Chapter 1)

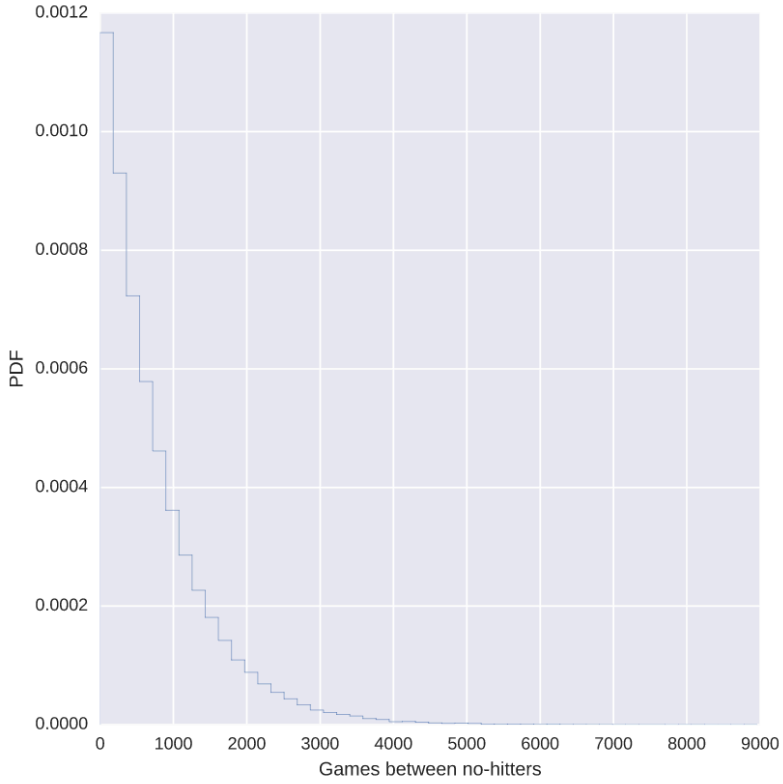
1)
How often do we get no-hitters?

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 *tau*\tau, the typical interval time. The value of the parameter ττ 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 ττ you found and plot the histogram as an approximation to the PDF.
NumPy, pandas, matlotlib.pyplot, and seaborn have been imported for you as `np`, `pd`, `plt`, and `sns`, respectively.

- Seed the random number generator with `42`.
- Compute the mean time (in units of number of games) between no-hitters.
- Draw 100,000 samples from an Exponential distribution with the parameter you computed from the mean of the inter-no-hitter times.
- Plot the theoretical PDF using `plt.hist()`. Remember to use keyword arguments `bins=50`, `normed=True`, and `histtype='step'`. Be sure to label your axes.

```
# 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()



2)

Do the data follow our story?

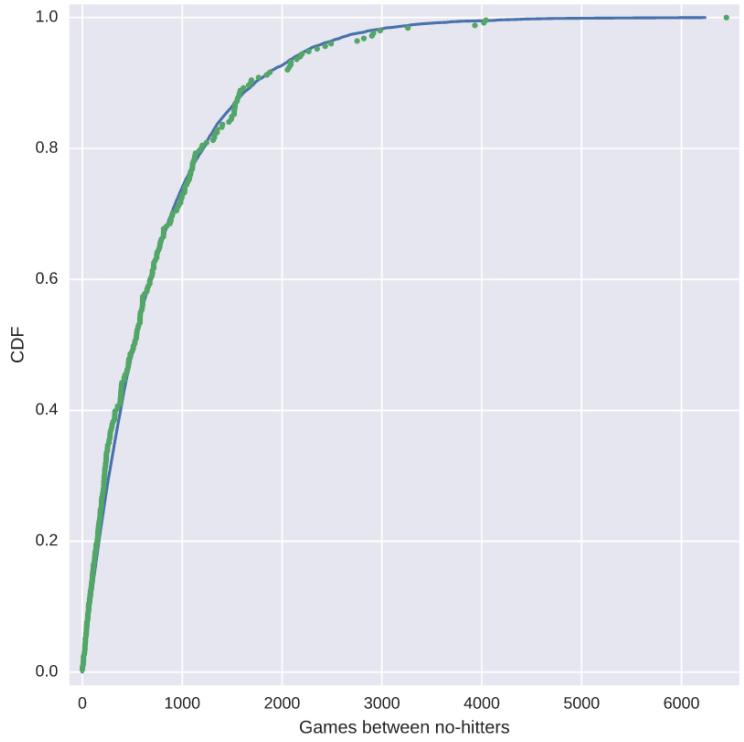
You have modeled no-hitters using an Exponential distribution. Create an ECDF of the real data. Overlay the theoretical CDF with the ECDF from the data. This helps you to verify that the Exponential distribution describes the observed data.
It may be helpful to remind yourself of the **function you created in the previous course** to compute the ECDF, as well as the code you wrote to **plot it**.

- Compute an ECDF from the actual time between no-hitters (`nohitter_times`). Use the `ecdf()` function you wrote in the prequel course.
- Create a CDF from the theoretical samples you took in the last exercise (`inter_nohitter_time`).
- Plot `x_theor` and `y_theor` as a line using `plt.plot()`. Then overlay the ECDF of the real data `x` and `y` as points. To do this, you have to specify the keyword arguments `marker = '.'` and `linestyle = 'none'` in addition to `x` and `y` inside `plt.plot()`.

- Set a 2% margin on the plot.

```
# 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()
```

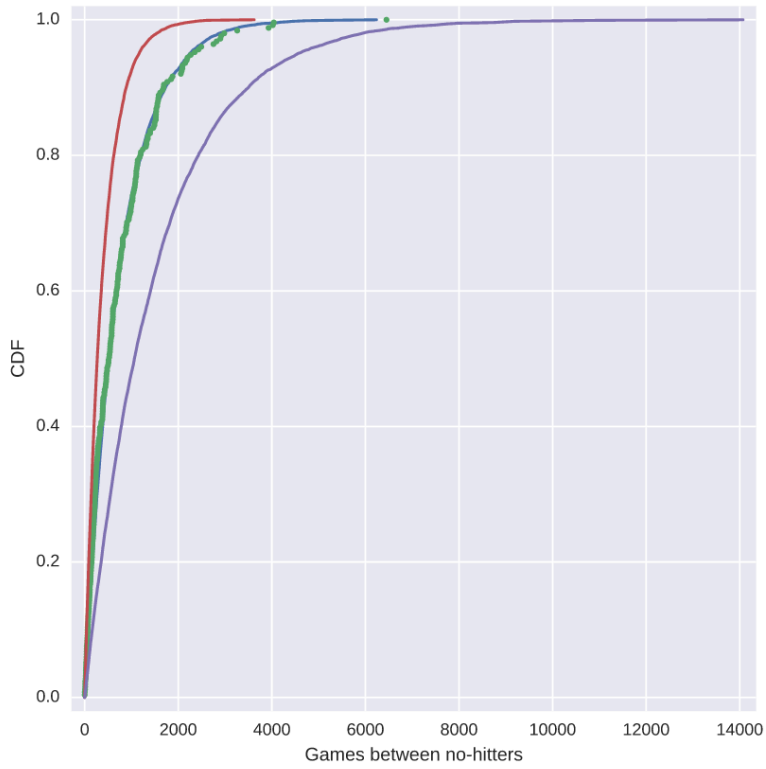


3)
How is this parameter optimal?

Now sample out of an exponential distribution with ττ being twice as large as the optimal ττ. Do it again for ττ 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 ττ you computed from the mean inter-no-hitter times is optimal in that it best reproduces the data.
Note: In this and all subsequent exercises, the random number generator is pre-seeded for you to save you some typing.

- Take `10000` samples out of an Exponential distribution with parameter $\tau_{1/2}$τ1/2 = `tau/2`.
- Take `10000` samples out of an Exponential distribution with parameter τ_2τ2 = `2*tau`.
- Generate CDFs from these two sets of samples using your `ecdf()` function.
- Add these two CDFs as lines to your plot

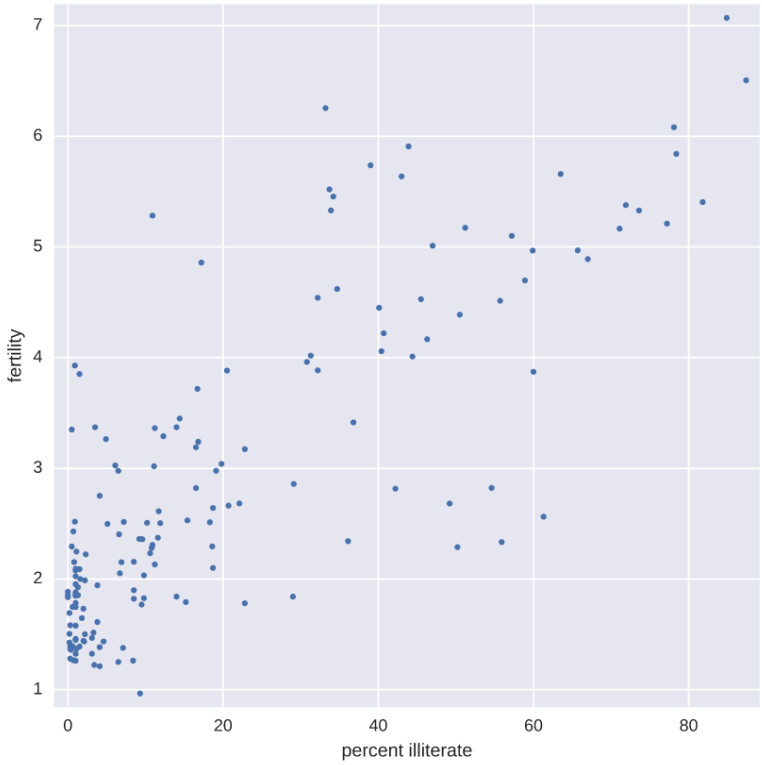
```
# Plot the theoretical CDFs
plt.plot(x_theor, y_theor)
plt.plot(x, y, marker='.', linestyle='none')
plt.margins(0.02)
plt.xlabel('Games between no-hitters')
plt.ylabel('CDF')

