COURSE 19 | SAMPLING AND POINT IN PYTHON

Lawal’s Note

2024-12-26

Table of contents



## 1 Chapter 1: Introduction to Sampling

Learn what sampling is and why it is so powerful. You’ll also learn about the problems caused by convenience sampling and the differences between true randomness and pseudo-randomness.

### 1.1 Chapter 1.1: Sampling and point estimates

Hi! Welcome to the course! I’m James, and I’ll be your host as we delve into the world of sampling data with Python. To start, let’s look at what sampling is and why it might be useful.

#### Estimating the population of France

Let’s consider the problem of counting how many people live in France. The standard approach is to take a census. This means contacting every household and asking how many people live there. There are lots of people in France. Since there are millions of people in France, this is a really expensive process. Even with modern data collection technology, most countries will only conduct a census every five or ten years due to the cost.

#### Sampling households

In 1786, Pierre-Simon Laplace realized you could estimate the population with less effort. Rather than asking every household who lived there, he asked a small number of households and used statistics to estimate the number of people in the whole population. This technique of working with a subset of the whole population is called sampling.

#### Population vs. sample

Two definitions are important for this course. The population is the complete set of data that we are interested in. The previous example involved the literal population of France, but in statistics, it doesn’t have to refer to people. One thing to bear in mind is that there is usually no equivalent of the census, so typically, we won’t know what the whole population is like - more on this in a moment. The sample is the subset of data that we are working with.

#### Coffee rating dataset

Picture a dataset of professional ratings of coffees. Each row corresponds to one coffee, and there are thirteen hundred and thirty-eight rows in the dataset. The coffee is given a score from zero to one hundred, which is stored in the total\_cup\_points column. Other columns contain contextual information like the variety and country of origin and scores between zero and ten for attributes of the coffee such as aroma and body. These scores are averaged across all the reviewers for that particular coffee. It doesn’t contain every coffee in the world, so we don’t know exactly what the population of coffees is. However, there are enough here that we can think of it as our population of interest.

#### Points vs. flavor: population

Let’s consider the relationship between cup points and flavor by selecting those two columns. This dataset contains all thirteen hundred and thirty-eight rows from the original dataset.

#### Points vs. flavor: 10 row sample

The pandas .sample method returns a random subset of rows. Setting n to ten means ten random rows are returned. By default, rows from the original dataset can’t appear in the sample dataset multiple times, so we are guaranteed to have ten unique rows in our sample.

#### Python sampling for Series

The .sample method also works on pandas Series. Here, using square-bracket subsetting retrieves the total\_cup\_points column as a Series, and the n argument specifies how many random values to return.

#### Population parameters & point estimates

A population parameter is a calculation made on the population dataset. We aren’t limited to counting values either; here, we calculate the mean of the cup points using NumPy. By contrast, a point estimate, or sample statistic, is a calculation based on the sample dataset. Here, the mean of the total cup points is calculated on the sample. Notice that the means are very similar but not identical.

#### Point estimates with pandas

Working with pandas can be easier than working with NumPy. These mean calculations can be performed using the .mean pandas method.

### 1.2 Exercise 1.1.1

#### Simple sampling with pandas

Throughout this chapter, you’ll be exploring song data from Spotify. Each row of this population dataset represents a song, and there are over 40,000 rows. Columns include the song name, the artists who performed it, the release year, and attributes of the song like its duration, tempo, and danceability. You’ll start by looking at the durations.

Your first task is to sample the Spotify dataset and compare the mean duration of the population with the sample.

#### Instructions

1. Sample 1000 rows from spotify, assigning to spotify\_sample.
2. Calculate the mean duration in minutes from spotify using pandas.
3. Calculate the mean duration in minutes from spotify\_sample using pandas.

# Importing pandas
import pandas as pd

# Importing the course arrays
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

# Sample 1000 rows from spotify\_population
spotify\_sample = spotify.sample(n=1000)

# Print the sample
print(spotify\_sample)

# Calculate the mean duration in mins from spotify\_population
mean\_dur\_pop = spotify['duration\_minutes'].mean()

# Calculate the mean duration in mins from spotify\_sample
mean\_dur\_samp = spotify\_sample['duration\_minutes'].mean()

# Print the means
print(mean\_dur\_pop)
print(mean\_dur\_samp)

 acousticness artists danceability duration\_ms \
2626 0.07680 ['Selena'] 0.537 279000.0
33257 0.47200 ['Hailee Steinfeld'] 0.378 248987.0
22464 0.22800 ['Post Malone'] 0.665 179613.0
39474 0.12400 ['YG'] 0.925 174763.0
39745 0.00844 ['Staind'] 0.515 293133.0
... ... ... ... ...
6040 0.04020 ['Logic', 'Wiz Khalifa'] 0.742 260320.0
7042 0.37800 ['Rockabye Baby!'] 0.756 283387.0
29796 0.42800 ['The Cheetah Girls'] 0.674 198360.0
34056 0.01080 ['Flying Lotus'] 0.603 167173.0
10862 0.17300 ['Joe Arroyo'] 0.714 276200.0

 duration\_minutes energy explicit id \
2626 4.650000 0.652 0.0 2BIfTo4TpRuWZoebrkQ1oX
33257 4.149783 0.388 0.0 7GCVboEDzfL3NKp1NrAgHR
22464 2.993550 0.498 1.0 5yuShbu70mtHXY0yLzCQLQ
39474 2.912717 0.400 1.0 3I8lepBVkxvZTJSpjqxY7A
39745 4.885550 0.594 0.0 0kh03T0zqCUJ8CoLESa26N
... ... ... ... ...
6040 4.338667 0.720 1.0 0jqBo5RYn008f4ZY8kPewW
7042 4.723117 0.422 0.0 04fVTZt9DDgOPbi7H5Sr7V
29796 3.306000 0.870 0.0 1g1Jor1zrllXn2ogj8KGAH
34056 2.786217 0.606 0.0 6OHWgUuDDedHRVhcg8vlaf
10862 4.603333 0.645 0.0 5jAvZlrknZT8rZlMVUsKvL

 instrumentalness key liveness loudness mode \
2626 0.003300 1.0 0.0536 -10.632 0.0
33257 0.000000 10.0 0.1570 -9.017 1.0
22464 0.000000 5.0 0.0757 -8.185 1.0
39474 0.000043 4.0 0.1220 -9.783 0.0
39745 0.006580 1.0 0.0752 -4.708 1.0
... ... ... ... ... ...
6040 0.000846 0.0 0.0839 -2.876 1.0
7042 0.155000 1.0 0.0666 -11.888 1.0
29796 0.000005 3.0 0.1270 -5.158 0.0
34056 0.001570 3.0 0.1080 -5.060 0.0
10862 0.000002 7.0 0.0782 -8.216 1.0

 name popularity release\_date \
2626 I Could Fall In Love 48.0 2005-01-01
33257 Wrong Direction 76.0 2020-01-01
22464 Go Flex 80.0 2016-12-09
39474 Laugh Now Kry Later! 68.0 2020-05-01
39745 Outside 41.0 2001-05-22
... ... ... ...
6040 Indica Badu 63.0 2018-03-09
7042 All I want Is You 46.0 2007-01-30
29796 Strut - From "The Cheetah Girls 2" 53.0 2006-01-01
34056 Massage Situation 52.0 2007-10-01
10862 Tal Para Cual 47.0 2003-02-14

 speechiness tempo valence year
2626 0.0316 158.003 0.5420 2005.0
33257 0.0360 72.293 0.1880 2020.0
22464 0.0832 81.967 0.1270 2016.0
39474 0.3180 110.063 0.0978 2020.0
39745 0.0262 146.564 0.2180 2001.0
... ... ... ... ...
6040 0.1160 77.011 0.9700 2018.0
7042 0.0790 81.998 0.8060 2007.0
29796 0.0912 120.027 0.5590 2006.0
34056 0.0688 93.060 0.4030 2007.0
10862 0.0377 118.363 0.8420 2003.0

[1000 rows x 20 columns]
3.8521519140900073
3.8010173333333337

|  |
| --- |
|  Note |
| *Notice that the mean song duration in the sample is similar, but not identical to the mean song duration in the whole population.* |

### 1.3 Exercise 1.1.2

#### Simple sampling and calculating with NumPy

You can also use numpy to calculate parameters or statistics from a list or pandas Series.

You’ll be turning it up to eleven and looking at the loudness property of each song.

#### Instructions

1. Create a pandas Series, loudness\_pop, by subsetting the loudness column from spotify.
* Sample loudness\_pop to get 100 random values, assigning to loudness\_samp.
1. Calculate the mean of loudness\_pop using numpy.
2. Calculate the mean of loudness\_samp using numpy.

# Importing pandas
import pandas as pd
import numpy as np

# Importing the course arrays
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

# Create a pandas Series from the loudness column of spotify\_population
loudness\_pop = spotify['loudness']

# Sample 100 values of loudness\_pop
loudness\_samp = loudness\_pop.sample(n=100)

print(loudness\_samp)

# Calculate the mean of loudness\_pop
mean\_loudness\_pop = np.mean(loudness\_pop)

# Calculate the mean of loudness\_samp
mean\_loudness\_samp = np.mean(loudness\_samp)

print(mean\_loudness\_pop)
print(mean\_loudness\_samp)

26730 -7.565
4420 -7.089
19764 -8.391
34349 -8.514
14736 -7.869
 ...
36632 -9.327
4595 -10.331
33082 -5.778
26469 -3.804
34078 -3.733
Name: loudness, Length: 100, dtype: float64
-7.366856851353947
-7.82295

|  |
| --- |
|  Note |
| *Again, notice that the calculated value (the mean) is close but not identical in each case.* |

### 1.4 Chapter 1.2: Convenience sampling

The point estimates you calculated in the previous exercises were very close to the population parameters that they were based on, but is this always the case?

#### The Literary Digest election prediction

In 1936, a newspaper called The Literary Digest ran an extensive poll to try to predict the next US presidential election. They phoned ten million voters and had over two million responses. About one-point-three million people said they would vote for Landon, and just under one million people said they would vote for Roosevelt. That is, Landon was predicted to get fifty-seven percent of the vote, and Roosevelt was predicted to get forty-three percent of the vote. Since the sample size was so large, it was presumed that this poll would be very accurate. However, in the election, Roosevelt won by a landslide with sixty-two percent of the vote. So what went wrong? Well, in 1936, telephones were a luxury, so the only people who had been contacted by The Literary Digest were relatively rich. The sample of voters was not representative of the whole population of voters, and so the poll suffered from sample bias. The data was collected by the easiest method, in this case, telephoning people. This is called convenience sampling and is often prone to sample bias. Before sampling, we need to think about our data collection process to avoid biased results.

#### Finding the mean age of French people

Let’s look at another example. While on vacation at Disneyland Paris, you start wondering about the mean age of French people. To get an answer, you ask ten people stood nearby about their ages. Their mean age is twenty-four-point-six years old. Do you think this will be a good estimate of the mean age of all French citizens?

#### How accurate was the survey?

On the left, you can see mean ages taken from the French census. Notice that the population has been gradually getting older as birth rates decrease and life expectancy increases. In 2015, the mean age was over forty, so our estimate of twenty-four-point-six is way off. The problem is that the family-friendly fun at Disneyland means that the sample ages weren’t representative of the general population. There are generally more eight-year-olds than eighty-year-olds riding rollercoasters.

#### Convenience sampling coffee ratings

Let’s return to the coffee ratings dataset and look at the mean cup points population parameter. The mean is about eighty-two. One form of convenience sampling would be to take the first ten rows, rather than the random rows we saw in the previous video. We can take the first 10 rows with the pandas head method. The mean cup points from this sample is higher at eighty-nine. The discrepancy suggests that coffees with higher cup points appear near the start of the dataset. Again, the convenience sample isn’t representative of the whole population.

#### Visualizing selection bias

Histograms are a great way to visualize the selection bias. We can create a histogram of the total cup points from the population, which contains values ranging from around 59 to around 91. The np.arange function can be used to create bins of width 2 from 59 to 91. Recall that the stop value in np.arange is exclusive, so we specify 93, not 91. Here’s the same code to generate a histogram for the convenience sample.

#### Distribution of a population and of a convenience sample

Comparing the two histograms, it is clear that the distribution of the sample is not the same as the population: all of the sample values are on the right-hand side of the plot.

#### Visualizing selection bias for a random sample

This time, we’ll compare the total\_cup\_points distribution of the population with a random sample of 10 coffees.

#### Distribution of a population and of a simple random sample

Notice how the shape of the distributions is more closely aligned when random sampling is used.

### 1.5 Exercise 1.2.1

#### Are findings from the sample generalizable?

You just saw how convenience sampling—collecting data using the easiest method—can result in samples that aren’t representative of the population. Equivalently, this means findings from the sample are not generalizable to the population. Visualizing the distributions of the population and the sample can help determine whether or not the sample is representative of the population.

The Spotify dataset contains an acousticness column, which is a confidence measure from zero to one of whether the track was made with instruments that aren’t plugged in. You’ll compare the acousticness distribution of the total population of songs with a sample of those songs.

#### Instructions

1. Plot a histogram of the acousticness from spotify with bins of width 0.01 from 0 to 1 using pandas .hist().
2. Update the histogram code to use the spotify\_mysterious\_sample dataset.

# Importing pandas
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course arrays
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

# Visualize the distribution of acousticness with a histogram
spotify['acousticness'].hist(bins=np.arange(0,1.01,0.01))
plt.show()

# Generate a convenience sample where acousticness is consistently higher
spotify\_high\_acousticness = spotify[(spotify['acousticness'] >= 0.85) & (spotify['acousticness'] <= 1.0)]

# Sample 1107 entries from the high acousticness subset
spotify\_mysterious\_sample = spotify\_high\_acousticness.sample(n=1107)

# Update the histogram to use spotify\_mysterious\_sample
spotify\_mysterious\_sample['acousticness'].hist(bins=np.arange(0, 1.01, 0.01))
plt.show()





### 1.6 Question

*Compare the two histograms you drew*. Are the acousticness values in the sample generalizable to the general population?

**No. The acousticness samples are consistently higher than those in the general population.**

**The acousticness values in the sample are all greater than 0.85, whereas they range from 0 to 1 in the whole population.**

### 1.7 Exercise 1.2.2

#### Are these findings generalizable?

Let’s look at another sample to see if it is representative of the population. This time, you’ll look at the duration\_minutes column of the Spotify dataset, which contains the length of the song in minutes.

#### Instructions

* Plot a histogram of duration\_minutes from spotify with bins of width 0.5 from 0 to 15 using pandas .hist().
* Update the histogram code to use the spotify\_mysterious\_sample2 dataset.

# Importing pandas
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course arrays
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

# Generate a convenience sample where duration\_minutes is within the specified range
spotify\_duration\_range = spotify[(spotify['duration\_minutes'] >= 0.8079999999) & (spotify['duration\_minutes'] <= 9.822)]

# Sample 50 entries from the spotify\_mysterious\_sample2 dataset
spotify\_mysterious\_sample2 = spotify\_duration\_range.sample(n=50)

# Visualize the distribution of duration\_minutes in the population with a histogram
spotify['duration\_minutes'].hist(bins=np.arange(0,15.5,0.5))
plt.show()

# Visualize the distribution of duration\_minutes as a histogram
spotify\_mysterious\_sample2['duration\_minutes'].hist(bins=np.arange(0, 15.5, 0.5))
plt.show()





### 1.8 Question

*Compare the two histograms you drew*. Are the duration values in the sample generalizable to the general population?

#### 1.8.1 Answer

Yes. The sample selected is likely a random sample of all songs in the population.

*The duration values in the sample show a similar distribution to those in the whole population, so the results are generalizable.*

### 1.9 Chapter 1.3: Pseudo-random number generation

You previously saw how to use a random sample to get results similar to those in the population. But how does a computer actually do this random sampling?

#### What does random mean?

There are several meanings of random in English. This definition from Oxford Languages is the most interesting for us. If we want to choose data points at random from a population, we shouldn’t be able to predict which data points would be selected ahead of time in some systematic way.

#### True random numbers

To generate truly random numbers, we typically have to use a physical process like flipping coins or rolling dice. The Hotbits service generates numbers from radioactive decay, and RANDOM.ORG generates numbers from atmospheric noise, which are radio signals generated by lightning. Unfortunately, these processes are fairly slow and expensive for generating random numbers.

<https://www.fourmilab.ch/hotbits>

<https://www.random.org>

#### Pseudo-random number generation

For most use cases, pseudo-random number generation is better since it is cheap and fast. Pseudo-random means that although each value appears to be random, it is actually calculated from the previous random number. Since you have to start the calculations somewhere, the first random number is calculated from what is known as a seed value. The word random is in quotes to emphasize that this process isn’t really random. If we start from a particular seed value, all future numbers will be the same.

#### Pseudo-random number generation example

For example, suppose we have a function to generate pseudo-random values called calc\_next\_random. To begin, we pick a seed number, in this case, one. calc\_next\_random does some calculations and returns three. We then feed three into calc\_next\_random, and it does the same set of calculations and returns two. And if we can keep feeding in the last number, it will return something apparently random. Although the process is deterministic, the trick to a random number generator is to make it look like the values are random.

#### Random number generating functions

NumPy has many functions for generating random numbers from statistical distributions. To use each of these, make sure to prepend each function name with numpy.random or np.random. Some of them, like .uniform and .normal, may be familiar. Others have more niche applications.

#### Visualizing random numbers

Let’s generate some pseudo-random numbers. The first arguments to each random number function specify distribution parameters. The size argument specifies how many numbers to generate, in this case, five thousand. We’ve chosen the beta distribution, and its parameters are named a and b. These random numbers come from a continuous distribution, so a great way to visualize them is with a histogram. Here, because the numbers were generated from the beta distribution, all the values are between zero and one.

#### Random numbers seeds

To set a random seed with NumPy, we use the .random.seed method. Random.seed takes an integer for the seed number, which can be any number you like. .normal generates pseudo-random numbers from the normal distribution. The loc and scale arguments set the mean and standard deviation of the distribution, and the size argument determines how many random numbers from that distribution will be returned. If we call .normal a second time, we get two different random numbers. If we reset the seed by calling random.seed with the same seed number, then call .normal again, we get the same numbers as before. This makes our code reproducible.

#### Using a different seed

Now let’s try a different seed. This time, calling .normal generates different numbers.

### 1.10 Exercise 1.3.1

#### Generating random numbers

You’ve used .sample() to generate pseudo-random numbers from a set of values in a DataFrame. A related task is to generate random numbers that follow a statistical distribution, like the uniform distribution or the normal distribution.

Each random number generation function has distribution-specific arguments and an argument for specifying the number of random numbers to generate.

#### Instructions

1. Generate 5000 numbers from a uniform distribution, setting the parameters low to -3 and high to 3.
2. Generate 5000 numbers from a normal distribution, setting the parameters loc to 5 and scale to 2.
3. Plot a histogram of uniforms with bins of width of 0.25 from -3 to 3 using plt.hist().
4. Plot a histogram of normals with bins of width of 0.5 from -2 to 13 using plt.hist().

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Generate random numbers from a Uniform(-3, 3)
uniforms = np.random.uniform(low=-3, high=3, size=5000)

# Print uniforms
print(uniforms)

# Generate random numbers from a Normal(5, 2)
normals = np.random.normal(loc=5, scale = 2, size= 5000)

# Print normals
print(normals)

# Plot a histogram of uniform values, binwidth 0.25
plt.hist(uniforms, bins=np.arange(-3,3.25,0.25))
plt.show()

# Plot a histogram of normal values, binwidth 0.5
plt.hist(normals, bins = np.arange(-2, 13.5, 0.5))
plt.show()

[ 2.96141096 -2.20215732 2.48169077 ... -0.34643415 -1.68871965
 0.49136264]
[7.8435651 7.06272169 6.08152619 ... 6.09734946 3.68504811 4.00530689]





### 1.11 Exercise 1.3.2

#### Understanding random seeds

While random numbers are important for many analyses, they create a problem: the results you get can vary slightly. This can cause awkward conversations with your boss when your script for calculating the sales forecast gives different answers each time.

Setting the seed for numpy’s random number generator helps avoid such problems by making the random number generation reproducible.

##### Question 1

Which statement about x and y is true?

import numpy as np
np.random.seed(123)
x = np.random.normal(size=5)
y = np.random.normal(size=5)

*The values of x are different from those of y*

##### Question 2

Which statement about x and y is true?

import numpy as np
np.random.seed(123)
x = np.random.normal(size=5)
np.random.seed(123)
y = np.random.normal(size=5)

*x and y have identical values.*

##### Question 3

Which statement about x and y is true?

import numpy as np
np.random.seed(123)
x = np.random.normal(size=5)
np.random.seed(456)
y = np.random.normal(size=5)

*The values of x are different from those of y.*

## 2 CHAPTER 2: Sampling Methods

It’s time to get hands-on and perform the four random sampling methods in Python: simple, systematic, stratified, and cluster.

