1 Introduction to Statistics

You're a citizen scientist who has started collecting data about rising water in the river next to where you live. For months, you painstakingly measure the water levels and enter your findings into a notebook. But at the end of it, what exactly do you have? What can all this data tell us?

In this lesson, we'll explore how we can use NumPy to analyze data. We'll learn different methods to calculate common statistical properties of a dataset, such as finding the mean and standard deviation. By the end, you'll be able to do basic analysis of a dataset and understand how we can use statistics to come to conclusions about data.

The statistical concepts that we'll cover include:

- Mean
- Median
- Percentiles
- Interquartile Range
- Outliers
- Standard Deviation

To start, we'll be analyzing single-variable datasets. One way to think of a single-variable dataset is that it contains answers to a question. For instance, we might ask 100 people, "How tall are you?" Their heights in inches would form our dataset.

For our purposes, we'll be organizing our datasets into NumPy arrays. To learn more about NumPy arrays, take our course Learn NumPy: Introduction.

1.1 Code

Introduction to Statistics with NumPy

After the river in your town flooded during a recent hurricane, you've become interested in collecting data about the its height. Every day for the past month, you walk to the river, measure the height of the water, and enter this information into a notebook.

Let's look at how you can use NumPy functions to analyze your dataset.

First, we'll import the NumPy module, so we can use its statistical calculation functions.

import numpy as np

water$_height = np.array([4.01, 4.03, 4.27, 4.29, 4.19, 4.15, 4.16, 4.23, 4.29, 4.19, 4.00, 4.22, 4.25, 4.19, 4.10, 4.14$

Let's use the function np.mean() to find the average water height:

np.mean(water$_height$)

5.2510000000000003

But wait! We should sort our data to see if there could be any measurements to throw our data off, or represent a deviation from the mean:

np.sort(water$_height$)

array([3.99, 4. , 4.01, 4.03, 4.03, 4.04, 4.08, 4.08, 4.1 , 4.11, 4.14, 4.15, 4.16, 4.19, 4.19, 4.19, 4.22, 4.23, 4.23, 4.23, 4.23, 4.25, 4.27, 4.29, 4.29, 6.18, 8.19, 11.2 , 14.03, 14.2])

Looks like that thunderstorm might have impacted the average height! Let's measure the median to see if its more representative of the dataset:

np.median(water$_height$)

4.1900000000000004

While the median tells us where half of our data lies, let's look at a value closer to the end of the dataset. We can use percentiles to use a data points position and get its value:

np.percentile(water$_height, 75$)

4.2649999999999997

So far, we've gotten a good idea about specific values. But what about the spread of our data? Let's calculate the standard deviation to understand how similar or how different each data point is:

np.std(water$_height$)

2.784585367099861

Great! Just using a few simple functions we've been able to quickly calculate several important measurements and can begin analyzing our dataset.

2 Mean

The first statistical concept we'll explore is mean, also commonly referred to as an average. The mean is a useful measurement to get the center of a dataset. NumPy has a built-in function to calculate the average or mean of arrays: np.mean

Let's say we want to find the average number of pounds of produce a person purchases per week. We administered a survey and received 1,000 responses:

survey$_responses = [5, 10.2, 4, .3...6.6]$

We can then transform the dataset into a NumPy array and use the function np.mean to calculate the average:

¿¿¿ survey$_array = np.array(survey_responses) >>> np.mean(survey_array)5.220$

2.1 Mean with NumPy

We can also use np.mean to calculate the percent of array elements that have a certain property.

As we know, a logical operator will evaluate each item in an array to see if it matches the specified condition. If the item matches the given condition, the item will evaluate as True and equal 1. If it does not match, it will be False and equal 0.

When np.mean calculates a logical statement, the resulting mean value will be equivalent to the total number of True items divided by the total array length.

In our produce survey example, we can use this calculation to find out the percentage of people who bought more than 8 pounds of produce each week:

¿¿¿ np.mean(survey$_array > 8)0.2$

The logical statement survey$_array > 8 evaluates which survey answers were greater than 8, and assigns them$

2.2 Mean with Multiple Dimension Arrays

Calculating the Mean of 2D Arrays

If we have a two-dimensional array, np.mean can calculate the means of the larger array as well as the interior values.

Let's imagine a game of ring toss at a carnival. In this game, you have three different chances to get all three rings onto a stick. In our ring$_toss array, each interior array (the arrays within th$

First, we can use np.mean to find the mean across all the arrays:

¿¿¿ ring$_toss = np.array([[1, 0, 0], [0, 0, 1], [1, 0, 1]]) >>> np.mean(ring_toss)0.44444444444444442$

To find the means of each interior array, we specify axis 1 (the "rows"):

¿¿¿ np.mean(ring$_toss, axis = 1)array([0.33333333, 0.33333333, 0.66666667])$

To find the means of each index position (i.e, mean of all 1st tosses, mean of all 2nd tosses, ...), we specify axis 0 (the "columns"):

¿¿¿ np.mean(ring$_toss, axis = 0)array([0.66666667, 0., 0.66666667])$

3 Outliers

As we can see, the mean is a helpful way to quickly understand different parts of our data. However, the mean is highly influenced by the specific values in our data set. What happens when one of those values is significantly different from the rest?

Values that don't fit within the majority of a dataset are known as outliers. It's important to identify outliers because if they go unnoticed, they can skew our data and lead to error in our analysis (like determining the mean). They can also be useful in pointing out errors in our data collection.

When we're able to identify outliers, we can then determine if they were due to an error in sample collection or whether or not they represent a significant but real deviation from the mean.

Suppose we want to determine the average height for 3rd graders. We measure several students at the local school, but accidentally measure one student in centimeters rather than in inches. If we're not paying attention, our dataset could end up looking like this:

[50, 50, 51, 49, 48, 127]

In this case, 127 would be an outlier.

Some outliers aren't the result of a mistake. For instance, suppose that one of our 3rd graders had skipped a grade and was actually a year younger than everyone else in the class:

[50, 50, 51, 49, 48, 45]

She might be significantly shorter at 45", but her height would still be an outlier.

Suppose that another student was just unusually tall for his age:

[50, 50, 51, 49, 48, 58.5]

His height of 58.5" would also be an outlier.

3.1 Sorting and Outliers

Sorting and Outliers

One way to quickly identify outliers is by sorting our data, Once our data is sorted, we can quickly glance at the beginning or end of an array to see if some values lie far beyond the expected range. We can use the NumPy function np.sort to sort our data.

Let's go back to our 3rd grade height example, and imagine an 8th grader walked into our experiement:

¿¿¿ heights = np.array([49.7, 46.9, 62, 47.2, 47, 48.3, 48.7])

If we use np.sort, we can immediately identify the taller student since their height (62") is noticeably outside the range of the dataset:

¿¿¿ np.sort(heights) array([46.9, 47. , 47.2, 48.3, 48.7, 49.7, 62])

4 Median

Another key metric that we can use in data analysis is the median. The median is the middle value of a dataset that's been ordered in terms of magnitude (from lowest to highest).

Let's look at the following array:

np.array([1, 1, 2, 3, 4, 5, 5])

In this example, the median would be 3, because it is positioned half-way between the minimum value and the maximum value.

If the length of our dataset was an even number, the median would be the value halfway between the two central values. So in the following example, the median would be 3.5:

np.array([1, 1, 2, 3, 4, 5, 5, 6])

But what if we had a very large dataset? It would get very tedious to count all of the values. Luckily, NumPy also has a function to calculate the median, np.median:

¿¿¿ my$_a$$rray = np.array([50, 38, 291, 59, 14]) >>> np.median(my_array)$50.0

5 Mean vs Median

In a dataset, the median value can provide an important comparison to the mean. Unlike a mean, the median is not affected by outliers. This becomes important in skewed datasets, datasets whose values are not distributed evenly.

6 Percentiles

As we know, the median is the middle of a dataset: it is the number for which 50

This type of point is called a percentile. The Nth percentile is defined as the point N

Let's look at the following array:

d = [1, 2, 3, 4, 4, 4, 6, 6, 7, 8, 8]

There are 11 numbers in the dataset. The 40th percentile will have 40

In NumPy, we can calculate percentiles using the function np.percentile, which takes two arguments: the array and the percentile to calculate.

Here's how we would use NumPy to calculate the 40th percentile of array d:

¿¿¿ d = np.array([1, 2, 3, 4, 4, 4, 6, 6, 7, 8, 8]) ¿¿¿ np.percentile(d, 40) 4.00

Some percentiles have specific names:

The 25th percentile is called the first quartile The 50th percentile is called the median The 75th percentile is called the third quartile

The minimum, first quartile, median, third quartile, and maximum of a dataset are called a five-number summary. This set of numbers is a great thing to compute when we get a new dataset.

The difference between the first and third quartile is a value called the interquartile range. For example, say we have the following array:

d = [1, 2, 3, 4, 4, 4, 6, 6, 7, 8, 8]

We can calculate the 25th and 75th percentiles using np.percentile:

np.percentile(d, 25) ¿¿¿ 3.5 np.percentile(d, 75) ¿¿¿ 6.5

Then to find the interquartile range, we subtract the value of the 25th percentile from the value of the 75th:

6.5 - 3.5 = 3

50

7 Standard Deviation

While the mean and median can tell us about the center of our data, they do not reflect the range of the data. That's where standard deviation comes in.

Similar to the interquartile range, the standard deviation tells us the spread of the data. The larger the standard deviation, the more spread out our data is from the center. The smaller the standard deviation, the more the data is clustered around the mean.

8 Review of NumPy

Let's review! In this lesson, you learned how to use NumPy to analyze single-variable datasets. Here's what we covered:

Using the np.sort method to locate outliers. Calculating central positions of a dataset using np.mean and np.median. Understanding the spread of our data using percentiles and the interquartile range. Finding the standard deviation of a dataset using np.std.

In our next lesson, we'll continue our exploration of NumPy and see how we can use it to analyze different statistical distributions. Follow the checkpoints below to practice what you just learned!

9 Distributions

A university wants to keep track of the popularity of different programs over time, to ensure that programs are allocated enough space and resources. You work in the admissions office and are asked to put together a set of visuals that show these trends to interested applicants. How can we calculate these distributions? Would we be able to see trends and predict the popularity of certain programs in the future? How would we show this information?

