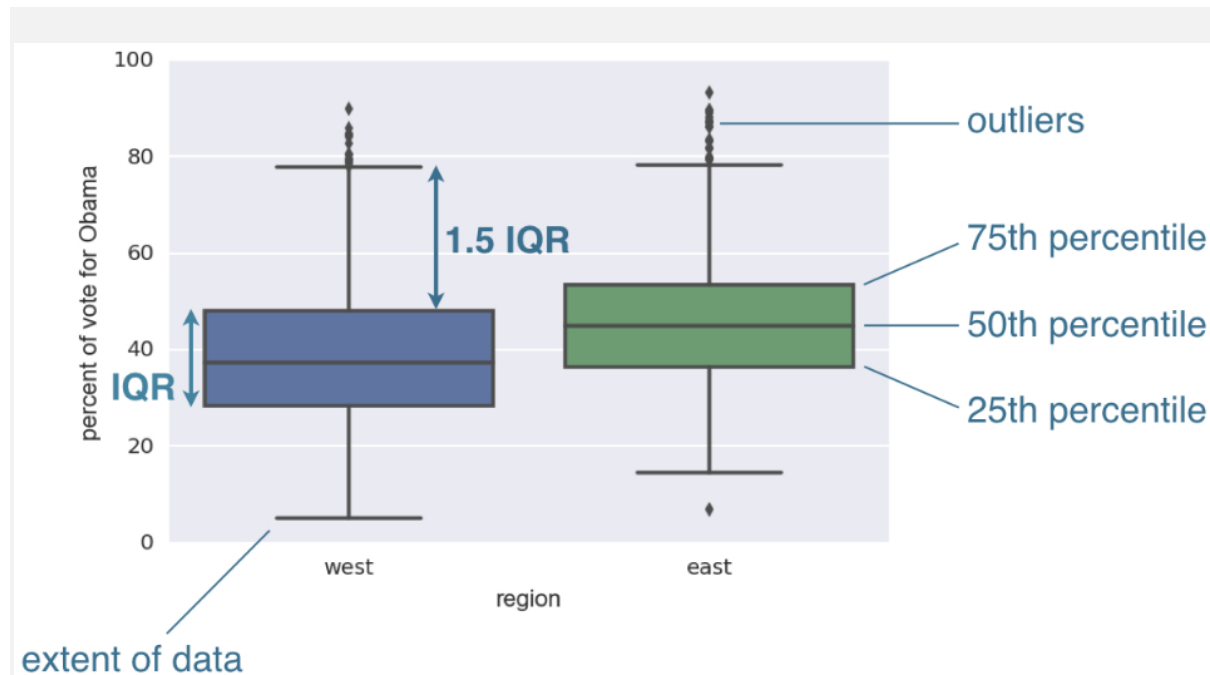
The mean of all measurements gives an indication of the typical magnitude of a measurement. It is computed using `np.mean()`.

```
# Compute the mean: 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')

<script.py> output:
I. versicolor: 4.26 cm
```

Percentiles, outliers, and box plots



Computing percentiles

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

```
# 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)

<script.py> output:
    [3.3    4.    4.35   4.6    4.9775]
```

Comparing percentiles to ECDF

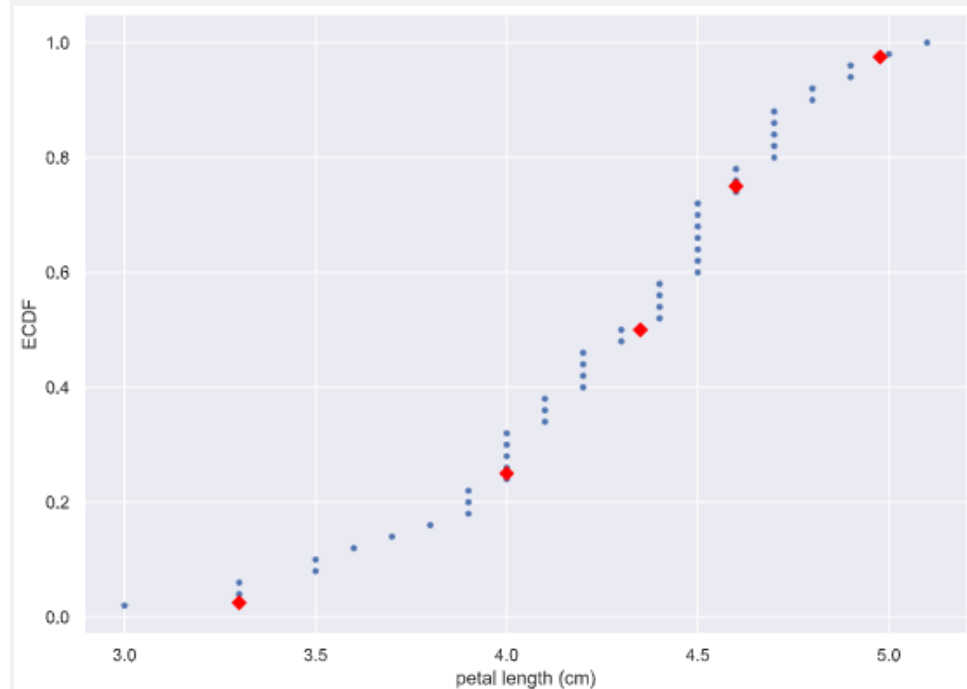
To see how the percentiles relate to the ECDF, you will plot the percentiles of *Iris versicolor* petal lengths you calculated in the last exercise on the ECDF plot you generated in chapter 1. The percentile variables from the previous exercise are available as `ptiles_vers` and `percentiles`.

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 ECDF
_ = plt.plot(x_vers, y_vers, marker='.', color='blue')
_ = 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

The code to produce the box plot is:

```
sns.boxplot(x='east_west', y='dem_share', data=df_all_states)
plt.xlabel('region')
plt.ylabel('percent of vote for Obama')
```

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

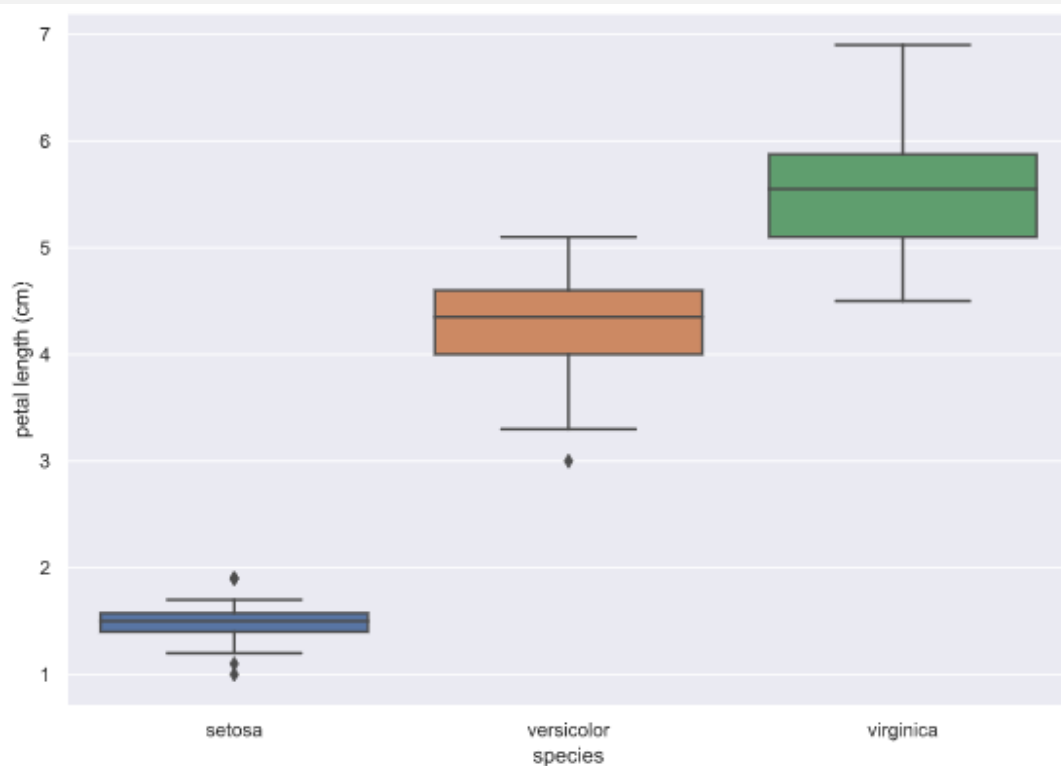
lengths. You have a pandas DataFrame, `df`, which contains the petal length data. Inspect the data frame `df` using `df.head()` to make sure you know what the pertinent columns are.

You can use `sns.boxplot?` or `help(sns.boxplot)` for more details on how to make box plots using seaborn.

