Statistical Thinking in Python (Part 2)

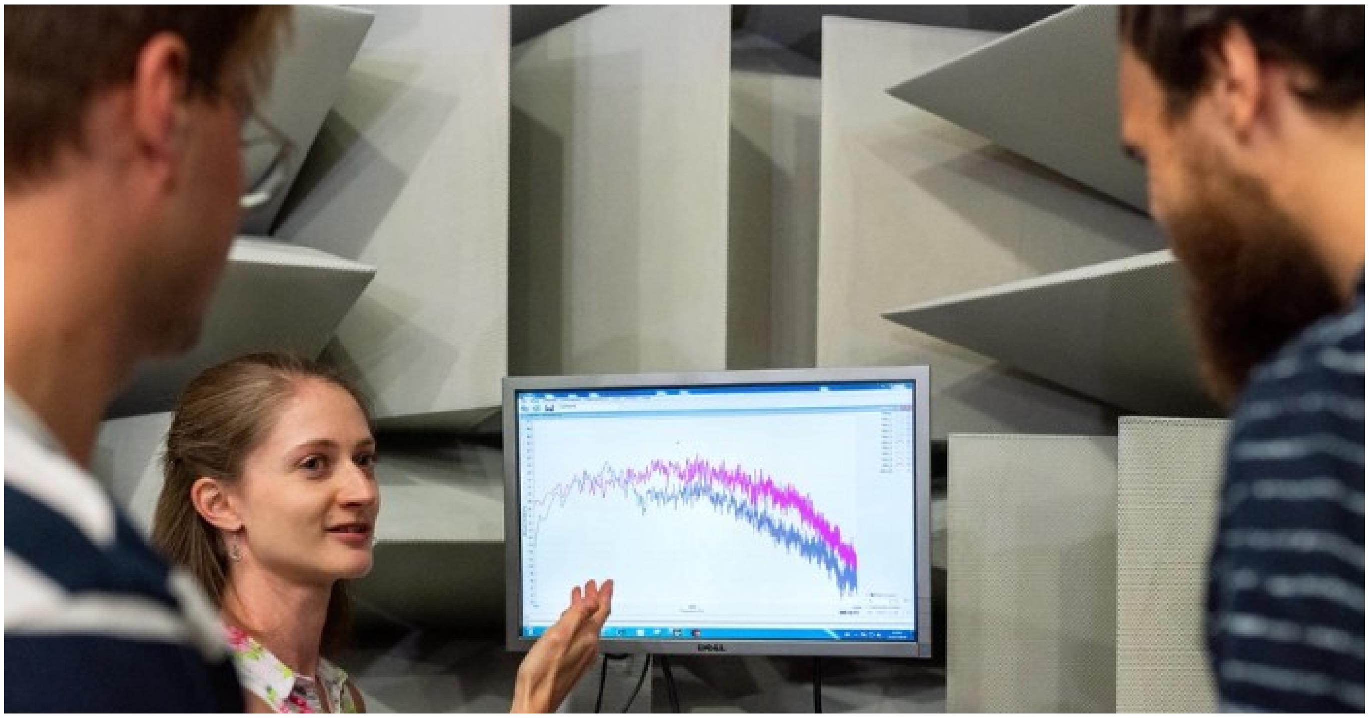
Continue to speak the statistical language of your data

Black Raven (James Ng)
Jun 27 · 43 min read ★

Previous tutorial: Statistical Thinking in Python (Part 1)

This is a tutorial to share what I have learnt in Statistical Thinking in Python (Part 2), capturing the learning objectives as well as my personal notes. The course is taught by Justin Bois from DataCamp, and it includes 5 chapters:

Chapter 1. Parameter estimation by optimization
Chapter 2. Bootstrap confidence intervals
Chapter 3. Introduction to hypothesis testing
Chapter 4. Hypothesis test examples
Chapter 5. Putting it all together: a case study



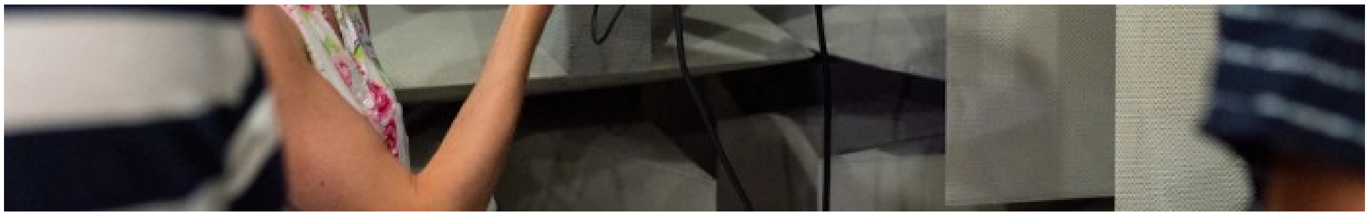


Photo by ThisisEngineering RAEng on Unsplash

After completing Statistical Thinking in Python (Part 1), you have the probabilistic mindset and foundational hacker stats skills to dive into data sets and extract useful information from them. In this course, you will do just that, expanding and honing your hacker stats toolbox to perform the two key tasks in statistical inference, parameter estimation and hypothesis testing. You will work with real data sets as you learn, culminating with analysis of measurements of the beaks of the Darwin's famous finches. You will emerge from this course with new knowledge and lots of practice under your belt, ready to attack your own inference problems out in the world.

. . .

Chapter 1. Parameter estimation by optimization

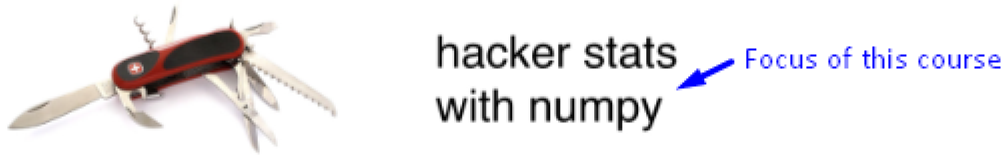
When doing statistical inference, we speak the language of probability. A probability distribution that describes your data has parameters. So, a major goal of statistical inference is to estimate the values of these parameters, which allows us to concisely and unambiguously describe our data and draw conclusions from it. In this chapter, you will learn how to find the optimal parameters, those that best describe your data.

Optimal parameters

Packages to do statistical inference:

scipy.stats

statsmodels



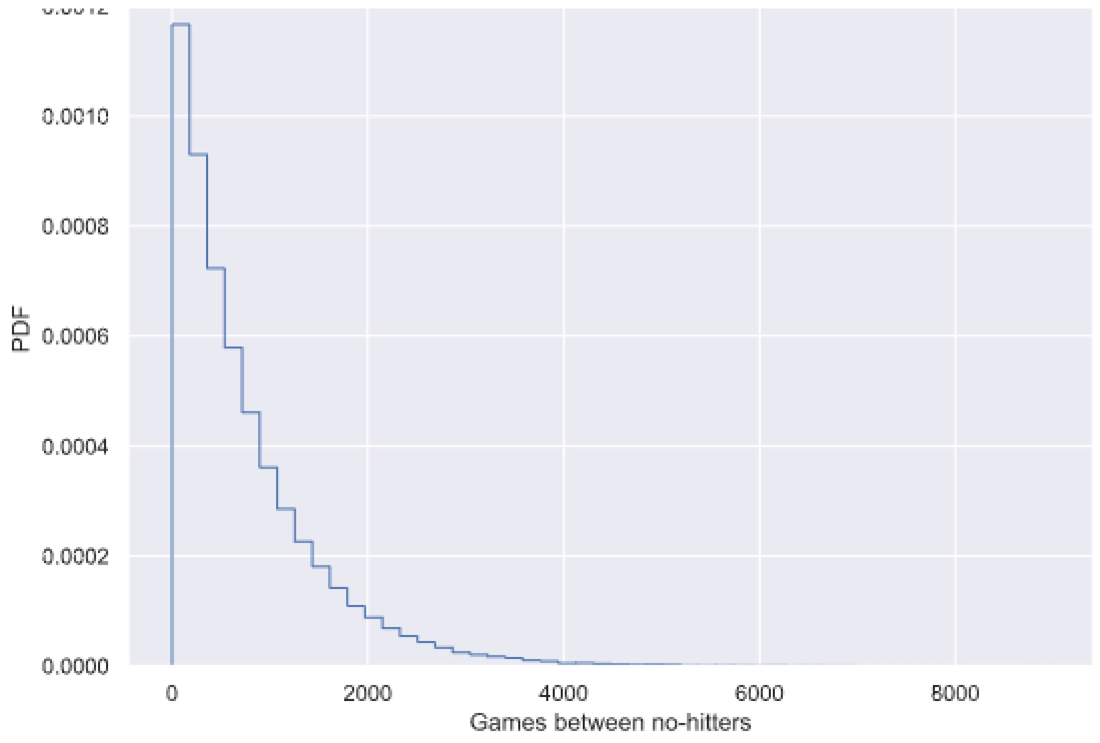
How often do we get no-hitters?

The number of games played between each no-hitter in the modern era (1901–2015) of Major League Baseball is stored in the array `nohitter_times`.

If you assume that no-hitters are described as a Poisson process, then the time between no-hitters is Exponentially distributed. As you have seen, the Exponential distribution has a single parameter, which we will call **τ**, the typical interval time. The value of the parameter **τ** that makes the exponential distribution best match the data is the mean interval time (where time is in units of number of games) between no-hitters.

Compute the value of this parameter from the data. Then, use `np.random.exponential()` to "repeat" the history of Major League Baseball by drawing inter-no-hitter times from an exponential distribution with the **τ** you found and plot the histogram as an approximation to the PDF.

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

# Compute mean no-hitter time: tau
tau = np.mean(nohitter_times)

# Draw out of an exponential distribution with parameter tau:
inter_nohitter_time
inter_nohitter_time = np.random.exponential(tau, 100000)

# Plot the PDF and label axes
_ = plt.hist(inter_nohitter_time,
             bins=50, normed=True, histtype='step')
_ = plt.xlabel('Games between no-hitters')
_ = plt.ylabel('PDF')

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



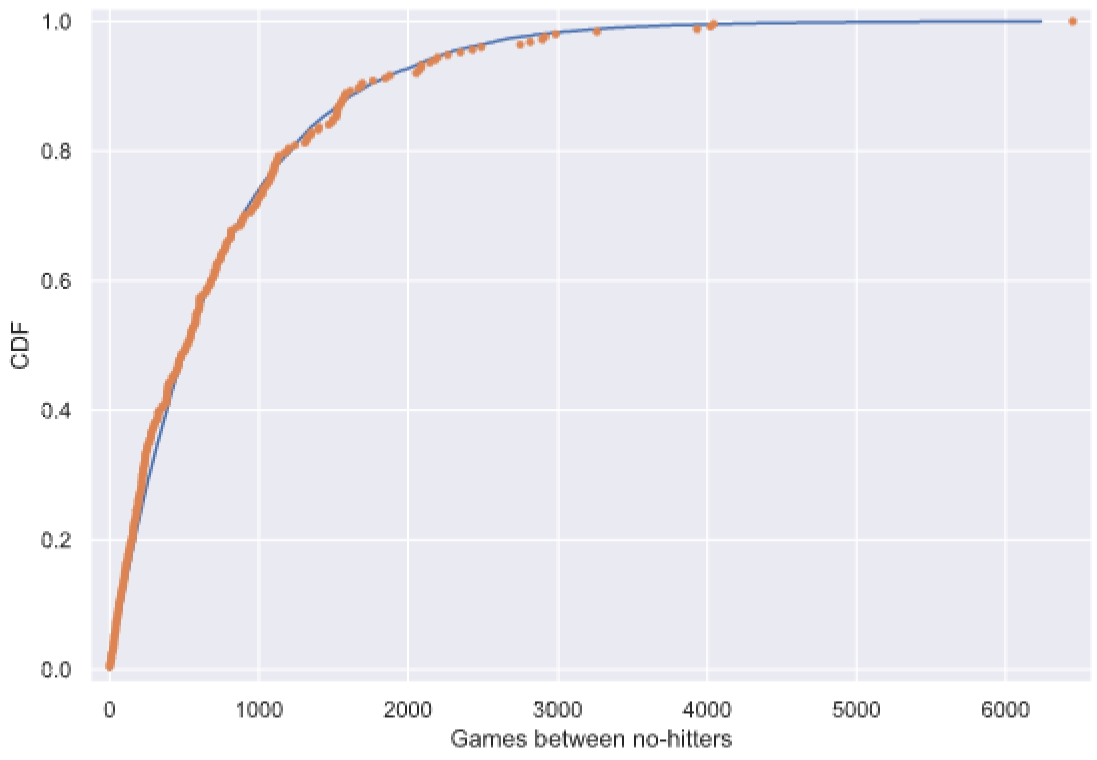
Observation: We see the typical shape of the Exponential distribution, going from a maximum at 0 and decaying to the right.

Do the data follow our story?

You have modeled no-hitters using an Exponential distribution. Create an ECDF of the real data. Overlay the theoretical CDF with the ECDF from the data. This helps you to verify that the Exponential distribution describes the observed data.

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

# Create an ECDF from real data: x, y
x, y = ecdf(nohitter_times)

# Create a CDF from theoretical samples: x_theor, y_theor
x_theor, y_theor = ecdf(inter_nohitter_time)

# Overlay the plots
plt.plot(x_theor, y_theor)
plt.plot(x, y, marker='.', linestyle='none')
```

```
# Margins and axis labels
plt.margins(0.02)
plt.xlabel('Games between no-hitters')
plt.ylabel('CDF')

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

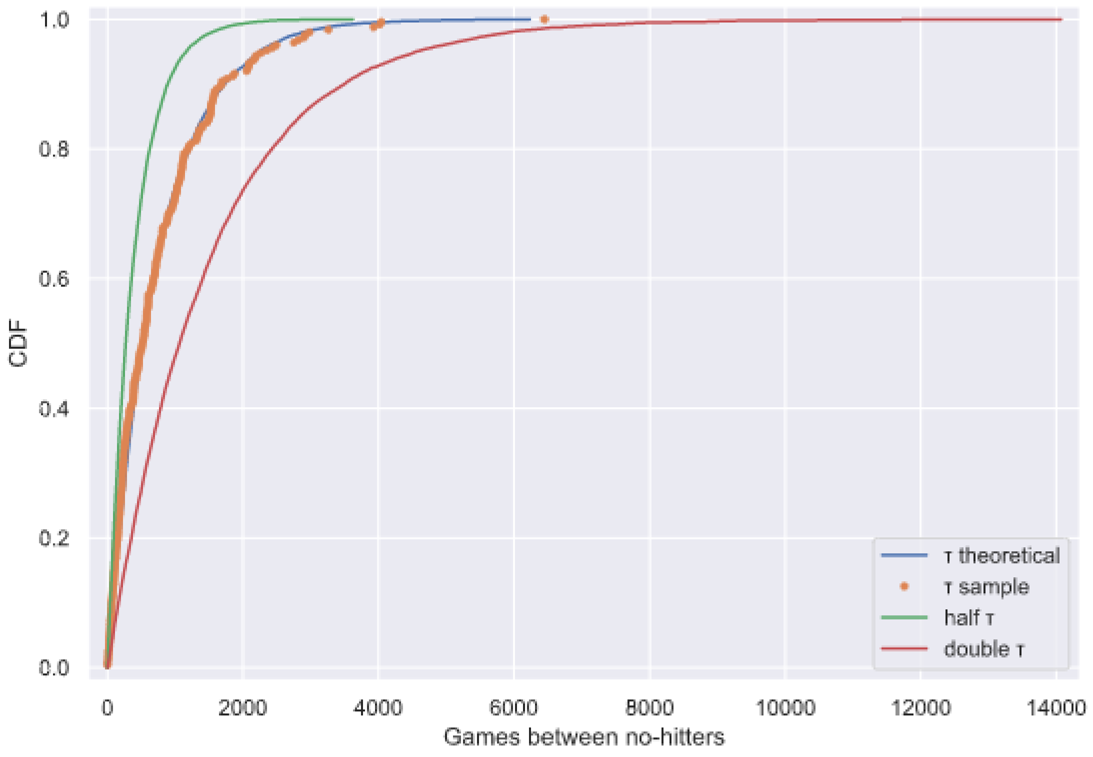


Observation: It looks like no-hitters in the modern era of Major League Baseball are Exponentially distributed. Based on the story of the Exponential distribution, this suggests that they are a random process; when a no-hitter will happen is independent of when the last no-hitter was.

How is this parameter optimal?

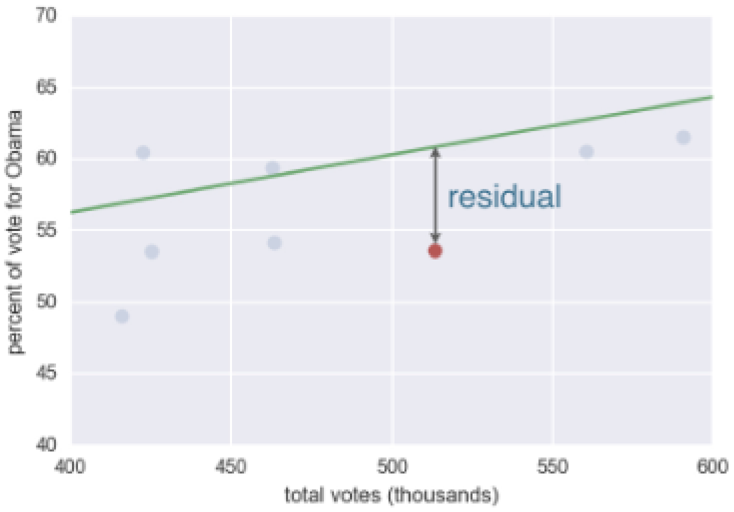
Now sample out of an exponential distribution with **τ** being twice as large as the optimal **τ**. Do it again for **τ** half as large. Make CDFs of these samples and overlay them with your data. You can see that they do not reproduce the data as well. Thus, the **τ** you computed from the mean inter-no-hitter times is optimal in that it best reproduces the data.

```
# Plot the theoretical CDFs
plt.plot(x_theor, y_theor)
plt.plot(x, y, marker='.', linestyle='none')
plt.margins(0.02)
plt.xlabel('Games between no-hitters')
plt.ylabel('CDF')

# Take samples with half tau: samples_half
samples_half = np.random.exponential(tau/2, 10000)

# Take samples with double tau: samples_double
samples_double = np.random.exponential(tau*2, 10000)

# Generate CDFs from these samples
x_half, y_half = ecdf(samples_half)
x_double, y_double = ecdf(samples_double)

# Plot these CDFs as lines
_ = plt.plot(x_half, y_half)
_ = plt.plot(x_double, y_double)

# Show the plot
plt.legend(['τ theoretical','τ sample','half τ','double τ'], loc=4)
plt.show()
```



Observation: The value of tau given by the mean matches the data best. In this way, **τ** (tau) is an optimal parameter.