In this lesson, we are going to learn how to use NumPy to analyze different distributions, generate random numbers to produce datasets, and use Matplotlib to visualize our findings.

This lesson will cover:

How to generate and graph histograms How to identify different distributions by their shape Normal distributions How standard deviations relate to normal distributions Binomial distributions

9.1 Statistical Distributions with NumPy

Imagine that you work as an admissions officer at a university. Part of your job is to collect, analyze, and visualize data that's relevant to interested applicants.

Recently, you've become interested in how histograms can show different distributions of populations and even occurences. You think that histograms would be useful in visualizing different trends, such as changes in department numbers and participation in extracurriculars. You also want to learn more about how you can use randomly generated distributions to make statistical calculations and predict the probability of future events, such as the sucess of your ultimate frisbee team.

For this lesson, we'll be using NumPy to calculate distributions and Matplotlib to graph our calculations.

import numpy as np from matplotlib import pyplot as plt

One set of data you want to analyze is enrollment in different degree programs. By looking at histograms of the number of years students are enrolled in a program, you can identify what programs are becoming more popular, which are falling out of favor, and which have steady, continual enrollment.

First, let's look at how many hundreds of students decide to enroll in Codecademy University and how many years they've been enrolled.

total$_e nrollment = [1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 5, 5, 5]$

plt.hist(total$_e nrollment, bins = 5, range = (1, 6))plt.title('StudentEnrollment(CodecademyUniversity)')$

The histogram above shows the University's total enrollment, which is fairly consistent. This is a uniform distribution and is what the University wants to see. Total enrollment is staying at a good level.

Now let's take a look at the enrollment specifically for students seeking a degree in History:

history$_e nrollment = [1, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 4, 4, 4, 5, 5, 5, 5, 5, 5]$

plt.hist(history$_e nrollment, bins = 5, range = (1, 6))plt.title('StudentEnrollment(HistoryDepartment)')p$

What does this histogram tell us? Well this is somewhat skewed left dataset, we can see that there are a lot more students who have been enrolled for 3 or 4 years over 1 and 2 years. This indicates that the History program is becoming less popular. Where are all the students going then?

The school recently invested a lot of money in a new building for the Computer Science Department. Let's take a look at enrollment and see if the investment is paying off.

cs$_e nrollment = [1, 1, 1, 1, 1, 2, 2, 2, 2, 3, 4, 4]$

plt.hist(cs$_e nrollment, bins = 5, range = (1, 6))plt.title('StudentEnrollment(ComputerScienceDepartme$

It looks like enrollment has skyrocketed for the Computer Science department in recent years. This could be because the University invested in the department, or it could be a sign that the sought after job skills in the real world are changing. Whatever the reason, the histograms let us clearly see the trends.

Interested applicants would like to know what kinds of SAT scores accepted students had. You've previously calculated that the mean score is 1250, with a standard deviation of 50.

Rather than gather every students score, you take what you know about the data and use a random number generator to generate a model.

sat$_s cores = np.random.normal(1250, 50, size = 100000)$

plt.hist(sat$_s cores, bins = 1000, range = (800, 1600))plt.title('AdmittedStudentSATScores')plt.xlabel('SA$

95

mean = 1250 one$_s td = 50$

two$_b elow = (mean - 2 * one_s td)printtwo_b elow$

1150

Looks like they're just below it! Better re-take that test.

One of the big draws to your school is your excellent ultimate frisbee team. The team wins about 70

ultimate = np.random.binomial(50, 0.70, size=10000) plt.hist(ultimate, range=(0, 50), bins=50, normed=True) plt.xlabel('Number of Games') plt.ylabel('Frequency') plt.show()

Since it's a little hard to see from the graph, let's calculate exactly what chance they have of winning 40 games:

ultimate = np.random.binomial(50, 0.70, size=10000) np.mean(ultimate == 40)

0.041000000000000002

Hmm, looks like it might be tough for the team to reach that number of wins, given the current data - but even more of a reason for this applicant to sign up and help the team improve!

10 Histograms

When we first look at a dataset, we want to be able to quickly understand certain things about it:

Do some values occur more often than others? What is the range of the dataset (i.e., the min and the max values)? Are there a lot of outliers?

We can visualize this information using a chart called a histogram.

For instance, suppose that we have the following dataset:

d = [1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 4, 4, 5]

A simple histogram might show us how many 1's, 2's, 3's, etc. we have in this dataset. Value Number of Samples 1 3 2 5 3 2 4 4 5 1

When graphed, our histogram would look like this:

Suppose we had a larger dataset with values ranging from 0 to 50. We might not want to know exactly how many 0's, 1's, 2's, etc. we have.

Instead, we might want to know how many values fall between 0 and 5, 6 and 10, 11 and 15, etc.

These groupings are called bins. All bins in a histogram are always the same size. The width of each bin is the distance between the minimum and maximum values of each bin. In our example, the width of each bin would be 5.

For a dataset like this, our histogram table would look like this: Bin Number of Values (0, 5) 2 (6, 10) 10 (11, 15) 11 ... ... (46, 50) 3

And if we were to graph this histogram, it would look like this:

We can graph histograms using a Python module known as Matplotlib. We're not going to go into detail about Matplotlib's plotting functions, but if you're interested in learning more, take our course Introduction to Matplotlib.

For now, familiarize yourself with the following syntax to draw a histogram:

This imports the plotting package. We only need to do this once. from matplotlib import pyplot as plt

This plots a histogram plt.hist(data)

This displays the histogram plt.show()

When we enter plt.hist with no keyword arguments, matplotlib will automatically make a histogram with 10 bins of equal width that span the entire range of our data.

If you want a different number of bins, use the keyword bins. For instance, the following code would give us 5 bins, instead of 10:

plt.hist(data, bins=5)

If you want a different range, you can pass in the minimum and maximum values that you want to histogram using the keyword range. We pass in a tuple of two numbers. The first number is the minimum value that we want to plot and the second value is the number that we want to plot up to, but not including.

For instance, if our dataset contained values between 0 and 100, but we only wanted to histogram numbers between 20 and 50, we could use this command:

We pass 51 so that our range includes 50 plt.hist(data, range=(20, 51))

Here's a complete example:

from matplotlib import pyplot as plt

d = np.array([1, 1, 1, 2, 2, 2, 2, 2, 3, 3, 4, 4, 4, 4, 5])

plt.hist(d, bins=5, range=(1, 6))

plt.show()

11 Examples of Distributions

Histograms and their datasets can be classified based on the shape of the graphed values. In the next two exercises, we'll look at two different ways of describing histograms.

One way to classify a dataset is by counting the number of distinct peaks present in the graph. Peaks represent concentrations of data. Let's look at the following examples:

A unimodal dataset has only one distinct peak. histogram

A bimodal dataset has two distinct peaks. This often happens when the data contains two different populations. histogram

A multimodal dataset has more than two peaks. histogram

A uniform dataset doesn't have any distinct peaks.

Most of the datasets that we'll be dealing with will be unimodal (one peak). We can further classify unimodal distributions by describing where most of the numbers are relative to the peak.

A symmetric dataset has equal amounts of data on both sides of the peak. Both sides should look about the same. histogram

A skew-right dataset has a long tail on the right of the peak, but most of the data is on the left. histogram

A skew-left dataset has a long tail on the left of the peak, but most of the data is on the right. histogram

The type of distribution affects the position of the mean and median. In heavily skewed distributions, the mean becomes a less useful measurement.

SYMMETRIC histogram

SKEW-RIGHT histogram

SKEW-LEFT histogram

12 Normal Distributions

The most common distribution in statistics is known as the normal distribution, which is a symmetric, unimodal distribution.

Lots of things follow a normal distribution:

The heights of a large group of people Blood pressure measurements for a group of healthy people Errors in measurements

Normal distributions are defined by their mean and standard deviation. The mean sets the "middle" of the distribution, and the standard deviation sets the "width" of the distribution. A larger standard deviation leads to a wider distribution. A smaller standard deviation leads to a skinnier distribution.

Here are a few examples of normal distributions with different means and standard deviations:

normal$_distribution$

As we can see, each set of data has the same "shape", but with slight differences depending on their mean and standard deviation.

We can generate our own normally distributed datasets using NumPy. Using these datasets can help us better understand the properties and behavior of different distributions. We can also use them to model results, which we can then use as a comparison to real data.

In order to create these datasets, we need to use a random number generator. The NumPy library has several functions for generating random numbers, including one specifically built to generate a set of numbers that fit a normal distribution: np.random.normal. This function takes the following keyword arguments:

loc: the mean for the normal distribution scale: the standard deviation of the distribution size: the number of random numbers to generate

a = np.random.normal(0, 1, size=100000)

If we were to plot this set of random numbers as a histogram, it would look like this:

13 Standard Deviation and Normal Distribution

We know that the standard deviation affects the "shape" of our normal distribution. The last exercise helps to give us a more quantitative understanding of this.

Suppose that we have a normal distribution with a mean of 50 and a standard deviation of 10. When we say "within one standard deviation of the mean", this is what we are saying mathematically:

lower$_bound = mean - std = 50 - 10 = 40$

upper$_bound = mean + std = 50 + 10 = 60$

It turns out that we can expect about 68

As we saw in the previous exercise, no matter what mean and standard deviation we choose, 68

In fact, here are a few more helpful rules for normal distributions:

689599.7

14 Binomial Distribution

It's known that a certain basketball player makes 30

However, he actually made 4 free throws out of 10 or 40

The binomial distribution can help us. It tells us how likely it is for a certain number of "successes" to happen, given a probability of success and a number of trials.

In this example:

The probability of success was 30The number of trials was 10 (he took 10 shots) The number of successes was 4 (he made 4 shots)

The binomial distribution is important because it allows us to know how likely a certain outcome is, even when it's not the expected one. From this graph, we can see that it's not that unlikely an outcome for our basketball player to get

4 free throws out of 10. However, it would be pretty unlikely for him to get all 10.

binomiall

There are some complicated formulas for determining these types of probabilities. Luckily for us, we can use NumPy - specifically, its ability to generate random numbers. We can use these random numbers to model a distribution of numerical data that matches the real-life scenario we're interested in understanding.