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

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



4)

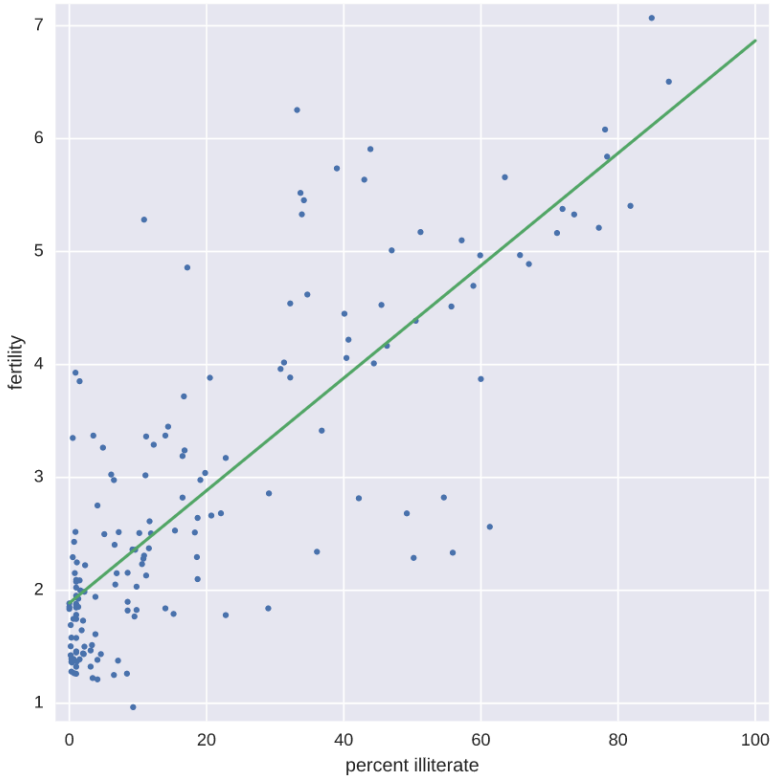
```
# Plot the illiteracy rate versus fertility
_ = plt.plot(illiteracy, fertility, marker='.', linestyle='none')

# 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))
```

0.804132402682



5)

```
# 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,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()
```



6)

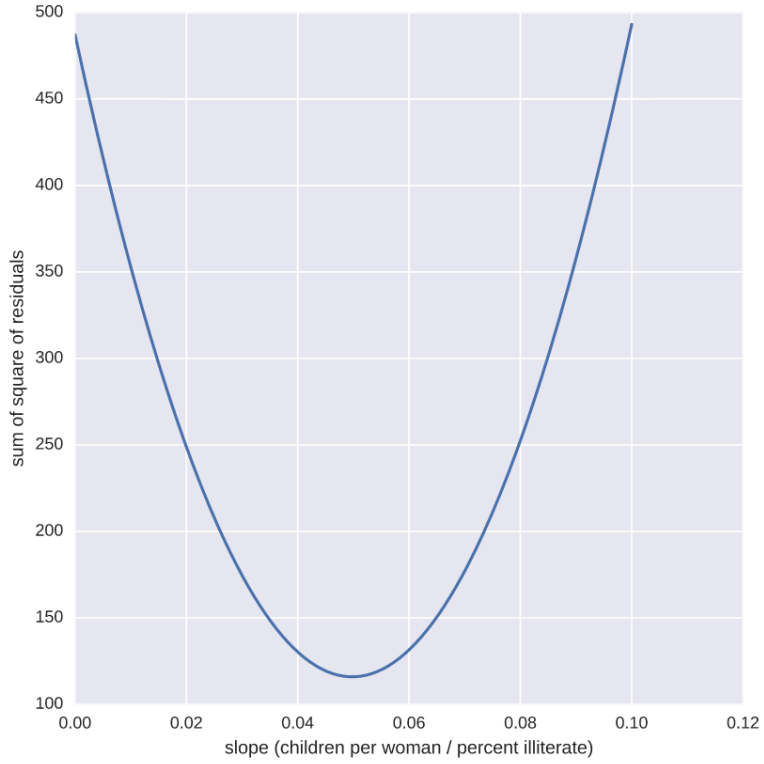
How is it optimal?

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?

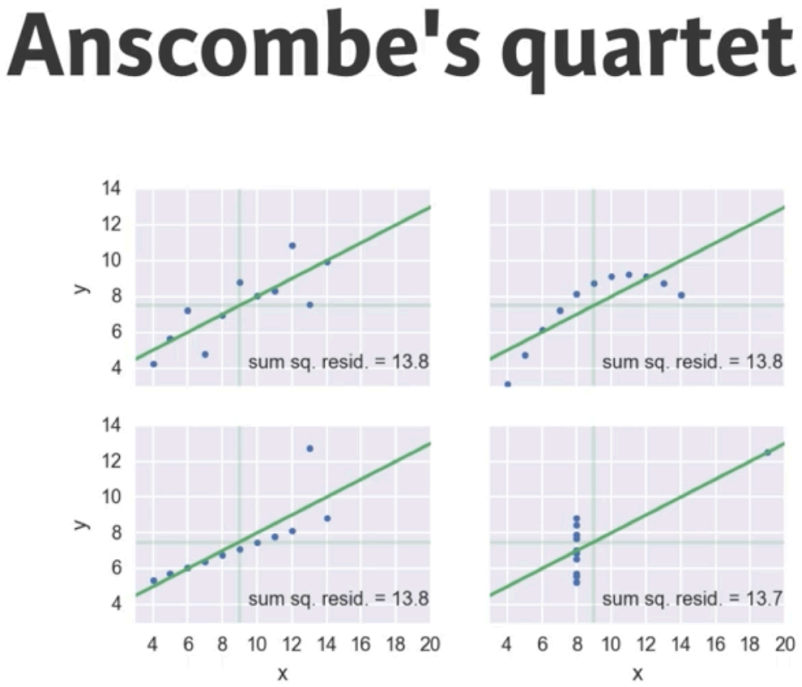
- Specify the values of the slope for which to compute the RSS.
  Use `np.linspace()` to get `200` points in the range between `0` and `0.1`. For example, to get `100` points in the range between `0` and `0.5`, you could use `np.linspace()` like so: `np.linspace(0, 0.5, 100)`.
- Initialize an array, `rss`, to contain the RSS using `np.empty_like()` and the array you created above. The `empty_like()` function returns a new array with the same shape and type as a given array (in this case, `a_vals`).
- Write a `for` loop to compute the sum of RSS of the slope. *Hint*: the RSS is given by `np.sum((y_data - a * x_data - b)**2)`. The variable `b` you computed in the last exercise is already in your namespace.
  Here, `fertility`is the `y_data` and `illiteracy` the `x_data`.
- Plot the RSS (`rss`) versus slope (`a_vals`).

```
# 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()
```



Note)



Chapter 2)

Note)

Resampling an array

Data:

```
[23.3, 27.1, 24.3, 25.7, 26.0]
```

Mean = 25.2

Resampled data:

```
[27.1, 26.0, 23.3, 25.7, 23.3]
```

Mean = 25.08

Resampling engine: `np.random.choice()`

```
In [1]: import numpy as np

In [2]: np.random.choice([1,2,3,4,5], size=5)
Out[2]: array([5, 3, 5, 5, 2])
```

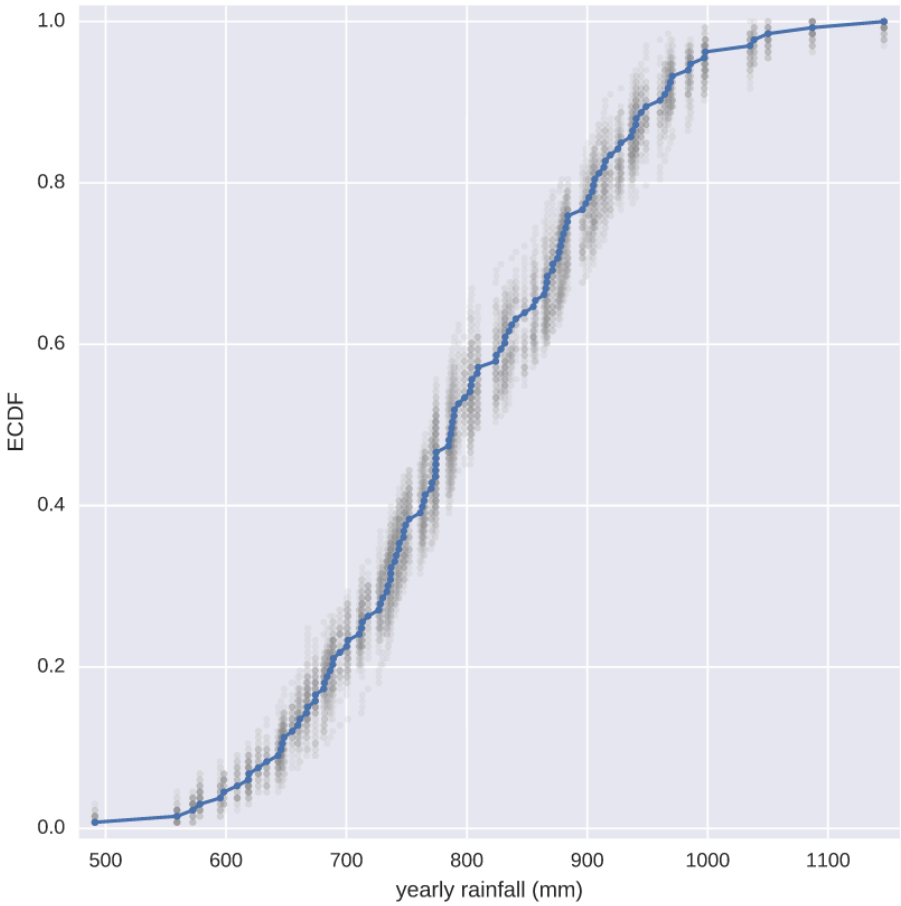
1)
Visualizing bootstrap samples

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.