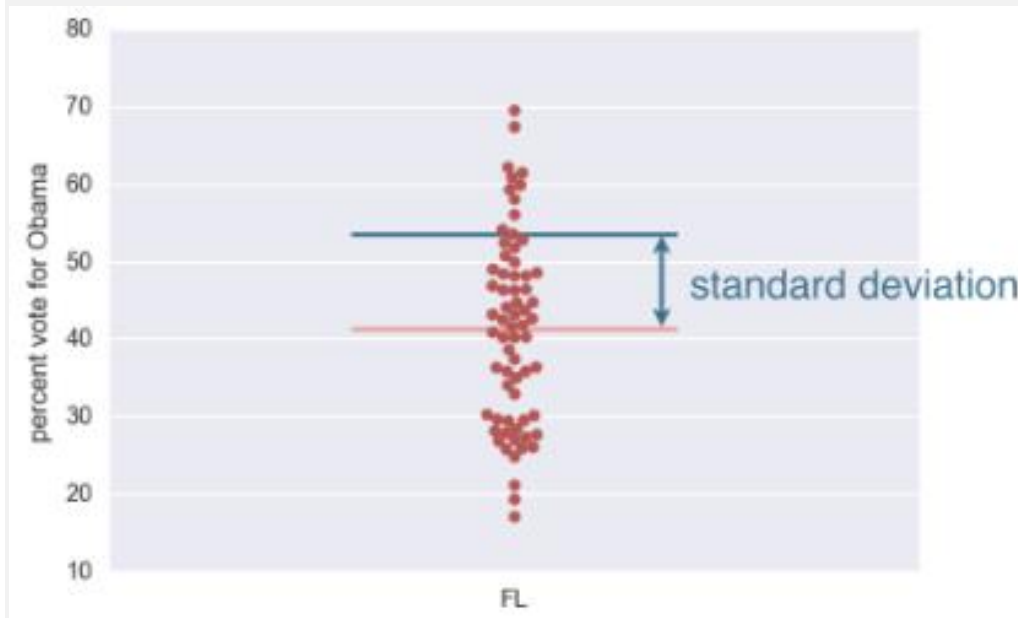
```
# Create box plot with Seaborn's default settings
sns.boxplot('species' , 'petal length (cm)', data=df)

# Label the axes
plt.xlabel('species' )
plt.ylabel('petal length (cm)')

# Show the plot
plt.show()
```



Variance and standard deviation



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 (using equation) compute the variance of the petal length of *Iris versicolor*. We will then use `np.var()` to compute it.

```
# Array of differences to mean: differences
differences = versicolor_petal_length - np.mean(versicolor_petal_length)

# Square the differences: diff_sq
diff_sq = differences**2
```

```
# Compute the mean square difference: variance_explicit
variance_explicit = np.mean(diff_sq)

# Compute the variance using NumPy: variance_np
variance_np = np.var(versicolor_petal_length)

# Print the results
print(variance_explicit, variance_np)

<script.py> output:
0.216400000000000004 0.216400000000000004
```

The standard deviation and the variance

The standard deviation is the square root of the variance. You will see this 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: variance
variance = np.var(versicolor_petal_length)

# Print the square root of the variance
print(np.sqrt(variance))

# Print the standard deviation
print(np.std(versicolor_petal_length))

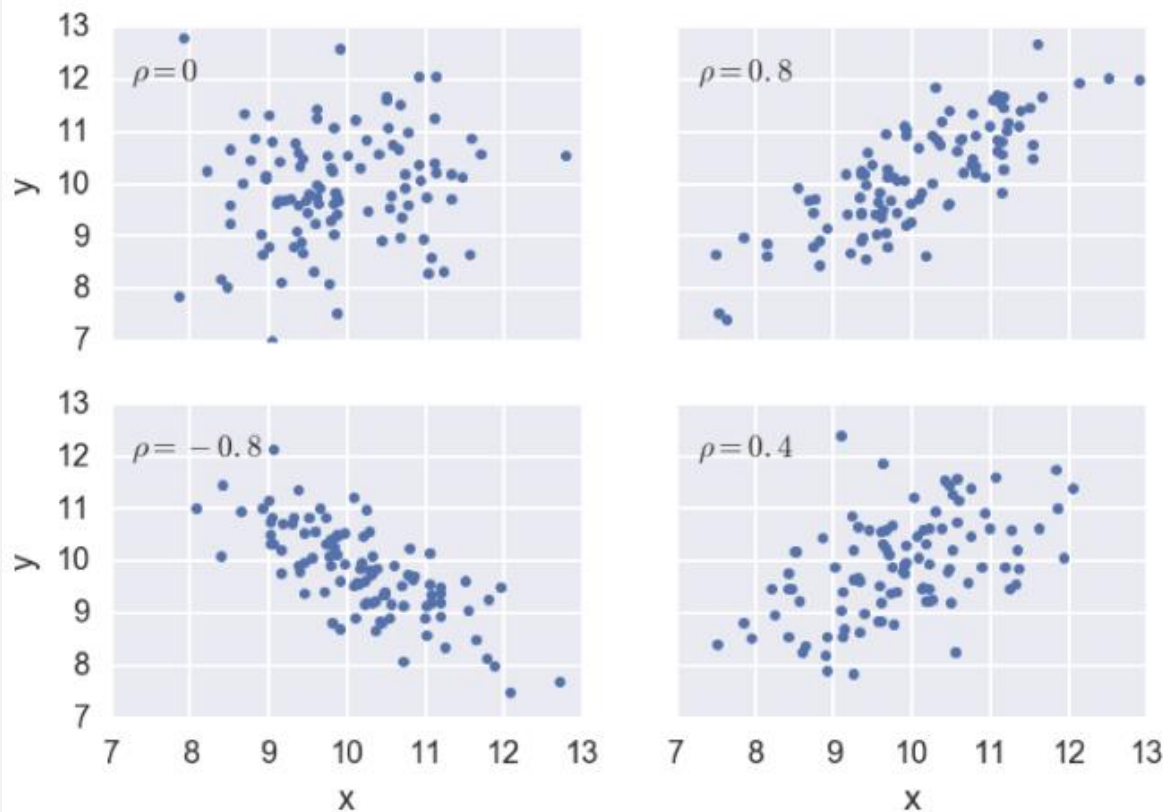
<script.py> output:
0.4651881339845203
0.4651881339845203
```

Observation: They are exactly the same!

Covariance and the Pearson correlation coefficient

Pearson correlation coefficient is calculated as:

$$\rho = \text{Pearson correlation} = \frac{\text{covariance}}{(\text{std of } x)(\text{std of } y)}$$
$$= \frac{\text{variability due to codependence}}{\text{independant variability}}$$



Some examples of Pearson Correlation Coefficient

Scatter plots

The code to produce the scatter plot is:

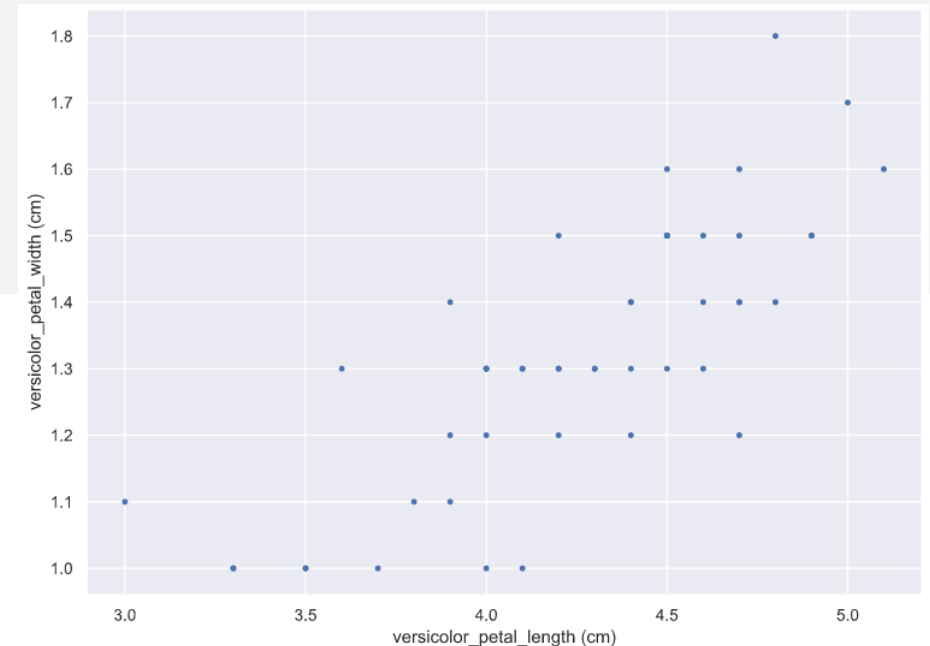
```
plt.plot(total_votes/1000, dem_share, marker='.', linestyle='none')
plt.xlabel('total votes (thousands)')
plt.ylabel('percent of vote for Obama')
```

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.