For instance, suppose we want to know the different probabilities of our basketball player making 1, 2, 3, etc. out of 10 shots.

NumPy has a function for generating binomial distributions: np.random.binomial, which we can use to determine the probability of different outcomes.

The function will return the number of successes for each "experiment".

It takes the following arguments:

N: The number of samples or trials P: The probability of success size: The number of experiments

Let's generate a bunch of "experiments" of our basketball player making 10 shots. We choose a big N to be sure that our probabilities converge on the correct answer.

Let's generate 10,000 "experiments" N = 10 shots P = 0.30 (30

a = np.random.binomial(10, 0.30, size=10000)

Now we have a record of 10,000 experiments. We can use Matplotlib to plot the results of all of these experiments:

plt.hist(a, range=(0, 10), bins=10, normed=True) plt.xlabel('Number of "Free Throws"') plt.ylabel('Frequency') plt.show()

binomiall

15 Binomial Distribution and Probability

Let's return to our original question:

Our basketball player has a 30

We can calculate a different probability by counting the percent of experiments with the same outcome, using the np.mean function.

Remember that taking the mean of a logical statement will give us the percent of values that satisfy our logical statement.

Let's calculate the probability that he makes 4 baskets:

a = np.random.binomial(10, 0.30, size=10000) np.mean(a == 4)

When we run this code, we might get:

¿¿ 0.1973

Remember, because we're using a random number generator, we'll get a slightly different result each time. With the large *size we chose, the calculated probability should be accurate to about 2 decimal places.*

So, our basketball player has a roughly 20

This suggests that what we observed wasn't that unlikely. It's quite possible that he hasn't got any better; he just got lucky.

16 Review

Let's review! In this lesson, you learned how to use NumPy to analyze different distributions and generate random numbers to produce datasets. Here's what we covered:

What is a histogram and how to map one using Matplotlib How to identify different dataset shapes, depending on peaks or distribution of data The definition of a normal distribution and how to use NumPy to generate one using NumPy's random number functions The relationships between normal distributions and standard deviations The definition of a binomial distribution

Now you can use NumPy to analyze and graph your own datasets! You should practice building your intuition about not only what the data says, but what conclusions can be drawn from your observations.

np.random.normal(loc = 16.3, scale = 3.3, size = 1000)

17 Quiz

Which of the following are the correct keyword arguments for generating a random distribution using np.random.binomial? N, P, size

What is a histogram? A chart that creates equally spaced bins and counts how many values from our dataset fall into each bin.

How many peaks does a unimodal dataset have? One

In a normal distribution, how much of the data lies within one standard deviation? 68

The average height of a male giraffe is 16.3 feet with a standard deviation of 3.3 feet. Which of the following will generate a random distribution of 1000 male giraffe heights using np.random.normal? np.random.normal(loc = 16.3, scale = 3.3, size = 1000)

Why do we use binomial distributions? Because they are effective at helping us understand the different probabilities that an event will occur.

18 Project Election

import codecademylib import numpy as np from matplotlib import pyplot as plt

survey$_r esponses = ['Ceballos','Kerrigan','Ceballos','Ceballos','Ceballos','Kerrigan','Kerrigan','Ceba$

total$_c eballos = sum([1 for n in survey_r esponses if n ==' Ceballos'])print(total_c eballos)$

percentage$_c eballos = 100 * total_c eballos/float(len(survey_r esponses))$

print(percentage$_c eballos)$

possible$_s urveys = np.random.binomial(len(survey_r esponses), 0.54, size = 10000)/float(len(survey_r espo$

plt.hist(possible$_s urveys, range = (0, 1), bins = 20)plt.show()$

ceballos$_l oss_s urveys = np.mean(possible_s urveys < 0.5) * 100$

print(ceballos$_l oss_s urveys)$

possible$_s urveys_l en = float(len(possible_s urveys))incorrect_p redictions = len(possible_s urveys[possible_s ur$ $0.5])ceballos_l oss_s urveys_2 = incorrect_p redictions/possible_s urveys_l enprint(ceballos_l oss_s urveys_2)$

large$_s urvey = np.random.binomial(float(7000), 0.54, size = 10000)/float(7000)$

ceballos$_l oss_n ew = np.mean(large_s urvey < 0.5)print(ceballos_l oss_n ew)$

large$_s urveys_l en = float(len(large_s urvey))incorrect_p redictions = len(large_s urvey[large_s urvey <$ $0.5])ceballos_l oss_s urveys_n ew_2 = incorrect_p redictions/large_s urveys_l enprint(ceballos_l oss_s urveys_n ew_2)$

19 Project Muncies

import codecademylib import numpy as np

calorie$_s tats = np.genfromtxt('cereal.csv', delimiter =',')calorie_s tats_s orted =$ $np.sort(calorie_s tats)$

average$_c alories = np.mean(calorie_s tats)median_c alories = np.median(calorie_s tats_s orted)nth_p ercentile =$ $np.percentile(calorie_s tats_s orted, 4)more_c alories = np.mean(calorie_s tats > 60)*$ $100calorie_s td = np.std(calorie_s tats)$

"' CrunchieMunchies are lit. They are more healthy than 96"'

print(calorie$_s td)print(more_c alories)print(nth_p ercentile)print(average_c alories)print(median_c alories)pri$

19.1 csv

70, 120, 70, 50, 110 , 110, 110, 130, 90, 90, 120, 110, 120, 110, 110, 110, 100, 110, 110, 110, 100, 110, 100, 100, 110, 110, 100, 120, 120, 110, 100, 110, 100, 110, 120, 120, 110, 110, 110, 140, 110, 100, 110, 100, 150, 150, 160, 100, 120, 140, 90, 130, 120, 100, 50, 50, 100, 100, 120, 100, 90, 110, 110, 80, 90, 90, 110, 110, 90, 110, 140, 100, 110, 110, 100, 100, 110

20 Bakery

import numpy as np

(Flour—Sugar—Eggs—Milk—Butter) cupcakes = np.array([2, 0.75, 2, 1, 0.5])

recipes = np.genfromtxt('recipes.csv', delimiter=',')

print(recipes)

eggs = recipes[:,2]

print(eggs == 1)

cookies = recipes[2,:]

print(cookies)

double$_batch = cupcakes * 2$

grocery$_list = double_batch + cookies$

print(grocery$_list$)

20.1 csv

2,0.75,2,1,0.5 1,0.125,1,1,0.125 2.75,1.5,1,0,1 4,0.5,2,2,0.5

21 Introduction To ScyPy

Say you work for a major social media website. Your boss always says "data drives all our decisions" and it seems to be true. Metrics are collected on all users of your website, terabytes of data stored in replicated databases.

One day, your boss wants to know if college students are engaging in your website. You pull up the records for users in that age bracket and look at them one by one. The first person only spent half a second on your website before closing the tab — that doesn't look good. But the second person was on the site for thirty minutes! That's a running average of 15 minutes site time per user, but you still have half a million records to look at.

On top of that, you need to compare it against other age brackets (and the average overall). That's going to take a lot of time if you do it all by hand, and you're still not sure what your methodology for proving college students spend enough time on your website to be "engaged".

When conducting data analysis, we want to say something meaningful about our data. Often, we want to know if a change or difference we see in a dataset is ”real” or if it’s just a normal fluctuation or a result of the specific sample of people we have chosen to measure. A difference we observe in our data is only important if we can be reasonably sure that it is representative of the population as a whole, and reasonably sure that our result is repeatable.

This question of whether a difference is significant or not is essential to making decisions based on that difference. Some instances where this might come up include:

Performing an A/B test — are the different observations really the results of different conditions (i.e., Condition A vs. Condition B)? Or just the result of random chance? Conducting a survey — is the fact that men gave slightly different responses than women a real difference between men and women? Or just the result of chance?

In this lesson, we will cover the fundamental concepts that will help us create tests to measure our confidence in our statistical results:

Sample means and population means The Central Limit Theorem Why we use hypothesis tests What errors we can come across and how to classify them

Are the Millennials Engaged?

You work at the global megacorp social network SpyPy. SpyPy has 1.5 billion daily users, and you want to make sure that people in the millennial age bracket are engaging with your website. Your boss seems particularly frazzled by this question, and he’s put it on you to find out. You decide that ”engagement” means spending more than the average of seven minutes on the website. You fire up your data-science stack in Python and first check the average time – which turns out to be near 11 whole minutes! But you can’t really tell if they’re really spending more time or if it’s just random chance that a few of your users left the browser open and walked away. You write the following code:

import spypy from scipy.stats import ttest$_1 samp$

millennial$_t imes = spypy.get_s ite_t imes_f or_d emographic('millennial')t_s tat, p_v al = ttest_1 samp(millennial_t imes, 7)$

if p$_v al < .05 : print”TheMillennialsareengaged!”else : print”TheMillennialsarenotengaged :$ (!”

The Millennials are engaged!

SpyPy: We’re Significantly Different

Well that’s great news! Millennials are, for the most part, spending around 10 minutes on your website. But before you break out the champagne glasses your boss is in a frenzy again, this time about Metropolitan Statistical Areas (MSAs). You are tasked with finding if people in cooler climates post more pictures on SpyPy than people in warmer climates. You cross corroborate with weather data and run a statistical test on the info.

from scipy.stats import ttest$_{i}nd$

warmer$_{w}eather_{p}icture_{c}ount = spypy.get_{n}umber_{p}ictures_{f}or_{c}limate('hot')colder_{w}eather_{p}icture_{c}ount =$ $spypy.get_{n}umber_{p}ictures_{f}or_{c}limate('cold')$

t$_{s}tat, p_{v}al = ttest_{i}nd(warmer_{w}eather_{p}icture_{c}ount, colder_{w}eather_{p}icture_{c}ount)$

if p$_{v}al < .05 : print"Peoplefromcolderclimatespostadifferentnumberofpicturescomparedtopeoplefromw$
$print"Climatedoesn'tappeartoaffectthenumberofpicturesposted"$

Climate doesn't appear to affect the number of pictures posted

SpyPy: Because We Care About Your Data

Seems like climate probably doesn't really affect the number of times people post pictures. Not really sure why that would've been the case anyway. SpyPy has a new feature that you think will get people to interact with the website for longer: SpyPy Stories. It is preliminarily being launched to 8 million users and the internal goal is to get 2 million people to post SpyPy Stories in the first week. Unfortunately, only 1,997,893 people posted SpyPy Stories this week. We want to know if this is a significant difference from our goal – did we pretty much meet it or did we seriously miss? You know how to answer this question:

from scipy.stats import binom$_{t}est$

number$_{o}f_{t}rials = 8000000expected_{s}uccesses = 2000000actual_{s}uccesses = 1997893expected_{s}uccess_{r}ate =$ $float(expected_{s}uccesses)/float(number_{o}f_{t}rials)$

p$_{v}al = binom_{t}est(actual_{s}uccesses, n = number_{o}f_{t}rials, p = expected_{s}uccess_{r}ate)ifp_{v}al <$ $0.05 : print"Wedidn'thitourtargetbyasignificantamount"else : print"Wejustmissedourtargetbyaverysm$

We just missed our target by a very small amount!