### 2.1 Chapter 2.1: Simple random and systematic sampling

There are several methods of sampling from a population. In this video, we’ll look at simple random sampling and systematic random sampling.

#### Simple random sampling

Simple random sampling works like a raffle or lottery. We start with our population of raffle tickets or lottery balls and randomly pick them out one at a time.

#### Simple random sampling of coffees

In our coffee ratings dataset, instead of raffle tickets or lottery balls, the population consists of coffee varieties. To perform simple random sampling, we take some at random, one at a time. Each coffee has the same chance as any other of being picked. When using this technique, sometimes we might end up with two coffees that were next to each other in the dataset, and sometimes we might end up with large areas of the dataset that were not selected from at all.

#### Simple random sampling with pandas

We’ve already seen how to do simple random sampling with pandas. We call .sample and set n to the size of the sample. We can also set the seed using the random\_state argument to generate reproducible results, just like we did pseudo-random number generation. Previously, by not setting random\_state when sampling, our code would generate a different random sample each time it was run.

#### Systematic sampling

Another sampling method is known as systematic sampling. This samples the population at regular intervals. Here, looking from top to bottom and left to right within each row, every fifth coffee is sampled.

#### Systematic sampling - defining the interval

Systematic sampling with pandas is slightly trickier than simple random sampling. The tricky part is determining how big the interval between each row should be for a given sample size. Suppose we want a sample size of five coffees. The population size is the number of rows in the whole dataset, and in this case, it’s one thousand three hundred and thirty-eight. The interval is the population size divided by the sample size, but because we want the answer to be an integer, we perform integer division with two forward slashes. This is like standard division but discards any fractional part. One-three-three-eight divided by five is actually two hundred and sixty-seven-point-six, and discarding the fractional part leaves two hundred and sixty-seven. Thus, to get a systematic sample of five coffees, we will select every two hundred sixty-seventh coffee in the dataset.

#### Systematic sampling - selecting the rows

To select every two hundred and sixty-seventh row, we call dot-iloc on coffee\_ratings and pass double-colons and the interval, which is 267 in this case. Double-colon interval tells pandas to select every two hundred and sixty-seventh row from zero to the end of the DataFrame.

#### The trouble with systematic sampling

There is a problem with systematic sampling, though. Suppose we are interested in statistics about the aftertaste attribute of the coffees. To examine this, first, we use reset\_index to create a column of index values in our DataFrame that we can plot. Plotting aftertaste against index shows a pattern. Earlier rows generally have higher aftertaste scores than later rows. This introduces bias into the statistics that we calculate. In general, it is only safe to use systematic sampling if a plot like this has no pattern; that is, it just looks like noise.

#### Making systematic sampling safe

To ensure that systematic sampling is safe, we can randomize the row order before sampling. dot-sample has an argument named frac that lets us specify the proportion of the dataset to return in the sample, rather than the absolute number of rows that n specifies. Setting frac to one randomly samples the whole dataset. In effect, this randomly shuffles the rows. Next, the indices need to be reset so that they go in order from zero again. Specifying drop equals True clears the previous row indexes, and chaining to another reset\_index call creates a column containing these new indexes. Redrawing the plot with the shuffled dataset shows no pattern between aftertaste and index. This is great, but note that once we’ve shuffled the rows, systematic sampling is essentially the same as simple random sampling.

### 2.2 Exercise 2.1.1

#### Simple random sampling

The simplest method of sampling a population is the one you’ve seen already. It is known as *simple random sampling* (sometimes abbreviated to “SRS”), and involves picking rows at random, one at a time, where each row has the same chance of being picked as any other.

In this chapter, you’ll apply sampling methods to a synthetic (fictional) employee attrition dataset from IBM, where “attrition” in this context means leaving the company.

#### Instructions

* Sample 70 rows from attrition using simple random sampling, setting the random seed to 18900217.
* Print the sample dataset, attrition\_samp. What do you notice about the indices?

# Importing pandas
import pandas as pd

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Sample 70 rows using simple random sampling and set the seed
attrition\_samp = attrition.sample(n=70, random\_state=18900217)

# Print the sample
print(attrition\_samp)

 Age Attrition BusinessTravel DailyRate Department \
1134 35 0.0 Travel\_Rarely 583 Research\_Development
1150 52 0.0 Non-Travel 585 Sales
531 33 0.0 Travel\_Rarely 931 Research\_Development
395 31 0.0 Travel\_Rarely 1332 Research\_Development
392 29 0.0 Travel\_Rarely 942 Research\_Development
... ... ... ... ... ...
361 27 0.0 Travel\_Frequently 1410 Sales
1180 36 0.0 Travel\_Rarely 530 Sales
230 26 0.0 Travel\_Rarely 1443 Sales
211 29 0.0 Travel\_Frequently 410 Research\_Development
890 30 0.0 Travel\_Frequently 1312 Research\_Development

 DistanceFromHome Education EducationField \
1134 25 Master Medical
1150 29 Master Life\_Sciences
531 14 Bachelor Medical
395 11 College Medical
392 15 Below\_College Life\_Sciences
... ... ... ...
361 3 Below\_College Medical
1180 2 Master Life\_Sciences
230 23 Bachelor Marketing
211 2 Below\_College Life\_Sciences
890 2 Master Technical\_Degree

 EnvironmentSatisfaction Gender ... PerformanceRating \
1134 High Female ... Excellent
1150 Low Male ... Excellent
531 Very\_High Female ... Excellent
395 High Male ... Excellent
392 Medium Female ... Excellent
... ... ... ... ...
361 Very\_High Female ... Outstanding
1180 High Female ... Excellent
230 High Female ... Excellent
211 Very\_High Female ... Excellent
890 Very\_High Female ... Excellent

 RelationshipSatisfaction StockOptionLevel TotalWorkingYears \
1134 High 1 16
1150 Medium 2 16
531 Very\_High 1 8
395 Very\_High 0 6
392 Low 1 6
... ... ... ...
361 Medium 2 6
1180 High 0 17
230 High 1 5
211 High 3 4
890 Very\_High 0 10

 TrainingTimesLastYear WorkLifeBalance YearsAtCompany \
1134 3 Good 16
1150 3 Good 9
531 5 Better 8
395 2 Good 6
392 2 Good 5
... ... ... ...
361 3 Better 6
1180 2 Good 13
230 2 Good 2
211 3 Better 3
890 2 Better 9

 YearsInCurrentRole YearsSinceLastPromotion YearsWithCurrManager
1134 10 10 1
1150 8 0 0
531 7 1 6
395 5 0 1
392 4 1 3
... ... ... ...
361 5 0 4
1180 7 6 7
230 2 0 0
211 2 0 2
890 7 0 7

[70 rows x 31 columns]

### 2.3 Exercise 2.1.2

#### Systematic sampling

One sampling method that avoids randomness is called systematic sampling. Here, you pick rows from the population at regular intervals.

For example, if the population dataset had one thousand rows, and you wanted a sample size of five, you could pick rows 0, 200, 400, 600, and 800.

#### Instructions

1.Set the sample size to 70. - Calculate the population size from attrition. - Calculate the interval between the rows to be sampled.

1. Systematically sample attrition to get the rows of the population at each interval, starting at 0; assign the rows to attrition\_sys\_samp

# Importing pandas
import pandas as pd

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Set the sample size to 70
sample\_size = 70

# Calculate the population size from attrition\_pop
pop\_size = len(attrition)

# Calculate the interval
interval = pop\_size//sample\_size

# Systematically sample 70 rows
attrition\_sys\_samp = attrition.iloc[::interval]

# Print the sample
print(attrition\_sys\_samp)

 Age Attrition BusinessTravel DailyRate Department \
0 21 0.0 Travel\_Rarely 391 Research\_Development
21 19 0.0 Travel\_Rarely 1181 Research\_Development
42 45 0.0 Travel\_Rarely 252 Research\_Development
63 23 0.0 Travel\_Rarely 373 Research\_Development
84 30 1.0 Travel\_Rarely 945 Sales
... ... ... ... ... ...
1365 48 0.0 Travel\_Rarely 715 Research\_Development
1386 48 0.0 Travel\_Rarely 1355 Research\_Development
1407 50 0.0 Travel\_Rarely 989 Research\_Development
1428 50 0.0 Non-Travel 881 Research\_Development
1449 52 0.0 Travel\_Rarely 699 Research\_Development

 DistanceFromHome Education EducationField EnvironmentSatisfaction \
0 15 College Life\_Sciences High
21 3 Below\_College Medical Medium
42 2 Bachelor Life\_Sciences Medium
63 1 College Life\_Sciences Very\_High
84 9 Bachelor Medical Medium
... ... ... ... ...
1365 1 Bachelor Life\_Sciences Very\_High
1386 4 Master Life\_Sciences High
1407 7 College Medical Medium
1428 2 Master Life\_Sciences Low
1449 1 Master Life\_Sciences High

 Gender ... PerformanceRating RelationshipSatisfaction \
0 Male ... Excellent Very\_High
21 Female ... Excellent Very\_High
42 Female ... Excellent Very\_High
63 Male ... Outstanding Very\_High
84 Male ... Excellent High
... ... ... ... ...
1365 Male ... Excellent High
1386 Male ... Excellent Medium
1407 Female ... Excellent Very\_High
1428 Male ... Excellent Very\_High
1449 Male ... Excellent Low

 StockOptionLevel TotalWorkingYears TrainingTimesLastYear \
0 0 0 6
21 0 1 3
42 0 1 3
63 1 1 2
84 0 1 3
... ... ... ...
1365 0 25 3
1386 0 27 3
1407 1 29 2
1428 1 31 3
1449 1 34 5

 WorkLifeBalance YearsAtCompany YearsInCurrentRole \
0 Better 0 0
21 Better 1 0
42 Better 1 0
63 Better 1 0
84 Good 1 0
... ... ... ...
1365 Best 1 0
1386 Better 15 11
1407 Good 27 3
1428 Better 31 6
1449 Better 33 18

 YearsSinceLastPromotion YearsWithCurrManager
0 0 0
21 0 0
42 0 0
63 0 1
84 0 0
... ... ...
1365 0 0
1386 4 8
1407 13 8
1428 14 7
1449 11 9

[70 rows x 31 columns]

### 2.4 Exercise 2.1.3

#### Is systematic sampling OK?

Systematic sampling has a problem: if the data has been sorted, or there is some sort of pattern or meaning behind the row order, then the resulting sample may not be representative of the whole population. The problem can be solved by shuffling the rows, but then systematic sampling is equivalent to simple random sampling.

Here you’ll look at how to determine whether or not there is a problem.

#### Instructions

1. Add an index column to attrition, assigning the result to attrition\_id.
	* Create a scatter plot of YearsAtCompany versus index for attrition\_id using pandas .plot().
2. Randomly shuffle the rows of attrition.
	* Reset the row indexes, and add an index column to attrition.
	* Repeat the scatter plot of YearsAtCompany versus index, this time using attrition\_shuffled.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Add an index column to attrition\_pop
attrition\_id = attrition.reset\_index()

# Plot YearsAtCompany vs. index for attrition\_pop\_id
attrition\_id.plot(x="index", y="YearsAtCompany", kind="scatter")
plt.show()

# Shuffle the rows of attrition\_pop
attrition\_shuffled = attrition.sample(frac=1)

# Reset the row indexes and create an index column
attrition\_shuffled = attrition\_shuffled.reset\_index(drop=True).reset\_index()

# Plot YearsAtCompany vs. index for attrition\_shuffled
attrition\_shuffled.plot(x="index", y="YearsAtCompany", kind="scatter")
plt.show()





|  |
| --- |
|  Question |
| Does a systematic sample always produce a sample similar to a simple random sample?*No, Systematic sampling has problems when the data are sorted or contain a pattern. Shuffling the rows makes it equivalent to simple random sampling*. |

### 2.5 Chapter 2.2: Stratified and weighted random sampling

Stratified sampling is a technique that allows us to sample a population that contains subgroups.

#### Coffees by country

For example, we could group the coffee ratings by country. If we count the number of coffees by country using the value\_counts method, we can see that these six countries have the most data.

1. 1 The dataset lists Hawaii and Taiwan as countries for convenience, as they are notable coffee-growing regions.

#### Filtering for 6 countries

To make it easier to think about sampling subgroups, let’s limit our analysis to these six countries. We can use the .isin method to filter the population and only return the rows corresponding to these six countries. This filtered dataset is stored as coffee\_ratings\_top.

#### Counts of a simple random sample

Let’s take a ten percent simple random sample of the dataset using .sample with frac set to 0.1. We also set the random\_state argument to ensure reproducibility. As with the whole dataset, we can look at the counts for each country. To make comparisons easier, we set normalize to True to convert the counts into a proportion, which shows what proportion of coffees in the sample came from each country.

#### Comparing proportions

Here are the proportions for the population and the ten percent sample side by side. Just by chance, in this sample, Taiwanese coffees form a disproportionately low percentage. The different makeup of the sample compared to the population could be a problem if we want to analyze the country of origin, for example.

#### Proportional stratified sampling

If we care about the proportions of each country in the sample closely matching those in the population, then we can group the data by country before taking the simple random sample. Note that we used the Python line continuation backslash here, which can be useful for breaking up longer chains of pandas code like this. Calling the .sample method after grouping takes a simple random sample within each country. Now the proportions of each country in the stratified sample are much closer to those in the population.

#### Equal counts stratified sampling

One variation of stratified sampling is to sample equal counts from each group, rather than an equal proportion. The code only has one change from before. This time, we use the n argument in .sample instead of frac to extract fifteen randomly-selected rows from each country. Here, the resulting sample has equal proportions of one-sixth from each country.

#### Weighted random sampling

A close relative of stratified sampling that provides even more flexibility is weighted random sampling. In this variant, we create a column of weights that adjust the relative probability of sampling each row. For example, suppose we thought that it was important to have a higher proportion of Taiwanese coffees in the sample than in the population. We create a condition, in this case, rows where the country of origin is Taiwan. Using the where function from NumPy, we can set a weight of two for rows that match the condition and a weight of one for rows that don’t match the condition. This means when each row is randomly sampled, Taiwanese coffees have two times the chance of being picked compared to other coffees. When we call .sample, we pass the column of weights to the weights argument.

#### Weighted random sampling results

Here, we can see that Taiwan now contains seventeen percent of the sampled dataset, compared to eight-point-five percent in the population. This sort of weighted sampling is common in political polling, where we need to correct for under- or over-representation of demographic groups.

#### 2.5.1 Exercise 2.2.1

#### Proportional stratified sampling

If you are interested in subgroups within the population, then you may need to carefully control the counts of each subgroup within the population. *Proportional stratified sampling* results in subgroup sizes within the sample that are representative of the subgroup sizes within the population. It is equivalent to performing a simple random sample on each subgroup.

#### Instructions

1. Get the proportion of employees by Education level from attrition.
2. Use proportional stratified sampling on attrition\_pop to sample 40% of each Education group, setting the seed to 2022.
3. Get the proportion of employees by Education level from attrition\_strat.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Proportion of employees by Education level
education\_counts\_pop = attrition['Education'].value\_counts(normalize=True)

# Print education\_counts\_pop
print(education\_counts\_pop)

# Proportional stratified sampling for 40% of each Education group
attrition\_strat = attrition.groupby('Education')\
.sample(frac=0.4, random\_state=2022)

# Print the sample
print(attrition\_strat)

# Calculate the Education level proportions from attrition\_strat
education\_counts\_strat = attrition\_strat['Education'].value\_counts(normalize=True)

# Print education\_counts\_strat
print(education\_counts\_strat)

Education
Bachelor 0.389116
Master 0.270748
College 0.191837
Below\_College 0.115646
Doctor 0.032653
Name: proportion, dtype: float64
 Age Attrition BusinessTravel DailyRate Department \
1191 53 0.0 Travel\_Rarely 238 Sales
407 29 0.0 Travel\_Frequently 995 Research\_Development
1233 59 0.0 Travel\_Frequently 1225 Sales
366 37 0.0 Travel\_Rarely 571 Research\_Development
702 31 0.0 Travel\_Frequently 163 Research\_Development
... ... ... ... ... ...
733 38 0.0 Travel\_Frequently 653 Research\_Development
1061 44 0.0 Travel\_Frequently 602 Human\_Resources
1307 41 0.0 Travel\_Rarely 1276 Sales
1060 33 0.0 Travel\_Rarely 516 Research\_Development
177 29 0.0 Travel\_Rarely 738 Research\_Development

 DistanceFromHome Education EducationField \
1191 1 Below\_College Medical
407 2 Below\_College Life\_Sciences
1233 1 Below\_College Life\_Sciences
366 10 Below\_College Life\_Sciences
702 24 Below\_College Technical\_Degree
... ... ... ...
733 29 Doctor Life\_Sciences
1061 1 Doctor Human\_Resources
1307 2 Doctor Life\_Sciences
1060 8 Doctor Life\_Sciences
177 9 Doctor Other

 EnvironmentSatisfaction Gender ... PerformanceRating \
1191 Very\_High Female ... Outstanding
407 Low Male ... Excellent
1233 Low Female ... Excellent
366 Very\_High Female ... Excellent
702 Very\_High Female ... Outstanding
... ... ... ... ...
733 Very\_High Female ... Excellent
1061 Low Male ... Excellent
1307 Medium Female ... Excellent
1060 Very\_High Male ... Excellent
177 Medium Male ... Excellent

 RelationshipSatisfaction StockOptionLevel TotalWorkingYears \
1191 Very\_High 0 18
407 Very\_High 1 6
1233 Very\_High 0 20
366 Medium 2 6
702 Very\_High 0 9
... ... ... ...
733 Very\_High 0 10
1061 High 0 14
1307 Medium 1 22
1060 Low 0 14
177 High 0 4

 TrainingTimesLastYear WorkLifeBalance YearsAtCompany \
1191 2 Best 14
407 0 Best 6
1233 2 Good 4
366 3 Good 5
702 3 Good 5
... ... ... ...
733 2 Better 10
1061 3 Better 10
1307 2 Better 18
1060 6 Better 0
177 2 Better 3

 YearsInCurrentRole YearsSinceLastPromotion YearsWithCurrManager
1191 7 8 10
407 4 1 3
1233 3 1 3
366 3 4 3
702 4 1 4
... ... ... ...
733 3 9 9
1061 7 0 2
1307 16 11 8
1060 0 0 0
177 2 2 2

[588 rows x 31 columns]
Education
Bachelor 0.389456
Master 0.270408
College 0.192177
Below\_College 0.115646
Doctor 0.032313
Name: proportion, dtype: float64

|  |
| --- |
|  Note |
| *By grouping then sampling, the size of each group in the sample is representative of the size of the sample in the population*. |

### 2.6 Exercise 2.2.2

#### Equal counts stratified sampling

If one subgroup is larger than another subgroup in the population, but you don’t want to reflect that difference in your analysis, then you can use *equal counts stratified sampling* to generate samples where each subgroup has the same amount of data. For example, if you are analyzing blood types, O is the most common blood type worldwide, but you may wish to have equal amounts of O, A, B, and AB in your sample.