```
# Make a scatter plot
plt.plot(versicolor_petal_length, versicolor_petal_width, marker='.', linestyle='none')

# Label the axes
plt.xlabel('versicolor_petal_length (cm)')
plt.ylabel('versicolor_petal_width (cm)')

# Show the result
plt.show()
```

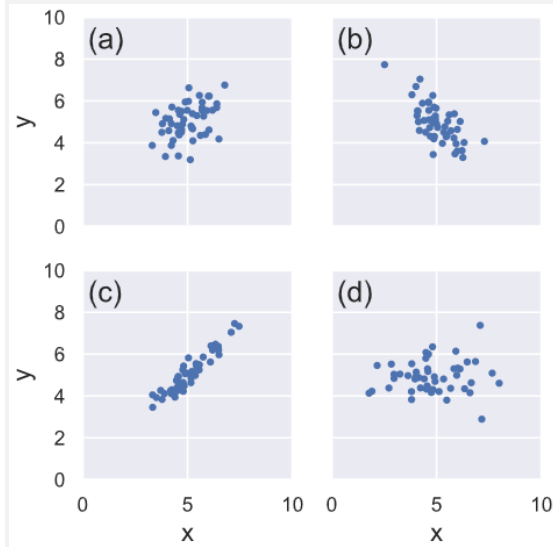


Observation: Some positive correlation.

Longer petals also tend to be wider.

Variance and covariance by looking

Consider four scatter plots of x-y data below, which has the highest variance in the variable x, the highest covariance, and negative covariance, respectively? 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 remember how the *I. versicolor* petal length and width are related, the scatter plot was generated in a previous exercise (Scatter plots).


```
# 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)

<script.py> output:
[[0.22081633 0.07310204]
 [0.07310204 0.03910612]]
0.07310204081632653
```

Computing the Pearson correlation coefficient

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 (because each array is exactly correlated to itself), 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*.

```
def pearson_r(x, y):
    """Compute Pearson correlation coeff between 2 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_length , versicolor_petal_width)

# Print the result
print(r)

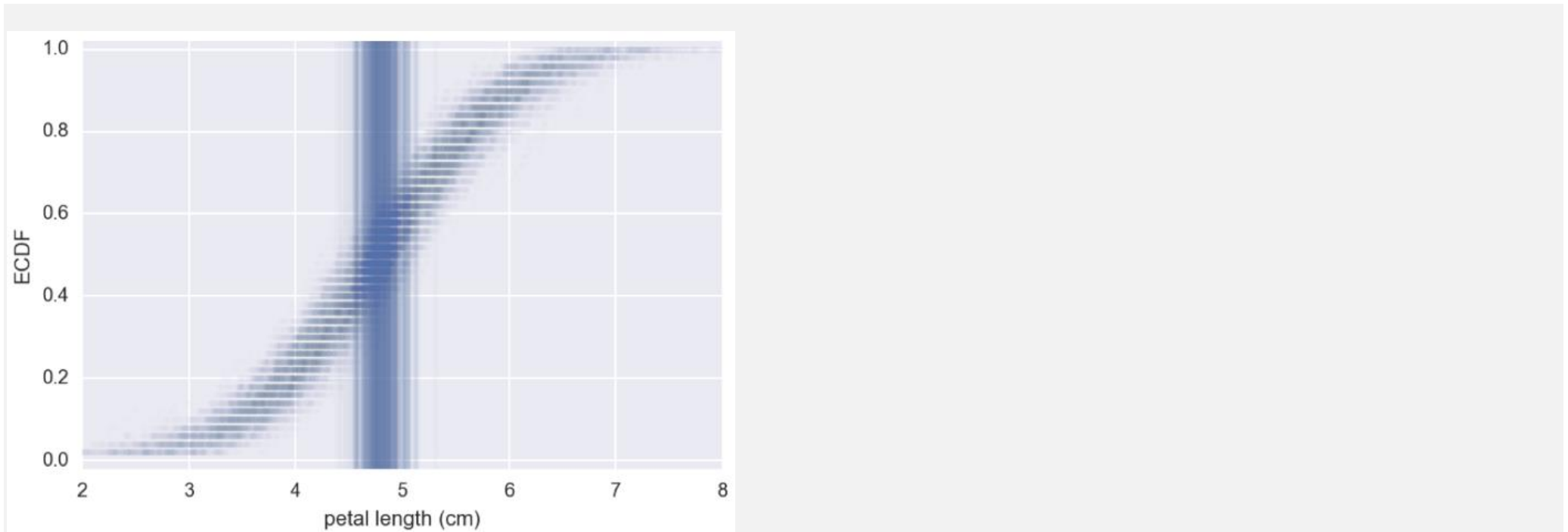
<script.py> output:
0.7866680885228169
```

Observation : If r is nearer to 1, it means the two variables have positive correlation.

Chapter 3: Thinking probabilistically – Discrete variables

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



Repeats of 50 measurements of petal length

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.

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?

These are why we use probabilistic language in statistical inference:

- 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.

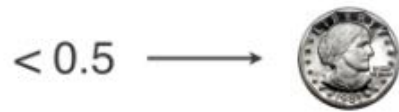
Probabilistic language is in fact very precise. It precisely describes uncertainty.

Random number generators and hacker statistics

The concepts of probability originated from studies of games of chance by Blaise Pascal and others in the 17th century.

Hacker statistics uses simulated repeated measurements to compute probabilities. These simulations are done using the `np.random` module.

`np.random.random()` : draw a number between 0 and 1



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. In each run, the function returns a random number 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.

To generate 4 random numbers, pass 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.

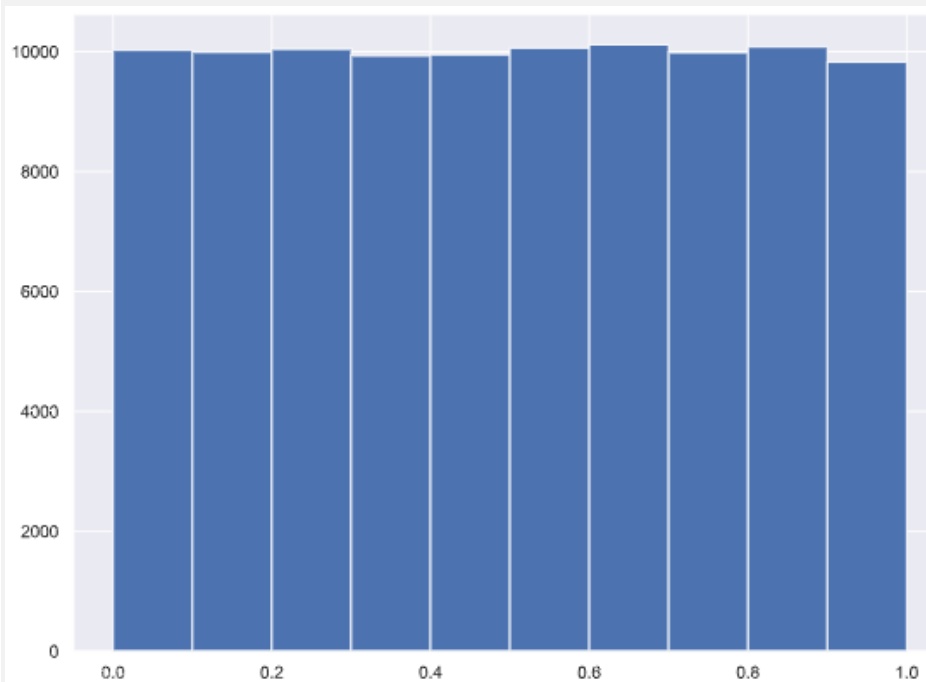
```
# Seed the random number generator
np.random.seed(42)

# 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()
```



Observation: 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 p of landing heads (success) and probability $1-p$ 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.

```
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 random number generator
np.random.seed(42)

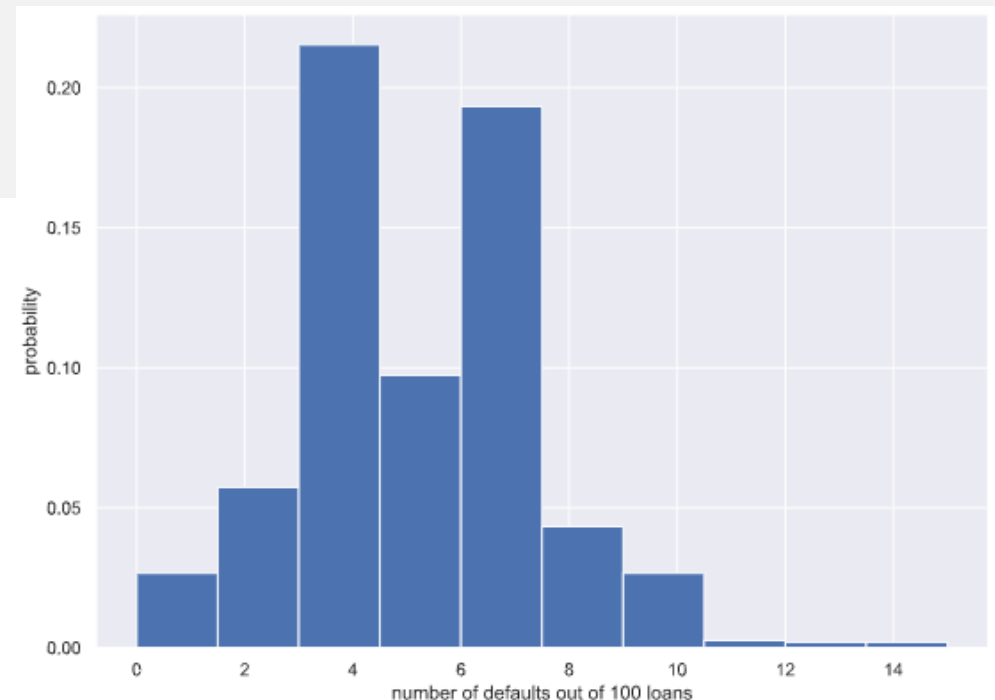
# Initialize the number of defaults: n_defaults
n_defaults = np.empty(1000)