- Write a `for` loop to acquire `50` bootstrap samples of the rainfall data and plot their ECDF.
    - Use `np.random.choice()` to generate a bootstrap sample from the NumPy array `rainfall`. Be sure that the `size` of the resampled array is `len(rainfall)`.
    - Use the function `ecdf()` that you wrote in the prequel to this course to generate the `x` and `y` values for the ECDF of the bootstrap sample `bs_sample`.
    - Plot the ECDF values. Specify `color='gray'` (to make gray dots) and `alpha=0.1` (to make them semi-transparent, since we are overlaying so many) in addition to the `marker='.'` and `linestyle='none'` keyword arguments.
- Use `ecdf()` to generate `x` and `y` values for the ECDF of the original rainfall data available in the array `rainfall`.

- Plot the ECDF values of the original data.

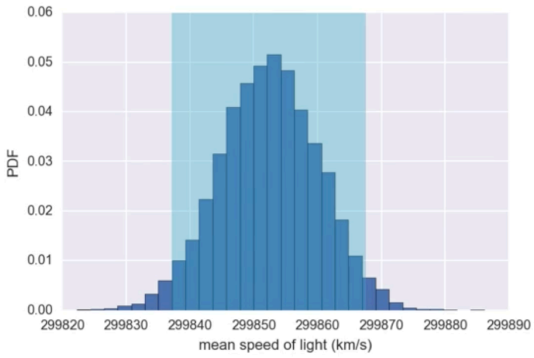
```
for _ in range(50):
    # Generate bootstrap sample: bs_sample
    bs_sample = np.random.choice(rainfall, size=len(rainfall))

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

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

# Make margins and label axes
plt.margins(0.02)
_ = plt.xlabel('yearly rainfall (mm)')
_ = plt.ylabel('ECDF')

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



Note)

Bootstrap confidence interval

```
In [1]: conf_int = np.percentile(bs_replicates, [2.5, 97.5])
Out[1]: array([ 299837.,  299868.])
```



2)
Generating many bootstrap replicates

The function `bootstrap_replicate_1d()` from the video is available in your namespace. 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.
For your reference, the `bootstrap_replicate_1d()` function is provided below:

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

- Define a function with call signature `draw_bs_reps(data, func, size=1)`.
  - Using `np.empty()`, initialize an array called `bs_replicates` of size `size`to hold all of the bootstrap replicates.
  - Write a `for` loop that ranges over `size` and computes a replicate using `bootstrap_replicate_1d()`. Refer to the exercise description above to see the function signature of `bootstrap_replicate_1d()`. Store the replicate in the appropriate index of `bs_replicates`.
  - Return the array of replicates `bs_replicates`

```
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
```

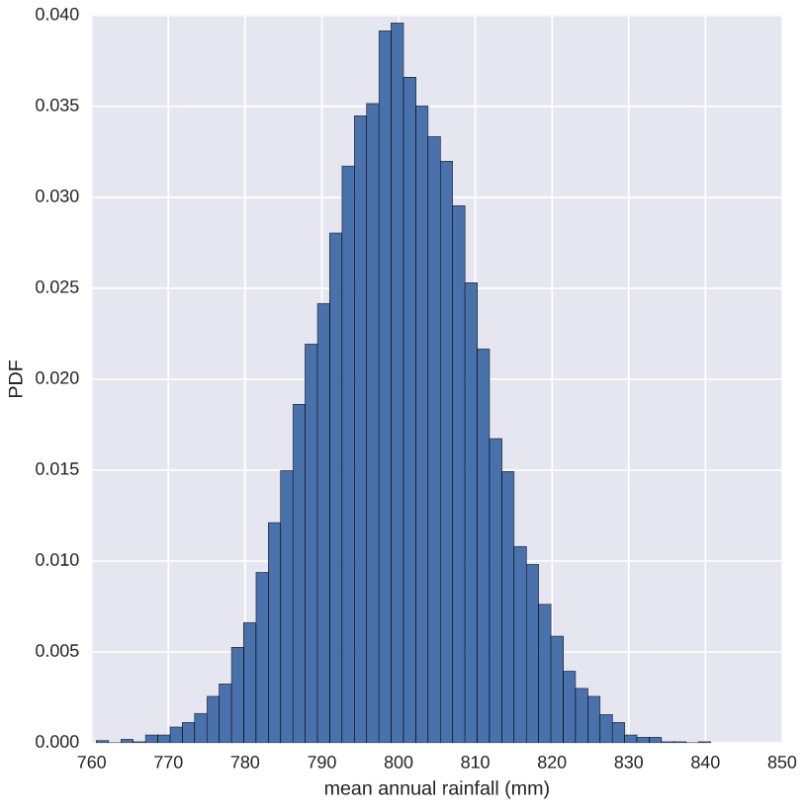
3)
Bootstrap replicates of the mean and the SEM

In this exercise, you will compute a bootstrap estimate of the probability distribution function of the mean annual rainfall at the Sheffield Weather Station. Remember, 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 has been pre-loaded for you into an array called `rainfall`.

- Draw `10000` bootstrap replicates of the **mean** annual rainfall using your `draw_bs_reps()` function and the `rainfall` array. *Hint*: Pass in `np.mean`for `func` to compute the mean.
    - As a reminder, `draw_bs_reps()` accepts 3 arguments: `data`, `func`, and `size`.
- Compute and print the standard error of the mean of `rainfall`.
    - The formula to compute this is `np.std(data) / np.sqrt(len(data))`.
- Compute and print the standard deviation of your bootstrap replicates `bs_replicates`.
- Make a histogram of the replicates using the `normed=True` keyword argument and `50` bins.

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



4)

Confidence intervals of rainfall data

A *confidence interval* gives bounds on the range of parameter values you might expect to get if we repeated 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 just take percentiles of your bootstrap replicates to get your confidence interval. Conveniently, you can use the `np.percentile()` function.

Using your 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` .

INSTRUCTIONS 50XP

Possible Answers

- (765, 776) mm/year — press 1
- (780, 821) mm/year — press 2
- (761, 817) mm/year — press 3
- (761, 841) mm/year — press 4

In [1]: np.percentile(bs_replicates,[2.5,97.5])

Out[1]: array([779.76992481, 820.95043233])

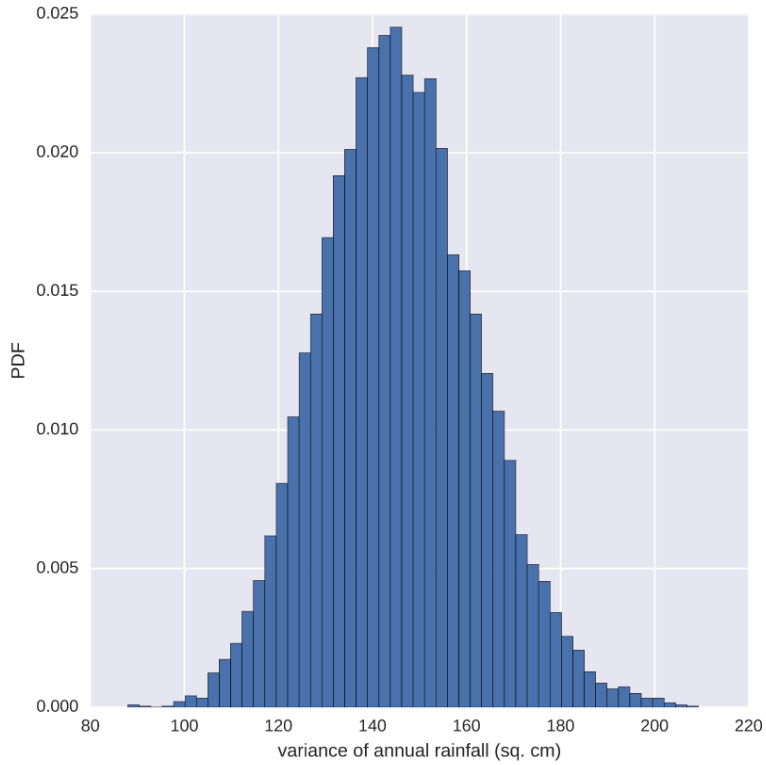
5)
Bootstrap replicates of other statistics

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.
Here, you will make use of the `draw_bs_reps()` function **you defined a few exercises ago**. It is provided below for your reference:

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

- Draw `10000` bootstrap replicates of the **variance** in annual rainfall using your `draw_bs_reps()` function. *Hint*: Pass in `np.var` for computing the variance.
- Divide your variance replicates - `bs_replicates` - by `100` to put the variance in units of square centimeters for convenience.
- Make a histogram of `bs_replicates` using the `normed=True` keyword argument and `50` bins.