Linear regression by least squares

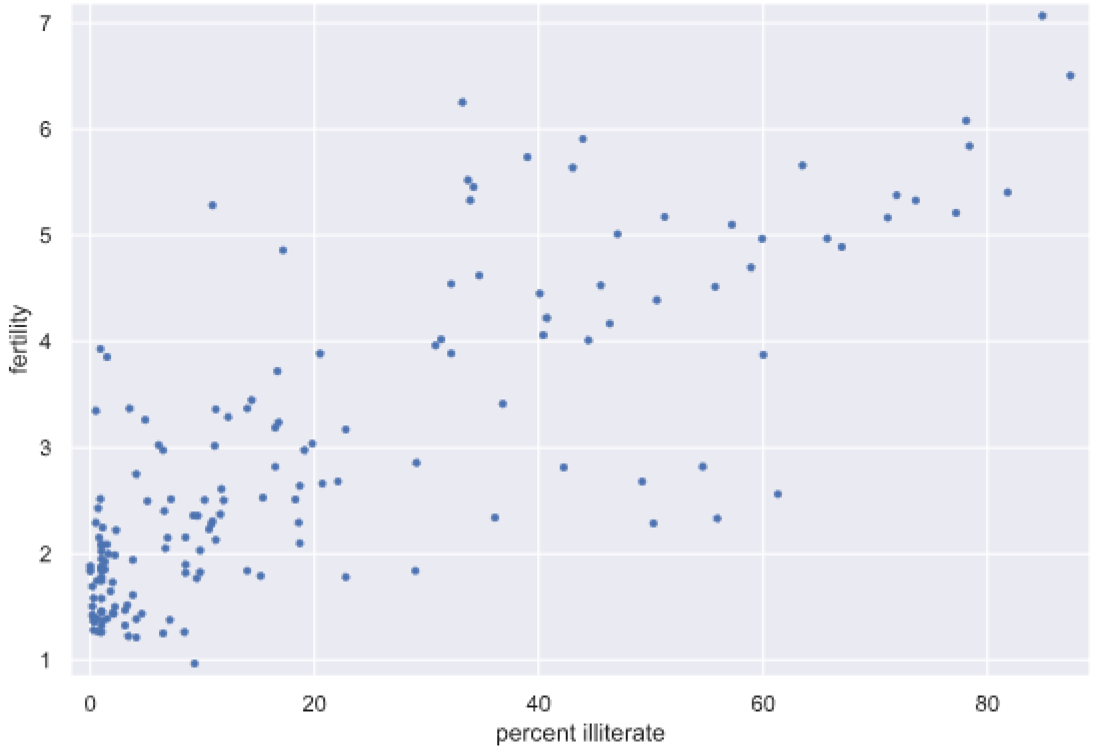


EDA of literacy/fertility data

In the next few exercises, we will look at the correlation between female literacy and fertility (defined as the average number of children born per woman) throughout the world. For ease of analysis and interpretation, we will work with the *il*literacy rate.

It is always a good idea to do some EDA ahead of our analysis. To this end, plot the fertility versus illiteracy and compute the Pearson correlation coefficient. The Numpy array `illiteracy` has the illiteracy rate among females for most of the world's nations. The array `fertility` has the corresponding fertility data.

```
def pearson_r(x, y):
    """Compute Pearson correlation coeff between two arrays."""
    corr_mat = np.corrcoef(x,y)
    return corr_mat[0,1]

# Plot the illiteracy rate versus fertility
_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')

# Set the margins and label axes
plt.margins(0.02)
_ = plt.xlabel('percent illiterate')
_ = plt.ylabel('fertility')

# Show the plot
plt.show()

# Show the Pearson correlation coefficient
print(pearson_r(illiteracy , fertility))
```



```
<script.py> output:
    0.8041324026815344
```

Observation: You can see the correlation between illiteracy and fertility by eye, and by the substantial Pearson correlation coefficient of 0.8. It is difficult to resolve in the scatter plot, but there are many points around near-zero illiteracy and about 1.8 children/woman.

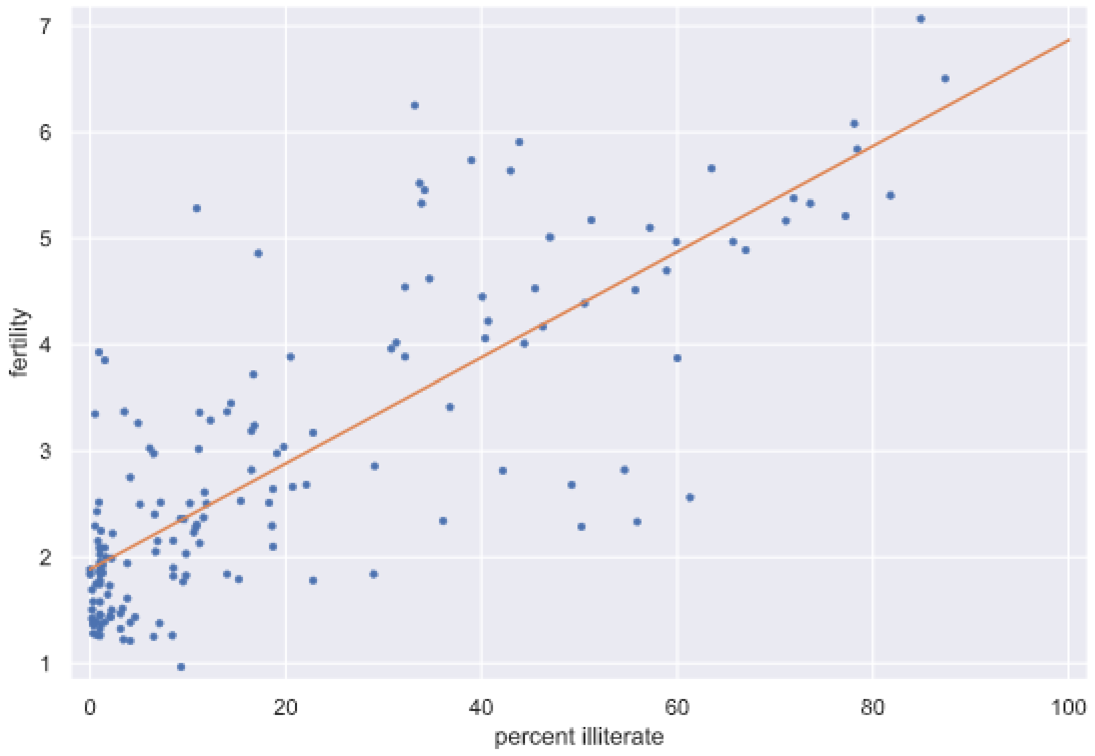
Linear regression

We will assume that fertility is a linear function of the female illiteracy rate. That is, f=**a**i+**b**, where **a** is the slope and **b** is the intercept. We can think of the intercept as the minimal fertility rate, probably somewhere between one and two. The slope tells us how the fertility rate varies with illiteracy. We can find the best fit line using `np. polyfit ()`.

Plot the data and the best fit line. Print out the slope and intercept. (Think: what are their units?)

```
# Plot the illiteracy rate versus fertility
_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')
plt.margins(0.02)
```

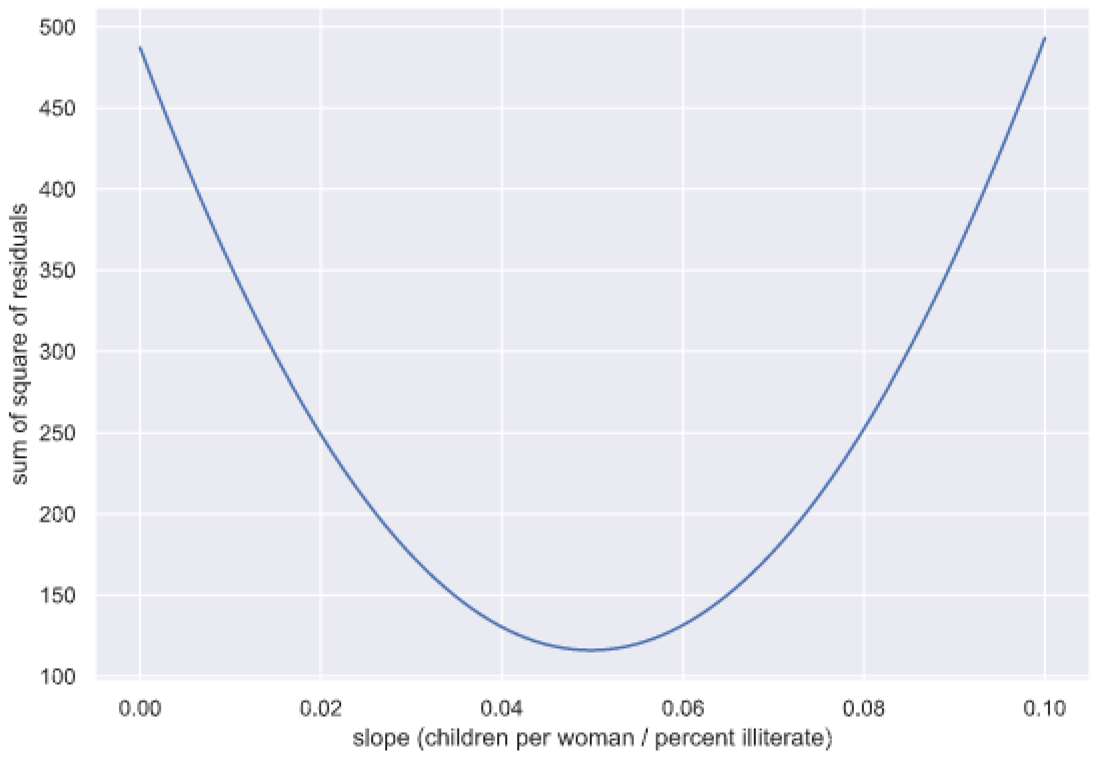
```
_ = plt.xlabel('percent illiterate')
_ = plt.ylabel('fertility')

# Perform a linear regression using np.polyfit(): a, b
a, b = np.polyfit(illiteracy, fertility, deg=1)

# Print the results to the screen
print('slope =', a, 'children per woman / percent illiterate')
print('intercept =', b, 'children per woman')

# Make theoretical line to plot
x = np.array([0, 100])
y = a * x + b

# Add regression line to your plot
_ = plt.plot(x, y)

# Draw the plot
plt.show()

<script.py> output:
    slope = 0.049798548 children per woman / percent illiterate
    intercept = 1.88805061 children per woman
```



How is it optimal?

The function `np.polyfit()` that you used to get your regression parameters finds the *optimal* slope and intercept. It is optimizing the sum of the squares of the residuals, also known as RSS (for residual sum of squares). In this exercise, you will plot the function that is being optimized, the RSS, versus the slope parameter `a`. To do this, fix the intercept to be what you found in the optimization. Then, plot the RSS vs. the slope. Where is it minimal?

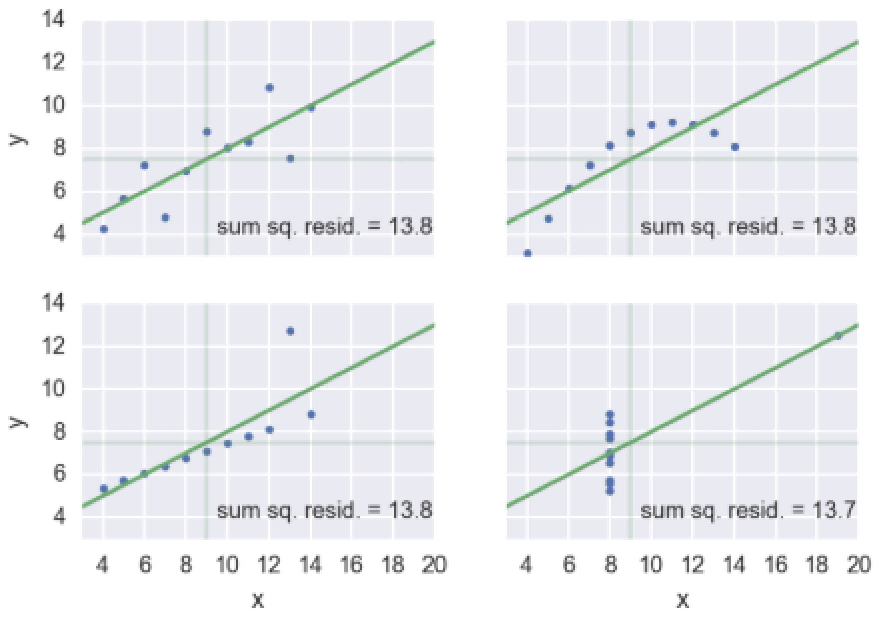
```
# Specify slopes to consider: a_vals
a_vals = np.linspace(0,0.1,200)

# Initialize sum of square of residuals: rss
rss = np.empty_like(a_vals)

# Compute sum of square of residuals for each value of a_vals
for i, a in enumerate(a_vals):
    rss[i] = np.sum((fertility - a*illiteracy - b)**2)

# Plot the RSS
plt.plot(a_vals, rss, '-')
plt.xlabel('slope (children per woman / percent illiterate)')
plt.ylabel('sum of square of residuals')

plt.show()
```



Observation: The minimum on the plot, that is the value of the slope that gives the minimum sum of the square of the residuals, is the same value you got when performing the regression.

The importance of EDA: Anscombe's quartet

The 4 graphs from Anscombe yield the same mean x, the same mean y, the same best fit linear regression line, and the same Sum Square Residual.



The importance of EDA

Why should exploratory data analysis be the first step in an analysis of data (after getting your data imported and cleaned, of course)?

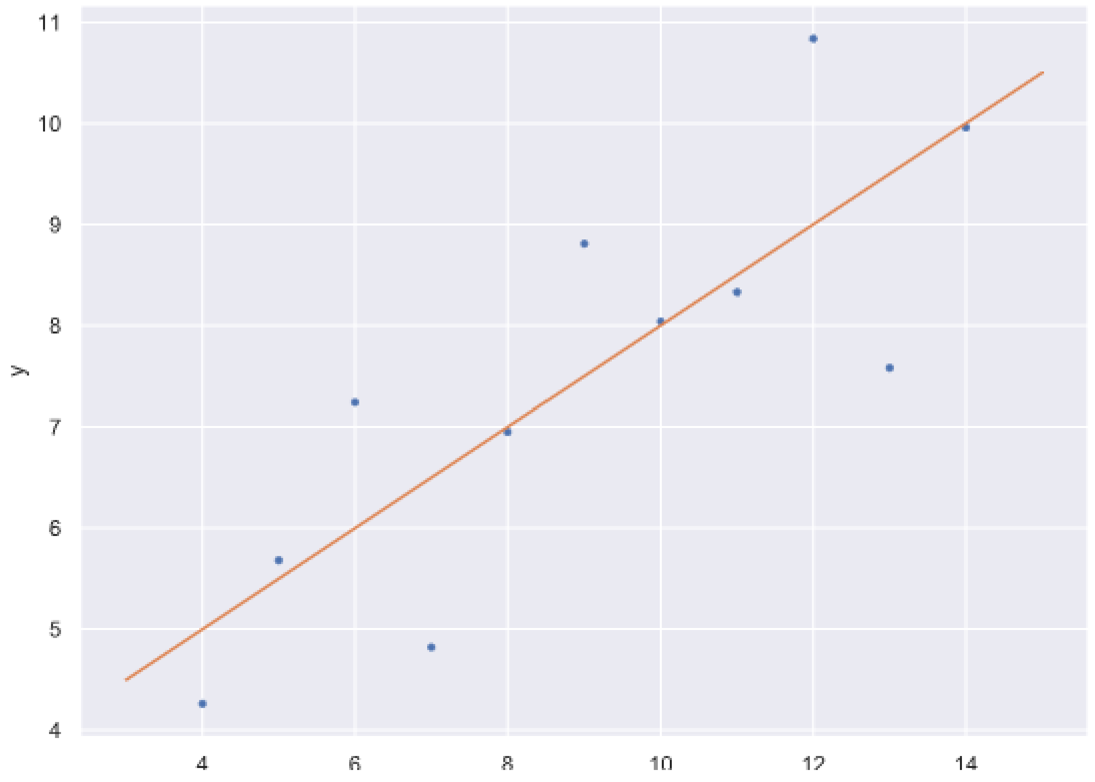
- You can be protected from misinterpretation of the type demonstrated by Anscombe's quartet.
- EDA provides a good starting point for planning the rest of your analysis.
- EDA is not really any more difficult than any of the subsequent analysis, so there is no excuse for not exploring the data.

Always do EDA as you jump into a data set.

Linear regression on appropriate Anscombe data

For practice, perform a linear regression on the data set from Anscombe's quartet that is most reasonably interpreted with linear regression.

```
# Perform linear regression: a, b
a, b = np.polyfit(x,y, deg=1)

# Print the slope and intercept
print(a, b)

# Generate theoretical x and y data: x_theor, y_theor
x_theor = np.array([3, 15])
y_theor = a * x_theor + b

# Plot the Anscombe data and theoretical line
_ = plt.plot(x,y, marker='.' , linestyle='none')
_ = plt.plot(x_theor, y_theor)

# Label the axes
plt.xlabel('x')
plt.ylabel('y')

# Show the plot
plt.show()

<script.py> output:
    0.5000909090909095 3.000090909090909
```



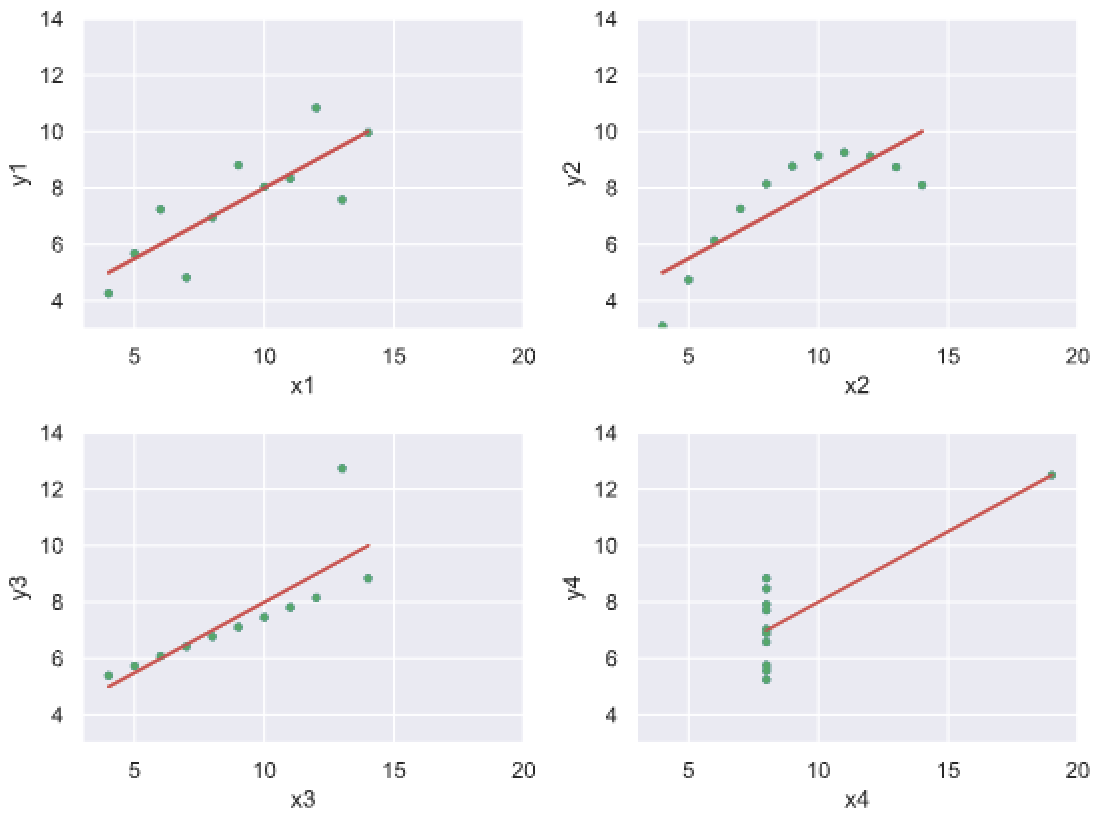
Linear regression on all Anscombe data

Now, to verify that all four of the Anscombe data sets have the same slope and intercept from a linear regression, you will compute the slope and intercept for each set. The data are stored in lists; `anscombe_x = [x1, x2, x3, x4]` and `anscombe_y = [y1, y2, y3, y4]`, where, for example, `x2` and `y2` are the **x** and **y** values for the second Anscombe data set.

```
# Iterate through x,y pairs
for x, y in zip(anscombe_x , anscombe_y ):
    # Compute the slope and intercept: a, b
    a, b = np.polyfit(x,y, deg=1)

# Print the result
    print('slope:', a, 'intercept:', b)

<script.py> output:
    slope: 0.5000909090909095 intercept: 3.000090909090909
    slope: 0.5000000000000004 intercept: 3.0009090909090896
    slope: 0.4997272727272731 intercept: 3.0024545454545453
    slope: 0.4999090909090908 intercept: 3.0017272727272735
```

Observation: Indeed, they all have the same slope and intercept.
Below are the plots to visualise this:

```
for i in range(4):
    plt.subplot(2,2,i+1)

    # plot the scatter plot
    plt.plot(anscombe_x[i], anscombe_y[i], marker = '.', linestyle =
'none')

    # plot the regression line
    a, b = np.polyfit(anscombe_x[i], anscombe_y[i], deg=1)
    x_theor = np.array([np.min(anscombe_x[i]),
np.max(anscombe_x[i])])
    y_theor = a * x_theor + b
    plt.plot(x_theor, y_theor)

    # axis label and limit
    plt.xlabel('x' + str(i+1))
    plt.ylabel('y' + str(i+1))
    plt.xlim(3,20)   # range of x-axis
    plt.ylim(3,14)   # range of y-axis

plt.show()
```



. . .