# 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 the axes
_ = plt.hist(n_defaults, normed=True)
_ = plt.xlabel('number of defaults out of 100 loans')
_ = plt.ylabel('probability')

# Show the plot
plt.show()
```

Note: 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 (`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?

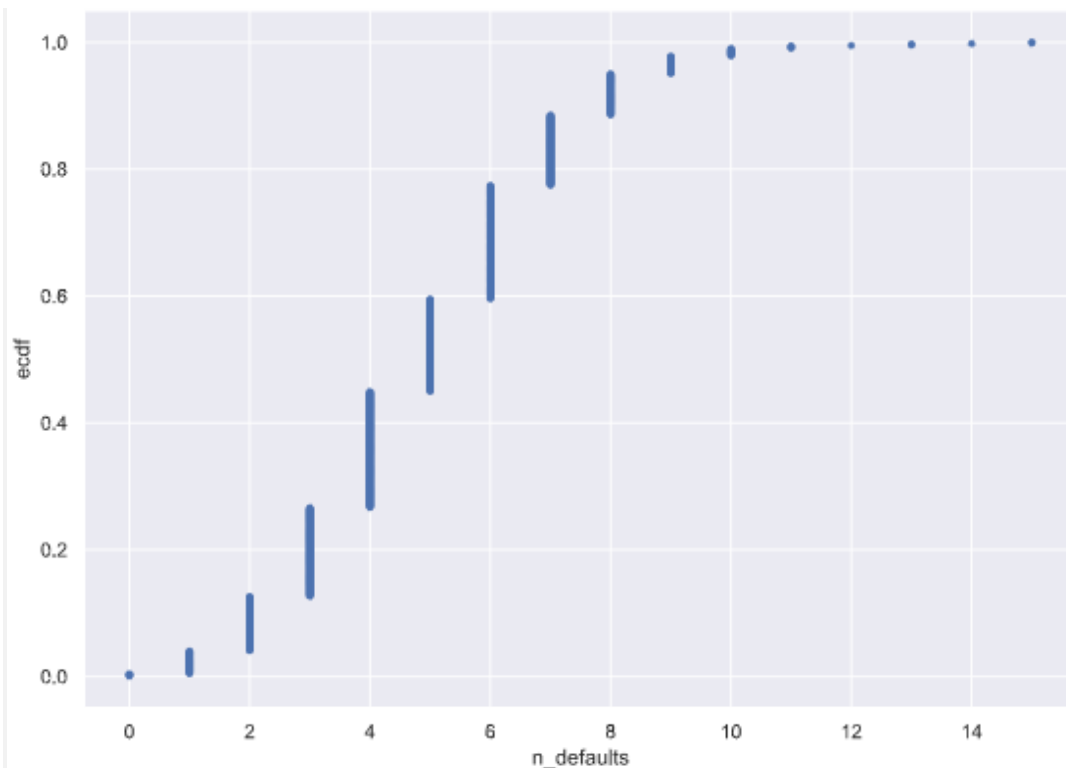
```
# Compute ECDF: x, y
x,y = ecdf(n_defaults)

# Plot the ECDF with labeled axes
plt.plot(x,y, marker='.', linestyle = 'none')
plt.xlabel('n_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))
```









```
script.py> output:  
Probability of losing money = 0.022
```

Observation: Most likely expect 5/100 defaults (mean=5). But we still have about a 2% chance of getting 10 or more defaults out of 100 loans.

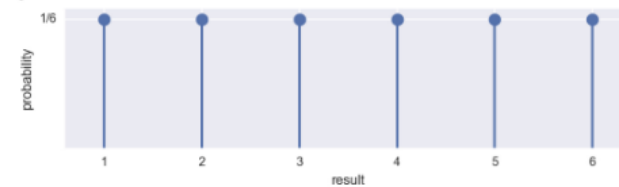
Probability distributions and stories: The Binomial distribution

Discrete Uniform PMF (Probability Mass Function) of a fair dice throw, with probability of $1/6$ for each event.

Tabular

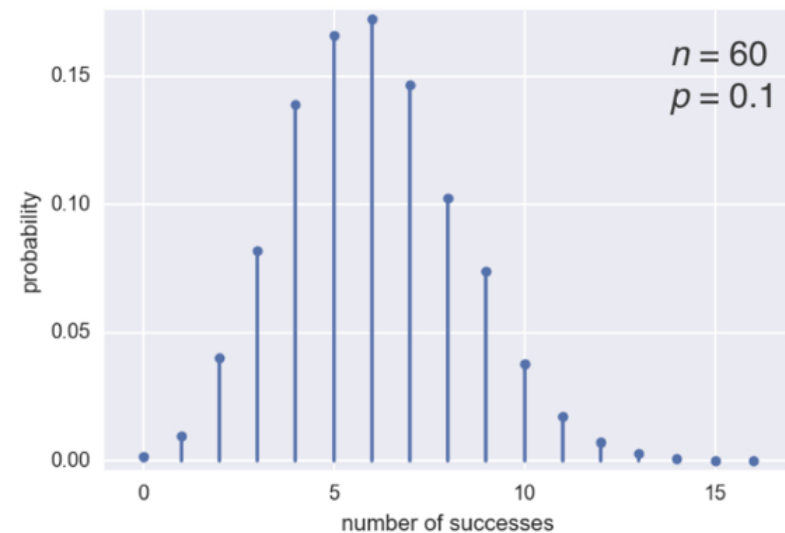
					
$1/6$	$1/6$	$1/6$	$1/6$	$1/6$	$1/6$

Graphical



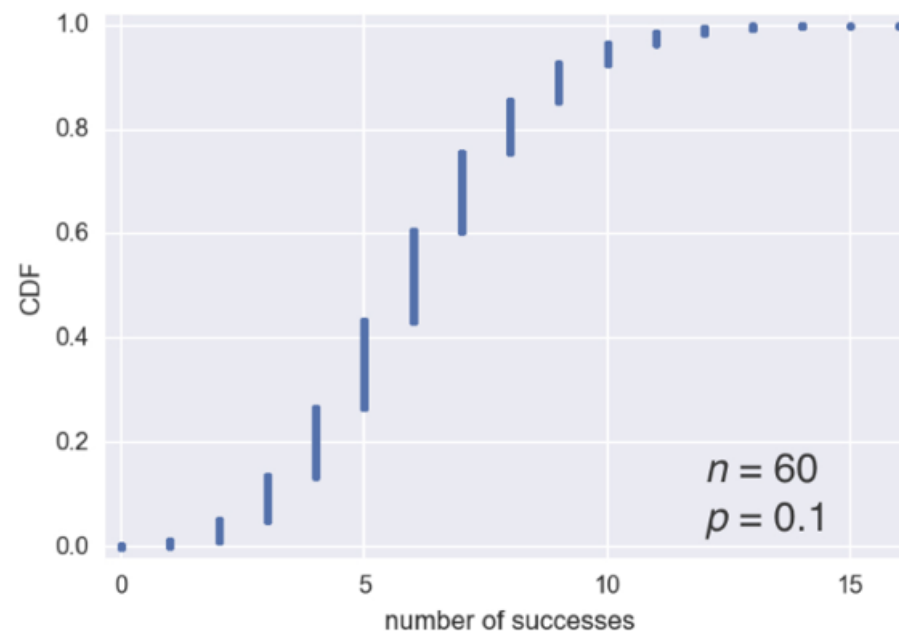
Binomial PMF (Probability Mass Function), with number of trials $n=60$ and probability of success $p=0.1$, repeated 10,000 times.

```
samples = np.random.binomial(60, 0.1, size=10000)  
n = 60  
p = 0.1
```



Binomial CDF (Cumulative Distribution Function), same samples above.