Looks like we came very close to hitting our target for SpyPy Stories! You've saved the day so many times already! Your boss comes by to thank you for all the hard work you put in today and says you've made significant contributions to the team. You tell her you're not sure if that's true, but you definitely have a way of finding out.

22 Sample and Population Means

Suppose you want to know the average height of an oak tree in your local park. On Monday, you measure 10 trees and get an average height of 32 ft. On Tuesday, you measure 12 different trees and reach an average height of 35 ft. On Wednesday, you measure the remaining 11 trees in the park, whose average height is 31 ft. Overall, the average height for all trees in your local park is 32.8 ft.

The individual measurements on Monday, Tuesday, and Wednesday are called samples. A sample is a subset of the entire population. The mean of each sample is the sample mean and it is an estimate of the population mean.

Note that the sample means (32 ft., 35 ft., and 31 ft.) were all close to the population mean (32.8 ft.), but were all slightly different from the population mean and from each other.

For a population, the mean is a constant value no matter how many times it's recalculated. But with a set of samples, the mean will depend on exactly what samples we happened to choose. From a sample mean, we can then extrapolate the mean of the population as a whole. There are many reasons we might use sampling, such as:

- We don't have data for the whole population.
- We have the whole population data, but it is so large that it is infeasible to analyze.
- We can provide meaningful answers to questions faster with sampling.

When we have a numerical dataset and want to know the average value, we calculate the mean. For a population, the mean is a constant value no matter how many times it's recalculated. But with a set of samples, the mean will depend on exactly what samples we happened to choose. From a sample mean, we can then extrapolate the mean of the population as a whole.

23 Central Limit Theorem

Perhaps, this time, you're a tailor of school uniforms at a middle school. You need to know the average height of people from 10-13 years old in order to know which sizes to make the uniforms. Knowing the best decisions are based on data, you set out to do some research at your local middle school.

Organizing with the school, you measure the heights of some students. Their average height is 57.5 inches. You know a little about sampling and decide that measuring 30 out of the 300 students gives enough data to assume 57.5 inches is roughly the average height of everyone at the middle school. You set to work with this dimension and make uniforms that fit people of this height, some smaller and some larger.

Unfortunately, when you go about making your uniforms many reports come back saying that they are too small. Something must have gone wrong with your decision-making process! You go back to collect the rest of the data: you measure the sixth graders one day (56.7, not so bad), the seventh graders after that (59 inches tall on average), and the eighth graders the next day (61.7 inches!). Your sample mean was so far off from your population mean. How did this happen?

Well, your sample selection was skewed to one direction of the total population. It looks like you must have measured more sixth graders than is representative of the whole middle school. How do you get an average sample height that looks more like the average population height?

In the previous exercise, we looked at different sets of samples taken from a population and how the mean of each set could be different from the population mean. This is a natural consequence of the fact that a set of samples has less data than the population to which it belongs. If our sample selection is poor then we will have a sample mean seriously skewed from our population mean.

There is one surefire way to mitigate the risk of having a skewed sample mean — take a larger set of samples. The sample mean of a larger sample set will more closely approximate the population mean. This phenomenon, known as the Central Limit Theorem, states that if we have a large enough sample size, all of our sample means will be sufficiently close to the population mean.

Later, we'll learn how to put numeric values on "large enough" and "sufficiently close".

24 Hypothesis Tests

When observing differences in data, a data analyst understands the possibility that these differences could be the result of random chance.

Suppose we want to know if men are more likely to sign up for a given programming class than women. We invite 100 men and 100 women to this class. After one week, 34 women sign up, and 39 men sign up. More men than women signed up, but is this a "real" difference?

We have taken sample means from two different populations, men and women. We want to know if the difference that we observe in these sample means reflects a difference in the population means. To formally answer this question, we need to re-frame it in terms of probability:

"What is the probability that men and women have the same level of interest in this class and that the difference we observed is just chance?"

In other words, "If we gave the same invitation to every person in the world, would more men still sign up?"

A more formal version is: "What is the probability that the two population means are the same and that the difference we observed in the sample means is just chance?"

These statements are all ways of expressing a null hypothesis. A null hypothesis is a statement that the observed difference is the result of chance.

Hypothesis testing is a mathematical way of determining whether we can be confident that the null hypothesis is false. Different situations will require different types of hypothesis testing, which we will learn about in the next lesson.

25 Type I or Type II

When we rely on automated processes to make our decisions for us, we need to be aware of how this automation can lead to mistakes. Computer programs are as fallible as the humans who design them. As humans capable of programming, the responsibility is on us to understand what can go wrong and what we can do to contain these foreseeable problems.

In statistical hypothesis testing, we concern ourselves primarily with two types of error. The first kind of error, known as a Type I error, is finding a correlation between things that are not related. This error is sometimes called a "false positive" and occurs when the null hypothesis is rejected even though it is true.

For example, let's say you conduct an A/B test for an online store and conclude that interface B is significantly better than interface A at directing traffic to a checkout page. You have rejected the null hypothesis that there is no difference between the two interfaces. If, in reality, your results were due to the groups you happened to pick, and there is actually no significant difference between interface A and interface B in the greater population, you have been the victim of a false positive.

The second kind of error, a Type II error, is failing to find a correlation between things that are actually related. This error is referred to as a "false negative" and occurs when the null hypothesis is accepted even though it is false.

For example, with the A/B test situation, let's say that after the test, you concluded that there was no significant difference between interface A and interface B. If there actually is a difference in the population as a whole, your test has resulted in a false negative.

26 P-Values

We have discussed how a hypothesis test is used to determine the validity of a null hypothesis. A hypothesis test provides a numerical answer, called a p-value, that helps us decide how confident we can be in the result. In this context, a p-value is the probability that we yield the observed statistics under the assumption that the null hypothesis is true.

A p-value of 0.05 would mean that there is a 5

Before conducting a hypothesis test, we determine the necessary threshold we would need before concluding that the results are significant. A higher p-value is more likely to give a false positive so if we want to be very sure that the result is not due to just chance, we will select a very small p-value.

It is important that we choose the significance level before we perform our statistical hypothesis tests to yield a p-value. If we wait until after we see the results, we might pick our threshold such that we get the result we want to see. For instance, if we're trying to publish our results, we might set a significance level that makes our results seem statistically significant. Choosing our significance level in advance helps keep us honest.

Generally, we want a p-value of less than 0.05, meaning that there is less than a 5

27 Examples

Suppose we were exploring the relationship between local honey and allergies. Which of these would be a statement of the null hypothesis? Local honey has no effect on allergies, any relationship between consuming local honey and allergic outbreaks is due to chance. Correct! The null hypothesis states that any difference observed within sample means is coincidental.

Which of the following describes a Type II error? A survey on preferred ice cream flavors not establishing a clear favorite when the majority of people prefer chocolate.

Which of these describes a sample mean? The mean of a subset of our population which is hopefully, but not necessarily, representative of the overall average.

What is a p-value? In a hypothesis test, a p-value is the probability that the null hypothesis is true.

Which of the following hypothesis tests would be used to compare two sets of numerical data?

Chi Square x 1 Sample T-Test x ANOVA x 2 Sample T-Test

Analysis of variance is used to determine if three or more numerical samples come from the same population.

Which of these is an accurate statement of the Central Limit Theorem? For a large enough sample size, our sample mean will be sufficiently close to the population mean.

What is a statistical hypothesis test? A way of quantifying the truth of a statement.

ANOVA is a type of hypothesis test, but does not cover all types of hypothesis test.

28 Key Points

Let's take a second and review. In this lesson, you learned the basics of the NumPy package. Here are some key points:

Arrays are a special type of list that allows us to store values in an organized manner. An array can be created by either defining it directly using np.array() or by importing a CSV using np.genfromtxt('file.csv', delimiter=','). An operation (such as addition) can be performed on every element in an array by simply performing it on the array itself. Elements can be selected from arrays using their index and array locations, both of which start at 0. Logical operations can be used to create new, more focused arrays out of larger arrays.

The next lesson will explore how to analyze these arrays and use means, medians, and standard deviations to tell a story. But first, practice what you've learned by working through the following checkpoints.

29 Hypothesis Tests

When we are trying to compare datasets, we often need a way to be confident knowing if datasets are significantly different from each other. Some situations involve correlating numerical data, such as:

a professor expects an exam average to be roughly 75a manager of a chain of stores wants to know if certain locations have different revenues on different days of the week. Are the revenue differences a result of natural fluctuations or a significant difference between the stores' sales patterns? a PM for a website wants to compare the time spent on different versions of a homepage. Does one version make users stay on the page significantly longer?

Others involve categorical data, such as:

a pollster wants to know if men and women have significantly different yogurt flavor preferences. Does a result where men more often answer "chocolate" as their favorite reflect a significant difference in the population? do different age groups have significantly different emotional reactions to different ads?

In this lesson, you will learn how about how we can use hypothesis testing to answer these questions. There are several different types of hypothesis tests for the various scenarios you may encounter. Luckily, SciPy has built-in functions that perform all of these tests for us, normally using just one line of code.

For numerical data, we will cover:

- One Sample T-Tests
- Two Sample T-Tests
- ANOVA
- Tukey Tests

For categorical data, we will cover:

- Binomial Tests
- Chi Square

After this lesson, you will have a wide range of tools in your arsenal to find meaningful correlations in data.