```
# Generate 10,000 bootstrap replicates of the variance: bs_replicates
bs_replicates = draw_bs_reps(rainfall,np.var,size=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()
```



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.

6)

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 ττ. Plot a histogram of your replicates and report a 95% confidence interval.

- Generate `10000` bootstrap replicates of ττ from the `nohitter_times` data using your `draw_bs_reps()` function. Recall that the the optimal ττ is calculated as the **mean** of the data.
- Compute the 95% confidence interval using `np.percentile()` and passing in two arguments: The array `bs_replicates`, and the list of percentiles - in this case `2.5` and `97.5`.

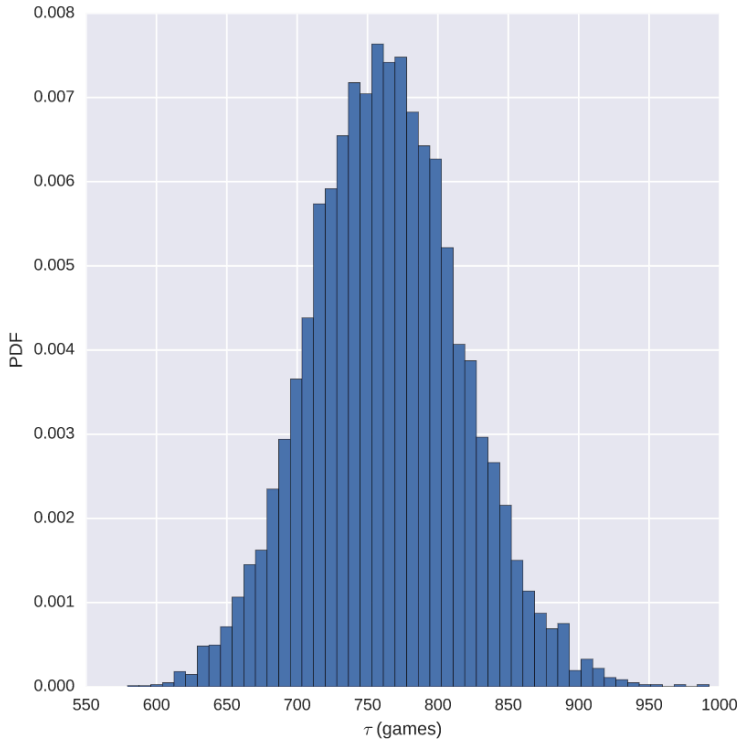
# Draw bootstrap replicates of the mean no-hitter time (equal to tau): bs_replicates
bs_replicates = draw_bs_reps(nohitter_times,np.mean,size=10000)

# 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')

95% confidence interval = [660.67280876 871.63077689] 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()
```



Note)

Generating a pairs bootstrap sample

```
In [1]: np.arange(7)
Out[1]: array([0, 1, 2, 3, 4, 5, 6])

In [1]: inds = np.arange(len(total_votes))

In [2]: bs_inds = np.random.choice(inds, len(inds))

In [3]: bs_total_votes = total_votes[bs_inds]

In [4]: bs_dem_share = dem_share[bs_inds]
```

7)
A function to do pairs bootstrap

As discussed in the video, 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.

- Define a function with call signature `draw_bs_pairs_linreg(x, y, size=1)`to perform pairs bootstrap estimates on linear regression parameters.
    - Use `np.arange()` to set up an array of indices going from `0` to `len(x)`. These are what you will resample and use them to pick values out of the `x`and `y` arrays.
    - Use `np.empty()` to initialize the slope and intercept replicate arrays to be of size `size`.
    - Write a `for` loop to:
        - Resample the indices `inds`. Use `np.random.choice()` to do this.
        - Make new xx and yy arrays `bs_x` and `bs_y` using the the resampled indices `bs_inds`. To do this, slice `x` and `y` with `bs_inds`.
        - Use `np.polyfit()` on the new xx and yy arrays and store the computed slope and intercept.

```
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
```

8)

Pairs bootstrap of literacy/fertility data

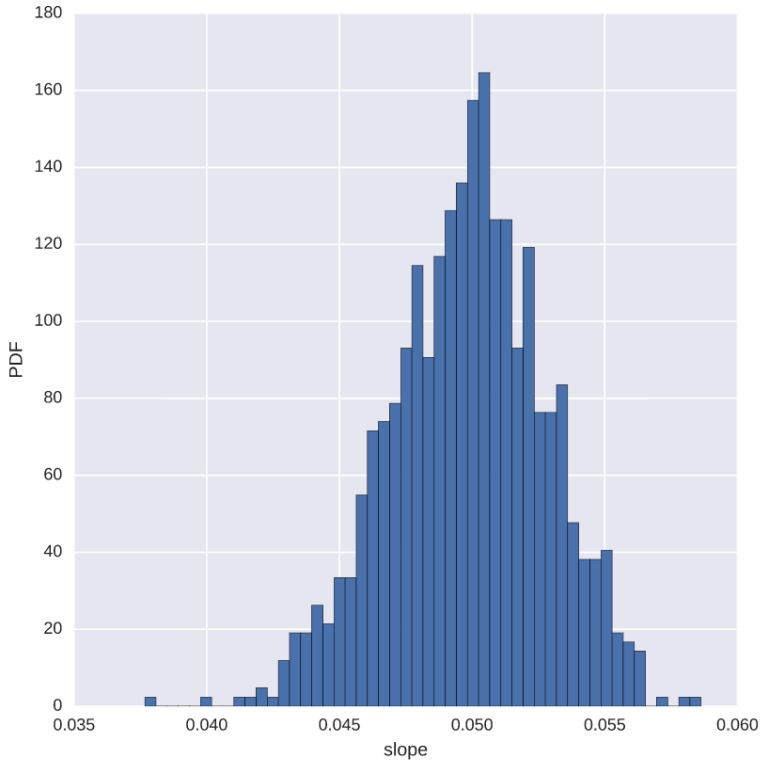
Using the function you just wrote, perform pairs bootstrap to plot a histogram describing the estimate of the slope from the illiteracy/fertility data. Also report the 95% confidence interval of the slope. The data is available to you in the NumPy arrays `illiteracy` and `fertility`.
As a reminder, `draw_bs_pairs_linreg()`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

- Use your `draw_bs_pairs_linreg()` function to take `1000` bootstrap replicates of the slope and intercept. The x-axis data is `illiteracy` and y-axis data is `fertility`.
- Compute and print the 95% bootstrap confidence interval for the slope

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

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

```
[ 0.04378061  0.0551616 ]
```

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



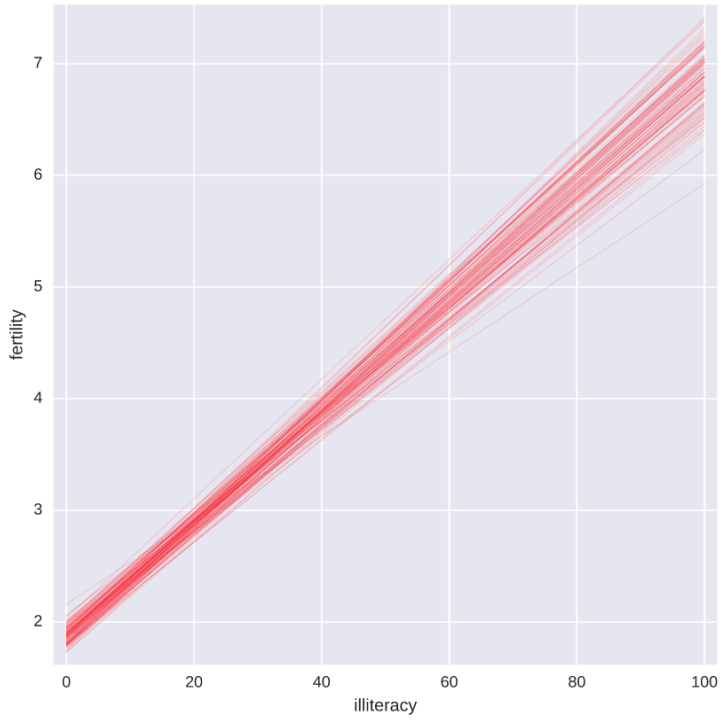
9)
Plotting bootstrap regressions

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 an array of xx-values consisting of `0` and `100` for the plot of the regression lines. Use the `np.array()` function for this.
- Write a `for` loop in which you plot a regression line with a slope and intercept given by the pairs bootstrap replicates. Do this for `100` lines.
    - When plotting the regression lines in each iteration of the `for` loop, recall the regression equation `y = a*x + b`.
      Here, `a` is `bs_slope_reps[i]`and `b` is `bs_intercept_reps[i]`.
    - Specify the keyword arguments `linewidth=0.5`, `alpha=0.2`, and `color='red'` in your call to `plt.plot()`.
- Make a scatter plot with `illiteracy` on the x-axis and `fertility` on the y-axis. Remember to specify the `marker='.'` and `linestyle='none'` keyword arguments.