```
import matplotlib.pyplot as plt
import seaborn as sns
sns.set()
x, y = ecdf(samples)
_ = plt.plot(x, y, marker='.', linestyle='none')
plt.margins(0.02)
_ = plt.xlabel('number of successes')
_ = plt.ylabel('CDF')
plt.show()
```



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.

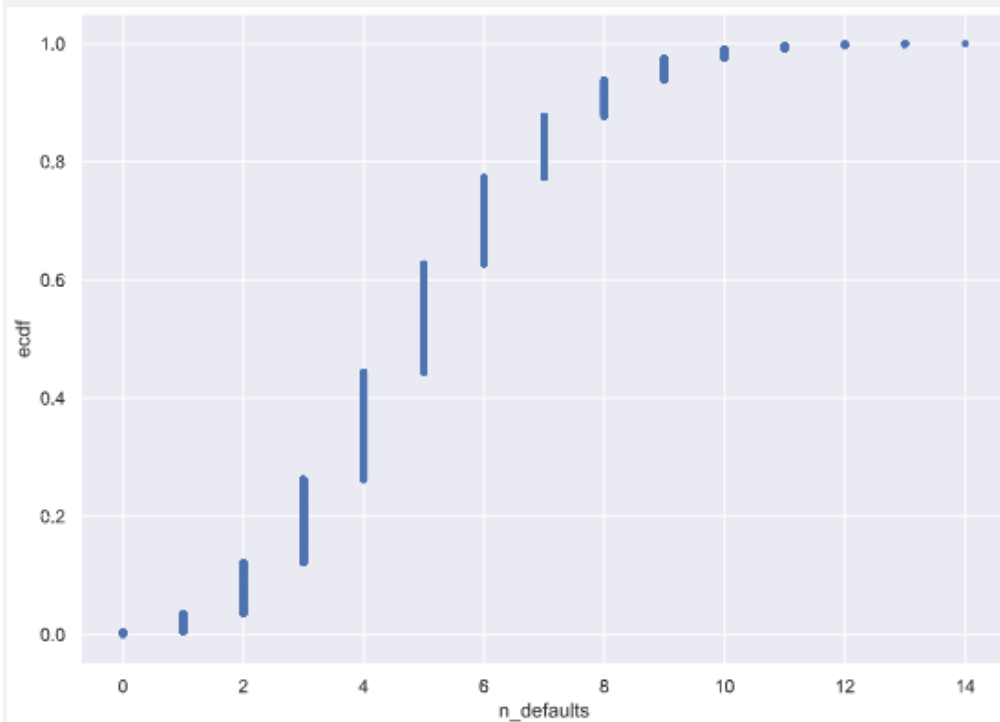
Note: pre-seeded `np.random.seed(42)`

```
# Take 10,000 samples out of the binomial distribution: n_defaults
n_defaults = np.random.binomial(n = 100 , p = 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('n_defaults')
plt.ylabel('ecdf')

# Show the plot
plt.show()
```



Observation: Using built-in algorithms to directly sample out of the distribution is *much* faster than custom-built algorithms.

Plotting the Binomial PMF

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.

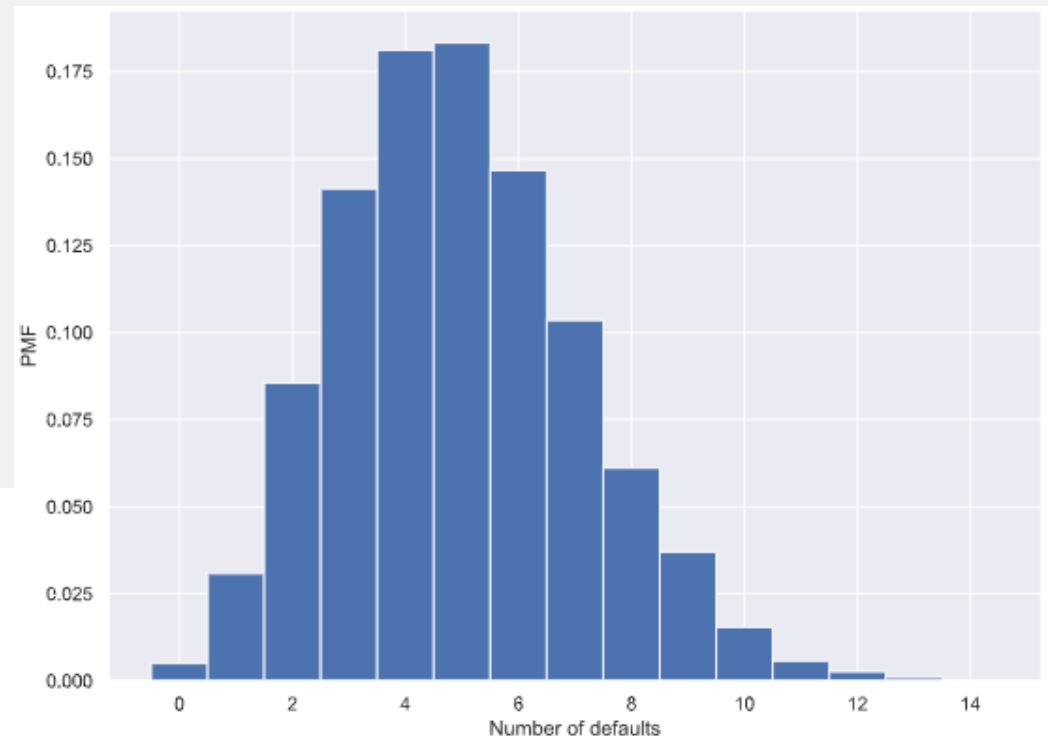
You have already sampled out of the Binomial distribution in the previous exercise, and the resulting samples are in the NumPy array `n_defaults`.

```
# 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()
```



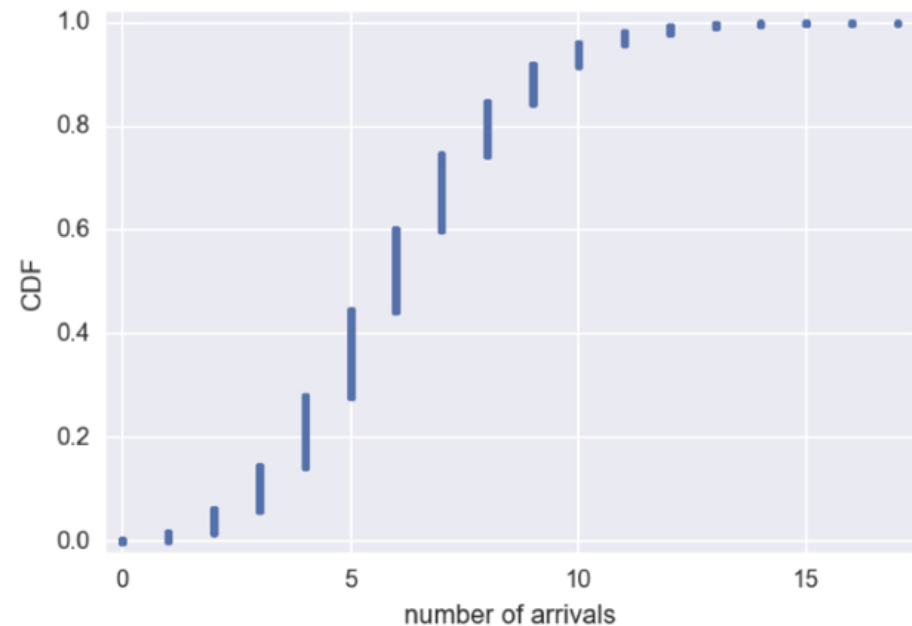
Poisson processes and the Poisson distribution

The timing of the next event is completely independent of when the previous event happened. For example:

- Natural births in a given hospital
- Hits on a website during a given hour
- Meteor strikes
- Aviation incidents

Poisson CDF (Cumulative Distribution Function) for number of hits arrived at a website per hour, with average $\lambda=6$.

```
samples = np.random.poisson(6, size=10000)
x, y = ecdf(samples)
_ = plt.plot(x, y, marker='.', linestyle='none')
plt.margins(0.02)
_ = plt.xlabel('number of successes')
_ = plt.ylabel('CDF')
plt.show()
```



Relationship between Binomial and Poisson distributions

For specific rare events and large samples ($n > 50$), Binomial distribution can approximate to Poisson distribution. 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 in the Poisson distribution, where we get on average 6 hits on a website per hour. So, the Poisson distribution with arrival rate equal to np approximates a Binomial distribution for n Bernoulli trials with probability p of success (with n large and p 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 n and p such that $np=10$.