#### Instructions

1. Use equal counts stratified sampling on attrition to get 30 employees from each Education group, setting the seed to 2022.
2. Get the proportion of employees by Education level from attrition\_eq.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Proportion of employees by Education level
education\_counts\_pop = attrition['Education'].value\_counts(normalize=True)

# Print education\_counts\_pop
print(education\_counts\_pop)

# Get 30 employees from each Education group
attrition\_eq = attrition.groupby('Education')\
.sample(n=30, random\_state=2022)

# Print the sample
print(attrition\_eq)

# Get the proportions from attrition\_eq
education\_counts\_eq = attrition\_eq['Education'].value\_counts(normalize=True)

# Print the results
print(education\_counts\_eq)

Education
Bachelor 0.389116
Master 0.270748
College 0.191837
Below\_College 0.115646
Doctor 0.032653
Name: proportion, dtype: float64
 Age Attrition BusinessTravel DailyRate Department \
1191 53 0.0 Travel\_Rarely 238 Sales
407 29 0.0 Travel\_Frequently 995 Research\_Development
1233 59 0.0 Travel\_Frequently 1225 Sales
366 37 0.0 Travel\_Rarely 571 Research\_Development
702 31 0.0 Travel\_Frequently 163 Research\_Development
... ... ... ... ... ...
774 33 0.0 Travel\_Rarely 922 Research\_Development
869 45 0.0 Travel\_Rarely 1015 Research\_Development
530 32 0.0 Travel\_Rarely 120 Research\_Development
1049 48 0.0 Travel\_Rarely 163 Sales
350 29 1.0 Travel\_Rarely 408 Research\_Development

 DistanceFromHome Education EducationField \
1191 1 Below\_College Medical
407 2 Below\_College Life\_Sciences
1233 1 Below\_College Life\_Sciences
366 10 Below\_College Life\_Sciences
702 24 Below\_College Technical\_Degree
... ... ... ...
774 1 Doctor Medical
869 5 Doctor Medical
530 6 Doctor Life\_Sciences
1049 2 Doctor Marketing
350 25 Doctor Technical\_Degree

 EnvironmentSatisfaction Gender ... PerformanceRating \
1191 Very\_High Female ... Outstanding
407 Low Male ... Excellent
1233 Low Female ... Excellent
366 Very\_High Female ... Excellent
702 Very\_High Female ... Outstanding
... ... ... ... ...
774 Low Female ... Excellent
869 High Female ... Excellent
530 High Male ... Outstanding
1049 Medium Female ... Excellent
350 High Female ... Excellent

 RelationshipSatisfaction StockOptionLevel TotalWorkingYears \
1191 Very\_High 0 18
407 Very\_High 1 6
1233 Very\_High 0 20
366 Medium 2 6
702 Very\_High 0 9
... ... ... ...
774 High 1 10
869 Low 0 10
530 Low 0 8
1049 Low 1 14
350 Medium 0 6

 TrainingTimesLastYear WorkLifeBalance YearsAtCompany \
1191 2 Best 14
407 0 Best 6
1233 2 Good 4
366 3 Good 5
702 3 Good 5
... ... ... ...
774 2 Better 6
869 3 Better 10
530 2 Better 5
1049 2 Better 9
350 2 Best 2

 YearsInCurrentRole YearsSinceLastPromotion YearsWithCurrManager
1191 7 8 10
407 4 1 3
1233 3 1 3
366 3 4 3
702 4 1 4
... ... ... ...
774 1 0 5
869 7 1 4
530 4 1 4
1049 7 6 7
350 2 1 1

[150 rows x 31 columns]
Education
Below\_College 0.2
College 0.2
Bachelor 0.2
Master 0.2
Doctor 0.2
Name: proportion, dtype: float64

|  |
| --- |
|  Note |
| *If you want each subgroup to have equal weight in your analysis, then equal counts stratified sampling is the appropriate technique.* |

### 2.7 Exercise 2.2.3

#### Weighted sampling

Stratified sampling provides rules about the probability of picking rows from your dataset at the subgroup level. A generalization of this is weighted sampling, which lets you specify rules about the probability of picking rows at the row level. The probability of picking any given row is proportional to the weight value for that row.

#### Instructions

1. Plot YearsAtCompany from attrition as a histogram with bins of width 1 from 0 to 40.
2. Sample 400 employees from attrition weighted by YearsAtCompany.
3. Plot YearsAtCompany from attrition\_weight as a histogram with bins of width 1 from 0 to 40.
4. Which is higher? The mean YearsAtCompany from attrition or the mean YearsAtCompany from attrition\_weight? **Answer**: *The weighted sample mean is around 11, which is higher than the population mean of around 7. The fact that the two numbers are different means that the weighted simple random sample is biased.*

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Plot YearsAtCompany from attrition\_pop as a histogram
attrition['YearsAtCompany'].hist(bins=np.arange(0,41,1))

# Sample 400 employees weighted by YearsAtCompany
attrition\_weight = attrition.sample(n=400, weights='YearsAtCompany')

# Print the sample
print(attrition\_weight)

# Plot YearsAtCompany from attrition\_weight as a histogram
attrition\_weight['YearsAtCompany'].hist(bins=np.arange(0, 41, 1))
plt.show()

# The mean YearsAtCompany from attrition dataset
print(attrition['YearsAtCompany'].mean())

# The mean YearsAtCompany from attrition\_weight
print(attrition\_weight['YearsAtCompany'].mean())

 Age Attrition BusinessTravel DailyRate Department \
1424 53 0.0 Travel\_Rarely 1219 Sales
1317 40 0.0 Travel\_Rarely 1137 Research\_Development
1452 53 1.0 Travel\_Rarely 607 Research\_Development
1236 39 0.0 Travel\_Frequently 505 Research\_Development
1445 52 1.0 Travel\_Rarely 266 Sales
... ... ... ... ... ...
985 55 1.0 Travel\_Rarely 436 Sales
930 52 1.0 Travel\_Rarely 723 Research\_Development
1119 49 0.0 Travel\_Rarely 470 Research\_Development
801 42 0.0 Travel\_Rarely 933 Research\_Development
1037 33 0.0 Non-Travel 1283 Sales

 DistanceFromHome Education EducationField \
1424 2 Master Life\_Sciences
1317 1 Master Life\_Sciences
1452 2 Doctor Technical\_Degree
1236 2 Master Technical\_Degree
1445 2 Below\_College Marketing
... ... ... ...
985 2 Below\_College Medical
930 8 Master Medical
1119 20 Master Medical
801 29 Bachelor Life\_Sciences
1037 2 Bachelor Marketing

 EnvironmentSatisfaction Gender ... PerformanceRating \
1424 Low Female ... Excellent
1317 Low Male ... Excellent
1452 High Female ... Excellent
1236 High Female ... Outstanding
1445 Low Female ... Excellent
... ... ... ... ...
985 High Male ... Excellent
930 High Male ... Excellent
1119 High Female ... Excellent
801 Medium Male ... Excellent
1037 Very\_High Female ... Excellent

 RelationshipSatisfaction StockOptionLevel TotalWorkingYears \
1424 High 0 31
1317 Low 1 22
1452 Medium 1 34
1236 Very\_High 0 20
1445 Very\_High 1 33
... ... ... ...
985 High 0 12
930 Low 0 11
1119 High 0 16
801 Very\_High 1 10
1037 Very\_High 0 13

 TrainingTimesLastYear WorkLifeBalance YearsAtCompany \
1424 3 Better 25
1317 3 Better 19
1452 4 Better 33
1236 1 Better 19
1445 3 Better 32
... ... ... ...
985 3 Good 9
930 3 Good 8
1119 2 Good 15
801 3 Good 9
1037 2 Good 11

 YearsInCurrentRole YearsSinceLastPromotion YearsWithCurrManager
1424 8 3 7
1317 7 11 16
1452 7 1 9
1236 6 11 8
1445 14 6 9
... ... ... ...
985 7 7 3
930 2 7 7
1119 11 5 11
801 8 7 8
1037 7 1 7

[400 rows x 31 columns]



7.0081632653061225
11.61

### 2.8 Chapter 2.3: Cluster sampling

One problem with stratified sampling is that we need to collect data from every subgroup. In cases where collecting data is expensive, for example, when we have to physically travel to a location to collect it, it can make our analysis prohibitively expensive. There’s a cheaper alternative called cluster sampling.

#### Stratified sampling vs. cluster sampling

The stratified sampling approach was to split the population into subgroups, then use simple random sampling on each of them. Cluster sampling means that we limit the number of subgroups in the analysis by picking a few of them with simple random sampling. We then perform simple random sampling on each subgroup as before.

#### Varieties of coffee

Let’s return to the coffee dataset and look at the varieties of coffee. In this image, each bean represents the whole subgroup rather than an individual coffee, and there are twenty-eight of them. To extract unique varieties, we use the .unique method. This returns an array, so wrapping it in the list function creates a list of unique varieties. Let’s suppose that it’s expensive to work with all of the different varieties. Enter cluster sampling.

#### Stage 1: sampling for subgroups

The first stage of cluster sampling is to randomly cut down the number of varieties, and we do this by randomly selecting them. Here, we’ve used the random.sample function from the random package to get three varieties, specified using the argument k.

#### Stage 2: sampling each group

The second stage of cluster sampling is to perform simple random sampling on each of the three varieties we randomly selected. We first filter the dataset for rows where the variety is one of the three selected, using the .isin method. To ensure that the isin filtering removes levels with zero rows, we apply the cat.remove\_unused\_categories method on the Series of focus, which is variety here. If we exclude this method, we might receive an error when sampling by variety level. The pandas code is the same as for stratified sampling. Here, we’ve opted for equal counts sampling, with five rows from each remaining variety.

#### Stage 2 output

Here’s the first few columns of the result. Notice that there are the fifteen rows, which we’d expect from sampling five rows from three varieties.

#### Multistage sampling

Note that we had two stages in the cluster sampling. We randomly sampled the subgroups to include, then we randomly sampled rows from those subgroups. Cluster sampling is a special case of multistage sampling. It’s possible to use more than two stages. A common example is national surveys, which can include several levels of administrative regions, like states, counties, cities, and neighborhoods.

### 2.9 Exercise 2.3.1

#### Performing cluster sampling

Now that you know when to use cluster sampling, it’s time to put it into action. In this exercise, you’ll explore the JobRole column of the attrition dataset. You can think of each job role as a subgroup of the whole population of employees.

Use a seed of 19790801 to set the seed with random.seed().

#### Instructions

* Create a list of unique JobRole values from attrition, and assign to job\_roles\_pop.
* Randomly sample four JobRole values from job\_roles\_pop.
1. Subset attrition\_pop for the sampled job roles by filtering for rows where JobRole is in job\_roles\_samp.
* Remove any unused categories from JobRole.
* For each job role in the filtered dataset, take a random sample of ten rows, setting the seed to 2022.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Set the seed
random.seed(19790801)

# Create a list of unique JobRole values
job\_roles\_pop = list(attrition['JobRole'].unique())

# Randomly sample four JobRole values
job\_roles\_samp = random.sample(job\_roles\_pop, k=4)

# Print the result
print(job\_roles\_samp)

# Filter for rows where JobRole is in job\_roles\_samp
jobrole\_condition = attrition['JobRole'].isin(job\_roles\_samp)
attrition\_filtered = attrition[jobrole\_condition]

# Print the result
print(attrition\_filtered)

# Remove categories with no rows
attrition\_filtered['JobRole'] = attrition\_filtered['JobRole'].cat.remove\_unused\_categories()

# Randomly sample 10 employees from each sampled job role
attrition\_clust = attrition\_filtered.groupby('JobRole')\
.sample(n=10, random\_state=2022)

# Print the sample
print(attrition\_clust)

['Research\_Director', 'Research\_Scientist', 'Human\_Resources', 'Manager']
 Age Attrition BusinessTravel DailyRate Department \
0 21 0.0 Travel\_Rarely 391 Research\_Development
5 27 0.0 Non-Travel 443 Research\_Development
6 18 0.0 Non-Travel 287 Research\_Development
10 18 0.0 Non-Travel 1431 Research\_Development
17 31 0.0 Travel\_Rarely 1082 Research\_Development
... ... ... ... ... ...
1462 54 0.0 Travel\_Rarely 584 Research\_Development
1464 55 0.0 Travel\_Rarely 452 Research\_Development
1465 55 0.0 Travel\_Rarely 1117 Sales
1466 58 0.0 Non-Travel 350 Sales
1469 58 1.0 Travel\_Rarely 286 Research\_Development

 DistanceFromHome Education EducationField EnvironmentSatisfaction \
0 15 College Life\_Sciences High
5 3 Bachelor Medical Very\_High
6 5 College Life\_Sciences Medium
10 14 Bachelor Medical Medium
17 1 Master Medical High
... ... ... ... ...
1462 22 Doctor Medical Medium
1464 1 Bachelor Medical Very\_High
1465 18 Doctor Life\_Sciences Low
1466 2 Bachelor Medical Medium
1469 2 Master Life\_Sciences Very\_High

 Gender ... PerformanceRating RelationshipSatisfaction \
0 Male ... Excellent Very\_High
5 Male ... Excellent High
6 Male ... Excellent Very\_High
10 Female ... Excellent High
17 Male ... Excellent Medium
... ... ... ... ...
1462 Female ... Outstanding High
1464 Male ... Excellent High
1465 Female ... Outstanding Very\_High
1466 Male ... Outstanding Very\_High
1469 Male ... Excellent Very\_High

 StockOptionLevel TotalWorkingYears TrainingTimesLastYear \
0 0 0 6
5 3 0 6
6 0 0 2
10 0 0 4
17 0 1 4
... ... ... ...
1462 1 36 6
1464 0 37 2
1465 0 37 2
1466 1 37 0
1469 0 40 2

 WorkLifeBalance YearsAtCompany YearsInCurrentRole \
0 Better 0 0
5 Good 0 0
6 Better 0 0
10 Bad 0 0
17 Better 1 1
... ... ... ...
1462 Better 10 8
1464 Better 36 10
1465 Better 10 9
1466 Good 16 9
1469 Better 31 15

 YearsSinceLastPromotion YearsWithCurrManager
0 0 0
5 0 0
6 0 0
10 0 0
17 1 0
... ... ...
1462 4 7
1464 4 13
1465 7 7
1466 14 14
1469 13 8

[526 rows x 31 columns]
 Age Attrition BusinessTravel DailyRate Department \
1348 44 1.0 Travel\_Rarely 1376 Human\_Resources
886 41 0.0 Non-Travel 552 Human\_Resources
983 39 0.0 Travel\_Rarely 141 Human\_Resources
88 27 1.0 Travel\_Frequently 1337 Human\_Resources
189 34 0.0 Travel\_Rarely 829 Human\_Resources
160 24 0.0 Travel\_Frequently 897 Human\_Resources
839 46 0.0 Travel\_Rarely 991 Human\_Resources
966 30 0.0 Travel\_Rarely 1240 Human\_Resources
162 28 0.0 Non-Travel 280 Human\_Resources
1231 37 0.0 Travel\_Rarely 1239 Human\_Resources
1375 44 0.0 Travel\_Rarely 1315 Research\_Development
1462 54 0.0 Travel\_Rarely 584 Research\_Development
1316 45 0.0 Travel\_Frequently 364 Research\_Development
1356 48 0.0 Travel\_Frequently 117 Research\_Development
1387 48 0.0 Non-Travel 1262 Research\_Development
1321 54 0.0 Non-Travel 142 Human\_Resources
1266 50 0.0 Travel\_Rarely 1452 Research\_Development
1330 46 0.0 Travel\_Rarely 406 Sales
1052 59 0.0 Travel\_Rarely 1089 Sales
1449 52 0.0 Travel\_Rarely 699 Research\_Development
1439 58 0.0 Travel\_Rarely 1055 Research\_Development
1339 58 0.0 Travel\_Frequently 1216 Research\_Development
1426 49 0.0 Travel\_Rarely 1245 Research\_Development
1415 48 0.0 Travel\_Rarely 1224 Research\_Development
1322 51 0.0 Travel\_Rarely 684 Research\_Development
1284 40 0.0 Travel\_Rarely 1308 Research\_Development
1149 37 0.0 Travel\_Rarely 161 Research\_Development
1126 42 0.0 Travel\_Rarely 810 Research\_Development
1374 46 0.0 Travel\_Rarely 1009 Research\_Development
1050 33 0.0 Travel\_Rarely 213 Research\_Development
86 26 0.0 Travel\_Rarely 482 Research\_Development
930 52 1.0 Travel\_Rarely 723 Research\_Development
860 37 0.0 Travel\_Rarely 674 Research\_Development
36 20 1.0 Travel\_Rarely 1362 Research\_Development
997 32 0.0 Travel\_Rarely 824 Research\_Development
1358 45 0.0 Travel\_Rarely 1339 Research\_Development
993 41 0.0 Travel\_Frequently 1200 Research\_Development
421 34 0.0 Travel\_Rarely 181 Research\_Development
789 28 1.0 Travel\_Rarely 654 Research\_Development
94 36 1.0 Travel\_Rarely 318 Research\_Development

 DistanceFromHome Education EducationField \
1348 1 College Medical
886 4 Bachelor Human\_Resources
983 3 Bachelor Human\_Resources
88 22 Bachelor Human\_Resources
189 3 College Human\_Resources
160 10 Bachelor Medical
839 1 College Life\_Sciences
966 9 Bachelor Human\_Resources
162 1 College Life\_Sciences
1231 8 College Other
1375 3 Master Other
1462 22 Doctor Medical
1316 25 Bachelor Medical
1356 22 Bachelor Medical
1387 1 Master Medical
1321 26 Bachelor Human\_Resources
1266 11 Bachelor Life\_Sciences
1330 3 Below\_College Marketing
1052 1 College Technical\_Degree
1449 1 Master Life\_Sciences
1439 1 Bachelor Medical
1339 15 Master Life\_Sciences
1426 18 Master Life\_Sciences
1415 10 Bachelor Life\_Sciences
1322 6 Bachelor Life\_Sciences
1284 14 Bachelor Medical
1149 10 Bachelor Life\_Sciences
1126 23 Doctor Life\_Sciences
1374 2 Bachelor Life\_Sciences
1050 7 Bachelor Medical
86 1 College Life\_Sciences
930 8 Master Medical
860 13 Bachelor Medical
36 10 Below\_College Medical
997 5 College Life\_Sciences
1358 7 Bachelor Life\_Sciences
993 22 Bachelor Life\_Sciences
421 2 Master Medical
789 1 College Life\_Sciences
94 9 Bachelor Medical

 EnvironmentSatisfaction Gender ... PerformanceRating \
1348 Medium Male ... Excellent
886 High Male ... Excellent
983 High Female ... Excellent
88 Low Female ... Excellent
189 High Male ... Excellent
160 Low Male ... Excellent
839 Very\_High Female ... Excellent
966 High Male ... Excellent
162 High Male ... Excellent
1231 High Male ... Excellent
1375 Very\_High Male ... Excellent
1462 Medium Female ... Outstanding
1316 Medium Female ... Outstanding
1356 Very\_High Female ... Excellent
1387 Low Male ... Outstanding
1321 Very\_High Female ... Excellent
1266 High Female ... Excellent
1330 Low Male ... Excellent
1052 Medium Male ... Excellent
1449 High Male ... Excellent
1439 Very\_High Female ... Outstanding
1339 Low Male ... Excellent
1426 Very\_High Male ... Excellent
1415 Very\_High Male ... Excellent
1322 Low Male ... Excellent
1284 High Male ... Excellent
1149 High Female ... Outstanding
1126 Low Female ... Excellent
1374 Low Male ... Excellent
1050 High Male ... Excellent
86 Medium Female ... Excellent
930 High Male ... Excellent
860 Low Male ... Excellent
36 Very\_High Male ... Excellent
997 Very\_High Female ... Excellent
1358 Medium Male ... Excellent
993 Very\_High Female ... Excellent
421 Very\_High Male ... Excellent
789 Low Female ... Excellent
94 Very\_High Male ... Excellent

 RelationshipSatisfaction StockOptionLevel TotalWorkingYears \
1348 Very\_High 1 24
886 Medium 1 10
983 High 1 12
88 Low 0 1
189 High 1 4
160 Very\_High 1 3
839 High 0 10
966 Very\_High 0 12
162 Medium 1 3
1231 High 0 19
1375 Low 1 26
1462 High 1 36
1316 High 0 22
1356 Medium 1 24
1387 High 0 27
1321 High 0 23
1266 Medium 0 21
1330 Very\_High 1 23
1052 High 1 14
1449 Low 1 34
1439 High 1 32
1339 Medium 0 23
1426 High 1 31
1415 Very\_High 0 29
1322 High 0 23
1284 Low 0 21
1149 Low 1 16
1126 Medium 0 16
1374 High 0 26
1050 Very\_High 0 14
86 High 1 1
930 Low 0 11
860 Low 0 10
36 Very\_High 0 1
997 Low 1 12
1358 High 1 25
993 Low 2 12
421 Low 3 6
789 Very\_High 0 10
94 Low 1 2

 TrainingTimesLastYear WorkLifeBalance YearsAtCompany \
1348 1 Better 20
886 4 Better 3
983 3 Bad 8
88 2 Better 1
189 1 Bad 3
160 2 Better 2
839 3 Best 7
966 2 Bad 11
162 2 Better 3
1231 4 Good 10
1375 2 Best 2
1462 6 Better 10
1316 4 Better 0
1356 3 Better 22
1387 3 Good 5
1321 3 Better 5
1266 5 Better 5
1330 3 Better 12
1052 1 Bad 6
1449 5 Better 33
1439 3 Better 9
1339 3 Better 2
1426 5 Better 31
1415 3 Better 22
1322 5 Better 20
1284 2 Best 20
1149 2 Better 16
1126 2 Better 1
1374 2 Bad 3
1050 3 Best 13
86 3 Good 1
930 3 Good 8
860 2 Better 10
36 5 Better 1
997 2 Better 7
1358 2 Better 1
993 4 Good 6
421 3 Better 5
789 4 Better 7
94 0 Good 1

 YearsInCurrentRole YearsSinceLastPromotion YearsWithCurrManager
1348 6 3 6
886 2 1 2
983 3 3 6
88 0 0 0
189 2 0 2
160 2 2 1
839 6 5 7
966 9 4 7
162 2 2 2
1231 0 4 7
1375 2 0 1
1462 8 4 7
1316 0 0 0
1356 17 4 7
1387 4 2 1
1321 3 4 4
1266 4 4 4
1330 9 4 9
1052 4 0 4
1449 18 11 9
1439 8 1 5
1339 2 2 2
1426 9 0 9
1415 10 12 9
1322 18 15 15
1284 7 4 9
1149 11 6 8
1126 0 0 0
1374 2 0 1
1050 9 3 7
86 0 1 0
930 2 7 7
860 8 3 7
36 0 1 1
997 1 2 5
1358 0 0 0
993 2 3 3
421 0 1 2
789 7 3 7
94 0 0 0

[40 rows x 31 columns]

### 2.10 Chapter 2.4: Comparing sampling methods

Let’s review the various sampling techniques we learned about.