- Label the axes, set a 2% margin, and show the plot

```
# 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.show()

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



Chapter 3)

Note)

Percent vote for Obama

	PA	OH	PA — OH difference
mean	45.5%	44.3%	1.2%
median	44.0%	43.7%	0.4%
standard deviation	9.8%	9.9%	—0.1%

Generating a permutation sample

```
In [1]: import numpy as np

In [2]: dem_share_both = np.concatenate(
   ...:                         (dem_share_PA, dem_share_OH))

In [3]: dem_share_perm = np.random.permutation(dem_share_both)

In [4]: perm_sample_PA = dem_share_perm[:len(dem_share_PA)]

In [5]: perm_sample_OH = dem_share_perm[len(dem_share_PA):]
```

1)
Generating a permutation sample

In the video, you learned that 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.
Remember, 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.

- Concatenate the two input arrays into one using `np.concatenate()`. Be sure to pass in `data1` and `data2` as one argument `(data1, data2)`.
- Use `np.random.permutation()` to permute the concatenated array.
- Store the first `len(data1)` entries of `permuted_data` as `perm_sample_1`and the last `len(data2)` entries of `permuted_data` as `perm_sample_2`. In practice, this can be achieved by using `:len(data1)` and `len(data1):` to slice `permuted_data`.
- Return `perm_sample_1` and `perm_sample_2`.

```
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: perm_sample_1, perm_sample_2
    perm_sample_1 = permuted_data[0:len(data1)]
    perm_sample_2 = permuted_data[len(data1):]

    return perm_sample_1, perm_sample_2
```

2)
Visualizing permutation sampling

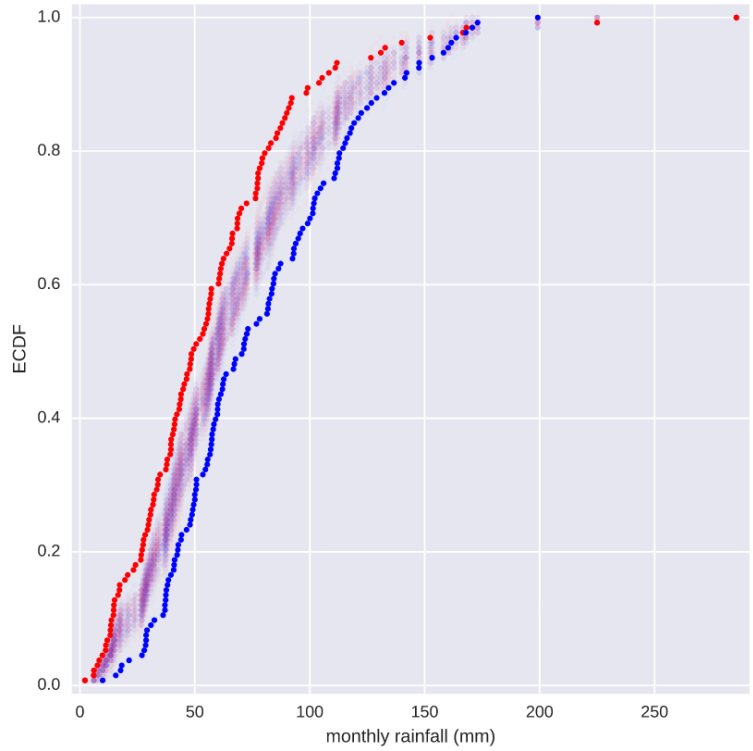
To help see how permutation sampling works, in this exercise you will generate permutation samples and look at them graphically.
We will use the Sheffield Weather Station data again, this time considering the monthly rainfall in July (a dry month) and November (a wet month). We expect these might be 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_july` 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)))`.

- Write a `for` loop to 50 generate permutation samples, compute their ECDFs, and plot them.
    - Generate a permutation sample pair from `rain_july` and `rain_november`using your `permutation_sample()` function.
    - Generate the `x` and `y` values for an ECDF for each of the two permutation samples for the ECDF using your `ecdf()` function.
    - Plot the ECDF of the first permutation sample (`x_1` and `y_1`) as dots. Do the same for the second permutation sample (`x_2` and `y_2`).
- Generate `x` and `y` values for ECDFs for the `rain_july` and `rain_november` data and plot the ECDFs using respectively the keyword arguments `color='red'` and `color='blue'`.

- Label your axes, set a 2% margin, and show your plot.

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

# Create and plot ECDFs from original data
```

```
x_1, y_1 = ecdf(rain_july)
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')

# Label axes, set margin, and show plot
plt.margins(0.02)
_ = plt.xlabel('monthly rainfall (mm)')
_ = plt.ylabel('ECDF')
plt.show()
```



Notice that 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. July and November rainfall are not identically distributed.

Note)

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
- **NOT** the probability that the null hypothesis is true

3)
Generating permutation replicates

As discussed in the video, a permutation replicate is a single value of a statistic computed from a permutation sample. As the `draw_bs_reps()` function **you wrote in chapter 2** 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.

- Define a function with this signature: `draw_perm_reps(data_1, data_2, func, size=1)`.
  - Initialize an array to hold the permutation replicates using `np.empty()`.
  - Write a `for` loop to:
    - Compute a permutation sample using your `permutation_sample()`function
    - Pass the sample into `func` to compute the replicate and store the result in your array of replicates.
  - Return the array of replicates.

```
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
        perm_sample_1, perm_sample_2 = permutation_sample(data_1,data_2)
```

```
        # Compute the test statistic
        perm_replicates[i] = func(perm_sample_1, perm_sample_2)

    return perm_replicates
```

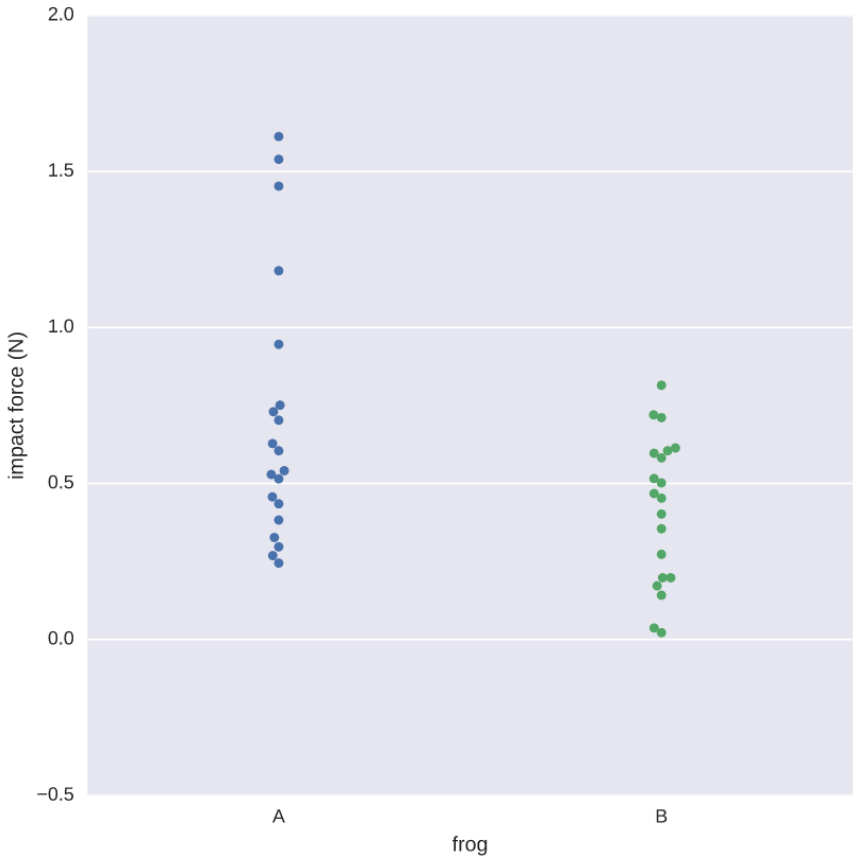
4)

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, remember, 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).

- Use `sns.swarmplot()` to make a bee swarm plot of the data by specifying the `x`, `y`, and `data` keyword arguments.
- Label your axes.
- Show the plot.

```
# 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()
```



Eyeballing it, 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.

5)
Permutation test on frog data

The average strike force of Frog A was 0.71 Newtons (N), and that of Frog 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.
For your convenience, the data has been stored in the arrays `force_a` and `force_b`.

- Define a function with call signature `diff_of_means(data_1, data_2)` that returns the differences in means between two data sets, mean of `data_1`minus mean of `data_2`.
- Use this function to compute the empirical difference of means that was observed in the frogs.

- Draw 10,000 permutation replicates of the difference of means.
- Compute the p-value.

```
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)
```

p-value = 0.0063

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. A p-value below 0.01 is typically said to be "statistically significant,", but: warning! warning! warning! You have computed a p-value; it is a number. I encourage you not to distill it to a yes-or-no phrase. p = 0.006 and p = 0.000000006 are both said to be "statistically significant," but they are definitely not the same!

Note)

Pipeline for hypothesis testing

- Clearly state the null hypothesis
- Define your test statistic
- Generate many sets of simulated data assuming the null hypothesis is true
- Compute the test statistic for each simulated data set
- The p-value is the fraction of your simulated data sets for which the test statistic is at least as extreme as for the real data

6)

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. Remember, 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.

- Translate the impact forces of Frog B such that its mean is 0.55 N.
- Use your `draw_bs_reps()` function to take 10,000 bootstrap replicates of the mean of your translated forces.
- Compute the p-value by finding the fraction of your bootstrap replicates that are less than the observed mean impact force of Frog B. Note that the variable of interest here is `force_b`.

```
# 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)
```

p = 0.0046

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

7)

A bootstrap test for identical distributions

In the video, we looked at a one-sample test, but we can do two sample tests. We can even test the same hypothesis that we tested with a permutation test: that the Frog A and Frog B have identically distributed impact forces. To do this test on two arrays with `n1` and `n2` entries, we do a very similar procedure as a permutation test. We concatenate the arrays, generate a bootstrap sample from it, and take the first `n1` entries of the bootstrap sample as belonging to the first data set and the last `n2` as belonging to the second. We then compute the test statistic, e.g., the difference of means, to get a bootstrap replicate. The p-value is the number of bootstrap replicates for which the test statistic is less than what was observed. Now, you will perform a bootstrap test of the hypothesis that Frog A and Frog B have identical distributions of impact forces using the difference of means test statistic.