Chapter 2. Bootstrap confidence intervals

To "pull yourself up by your bootstraps" is a classic idiom meaning that you achieve a difficult task by yourself with no help at all. In statistical inference, you want to know what would happen if you could repeat your data acquisition an infinite number of times. This task is impossible, but can we use only the data we actually have to get close to the same result as an infinitude of experiments? The answer is yes! The technique to do it is aptly called bootstrapping. This chapter will introduce you to this extraordinarily powerful tool.

Generating bootstrap replicates

Bootstrapping is the use of resampled data to perform statistical inference. A **bootstrap sample** is a resampled array that was drawn from the original data with replacement, which can be done using `np.random.choice([original_array], size=sample_size)`.

A **bootstrap replicate** is the value of summary statistics computed from a resampled array (bootstrap sample).

Getting the terminology down

Getting tripped up over terminology is a common cause of frustration in students. Unfortunately, you often will read and hear other data scientists using different terminology for bootstrap samples and replicates. This is even more reason why we need everything to be clear and consistent for this course. So, before going forward discussing bootstrapping, let's get our terminology down. If we have a data set with **n** repeated measurements, a **bootstrap sample** is an array of length **n** that was drawn from the original data with replacement.

What is a **bootstrap replicate**?
Answer: Bootstrap replicate is a single value of a statistic computed from a bootstrap sample.

Bootstrapping by hand

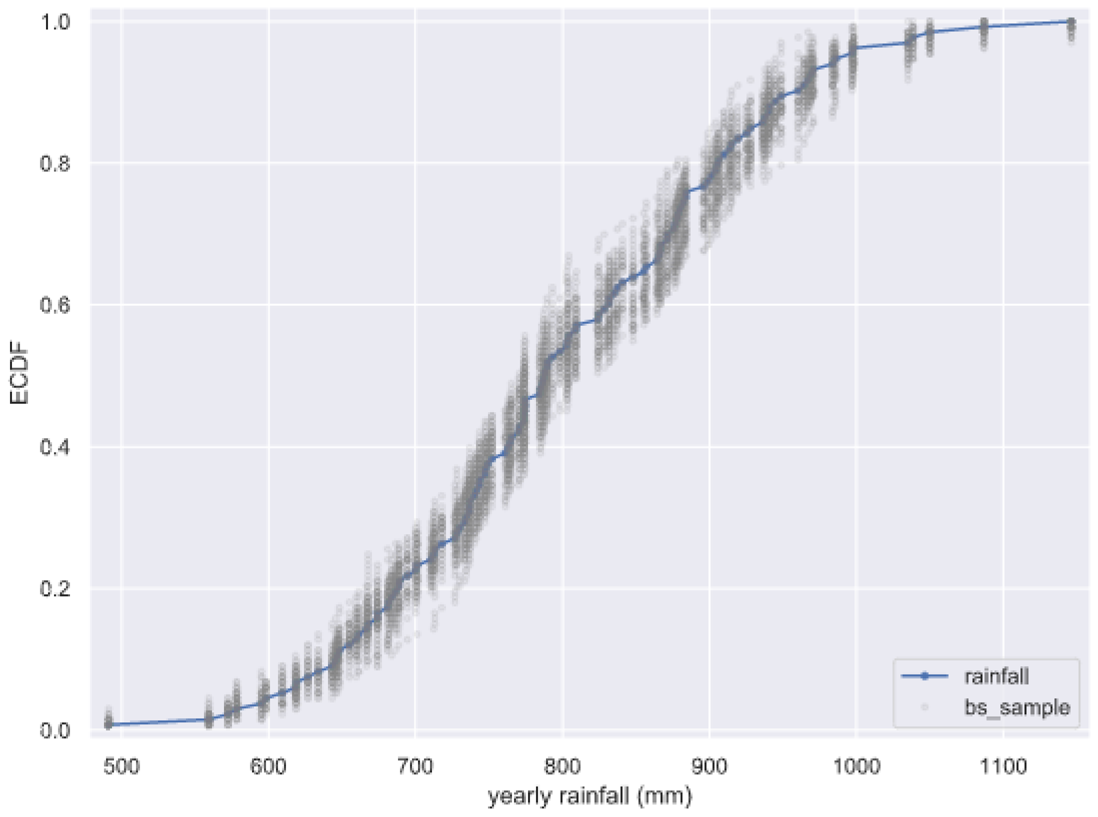
To help you gain intuition about how bootstrapping works, imagine you have a data set that has only three points, `[-1, 0, 1]`. How many unique bootstrap samples can be drawn (e.g., `[-1, 0, 1]` and `[1, 0, -1]` *are* unique), and what is the maximum mean you can get from a bootstrap sample?
Answer: There are 27 unique samples, and the maximum mean is 1.

Observation: There are 27 total bootstrap samples, and one of them, `[1,1,1]` has a mean of 1. Conversely, 7 of them have a mean of zero.

Visualizing bootstrap samples

In this exercise, you will generate bootstrap samples from the set of annual rainfall data measured at the Sheffield Weather Station in the UK from 1883 to 2015. The data are stored in the NumPy array `rainfall` in units of millimeters (mm). By graphically displaying the bootstrap samples with an ECDF, you can get a feel for how bootstrap sampling allows probabilistic descriptions of data.

```
# Compute and plot ECDF from original data
x, y = ecdf(rainfall)
_ = plt.plot(x, y, marker='.')

for _ in range(50):
    # Generate bootstrap sample: bs_sample
    bs_sample = np.random.choice(rainfall, size=len(rainfall))

    # Compute and plot ECDF from bootstrap sample
    x, y = ecdf(bs_sample)
    _ = plt.plot(x, y, marker='.', linestyle='none',
                 c='gray', alpha=0.1)

# Make margins and label axes
plt.margins(0.02)
_ = plt.xlabel('yearly rainfall (mm)')
_ = plt.ylabel('ECDF')
plt.legend(['rainfall','bs_sample'], loc=4)

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



Observation: The bootstrap samples give an idea of how the distribution of rainfalls is spread.

Bootstrap confidence intervals

If we repeated measurements over and over again, p% of the observed values would lie within the **p% confidence interval**.

```
95_confidence_interval = np.percentile(bs_replicates, [2.5, 97.5])
```

Generating many bootstrap replicates

The function `bootstrap_replicate_1d()` is available. Now you'll write another function, `draw_bs_reps(data, func, size=1)`, which generates many bootstrap replicates from the data set. This function will come in handy for you again and again as you compute confidence intervals and later when you do hypothesis tests.

```
def bootstrap_replicate_1d(data, func):
    """Generate bootstrap replicate of 1D data."""
    bs_sample = np.random.choice(data, len(data))
    return func(bs_sample)

def draw_bs_reps(data, func, size=1):
    """Draw bootstrap replicates."""

    # Initialize array of replicates: bs_replicates
    bs_replicates = np.empty(size)

    # Generate replicates
    for i in range(size):
        bs_replicates[i] = bootstrap_replicate_1d(data , func)

    return bs_replicates
```

Note: This function `draw_bs_reps()` will be a workhorse for you!

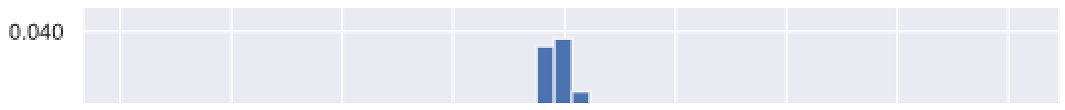
Bootstrap replicates of the mean and the SEM (standard error of the mean)

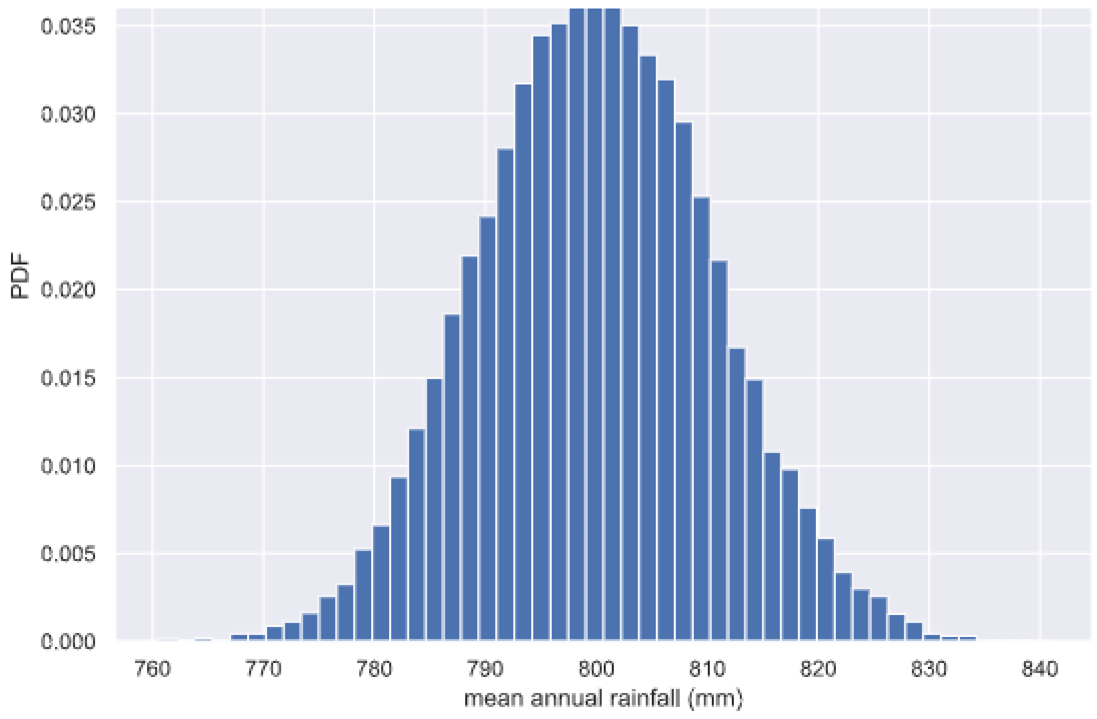
In this exercise, you will compute a bootstrap estimate of the probability density function of the mean annual rainfall at the Sheffield Weather Station. We are estimating the mean annual rainfall we would get if the Sheffield Weather Station could repeat all of the measurements from 1883 to 2015 over and over again. This is a *probabilistic* estimate of the mean. You will plot the PDF as a histogram, and you will see that it is Normal.

In fact, it can be shown theoretically that under not-too-restrictive conditions, the value of the mean will always be Normally distributed. (This does not hold in general, just for the mean and a few other statistics.) The standard deviation of this distribution, called the **standard error of the mean**, or SEM, is given by the standard deviation of the data divided by the square root of the number of data points, i.e., for a data set, `sem = np.std(data) / np.sqrt(len(data))`. Using hacker statistics, you get this same result without the need to derive it, but you will verify this result from your bootstrap replicates.

The dataset is in an array called `rainfall`.

```
# Take 10,000 bootstrap replicates of the mean: bs_replicates
bs_replicates = draw_bs_reps(rainfall, np.mean, 10000)

# Compute and print SEM
sem = np.std(rainfall) / np.sqrt(len(rainfall))
print(sem)

# Compute and print standard deviation of bootstrap replicates
bs_std = np.std(bs_replicates)
print(bs_std)

# Make a histogram of the results
_ = plt.hist(bs_replicates, bins=50, normed=True)
_ = plt.xlabel('mean annual rainfall (mm)')
_ = plt.ylabel('PDF')

# Show the plot
plt.show()

<script.py> output:
    10.51054915050619
    10.465927071184412
```





Observation: The SEM we got from the known expression and the bootstrap replicates is the same and the distribution of the bootstrap replicates of the mean is Normal.

Confidence intervals of rainfall data

A *confidence interval* gives upper and lower bounds on the range of parameter values you might expect to get if we repeat our measurements. For named distributions, you can compute them analytically or look them up, but one of the many beautiful properties of the bootstrap method is that you can take percentiles of your bootstrap replicates to get your confidence interval. Conveniently, you can use the `np.percentile()` function.

Use the bootstrap replicates you just generated to compute the 95% confidence interval. That is, give the 2.5th and 97.5th percentile of your bootstrap replicates stored as `bs_replicates`. What is the 95% confidence interval?

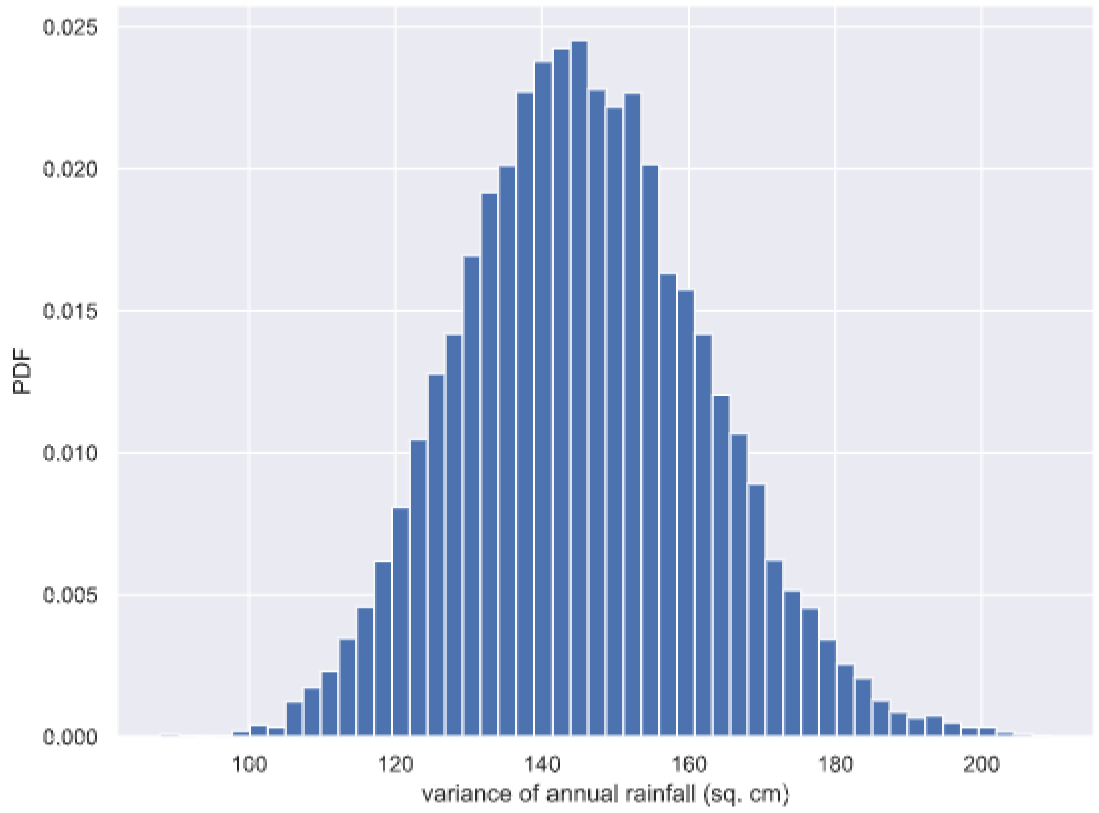
Answer: (780, 821) mm/year

```
In [1]: np.percentile(bs_replicates, [2.5, 97.5])
Out[1]: array([779.76992481, 820.95043233])
```

Bootstrap replicates of other statistics

We saw in a previous exercise that the mean is Normally distributed. This does not necessarily hold for other statistics, but no worry: as hackers, we can always take bootstrap replicates! In this exercise, you'll generate bootstrap replicates for the variance of the annual rainfall at the Sheffield Weather Station and plot the histogram of the replicates.

```
def draw_bs_reps(data, func, size=1):
    return np.array([bootstrap_replicate_1d(data, func) for _ in 
range(size)])

# Generate 10k bootstrap replicates of the variance: bs_replicates
bs_replicates = draw_bs_reps(rainfall, np.var, 10000)

# Put the variance in units of square centimeters
bs_replicates = bs_replicates/100

# Make a histogram of the results
_ = plt.hist(bs_replicates, normed=True, bins=50)
_ = plt.xlabel('variance of annual rainfall (sq. cm)')
_ = plt.ylabel('PDF')

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



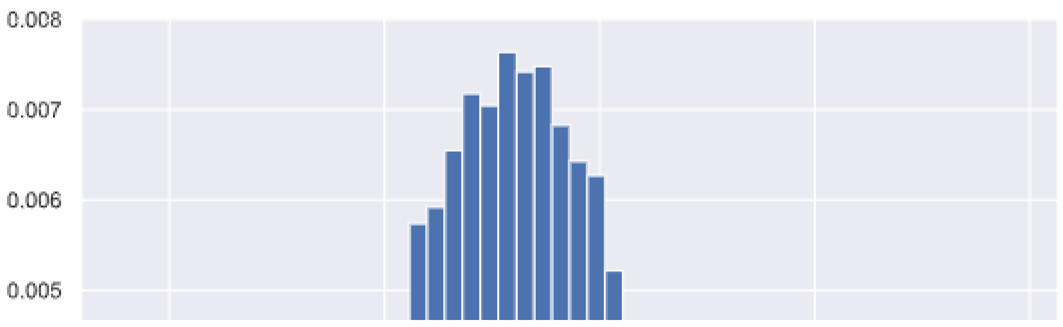
Observation: This is not normally distributed, as it has a longer tail to the right. Note that you can also compute a confidence interval on the variance, or any other statistic, using `np.percentile()` with your bootstrap replicates.