#### Review of sampling techniques - setup

For convenience, we’ll stick to the six countries with the most coffee varieties that we used before. This corresponds to eight hundred and eighty rows and eight columns, retrieved using the .shape attribute.

#### Review of simple random sampling

Simple random sampling uses .sample with either n or frac set to determine how many rows to pseudo-randomly choose, given a seed value set with random\_state. The simple random sample returns two hundred and ninety-three rows because we specified frac as one-third, and one-third of eight hundred and eighty is just over two hundred and ninety-three.

#### Review of stratified sampling

Stratified sampling groups by the country subgroup before performing simple random sampling on each subgroup. Given each of these top countries have quite a few rows, stratifying produces the same number of rows as the simple random sample.

#### Review of cluster sampling

In the cluster sample, we’ve used two out of six countries to roughly mimic frac equals one-third from the other sample types. Setting n equal to one-sixth of the total number of rows gives roughly equal sample sizes in each of the two subgroups. Using .shape again, we see that this cluster sample has close to the same number of rows: two-hundred-ninety-two compared to two-hundred-ninety-three for the other sample types.

#### Calculating mean cup points

Let’s calculate a population parameter, the mean of the total cup points. When the population parameter is the mean of a field, it’s often called the population mean. Remember that in real-life scenarios, we typically wouldn’t know what the population mean is. Since we have it here, though, we can use this value of eighty-one-point-nine as a gold standard to measure against. Now we’ll calculate the same value using each of the sampling techniques we’ve discussed. These are point estimates of the mean, often called sample means. The simple and stratified sample means are really close to the population mean. Cluster sampling isn’t quite as close, but that’s typical. Cluster sampling is designed to give us an answer that’s almost as good while using less data.

#### Mean cup points by country: simple random

Here’s a slightly more complicated calculation of the mean total cup points for each country. We group by country before calculating the mean to return six numbers. So how do the numbers from the simple random sample compare? The sample means are pretty close to the population means.

#### Mean cup points by country: stratified

The same is true of the sample means from the stratified technique. Each sample mean is relatively close to the population mean.

#### Mean cup points by country: cluster

With cluster sampling, while the sample means are pretty close to the population means, the obvious limitation is that we only get values for the two countries that were included in the sample. If the mean cup points for each country is an important metric in our analysis, cluster sampling would be a bad idea.

### 2.11 Exercise 2.4.1

#### 3 kinds of sampling

You’re going to compare the performance of point estimates using simple, stratified, and cluster sampling. Before doing that, you’ll have to set up the samples.

You’ll use the RelationshipSatisfaction column of the attrition dataset, which categorizes the employee’s relationship with the company. It has four levels: Low, Medium, High, and Very\_High.

#### Instructions

1. Perform simple random sampling on attrition to get one-quarter of the population, setting the seed to 2022.
2. Perform stratified sampling on attrition to sample one-quarter of each RelationshipSatisfaction group, setting the seed to 2022.
3. Create a list of unique values from attrition’s RelationshipSatisfaction column. Randomly sample satisfaction\_unique to get two values. Subset the population for rows where RelationshipSatisfaction is in satisfaction\_samp and clear any unused categories from RelationshipSatisfaction; assign to attrition\_clust\_prep. Perform cluster sampling on the selected satisfaction groups, sampling one quarter of the *population* and setting the seed to 2022.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Perform simple random sampling to get 0.25 of the population
attrition\_srs = attrition.sample(frac=1/4, random\_state=2022)

# Perform stratified sampling to get 0.25 of each relationship group
attrition\_strat = attrition.groupby('RelationshipSatisfaction')\
.sample(frac=1/4, random\_state=2022)

# Create a list of unique RelationshipSatisfaction values
satisfaction\_unique = list(attrition['RelationshipSatisfaction'].unique())

# Randomly sample 2 unique satisfaction values
satisfaction\_samp = random.sample(satisfaction\_unique, k=2)

# Filter for satisfaction\_samp and clear unused categories from RelationshipSatisfaction
satis\_condition = attrition['RelationshipSatisfaction'].isin(satisfaction\_samp)
attrition\_clust\_prep = attrition[satis\_condition]
attrition\_clust\_prep['RelationshipSatisfaction'] = attrition\_clust\_prep['RelationshipSatisfaction'].cat.remove\_unused\_categories()

# Perform cluster sampling on the selected group, getting 0.25 of attrition\_clust\_prep
attrition\_clust = attrition\_clust\_prep.groupby("RelationshipSatisfaction")\
.sample(n=len(attrition) // 6, random\_state=2022)

print(attrition\_clust)

 Age Attrition BusinessTravel DailyRate Department \
1381 45 1.0 Travel\_Rarely 1449 Sales
1357 42 0.0 Travel\_Rarely 300 Research\_Development
924 30 0.0 Travel\_Rarely 288 Research\_Development
1224 46 0.0 Travel\_Rarely 1003 Research\_Development
1277 48 0.0 Travel\_Rarely 1236 Research\_Development
... ... ... ... ... ...
357 27 0.0 Travel\_Rarely 798 Research\_Development
424 44 1.0 Travel\_Frequently 429 Research\_Development
1182 36 0.0 Travel\_Frequently 884 Research\_Development
1055 34 0.0 Travel\_Frequently 669 Research\_Development
962 34 0.0 Travel\_Rarely 1031 Research\_Development

 DistanceFromHome Education EducationField EnvironmentSatisfaction \
1381 2 Bachelor Marketing Low
1357 2 Bachelor Life\_Sciences Low
924 2 Bachelor Life\_Sciences High
1224 8 Master Life\_Sciences Very\_High
1277 1 Master Life\_Sciences Very\_High
... ... ... ... ...
357 6 Master Medical Low
424 1 College Medical High
1182 23 College Medical High
1055 1 Bachelor Medical Very\_High
962 6 Master Life\_Sciences High

 Gender ... PerformanceRating RelationshipSatisfaction \
1381 Female ... Excellent Low
1357 Male ... Excellent Low
924 Male ... Excellent Low
1224 Female ... Outstanding Low
1277 Female ... Excellent Low
... ... ... ... ...
357 Female ... Excellent High
424 Male ... Excellent High
1182 Male ... Excellent High
1055 Male ... Outstanding High
962 Female ... Excellent High

 StockOptionLevel TotalWorkingYears TrainingTimesLastYear \
1381 0 26 2
1357 0 24 2
924 3 11 3
1224 3 19 2
1277 1 21 3
... ... ... ...
357 2 6 5
424 3 6 2
1182 1 17 3
1055 0 14 3
962 1 12 3

 WorkLifeBalance YearsAtCompany YearsInCurrentRole \
1381 Better 24 10
1357 Good 22 6
924 Better 11 10
1224 Better 16 13
1277 Bad 3 2
... ... ... ...
357 Good 5 3
424 Good 5 3
1182 Better 5 2
1055 Better 13 9
962 Better 1 0

 YearsSinceLastPromotion YearsWithCurrManager
1381 1 11
1357 4 14
924 10 8
1224 1 7
1277 0 2
... ... ...
357 0 3
424 2 3
1182 0 3
1055 4 9
962 0 0

[490 rows x 31 columns]

### 2.12 Exercise 2.4.4

#### Comparing point estimates

Now that you have three types of sample (simple, stratified, and cluster), you can compare point estimates from each sample to the population parameter. That is, you can calculate the same summary statistic on each sample and see how it compares to the summary statistic for the population.

Here, we’ll look at how satisfaction with the company affects whether or not the employee leaves the company. That is, you’ll calculate the proportion of employees who left the company (they have an Attrition value of 1) for each value of RelationshipSatisfaction.

#### Instructions

1. Group attrition by RelationshipSatisfaction levels and calculate the mean of Attrition for each level.
2. Calculate the proportion of employee attrition for each relationship satisfaction group, this time on the simple random sample, attrition\_srs.
3. Calculate the proportion of employee attrition for each relationship satisfaction group, this time on the stratified sample, attrition\_strat.
4. Calculate the proportion of employee attrition for each relationship satisfaction group, this time on the cluster sample, attrition\_clust.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Perform simple random sampling to get 0.25 of the population
attrition\_srs = attrition.sample(frac=1/4, random\_state=2022)

# Perform stratified sampling to get 0.25 of each relationship group
attrition\_strat = attrition.groupby('RelationshipSatisfaction')\
.sample(frac=1/4, random\_state=2022)

# Mean Attrition by RelationshipSatisfaction group
mean\_attrition\_pop = attrition.groupby('RelationshipSatisfaction')\
['Attrition'].mean()

# Print the result
print(mean\_attrition\_pop)

# Calculate the same thing for the simple random sample
mean\_attrition\_srs = attrition\_srs.groupby('RelationshipSatisfaction')\
['Attrition'].mean()

# Print the result
print(mean\_attrition\_srs)

# Calculate the same thing for the stratified sample
mean\_attrition\_strat = attrition\_strat.groupby('RelationshipSatisfaction')\
['Attrition'].mean()

# Print the result
print(mean\_attrition\_strat)

# Calculate the same thing for the cluster sample
mean\_attrition\_clust = attrition\_clust.groupby('RelationshipSatisfaction')\
['Attrition'].mean()

# Print the result
print(mean\_attrition\_clust)

RelationshipSatisfaction
Low 0.206522
Medium 0.148515
High 0.154684
Very\_High 0.148148
Name: Attrition, dtype: float64
RelationshipSatisfaction
Low 0.134328
Medium 0.164179
High 0.160000
Very\_High 0.155963
Name: Attrition, dtype: float64
RelationshipSatisfaction
Low 0.144928
Medium 0.078947
High 0.165217
Very\_High 0.129630
Name: Attrition, dtype: float64
RelationshipSatisfaction
Low 0.191837
High 0.134694
Name: Attrition, dtype: float64

## 3 CHAPTER 3: Sampling Distributions

Let’s test your sampling. In this chapter, you’ll discover how to quantify the accuracy of sample statistics using relative errors, and measure variation in your estimates by generating sampling distributions.

### 3.1 Chapter 3.1: Relative error of point estimates

Let’s see how the size of the sample affects the accuracy of the point estimates we calculate.

#### Sample size is number of rows

The sample size, calculated here with the len function, is the number of observations, that is, the number of rows in the sample. That’s true whichever method we use to create the sample. We’ll stick to looking at simple random sampling since it works well in most cases and it’s easier to reason about.

#### Various sample sizes

Let’s calculate a population parameter, the mean cup points of the coffees. It’s around eighty-two-point-one-five. This is our gold standard to compare against. If we take a sample size of ten, the point estimate of this parameter is wrong by about point-eight-eight. Increasing the sample size to one hundred gets us closer; the estimate is only wrong by about point-three-four. Increasing the sample size further to one thousand brings the estimate to about point-zero-three away from the population parameter. In general, larger sample sizes will give us more accurate results.

#### Relative errors

For any of these sample sizes, we want to compare the population mean to the sample mean. This is the same code we just saw, but with the numerical sample size replaced with a variable named sample\_size. The most common metric for assessing the difference between the population and a sample mean is the relative error. The relative error is the absolute difference between the two numbers; that is, we ignore any minus signs, divided by the population mean. Here, we also multiply by one hundred to make it a percentage.

#### Relative error vs. sample size

Here’s a line plot of relative error versus sample size. We see that the relative error decreases as the sample size increases, and beyond that, the plot has other important properties. Firstly, the blue line is really noisy, particularly for small sample sizes. If our sample size is small, the sample mean we calculate can be wildly different by adding one or two more random rows to the sample. Secondly, the amplitude of the line is quite steep, to begin with. When we have a small sample size, adding just a few more samples can give us much better accuracy. Further to the right of the plot, the line is less steep. If we already have a large sample size, adding a few more rows to the sample doesn’t bring as much benefit. Finally, at the far right of the plot, where the sample size is the whole population, the relative error decreases to zero.

### 3.2 Exercise 3.1.1

#### Calculating relative errors

The size of the sample you take affects how accurately the point estimates reflect the corresponding population parameter. For example, when you calculate a sample mean, you want it to be close to the population mean. However, if your sample is too small, this might not be the case.

The most common metric for assessing accuracy is *relative error*. This is the absolute difference between the population parameter and the point estimate, all divided by the population parameter. It is sometimes expressed as a percentage.

#### Instructions

1. Generate a simple random sample from attrition\_pop of fifty rows, setting the seed to 2022.
* Calculate the mean employee Attrition in the sample.
* Calculate the relative error between mean\_attrition\_srs50 and mean\_attrition\_pop as a *percentage*.
1. Calculate the *relative error percentage* again. This time, use a simple random sample of one hundred rows of attrition.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Population Attrtion mean
mean\_attrition\_pop = attrition['Attrition'].mean()

# Print the result
print(mean\_attrition\_pop)

# Generate a simple random sample of 50 rows, with seed 2022
attrition\_srs50 = attrition.sample(n=50, random\_state = 2022)

# Calculate the mean employee attrition in the sample
mean\_attrition\_srs50 = attrition\_srs50['Attrition'].mean()

# Calculate the relative error percentage
rel\_error\_pct50 = 100 \* abs(mean\_attrition\_pop - mean\_attrition\_srs50)/mean\_attrition\_pop

# Print rel\_error\_pct50
print(rel\_error\_pct50)

# Generate a simple random sample of 100 rows, with seed 2022
attrition\_srs100 = attrition.sample(n=100, random\_state = 2022)

# Calculate the mean employee attrition in the sample
mean\_attrition\_srs100 = attrition\_srs100['Attrition'].mean()

# Calculate the relative error percentage
rel\_error\_pct100 = 100 \* abs(mean\_attrition\_pop - mean\_attrition\_srs100)/mean\_attrition\_pop

# Print rel\_error\_pct100
print(rel\_error\_pct100)

0.16122448979591836
62.78481012658227
6.962025316455695

### 3.3 Chapter 3.2: Creating a sampling distribution

We just saw how point estimates like the sample mean will vary depending on which rows end up in the sample.

#### Same code, different answer

For example, this same code to calculate the mean cup points from a simple random sample of thirty coffees gives a slightly different answer each time. Let’s try to visualize and quantify this variation.

#### Same code, 1000 times

A for loop lets us run the same code many times. It’s especially useful for situations like this where the result contains some randomness. We start by creating an empty list to store the means. Then, we set up the for loop to repeatedly sample 30 coffees from coffee\_ratings a total of 1000 times, calculating the mean cup points each time. After each calculation, we append the result, also called a replicate, to the list. Each time the code is run, we get one sample mean, so running the code a thousand times generates a list of one thousand sample means.

#### Distribution of sample means for size 30

The one thousand sample means form a distribution of sample means. To visualize a distribution, the best plot is often a histogram. Here we can see that most of the results lie between eighty-one and eighty-three, and they roughly follow a bell-shaped curve, like a normal distribution. There’s an important piece of jargon we need to know here. A distribution of replicates of sample means, or other point estimates, is known as a sampling distribution.

#### Different sample sizes

Here are histograms from running the same code again with different sample sizes. When we decrease the original sample size of thirty to six, we can see from the x-values that the range of the results is broader. The bulk of the results now lie between eighty and eighty-four. On the other hand, increasing the sample size to one hundred and fifty results in a much narrower range. Now most of the results are between eighty-one-point-eight and eighty-two-point-six. As we saw previously, bigger sample sizes give us more accurate results. By replicating the sampling many times, as we’ve done here, we can quantify that accuracy.

### 3.4 Exercise 3.2.1

#### Replicating samples

When you calculate a point estimate such as a sample mean, the value you calculate depends on the rows that were included in the sample. That means that there is some randomness in the answer. In order to quantify the variation caused by this randomness, you can create many samples and calculate the sample mean (or another statistic) for each sample.

#### Instructions

1. Replicate the provided code so that it runs 500 times. Assign the resulting list of sample means to mean\_attritions.
2. Draw a histogram of the mean\_attritions list with 16 bins.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Create an empty list
mean\_attritions = []
# Loop 500 times to create 500 sample means
for i in range(500):
 mean\_attritions.append(
 attrition.sample(n=60)['Attrition'].mean()
 )

# Print out the first few entries of the list
print(mean\_attritions[0:5])

# Create a histogram of the 500 sample means
plt.hist(mean\_attritions, bins=16)
plt.show()

[0.18333333333333332, 0.18333333333333332, 0.16666666666666666, 0.11666666666666667, 0.16666666666666666]



### 3.5 Chapter 3.3: Approximate sampling distributions

In the last exercise, we saw that while increasing the number of replicates didn’t affect the relative error of the sample means; it did result in a more consistent shape to the distribution.

#### 4 dice

Let’s consider the case of four six-sided dice rolls. We can generate all possible combinations of rolls using the expand\_grid function, which is defined in the pandas documentation, and uses the itertools package. There are six to the power four, or one-thousand-two-hundred-ninety-six possible dice roll combinations.

#### Mean roll

Let’s consider the mean of the four rolls by adding a column to our DataFrame called mean\_roll. mean\_roll ranges from 1, when four ones are rolled, to 6, when four sixes are rolled.

#### Exact sampling distribution

Since the mean roll takes discrete values instead of continuous values, the best way to see the distribution of mean\_roll is to draw a bar plot. First, we convert mean\_roll to a categorical by setting its type to category. We are interested in the counts of each value, so we use dot-value\_counts, passing the sort equals False argument. This ensures the x-axis ranges from one to six instead of sorting the bars by frequency. Chaining .plot to value\_counts, and setting kind to "bar", produces a bar plot of the mean roll distribution. This is the exact sampling distribution of the mean roll because it contains every single combination of die rolls.

#### The number of outcomes increases fast

If we increase the number of dice in our scenario, the number of possible outcomes increases by a factor of six each time. These values can be shown by creating a DataFrame with two columns: n\_dice, ranging from 1 to 100, and n\_outcomes, which is the number of possible outcomes, calculated using six to the power of the number of dice. With just one hundred dice, the number of outcomes is about the same as the number of atoms in the universe: six-point-five times ten to the seventy-seventh power. Long before you start dealing with big datasets, it becomes computationally impossible to calculate the exact sampling distribution. That means we need to rely on approximations.

#### Simulating the mean of four dice rolls

We can generate a sample mean of four dice rolls using NumPy’s random.choice method, specifying size as four. This will randomly choose values from a specified list, in this case, four values from the numbers one to six, which is created using a range from one to seven wrapped in the list function. Notice that we set replace equals True because we can roll the same number several times.

#### Simulating the mean of four dice rolls

Then we use a for loop to generate lots of sample means, in this case, one thousand. We again use the .append method to populate the sample means list with our simulated sample means. The output contains a sampling of many of the same values we saw with the exact sampling distribution.

#### Approximate sampling distribution

Here’s a histogram of the approximate sampling distribution of mean rolls. This time, it uses the simulated rather than the exact values. It’s known as an approximate sampling distribution. Notice that although it isn’t perfect, it’s pretty close to the exact sampling distribution. Usually, we don’t have access to the whole population, so we can’t calculate the exact sampling distribution. However, we can feel relatively confident that using an approximation will provide a good guess as to how the sampling distribution will behave.

### 3.6 Exercise 3.3.1

#### Exact sampling distribution

To quantify how the point estimate (sample statistic) you are interested in varies, you need to know all the possible values it can take and how often. That is, you need to know its distribution.

The distribution of a sample statistic is called the *sampling distribution*. When we can calculate this exactly, rather than using an approximation, it is known as the *exact sampling distribution*.