```
# Draw 10,000 samples out of Poisson distribution: samples_poisson
samples_poisson = np.random.poisson(10, 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], 10000)
    # Print results
    print('n =', n[i], 'Binom:', np.mean(samples_binomial),
          np.std(samples_binomial))
```

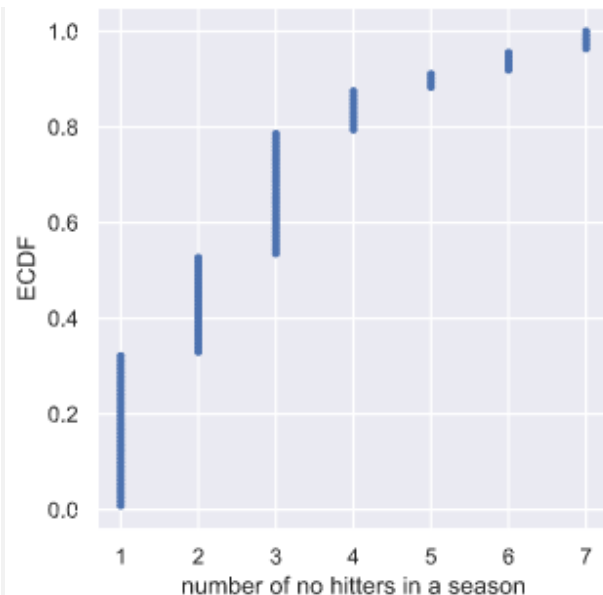


```
<script.py> output:
Poisson:      10.0186 3.144813832327758
n = 20 Binom: 9.9637 2.2163443572694206
n = 100 Binom: 9.9947 3.0135812433050484
n = 1000 Binom: 9.9985 3.139378561116833
```

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 below:



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.

Observation: When we have specific 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.

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 10,000 samples out of Poisson distribution: n_nohitters
n_nohitters = np.random.poisson(251/115, 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 / len(n_nohitters)

# Print the result
print('Probability of seven or more no-hitters:', p_large)

<script.py> output:
    Probability of seven or more no-hitters: 0.0067
```

Observation: 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

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.

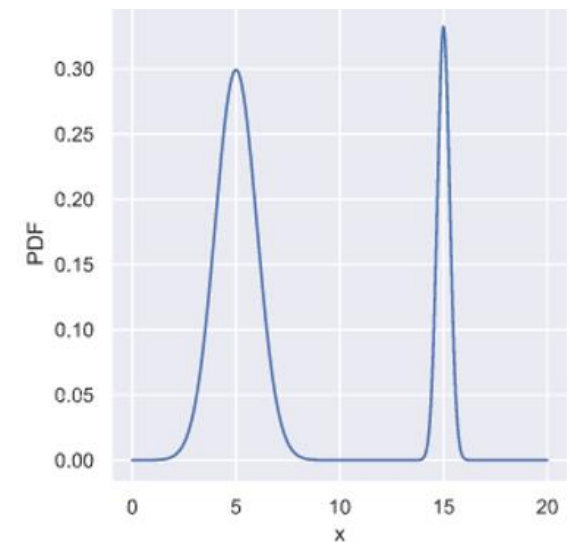
Probability density functions (PDF)

PDF is a continuous analog to the PMF. PDF is a mathematical description of the relative likelihood of observing a value of a continuous variable.

Interpreting PDFs

Consider the PDF shown below, which of the following is true?

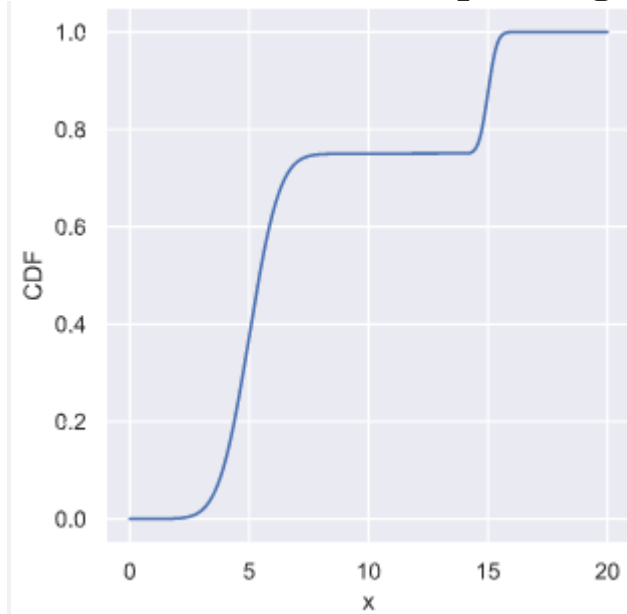
- ☐ x is more likely to be **less than 10** than to be greater than 10.
- ☐ x is more likely to be **greater than 10** than to be less than 10.
- ☐ We cannot tell from the PDF if x is more likely to be greater than or less than 10.
- ☐ This is not a valid PDF because it has two peaks.



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

Below is the CDF corresponding to the PDF you considered in the last exercise.

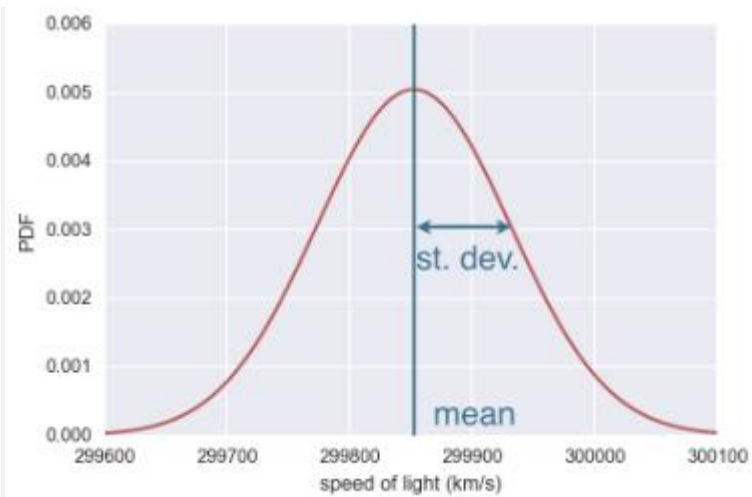


Using the CDF, what is the probability that x is greater than 10?

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.

Introduction to the Normal distribution

Normal distribution of a continuous variable has its PDF having a single symmetric peak and long tapering tails on both ends.



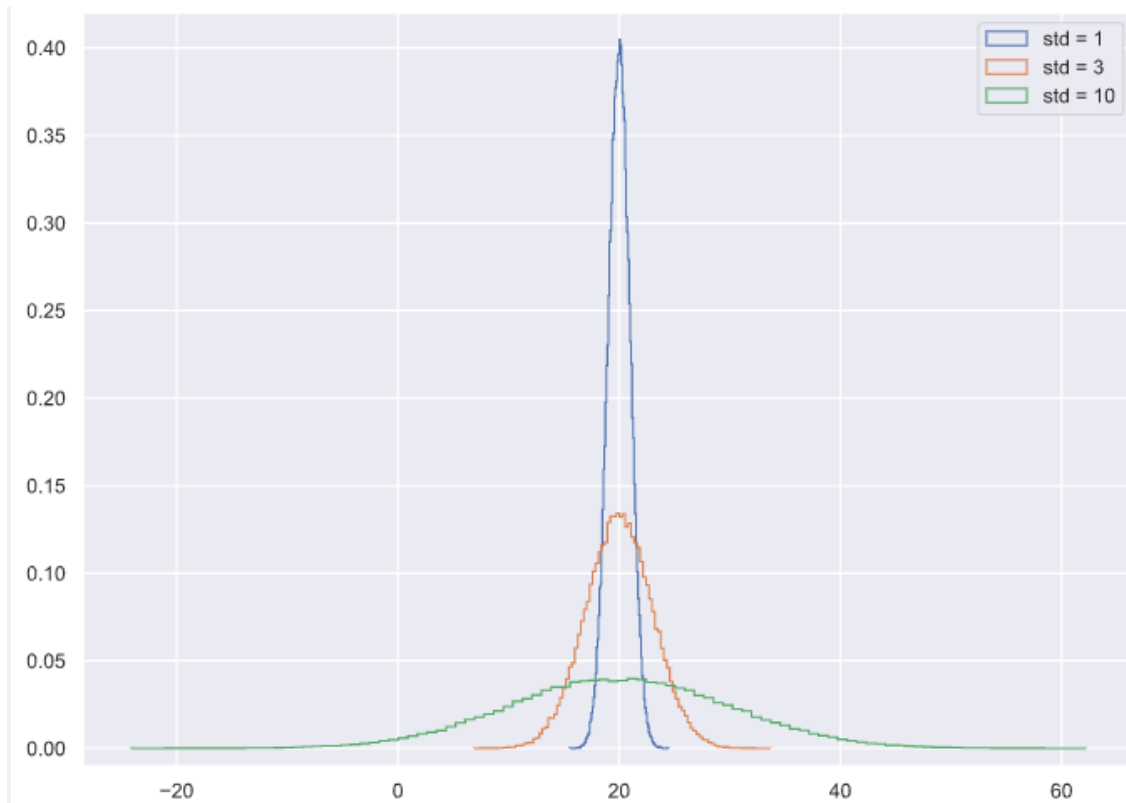
The Normal PDF

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 100000 samples from Normal distribution with stds of interest: samples_std1, samples_std3,
samples_std10
samples_std1 = np.random.normal(20,1, 100000)
samples_std3 = np.random.normal(20,3, 100000)
samples_std10 = np.random.normal(20,10, 100000)

# Make histograms
plt.hist(samples_std1, normed=True , histtype='step', bins=100)
plt.hist(samples_std3, normed=True , histtype='step', bins=100)
plt.hist(samples_std10, normed=True , histtype='step', bins=100)