The two arrays are available to you as `force_a` and `force_b`.

- Compute the observed difference in impact force using the `diff_of_means()`function you already wrote.
- Create an array that is the concatenation of `force_a` and `force_b`.
- Initialize the `bs_replicates` array using `np.empty()` to store 10,000 bootstrap replicates.
- Write a `for` loop to
  - Generate a bootstrap sample from the concatenated array.
  - Compute the difference in means between the first `len(force_a)` last `len(force_b)` entries of the bootstrap sample.
- Compute and print the p-value from your bootstrap replicates.

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

# Concatenate forces: forces_concat
forces_concat = np.concatenate((force_a,force_b))

# Initialize bootstrap replicates: bs_replicates
```

```
bs_replicates = np.empty(10000)

for i in range(10000):
    # Generate bootstrap sample
    bs_sample = np.random.choice(forces_concat, size=len(forces_concat))

    # Compute replicate
    bs_replicates[i] = diff_of_means(bs_sample[:len(force_a)],
                                     bs_sample[len(force_a):])

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

p-value = 0.0055

You may remember that we got p = 0.0063 from the permutation test, and here we got p = 0.0055. These are very close, and indeed the tests are testing the same thing. However, the permutation test *exactly* simulates the null hypothesis that the data come from the same distribution, whereas the bootstrap test *approximately* simulates it. As we will see, though, the bootstrap hypothesis test, while approximate, is more versatile.

8)

A two-sample bootstrap hypothesis test for difference of means.

You performed a one-sample bootstrap hypothesis test, which is impossible to do with permutation. Testing the hypothesis that two samples have the same distribution may be done with a bootstrap test, but a permutation test is preferred because it is more accurate (exact, in fact). But therein lies the limit of a permutation test; it is not very versatile. 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. This, too, is 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 already in your namespace.

- Compute the mean of all forces (from `forces_concat`) using `np.mean()`.
- Generate shifted data sets for *both* `force_a` *and* `force_b` such that the mean of each is the mean of the concatenated array of impact forces.

- Generate 10,000 bootstrap replicates of the mean each for the two shifted arrays.

- Compute the bootstrap replicates of the difference of means by subtracting the replicates of the shifted impact force of Frog B from those of Frog A.
- Compute and print the p-value from your bootstrap replicates.

```
# 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, size=10000)
bs_replicates_b = draw_bs_reps(force_b_shifted, np.mean, size=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)
```

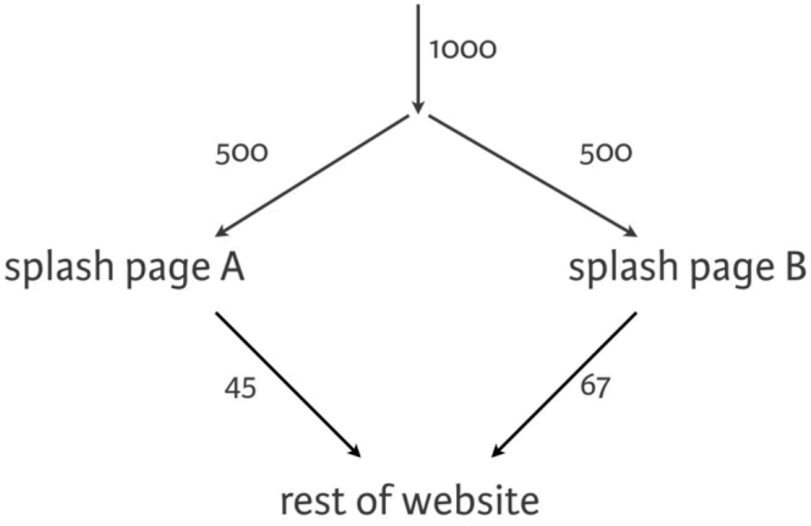
p-value = 0.0043

Not surprisingly, the more forgiving hypothesis, only that the means are equal as opposed to having identical distributions, gives a higher p-value. Again, it is important to carefully think about what question you want to ask. Are you only interested in the mean impact force, or the distribution of impact forces?

Chapter 4)

Note)

Is your redesign effective?



Permutation test of clicks through

```
In [1]: import numpy as np

In [2]: # clickthrough_A, clickthrough_B: arr. of 1s and 0s

In [3]: 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
   ...:

In [4]: diff_frac_obs = diff_frac(clickthrough_A,
   ...:                           clickthrough_B)
```

Permutation test of clicks through

```
In [1]: perm_replicates = np.empty(10000)

In [2]: for i in range(10000):
    ...:     perm_replicates[i] = permutation_replicate(
    ...:                 clickthrough_A, clickthrough_B, diff_frac)
    ...:

In [3]: p_value = np.sum(perm_replicates >= diff_frac_obs) / 10000

In [4]: p_value
Out[4]: 0.016
```

1)
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 yay. 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"

- Construct Boolean arrays, `dems` and `reps` that contain the votes of the respective parties; e.g., `dems` has 153 `True` entries and 91 `False` entries.
- Write a function, `frac_yay_dems(dems, reps)` that returns the fraction of Democrats that voted yay. The first input is an array of Booleans, Two inputs are required to use your `draw_perm_reps()` function, but the second is not used.
- Use your `draw_perm_reps()` function to draw 10,000 permutation replicates of the fraction of Democrat yay votes.

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

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

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

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

p-value = 0.0002

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.

2)

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 is actually 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." Since you will be using your `draw_perm_reps()` function in this exercise, it may be useful to remind yourself of its call signature: `draw_perm_reps(d1, d2, func, size=1)` or even **referring back** to the chapter 3 exercise in which you defined it.

- using `diff_of_means()`.
- Generate 10,000 permutation replicates of the difference of means using `draw_perm_reps()`.
- Compute and print the p-value.

```
# 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)
```

p-val = 0.0001

Your p-value if 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

Note)

Hypothesis test of correlation

- Posit null hypothesis: the two variables are completely uncorrelated
- Simulate data assuming null hypothesis is true
- Use Pearson correlation, ρ, as test statistic
- Compute p-value as fraction of replicates that have ρ at least as large as observed.

3)

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 to do it?

ANSWER THE QUESTION 50XP

Possible Answers

- Choose 162 random numbers to represent the illiteracy rate and 162 random numbers to represent the corresponding fertility rate. press 1
- Do a pairs bootstrap: Sample pairs of observed data with replacement to generate a new set of (illiteracy, fertility) data. press 2
- Do a bootstrap sampling in which you sample 162 illiteracy values with replacement and then 162 fertility values with replacement. press 3
- Do a permutation test: Permute the illiteracy values but leave the fertility values fixed to generate a new set of (illiteracy, fertility) data. press 4
- Do a permutation test: Permute both the illiteracy and fertility values to generate a new set of (illiteracy, fertility data). press 5

4)

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.
The function `pearson_r()` that you **wrote in the prequel to this course** for computing the Pearson correlation coefficient is already in your name space.

- Compute the observed Pearson correlation between `illiteracy` and `fertility`.

- Initialize an array to store your permutation replicates.
- Write a `for` loop to draw 10,000 replicates:
    - Permute the `illiteracy` measurements using `np.random.permutation()`.
    - Compute the Pearson correlation between the permuted illiteracy array, `illiteracy_permuted`, and `fertility`.

- Compute and print the p-value from the replicates.

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

# 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)
```

```
p-val = 0.0
```

You got a p-value of 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.

5)

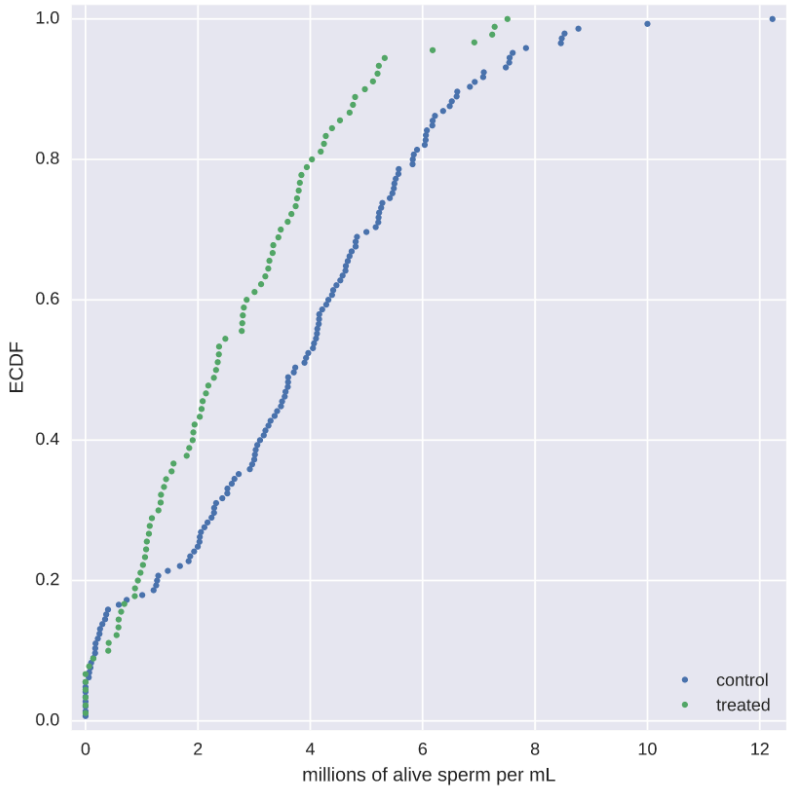
Do neonicotinoid insecticides have unintended consequences?

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 a recent study, Straub, et al. (***Proc. Roy. Soc. B*, 2016**) investigated the effects of neonicotinoids on the sperm of pollinating bees. In this and the next exercise, you will study how the 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`).

- Use your `ecdf()` function to generate `x,y` values from the `control` and `treated` arrays for plotting the ECDFs.
- Plot the ECDFs on the same plot.