Let’s take another look at the sampling distribution of dice rolls. This time, we’ll look at five eight-sided dice. (These have the numbers one to eight.)

#### Instructions

1. Expand a grid representing 5 8-sided dice. That is, create a DataFrame with five columns from a dictionary, named die1 to die5. The rows should contain all possibilities for throwing five dice, each numbered 1 to 8.
2. Add a column, mean\_roll, to dice, that contains the mean of the five rolls as a categorical.
3. Create a bar plot of the mean\_roll categorical column, so it displays the count of each mean\_roll in increasing order from 1.0 to 8.0.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Function to create a grid of all possible combinations
def expand\_grid(dictionary):
 from itertools import product
 return pd.DataFrame([row for row in product(\*dictionary.values())], columns=dictionary.keys())

# Expand a grid representing 5 8-sided dice
dice = expand\_grid(
 {'die1': range(1, 9),
 'die2': range(1, 9),
 'die3': range(1, 9),
 'die4': range(1, 9),
 'die5': range(1, 9)}
)

# Print the result
print(dice)

# Add a column of mean rolls and convert to a categorical
dice['mean\_roll'] = (dice['die1']+ dice['die2']+ dice['die3']+ dice['die4']+ dice['die5'])/5

dice['mean\_roll'] = dice['mean\_roll'].astype('category')

# Print result
print(dice)

# Draw a bar plot of mean\_roll
dice['mean\_roll'].value\_counts(sort=False).plot(kind='bar')
plt.show()

 die1 die2 die3 die4 die5
0 1 1 1 1 1
1 1 1 1 1 2
2 1 1 1 1 3
3 1 1 1 1 4
4 1 1 1 1 5
... ... ... ... ... ...
32763 8 8 8 8 4
32764 8 8 8 8 5
32765 8 8 8 8 6
32766 8 8 8 8 7
32767 8 8 8 8 8

[32768 rows x 5 columns]
 die1 die2 die3 die4 die5 mean\_roll
0 1 1 1 1 1 1.0
1 1 1 1 1 2 1.2
2 1 1 1 1 3 1.4
3 1 1 1 1 4 1.6
4 1 1 1 1 5 1.8
... ... ... ... ... ... ...
32763 8 8 8 8 4 7.2
32764 8 8 8 8 5 7.4
32765 8 8 8 8 6 7.6
32766 8 8 8 8 7 7.8
32767 8 8 8 8 8 8.0

[32768 rows x 6 columns]



### 3.7 Exercise 3.3.2

#### Generating an approximate sampling distribution

Calculating the exact sampling distribution is only possible in very simple situations. With just five eight-sided dice, the number of possible rolls is 8\*\*5, which is over thirty thousand. When the dataset is more complicated, for example, where a variable has hundreds or thousands of categories, the number of possible outcomes becomes too difficult to compute exactly.

In this situation, you can calculate an *approximate sampling distribution* by simulating the exact sampling distribution. That is, you can repeat a procedure over and over again to simulate both the sampling process and the sample statistic calculation process.

#### Instructions

1. Sample one to eight, five times, with replacement. Assign to five\_rolls.
* Calculate the mean of five\_rolls.
1. Replicate the sampling code 1000 times, assigning each result to the list sample\_means\_1000.
2. Plot sample\_means\_1000 as a histogram with 20 bins.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Sample one to eight, five times, with replacement
five\_rolls = np.random.choice(list(range(1, 9)), size=5, replace=True)

# Print the mean of five\_rolls
print(five\_rolls.mean())

# Replicate the sampling code 1000 times
sample\_means\_1000 = []
for i in range(1000):
 sample\_means\_1000.append(
 np.random.choice(list(range(1, 9)), size=5, replace=True).mean()
 )

# Print the first 10 entries of the result
print(sample\_means\_1000[0:10])

# Draw a histogram of sample\_means\_1000 with 20 bins
plt.hist(sample\_means\_1000, bins=20)
plt.show()

5.2
[3.4, 3.6, 4.2, 3.4, 4.0, 4.2, 4.4, 7.8, 2.8, 5.8]



### 3.8 Chapter 3.4: Standard errors and the Central Limit Theorem

The Gaussian distribution (also known as the normal distribution) plays an important role in statistics. Its distinctive bell-shaped curve has been cropping up throughout this course.

#### Sampling distribution of mean cup points

Here are approximate sampling distributions of the mean cup points from the coffee dataset. Each histogram shows five thousand replicates, with different sample sizes in each case. Look at the x-axis labels. We already saw how increasing the sample size results in greater accuracy in our estimates of the population parameter, so the width of the distribution shrinks as the sample size increases. When the sample size is five, the x-axis ranges from seventy-six to eighty-six, whereas, for a sample size of three hundred and twenty, the range is from eighty-one-point-six to eighty-two-point-six. Now, look at the shape of each distribution. As the sample size increases, we can see that the shape of the curve gets closer and closer to being a normal distribution. At sample size five, the curve is only a very loose approximation since it isn’t very symmetric. By sample size eighty, it is a very reasonable approximation.

#### Consequences of the central limit theorem

What we just saw is, in essence, what the central limit theorem tells us. The means of independent samples have normal distributions. Then, as the sample size increases, we see two things. The distribution of these averages gets closer to being normal, and the width of this sampling distribution gets narrower.

#### Population & sampling distribution means

Recall the population parameter of the mean cup points. We’ve seen this calculation before, and its value is eighty-two-point-one-five. We can also calculate summary statistics on our sampling distributions to see how they compare. For each of our four sampling distributions, if we take the mean of our sample means, we can see that we get values that are pretty close to the population parameter that the sampling distributions are trying to estimate.

#### Population & sampling distribution standard deviations

Now let’s consider the standard deviation of the population cup points. It’s about two-point-seven. By comparison, if we take the standard deviation of the sample means from each of the sampling distributions using NumPy, we get much smaller numbers, and they decrease as the sample size increases. Note that when we are calculating a population standard deviation with pandas .std, we must specify ddof equals zero, as .std calculates a sample standard deviation by default. When we are calculating a standard deviation on a sample of the population using NumPy’s std function, like in these calculations on the sampling distribution, we must specify a ddof of one. So what are these smaller standard deviation values?

#### Population mean over square root sample size

One other consequence of the central limit theorem is that if we divide the population standard deviation, in this case around 2.7, by the square root of the sample size, we get an estimate of the standard deviation of the sampling distribution for that sample size. It isn’t exact because of the randomness involved in the sampling process, but it’s pretty close.

#### Standard error

We just saw the impact of the sample size on the standard deviation of the sampling distribution. This standard deviation of the sampling distribution has a special name: the standard error. It is useful in a variety of contexts, from estimating population standard deviation to setting expectations on what level of variability we would expect from the sampling process.

### 3.9 Exercise 3.4.1

#### Population & sampling distribution means

One of the useful features of sampling distributions is that you can quantify them. Specifically, you can calculate summary statistics on them. Here, you’ll look at the relationship between the mean of the sampling distribution and the population parameter’s mean.

Three sampling distributions are provided. For each, the employee attrition dataset was sampled using simple random sampling, then the mean attrition was calculated. This was done 1000 times to get a sampling distribution of mean attritions. One sampling distribution used a sample size of 5 for each replicate, one used 50, and one used 500.

#### Instructions

1. Calculate the mean of sampling\_distribution\_5, sampling\_distribution\_50, and sampling\_distribution\_500 (a mean of sample means).

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Set a seed for reproducibility
random\_seed = 2021

# Create three empty lists to hold the sampling distributions
sampling\_distribution\_5 = [] # Sample size of 5
sampling\_distribution\_50 = [] # Sample size of 50
sampling\_distribution\_500 = [] # Sample size of 500

# Perform biased sampling and calculate mean attrition 1000 times for each sample size
for i in range(1000):
 # Sample size = 5 (heavier weights toward high attrition)
 sampling\_distribution\_5.append(
 attrition.sample(n=5, random\_state=random\_seed + i)['Attrition'].mean()
 )

 # Sample size = 50 (bias reduces as sample size increases)
 sampling\_distribution\_50.append(
 attrition.sample(n=50, random\_state=random\_seed + i)['Attrition'].mean()
 )

 # Sample size = 500 (approaching unbiased mean)
 sampling\_distribution\_500.append(
 attrition.sample(n=500, random\_state=random\_seed + i)['Attrition'].mean()
 )

# Optional: Convert the sampling distributions to DataFrame for analysis
sampling\_df = pd.DataFrame({
 'Sample\_Size\_5': sampling\_distribution\_5,
 'Sample\_Size\_50': sampling\_distribution\_50,
 'Sample\_Size\_500': sampling\_distribution\_500
})

# Calculate the mean of the mean attritions for each sampling distribution
mean\_of\_means\_5 = np.mean(sampling\_distribution\_5)
mean\_of\_means\_50 = np.mean(sampling\_distribution\_50)
mean\_of\_means\_500 = np.mean(sampling\_distribution\_500)

# Print the results
print(mean\_of\_means\_5)
print(mean\_of\_means\_50)
print(mean\_of\_means\_500)

0.155
0.15998
0.160622

|  |
| --- |
|  Note |
| *Even for small sample sizes, the mean of the sampling distribution is a good approximation of the population mean.* |

### 3.10 Exercise 3.4.2

#### Population & sampling distribution variation

You just calculated the mean of the sampling distribution and saw how it is an estimate of the corresponding population parameter. Similarly, as a result of the central limit theorem, the standard deviation of the sampling distribution has an interesting relationship with the population parameter’s standard deviation and the sample size.

#### Instructions

1. Calculate the standard deviation of sampling\_distribution\_5, sampling\_distribution\_50, and sampling\_distribution\_500 (a standard deviation of sample means).

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course arrays
attrition = pd.read\_feather("datasets/attrition.feather")

# Set a seed for reproducibility
random\_seed = 2021

# Create three empty lists to hold the sampling distributions
sampling\_distribution\_5 = [] # Sample size of 5
sampling\_distribution\_50 = [] # Sample size of 50
sampling\_distribution\_500 = [] # Sample size of 500

# Perform biased sampling and calculate mean attrition 1000 times for each sample size
for i in range(1000):
 # Sample size = 5 (heavier weights toward high attrition)
 sampling\_distribution\_5.append(
 attrition.sample(n=5, random\_state=random\_seed + i)['Attrition'].mean()
 )

 # Sample size = 50 (bias reduces as sample size increases)
 sampling\_distribution\_50.append(
 attrition.sample(n=50, random\_state=random\_seed + i)['Attrition'].mean()
 )

 # Sample size = 500 (approaching unbiased mean)
 sampling\_distribution\_500.append(
 attrition.sample(n=500, random\_state=random\_seed + i)['Attrition'].mean()
 )

# Optional: Convert the sampling distributions to DataFrame for analysis
sampling\_df = pd.DataFrame({
 'Sample\_Size\_5': sampling\_distribution\_5,
 'Sample\_Size\_50': sampling\_distribution\_50,
 'Sample\_Size\_500': sampling\_distribution\_500
})

# Calculate the std. dev. of the mean attritions for each sampling distribution
sd\_of\_means\_5 = np.std(sampling\_distribution\_5, ddof = 1)
sd\_of\_means\_50 = np.std(sampling\_distribution\_50, ddof = 1)
sd\_of\_means\_500 = np.std(sampling\_distribution\_500, ddof = 1)

# Print the results
print(sd\_of\_means\_5)
print(sd\_of\_means\_50)
print(sd\_of\_means\_500)

0.15244093360458746
0.04970785119546479
0.014243454356018837

|  |
| --- |
|  Note |
| *The amount of variation in the sampling distribution is related to the amount of variation in the population and the sample size. This is another consequence of the Central Limit Theorem.* |

## 4 CHAPTER 4: Bootstrap Distributions

You’ll get to grips with resampling to perform bootstrapping and estimate variation in an unknown population. You’ll learn the difference between sampling distributions and bootstrap distributions using resampling.

### 4.1 Chapter 4.1: Introduction to bootstrapping

So far, we’ve mostly focused on the idea of sampling without replacement.

#### With or without

Sampling without replacement is like dealing a pack of cards. When we deal the ace of spades to one player, we can’t then deal the ace of spades to another player. Sampling with replacement is like rolling dice. If we roll a six, we can still get a six on the next roll. Sampling with replacement is sometimes called resampling. We’ll use the terms interchangeably.

#### Simple random sampling without replacement

If we take a simple random sample without replacement, each row of the dataset, or each type of coffee, can only appear once in the sample.

#### Simple random sampling with replacement

If we sample with replacement, it means that each row of the dataset, or each coffee, can be sampled multiple times.

#### Why sample with replacement?

So far, we’ve been treating the coffee\_ratings dataset as the population of all coffees. Of course, it doesn’t include every coffee in the world, so we could treat the coffee dataset as just being a big sample of coffees. To imagine what the whole population is like, we need to approximate the other coffees that aren’t in the dataset. Each of the coffees in the sample dataset will have properties that are representative of the coffees that we don’t have. Resampling lets us use the existing coffees to approximate those other theoretical coffees.

#### Coffee data preparation

To keep it simple, let’s focus on three columns of the coffee dataset. To make it easier to see which rows ended up in the sample, we’ll add a row index column called index using the reset\_index method.

#### Resampling with .sample()

To sample with replacement, we call sample as usual but set the replace argument to True. Setting frac to 1 produces a sample of the same size as the original dataset.

#### Repeated coffees

Counting the values of the index column shows how many times each coffee ended up in the resampled dataset. Some coffees were sampled four or five times.

#### Missing coffees

That means that some coffees didn’t end up in the resample. By taking the number of distinct index values in the resampled dataset, using len on drop\_duplicates, we see that eight hundred and sixty-eight different coffees were included. By comparing this number with the total number of coffees, we can see that four hundred and seventy coffees weren’t included in the resample.

#### Bootstrapping

We’re going to use resampling for a technique called bootstrapping. In some sense, bootstrapping is the opposite of sampling from a population. With sampling, we treat the dataset as the population and move to a smaller sample. With bootstrapping, we treat the dataset as a sample and use it to build up a theoretical population. A use case of bootstrapping is to try to understand the variability due to sampling. This is important in cases where we aren’t able to sample the population multiple times to create a sampling distribution.

#### Bootstrapping process

The bootstrapping process has three steps. First, randomly sample with replacement to get a resample the same size as the original dataset. Then, calculate a statistic, such as a mean of one of the columns. Note that the mean isn’t always the choice here and bootstrapping allows for complex statistics to be computed, too. Then, replicate this many times to get lots of these bootstrap statistics. Earlier in the course, we did something similar. We took a simple random sample, then calculated a summary statistic, then repeated those two steps to form a sampling distribution. This time, when we’ve used resampling instead of sampling, we get a bootstrap distribution.

#### Bootstrapping coffee mean flavor

The resampling step uses the code we just saw: calling sample with frac set to one and replace set to True. Calculating a bootstrap statistic can be done with mean from NumPy. In this case, we’re calculating the mean flavor score. To repeat steps one and two one thousand times, we can wrap the code in a for loop and append the statistics to a list.

#### Bootstrap distribution histogram

Here’s a histogram of the bootstrap distribution of the sample mean. Notice that it is close to following a normal distribution.

### 4.2 Exercise 4.1.1

#### Generating a bootstrap distribution

The process for generating a bootstrap distribution is similar to the process for generating a sampling distribution; only the first step is different.

To make a sampling distribution, you start with the population and sample without replacement. To make a bootstrap distribution, you start with a sample and sample that with replacement. After that, the steps are the same: calculate the summary statistic that you are interested in on that sample/resample, then replicate the process many times. In each case, you can visualize the distribution with a histogram.

Here, spotify\_sample is a subset of the spotify dataset. To make it easier to see how resampling works, a row index column called 'index' has been added, and only the artist name, song name, and danceability columns have been included.

#### Instructions

1. Generate a single bootstrap resample from spotify\_sample.
2. Calculate the mean of the danceability column of spotify\_1\_resample using numpy.
3. Replicate the expression provided 1000 times.
4. Create a bootstrap distribution by drawing a histogram of mean\_danceability\_1000.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course array
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

# Subset of spotify sample to use
spotify\_sample = spotify.sample(n=41656)[['artists', 'name', 'danceability']]
spotify\_sample['index'] = spotify\_sample.index

# Reorder columns to make 'index' the first column
spotify\_sample = spotify\_sample[['index', 'artists', 'name', 'danceability']]

# Generate 1 bootstrap resample
spotify\_1\_resample = spotify\_sample.sample(frac=1, replace = True)

# Print the resample
print(spotify\_1\_resample)

# Calculate of the danceability column of spotify\_1\_resample
mean\_danceability\_1 = np.mean(spotify\_1\_resample['danceability'])

# Print the result
print(mean\_danceability\_1)

# Replicate this 1000 times
mean\_danceability\_1000 = []
for i in range(1000):
 mean\_danceability\_1000.append(
 np.mean(spotify\_sample.sample(frac=1, replace=True)['danceability'])
 )

# Print the result
print(mean\_danceability\_1000)

# Draw a histogram of the resample means
plt.hist(mean\_danceability\_1000)
plt.show()

 index artists \
36560 36560 ['Duelo']
32711 32711 ['Omarion', 'Chris Brown', 'Jhené Aiko']
7129 7129 ['KK', 'Shilpa Rao']
28077 28077 ['Usher']
14234 14234 ['Don Omar', 'Zion & Lennox']
... ... ...
1158 1158 ['Ace Hood', 'Chris Brown']
14521 14521 ['XXXTENTACION']
18894 18894 ['Brownside']
27551 27551 ['Madvillain', 'Viktor Vaughn']
9247 9247 ['Roy Hargrove']

 name danceability
36560 Malabares 0.814
32711 Post to Be (feat. Chris Brown & Jhene Aiko) 0.733
7129 Khuda Jaane 0.382
28077 More - RedOne Jimmy Joker Remix 0.551
14234 Te Quiero Pa´Mi 0.759
... ... ...
1158 Body 2 Body 0.603
14521 HEARTEATER 0.780
18894 Gang Related 0.752
27551 Fancy Clown 0.537
9247 Strasbourg / St. Denis 0.701