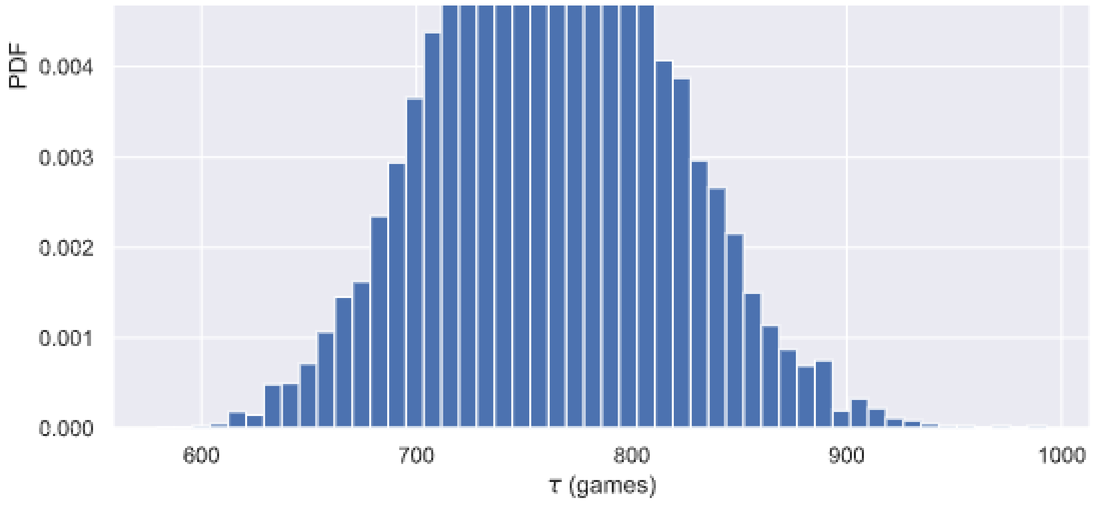
```
print(np.percentile(bs_replicates, [2.5, 97.5]))
# Output: [114.87926838 179.45385568]
```

Confidence interval on the rate of no-hitters

Consider again the inter-no-hitter intervals for the modern era of baseball. Generate 10,000 bootstrap replicates of the optimal parameter **τ**. Plot a histogram of your replicates and report a 95% confidence interval.

```
# Draw bootstrap replicates of the mean no-hitter time (equal to
tau): bs_replicates
bs_replicates = draw_bs_reps(nohitter_times, np.mean, 10000)

# Compute the 95% confidence interval: conf_int
conf_int = np.percentile( bs_replicates, [2.5, 97.5] )

# Print the confidence interval
print('95% confidence interval =', conf_int, 'games')

# Plot the histogram of the replicates
_ = plt.hist(bs_replicates, bins=50, normed=True)
_ = plt.xlabel(r'$\tau$ (games)')
_ = plt.ylabel('PDF')

# Show the plot
plt.show()

<script.py> output:
    95% confidence interval = [660.67280876 871.63077689] games
```





Observation: This gives you an estimate of what the typical time between no-hitters is. It could be anywhere between 660 and 872 games.

Pairs bootstrap

To perform bootstrap estimates to get the confidence interval on the slope and intercept of the linear regression model.

A function to do pairs bootstrap

Pairs bootstrap involves resampling pairs of data. Each collection of pairs fit with a line, in this case using `np.polyfit()` . We do this again and again, getting bootstrap replicates of the parameter values. To have a useful tool for doing pairs bootstrap, you will write a function to perform pairs bootstrap on a set of `x,y` data.

```
def draw_bs_pairs_linreg(x, y, size=1):
    """Perform pairs bootstrap for linear regression."""

    # Set up array of indices to sample from: inds
    inds = np.arange(len(x))

    # Initialize replicates: bs_slope_reps, bs_intercept_reps
    bs_slope_reps = np.empty(size)
    bs_intercept_reps = np.empty(size)

    # Generate replicates
    for i in range(size):
        bs_inds = np.random.choice(inds, size=len(inds))
        bs_x, bs_y = x[bs_inds], y[bs_inds]
        bs_slope_reps[i], bs_intercept_reps[i] = np.polyfit(bs_x,
bs_y, 1)
```

```
    return bs_slope_reps, bs_intercept_reps
```

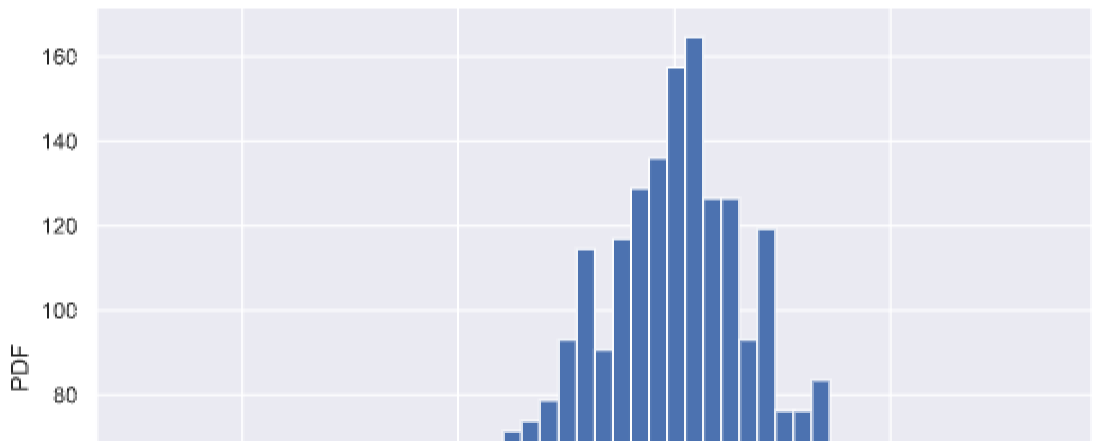
Note: This function `draw_bs_pairs_linreg()` will also be useful later.

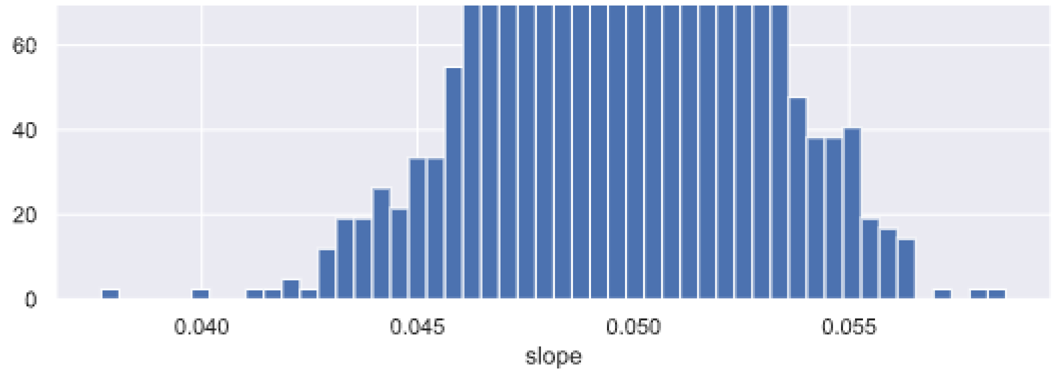
Pairs bootstrap of literacy/fertility data

Using the function you just wrote, perform pairs bootstrap to plot a histogram describing the estimate of the slope from the illiteracy/fertility data. Also report the 95% confidence interval of the slope. The data is available to you in the NumPy arrays `illiteracy` and `fertility`.

As a reminder, `draw_bs_pairs_linreg()` has a function signature of `draw_bs_pairs_linreg(x, y, size=1)`, and it returns two values: `bs_slope_reps` and `bs_intercept_reps`.

```
# Generate replicates of slope and intercept using pairs bootstrap
bs_slope_reps, bs_intercept_reps = draw_bs_pairs_linreg(illiteracy ,
fertility, 1000)

# Compute and print 95% CI for slope
print(np.percentile(bs_slope_reps, [2.5, 97.5]))

# Plot the histogram
_ = plt.hist(bs_slope_reps, bins=50, normed=True)
_ = plt.xlabel('slope')
_ = plt.ylabel('PDF')
plt.show()

<script.py> output:
    [0.04378061 0.0551616 ]
```

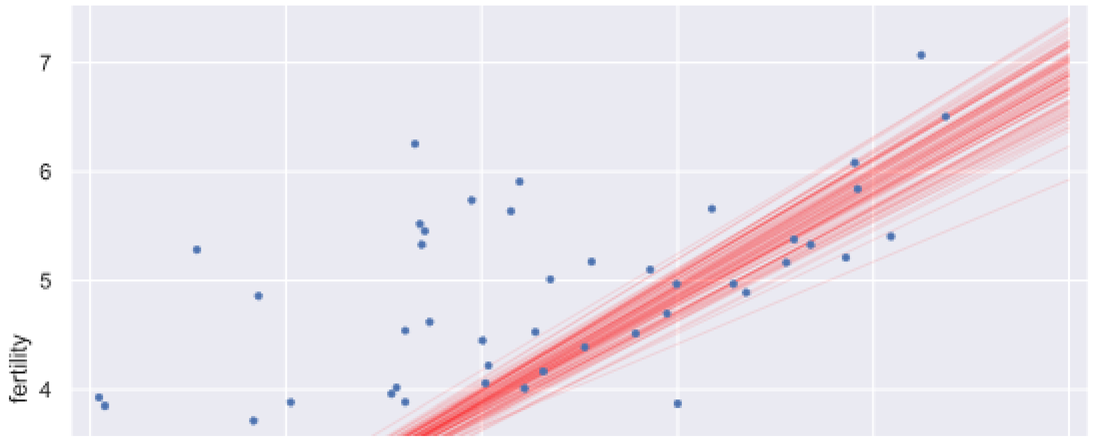


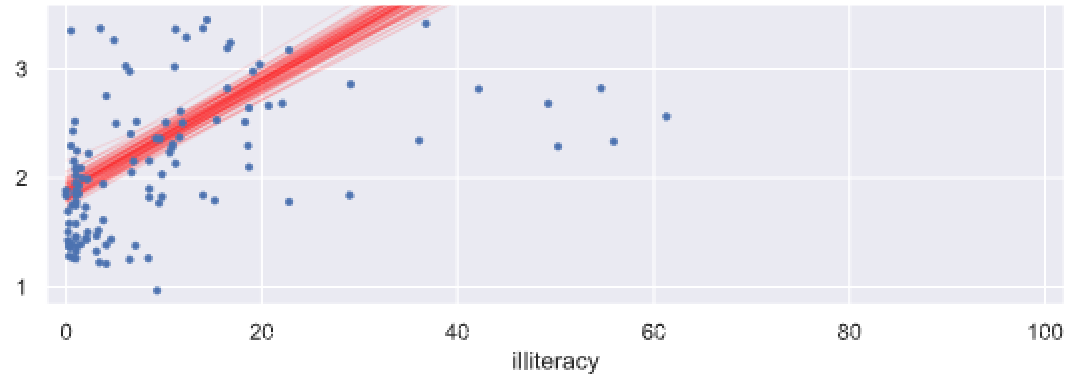


Plotting bootstrap regressions

A nice way to visualize the variability we might expect in a linear regression is to plot the line you would get from each bootstrap replicate of the slope and intercept. Do this for the first 100 of your bootstrap replicates of the slope and intercept (stored as `bs_slope_reps` and `bs_intercept_reps`).

```
# Generate array of x-values for bootstrap lines: x
x = np.array([0, 100])

# Plot the bootstrap lines
for i in range(100):
    _ = plt.plot(x, bs_slope_reps[i]*x + bs_intercept_reps[i],
                 linewidth=0.5, alpha=0.2, color='red')

# Plot the data
_ = plt.plot(illiteracy,fertility, marker='.', linestyle='none')

# Label axes, set the margins, and show the plot
_ = plt.xlabel('illiteracy')
_ = plt.ylabel('fertility')
plt.margins(0.02)
plt.show()
```





You now have some serious chops for parameter estimation. Let's move on to hypothesis testing!

. . .

Chapter 3. Introduction to hypothesis testing

You now know how to define and estimate parameters given a model. But the question remains: how reasonable is it to observe your data if a model is true? This question is addressed by hypothesis tests. They are the icing on the inference cake. After completing this chapter, you will be able to carefully construct and test hypotheses using hacker statistics.

Formulating and simulating a hypothesis

Null hypothesis: another name for the hypothesis you are testing
Permutation: random reordering of entries in an array

Generating a permutation sample

Permutation sampling is a great way to simulate the hypothesis that two variables have identical probability distributions. This is often a hypothesis you want to test, so in this exercise, you will write a function to generate a permutation sample from two data sets.

A permutation sample of two arrays having respectively `n1` and `n2` entries is constructed by concatenating the arrays together, scrambling the contents of the concatenated array, and then taking the first `n1` entries as the permutation sample of the first array and the last `n2` entries as the permutation sample of the second array.

```
def permutation_sample(data1, data2):
    """Generate a permutation sample from two data sets."""

    # Concatenate the data sets: data
    data = np.concatenate((data1, data2))

    # Permute the concatenated array: permuted_data
    permuted_data = np.random.permutation(data)

    # Split permuted array into 2: perm_sample_1, perm_sample_2
    perm_sample_1 = permuted_data[:len(data1)]
    perm_sample_2 = permuted_data[len(data1):]

    return perm_sample_1, perm_sample_2
```

Visualizing permutation sampling

To help see how permutation sampling works, in this exercise you will generate permutation samples and look at them graphically.

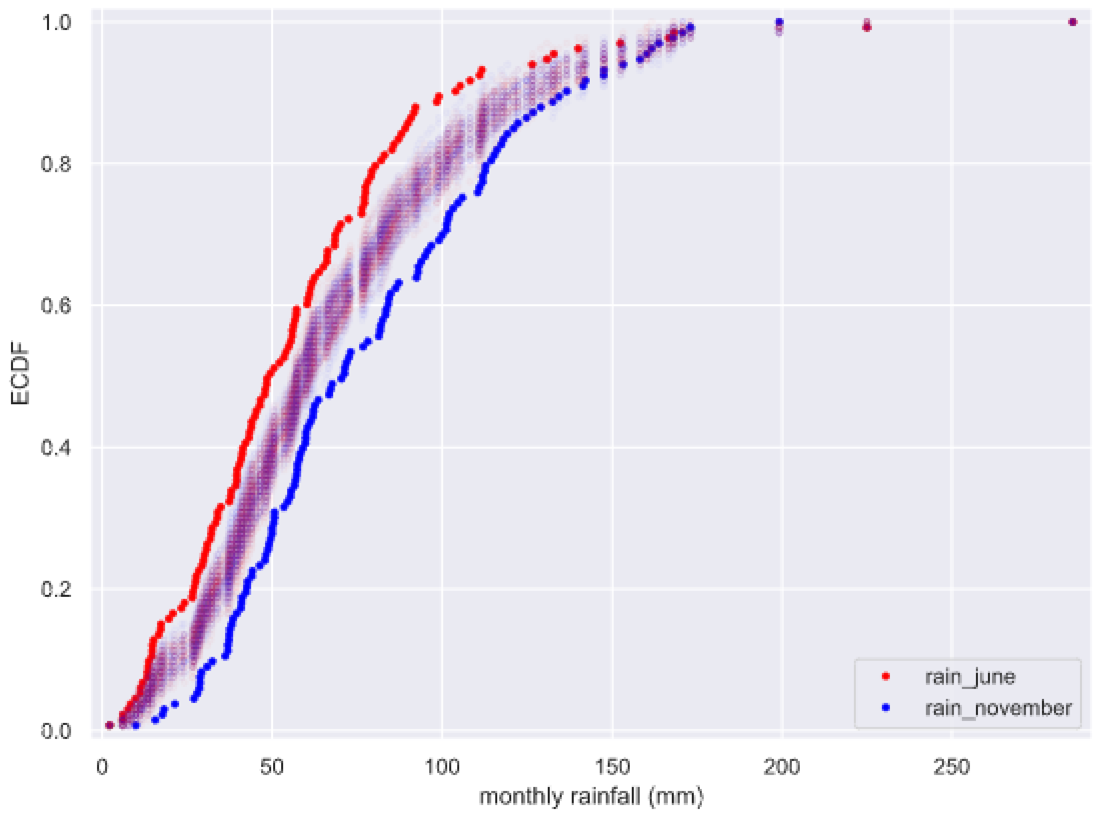
We will use the Sheffield Weather Station data again, this time considering the monthly rainfall in June (a dry month) and November (a wet month). We expect these might be differently distributed, so we will take permutation samples to see how their ECDFs *would look if* they were identically distributed.

The data are stored in the Numpy arrays `rain_june` and `rain_november`.

As a reminder, `permutation_sample()` has a function signature of `permutation_sample(data_1, data_2)` with a return value of `permuted_data[:len(data_1)], permuted_data[len(data_1):]`, where `permuted_data = np.random.permutation(np.concatenate((data_1, data_2)))`.

```
# Create and plot ECDFs from original data
x_1, y_1 = ecdf(rain_june)
x_2, y_2 = ecdf(rain_november)
_ = plt.plot(x_1, y_1, marker='.', linestyle='none', color='red')
_ = plt.plot(x_2, y_2, marker='.', linestyle='none', color='blue')

for _ in range(50):
    # Generate permutation samples
    perm_sample_1, perm_sample_2 =
permutation_sample(rain_june,rain_november)
```

```
        # Compute ECDFs
        x_1, y_1 = ecdf(perm_sample_1)
        x_2, y_2 = ecdf(perm_sample_2)

        # Plot ECDFs of permutation sample
        _ = plt.plot(x_1, y_1, marker='.', linestyle='none',
                     color='red', alpha=0.02)
        _ = plt.plot(x_2, y_2, marker='.', linestyle='none',
                     color='blue', alpha=0.02)

    # Label axes, set margin, and show plot
    plt.margins(0.02)
    _ = plt.xlabel('monthly rainfall (mm)')
    _ = plt.ylabel('ECDF')
    plt.legend(['rain_june','rain_november'], loc=4)
    plt.show()
```



Observation: The permutation samples ECDFs overlap and give a purple haze. None of the ECDFs from the permutation samples overlap with the observed data, suggesting that the hypothesis is not commensurate with the data. June and November rainfall are not identically distributed.

Test statistics and p-values

Hypothesis testing: Assessment of how reasonable the observed data are, assuming a hypothesis is true
Test statistic: A single number that can be computed from observed data and from data you simulate under the null hypothesis
p-value: The probability of obtaining a value of your test statistic that is at least as extreme as what was observed, under the assumption the null hypothesis is true
Statistical significance: Determined when p-value is small

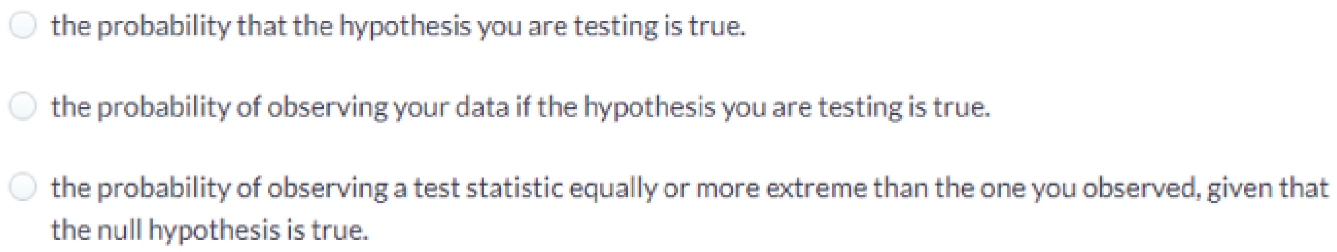
Test statistics

When performing hypothesis tests, your choice of test statistic should be:
Answer: be pertinent to the question you are seeking to answer in your hypothesis test.

The most important thing to consider is: What are you asking?

What is a p-value?

The p-value is generally a measure of:



- the probability that the hypothesis you are testing is true.
- the probability of observing your data if the hypothesis you are testing is true.
- the probability of observing a test statistic equally or more extreme than the one you observed, given that the null hypothesis is true.

Answer: the probability of observing a test statistic equally or more extreme than the one you observed, given that the null hypothesis is true.

Generating permutation replicates

A permutation replicate is a single value of a statistic computed from a permutation sample. As the `draw_bs_reps()` function is useful for you to generate bootstrap replicates, it is useful to have a similar function, `draw_perm_reps()` , to generate permutation replicates. You will write this useful function in this exercise.