```
# 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()
```



The ECDFs show a pretty clear difference between the treatment and control; treated bees have fewer alive sperm

6)

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.

For your reference, the call signature for the `draw_bs_reps()` function **you wrote in chapter 2** is `draw_bs_reps(data, func, size=1)`.

- Compute the mean alive sperm count of `control` minus that of `treated`.
- Compute the mean of all alive sperm counts. To do this, first concatenate `control` and `treated` and take the mean of the concatenated array.
- Generate shifted data sets for both `control` and `treated` such that the shifted data sets have the same mean. This has already been done for you.
- Generate 10,000 bootstrap replicates of the mean each for the two shifted arrays. Use your `draw_bs_reps()` function.

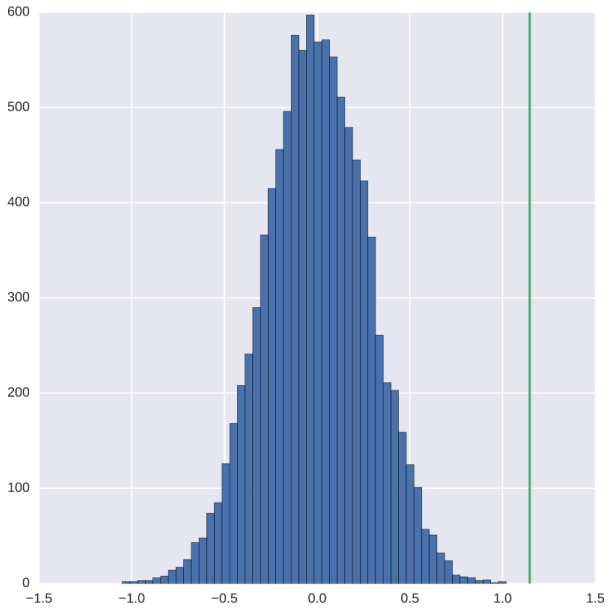
- Compute the bootstrap replicates of the difference of means.

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)
```

p-value = 0.0

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, when I did the calculation with 10 million replicates, I got a p-value of 2e-05



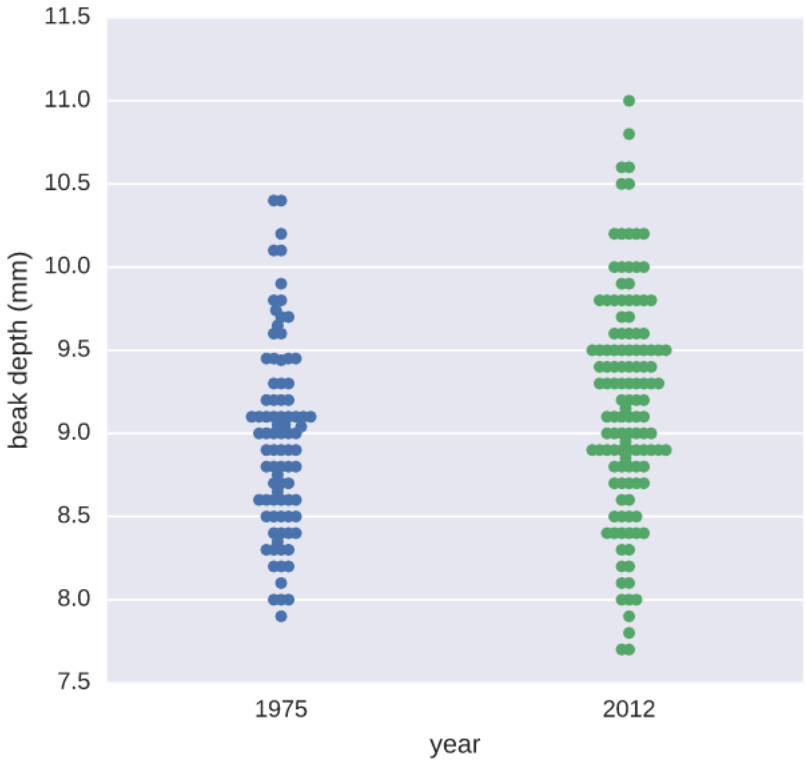
Chapter 5)

1)
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).

- Create the beeswarm plot.
- Label the axes.
- Show the plot.

```
# 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()
```



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

2)

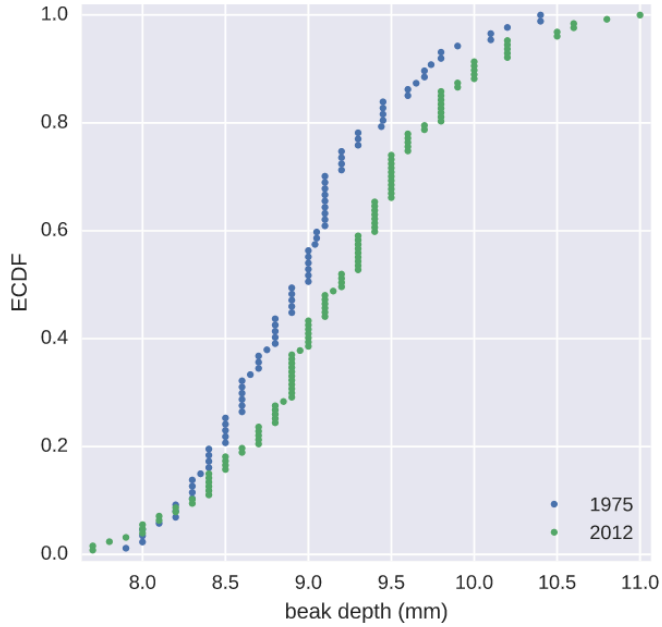
ECDFs of beak depths

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.
For your convenience, the beak depths for the respective years has been stored in the NumPy arrays `bd_1975` and `bd_2012`.

- Compute the ECDF for the 1975 and 2012 data.
- Plot the two ECDFs.
- Set a 2% margin and add axis labels and a legend to the plot.

```
# 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()
```



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

3)

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.
Since in this exercise you will use the `draw_bs_reps()` function you wrote in chapter 2, it may be helpful to **refer back to it**.

- Compute the difference of the sample means.
- Take 10,000 bootstrap replicates of the mean for the 1975 beak depths using your `draw_bs_reps()` function. Also get 10,000 bootstrap replicates of the mean for the 2012 beak depths.
- Subtract the 1975 replicates from the 2012 replicates to get bootstrap replicates of the difference of means.
- Use the replicates to compute the 95% confidence interval.

```
# 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')
```

```
difference of means = 0.226220472441 mm
95% confidence interval = [ 0.05633521  0.39190544] mm
```

4)

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.

- Make a concatenated array of the 1975 and 2012 beak depths and compute and store its mean.
- Shift `bd_1975` and `bd_2012` such that their means are equal to the one you just computed for the combined data set.

- Take 10,000 bootstrap replicates of the mean each for the 1975 and 2012 beak depths.

- Subtract the 1975 replicates from the 2012 replicates to get bootstrap replicates of the difference.
- Compute and print the p-value. The observed difference in means you computed in the last exercise is still in your namespace as `mean_diff`.

```
# 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)
```

p = 0.0034

We get a p-value of 0.0034, which suggests that there is a statistically significant difference. 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.

5)

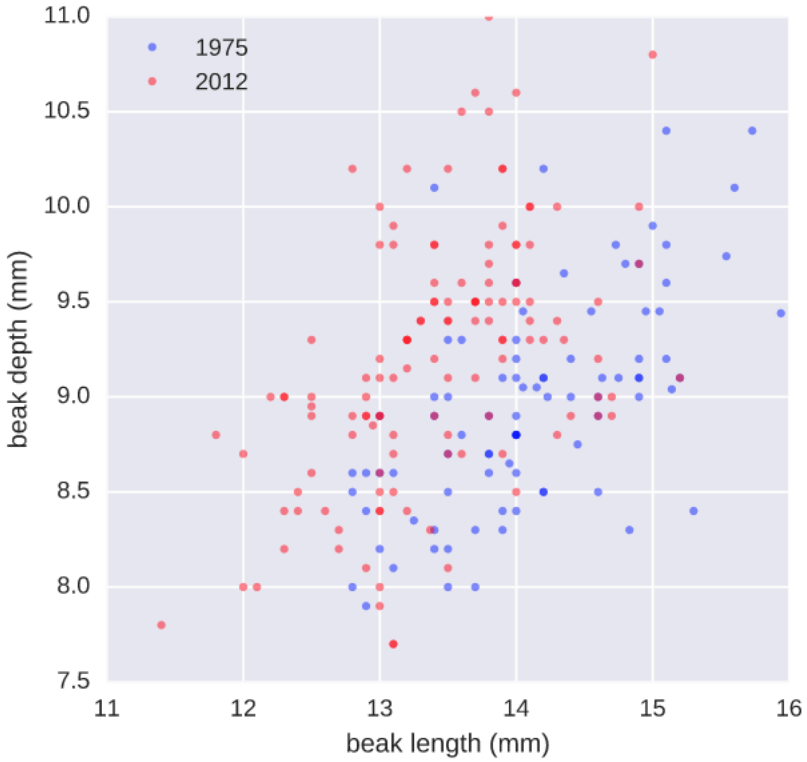
EDA of beak length and depth

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 a scatter plot of the 1975 data. Use the `color='blue'` keyword argument. Also use an `alpha=0.5` keyword argument to have transparency in case data points overlap.
- Do the same for the 2012 data, but use the `color='red'` keyword argument.
- Add a legend and label the axes.
- Show your 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')

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



In 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

6)
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.
You will use the `draw_bs_pairs_linreg()` function you **wrote back in chapter 2**. As a reminder, its call signature is `draw_bs_pairs_linreg(x, y, size=1)`, and it returns `bs_slope_reps` and `bs_intercept_reps`.