[41656 rows x 4 columns]
0.5915618254273093
[0.5895104162665642, 0.5920538649894372, 0.5909811023622048, 0.5905931990589591, 0.5920963414634145, 0.5900679782024197, 0.59236746207029, 0.5916259050316881, 0.5913812560015365, 0.5927393052621471, 0.5916140579988477, 0.5909874663913962, 0.5906561095640483, 0.5906907048204341, 0.590806491261763, 0.5911371663145766, 0.5911827059727289, 0.5907639571730363, 0.5912479258690225, 0.5925287185519492, 0.5911390772037642, 0.5914279095448435, 0.5910838990781639, 0.591494809871327, 0.5915366501824467, 0.5921772109660073, 0.5910359275974649, 0.591645602074131, 0.5914523285961206, 0.5903951315536777, 0.5917938256193586, 0.5924170827731899, 0.5911717279623583, 0.5912213270597273, 0.5900511306894565, 0.5935490445554061, 0.5910545251584405, 0.5908897133666219, 0.5912672868254273, 0.5903685639523719, 0.5913791146533512, 0.5912484227962359, 0.5904973569233724, 0.5905739293259075, 0.5898012339158824, 0.5893314768580757, 0.5907354474745536, 0.5904249879969272, 0.5904865949683119, 0.5895198986940656, 0.5921094200115229, 0.5904701315536778, 0.5908653975417707, 0.5921549524678318, 0.5913408056462455, 0.5902100993854428, 0.5907465143076628, 0.591494236124448, 0.5923626968503938, 0.5911362540810448, 0.5909409712886499, 0.5889749687920108, 0.5908306870558864, 0.5919921427885539, 0.5915163457845208, 0.5901156544075282, 0.5922967471672749, 0.5905169723449202, 0.5911633858267717, 0.5919707149030151, 0.5916678989821396, 0.5899688112156712, 0.5915207245054734, 0.5910538649894373, 0.5900746447090456, 0.5907513971576723, 0.5907395837334357, 0.5901932734780104, 0.5933596168619167, 0.5923924884770502, 0.5921238116957942, 0.590638188976378, 0.5925187704052238, 0.5909347753024774, 0.591133584597657, 0.5914398838102554, 0.5917834309583252, 0.5908734083925484, 0.5917262939312464, 0.5927851594008066, 0.5928907288265797, 0.5912004225081622, 0.5923133066064913, 0.5924545203572115, 0.5908085677933551, 0.5932357187439984, 0.591788145765316, 0.590362207125024, 0.5897481923372383, 0.5906046115805647, 0.5923701267524486, 0.5911726353946611, 0.590763292202804, 0.592460673132322, 0.5919994286537353, 0.5908861700595354, 0.5909665978490495, 0.592525, 0.5911781039946226, 0.5909294099289418, 0.593399332629153, 0.590651567601306, 0.5910354114653351, 0.5903694497791434, 0.5915056246399077, 0.5927656736124447, 0.5904397445746112, 0.59141597128865, 0.5918214782984443, 0.5914657768388708, 0.5934042995006721, 0.5923447426541194, 0.5904835869982715, 0.5912211590167082, 0.589853540906472, 0.5915552189360476, 0.5906711830228539, 0.5919664370078741, 0.5926307230651047, 0.5917412353562512, 0.5910503552909545, 0.5895263923564433, 0.591842277222969, 0.5904030703860188, 0.5909793331092761, 0.5898830492606107, 0.5907840071058191, 0.5908963246591127, 0.5917549212598424, 0.5907246183022854, 0.5912272973881314, 0.590407499519877, 0.5906899438256193, 0.5902842879777224, 0.5906084021509506, 0.5909112060687536, 0.5915612324755137, 0.5913810207413097, 0.5914720400422508, 0.5924562536009218, 0.5905570938160168, 0.5882718335893989, 0.5898204964470904, 0.5912275518532744, 0.5929756289610141, 0.5914815104666793, 0.590001257922028, 0.5914331404839639, 0.5917607043403111, 0.5908593479930863, 0.5908518652775111, 0.5917406784136738, 0.5910876968503938, 0.5894494982715576, 0.5914262147109659, 0.5911345640483964, 0.5913416410601113, 0.590834235164202, 0.5913730699058959, 0.592596903207221, 0.5895507609948146, 0.5914924860764357, 0.5922455468599962, 0.5907647301709237, 0.5913652174956788, 0.5907198650854619, 0.5909580516612252, 0.5913367630113309, 0.5911002856731323, 0.5899691905127713, 0.5907312536009217, 0.5892633138083349, 0.591898622047244, 0.5905681366429807, 0.5910036417322834, 0.591160200211254, 0.59094484828116, 0.5900072714614941, 0.5901597536969464, 0.5918007369886691, 0.5930675364893413, 0.5896374231803342, 0.5917065128672941, 0.5904359155943921, 0.5913095784520837, 0.5898987444785866, 0.5897662953716152, 0.59041268964855, 0.5918970640483964, 0.5905993734396006, 0.5917523526022661, 0.5915707557134626, 0.5903329148261954, 0.5919412449587094, 0.5925458349337429, 0.589966057710774, 0.5907292466871519, 0.5911432734780104, 0.5911932590743231, 0.590324767140388, 0.5906311431726523, 0.5914263467447666, 0.5915819449779144, 0.591100254465143, 0.5924200907432302, 0.5906646749567889, 0.5904836590167083, 0.5925018412713655, 0.5914258882273863, 0.5915346936815825, 0.5920694713846745, 0.5904264091607451, 0.5900934151142693, 0.592132077011715, 0.590754561167659, 0.5923644997119263, 0.5906195770117151, 0.5913290498367583, 0.591609652871135, 0.590889094008066, 0.5906454484347993, 0.5922680838294603, 0.589898470808527, 0.5915262627232573, 0.5909589038793931, 0.5899698554830036, 0.5894641420203572, 0.5917860500288072, 0.5908633882273862, 0.5897776862876898, 0.5911361268484732, 0.5907302477434223, 0.5912980050893029, 0.5893935735548301, 0.5907020525254464, 0.5924074971192624, 0.5901721024582294, 0.5910894324947187, 0.5900547196082198, 0.5913464903015172, 0.5902943513539466, 0.5915473233147686, 0.5903019300941041, 0.5911154527559055, 0.5917445289994239, 0.5910247599385443, 0.5921206884962551, 0.5926452347801038, 0.5929926373151527, 0.5915101882081814, 0.590146984828116, 0.58922738621087, 0.5918866045707701, 0.5913529191473018, 0.592887168715191, 0.5915427285385059, 0.5922062776070675, 0.5908259842519685, 0.5908485668331093, 0.5908155487804877, 0.5904014451699635, 0.5908670875744192, 0.5921490085461879, 0.5908414850201652, 0.5920792706932975, 0.5920229474745535, 0.5913458181294411, 0.5910327107739581, 0.5916653327251776, 0.5905267596504705, 0.5900653783368544, 0.5923292971000577, 0.5932972873055502, 0.5923618374303822, 0.5919652775110429, 0.5926225297676205, 0.5917505425388899, 0.5923332221048588, 0.5923662689648551, 0.5910427405415786, 0.5905266828308047, 0.5902947306510468, 0.5901757489917419, 0.590131594488189, 0.5910178653735356, 0.5906333397349722, 0.5907370246783176, 0.5907637987324754, 0.5903726906087958, 0.5898537449587093, 0.5922480386979065, 0.5910296187824081, 0.5900186647781832, 0.5926432134626465, 0.5909962670443635, 0.5895221168619166, 0.5909472296908008, 0.5915199178989822, 0.5908074875168043, 0.5920343479930862, 0.591624284616862, 0.5902977218167851, 0.5905270933358939, 0.5914892452467831, 0.5902005137315154, 0.5923668499135779, 0.5913649510274631, 0.591306431246399, 0.5914231491261762, 0.5901321418283081, 0.5921464326867679, 0.5904616669867485, 0.5911870462838487, 0.5909412833685423, 0.5907421067793355, 0.591504750816209, 0.5905166770693298, 0.590216535433071, 0.5910725801805262, 0.5900449275014404, 0.5907752376608412, 0.5914499063760323, 0.5917791482619552, 0.5904276694833878, 0.5895461254081045, 0.5904834645669291, 0.5913697354522758, 0.5918416386594968, 0.5921093719992317, 0.5916242246014981, 0.590772299308623, 0.5913876344344152, 0.5899980362972921, 0.592214739773382, 0.5926581428845785, 0.5926025470520454, 0.590489929421932, 0.593214571730363, 0.591549517476474, 0.5907733579796428, 0.589716686671788, 0.5921093647973881, 0.5910049812752064, 0.5898160072978683, 0.5915966607451508, 0.5912878288841943, 0.5907502616669866, 0.5908610020165163, 0.5918163577875937, 0.5922746855194931, 0.5916611412521606, 0.5920023021893606, 0.590498324371039, 0.5915797340119071, 0.5905590719224121, 0.5910278591319378, 0.5923411465335126, 0.5911283368542347, 0.5916785745150759, 0.5904282144228924, 0.5931419819473785, 0.5912932518724794, 0.5893895957365086, 0.590654376320338, 0.5914407720376416, 0.5898587094296139, 0.5911709501632418, 0.5916655727866333, 0.5914198722873055, 0.5898745126752449, 0.5910201651622815, 0.5888763083349338, 0.5919166242558095, 0.592241665066257, 0.5903605939120415, 0.5923793235068178, 0.5907178533704628, 0.5916398886114845, 0.59051193105435, 0.5920392308430958, 0.5902751440368734, 0.5909317433262915, 0.5921461782216247, 0.5911793907240254, 0.5904648958133282, 0.5914622455348569, 0.5921820410024966, 0.5912036297292106, 0.5911381049548685, 0.5923477842327637, 0.5921147661801421, 0.5910707197042443, 0.5922372911465336, 0.5907696754369118, 0.5910222849049357, 0.5926157120222777, 0.5915913025734588, 0.5908865061455731, 0.5906375624159785, 0.5905398886114847, 0.5909829220280393, 0.5903094680238141, 0.5918502136546956, 0.5905219584213559, 0.5923888683502978, 0.5909354378720953, 0.5911624423852505, 0.5909809751296332, 0.590269598617246, 0.5900417538889956, 0.5913586806222393, 0.5900445962166314, 0.5913309511234877, 0.5907160457077011, 0.5902933070866141, 0.5921722248895718, 0.5913568585557903, 0.5903618014211638, 0.5904801061071634, 0.5926760610716344, 0.5925460677933552, 0.5925542898982139, 0.589064029191473, 0.5910023742077972, 0.5914539658152487, 0.5906170155559823, 0.5904142308430959, 0.5903994334549645, 0.5918233459765699, 0.5919954676397158, 0.5912229282696371, 0.5904264763779529, 0.5922159784904935, 0.59099430574227, 0.5918249735932398, 0.5906616069713847, 0.5909830900710582, 0.5909147157672363, 0.59135683214903, 0.5905852866333783, 0.5909721024582292, 0.5916164057998847, 0.5908632561935856, 0.5898437944113694, 0.5921035937199922, 0.5909051541194545, 0.5917114005185328, 0.5905484636066832, 0.590701051469176, 0.5912445602074132, 0.5917538769925101, 0.5897924980795084, 0.5924890051853274, 0.5914305070097945, 0.5910697066449011, 0.5910429373919723, 0.5906276766852314, 0.592151946898406, 0.5906092183599002, 0.5916534784904935, 0.5922059031111965, 0.590911496543115, 0.5923981443249473, 0.5892332533128481, 0.5908811743806415, 0.5911278159208757, 0.5900409616861917, 0.5919396653543308, 0.5905296019781064, 0.5924461734203956, 0.5917929710005762, 0.593105254945266, 0.5915802333397351, 0.5899635178605722, 0.5911470160361053, 0.5915435423468407, 0.5922400278471289, 0.5908435327443825, 0.5908038145765316, 0.5909375696178221, 0.5916684391204149, 0.5902631001536394, 0.5911292490877664, 0.5905174524678317, 0.5916395117149991, 0.591109945746111, 0.591521127808719, 0.5921187271941618, 0.5919770741309774, 0.5912502040522374, 0.5918139163625888, 0.5924842879777223, 0.5906234780103707, 0.5910093599961591, 0.589843931246399, 0.591162159112733, 0.5917802429421932, 0.5914565152679087, 0.5908299548684464, 0.5922724361436527, 0.5934835773958134, 0.5916694833877473, 0.5909251968503937, 0.5911905511811024, 0.5903563208181295, 0.5909329484347994, 0.5918504201075474, 0.5928208877472632, 0.589818568753601, 0.5909077347801037, 0.591140450355291, 0.5912375960245823, 0.5911795323602843, 0.5905165978490493, 0.5922111412521605, 0.5907123775686576, 0.590964756577684, 0.5907081356827348, 0.5902170275590551, 0.589956572882658, 0.5915438280199731, 0.5898980386979067, 0.5909591751488381, 0.5916436431726522, 0.5903003696946418, 0.5911154719608219, 0.5902762435183408, 0.5903332461110044, 0.5928585293835222, 0.5913845352410219, 0.5916196682350681, 0.591565097464951, 0.5928166722681006, 0.5925824299020549, 0.5917357451507586, 0.5907221096600729, 0.5913980170923757, 0.590945395621279, 0.5902898261955061, 0.589764571730363, 0.5915712142308431, 0.5915522325715383, 0.5915152631073555, 0.5909711998271558, 0.5910197378528903, 0.590958183695026, 0.5908760010562705, 0.5923513707509123, 0.5916730699058959, 0.5906152342999809, 0.5931575715383137, 0.5913665450355291, 0.591552136546956, 0.5899684391204147, 0.5923469896293451, 0.5900233195698099, 0.5923360476281928, 0.5910308502976762, 0.5900624903975418, 0.5922887987324755, 0.5912821826387554, 0.5921813616285769, 0.5910442385250625, 0.5930794411369311, 0.5917892068369504, 0.5935855002880738, 0.5922583133282121, 0.5907421235836374, 0.5914599577491838, 0.5910857835605915, 0.5918756097560975, 0.590704474745535, 0.5911015315920876, 0.5898670083541386, 0.591249474265412, 0.5913189024390244, 0.5896047700211254, 0.5925711302093336, 0.5912210269829076, 0.5918060615517572, 0.590475038409833, 0.5920414706164778, 0.5921699659112734, 0.5918078428077588, 0.5899075139235644, 0.5900431630497407, 0.5898658176493182, 0.592721204628385, 0.5913020213174573, 0.5909163745918956, 0.5910591223353178, 0.592325840215095, 0.5901460077779911, 0.5916420323602842, 0.5902679301901287, 0.5911964326867678, 0.5914916362588822, 0.5919113621086999, 0.5915742438064145, 0.5915785577107742, 0.5904565248703668, 0.5906581044747455, 0.5911656280007681, 0.5913880761474937, 0.5931929685999616, 0.5906928101594008, 0.5912401430766276, 0.5911526574803149, 0.5907070746111004, 0.5911423372383331, 0.5901576075475321, 0.5909340815248704, 0.5908506409640868, 0.5902037665642405, 0.5923575931438448, 0.5918306414442097, 0.591129815632802, 0.5905999831956982, 0.5914509674476667, 0.5920248583637411, 0.5920281976185903, 0.5905774726329941, 0.5894948290762436, 0.5913104474745534, 0.589961371711158, 0.591323341175341, 0.5907690320722104, 0.5903451579604378, 0.5910262963318609, 0.5904510802765507, 0.5924756385634723, 0.5907459237564816, 0.5920856515267908, 0.5902642644516997, 0.591139322066449, 0.591283980699059, 0.5901610404263491, 0.5889461206068755, 0.5905234924140579, 0.5925484468023814, 0.5905168787209525, 0.5903787905703859, 0.5914645453236028, 0.5929404383522183, 0.5912217471672748, 0.5912485788361822, 0.5930972080852698, 0.5904167442865373, 0.5907573866909929, 0.5911085749951988, 0.591444994718648, 0.5917371879201075, 0.5911553509698483, 0.5914694785865181, 0.5915081692913386, 0.5908304493950451, 0.5905993590359132, 0.5906368134242366, 0.590271811983868, 0.5907471096600729, 0.5899977482235452, 0.5913617486076436, 0.591874474265412, 0.5918312872095256, 0.5916637915306319, 0.5904788673900518, 0.5907520069137698, 0.5918353490493565, 0.59114815632802, 0.5907277174956789, 0.5904128672940273, 0.5918678557710774, 0.5917247527367006, 0.5917426517188401, 0.5925552573458804, 0.591164576531592, 0.5908115373535625, 0.5919541290570386, 0.5906556630497407, 0.5927567265219897, 0.589086182062608, 0.5931209093527944, 0.5899561575763395, 0.5914389451699636, 0.5911634314384483, 0.5918348065104667, 0.5916720832533129, 0.5904079004225081, 0.5913629681198387, 0.5909809535241022, 0.5915141564240445, 0.5914379513155368, 0.5904217471672747, 0.5920502592663722, 0.5912087238333013, 0.5918015988092953, 0.5926665066256961, 0.5907064984636067, 0.591054587574419, 0.5897825883426158, 0.58968142644517, 0.5915306702515843, 0.5913559487228731, 0.5896099145381217, 0.5910786897445747, 0.5931198650854619, 0.5907970232379489, 0.5906441016900327, 0.591794577011715, 0.5910762435183406, 0.5910730290954485, 0.5919165474361435, 0.5901577107739582, 0.5910544771461494, 0.5920410000960246, 0.5916246062992125, 0.590516751488381, 0.5899692073170733, 0.5904187055886306, 0.5925048300364892, 0.5894535505089303, 0.5905697690608795, 0.5923380209333589, 0.589893026214711, 0.5901097080852697, 0.5914770909352794, 0.5915854018628768, 0.5906573890916075, 0.5906722897061647, 0.5925765868062224, 0.5914911849433454, 0.5897879081044747, 0.5902611964662955, 0.59070553101594, 0.5921291362588823, 0.591566264163626, 0.5912934583253312, 0.5924487108699827, 0.5904612492798158, 0.5906321130209333, 0.5917593071826388, 0.5917744646629537, 0.5901601473977338, 0.5906756817745343, 0.5895982307470713, 0.5909275614557327, 0.590955022085654, 0.5913236244478586, 0.593070542058767, 0.5909419603418474, 0.5911718503937008, 0.5906150614557326, 0.5919772997887459, 0.5909845304397925, 0.591160699539082, 0.5915333469368159, 0.5897344680238141, 0.5907523814096408, 0.591440368734396, 0.5901951051469175, 0.5914118998463607, 0.5916272229690801, 0.5911126752448628, 0.5911374111772614, 0.5918231323218743, 0.5897556846552717, 0.5926011162857691, 0.5919691785096985, 0.5911894324947187, 0.5922375024006146, 0.5904272301709238, 0.5914091847512963, 0.5911750456116766, 0.5910951771653542, 0.591593487132706, 0.5903309895333205, 0.5920006865757633, 0.5914140363933167, 0.5924226065872863, 0.5903897661801422, 0.590944560207413, 0.5908001104282697, 0.5921178413673901, 0.5908666962742462, 0.591192939792587, 0.590640709621663, 0.592799474265412, 0.59089981035145, 0.5916036345304398, 0.5921616837910505, 0.5905093407912425, 0.5906862060687535, 0.5928581380833494, 0.590936057230651, 0.5902311599769541, 0.5920122887459189, 0.5909940704820434, 0.5929217591703477, 0.5917911225273671, 0.5902577083733436, 0.5891326339542923, 0.5925283056462454, 0.5907077035721146, 0.5919455012483196, 0.5905858579796428, 0.5930226618014212, 0.5903374567889379, 0.5922715551181104, 0.5905840311119647, 0.5895554205876704, 0.5922397757826003, 0.5903863621086999, 0.5911192865373537, 0.591950965047052, 0.5913642260418668, 0.5908873823698868, 0.5905810879585174, 0.5922888419435375, 0.5901008114077204, 0.5903280631841752, 0.5917646917610908, 0.5905746639139621, 0.5909198986940657, 0.5919477146149414, 0.591699668715191, 0.590746631937776, 0.591946377472633, 0.5910023670059535, 0.5918650182446706, 0.5907288169771463, 0.5909257177837527, 0.5921998775686576, 0.5911396893604762, 0.591654083445362, 0.5912180574227002, 0.592598324371039, 0.5911059175148838, 0.5894813040138275, 0.5910593671980028, 0.5887396125408104, 0.5918942289226042, 0.5903296956020742, 0.5911299644709045, 0.591719956308815, 0.5914278639331668, 0.5917781376032265, 0.5915208205300557, 0.5888879633186096, 0.5912036681390436, 0.5913317289226041, 0.5910916026502785, 0.5915978946610333, 0.5927605026886883, 0.5917458757441905, 0.5908031952179756, 0.5927210557902823, 0.5923026118686384, 0.5913688112156712, 0.5898892716535433, 0.5912295755713464, 0.5915398285961206, 0.5920121014979834, 0.5912810807566737, 0.5906077539850201, 0.5911751464374881, 0.5918932134626465, 0.5898934463222586, 0.5910569521797581, 0.5921712502400613, 0.591300547340119, 0.5919500672172077, 0.5916171547916266, 0.5903555670251583, 0.5891500408104474, 0.5918925604954869, 0.5916024750336085, 0.5923243758402151, 0.5905685327443825, 0.5918200043211063, 0.5925403783368541, 0.5908877208565393, 0.5914566232955637, 0.5910984083925485, 0.5907274366237757, 0.5920248199539082, 0.5915856467255618, 0.5909526574803149, 0.5913372815440752, 0.5910542874975994, 0.5904769228922604, 0.5891968960053773, 0.5911049644709045, 0.5906783512579219, 0.5923529767620511, 0.5915117438064144, 0.5905141492222009, 0.590865015844056, 0.592404604378721, 0.5897190248703669, 0.5910228898598042, 0.5902570770117149, 0.5905243566352987, 0.5920790066256962, 0.5923794507393894, 0.5925462862492799, 0.5929610356251199, 0.5925399702323795, 0.5906993446322258, 0.590764881409641, 0.5910344800268869, 0.5917554926061073, 0.5902258810255425, 0.5914879153063185, 0.5932616525830614, 0.5911077203764162, 0.592049392644517, 0.5904959645669291, 0.5923019757057806, 0.5914594920299597, 0.5914314000384099, 0.5908335293835222, 0.5893521029383523, 0.5920854546763972, 0.5908657552333397, 0.5925806582485116, 0.5902603202419818, 0.5914541578644132, 0.5918341415402343, 0.590723991741886, 0.5893890195890148, 0.5907534256769732, 0.5908773141924332, 0.5909922172076053, 0.5901537857691569, 0.5910998295563664, 0.5913653471288649, 0.5915506625696177, 0.5903682998847705, 0.5912238573074707, 0.5923461206068753, 0.5903535673132323, 0.5905137219128096, 0.5916193321490301, 0.5922834045515653, 0.5918762195121952, 0.5930626728442482, 0.5908968359900134, 0.5916877232571539, 0.5907557686767813, 0.5927893988861147, 0.5920899558286922, 0.591353015171884, 0.5906445362012676, 0.5915367870174765, 0.5909040210293836, 0.5906092327635875, 0.5900402270981372, 0.5911171763971578, 0.5906187175917035, 0.590494663433839, 0.591576740445554, 0.5903194953908201, 0.5911900182446707, 0.5907950787401575, 0.5917896005377375, 0.591022006433647, 0.5902348401190705, 0.5904482715575187, 0.5915358267716535, 0.5910615997695411, 0.5916588702707893, 0.5912389811791817, 0.5913589518916842, 0.5906167274822355, 0.5910779335509891, 0.5921085509890531, 0.5920041170539659, 0.5911538937968118, 0.5920053149606299, 0.591475038409833, 0.5914230507009796, 0.5909012219128096, 0.5922032264259651, 0.5913568945650086, 0.5930200115229498, 0.5909320458037258, 0.5906880089302862, 0.5915144348953332, 0.5912346864797388, 0.5923128432878816]



### 4.3 Chapter 4.2: Comparing sampling and bootstrap distributions

#### Coffee focused subset

we took a focused subset of the coffee dataset. Here’s a five hundred row sample from it.