The function has call signature `draw_perm_reps(data_1, data_2, func, size=1)` . Importantly, `func` must be a function that takes *two* arrays as arguments. In most circumstances, `func` will be **a function you write yourself**.

```
def draw_perm_reps(data_1, data_2, func, size=1):
    """Generate multiple permutation replicates."""

    # Initialize array of replicates: perm_replicates
    perm_replicates = np.empty(size)

    for i in range(size):
        # Generate permutation sample
        p_sample_1, p_sample_2 = permutation_sample(data_1, data_2)

        # Compute the test statistic
        perm_replicates[i] = func(p_sample_1, p_sample_2)

    return perm_replicates
```

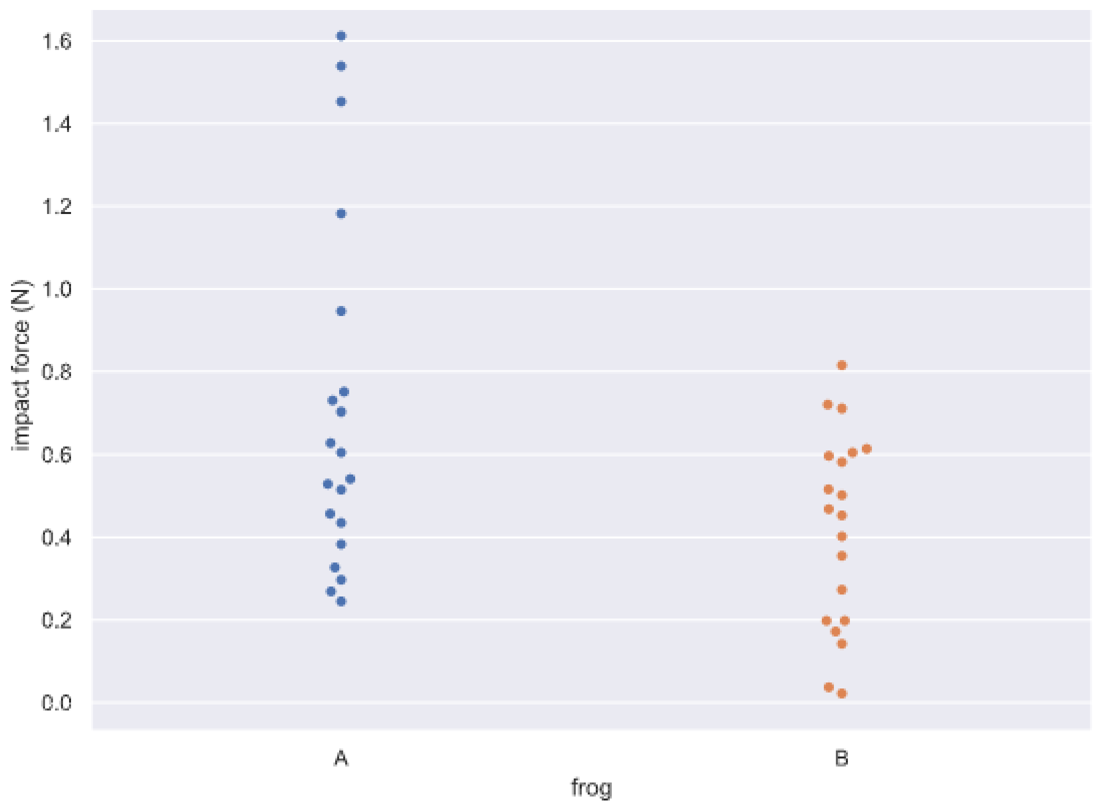
Look before you leap: EDA before hypothesis testing

Kleinteich and Gorb (*Sci. Rep.*, **4**, 5225, 2014) performed an interesting experiment with South American horned frogs. They held a plate connected to a force transducer, along with a bait fly, in front of them. They then measured the impact force and adhesive force of the frog's tongue when it struck the target.

Frog A is an adult and Frog B is a juvenile. The researchers measured the impact force of 20 strikes for each frog. In the next exercise, we will test the hypothesis that the two frogs have the same distribution of impact forces. But it is important to do EDA first! Let's make a bee swarm plot for the data. They are stored in a Pandas data frame, `df`, where column `ID` is the identity of the frog and column `impact_force` is the impact force in Newtons (N).

```
df.head()
   ID  impact_force
20  A         1.612
21  A         0.605
22  A         0.327
23  A         0.946
24  A         0.54

# Make bee swarm plot
_ = sns.swarmplot(x='ID', y='impact_force', data=df)

# Label axes
_ = plt.xlabel('frog')
_ = plt.ylabel('impact force (N)')
```

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



Observation: Visually, it does not look like they come from the same distribution. Frog A, the adult, has three or four very hard strikes, and Frog B, the juvenile, has a couple weak ones. However, it is possible that with only 20 samples it might be too difficult to tell if they have difference distributions, so we should proceed with the hypothesis test.

Permutation test on frog data

The average strike force of Frog A (array `force_a`) was 0.71 Newtons (N), and that of Frog B (array `force_b`) was 0.42 N for a difference of 0.29 N. It is possible the frogs strike with the same force and this observed difference was by chance. You will compute the probability of getting at least a 0.29 N difference in mean strike force under the hypothesis that the distributions of strike forces for the two frogs are identical. We use a permutation test with a test statistic of the difference of means to test this hypothesis.

```
def diff_of_means(data_1, data_2):
    """Difference in means of two arrays."""
    # The difference of means of data_1, data_2: diff
    diff = np.mean(data_1) - np.mean(data_2)
    return diff

# Compute difference of mean impact force from experiment:
empirical_diff_means
empirical_diff_means = diff_of_means(force_a , force_b)

# Draw 10,000 permutation replicates: perm_replicates
perm_replicates = draw_perm_reps(force_a , force_b,
                                 diff_of_means, size=10000)

# Compute p-value: p
p = np.sum(perm_replicates >= empirical_diff_means) /
len(perm_replicates)

# Print the result
print('p-value =', p)

<script.py> output:
    p-value = 0.0063
```

Observation: The p-value tells you that there is about a 0.6% chance that you would get the difference of means observed in the experiment if frogs were exactly the same. So the strike forces are significantly different statistically.

Bootstrap hypothesis tests

Pipeline for hypothesis testing:

1. state null hypothesis
2. define test statistic
3. generate many sets of simulated data, assuming null hypothesis is true
4. compute test statistic for each simulated data set
5. the p-value is the fraction of simulated data sets for which the test statistic is at least as extreme as for the real data

A one-sample bootstrap hypothesis test

Another juvenile frog was studied, Frog C, and you want to see if Frog B and Frog C have similar impact forces. Unfortunately, you do not have Frog C's impact forces available, but you know they have a mean of 0.55 N. Because you don't have the original data, you cannot do a permutation test, and you cannot assess the hypothesis that the forces from Frog B and Frog C come from the same distribution. You will therefore test another, less restrictive hypothesis: The mean strike force of Frog B is equal to that of Frog C.

To set up the bootstrap hypothesis test, you will take the mean as our test statistic. Your goal is to calculate the probability of getting a mean impact force less than or equal to what was observed for Frog B *if the hypothesis that the true mean of Frog B's impact forces is equal to that of Frog C is true*. You first translate all of the data of Frog B such that the mean is 0.55 N. This involves adding the mean force of Frog C and subtracting the mean force of Frog B from each measurement of Frog B. This leaves other properties of Frog B's distribution, such as the variance, unchanged.

```
# Make an array of translated impact forces: translated_force_b
translated_force_b = force_b - np.mean(force_b) + 0.55

# Take bootstrap replicates of Frog B's translated impact forces:
bs_replicates
bs_replicates = draw_bs_reps(translated_force_b, np.mean, 10000)

# Compute fraction of replicates that are less than the observed Frog
B force: p
p = np.sum(bs_replicates <= np.mean(force_b)) / 10000

# Print the p-value
print('p = ', p)

<script.py> output:
    p =  0.0046
```

Observation: The low p-value suggests that the null hypothesis (Frog B and Frog C have the same mean impact force) is false.

A two-sample bootstrap hypothesis test for difference of means

We now want to test the hypothesis that Frog A and Frog B have the same mean impact force, but not necessarily the same distribution, which is also impossible with a permutation test.

To do the two-sample bootstrap test, we shift *both* arrays to have the same mean, since we are simulating the hypothesis that their means are, in fact, equal. We then draw bootstrap samples out of the shifted arrays and compute the difference in means. This constitutes a bootstrap replicate, and we generate many of them. The p-value is the fraction of replicates with a difference in means greater than or equal to what was observed.

The objects `forces_concat` and `empirical_diff_means` are available.

```
# Compute mean of all forces: mean_force
mean_force = np.mean(forces_concat)

# Generate shifted arrays
force_a_shifted = force_a - np.mean(force_a) + mean_force
force_b_shifted = force_b - np.mean(force_b) + mean_force

# Compute 10,000 bootstrap replicates from shifted arrays
bs_replicates_a = draw_bs_reps(force_a_shifted, np.mean, 10000)
bs_replicates_b = draw_bs_reps(force_b_shifted, np.mean, 10000)

# Get replicates of difference of means: bs_replicates
bs_replicates = bs_replicates_a - bs_replicates_b

# Compute and print p-value: p
p = np.sum(bs_replicates >= empirical_diff_means) /
len(bs_replicates)
print('p-value =', p)

<script.py> output:
    p-value = 0.0043
```

Observation: You got a similar result as when you did the permutation test. Think carefully about what question you want to ask. Are you only interested in the mean impact force (use permutation test)? Or interested in the distribution of impact forces (use two-sample bootstrap hypothesis test)?

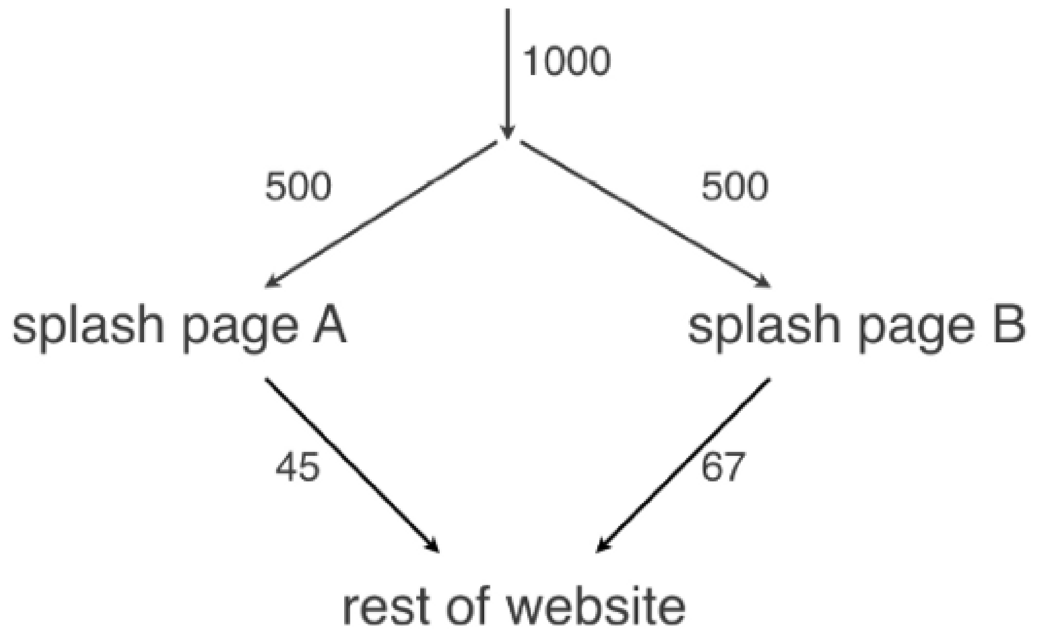
. . .

Chapter 4. Hypothesis test examples

As you saw from the last chapter, hypothesis testing can be a bit tricky. You need to define the null hypothesis, figure out how to simulate it, and define clearly what it means to be “more extreme” in order to compute the p-value. Like any skill, practice makes perfect, and this chapter gives you some good practice with hypothesis tests.

A/B testing

Problem statement: Is your redesign effective?



H0: no difference in traffic
H1: new design has higher traffic
Test statistic: Permutation test of clicks through

```
def permutation_replicate(data1, data2, func, size=1):
    """Generate multiple permutation replicates."""
    # Initialize array of replicates: perm_replicates
    perm_replicates = np.empty(size)
    for i in range(size):
        # Generate permutation sample
        p_sample_1, p_sample_2 = permutation_sample(data1, data2)
        # Compute the test statistic
        perm_replicates[i] = func(p_sample_1, p_sample_2)
    return perm_replicates

def diff_frac(data_A, data_B):
    frac_A = np.sum(data_A) / len(data_A)
    frac_B = np.sum(data_B) / len(data_B)
    return frac_B - frac_A

diff_frac_observed = diff_frac(clickthrough_A, clickthrough_B)
# clickthrough_A, clickthrough_B: arr of 1s and 0s

perm_replicates = np.empty(10000)
for i in range(10000):
    perm_replicates[i] = permutation_replicate(clickthrough_A,
clickthrough_B, diff_frac)

p-vlaue = np.sum(perm_replicates >= diff_fracc_obs) / 10000
# 0.016, ie, reject null hypo, redesign is an improvement
```

The vote for the Civil Rights Act in 1964

The Civil Rights Act of 1964 was one of the most important pieces of legislation ever passed in the USA. Excluding "present" and "abstain" votes, *153 House Democrats and 136 Republicans voted yea*. However, *91 Democrats and 35 Republicans voted nay*. Did party affiliation make a difference in the vote?

To answer this question, you will evaluate the hypothesis that the party of a House member has no bearing on his or her vote. You will use the fraction of Democrats voting in favor as your test statistic and evaluate the probability of observing a fraction of Democrats voting in favor at least as small as the observed fraction of 153/244. (That's right, at least as *small* as. In 1964, it was the *Democrats* who were less progressive on civil rights issues.) To do this, permute the party labels of the House voters and then arbitrarily divide them into "Democrats" and "Republicans" and compute the fraction of Democrats voting yea.

```
# Construct arrays of data: dems, reps
dems = np.array([True] * 153 + [False] * 91)
reps = np.array([True] * 136 + [False] * 35)

def frac_yea_dems(dems, reps):
    """Compute fraction of Democrat yea votes."""
    frac = np.sum(dems) / len(dems)
    return frac

# Acquire permutation samples: perm_replicates
perm_replicates = draw_perm_reps(dems, reps, frac_yea_dems, 10000)
```

```
# Compute and print p-value: p
p = np.sum(perm_replicates <= 153/244) / len(perm_replicates)
print('p-value =', p)

<script.py> output:
    p-value = 0.0002
```

Observation: This small p-value suggests that party identity had a lot to do with the voting. Importantly, the South had a higher fraction of Democrat representatives, and consequently also a more racist bias.

What is equivalent?

You have experience matching stories to probability distributions. Similarly, you use the same procedure for two different A/B tests if their stories match. In the Civil Rights Act example you just did, you performed an A/B test on voting data, which has a Yes/No type of outcome for each subject (in that case, a voter). Which of the following situations involving testing by a web-based company has an equivalent set up for an A/B test as the one you just did with the Civil Rights Act of 1964?
Answer: You measure the number of people who click on an ad on your company's website before and after changing its color.

The "Democrats" are those who view the ad before the color change, and the "Republicans" are those who view it after.

A time-on-website analog

It turns out that you already did a hypothesis test analogous to an A/B test where you are interested in how much time is spent on the website before and after an ad campaign. The frog tongue force (a continuous quantity like time on the website) is an analog. "Before" = Frog A and "after" = Frog B. Let's practice this again with something that actually is a before/after scenario.

We return to the no-hitter data set. In 1920, Major League Baseball implemented important rule changes that ended the so-called dead ball era. Importantly, the pitcher was no longer allowed to spit on or scuff the ball, an activity that greatly favors pitchers. In this problem you will perform an A/B test to determine if these rule changes resulted in a slower rate of no-hitters (i.e., longer average time between no-hitters) using the difference in mean inter-no-hitter time as your test statistic. The inter-no-hitter times for the respective eras are stored in the arrays `nht_dead` and `nht_live`, where "nht" is meant to stand for "no-hitter time."

```
def diff_of_means(data_1, data_2):
    """Difference in means of two arrays."""
    # The difference of means of data_1, data_2: diff
    diff = np.mean(data_1) - np.mean(data_2)
    return diff

def permutation_sample(data1, data2):
    """Generate a permutation sample from two data sets."""
    # Concatenate the data sets: data
    data = np.concatenate((data1, data2))
    # Permute the concatenated array: permuted_data
    permuted_data = np.random.permutation(data)
    # Split the permuted array into two: p_sample_1, p_sample_2
    p_sample_1 = permuted_data[:len(data1)]
    p_sample_2 = permuted_data[len(data1):]
    return p_sample_1, p_sample_2

def draw_perm_reps(data_1, data_2, func, size=1):
    """Generate multiple permutation replicates."""
    # Initialize array of replicates: perm_replicates
    perm_replicates = np.empty(size)
    for i in range(size):
        # Generate permutation sample
        p_sample_1, p_sample_2 = permutation_sample(data_1, data_2)
        # Compute the test statistic
        perm_replicates[i] = func(p_sample_1, p_sample_2)
    return perm_replicates


# Compute the observed difference in mean inter-no-hitter times:
nht_diff_obs
nht_diff_obs = diff_of_means(nht_dead, nht_live)

# Acquire 10,000 permutation replicates of difference in mean no-
hitter time: perm_replicates
perm_replicates = draw_perm_reps(nht_dead, nht_live, diff_of_means,
size=10000)

# Compute and print the p-value: p
p = np.sum(perm_replicates <= nht_diff_obs) /len(perm_replicates)
print('p-val =', p)

<script.py> output:
    p-val = 0.0001
```

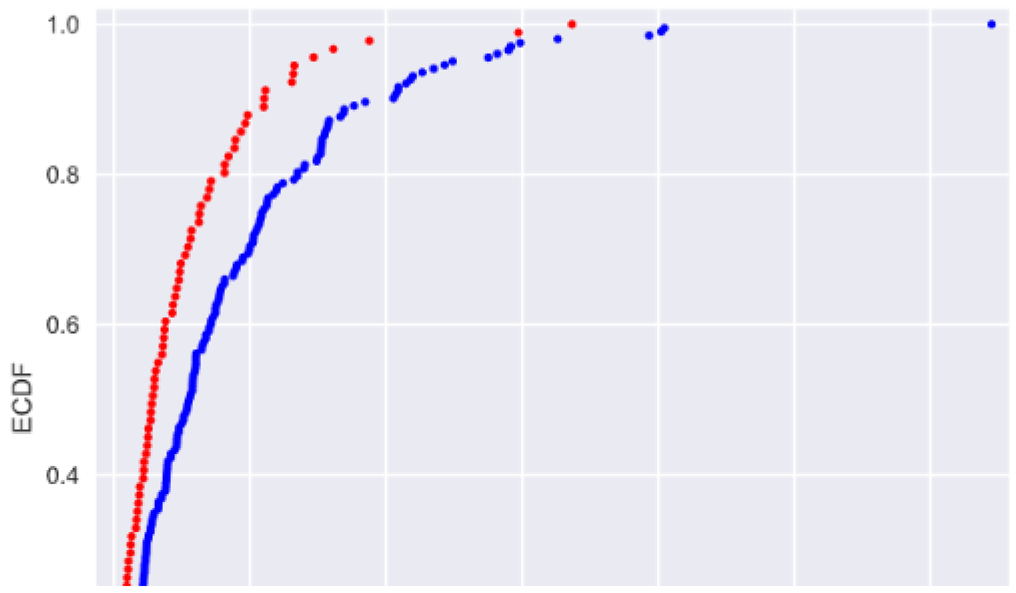
Observation: The p-value is 0.0001, which means that only one out of your 10,000 replicates had a result as extreme as the actual difference between the dead ball and live ball eras. This suggests strong statistical significance. Watch out, though, you could very well have gotten zero replicates that were as extreme as the observed value. This just means that the p-value is quite small, almost certainly smaller than 0.001.