29.1 Sample T-Testing

Let’s imagine the fictional business BuyPie, which sends ingredients for pies to your household, so that you can make them from scratch. Suppose that a product manager wants the average age of visitors to BuyPie.com to be 30. In the past hour, the website had 100 visitors and the average age was 31. Are the visitors too old? Or is this just the result of chance and a small sample size?

We can test this using a univariate T-test. A univariate T-test compares a sample mean to a hypothetical population mean. It answers the question ”What is the probability that the sample came from a distribution with the desired mean?”

When we conduct a hypothesis test, we want to first create a null hypothesis, which is a prediction that there is no significant difference. The null hypothesis that this test examines can be phrased as such: ”The set of samples belongs to a population with the target mean”.

The result of the 1 Sample T Test is a p-value, which will tell us whether or not we can reject this null hypothesis. Generally, if we receive a p-value of less than 0.05, we can reject the null hypothesis and state that there is a significant difference.

SciPy has a function called ttest$_1 samp, whichperformsa1SampleT-Testforyou.$

ttest$_1 samprequirestwoinputs, adistributionofvaluesandanexpectedmean:$

tstat, pval = ttest$_1 samp(example_distribution, expected_mean)printpval$

It also returns two outputs: the t-statistic (which we won’t cover in this course), and the p-value — telling us how confident we can be that the sample of values came from a distribution with the mean specified.

In the last exercise, we got a p-value that was much higher than 0.05, so we cannot reject the null hypothesis. Does this mean that if we wait for more visitors to BuyPie, the average age would definitely be 30 and not 31? Not necessarily. In fact, in this case, we know that the mean of our sample was 31.

P-values give us an idea of how confident we can be in a result. Just because we don’t have enough data to detect a difference doesn’t mean that there isn’t one. Generally, the more samples we have, the smaller a difference we’ll be able to detect. You can learn more about the exact relationship between the number of samples and detectable differences in the Sample Size Determination course.

To gain some intuition on how our confidence levels can change, let’s explore some distributions with different means and how our p-values from the 1 Sample T-Tests change.

29.2 2 Sample T-Test

Suppose that last week, the average amount of time spent per visitor to a website was 25 minutes. This week, the average amount of time spent per visitor to a website was 28 minutes. Did the average time spent per visitor change? Or is this part of natural fluctuations?

One way of testing whether this difference is significant is by using a 2 Sample T-Test. A 2 Sample T-Test compares two sets of data, which are both approximately normally distributed.

The null hypothesis, in this case, is that the two distributions have the same mean.

We can use SciPy's ttest$_i ndfunctiontoperforma2SampleT-Test.Ittakesthetwodistributionsasinputsandr$
$statistic(whichwedon'tuse), andap-value.Ifyoucan'trememberwhatap-valueis, refertotheearlierexerci$
$tests.$

29.3 Dangers of Multiple T-Tests

Suppose that we own a chain of stores that sell ants, called VeryAnts. There are three different locations: A, B, and C. We want to know if the average ant sales over the past year are significantly different between the three locations.

At first, it seems that we could perform T-tests between each pair of stores.

We know that the p-value is the probability that we incorrectly reject the null hypothesis on each t-test. The more t-tests we perform, the more likely that we are to get a false positive, a Type I error.

For a p-value of 0.05, if the null hypothesis is true then the probability of obtaining a significant result is 1 – 0.05 = 0.95. When we run another t-test, the probability of still getting a correct result is 0.95 * 0.95, or 0.9025. That means our probability of making an error is now close to 10

from scipy.stats import ttest$_i ndimportnumpyasnp$

a = np.genfromtxt("store$_a.csv", delimiter = ",")b = np.genfromtxt("store_b.csv", delimiter =$
$",")c = np.genfromtxt("store_c.csv", delimiter = ",")$

a$_m ean = np.mean(a)b_m ean = np.mean(b)c_m ean = np.mean(c)$

a$_s td = np.std(a)b_s td = np.std(b)c_s td = np.std(c)$

a$_{bt}val, a_{bp}val = ttest_i nd(a, b)a_{ct}val, a_{cp}val = ttest_i nd(a, c)c_{bt}val, b_{cp}val = ttest_i nd(b, c)$

print(a$_{bp}val)print(a_{cp}val)print(b_{cp}val)$

error$_p rob = (1 - 0.95 * *3)$

print(error$_p rob)$

29.4 ANOVA

In the last exercise, we saw that the probability of making a Type I error got dangerously high as we performed more t-tests.

When comparing more than two numerical datasets, the best way to preserve a Type I error probability of 0.05 is to use ANOVA. ANOVA (Analysis of Variance) tests the null hypothesis that all of the datasets have the same mean. If we reject the null hypothesis with ANOVA, we're saying that at least one of the sets has a different mean; however, it does not tell us which datasets are different.

We can use the SciPy function f$_{o}neway to perform ANOVA on multiple datasets. It takes in each dataset as adi$
$statistic and the p-value. For example, if we were comparing scores on a videogame between math majors, writ$

fstat, pval = f$_{o}neway(scores_{m}athematicians, scores_{w}riters, scores_{p}sychologists)$

The null hypothesis, in this case, is that all three populations have the same mean score on this videogame. If we reject this null hypothesis (if we get a p-value less than 0.05), we can say that we are reasonably confident that a pair of datasets is significantly different. After using only ANOVA, we can't make any conclusions on which two populations have a significant difference.

Let's look at an example of ANOVA in action.

29.5 Assumptions of Numerical Hypothesis Tests

Before we use numerical hypothesis tests, we need to be sure that the following things are true: 1. The samples should each be normally distributed...ish

Data analysts in the real world often still perform hypothesis on sets that aren't exactly normally distributed. What is more important is to recognize if there is some reason to believe that a normal distribution is especially unlikely. If your dataset is definitively not normal, the numerical hypothesis tests won't work as intended.

For example, imagine we have three datasets, each representing a day of traffic data in three different cities. Each dataset is independent, as traffic in one city should not impact traffic in another city. However, it is unlikely that each dataset is normally distributed. In fact, each dataset probably has two distinct peaks, one at the morning rush hour and one during the evening rush hour. The histogram of a day of traffic data might look something like this:

histogram

In this scenario, using a numerical hypothesis test would be inappropriate. 2. The population standard deviations of the groups should be equal

For ANOVA and 2-Sample T-Tests, using datasets with standard deviations that are significantly different from each other will often obscure the differences in group means.

To check for similarity between the standard deviations, it is normally sufficient to divide the two standard deviations and see if the ratio is "close enough" to 1. "Close enough" may differ in different contexts but generally staying within 103. The samples must be independent

When comparing two or more datasets, the values in one distribution should not affect the values in another distribution. In other words, knowing more about one distribution should not give you any information about any other distribution.

Here are some examples where it would seem the samples are not independent:

the number of goals scored per soccer player before, during, and after undergoing a rigorous training regimen a group of patients' blood pressure levels before, during, and after the administration of a drug

It is important to understand your datasets before you begin conducting hypothesis tests on it so that you know you are choosing the right test.

29.6 Tukeys Range Test

Let's say that we have performed ANOVA to compare three sets of data from the three VeryAnts stores. We received the result that there is some significant difference between datasets.

Now, we have to find out which datasets are different.

We can perform a Tukey's Range Test to determine the difference between datasets.

If we feed in three datasets, such as the sales at the VeryAnts store locations A, B, and C, Tukey's Test can tell us which pairs of locations are distinguishable from each other.

The function to perform Tukey's Range Test is pairwise$_tukeyhsd, whichisfoundinstatsmodel, notscipy.We$

For example, if we were looking to compare mean scores of movies that are dramas, comedies, or documentaries, we would make a call to pairwise$_tukeyhsdlikethis :$

movie$_scores = np.concatenate([drama_scores, comedy_scores, documentary_scores])labels = ['drama']*len(drama_scores)+['comedy']*len(comedy_scores)+['documentary']* len(documentary_scores)$

tukey$_results = pairwise_tukeyhsd(movie_scores, labels, 0.05)$

It will return a table of information, telling you whether or not to reject the null hypothesis for each pair of datasets.

29.7 Binomial Tests

Let's imagine that we are analyzing the percentage of customers who make a purchase after visiting a website. We have a set of 1000 customers from this month, 58 of whom made a purchase. Over the past year, the number of visitors per every 1000 who make a purchase hovers consistently at around 72. Thus, our marketing department has set our target number of purchases per 1000 visits to be 72. We would like to know if this month's number, 58, is a significant difference from that target or a result of natural fluctuations.

How do we begin comparing this, if there's no mean or standard deviation that we can use? The data is divided into two discrete categories, "made a purchase" and "did not make a purchase".

So far, we have been working with numerical datasets. The tests we have looked at, the 1- and 2-Sample T-Tests, ANOVA, and Tukey's Range test, will not work if we can't find the means of our distributions and compare them.

If we have a dataset where the entries are not numbers, but categories instead, we have to use different methods.

To analyze a dataset like this, with two different possibilities for entries, we can use a Binomial Test. A Binomial Test compares a categorical dataset to some expectation.

Examples include:

Comparing the actual percent of emails that were opened to the quarterly goals Comparing the actual percentage of respondents who gave a certain survey response to the expected survey response Comparing the actual number of heads from 1000 coin flips of a weighted coin to the expected number of heads

The null hypothesis, in this case, would be that there is no difference between the observed behavior and the expected behavior. If we get a p-value of less than 0.05, we can reject that hypothesis and determine that there is a difference between the observed and expectation.

SciPy has a function called binom$_{t}est, whichperformsaBinomialTestforyou.$

binom$_{t}estrequiresthreeinputs, thenumberofobservedsuccesses, thenumberoftotaltrials, andanexpectedp$

pval = binom$_{t}est(525, n = 1000, p = 0.5)$

It returns a p-value, telling us how confident we can be that the sample of values was likely to occur with the specified probability. If we get a p-value less than 0.05, we can reject the null hypothesis and say that it is likely the coin is actually weighted, and that the probability of getting heads is statistically different than 0.5.