#### The bootstrap of mean coffee flavors

Here, we generate a bootstrap distribution of the mean coffee flavor scores from that sample. .sample generates a resample, np.mean calculates the statistic, and the for loop with .append repeats these steps to produce a distribution of bootstrap statistics.

#### Mean flavor bootstrap distribution

Observing the histogram of the bootstrap distribution, which is close to a normal distribution.

#### Sample, bootstrap distribution, population means

Here’s the mean flavor score from the original sample. In the bootstrap distribution, each value is an estimate of the mean flavor score. Recall that each of these values corresponds to one potential sample mean from the theoretical population. If we take the mean of those means, we get our best guess of the population mean. The two values are really close. However, there’s a problem. The true population mean is actually a little different.

#### Interpreting the means

The behavior that you just saw is typical. The bootstrap distribution mean is usually almost identical to the original sample mean. However, that is not often a good thing. If the original sample wasn’t closely representative of the population, then the bootstrap distribution mean won’t be a good estimate of the population mean. Bootstrapping cannot correct any potential biases due to differences between the sample and the population.

#### Sample sd vs. bootstrap distribution sd

While we do have that limitation in estimating the population mean, one great thing about distributions is that we can also quantify variation. The standard deviation of the sample flavors is around 0.354. Recall that pandas .std calculates a sample standard deviation by default. If we calculate the standard deviation of the bootstrap distribution, specifying a ddof of one, then we get a completely different number. So what’s going on here?

#### Sample, bootstrap dist’n, pop’n standard deviations

Remember that one goal of bootstrapping is to quantify what variability we might expect in our sample statistic as we go from one sample to another. Recall that this quantity is called the standard error as measured by the standard deviation of the sampling distribution of that statistic. The standard deviation of the bootstrap means can be used as a way to estimate this measure of uncertainty. If we multiply that standard error by the square root of the sample size, we get an estimate of the standard deviation in the original population. Our estimate of the standard deviation is around point-three-five-three. The true standard deviation is around point-three-four-one, so our estimate is pretty close. In fact, it is closer than just using the sample standard deviation alone.

#### Interpreting the standard errors

To recap, the estimated standard error is the standard deviation of the bootstrap distribution values for our statistic of interest. This estimated standard error times the square root of the sample size gives a really good estimate of the standard deviation of the population. That is, although bootstrapping was poor at estimating the population mean, it is generally great for estimating the population standard deviation.

### 4.4 Exercise 4.2.1

#### Sampling distribution vs. bootstrap distribution

The sampling distribution and bootstrap distribution are closely linked. In situations where you can repeatedly sample from a population (these occasions are rare), it’s helpful to generate both the sampling distribution and the bootstrap distribution, one after the other, to see how they are related.

Here, the statistic you are interested in is the mean popularity score of the songs.

#### Instructions

1. Generate a sampling distribution of 2000 replicates.
* Sample 500 rows of the population without replacement and calculate the mean popularity.
1. Generate a bootstrap distribution of 2000 replicates.
* Sample 500 rows of the sample with replacement and calculate the mean popularity.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course array
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

spotify\_sample = spotify.sample(n=500)

mean\_popularity\_2000\_samp = []

# Generate a sampling distribution of 2000 replicates
for i in range(2000):
 mean\_popularity\_2000\_samp.append(
 # Sample 500 rows and calculate the mean popularity
 spotify.sample(n=500)['popularity'].mean()
 )

# Print the sampling distribution results
print(mean\_popularity\_2000\_samp)

mean\_popularity\_2000\_boot = []

# Generate a bootstrap distribution of 2000 replicates
for i in range(2000):
 mean\_popularity\_2000\_boot.append(
 # Resample 500 rows and calculate the mean popularity
 np.mean(spotify\_sample.sample(frac=1, replace=True)['popularity'])
 )