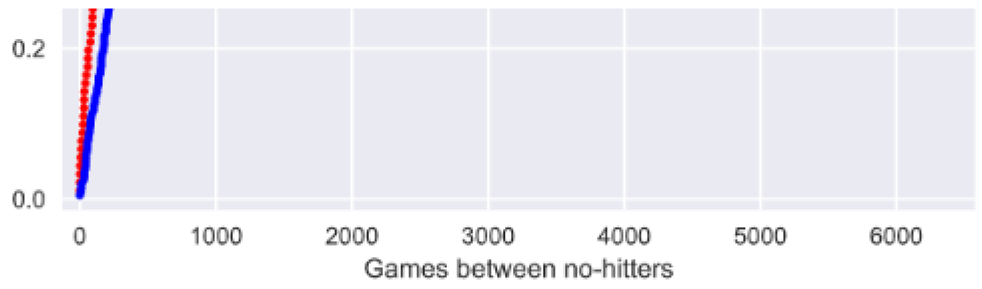
What should you have done first?

That was a nice hypothesis test you just did to check out whether the rule changes in 1920 changed the rate of no-hitters. But what *should* you have done with the data first? Answer: Performed EDA, perhaps plotting the ECDFs of inter-no-hitter times in the dead ball and live ball eras.

Always a good idea to do EDA first! Refer to the ECDFs plot below, visually, the null hypothesis that the distributions are the same is almost certainly **not** true.

```
# Create and plot ECDFs
x_1, y_1 = ecdf(nht_dead)
x_2, y_2 = ecdf(nht_live)
_ = plt.plot(x_1, y_1, marker='.', linestyle='none', color='red')
_ = plt.plot(x_2, y_2, marker='.', linestyle='none', color='blue')

# Label axes, set margin, and show plot
plt.margins(0.02)
_ = plt.xlabel('Games between no-hitters')
_ = plt.ylabel('ECDF')
plt.show()
```





Test of correlation

H0: there is no correlation between the 2 variables

H1: the 2 variables are correlated

Assume null hypothesis is true, simulate data

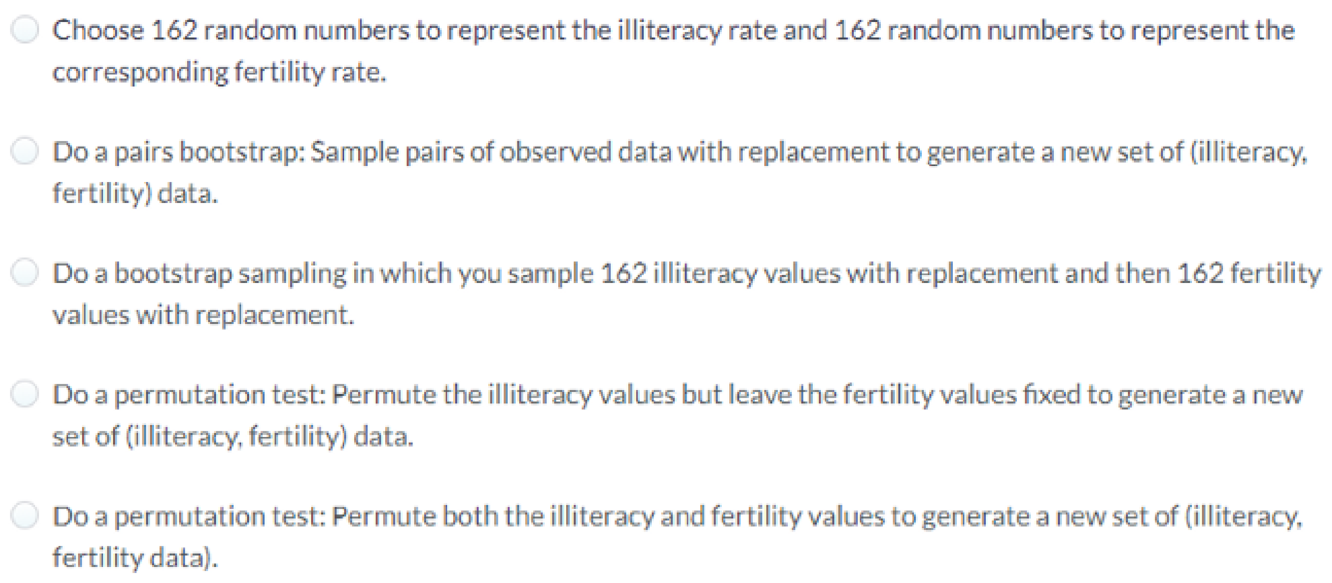
Use Pearson correlation **r** as test statistics

Compute the p-value as fraction of replicates that have **r** at least as large as observed

Simulating a null hypothesis concerning correlation

The observed correlation between female illiteracy and fertility in the data set of 162 countries may just be by chance; the fertility of a given country may actually be totally independent of its illiteracy. You will test this null hypothesis in the next exercise.

To do the test, you need to simulate the data assuming the null hypothesis is true. Of the following choices, which is the best way to do it?



- Choose 162 random numbers to represent the illiteracy rate and 162 random numbers to represent the corresponding fertility rate.
- Do a pairs bootstrap: Sample pairs of observed data with replacement to generate a new set of (illiteracy, fertility) data.
- Do a bootstrap sampling in which you sample 162 illiteracy values with replacement and then 162 fertility values with replacement.
- Do a permutation test: Permute the illiteracy values but leave the fertility values fixed to generate a new set of (illiteracy, fertility) data.
- Do a permutation test: Permute both the illiteracy and fertility values to generate a new set of (illiteracy, fertility data).

Answer: Do a permutation test: Permute the illiteracy values but leave the fertility values fixed to generate a new set of (illiteracy, fertility) data. This exactly simulates the null hypothesis and does so more **efficiently** than the last option. It is exact because it uses all data and eliminates any correlation because which illiteracy value pairs to which fertility value is shuffled.

The last option: Do a permutation test: Permute both the illiteracy and fertility values to generate a new set of (illiteracy, fertility data). This works perfectly, and is exact because it uses all data and eliminates any correlation because we shuffle which illiteracy value pairs to which fertility value. However, it is **not necessary**, and **computationally inefficient**, to permute both illiteracy and fertility.

Hypothesis test on Pearson correlation

The observed correlation between female illiteracy and fertility may just be by chance; the fertility of a given country may actually be totally independent of its illiteracy. You will test this hypothesis. To do so, permute the illiteracy values but leave the fertility values fixed. This simulates the hypothesis that they are totally independent of each other. For each permutation, compute the Pearson correlation coefficient and assess how many of your permutation replicates have a Pearson correlation coefficient greater than the observed one.

```
def pearson_r(x, y):
    """Compute Pearson correlation coeff between two arrays."""
    # Compute correlation matrix: corr_mat
    corr_mat = np.corrcoef(x, y)
    # Return entry [0,1]
    return corr_mat[0,1]

# Compute observed correlation: r_obs
r_obs = pearson_r(illiteracy, fertility)
# r_obs = 0.8041324026815344

# Initialize permutation replicates: perm_replicates
perm_replicates = np.empty(10000)

# Draw replicates
for i in range(10000):
    # Permute illiteracy measurments: illiteracy_permuted
    illiteracy_permuted = np.random.permutation(illiteracy)

    # Compute Pearson correlation
    perm_replicates[i] = pearson_r(illiteracy_permuted, fertility)
```

```
# Compute p-value: p
p = np.sum(perm_replicates >= r_obs) /len(perm_replicates)
print('p-val =', p)

<script.py> output:
    p-val = 0.0
```

Observation: The p-value is zero. In hacker statistics, this means that your p-value is very low, since you never got a single replicate in the 10,000 you took that had a Pearson correlation greater than the observed one. You could try increasing the number of replicates you take to continue to move the upper bound on your p-value lower and lower.

Do neonicotinoid insecticides have unintended consequences?

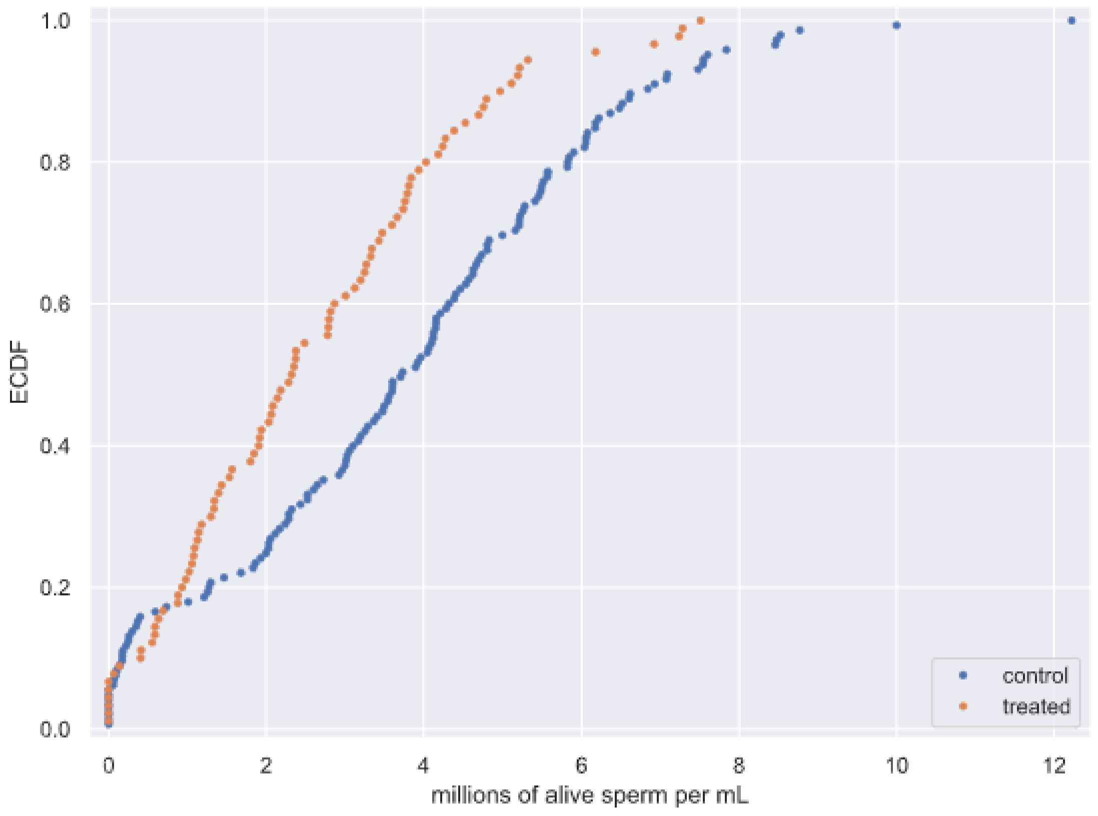
As a final exercise in hypothesis testing before we put everything together in our case study in the next chapter, you will investigate the effects of neonicotinoid insecticides on bee reproduction. These insecticides are very widely used in the United States to combat aphids and other pests that damage plants.

In this and the next exercise, you will study how the neonicotinoids pesticide treatment affected the count of live sperm per half milliliter of semen.

First, we will do EDA, as usual. Plot ECDFs of the alive sperm count for untreated bees (stored in the Numpy array `control`) and bees treated with pesticide (stored in the Numpy array `treated`).

```
# Compute x,y values for ECDFs
x_control, y_control = ecdf(control)
x_treated, y_treated = ecdf(treated)

# Plot the ECDFs
plt.plot(x_control, y_control, marker='.', linestyle='none')
plt.plot(x_treated, y_treated, marker='.', linestyle='none')

# Set the margins
plt.margins(0.02)

# Add a legend
plt.legend(('control', 'treated'), loc='lower right')
```

```
# Label axes and show plot
plt.xlabel('millions of alive sperm per mL')
plt.ylabel('ECDF')
plt.show()
```



Observation: The ECDFs show a pretty clear difference between the treatment and control; treated bees have fewer alive sperm. Let’s now do a hypothesis test in the next exercise.

Bootstrap hypothesis test on bee sperm counts

Now, you will test the following hypothesis: On average, male bees treated with neonicotinoid insecticide have the same number of active sperm per milliliter of semen than do untreated male bees. You will use the difference of means as your test statistic.

```
def bootstrap_replicate_1d(data, func):
    return func(np.random.choice(data, size=len(data)))

def draw_bs_reps(data, func, size=1):
    """Draw bootstrap replicates."""
    # Initialize array of replicates: bs_replicates
    bs_replicates = np.empty(size)
    # Generate replicates
```

```
        for i in range(size):
            bs_replicates[i] = bootstrap_replicate_1d(data, func)
        return bs_replicates

    # Compute the difference in mean sperm count: diff_means
    diff_means = np.mean(control) - np.mean(treated)

    # Compute mean of pooled data: mean_count
    mean_count = np.mean(np.concatenate((control, treated)))

    # Generate shifted data sets
    control_shifted = control - np.mean(control) + mean_count
    treated_shifted = treated - np.mean(treated) + mean_count

    # Generate bootstrap replicates
    bs_reps_control = draw_bs_reps(control_shifted,
                           np.mean, size=10000)
    bs_reps_treated = draw_bs_reps(treated_shifted,
                           np.mean, size=10000)

    # Get replicates of difference of means: bs_replicates
    bs_replicates = bs_reps_control - bs_reps_treated

    # Compute and print p-value: p
    p = np.sum(bs_replicates >= np.mean(control) - np.mean(treated)) \
                / len(bs_replicates)
    print('p-value =', p)

    <script.py> output:
        p-value = 0.0
```

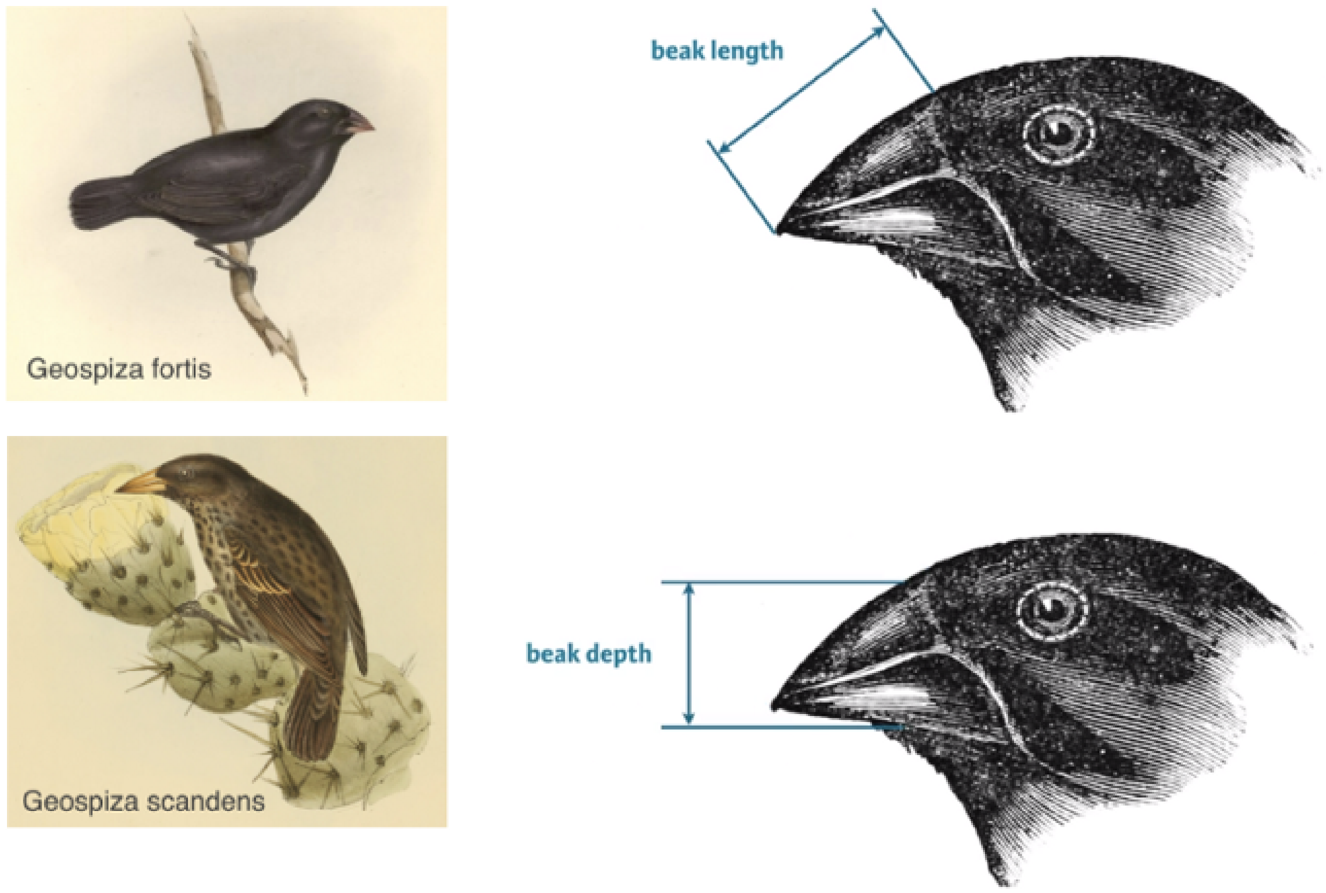
Observation: The p-value is small, most likely less than 0.0001, since you never saw a bootstrap replicated with a difference of means at least as extreme as what was observed. In fact, with 10 million replicates, the calculated p-value is `2e-05`.

. . .

Chapter 5. Putting it all together: a case study

Every year for the past 40-plus years, Peter and Rosemary Grant have gone to the Galápagos island of Daphne Major and collected data on Darwin's finches. Using your skills in statistical inference, you will spend this chapter with their data, and witness first hand, through data, evolution in action. It's an exhilarating way to end the course!

Finch beaks and the need for statistics



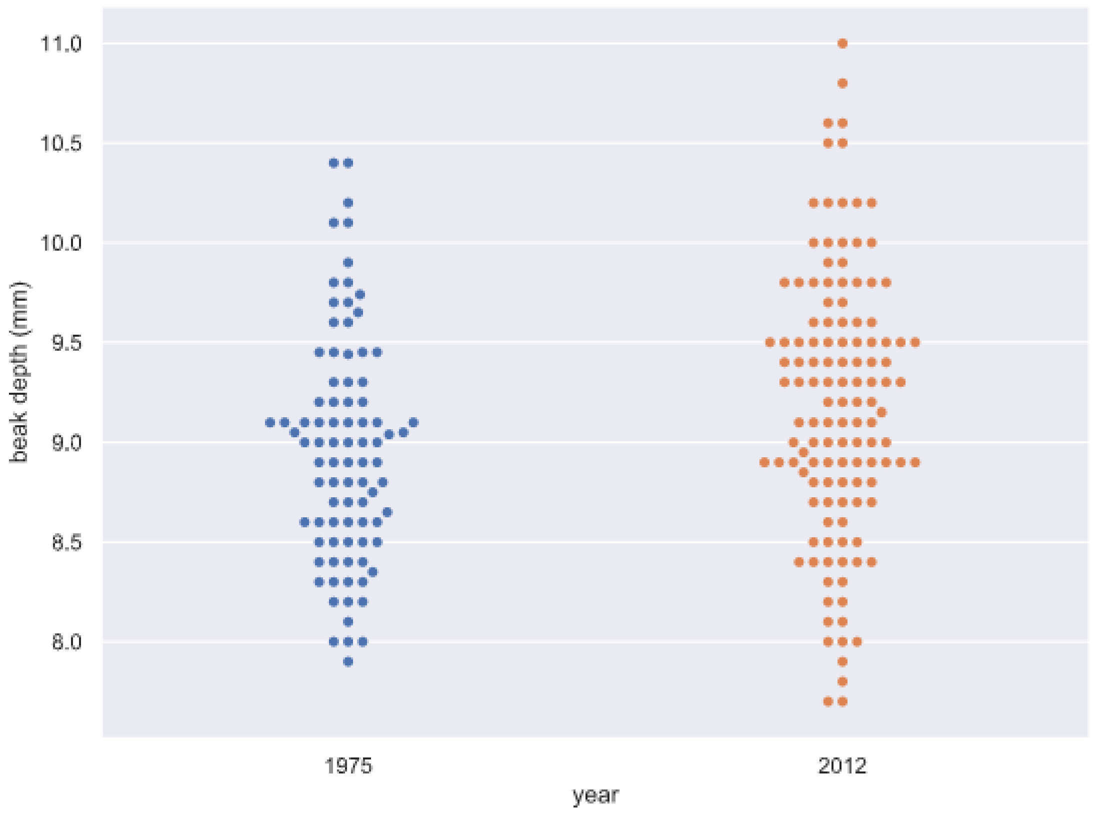
EDA of beak depths of Darwin's finches

For your first foray into the Darwin finch data, you will study how the beak depth (the distance, top to bottom, of a closed beak) of the finch species *Geospiza scandens* has changed over time. The Grants have noticed some changes of beak geometry depending on the types of seeds available on the island, and they also noticed that there was some interbreeding with another major species on Daphne Major, *Geospiza fortis*. These effects can lead to changes in the species over time.