29.8 Test

You regularly order delivery from two different Pho restaurants, "What the Pho" and "Pho Tonic". You want to know if there's a significant difference between these two restaurants' average time to deliver to your house. What test could you use to determine this? 2 Sample T-Test

Let's say we run a 1 Sample T-Test on means for an exam. We expect the mean to be 75We cannot confidently reject the null-hypothesis, so we do not have enough data to say that the mean on this exam is different from 75

Let's say that last month 7Chi square

Let's say we are comparing the time that users spend on three different versions of a landing page for a website. What test do we use to determine if there is a significant difference between any two of the sets? ANOVA

You own a juice bar and you theorize that 75Binomial Test

You've collected data on 1000 different sites that end with .com, .edu, and .org and have recorded the number of each that have Times New Roman, Helvetica, or another font as their main font. What test can you use to determine if there's a relationship between top-level domain and font type? Chi Square

You've surveyed 10 people who work in finance, 10 people who work in education, and 10 people who work in the service industry on how many cups of coffee they drink per day. What test can you use to determine if there is a significant difference between the average coffee consumption of these three groups? ANOVA

You just bought a new tea kettle that is supposed to heat water to boiling in 2 minutes. What kind of test can you run to determine if the time-to-boil is averaging significantly more than 2 minutes? 1 Sample T Test

If we perform an ANOVA test on 3 datasets and reject the null hypothesis, what test should we perform to determine which pairs of datasets are different? Tukey's Range Test

What kind of test would you use to see if men and women identify differently as "Republican", "Democrat", or "Independent"? Chi Square

30 Pandas in Python

Pandas is a Python module for working with tabular data (i.e., data in a table with rows and columns). Tabular data has a lot of the same functionality as SQL or Excel, but Pandas adds the power of Python.

In order to get access to the Pandas module, we'll need to install the module and then import it into a Python file.

The pandas module is usually imported at the top of a Python file under the alias pd.

import pandas as pd

A DataFrame is an object that stores data as rows and columns. You can think of a DataFrame as a spreadsheet or as a SQL table. You can manually create a DataFrame or fill it with data from a CSV, an Excel spreadsheet, or a SQL query.

DataFrames have rows and columns. Each column has a name, which is a string. Each row has an index, which is an integer. DataFrames can contain many different data types: strings, ints, floats, tuples, etc.

You can pass in a dictionary to pd.DataFrame(). Each key is a column name and each value is a list of column values. The columns must all be the same length or you will get an error. Here's an example:

df1 = pd.DataFrame('name': ['John Smith', 'Jane Doe', 'Joe Schmo'], 'address': ['123 Main St.', '456 Maple Ave.', '789 Broadway'], 'age': [34, 28, 51])

This command creates a DataFrame called df1 that looks like this:

You can also add data using lists.

For example, you can pass in a list of lists, where each one represents a row of data. Use the keyword argument columns to pass a list of column names.

df2 = pd.DataFrame([['John Smith', '123 Main St.', 34], ['Jane Doe', '456 Maple Ave.', 28], ['Joe Schmo', '789 Broadway', 51]], columns=['name', 'address', 'age'])

This command produces a DataFrame df2 that looks like this:

Comma Separated Variables (CSV)

We now know how to create our own DataFrame. However, most of the time, we'll be working with datasets that already exist. One of the most common formats for big datasets is the CSV.

CSV (comma separated values) is a text-only spreadsheet format. You can find CSVs in lots of places:

Online datasets (here's an example from data.gov) Export from Excel or Google Sheets Export from SQL

The first row of a CSV contains column headings. All subsequent rows contain values. Each column heading and each variable is separated by a comma:

column1,column2,column3 value1,value2,value3

That example CSV represents the following table:

Loading and Saving CSVs

When you have data in a CSV, you can load it into a DataFrame in Pandas using .read$_csv()$:

pd.read$_csv('my-csv-file.csv')$

In the example above, the .read$_csv()methodiscalled.TheCSVfilecalledmy-csv-fileispassedinasanargument.$

We can also save data to a CSV, using .to$_csv()$.

df.to$_csv('new-csv-file.csv')$

In the example above, the .to$_csv()methodiscalledondf(whichrepresentsaDataFrameobject).Thenameofth$
$csv{-}file.csv).Bydefault, thismethodwillsavetheCSVfileinyourcurrentdirectory.$

Inspect a DataFrame

When we load a new DataFrame from a CSV, we want to know what it looks like.

If it's a small DataFrame, you can display it by typing print(df).

If it's a larger DataFrame, it's helpful to be able to inspect a few items without having to look at the entire DataFrame.

The method .head() gives the first 5 rows of a DataFrame. If you want to see more rows, you can pass in the positional argument n. For example, df.head(10) would show the first 10 rows.

The method df.info() gives some statistics for each column.

Now we know how to create and load data. Let's select parts of those datasets that are interesting or important to our analyses.

Suppose you have the DataFrame called customers, which contains the ages of your customers: name age Rebecca Erikson 35 Thomas Roberson 28 Diane Ochoa 42 ... ...

Perhaps you want to take the average or plot a histogram of the ages. In order to do either of these tasks, you'd need to select the column.

There are two possible syntaxes for selecting all values from a column:

Select the column as if you were selecting a value from a dictionary using a key. In our example, we would type customers['age'] to select the ages. If the name of a column follows all of the rules for a variable name (doesn't start with a number, doesn't contain spaces or special characters, etc.), then you can select it using the following notation: df.MySecondColumn. In our example, we would type customers.age.

When we select a single column, the result is called a Series.

E.g.

import pandas as pd

df = pd.DataFrame([['January', 100, 100, 23, 100], ['February', 51, 45, 145, 45], ['March', 81, 96, 65, 96], ['April', 80, 80, 54, 180], ['May', 51, 54, 54, 154], ['June', 112, 109, 79, 129]], columns=['month', 'clinic$_east','clinic_north','clinic_south','clinic_west'])$

clinic$_north = df.clinic_northclinic_north = df['clinic_north']$

print(type(clinic$_north))print(type(df))$

Selecting Multiple Columns

When you have a larger DataFrame, you might want to select just a few columns.

For instance, let's return to a DataFrame of orders from ShoeFly.com: id first$_{n}amelast_{n}ameemailshoe_{t}ypeshoe_{m}aterialshoe_{c}olor54791RebeccaLindsayRebeccaLindsay57@hotmai$
$leatherblack53450EmilyJoyceEmilyJoyce25@gmail.comballetflatsfaux-leathernavy91987JoyceWaller$
$leatherred$

We might just be interested in the customer's last$_{n}ameandemail.WewantaDataFramelikethis:$
$last_{n}ameemailLindsayRebeccaLindsay57@hotmail.comJoyceEmilyJoyce25@gmail.comWallerJoyce.W$

To select two or more columns from a DataFrame, we use a list of the column names. To create the DataFrame shown above, we would use:

new$_{d}f = orders[['last_{n}ame','email']]$

*Note: *Make sure that you have a double set of brackets ([[]]), or this command won't work!

E.g. import pandas as pd

df = pd.DataFrame([['January', 100, 100, 23, 100], ['February', 51, 45, 145, 45], ['March', 81, 96, 65, 96], ['April', 80, 80, 54, 180], ['May', 51, 54, 54, 154], ['June', 112, 109, 79, 129]], columns=['month', 'clinic$_{e}ast','clinic_{n}orth','clinic_{s}outh','clinic_{w}est'])$

clinic$_{n}orth_{s}outh = df[['clinic_{n}orth','clinic_{s}outh']]$

print(type(clinic$_{n}orth_{s}outh$))

Select Rows

Let's revisit our orders from ShoeFly.com: id first$_{n}amelast_{n}ameemailshoe_{t}ypeshoe_{m}aterialshoe_{c}olor54791$
$leatherblack53450EmilyJamesEmilyJames25@gmail.comballetflatsfaux-leathernavy91987JoyceWal$
$leatherred...$

Maybe our Customer Service department has just received a message from Joyce Waller, so we want to know exactly what she ordered. We want to select this single row of data.

DataFrames are zero-indexed, meaning that we start with the 0th row and count up from there. Joyce Waller's order is the 2nd row.

We select it using the following command:

orders.iloc[2]

When we select a single row, the result is a Series (just like when we select a single column).

Selecting Multiple Rows

You can also select multiple rows from a DataFrame.

Here are a few more rows from ShoeFly.com's orders DataFrame: id first$_{n}amelast_{n}ameemailshoe_{t}ypeshoe_{m}$
$leatherblack53450EmilyJoyceEmilyJoyce25@gmail.comballetflatsfaux-leathernavy91987JoyceWaller$

*leatherred*79357*AndrewBanksAB*4318@*gmail.combootsleatherbrown*52386*JulieMarshJulieMarsh*59@*g*
*leathernavy*21586*GabrielPorterGabrielPorter*24@*gmail.comclogsleatherbrown*

Here are some different ways of selecting multiple rows:

orders.iloc[3:7] would select all rows starting at the 3rd row and up to but not including the 7th row (i.e., the 3rd row, 4th row, 5th row, and 6th row)

id first$_{n}$*amelast*$_{n}$*ameemailshoe*$_{t}$*ypeshoe*$_{m}$*aterialshoe*$_{c}$*olor*14437*JustinEricksonJustin.Erickson*@*outloo*
*leatherred*79357*AndrewBanksAB*4318@*gmail.combootsleatherbrown*52386*JulieMarshJulieMarsh*59@*g*

orders.iloc[:4] would select all rows up to, but not including the 4th row (i.e., the 0th, 1st, 2nd, and 3rd rows)

id first$_{n}$*amelast*$_{n}$*ameemailshoe*$_{t}$*ypeshoe*$_{m}$*aterialshoe*$_{c}$*olor*54791*RebeccaLindsayRebeccaLindsay*57@*hotm*
*leatherblack*53450*EmilyJoyceEmilyJoyce*25@*gmail.comballetflatsfaux*−*leathernavy*91987*JoyceWalle*
leatherred

orders.iloc[-3:] would select the rows starting at the 3rd to last row and up to and including the final row

id first$_{n}$*amelast*$_{n}$*ameemailshoe*$_{t}$*ypeshoe*$_{m}$*aterialshoe*$_{c}$*olor*20487*ThomasJensenTJ*5470@*gmail.comclogs*
*leathernavy*21586*GabrielPorterGabrielPorter*24@*gmail.comclogsleatherbrown*

Select Rows with Logic I

You can select a subset of a DataFrame by using logical statements:

df[df.MyColumnName == desired$_{c}$*olumn*$_{v}$*alue*]

We have a large DataFrame with information about our customers. A few of the many rows look like this: name address phone age Martha Jones 123 Main St. 234-567-8910 28 Rose Tyler 456 Maple Ave. 212-867-5309 22 Donna Noble 789 Broadway 949-123-4567 35 Amy Pond 98 West End Ave. 646-555-1234 29 Clara Oswald 54 Columbus Ave. 714-225-1957 31 ... ...