# Make a legend, set limits and show plot
_ = plt.legend(('std = 1', 'std = 3', 'std = 10'))
plt.ylim(-0.01, 0.42)
plt.show()
```



Observation: Different standard deviations result in PDFs of different widths. The peaks are all centered at the mean of 20.

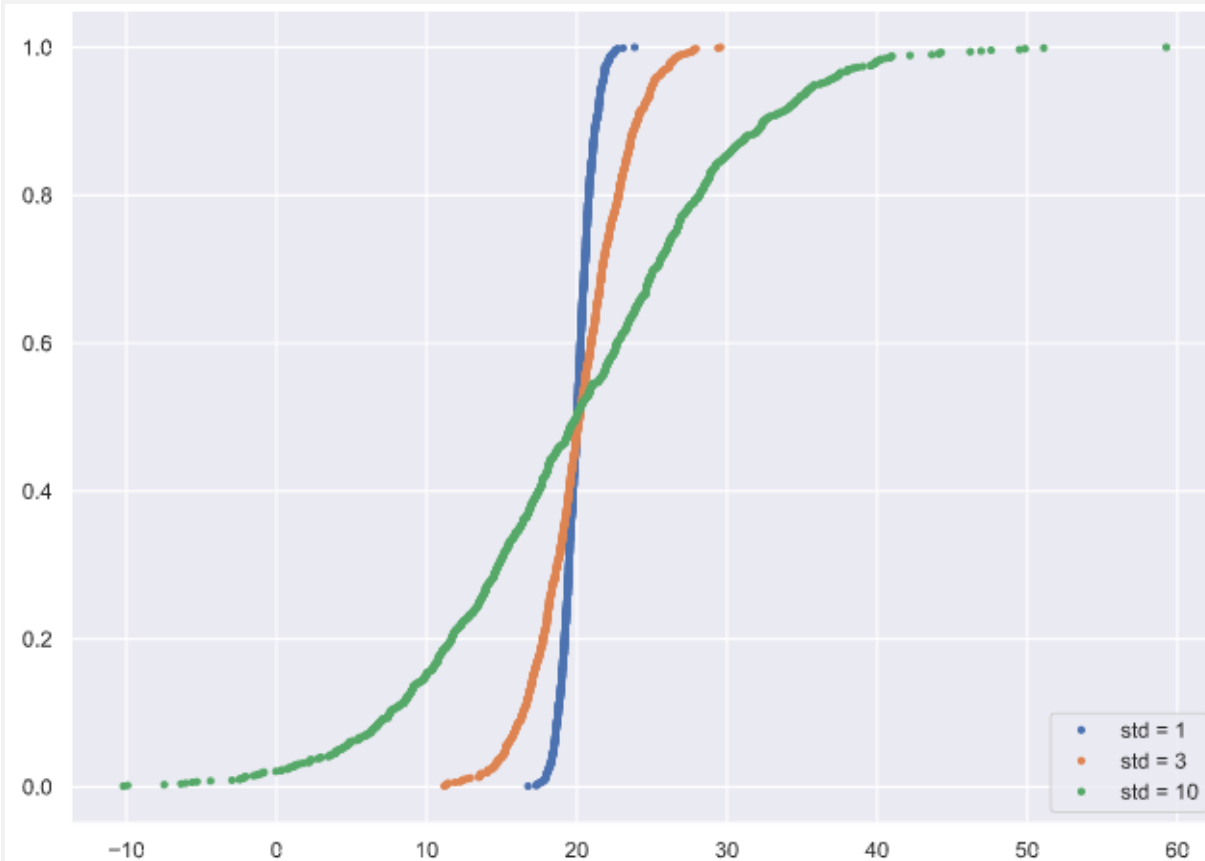
The Normal CDF

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 (`samples_std1`, `samples_std3`, and `samples_std10`), generate and plot the CDFs.

```
# Generate CDFs
x_std1, y_std1 = ecdf(samples_std1)
x_std3, y_std3 = ecdf(samples_std3)
x_std10, y_std10 = ecdf(samples_std10)
```

```
# 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')

# Make a legend and show the plot
plt.legend(('std = 1', 'std = 3', 'std = 10'), loc='lower right')
plt.show()
```



Observation: 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.

The Normal distribution: Properties and warnings

[Johann Carl Friedrich Gauss](#) was the German mathematician and physicist who invented the Gaussian distribution (ie, normal distribution).

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?

Answer: mean=3, standard_deviation=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](#) 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?

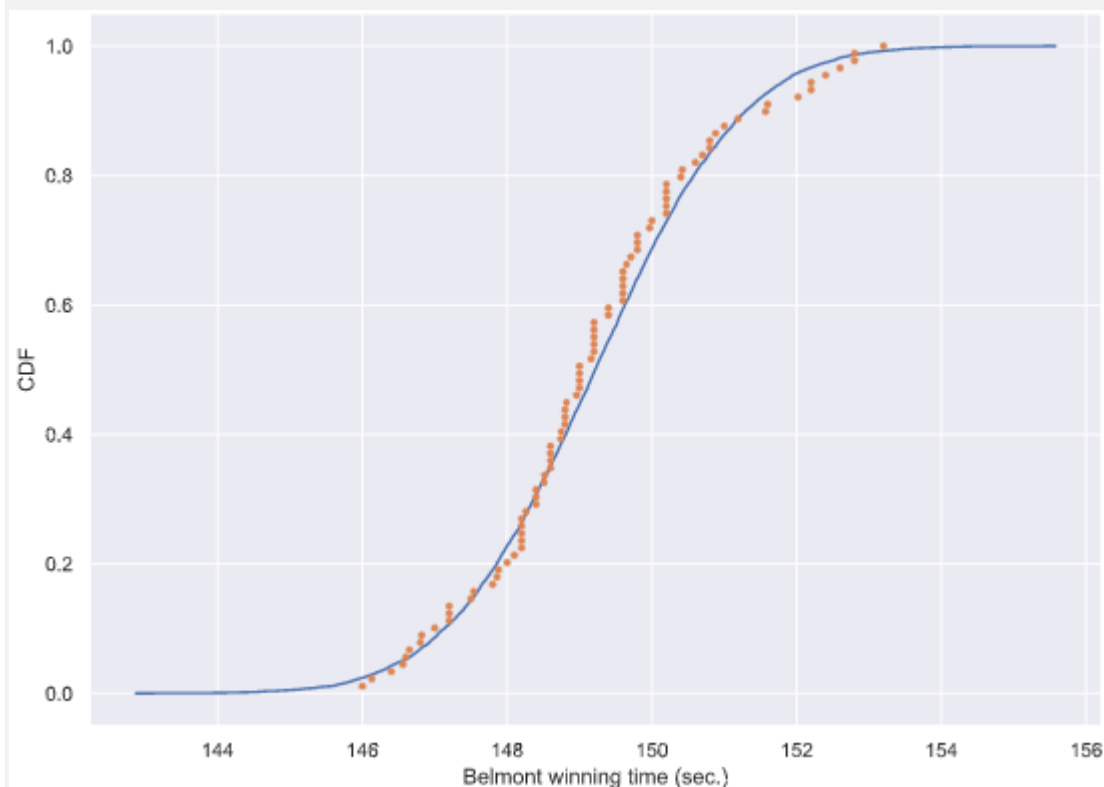
```
# Compute mean and standard deviation: mu, sigma
mu = np.mean(belmont_no_outliers)
sigma = np.std(belmont_no_outliers)

# Sample out of a normal distribution with this mu and sigma: samples
samples = np.random.normal(mu, sigma, 10000)
```



```
# Get the CDF of the samples and of the data
x_theor, y_theor = ecdf(samples)
x, y = ecdf(belmont_no_outliers)