In the next few problems, you will look at the beak depth of *G. scandens* on Daphne Major in 1975 and in 2012. To start with, let's plot all of the beak depth measurements in 1975 and 2012 in a bee swarm plot.

The data are stored in a pandas DataFrame called `df` with columns `'year'` and `'beak_depth'`. The units of beak depth are millimeters (mm).

```
df.head()
   beak_depth  year
0         8.4  1975
1         8.8  1975
2         8.4  1975
3         8.0  1975
4         7.9  1975

# Create bee swarm plot
_ = sns.swarmplot('year', 'beak_depth', data=df)

# Label the axes
_ = plt.xlabel('year')
_ = plt.ylabel('beak depth (mm)')

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



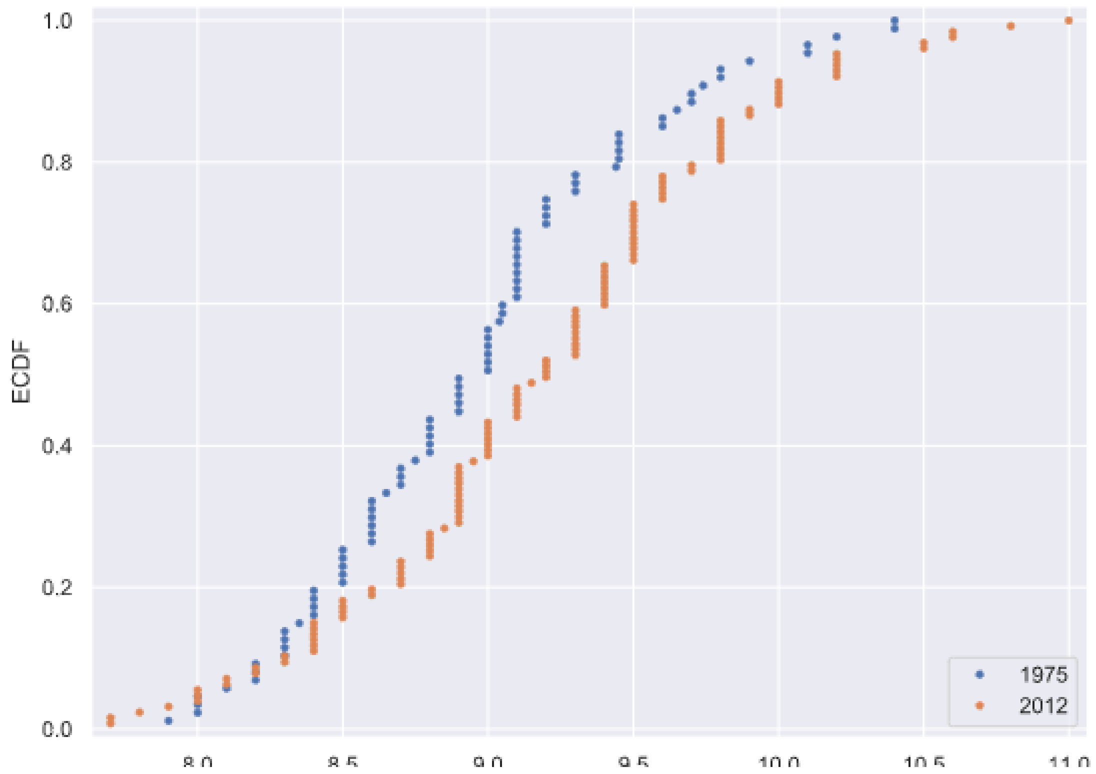
Observation: It is kind of hard to see if there is a clear difference between the 1975 and 2012 data set. Visually, it appears as though the mean of the 2012 data set might be slightly higher, and it might have a bigger variance.

ECDFs of beak depths

While bee swarm plots are useful, we found that ECDFs are often even better when doing EDA. Plot the ECDFs for the 1975 and 2012 beak depth measurements on the same plot.

The beak depths for the respective years has been stored in the NumPy arrays `bd_1975` and `bd_2012`.

```
# Compute ECDFs
x_1975, y_1975 = ecdf(bd_1975)
x_2012, y_2012 = ecdf(bd_2012)

# Plot the ECDFs
_ = plt.plot(x_1975, y_1975, marker='.', linestyle='none')
_ = plt.plot(x_2012, y_2012, marker='.', linestyle='none')

# Set margins
plt.margins(0.02)

# Add axis labels and legend
_ = plt.xlabel('beak depth (mm)')
_ = plt.ylabel('ECDF')
_ = plt.legend(('1975', '2012'), loc='lower right')

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



Observation: The differences are much clearer in the ECDF. The mean is larger in the 2012 data, and the variance does appear larger as well.

Parameter estimates of beak depths

Estimate the *difference* of the mean beak depth of the *G. scandens* samples from 1975 and 2012 and report a 95% confidence interval.

```
def draw_bs_reps(data, func, size=1):
    """Draw bootstrap replicates."""
    # Initialize array of replicates: bs_replicates
    bs_replicates = np.empty(size)
    # Generate replicates
    for i in range(size):
        bs_replicates[i] = bootstrap_replicate_1d(data, func)
    return bs_replicates

# Compute the difference of the sample means: mean_diff
mean_diff = np.mean(bd_2012) - np.mean(bd_1975)

# Get bootstrap replicates of means
bs_replicates_1975 = draw_bs_reps(bd_1975, np.mean, size=10000)
bs_replicates_2012 = draw_bs_reps(bd_2012, np.mean, size=10000)

# Compute samples of difference of means: bs_diff_replicates
bs_diff_replicates = bs_replicates_2012 - bs_replicates_1975

# Compute 95% confidence interval: conf_int
conf_int = np.percentile(bs_diff_replicates, [2.5, 97.5])

# Print the results
print('difference of means =', mean_diff, 'mm')
print('95% confidence interval =', conf_int, 'mm')

<script.py> output:
    difference of means = 0.22622047244094645 mm
    95% confidence interval = [0.05633521 0.39190544] mm
```

Hypothesis test: Are beaks deeper in 2012?

Your plot of the ECDF and determination of the confidence interval make it pretty clear that the beaks of *G. scandens* on Daphne Major have gotten deeper. But is it possible that this effect is just due to random chance? In other words, what is the probability that we would get the observed difference in mean beak depth if the means were the same?

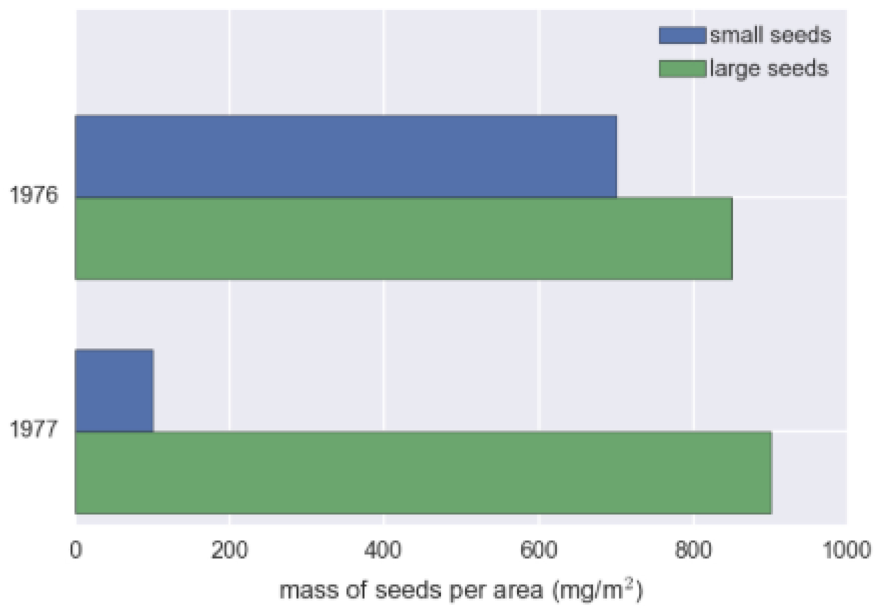
Be careful! The hypothesis we are testing is *not* that the beak depths come from the same distribution. For that we could use a permutation test. **The hypothesis is that the means are equal.** To perform this hypothesis test, we need to shift the two data sets so that they have the same mean and then use bootstrap sampling to compute the difference of means.

```
# Compute mean of combined data set: combined_mean
combined_mean = np.mean(np.concatenate((bd_1975, bd_2012)))

# Shift the samples
bd_1975_shifted = bd_1975 - np.mean(bd_1975) + combined_mean
bd_2012_shifted = bd_2012 - np.mean(bd_2012) + combined_mean

# Get bootstrap replicates of shifted data sets
bs_replicates_1975 = draw_bs_reps(bd_1975_shifted, np.mean,
size=10000)
bs_replicates_2012 = draw_bs_reps(bd_2012_shifted, np.mean,
size=10000)

# Compute replicates of difference of means: bs_diff_replicates
bs_diff_replicates = bs_replicates_2012 - bs_replicates_1975

# Compute the p-value
p = np.sum(bs_diff_replicates >= mean_diff) / len(bs_diff_replicates)

# Print p-value
print('p =', p)

<script.py> output:
    p = 0.0034
```

Observation: The p-value is 0.0034, which suggests that there is a statistically significant difference (ie, the means were not the same). But remember: it is very important to know how different they are! In the previous exercise, you got a difference of 0.2 mm between the means. You should combine this with the statistical significance. Changing by 0.2 mm in 37 years is substantial by evolutionary standards. If it kept changing at that rate, the beak depth would double in only 400 years.

Variation in beak shapes

The drought of winter in 1976/1977 resulted in the death of plants that produce small seeds on the island. The larger seeds require deeper beaks to crack them. So large beak birds survived and reproduced.



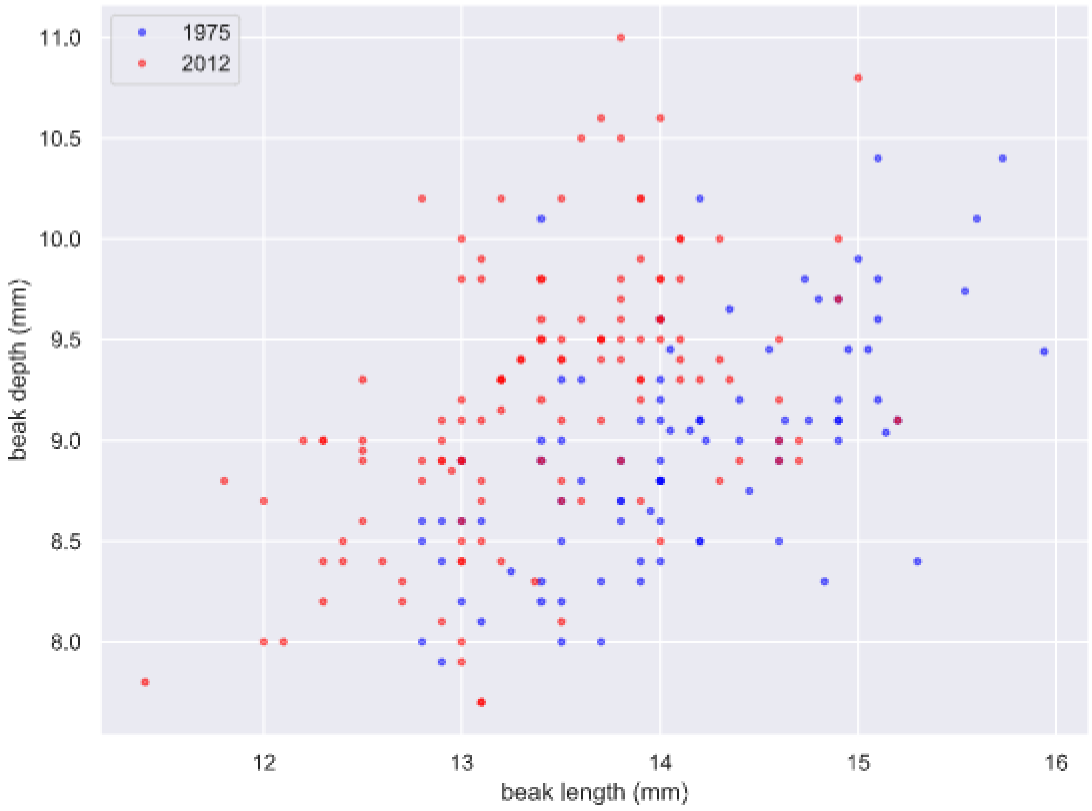
The beak length and beak depth might both change over time.

EDA of beak length and depth

The beak length data are stored as `bl_1975` and `bl_2012`, again with units of millimeters (mm). You still have the beak depth data stored in `bd_1975` and `bd_2012`. Make scatter plots of beak depth (y-axis) versus beak length (x-axis) for the 1975 and 2012 specimens.

```
# Make scatter plot of 1975 data
_ = plt.plot(bl_1975, bd_1975, marker='.',
             linestyle='None', color='blue', alpha=0.5)

# Make scatter plot of 2012 data
_ = plt.plot(bl_2012, bd_2012, marker='.',
            linestyle='None', color='red', alpha=0.5)

# Label axes and make legend
_ = plt.xlabel('beak length (mm)')
_ = plt.ylabel('beak depth (mm)')
_ = plt.legend(('1975', '2012'), loc='upper left')
```

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



Observation: Looking at the plot, we see that beaks got deeper (the red points are higher up in the y-direction), but not really longer. If anything, they got a bit shorter, since the red dots are to the left of the blue dots. So, it does not look like the beaks kept the same shape; they became shorter and deeper.

Linear regressions

Perform a linear regression for both the 1975 and 2012 data. Then, perform pairs bootstrap estimates for the regression parameters. Report 95% confidence intervals on the slope and intercept of the regression line.

The beak length data are stored as `bl_1975` and `bl_2012`, and the beak depth data is stored in `bd_1975` and `bd_2012`.

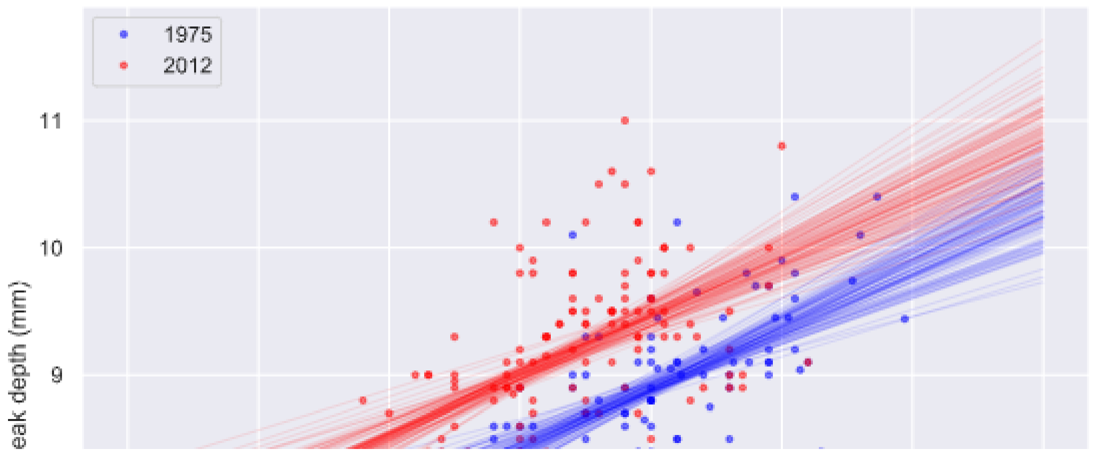
```
    def draw_bs_pairs_linreg(x, y, size=1):
        """Perform pairs bootstrap for linear regression."""
        # Set up array of indices to sample from: inds
        inds = np.arange(len(x))
        # Initialize replicates: bs_slope_reps, bs_intercept_reps
        bs_slope_reps = np.empty(size)
        bs_intercept_reps = np.empty(size)
        # Generate replicates
        for i in range(size):
            bs_inds = np.random.choice(inds, size=len(inds))
            bs_x, bs_y = x[bs_inds], y[bs_inds]
            bs_slope_reps[i], bs_intercept_reps[i] = np.polyfit(bs_x, 
    bs_y, deg=1)
        return bs_slope_reps, bs_intercept_reps

    # Compute the linear regressions
    slope_1975, intercept_1975 = np.polyfit(bl_1975, bd_1975, deg=1)
    slope_2012, intercept_2012 = np.polyfit(bl_2012, bd_2012, deg=1)

    # Perform pairs bootstrap for the linear regressions
    bs_slope_reps_1975, bs_intercept_reps_1975 = \
            draw_bs_pairs_linreg(bl_1975, bd_1975, size=1000)
    bs_slope_reps_2012, bs_intercept_reps_2012 = \
            draw_bs_pairs_linreg(bl_2012, bd_2012, size=1000)

    # Compute confidence intervals of slopes
    slope_conf_int_1975 = np.percentile(bs_slope_reps_1975, [2.5, 97.5])
    slope_conf_int_2012 = np.percentile(bs_slope_reps_2012, [2.5, 97.5])
    intercept_conf_int_1975 = np.percentile(bs_intercept_reps_1975, [2.5, 
    97.5])
    intercept_conf_int_2012 = np.percentile(bs_intercept_reps_2012, [2.5, 
    97.5])

    # Print the results
    print('1975: slope =', slope_1975,
          'conf int =', slope_conf_int_1975)
    print('1975: intercept =', intercept_1975,
          'conf int =', intercept_conf_int_1975)
    print('2012: slope =', slope_2012,
          'conf int =', slope_conf_int_2012)
    print('2012: intercept =', intercept_2012,
          'conf int =', intercept_conf_int_2012)

    <script.py> output:
    1975: slope = 0.465205169160593 conf int = [0.33851226 0.59306491]
    1975: intercept = 2.39087523658 conf int = [0.64892945 4.18037063]
    2012: slope = 0.462630358835313 conf int = [0.33137479 0.60695527]
    2012: intercept = 2.97724749823 conf int = [1.06792753 4.70599387]
```

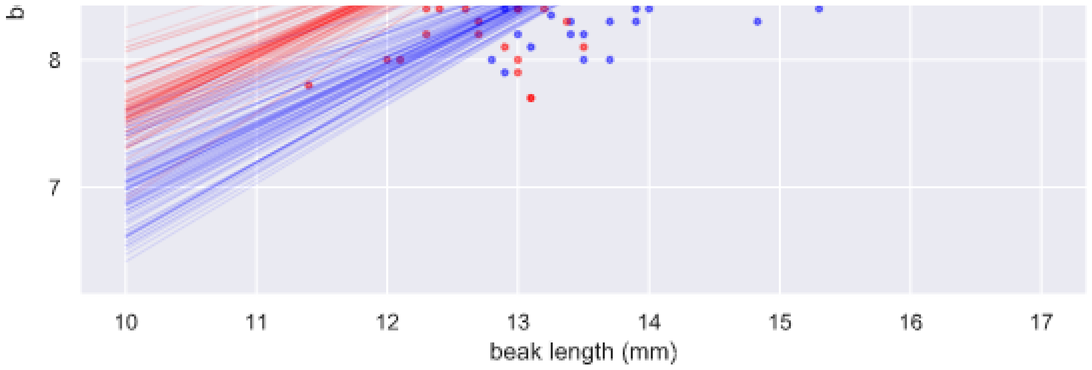
Observation: Apparently they have the same slope, but different intercepts.