Suppose we want to select all rows where the customer's age is 30. We would use:

df[df.age == 30]

In Python, == is how we test if a value is exactly equal to another value.

We can use other logical statements, such as:

Greater Than, ¿ — Here, we select all rows where the customer's age is greater than 30:

df[df.age ¿ 30]

Less Than, ¡ — Here, we select all rows where the customer's age is less than 30:

df[df.age ¡ 30]

Not Equal, != — This snippet selects all rows where the customer's name is not Clara Oswald:

df[df.name != 'Clara Oswald']

Select Rows with Logic II

You can also combine multiple logical statements, as long as each statement is in parentheses.

For instance, suppose we wanted to select all rows where the customer's age was under 30 or the customer's name was "Martha Jones": name address phone age Martha Jones 123 Main St. 234-567-8910 28 Rose Tyler 456 Maple Ave. 212-867-5309 22 Donna Noble 789 Broadway 949-123-4567 35 Amy Pond 98 West End Ave. 646-555-1234 29 Clara Oswald 54 Columbus Ave. 714-225-1957 31 ...

We could use the following code:

df[(df.age ¡ 30) — (df.name == 'Martha Jones')]

In Python, — means "or" and means "and".

Select Rows with Logic III

Suppose we want to select the rows where the customer's name is either "Martha Jones", "Rose Tyler" or "Amy Pond". name address phone age Martha Jones 123 Main St. 234-567-8910 28 Rose Tyler 456 Maple Ave. 212-867-5309 22 Donna Noble 789 Broadway 949-123-4567 35 Amy Pond 98 West End Ave. 646-555-1234 29 Clara Oswald 54 Columbus Ave. 714-225-1957 31 ... ...

We could use the isin command to check that df.name is one of a list of values:

df[df.name.isin(['Martha Jones', 'Rose Tyler', 'Amy Pond'])]

Setting indices

When we select a subset of a DataFrame using logic, we end up with non-consecutive indices. This is inelegant and makes it hard to use .iloc().

We can fix this using the method .reset$_index().Forexample, hereisaDataFramecalleddfwithnon-consecutiveindices : FirstNameLastName0JohnSmith4JaneDoe7JoeSchmo$

If we use the command df.reset$_index(), wegetanewDataFramewithanewsetofindices : indexFirstNameLastName00JohnSmith14JaneDoe27JoeSchmo$

Note that the old indices have been moved into a new column called 'index'. Unless you need those values for something special, it's probably better to use the keyword drop=True so that you don't end up with that extra column. If we run the command df.reset$_index(drop = True), wegetanewDataFramethatlookslikethis : FirstNameLastName0JohnSmith1JaneDoe2JoeSchmo$

Using .reset$_index()willreturnanewDataFrame, butweusuallyjustwanttomodifyyourexistingDataFrame.$
$TruewecanjustmodifyyourexistingDataFrame.$

Review

You've completed the lesson! You've just learned the basics of working with a single table in Pandas, including:

Create a table from scratch Loading data from another file Selecting certain rows or columns of a table

Let's practice what you've learned.

30.1 Modifying Dataframes

Modifying DataFrames

In the previous lesson, you learned what a DataFrame is and how to select subsets of data from one.

In this lesson, you'll learn how to modify an existing DataFrame. Some of the skills you'll learn include:

Adding columns to a DataFrame Using lambda functions to calculate complex quantities Renaming columns

Adding a Column I

Sometimes, we want to add a column to an existing DataFrame. We might want to add new information or perform a calculation based on the data that we already have.

One way that we can add a new column is by giving a list of the same length as the existing DataFrame.

Suppose we own a hardware store called The Handy Woman and have a DataFrame containing inventory information: Product ID Product Description Cost to Manufacture Price 1 3 inch screw 0.50 0.75 2 2 inch nail 0.10 0.25 3 hammer 3.00 5.50 4 screwdriver 2.50 3.00

It looks like the actual quantity of each product in our warehouse is missing!

Let's use the following code to add that information to our DataFrame.

df['Quantity'] = [100, 150, 50, 35]

Our new DataFrame looks like this: Product ID Product Description Cost to Manufacture Price Quantity 1 3 inch screw 0.50 0.75 100 2 2 inch nail 0.10 0.25 150 3 hammer 3.00 5.50 50 4 screwdriver 2.50 3.00 35

Adding a Column II

We can also add a new column that is the same for all rows in the DataFrame. Let's return to our inventory example: Product ID Product Description Cost to Manufacture Price 1 3 inch screw 0.50 0.75 2 2 inch nail 0.10 0.25 3 hammer 3.00 5.50 4 screwdriver 2.50 3.00

Suppose we know that all of our products are currently in-stock. We can add a column that says this:

df['In Stock?'] = True

Now all of the rows have a column called In Stock? with value True. Product ID Product Description Cost to Manufacture Price In Stock? 1 3 inch screw 0.50 0.75 True 2 2 inch nail 0.10 0.25 True 3 hammer 3.00 5.50 True 4 screwdriver 2.50 3.00 True

Adding a Column III

Finally, you can add a new column by performing a function on the existing columns.

Maybe we want to add a column to our inventory table with the amount of sales tax that we need to charge for each item. The following code multiplies each Price by 0.075, the sales tax for our state:

df['Sales Tax'] = df.Price * 0.075

Now our table has a column called Sales Tax: Product ID Product Description Cost to Manufacture Price Sales Tax 1 3 inch screw 0.50 0.75 0.06 2 2 inch nail 0.10 0.25 0.02 3 hammer 3.00 5.50 0.41 4 screwdriver 2.50 3.00 0.22

Performing Column Operations

In the previous exercise, we learned how to add columns to a DataFrame.

Often, the column that we want to add is related to existing columns, but requires a calculation more complex than multiplication or addition.

For example, imagine that we have the following table of customers. Name Email JOHN SMITH john.smith@gmail.com Jane Doe jdoe@yahoo.com joe schmo joeschmo@hotmail.com

It's a little annoying that the capitalization is different for each row. Perhaps we'd like to make it more consistent by making all of the letters uppercase.

We can use the apply function to apply a function to every value in a particular column. For example, this code overwrites the existing 'Name' columns by applying the function upper to every row in 'Name'.

from string import upper

df['Name'] = df.Name.apply(upper)

The result: Name Email JOHN SMITH john.smith@gmail.com JANE DOE jdoe@yahoo.com JOE SCHMO joeschmo@hotmail.com

Reviewing Lambda Function

A lambda function is a way of defining a function in a single line of code. Usually, we would assign them to a variable.

For example, the following lambda function multiplies a number by 2 and then adds 3:

mylambda = lambda x: (x * 2) + 3 print(mylambda(5))

The output:

¿ 13

Lambda functions work with all types of variables, not just integers! Here is an example that takes in a string, assigns it to the temporary variable x, and then converts it into lowercase:

stringlambda = lambda x: x.lower() print(stringlambda(”Oh Hi Mark!”))

The output:

¿ ”oh hi mark!”

Learn more about lambda functions in this article!

Reviewing Lambda Function: If Statements

We can make our lambdas more complex by using a modified form of an if statement.

Suppose we want to pay workers time-and-a-half for overtime (any work above 40 hours per week). The following function will convert the number of hours into time-and-a-half hours using an if statement:

def myfunction(x): if x ¿ 40: return 40 + (x - 40) * 1.50 else: return x

Below is a lambda function that does the same thing:

myfunction = lambda x: 40 + (x - 40) * 1.50 if x ¿ 40 else x

In general, the syntax for an if function in a lambda function is:

lambda x: [OUTCOME IF TRUE] if [CONDITIONAL] else [OUTCOME IF FALSE]

Applying a Lambda to a Column

In Pandas, we often use lambda functions to perform complex operations on columns. For example, suppose that we want to create a column containing the email provider for each email address in the following table: Name Email JOHN SMITH john.smith@gmail.com Jane Doe jdoe@yahoo.com joe schmo joeschmo@hotmail.com

We could use the following code with a lambda function:

df['Email Provider'] = df.Email.apply(lambda x: x.split('@')[-1])

The result would be: Name Email Email Provider JOHN SMITH john.smith@gmail.com gmail.com Jane Doe jdoe@yahoo.com yahoo.com joe schmo joeschmo@hotmail.com hotmail.com

Applying a Lambda to a Row

We can also operate on multiple columns at once. If we use apply without specifying a single column and add the argument axis=1, the input to our lambda function will be an entire row, not a column. To access particular values of the row, we use the syntax row.column$_{n}ameorrow[‘column_{n}ame'].$

Suppose we have a table representing a grocery list: Item Price Is taxed? Apple 1.00 No Milk 4.20 No Paper Towels 5.00 Yes Light Bulbs 3.75 Yes

If we want to add in the price with tax for each line, we'll need to look at two columns: Price and Is taxed?.

If Is taxed? is Yes, then we'll want to multiply Price by 1.075 (for 7.5

If Is taxed? is No, we'll just have Price without multiplying it.

We can create this column using a lambda function and the keyword axis=1:

df['Price with Tax'] = df.apply(lambda row: row['Price'] * 1.075 if row['Is taxed?'] == 'Yes' else row['Price'], axis=1)

Renaming Columns

When we get our data from other sources, we often want to change the column names. For example, we might want all of the column names to follow variable name rules, so that we can use df.column$_{n}ame(whichtab-completes)ratherthandf['column_{n}ame'](whichta$

You can change all of the column names at once by setting the .columns property to a different list. This is great when you need to change all of the column names at once, but be careful! You can easily mislabel columns if you get the ordering wrong. Here's an example:

df = pd.DataFrame('name': ['John', 'Jane', 'Sue', 'Fred'], 'age': [23, 29, 21, 18]) df.columns = ['First Name', 'Age']

This command edits the existing DataFrame df.