# 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()
```



Observation: 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 a million samples out of the Normal distribution: samples
samples = np.random.normal(mu, sigma, 1000000)

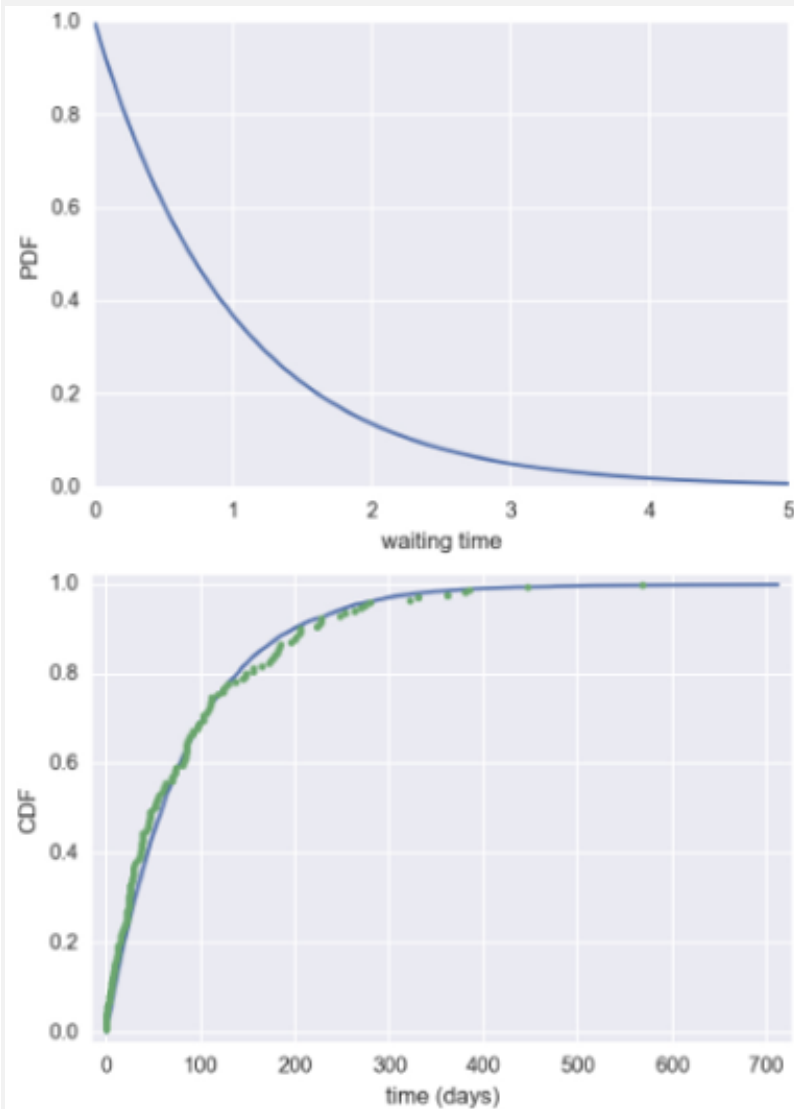
# Compute the fraction that are faster than 144 seconds: prob
prob = np.sum(samples<=144) / len(samples)

# Print the result
print('Probability of besting Secretariat:', prob)<script.py> output:
    Probability of besting Secretariat: 0.000635
```

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

The Exponential distribution

The waiting time between arrivals of a Poisson process is Exponentially distributed. The Exponential PDF and CDF curves are:



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?

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, and Secretariat is *rare*!

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.

```
def successive_poisson(tau1, tau2, size=1):  
    """Compute time for arrival of 2 successive Poisson events."""  
    # 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

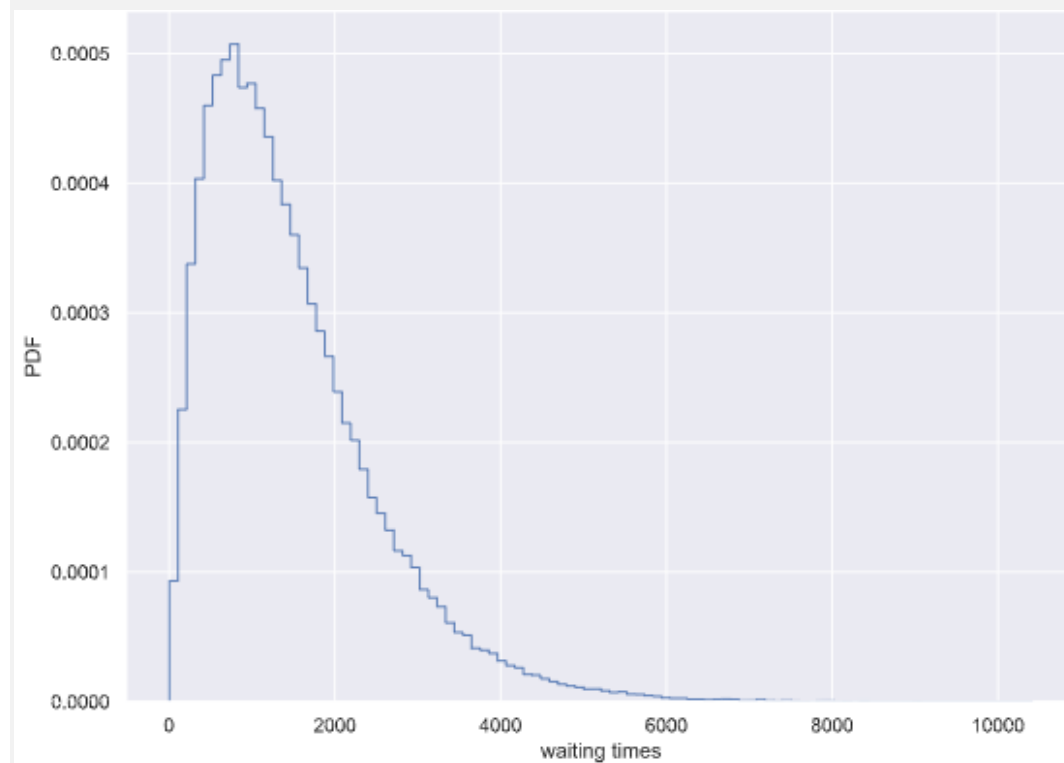
Now, you'll use the above sampling function to compute the waiting time to observe a no-hitter and hitting of the cycle. The mean waiting time for a no-hitter is 764 games, and the mean waiting time for hitting the cycle is 715 games.

```
# Draw samples of waiting times: waiting_times
waiting_times = successive_poisson(764, 715, 100000)

# Make the histogram
plt.hist(waiting_times, bins=100, normed=True, histtype='step')

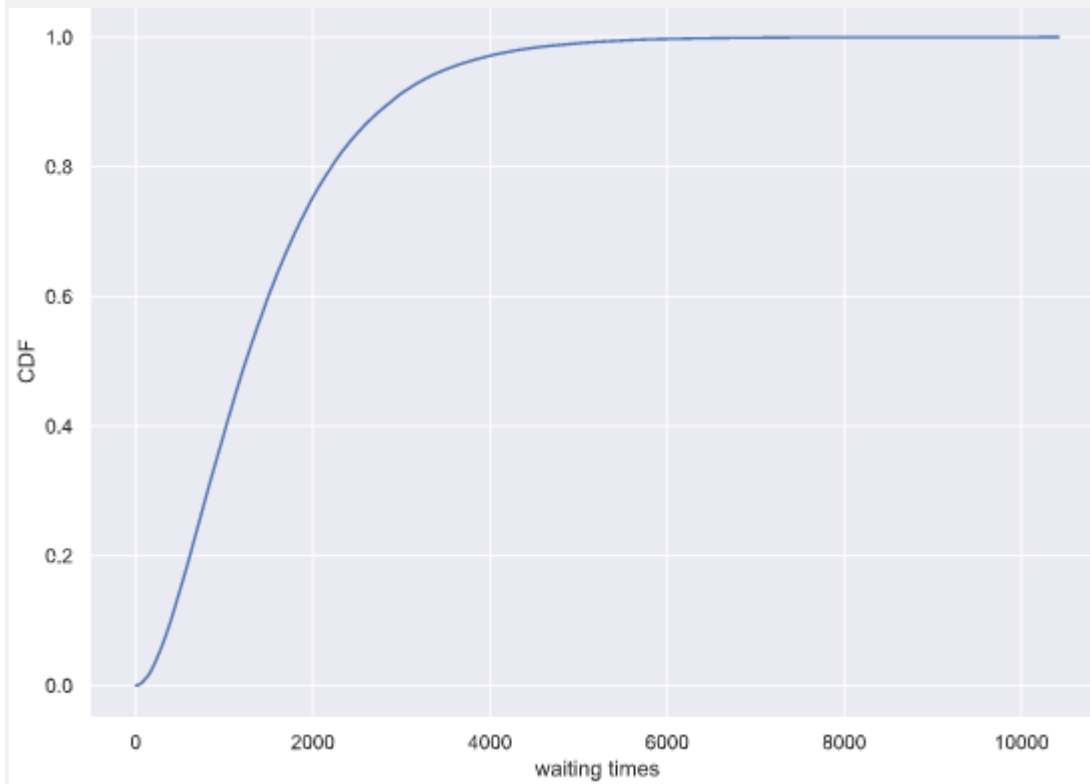
# Label axes
plt.xlabel('waiting times')
plt.ylabel('PDF')

# Show the plot
plt.show()
```



Observation: 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.

```
x,y = ecdf(waiting_times)
plt.plot(x,y)
plt.xlabel('waiting times')
plt.ylabel('CDF')
plt.show()
```



Summary (Part 1)

You can now do the following:

- Graphical exploratory data analysis (EDA), Quantative EDA
- Construct (beautiful) instructive plots, including histogram, swarmplot, Empirical Cumulative Distribution Functions (ECDF), Box Plots, Scatter Plots
- Compute informative summary statistics, Covariance, Pearson correlation coefficient
- Code with hacker statistics using `np.random`
- Bernoulli trials, Binomial distribution, Poisson distribution
- Think probabilistically—discrete variables, continuous variables
- Probability mass function, probability density function, cumulative distribution function (CDF)
- Normal distribution, Normal CDF
- Exponential distribution

Next sequel (Part 2)

In the next tutorial [Statistical Thinking in Python \(Part 2\)](#), these topics will be covered...

- Estimate parameter values
- Perform linear regressions
- Compute confidence intervals
- Perform hypothesis tests including A/B testing