- Compute the slope and intercept for both the 1975 and 2012 data sets.

- Obtain 1000 pairs bootstrap samples for the linear regressions using your `draw_bs_pairs_linreg()` function.
- Compute 95% confidence intervals for the slopes and the intercepts.

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

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

# 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)
```

```
1975: slope = 0.465205169161 conf int = [ 0.33851226  0.59306491]
1975: intercept = 2.39087523658 conf int = [ 0.64892945  4.18037063]
2012: slope = 0.462630358835 conf int = [ 0.33137479  0.60695527]
2012: intercept = 2.97724749824 conf int = [ 1.06792753  4.70599387]
```

It looks like they have the same slope, but different intercepts.

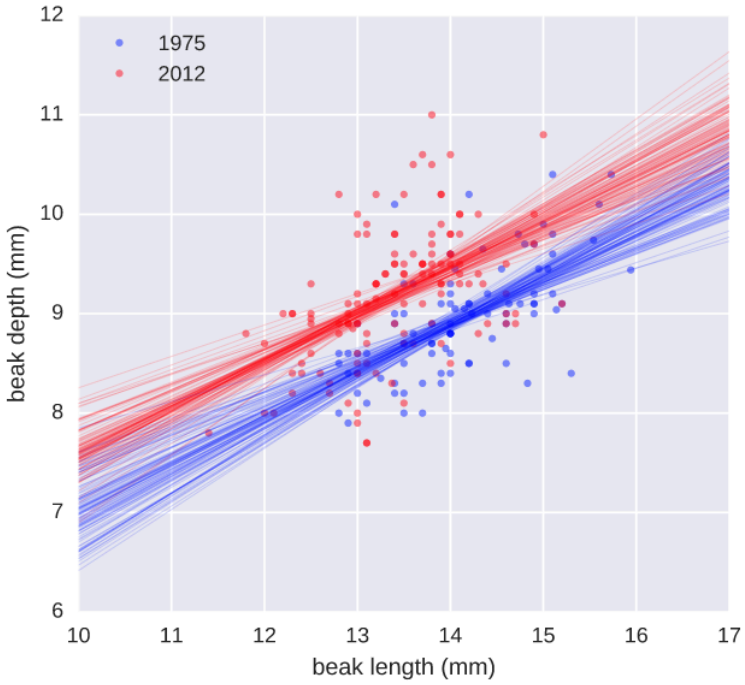
7)

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()`.

- Generate the xx-values for the bootstrap lines using `np.array()`. They should consist of 10 mm and 17 mm.
- Write a `for` loop to plot 100 of the bootstrap lines for the 1975 and 2012 data sets. The lines for the 1975 data set should be `'blue'` and those for the 2012 data set should be `'red'`.

```
# 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()
```



8)

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

- Make arrays of the beak length to depth ratio of each bird for 1975 and for 2012.
- Compute the mean of the length to depth ratio for 1975 and for 2012.
- Generate 10,000 bootstrap replicates each for the mean ratio for 1975 and 2012 using your `draw_bs_reps()` function.
- Get a **99%** bootstrap confidence interval for the length to depth ratio for 1975 and 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)
```

```
1975: mean ratio = 1.57888237719 conf int = [ 1.55668803  1.60073509]
2012: mean ratio = 1.46583422768 conf int = [ 1.44363932  1.48729149]
```

9)

How different is the ratio?

In the last exercise, you showed that the mean beak length to depth ratio was 1.58 in 1975 and 1.47 in 2012. The low end of the 1975 99% confidence interval was 1.56 mm and the high end of the 99% confidence interval in 2012 was 1.49 mm. In addition to these results, what would you say about the ratio of beak length to depth?

ANSWER THE QUESTION 50XP

Possible Answers

- 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. — press 1
- 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. — press 2

When the confidence intervals are not even close to overlapping, the effect is much bigger than variation. You can do a p-value, but the result is already clear

10)

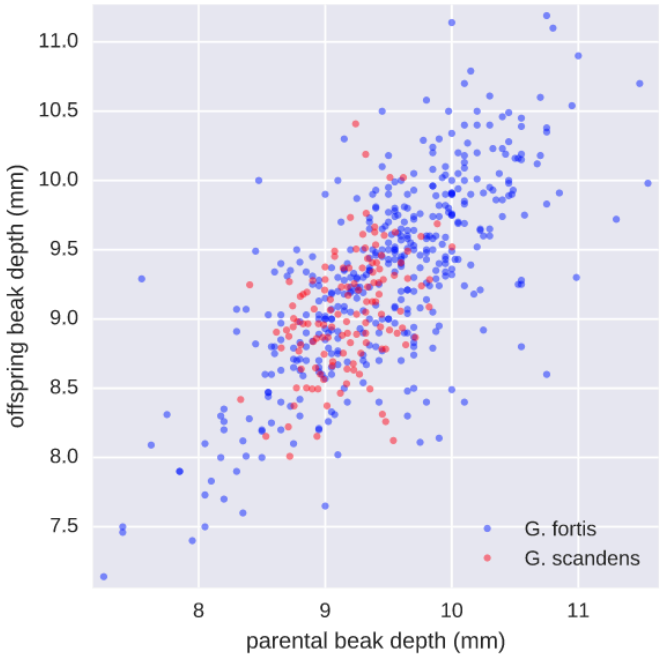
EDA of heritability

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.

- Generate scatter plots for both species. Display the data for *G. fortis* in blue and *G. scandens* in red.

- Set the margins, label the axes, make a legend, and show the plot.

```
# 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)

# Set margins
plt.margins(0.02)

# 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()
```



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

11)

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.
You have **already written** a function to do pairs bootstrap to get estimates for parameters derived from linear regression. Your task in this exercise is to modify that function 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 the pairs samples defined by `func(bs_x, bs_y)`. In the next exercise, you will use `pearson_r` for `func`.

- We have provided your original `draw_bs_pairs_linreg()` function (named as `draw_bs_pairs()`). Modify this function to make the `draw_bs_pairs()` function described above. Be sure to adjust the doc string appropriately, and remember that in this modified function, you only need to return a single statistic.
- Things to keep in mind: The modified function requires an additional `func` parameter and returns only `bs_replicates`, as opposed to `bs_slope_reps` and `bs_intercept_reps` as the function in the sample code does.

```
def draw_bs_pairs(x, y, func,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_replicates = np.empty(size)


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

    return bs_replicates
```

12)

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.

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

- Use the `pearson_r()` function **you wrote in the prequel to this course** to compute the Pearson correlation coefficient for *G. scandens* and *G. fortis*.
- Acquire 1000 pairs bootstrap replicates of the Pearson correlation coefficient using the `draw_bs_pairs()` function you wrote in the previous exercise for *G. scandens* and *G. fortis*.

- Compute the 95% confidence interval for both using your bootstrap replicates.

```
# 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,1000)

bs_replicates_fortis = draw_bs_pairs(bd_parent_fortis,bd_offspring_fortis,pearson_r,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)
```

```
G. scandens: 0.41170636294 [ 0.26564228  0.54388972]
G. fortis: 0.728341239552 [ 0.6694112   0.77840616]
```

It is clear from the confidence intervals that beak depth of the offspring of *G. fortis* parents is more strongly correlated with their offspring than their *G. scandens* counterparts.

13)
Measuring heritability

Remember that 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.
Remember, the data are stored in `bd_parent_scandens`, `bd_offspring_scandens`, `bd_parent_fortis`, and `bd_offspring_fortis`.

- Write a function `heritability(parents, offspring)` that computes heritability defined as the ratio of the covariance of the trait in parents and offspring divided by the variance of the trait in the parents. *Hint*: Remind yourself of the `np.cov()` function we covered in the prequel to this course.
- Use this function to compute the heritability for *G. scandens* and *G. fortis*.
- Acquire 1000 bootstrap replicates of the heritability using pairs bootstrap for *G. scandens* and *G. fortis*.
- Compute the 95% confidence interval for both using your bootstrap replicates

```
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)
```

G. scandens: 0.548534086869 [0.34395487 0.75638267]
G. fortis: 0.722905191144 [0.64655013 0.79688342]

Here again, we see that *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.

14)
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 your array of replicates of heritability. We will take 10,000 pairs permutation replicates.
- Write a `for` loop to generate your replicates.
    - Permute the `bd_parent_scandens` array using `np.random.permutation()`.
    - Compute the heritability between the permuted array and the `bd_offspring_scandens` array using the `heritability()` function you wrote in the last exercise. Store the result in the replicates array.
- Compute the p-value as the number of replicates that are greater than the observed `heritability_scandens`

```
# 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)
```

p-val = 0.0

You get a p-value of 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*. If you like, you can plot a histogram of the heritability replicates to get a feel for how extreme of a value of heritability you might expect by chance.