Renaming Columns II

You also can rename individual columns by using the .rename method. Pass a dictionary like the one below to the columns keyword argument:

'old$_{c}olumn_{n}ame1' :' new_{c}olumn_{n}ame1',' old_{c}olumn_{n}ame2' :' new_{c}olumn_{n}ame2'$

Here's an example:

df = pd.DataFrame('name': ['John', 'Jane', 'Sue', 'Fred'], 'age': [23, 29, 21, 18]) df.rename(columns= 'name': 'First Name', 'age': 'Age', inplace=True)

The code above will rename name to First Name and age to Age.

Using rename with only the columns keyword will create a new DataFrame, leaving your original DataFrame unchanged. That's why we also passed in the keyword argument inplace=True. Using inplace=True lets us edit the original DataFrame.

There are several reasons why .rename is preferable to .columns:

You can rename just one column You can be specific about which column names are getting changed (with .column you can accidentally switch column names if you're not careful)

*Note: *If you misspell one of the original column names, this command won't fail. It just won't change anything.

Review

Great job! In this lesson, you learned how to modify an existing DataFrame. Some of the skills you've learned include:

Adding columns to a DataFrame Using lambda functions to calculate complex quantities Renaming columns

Let's practice what you just learned!

example project

import pandas as pd

inventory = pd.read$_c sv('inventory.csv')$

print(inventory.head(10))

staten$_i sland = inventory.head(10)$

product$_r equest = staten_i sland['product_d escription']seed_r equest = staten_i sland[staten_i sland.product_t ype$
$seeds']$

print(product$_r equest)print(seed_r equest)$

inventory["in$_s tock"] = inventory.quantity > 0inventory["total_v alue"] = inventory.price*$
$inventory.quantitycombine_l ambda = lambdarow :' -'.format(row.product_t ype, row.product_d escription)$

inventory['full$_d escription'] = inventory.apply(combine_l ambda, axis = 1)$

print(inventory.head())

30.2 Aggregates in Pandas

Introduction

This lesson you will learn about aggregates in Pandas. An aggregate statistic is a way of creating a single number that describes a group of numbers. Common aggregate statistics incluse mean, median, or standard deviation.

You will also learn how to rearrange a DataFrame into a pivot table, which is a great way to compare data across two dimensions.

Calculating Column Statistics

In the previous lesson, you learned how to perform operations on each value in a column using apply.

In this exercise, you will learn how to combine all of the values from a column for a single calculation.

Some examples of this type of calculation include:

The DataFrame customers contains the names and ages of all of your customers. You want to find the median age:

print(customers.age) ¿¿ [23, 25, 31, 35, 35, 46, 62] print(customers.age.median()) ¿¿ 35

The DataFrame shipments contains address information for all shipments that you've sent out in the past year. You want to know how many different states you have shipped to (and how many shipments went to the same state).

print(shipments.state) ¿¿ ['CA', 'CA', 'CA', 'CA', 'NY', 'NY', 'NJ', 'NJ', 'NJ', 'NJ', 'NJ', 'NJ', 'NJ'] print(shipments.state.nunique()) ¿¿ 3

The DataFrame inventory contains a list of types of t-shirts that your company makes. You want a list of the colors that your shirts come in.

print(inventory.color) ¿¿ ['blue', 'blue', 'blue', 'blue', 'blue', 'green', 'green', 'orange', 'orange', 'orange'] print(inventory.color.unique()) ¿¿ ['blue', 'green', 'orange']

The general syntax for these calculations is:

df.column$_name.command()$

The following table summarizes some common commands: Command Description mean Average of all values in column std Standard deviation median Median max Maximum value in column min Minimum value in column count Number of values in column nunique Number of unique values in column unique List of unique values in column

Calculating Aggregate Functions I

When we have a bunch of data, we often want to calculate aggregate statistics (mean, standard deviation, median, percentiles, etc.) over certain subsets of the data.

Suppose we have a grade book with columns student, assignment$_name, andgrade.Thefirstfewlineslooklike$ $studentassignment_namegradeAmyAssignment175AmyAssignment235BobAssignment199BobAssignme$

We want to get an average grade for each student across all assignments. We could do some sort of loop, but Pandas gives us a much easier option: the method .groupby.

For this example, we'd use the following command:

grades = df.groupby('student').grade.mean()

The output might look something like this: student grade Amy 80 Bob 90 Chris 75 ...

In general, we use the following syntax to calculate aggregates:

df.groupby('column1').column2.measurement()

where:

column1 is the column that we want to group by ('student' in our example) column2 is the column that we want to perform a measurement on (grade in our example) measurement is the measurement function we want to apply (mean in our example)

Calculating Aggregate Functions II

After using groupby, we often need to clean our resulting data.

As we saw in the previous exercise, the groupby function creates a new Series, not a DataFrame. For our ShoeFly.com example, the indices of the Series were different values of shoe$_t ype, andthenamepropertywasprice.$

Usually, we'd prefer that those indices were actually a column. In order to get that, we can use reset$_i ndex().ThiswilltransformourSeriesintoaDataFrameandmovetheindicesintotheir$

Generally, you'll always see a groupby statement followed by reset$_i ndex:$

df.groupby('column1').column2.measurement() .reset$_i ndex()$

When we use groupby, we often want to rename the column we get as a result. For example, suppose we have a DataFrame teas containing data on types of tea: id tea category caffeine price 0 earl grey black 38 3 1 english breakfast black 41 3 2 irish breakfast black 37 2.5 3 jasmine green 23 4.5 4 matcha green 48 5 5 camomile herbal 0 3 ...

We want to find the number of each category of tea we sell. We can use:

teas$_c ounts = teas.groupby('category').id.count().reset_i ndex()$

This yields a DataFrame that looks like: category id 0 black 3 1 green 4 2 herbal 8 3 white 2 ...

The new column contains the counts of each category of tea sold. We have 3 black teas, 4 green teas, and so on. However, this column is called id because we used the id column of teas to calculate the counts. We actually want to call this column counts. Remember that we can rename columns:

teas$_c ounts = teas_c ounts.rename(columns = "id" : "counts")$

Our DataFrame now looks like: category counts 0 black 3 1 green 4 2 herbal 8 3 white 2 ...

Calculating Aggregate Functions III

Sometimes, the operation that you want to perform is more complicated than mean or count. In those cases, you can use the apply method and lambda functions, just like we did for individual column operations. Note that the input to our lambda function will always be a list of values.

A great example of this is calculating percentiles. Suppose we have a DataFrame of employee information called df that has the following columns:

id: the employee's id number name: the employee's name wage: the employee's hourly wage category: the type of work that the employee does

Our data might look something like this: id name wage category 10131 Sarah Carney 39 product 14189 Heather Carey 17 design 15004 Gary Mercado 33 marketing 11204 Cora Copaz 27 design ...

If we want to calculate the 75th percentile (i.e., the point at which 75

np.percentile can calculate any percentile over an array of values high$_earners = df.groupby('category').wage.apply(lambdax : np.percentile(x, 75)).reset_index()$

The output, high$_earnersmightlooklikethis : categorywage0design231marketing352product48...$

Calculating Aggregate Functions IV

Sometimes, we want to group by more than one column. We can easily do this by passing a list of column names into the groupby method.

Imagine that we run a chain of stores and have data about the number of sales at different locations on different days: Location Date Day of Week Total Sales West Village February 1 W 400 West Village February 2 Th 450 Chelsea February 1 W 375 Chelsea February 2 Th 390

We suspect that sales are different at different locations on different days of the week. In order to test this hypothesis, we could calculate the average sales for each store on each day of the week across multiple months. The code would look like this:

df.groupby(['Location', 'Day of Week'])['Total Sales'].mean().reset$_index()$

The results might look something like this: Location Day of Week Total Sales Chelsea M 402.50 Chelsea Tu 422.75 Chelsea W 452.00 ... West Village M 390 West Village Tu 400 ...

Pivot Tables

When we perform a groupby across multiple columns, we often want to change how our data is stored. For instance, recall the example where we are running a chain of stores and have data about the number of sales at different locations on different days: Location Date Day of Week Total Sales West Village February 1 W 400 West Village February 2 Th 450 Chelsea February 1 W 375 Chelsea February 2 Th 390 We suspected that there might be different sales on different days of the week at different stores, so we performed a groupby across two different columns (Location and Day of Week). This gave us results that looked like this: Location Day of Week Total Sales Chelsea M 300 Chelsea Tu 310 Chelsea W 320 Chelsea Th 290 ... West Village Th 400 West Village F 390 West Village Sa 250 ... In order to test our hypothesis, it would be more useful if the table was formatted like this: Location M Tu W Th F Sa Su Chelsea 400 390 250 275 300 150 175 West Village 300 310 350 400 390 250 200 ...

Reorganizing a table in this way is called pivoting. The new table is called a pivot table.

In Pandas, the command for pivot is:

df.pivot(columns='ColumnToPivot', index='ColumnToBeRows', values='ColumnToBeValues')

For our specific example, we would write the command like this:

First use the groupby statement: unpivoted = df.groupby(['Location', 'Day of Week'])['Total Sales'].mean().reset$_index()Nowpivotthetablepivoted = unpivoted.pivot(columns =' DayofWeek', index =' Location', values =' TotalSales')$

Just like with groupby, the output of a pivot command is a new DataFrame, but the indexing tends to be "weird", so we usually follow up with .reset$_index()$.

Review

This lesson introduced you to aggregates in Pandas. You learned:

How to perform aggregate statistics over individual rows with the same value using groupby. How to rearrange a DataFrame into a pivot table, a great way to compare data across two dimensions.

import codecademylib import pandas as pd

user$_visits = pd.read_csv('page_visits.csv')$

print(user$_visits.head())$

click$_source = user_visits.groupby(["utm_source"]).id.count().reset_index()$

print(click$_source)$

click$_source_by_month = user_visits.groupby(["utm_source", "month"]).id.count().reset_index()$

print(click$_source_by_month)$

click$_source_by_month_pivot = click_source_by_month.pivot(columns = "month", index = "utm_source", values = "id").reset_index()$

print(click$_source_by_month_pivot)$