Displaying the linear regression results

Now, you will display your linear regression results on the scatter plot, the code for which is already pre-written for you from your previous exercise. To do this, take the first 100 bootstrap samples (stored in `bs_slope_reps_1975`, `bs_intercept_reps_1975`, `bs_slope_reps_2012`, and `bs_intercept_reps_2012`) and plot the lines with `alpha=0.2` and `linewidth=0.5` keyword arguments to `plt.plot()`.

```
# Make scatter plot of 1975 data
_ = plt.plot(bl_1975, bd_1975, marker='.',
             linestyle='none', color='blue', alpha=0.5)

# Make scatter plot of 2012 data
_ = plt.plot(bl_2012, bd_2012, marker='.',
             linestyle='none', color='red', alpha=0.5)

# Label axes and make legend
_ = plt.xlabel('beak length (mm)')
_ = plt.ylabel('beak depth (mm)')
_ = plt.legend(('1975', '2012'), loc='upper left')

# Generate x-values for bootstrap lines: x
x = np.array([10, 17])

# Plot the bootstrap lines
for i in range(100):
    plt.plot(x, bs_slope_reps_1975[i] * x +
bs_intercept_reps_1975[i],
             linewidth=0.5, alpha=0.2, color='blue')
    plt.plot(x, bs_slope_reps_2012[i] * x +
bs_intercept_reps_2012[i],
             linewidth=0.5, alpha=0.2, color='red')

# Draw the plot again
plt.show()
```





Beak length to depth ratio

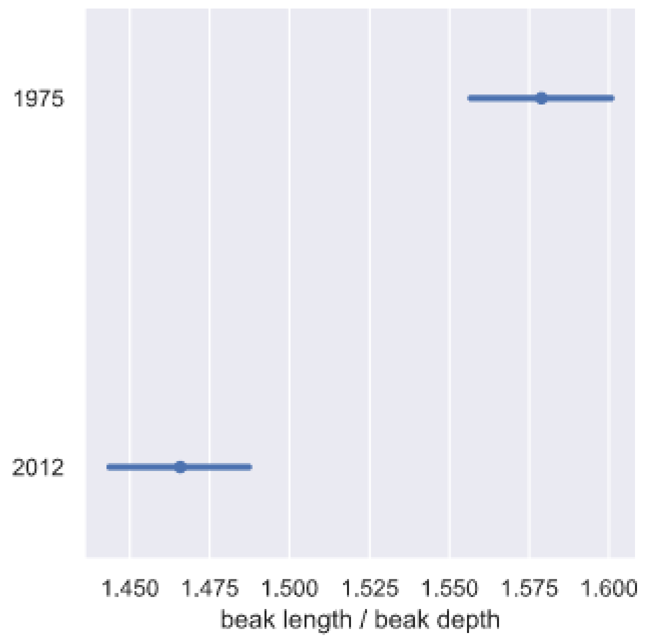
The linear regressions showed interesting information about the beak geometry. The slope was the same in 1975 and 2012, suggesting that for every millimeter gained in beak length, the birds gained about half a millimeter in depth in both years. However, if we are interested in the shape of the beak, we want to compare the *ratio* of beak length to beak depth. Let's make that comparison.

The data are stored in `bd_1975`, `bd_2012`, `bl_1975`, and `bl_2012`.

```
# Compute length-to-depth ratios
ratio_1975 = bl_1975 / bd_1975
ratio_2012 = bl_2012 / bd_2012

# Compute means
mean_ratio_1975 = np.mean(ratio_1975)
mean_ratio_2012 = np.mean(ratio_2012)

# Generate bootstrap replicates of the means
bs_replicates_1975 = draw_bs_reps(ratio_1975, np.mean, size=10000)
bs_replicates_2012 = draw_bs_reps(ratio_2012, np.mean, size=10000)

# Compute the 99% confidence intervals
conf_int_1975 = np.percentile(bs_replicates_1975, [0.5, 99.5])
conf_int_2012 = np.percentile(bs_replicates_2012, [0.5, 99.5])

# Print the results
print('1975: mean ratio =', mean_ratio_1975,
      'conf int =', conf_int_1975)
print('2012: mean ratio =', mean_ratio_2012,
      'conf int =', conf_int_2012)

<script.py> output:
1975: mean ratio = 1.57888237718 conf int = [1.5566880 1.6007350]
2012: mean ratio = 1.46583422768 conf int = [1.4436393 1.4872914]
```

How different is the ratio?

In the previous exercise, you computed the mean beak length to depth ratio with 99% confidence intervals for 1975 and for 2012, shown graphically in the plot below.



In addition to these results, what would you say about the ratio of beak length to depth?
Answer: The mean beak length-to-depth ratio decreased by about 0.1, or 7%, from 1975 to 2012. The 99% confidence intervals are not even close to overlapping, so this is a real change. The beak shape changed.
It is impossible to say if this is a real effect or just due to noise without computing a p-value. Let me compute the p-value and get back to you.

Calculation of heritability

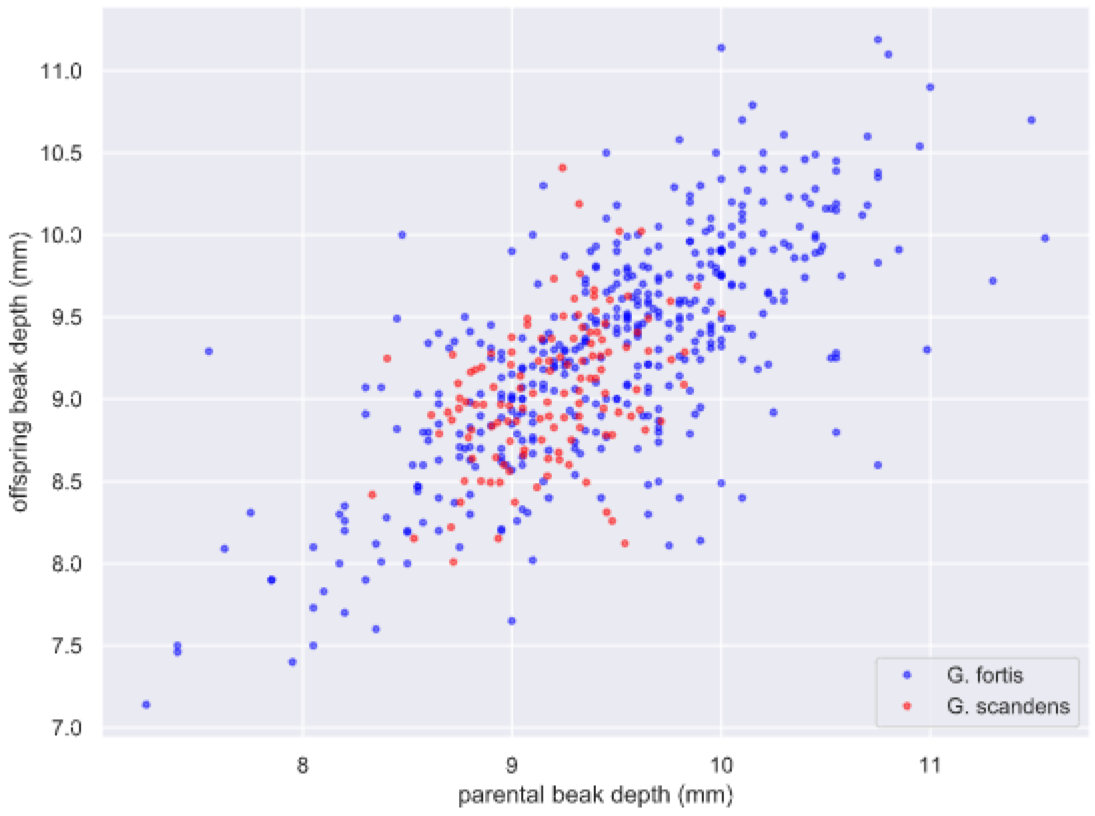
To investigate the tendency for parental traits to be inherited by offspring.

EDA of heritability

The array `bd_parent_scandens` contains the average beak depth (in mm) of two parents of the species `G. scandens`. The array `bd_offspring_scandens` contains the average beak depth of the offspring of the respective parents. The arrays `bd_parent_fortis` and `bd_offspring_fortis` contain the same information about measurements from *G. fortis* birds.

Make a scatter plot of the average offspring beak depth (y-axis) versus average parental beak depth (x-axis) for both species. Use the `alpha=0.5` keyword argument to help you see overlapping points.

```
# Make scatter plots
_ = plt.plot(bd_parent_fortis, bd_offspring_fortis,
             marker='.', linestyle='none', color='blue', alpha=0.5)
_ = plt.plot(bd_parent_scandens, bd_offspring_scandens,
             marker='.', linestyle='none', color='red', alpha=0.5)

# Label axes
_ = plt.xlabel('parental beak depth (mm)')
_ = plt.ylabel('offspring beak depth (mm)')

# Add legend
_ = plt.legend(('G. fortis', 'G. scandens'), loc='lower right')

# Show plot
plt.show()
```



Observation: It appears as though there is a stronger correlation in *G. fortis* than in *G. scandens*. This suggests that beak depth is more strongly inherited in *G. fortis*. We'll quantify this correlation next.

Correlation of offspring and parental data

In an effort to quantify the correlation between offspring and parent beak depths, we would like to compute statistics, such as the Pearson correlation coefficient, between parents and offspring. To get confidence intervals on this, we need to do a pairs bootstrap.

The function `draw_bs_pairs_linreg()` could do pairs bootstrap to get estimates for parameters derived from linear regression. Your task in this exercise is to make a new function with call signature `draw_bs_pairs(x, y, func, size=1)` that performs pairs bootstrap and computes a single statistic on pairs samples defined. The statistic of interest is computed by calling `func(bs_x, bs_y)` . In the next exercise, you will use `pearson_r` for `func` .

```
def draw_bs_pairs(x, y, func, size=1):
    """Perform pairs bootstrap for a single statistic."""

    # Set up array of indices to sample from: inds
    inds = np.arange(len(x))

    # Initialize replicates: bs_replicates
    bs_replicates = np.empty(size)

    # Generate replicates
    for i in range(size):
        bs_inds = np.random.choice(inds, size=len(inds))
        bs_x, bs_y = x[bs_inds], y[bs_inds]
        bs_replicates[i] = func(bs_x, bs_y)

    return bs_replicates
```

Pearson correlation of offspring and parental data

The Pearson correlation coefficient seems like a useful measure of how strongly the beak depth of parents are inherited by their offspring. Compute the Pearson correlation coefficient between parental and offspring beak depths for *G. scandens*. Do the same for *G. fortis*. Then, use the function you wrote in the last exercise to compute a 95% confidence interval using pairs bootstrap.

The data are stored in `bd_parent_scandens`, `bd_offspring_scandens`, `bd_parent_fortis`, and `bd_offspring_fortis`.

```
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 coefficients
r_scandens = pearson_r(bd_parent_scandens, bd_offspring_scandens)
r_fortis = pearson_r(bd_parent_fortis, bd_offspring_fortis)

# Acquire 1000 bootstrap replicates of Pearson r
bs_replicates_scandens = draw_bs_pairs(bd_parent_scandens,
bd_offspring_scandens, pearson_r, size=1000)

bs_replicates_fortis = draw_bs_pairs(bd_parent_fortis,
bd_offspring_fortis, pearson_r, size=1000)

# Compute 95% confidence intervals
conf_int_scandens = np.percentile(bs_replicates_scandens, [2.5,
97.5])
conf_int_fortis = np.percentile(bs_replicates_fortis, [2.5, 97.5])

# Print results
print('G. scandens:', r_scandens, conf_int_scandens)
print('G. fortis:', r_fortis, conf_int_fortis)

<script.py> output:
    G. scandens: 0.41170636294012 [0.26564228 0.54388972]
    G. fortis: 0.7283412395518487 [0.6694112  0.77840616]
```

Observation: From the confidence intervals, beak depth of the offspring of *G. fortis* parents is more strongly correlated with their offspring than their *G. scandens* counterparts.

Measuring heritability

The Pearson correlation coefficient is the ratio of the covariance to the geometric mean of the variances of the two data sets. This is a measure of the correlation between parents and offspring, but might not be the best estimate of heritability. If we stop and think, it makes more sense to define heritability as the ratio of the covariance between parent and offspring to the *variance of the parents alone*. In this exercise, you will estimate the heritability and perform a pairs bootstrap calculation to get the 95% confidence interval.

This exercise highlights a very important point. Statistical inference (and data analysis in general) is not a plug-n-chug enterprise. You need to think carefully about the questions you are seeking to answer with your data and analyze them appropriately. If you are interested in how heritable traits are, the quantity we defined as the heritability is more apt than the off-the-shelf statistic, the Pearson correlation coefficient.

The data are stored in `bd_parent_scandens`, `bd_offspring_scandens`, `bd_parent_fortis`, and `bd_offspring_fortis`.

```
def heritability(parents, offspring):
    """Compute the heritability from parent and offspring samples."""
    covariance_matrix = np.cov(parents, offspring)
    return covariance_matrix[0,1] / covariance_matrix[0,0]

# Compute the heritability
heritability_scandens = heritability(bd_parent_scandens,
bd_offspring_scandens)
heritability_fortis = heritability(bd_parent_fortis,
bd_offspring_fortis)

# Acquire 1000 bootstrap replicates of heritability
replicates_scandens = draw_bs_pairs(
        bd_parent_scandens, bd_offspring_scandens, heritability,
size=1000)

replicates_fortis = draw_bs_pairs(
        bd_parent_fortis, bd_offspring_fortis, heritability,
size=1000)

# Compute 95% confidence intervals
conf_int_scandens = np.percentile(replicates_scandens, [2.5, 97.5])
conf_int_fortis = np.percentile(replicates_fortis, [2.5, 97.5])

# Print results
print('G. scandens:', heritability_scandens, conf_int_scandens)
print('G. fortis:', heritability_fortis, conf_int_fortis)

<script.py> output:
    G. scandens: 0.54853408686859 [0.34395487 0.75638267]
    G. fortis: 0.7229051911438159 [0.64655013 0.79688342]
```

Observation: Again, *G. fortis* has stronger heritability than *G. scandens*. This suggests that the traits of *G. fortis* may be strongly incorporated into *G. scandens* by introgressive hybridization.

Is beak depth heritable at all in G. scandens?

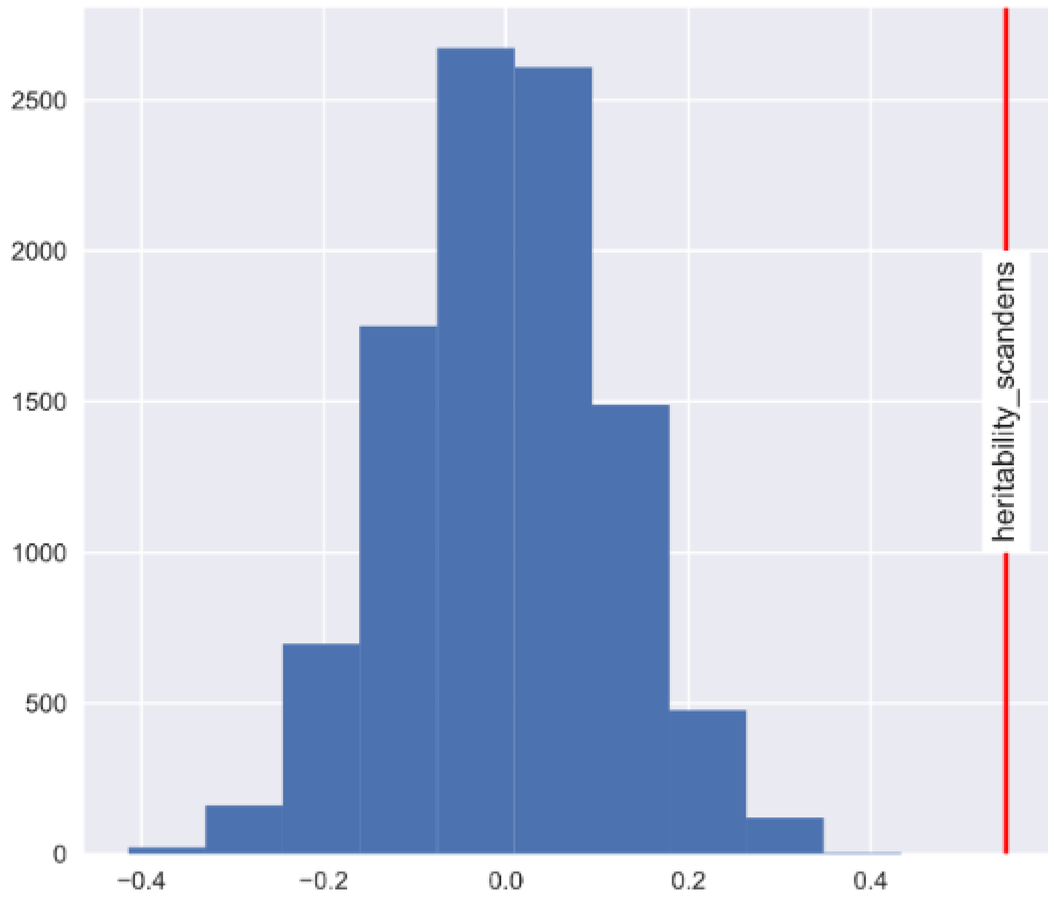
The heritability of beak depth in *G. scandens* seems low. It could be that this observed heritability was just achieved by chance and **beak depth is actually not really heritable in the species**. You will test that hypothesis here. To do this, you will do a **pairs permutation test**.

```
# Initialize array of replicates: perm_replicates
perm_replicates = np.empty(10000)

# Draw replicates
for i in range(10000):
    # Permute parent beak depths
    bd_parent_permuted = np.random.permutation(bd_parent_scandens)
    perm_replicates[i] = heritability(bd_parent_permuted,
                                      bd_offspring_scandens)

# Compute p-value: p
p = np.sum(perm_replicates >= heritability_scandens) /
len(perm_replicates)

# Print the p-value
print('p-val =', p)

<script.py> output:
    p-val = 0.0
```

Observation: The p-value is zero, which means that none of the 10,000 permutation pairs replicates you drew had a heritability high enough to match that which was observed. This strongly suggests that beak depth is heritable in *G. scandens*, just not as much as in *G. fortis*.

Refer to the histogram plot of the heritability replicates, see how extreme a value of heritability you might expect to happen by chance.

```
plt.hist(perm_replicates)
plt.axvline(x=heritability_scandens, color = 'red')
plt.text(heritability_scandens, 1500, 'heritability_scandens',
ha='center', va='center', rotation='vertical',
backgroundcolor='white')
plt.show()
```



Final thoughts

These are the **statistical thinking skills** acquired in this course:

1. Perform EDA

(a) Generate effective plots like ECDFs

(b) Compute summary statistics

2. Estimate parameters

(a) By optimisation, including linear regression

(b) Determine confidence intervals

3. Formulate and test statistical hypothesis

. . .

Reference: datasets used in this course:

- Anscombe data
- Bee sperm counts
- Female literacy and fertility
- Finch beaks (1975)
- Finch beaks (2012)
- Fortis beak depth heredity
- Frog tongue data
- Major League Baseball no-hitters
- Scandens beak depth heredity
- Sheffield Weather Station

Statistics Python Exploratory Data Analysis Data Visualization Probability