# Print the bootstrap distribution results
print(mean\_popularity\_2000\_boot)

[55.022, 54.85, 55.384, 55.386, 54.416, 54.744, 55.102, 55.472, 54.554, 55.266, 55.188, 54.916, 55.234, 54.472, 55.244, 55.2, 54.932, 55.75, 55.79, 55.358, 55.49, 55.526, 54.214, 53.94, 55.1, 55.18, 54.56, 54.942, 54.792, 54.842, 55.252, 54.586, 54.282, 54.616, 55.14, 55.046, 55.664, 55.762, 55.18, 55.342, 54.586, 54.52, 54.81, 54.812, 54.604, 54.33, 54.578, 54.744, 55.77, 54.716, 54.84, 54.464, 55.184, 53.77, 55.304, 54.874, 54.224, 54.02, 54.39, 54.542, 54.95, 54.242, 55.392, 54.344, 54.758, 54.92, 55.53, 55.272, 54.352, 55.726, 54.242, 55.106, 54.318, 54.262, 54.366, 54.406, 55.486, 54.75, 54.72, 55.268, 54.734, 54.75, 54.712, 55.55, 55.356, 55.364, 54.976, 55.046, 55.052, 54.852, 54.508, 54.862, 54.452, 54.906, 55.128, 55.21, 54.474, 54.09, 54.834, 54.312, 55.0, 54.602, 54.398, 54.682, 55.22, 54.976, 55.514, 55.52, 54.64, 55.092, 55.418, 54.622, 55.082, 54.956, 54.854, 55.37, 55.326, 54.76, 54.388, 54.984, 54.898, 54.738, 54.512, 54.708, 53.964, 54.87, 55.074, 54.848, 54.288, 54.72, 55.382, 55.056, 55.716, 55.362, 55.24, 54.77, 55.206, 55.124, 53.824, 55.768, 55.232, 54.626, 54.896, 55.102, 55.542, 55.254, 54.792, 54.814, 54.814, 54.814, 55.21, 54.064, 54.704, 54.592, 55.226, 54.15, 54.578, 55.23, 55.472, 54.708, 54.998, 55.214, 54.95, 55.266, 55.332, 54.134, 54.886, 54.738, 55.028, 54.588, 54.826, 55.356, 55.576, 54.754, 54.332, 55.068, 55.432, 55.232, 54.786, 55.192, 55.05, 55.478, 54.48, 55.594, 55.724, 54.918, 54.248, 54.534, 53.922, 54.906, 55.236, 54.614, 54.986, 55.12, 54.09, 54.59, 55.278, 55.69, 54.704, 54.858, 54.22, 55.308, 54.78, 54.214, 55.586, 54.958, 55.134, 55.054, 54.858, 55.22, 54.534, 55.284, 54.854, 56.028, 53.328, 54.642, 54.944, 54.226, 54.38, 55.192, 54.812, 55.23, 54.882, 54.248, 55.462, 54.186, 55.236, 54.856, 54.25, 54.968, 55.182, 54.646, 54.592, 55.024, 54.644, 54.062, 54.646, 54.844, 55.226, 54.934, 54.222, 54.876, 55.582, 55.03, 54.438, 55.38, 54.868, 55.462, 54.798, 54.734, 54.332, 55.268, 53.678, 54.698, 54.728, 54.072, 54.696, 53.622, 54.83, 54.728, 54.652, 53.866, 54.958, 54.304, 55.828, 55.112, 54.448, 55.182, 55.384, 54.962, 54.572, 53.994, 54.324, 54.68, 54.238, 54.818, 54.654, 54.69, 54.668, 53.816, 54.792, 54.192, 54.896, 55.098, 54.83, 55.106, 53.612, 55.1, 54.794, 55.234, 55.78, 54.674, 54.502, 55.194, 55.304, 54.598, 54.612, 56.096, 54.994, 54.756, 54.56, 54.178, 54.946, 55.118, 55.036, 55.258, 54.786, 54.736, 54.902, 55.026, 54.136, 54.954, 54.96, 54.566, 54.726, 55.65, 54.654, 55.684, 54.648, 55.422, 54.638, 55.37, 54.97, 54.53, 54.538, 55.238, 54.912, 55.094, 55.56, 54.91, 55.366, 55.716, 54.942, 54.816, 54.984, 54.836, 54.4, 54.326, 54.492, 54.932, 54.714, 55.012, 54.478, 55.712, 54.916, 55.118, 55.234, 54.734, 55.132, 54.68, 56.09, 54.578, 55.74, 55.286, 55.248, 55.834, 55.01, 55.014, 54.396, 54.632, 55.222, 54.754, 54.55, 54.734, 54.776, 54.914, 54.312, 54.864, 55.022, 55.112, 54.226, 54.066, 54.232, 55.138, 54.936, 54.76, 55.334, 54.934, 55.326, 55.442, 54.146, 55.146, 54.2, 56.036, 54.786, 56.32, 54.476, 54.746, 54.788, 54.942, 55.056, 54.264, 55.098, 54.916, 54.378, 54.19, 54.314, 54.56, 54.842, 54.692, 55.132, 55.16, 54.55, 55.396, 54.954, 55.494, 54.91, 55.172, 54.81, 54.532, 54.56, 54.304, 54.228, 55.176, 54.202, 55.366, 54.632, 55.388, 55.498, 55.448, 55.02, 54.446, 55.522, 54.782, 55.252, 55.044, 54.53, 55.022, 54.768, 55.064, 54.62, 53.958, 54.24, 54.684, 54.388, 54.714, 54.946, 54.716, 54.978, 54.608, 54.758, 54.642, 55.676, 54.346, 53.6, 55.624, 54.962, 54.036, 54.548, 54.68, 55.724, 54.372, 54.634, 54.504, 54.6, 54.264, 54.444, 53.71, 54.348, 55.39, 54.736, 54.518, 55.322, 53.64, 55.43, 54.986, 55.052, 53.994, 55.32, 54.262, 54.498, 54.752, 54.818, 54.348, 53.534, 53.706, 54.452, 54.372, 56.41, 54.76, 54.324, 54.5, 54.376, 54.888, 55.394, 55.732, 55.832, 54.462, 55.116, 54.842, 54.262, 55.058, 54.34, 54.494, 54.96, 54.364, 55.074, 54.806, 54.97, 54.886, 55.102, 54.696, 54.836, 55.432, 54.14, 55.432, 55.186, 55.552, 53.93, 55.096, 54.472, 54.274, 54.838, 55.156, 55.058, 54.116, 55.958, 54.924, 54.428, 54.85, 55.154, 54.974, 54.712, 54.844, 54.81, 54.442, 55.138, 55.01, 54.566, 55.098, 54.708, 54.73, 54.852, 55.524, 53.682, 54.976, 54.368, 55.544, 55.532, 55.156, 54.444, 54.546, 54.482, 55.668, 55.202, 55.848, 54.64, 54.776, 54.932, 54.716, 54.722, 54.48, 55.242, 55.066, 55.176, 54.394, 54.702, 54.654, 55.082, 54.402, 54.838, 54.612, 54.616, 54.882, 54.726, 54.66, 55.148, 54.964, 54.628, 55.184, 54.648, 53.618, 55.158, 55.37, 55.314, 54.39, 54.636, 55.226, 55.398, 54.826, 55.152, 54.402, 54.762, 54.698, 54.872, 54.47, 55.228, 55.086, 54.814, 55.106, 55.514, 55.078, 54.14, 54.682, 55.24, 54.43, 54.472, 54.32, 54.53, 55.062, 55.03, 55.29, 54.756, 53.926, 55.186, 54.678, 54.246, 55.05, 55.26, 54.85, 54.78, 54.032, 54.542, 54.882, 55.144, 54.276, 55.294, 54.494, 54.498, 55.742, 54.012, 54.948, 54.964, 54.62, 54.866, 54.82, 54.254, 55.194, 53.896, 55.096, 54.864, 55.498, 54.952, 54.45, 55.042, 55.17, 54.314, 54.36, 55.016, 55.882, 55.346, 55.01, 54.868, 55.44, 54.968, 55.152, 54.598, 54.026, 55.492, 55.03, 55.27, 55.12, 54.646, 55.612, 54.32, 54.906, 54.914, 55.386, 55.502, 54.802, 55.162, 55.048, 55.186, 55.242, 55.338, 54.764, 54.526, 54.24, 55.354, 54.276, 55.42, 54.668, 55.308, 54.778, 54.822, 54.764, 54.284, 54.934, 55.172, 54.69, 54.662, 54.954, 55.404, 54.448, 54.754, 54.868, 55.388, 54.98, 54.4, 54.444, 54.774, 55.248, 55.256, 54.104, 55.212, 54.144, 55.09, 55.134, 55.112, 54.812, 54.858, 54.838, 54.364, 54.238, 54.876, 54.966, 54.272, 55.54, 55.14, 54.606, 54.59, 54.496, 54.92, 54.774, 55.162, 54.642, 54.844, 54.49, 55.854, 55.394, 54.616, 55.322, 54.33, 54.618, 54.38, 54.494, 54.784, 54.322, 55.558, 55.368, 54.472, 54.708, 53.618, 55.064, 54.43, 54.432, 54.43, 54.656, 55.498, 54.21, 54.956, 55.0, 54.726, 55.118, 55.442, 55.21, 54.96, 54.556, 55.108, 55.162, 54.248, 55.14, 54.972, 55.578, 55.214, 54.4, 55.57, 54.804, 54.958, 54.23, 54.078, 55.014, 56.09, 55.626, 54.946, 54.696, 54.322, 54.164, 55.676, 54.516, 55.948, 54.842, 55.618, 55.856, 55.336, 54.724, 54.688, 55.626, 55.482, 54.69, 55.35, 54.802, 54.812, 54.752, 54.554, 54.052, 54.462, 54.728, 54.904, 55.134, 54.444, 55.36, 54.88, 54.874, 54.39, 53.78, 54.256, 54.668, 55.262, 55.264, 55.434, 54.966, 55.27, 56.006, 54.956, 56.03, 53.946, 55.142, 55.882, 55.092, 54.338, 55.224, 54.968, 54.146, 54.412, 54.262, 54.0, 54.996, 54.648, 55.646, 54.684, 54.726, 54.964, 55.026, 55.064, 55.752, 55.046, 54.67, 54.6, 54.622, 54.208, 54.192, 55.158, 54.61, 53.94, 54.896, 54.432, 54.032, 54.918, 54.982, 55.104, 55.236, 55.55, 54.826, 55.386, 55.788, 55.068, 54.264, 54.136, 54.116, 55.108, 54.452, 55.496, 54.8, 54.768, 53.822, 54.326, 54.634, 55.28, 55.362, 54.538, 54.472, 54.784, 55.446, 55.412, 54.164, 54.852, 55.794, 54.664, 54.92, 55.07, 54.704, 54.594, 54.524, 55.098, 54.414, 53.988, 55.092, 55.054, 54.746, 54.524, 54.658, 54.682, 54.57, 55.04, 54.694, 54.408, 55.14, 55.128, 55.156, 55.796, 54.518, 55.036, 54.134, 54.808, 54.336, 55.252, 55.358, 53.828, 54.306, 54.004, 54.128, 54.834, 54.354, 54.948, 54.034, 55.638, 54.862, 55.684, 54.328, 54.172, 54.72, 54.822, 55.144, 55.1, 54.966, 55.1, 54.484, 55.38, 54.876, 54.582, 54.084, 54.414, 55.3, 54.798, 54.95, 53.8, 54.172, 54.38, 54.414, 54.686, 55.04, 55.094, 55.336, 54.53, 55.566, 54.506, 54.568, 55.208, 55.136, 55.04, 54.046, 55.092, 54.4, 55.178, 54.308, 55.284, 54.882, 55.222, 54.22, 55.026, 55.422, 55.508, 53.944, 54.74, 54.592, 55.56, 55.694, 54.666, 54.394, 55.058, 54.836, 54.6, 54.654, 54.874, 55.586, 54.754, 54.822, 54.48, 55.692, 54.98, 55.094, 54.426, 55.266, 55.276, 55.648, 54.678, 55.876, 55.574, 54.528, 55.08, 54.354, 54.304, 53.864, 54.88, 55.416, 54.766, 55.858, 54.764, 55.208, 55.25, 54.218, 55.922, 55.48, 54.552, 55.016, 54.298, 54.792, 54.92, 54.67, 55.096, 55.536, 54.062, 55.114, 56.192, 54.44, 54.434, 55.454, 54.96, 55.46, 55.146, 54.212, 54.898, 54.704, 54.732, 54.598, 53.77, 54.748, 55.542, 54.746, 54.97, 54.636, 55.024, 55.31, 54.476, 54.298, 54.624, 55.158, 55.342, 54.942, 54.97, 54.874, 53.662, 54.044, 55.258, 55.6, 54.58, 55.206, 55.324, 54.796, 53.508, 54.504, 54.57, 53.996, 54.428, 54.294, 55.902, 54.208, 55.018, 54.296, 53.922, 55.526, 54.976, 55.228, 54.964, 54.188, 54.614, 54.462, 55.17, 55.082, 55.158, 55.846, 54.034, 55.242, 54.722, 55.324, 55.156, 54.726, 54.618, 55.194, 55.542, 55.236, 54.892, 55.214, 54.584, 53.948, 53.532, 54.96, 54.846, 54.51, 55.088, 54.012, 54.958, 54.544, 54.056, 55.604, 55.594, 55.628, 54.532, 54.91, 54.708, 55.054, 53.734, 54.93, 55.014, 54.97, 54.982, 53.876, 54.824, 55.154, 54.476, 55.39, 55.418, 55.9, 54.706, 54.156, 55.216, 54.172, 54.846, 55.056, 54.264, 55.068, 53.806, 55.196, 55.498, 54.82, 54.518, 55.258, 55.656, 54.608, 54.542, 55.434, 55.352, 54.474, 54.706, 54.818, 54.508, 54.492, 54.48, 55.704, 55.144, 54.472, 54.614, 54.124, 54.856, 54.758, 54.65, 54.954, 54.41, 53.956, 54.944, 54.932, 54.512, 54.952, 55.222, 54.516, 54.776, 54.992, 54.52, 54.728, 54.778, 54.592, 55.118, 54.258, 54.6, 54.498, 55.592, 55.294, 54.568, 54.426, 55.446, 54.972, 54.924, 54.46, 55.086, 55.374, 54.758, 54.382, 55.146, 54.066, 55.268, 54.73, 54.5, 55.346, 55.172, 55.738, 54.766, 54.934, 54.5, 55.44, 54.432, 54.424, 54.736, 54.766, 54.428, 54.558, 55.51, 55.28, 55.436, 54.428, 55.75, 55.066, 55.678, 54.656, 55.01, 54.94, 55.06, 54.682, 55.432, 54.568, 54.878, 54.88, 54.372, 53.992, 54.552, 54.686, 55.01, 54.66, 54.684, 55.616, 54.156, 53.774, 55.14, 54.61, 54.556, 55.46, 55.162, 54.766, 54.632, 55.454, 55.304, 55.078, 55.208, 55.65, 54.28, 54.972, 54.474, 55.108, 54.57, 54.762, 54.658, 55.87, 54.996, 54.838, 54.87, 54.456, 54.566, 55.602, 54.29, 54.828, 54.546, 54.966, 55.112, 54.47, 54.452, 55.456, 54.324, 55.008, 54.864, 54.756, 54.79, 54.166, 55.52, 55.068, 54.33, 56.032, 54.814, 54.546, 54.072, 54.53, 54.518, 54.63, 54.476, 54.05, 54.684, 54.358, 54.742, 54.42, 54.626, 54.878, 54.264, 54.034, 54.348, 55.28, 54.44, 54.944, 54.728, 55.026, 55.024, 55.024, 55.924, 54.598, 55.326, 54.482, 54.652, 54.39, 53.88, 54.686, 55.138, 54.654, 54.086, 54.466, 54.914, 54.73, 54.39, 55.274, 54.194, 55.56, 54.89, 54.066, 53.806, 56.062, 54.546, 54.974, 55.08, 54.114, 55.468, 54.432, 55.854, 54.768, 55.166, 54.208, 55.424, 55.724, 54.316, 56.112, 54.124, 54.47, 54.918, 54.838, 54.856, 54.276, 54.728, 54.262, 54.416, 54.846, 54.196, 54.328, 54.494, 54.756, 54.878, 54.092, 54.392, 55.51, 55.012, 53.178, 54.446, 54.622, 54.776, 54.602, 55.694, 55.5, 55.41, 54.508, 54.65, 54.254, 55.158, 55.252, 54.938, 54.736, 55.088, 54.762, 53.572, 54.95, 54.65, 54.208, 53.61, 54.37, 54.38, 55.296, 54.504, 54.77, 54.668, 54.456, 55.476, 54.876, 54.9, 53.838, 55.19, 54.776, 54.452, 53.938, 54.874, 54.11, 55.17, 55.086, 54.884, 54.288, 54.702, 54.348, 54.48, 54.952, 55.126, 54.61, 54.07, 54.786, 54.364, 54.914, 54.522, 55.598, 54.596, 54.548, 55.134, 54.822, 54.852, 54.506, 54.946, 54.354, 55.076, 54.354, 55.73, 55.634, 55.058, 53.958, 55.916, 54.53, 55.664, 55.462, 54.266, 55.062, 55.26, 54.614, 54.794, 54.76, 53.552, 54.194, 54.994, 55.312, 54.72, 54.552, 54.498, 54.172, 55.342, 54.648, 54.606, 55.156, 54.72, 54.974, 54.918, 55.14, 55.21, 55.512, 55.1, 55.172, 54.876, 54.578, 55.412, 54.61, 54.572, 54.968, 53.734, 55.076, 55.39, 54.748, 55.248, 54.166, 55.298, 54.752, 54.584, 54.35, 54.054, 55.128, 54.894, 55.27, 55.048, 55.686, 54.378, 54.02, 55.176, 55.29, 54.952, 55.412, 54.2, 54.74, 54.892, 55.456, 55.642, 54.806, 54.996, 55.046, 55.49, 54.954, 55.54, 55.71, 54.644, 55.438, 53.612, 55.16, 54.876, 54.862, 54.532, 54.652, 55.084, 54.46, 55.53, 54.754, 55.338, 55.032, 55.066, 54.056, 54.67, 54.972, 54.552, 54.894, 55.914, 54.834, 55.312, 54.324, 54.664, 53.95, 56.068, 54.284, 55.12, 54.216, 54.738, 54.524, 56.096, 55.648, 55.71, 55.254, 55.67, 55.932, 55.292, 55.344, 54.756, 55.218, 54.032, 54.212, 54.98, 54.488, 54.872, 55.112, 55.262, 54.846, 54.198, 55.154, 54.654, 54.842, 55.378, 54.51, 54.484, 55.084, 54.226, 55.26, 55.092, 55.872, 54.498, 54.62, 54.754, 55.034, 54.706, 54.656, 55.102, 55.338, 55.482, 55.378, 54.688, 54.426, 55.826, 54.756, 55.532, 54.318, 55.716, 56.312, 54.546, 54.864, 54.4, 55.334, 55.39, 54.932, 55.108, 54.722, 55.314, 54.414, 55.368, 55.69, 54.288, 54.226, 55.012, 55.938, 54.556, 56.346, 54.476, 55.252, 55.438, 55.59, 54.362, 54.192, 55.592, 54.204, 55.506, 54.462, 54.756, 55.288, 55.114, 54.782, 54.466, 55.084, 54.374, 55.91, 54.856, 55.54, 54.874, 54.512, 55.752, 54.822, 55.048, 56.018, 54.628, 55.61, 55.102, 54.502, 54.064, 55.354, 54.766, 53.702, 54.436, 54.792, 55.566, 55.08, 54.422, 54.21, 54.832, 54.1, 55.28, 54.88, 54.604, 54.658, 54.764, 55.034, 53.692, 54.618, 53.336, 55.22, 53.908, 54.908, 54.166, 55.17, 54.336, 54.856, 54.634, 54.44, 54.656, 55.29, 55.236, 55.002, 54.208, 54.398, 55.346, 55.5, 55.106, 55.148, 54.428, 55.442, 55.126, 55.048, 54.396, 55.63, 55.07, 54.954, 54.71, 54.192, 54.988, 55.086, 55.304, 54.268, 55.052, 55.348, 54.334, 55.13, 54.372, 55.362, 55.282, 54.23, 55.004, 54.164, 54.672, 54.1, 55.718, 54.764, 53.906, 55.054, 55.056, 55.546, 55.286, 54.892, 54.426, 55.03, 54.958, 54.246, 54.404, 54.834, 54.918, 55.132, 54.972, 53.814, 54.678, 54.5, 54.478, 54.348, 54.636, 55.252, 55.12, 55.228, 55.078, 55.94, 54.42, 54.418, 55.012, 54.874, 54.24, 54.798, 55.102, 54.678, 54.158, 56.008, 54.992, 54.294, 54.104, 54.894, 55.362, 54.538, 54.284, 54.828, 55.0, 54.852, 54.334, 54.634, 54.848, 54.838, 55.526, 54.616, 54.606, 54.644, 54.58, 54.836, 54.648, 54.702, 54.368, 55.208, 54.694, 54.89, 54.642, 55.01, 53.848, 54.54, 55.522, 54.248, 54.392, 54.21, 55.134, 54.81, 54.406, 54.654, 54.418, 54.054, 55.376, 55.062, 54.524, 54.332, 54.356, 55.52, 54.466, 55.532, 54.604, 55.096, 55.032, 55.158, 55.75, 54.264, 53.66, 55.218, 54.784, 55.726, 53.496, 54.742, 54.726, 54.994, 54.338, 54.288, 53.816, 53.446, 55.286, 54.56, 54.728, 54.256, 55.732, 54.724, 54.786, 54.232, 55.214, 54.888, 55.386, 55.39, 54.182, 55.058, 55.666, 54.572, 53.456, 54.724, 54.356, 55.27, 54.746, 55.806, 54.246, 55.642, 54.726, 54.446, 55.192, 54.176, 55.026, 54.71, 54.376, 56.108, 54.764, 54.862, 54.478, 55.728, 55.034, 55.52, 54.694, 54.778, 55.416, 54.39, 54.314, 54.22, 54.474, 54.458, 54.798, 53.83, 55.366, 54.416, 53.598, 54.642, 54.64, 54.862, 55.81, 54.304, 55.394, 55.12, 55.122, 56.102, 54.442, 54.752, 54.018, 53.782, 55.17, 55.544, 55.772, 54.814, 54.822, 54.4, 55.252, 55.034, 54.528, 55.084, 55.052, 54.526, 55.036, 54.256, 54.258, 54.506, 54.846, 54.384, 54.03, 55.186, 53.472, 56.24, 55.304, 55.164, 54.58, 55.394, 54.296, 54.414, 54.458, 54.254, 54.918, 55.112, 54.508, 53.932, 54.858, 55.246, 55.342, 54.898, 54.648, 54.95, 55.408, 55.3, 54.016, 54.904, 54.296, 54.168, 54.284, 54.486, 55.08, 55.522, 54.286, 54.822, 54.758, 54.768, 55.116, 54.778, 54.15, 54.836, 55.712, 54.642, 55.594, 55.05, 54.774, 55.546, 54.564, 55.772, 55.03, 55.192, 55.55, 54.744, 54.502, 54.96, 53.828, 55.142, 53.948, 54.118, 55.764, 54.834, 53.944, 54.842, 54.662, 54.89, 55.162, 54.974, 54.62, 54.876, 55.338, 55.862, 54.436, 54.69, 54.826, 54.984, 54.68, 55.53, 55.37, 54.414, 54.726, 54.338, 54.814, 54.958, 55.318, 54.608, 54.7, 54.992, 54.976, 54.972, 54.672, 54.528, 55.15, 54.21, 54.24, 54.418, 55.592, 55.348, 55.712, 54.414, 55.652, 54.574, 54.208, 54.724, 54.798, 54.95, 54.618, 55.104, 54.646, 54.768, 55.24, 54.474, 54.844, 54.776, 55.108, 55.28, 54.702, 54.77, 55.304, 53.874, 54.34, 54.894, 54.154, 55.236, 55.018, 54.6, 54.864, 54.858, 54.768, 55.458, 54.572, 54.176, 54.612, 54.126, 55.54, 53.876, 54.814, 54.74, 53.93, 54.468, 54.542, 55.352, 54.364, 54.91, 54.238, 54.932, 54.73, 55.09]
[54.382, 53.756, 54.542, 54.378, 54.744, 54.542, 54.932, 54.014, 53.176, 54.074, 54.34, 54.51, 55.43, 55.146, 55.548, 55.528, 54.21, 53.848, 54.334, 54.81, 53.782, 54.102, 55.396, 54.366, 54.642, 54.09, 55.02, 53.434, 55.364, 54.052, 54.286, 54.432, 54.49, 54.578, 53.844, 54.258, 54.552, 55.428, 54.8, 53.232, 54.912, 54.27, 54.226, 54.324, 55.114, 54.098, 54.734, 54.022, 53.658, 54.462, 53.55, 54.476, 54.536, 54.65, 54.314, 54.174, 54.342, 54.31, 54.828, 54.586, 54.516, 54.412, 54.02, 54.22, 54.468, 54.596, 54.51, 54.214, 54.65, 53.548, 53.778, 53.872, 54.758, 54.444, 53.542, 53.872, 54.0, 54.384, 54.358, 53.532, 54.886, 53.952, 54.15, 54.712, 54.626, 54.244, 54.368, 55.0, 53.554, 54.104, 54.836, 54.124, 53.576, 54.442, 54.222, 54.33, 53.788, 54.708, 54.036, 54.384, 53.682, 54.526, 55.292, 54.38, 54.024, 53.776, 53.93, 53.838, 54.224, 54.26, 55.446, 55.3, 54.436, 53.966, 53.814, 53.686, 54.338, 54.66, 52.842, 54.112, 54.038, 54.176, 54.94, 54.252, 54.118, 53.666, 54.134, 54.062, 53.642, 54.334, 54.454, 54.826, 54.492, 53.8, 53.43, 55.39, 54.722, 53.924, 54.596, 54.136, 53.7, 54.416, 54.83, 55.366, 54.262, 55.224, 55.126, 55.112, 54.39, 54.786, 54.058, 54.032, 54.564, 53.792, 54.172, 53.808, 53.814, 54.822, 54.434, 54.844, 53.992, 54.42, 53.778, 54.252, 53.476, 54.164, 54.116, 53.928, 55.08, 54.36, 54.218, 53.4, 54.56, 54.81, 54.594, 56.114, 54.482, 54.446, 53.95, 54.444, 53.852, 53.736, 54.456, 54.47, 54.762, 54.786, 53.89, 53.626, 54.574, 53.992, 54.266, 54.518, 54.41, 53.952, 54.638, 54.36, 54.474, 53.448, 54.974, 54.182, 54.262, 54.384, 55.194, 54.208, 54.016, 55.24, 54.308, 54.164, 54.254, 54.226, 54.57, 54.748, 54.89, 54.504, 54.28, 54.086, 53.412, 54.636, 53.72, 54.762, 54.912, 54.716, 53.646, 54.044, 54.608, 54.094, 54.48, 53.778, 54.644, 54.41, 54.576, 54.278, 53.82, 53.52, 54.408, 54.058, 53.926, 54.254, 55.826, 53.214, 54.196, 54.866, 53.974, 54.81, 54.62, 54.75, 54.504, 55.002, 54.292, 54.354, 54.18, 54.738, 54.478, 54.648, 54.006, 53.466, 54.61, 54.152, 54.748, 54.196, 53.942, 54.262, 54.24, 54.356, 53.576, 54.354, 54.426, 54.548, 54.606, 55.584, 53.644, 54.44, 54.222, 53.816, 54.172, 53.752, 53.978, 54.934, 53.934, 54.346, 54.614, 54.062, 55.042, 55.124, 54.094, 54.77, 54.376, 54.894, 54.35, 54.706, 53.532, 53.402, 54.396, 54.744, 54.594, 54.712, 53.294, 53.706, 55.268, 53.894, 54.208, 53.908, 54.748, 54.438, 54.444, 53.996, 54.394, 53.784, 53.262, 54.264, 53.744, 55.104, 54.332, 54.55, 54.482, 54.508, 54.102, 55.332, 54.95, 54.648, 54.608, 54.184, 53.396, 53.904, 54.352, 54.344, 55.176, 54.162, 54.314, 54.15, 54.264, 54.022, 53.932, 54.602, 54.872, 54.436, 54.512, 54.404, 54.668, 54.386, 54.856, 54.29, 54.156, 54.254, 54.372, 54.226, 53.73, 53.886, 54.558, 53.746, 54.744, 53.408, 54.09, 54.584, 54.566, 53.93, 54.226, 55.316, 53.908, 53.82, 54.394, 54.778, 53.532, 54.378, 53.586, 54.202, 54.612, 53.582, 53.732, 54.158, 54.298, 55.002, 53.996, 53.572, 53.748, 54.078, 54.472, 54.758, 54.514, 54.878, 54.244, 54.194, 54.164, 53.954, 54.496, 54.602, 54.238, 54.302, 54.104, 54.834, 54.514, 55.21, 54.372, 54.246, 54.128, 54.532, 53.71, 54.804, 53.52, 53.916, 54.87, 55.262, 54.26, 53.946, 53.892, 55.206, 54.428, 54.808, 53.826, 54.912, 54.852, 54.732, 54.506, 54.728, 53.848, 54.23, 54.86, 55.21, 53.89, 54.532, 53.706, 54.71, 55.336, 53.306, 54.778, 54.052, 53.686, 54.332, 54.462, 53.814, 54.004, 54.06, 55.06, 54.17, 54.45, 55.358, 54.212, 54.168, 54.266, 53.914, 53.546, 54.004, 54.24, 54.38, 54.216, 53.63, 54.972, 54.394, 54.384, 53.416, 54.638, 54.274, 54.624, 55.166, 53.968, 53.296, 53.688, 54.398, 54.242, 54.388, 54.04, 54.728, 53.796, 53.998, 54.386, 53.868, 54.596, 54.472, 54.524, 54.206, 55.072, 55.224, 54.368, 54.652, 54.572, 53.882, 55.278, 54.166, 54.18, 54.502, 54.508, 53.86, 54.83, 53.786, 53.92, 54.34, 54.18, 53.87, 54.064, 54.612, 53.51, 53.94, 54.206, 54.646, 53.422, 53.886, 54.776, 53.398, 54.06, 55.036, 53.866, 53.918, 54.118, 54.162, 53.622, 54.948, 54.628, 54.484, 53.84, 54.248, 53.816, 54.102, 54.0, 54.41, 53.942, 55.114, 54.698, 54.114, 54.262, 53.688, 54.422, 54.23, 55.374, 54.334, 53.574, 54.004, 54.664, 54.976, 54.564, 55.058, 53.5, 53.666, 54.654, 54.606, 54.212, 54.182, 54.922, 54.046, 54.692, 54.568, 53.144, 54.022, 54.22, 53.664, 54.11, 53.758, 55.034, 54.104, 54.2, 53.394, 55.192, 54.018, 54.304, 54.802, 53.734, 54.05, 54.182, 54.076, 54.866, 54.53, 54.754, 53.82, 55.262, 54.558, 54.78, 54.11, 53.612, 53.966, 54.034, 55.258, 53.61, 54.976, 53.774, 53.998, 54.618, 53.258, 54.758, 55.148, 54.726, 53.636, 54.318, 54.474, 55.276, 54.086, 54.54, 54.5, 54.254, 54.114, 54.676, 54.402, 54.078, 54.108, 55.09, 53.762, 54.2, 53.724, 55.148, 54.238, 53.962, 53.816, 54.45, 54.628, 54.304, 53.622, 54.234, 54.75, 54.324, 54.806, 54.906, 54.11, 53.742, 54.136, 54.774, 53.676, 53.726, 53.982, 54.164, 53.902, 54.428, 53.756, 54.196, 53.914, 54.84, 55.298, 54.126, 54.218, 55.208, 54.224, 53.568, 54.658, 54.036, 54.374, 54.396, 53.972, 54.264, 54.28, 54.81, 54.288, 54.378, 55.344, 54.22, 54.068, 54.486, 54.382, 54.154, 54.014, 54.468, 54.186, 53.538, 54.686, 54.254, 54.452, 53.548, 53.742, 54.662, 54.522, 54.112, 54.492, 54.83, 53.938, 54.49, 54.108, 54.24, 54.334, 54.586, 54.204, 54.15, 54.838, 53.97, 53.926, 54.362, 54.274, 54.38, 54.932, 53.238, 54.9, 54.442, 54.368, 54.182, 53.966, 54.188, 54.008, 54.172, 54.886, 54.468, 54.654, 54.498, 54.166, 53.598, 54.414, 53.232, 53.664, 54.474, 53.632, 53.324, 54.188, 54.564, 53.47, 53.732, 54.444, 53.734, 53.918, 54.444, 54.296, 54.394, 54.728, 54.432, 54.498, 54.498, 54.754, 55.044, 54.414, 54.648, 54.592, 53.938, 54.744, 54.644, 54.228, 54.482, 54.49, 54.644, 54.46, 53.986, 54.826, 54.194, 54.646, 54.752, 54.048, 54.438, 53.67, 54.32, 53.984, 54.298, 54.018, 54.58, 54.262, 54.088, 54.014, 53.678, 53.972, 55.058, 53.568, 54.736, 54.17, 54.55, 54.036, 53.984, 54.25, 54.124, 54.206, 54.054, 53.97, 54.0, 54.644, 54.854, 53.442, 54.036, 53.504, 53.594, 55.504, 54.252, 54.844, 54.608, 54.178, 54.0, 55.094, 54.326, 54.156, 54.126, 55.066, 53.748, 55.058, 52.858, 54.114, 53.532, 54.32, 54.424, 54.576, 54.924, 54.616, 54.362, 54.142, 54.494, 53.114, 54.118, 54.542, 54.7, 53.978, 54.56, 54.086, 54.368, 54.806, 54.726, 54.642, 53.78, 54.7, 53.8, 54.564, 54.344, 54.298, 54.798, 54.232, 54.758, 53.972, 55.044, 55.204, 54.996, 54.696, 55.33, 54.052, 53.262, 54.596, 54.26, 53.782, 54.446, 53.48, 53.622, 53.824, 54.63, 53.918, 53.674, 55.234, 54.068, 54.17, 55.238, 54.514, 55.316, 54.766, 54.922, 54.36, 53.548, 53.976, 54.288, 53.594, 54.59, 54.166, 54.576, 54.668, 53.632, 54.462, 54.17, 54.566, 53.64, 55.048, 54.924, 54.09, 54.346, 54.186, 53.846, 54.132, 54.672, 53.524, 54.71, 54.534, 53.874, 54.924, 53.882, 53.474, 54.074, 54.704, 54.512, 54.328, 53.684, 55.014, 54.136, 53.712, 54.346, 54.47, 54.802, 54.494, 54.302, 53.632, 53.786, 54.628, 54.58, 53.774, 55.434, 55.48, 54.262, 54.286, 55.224, 54.22, 54.57, 53.698, 54.292, 54.634, 54.906, 53.99, 54.002, 53.934, 53.932, 54.656, 55.164, 54.548, 55.34, 54.264, 54.706, 54.926, 54.576, 54.076, 53.434, 53.662, 54.262, 54.38, 53.706, 54.534, 54.28, 54.362, 54.918, 54.26, 53.8, 55.42, 53.832, 54.096, 54.464, 54.246, 54.838, 54.704, 54.488, 55.102, 54.254, 54.834, 53.824, 54.09, 53.406, 54.61, 54.176, 53.84, 54.764, 53.624, 55.008, 53.398, 55.068, 54.618, 54.456, 54.576, 54.592, 54.914, 54.886, 54.036, 55.018, 54.024, 53.546, 54.076, 54.844, 54.202, 54.174, 54.12, 53.632, 53.55, 54.434, 54.026, 54.06, 54.73, 53.798, 54.048, 54.526, 54.15, 54.33, 54.344, 53.848, 53.712, 55.634, 53.654, 53.054, 54.178, 53.758, 53.518, 54.084, 54.608, 54.312, 54.662, 53.87, 54.03, 54.566, 54.15, 54.378, 54.334, 55.076, 54.748, 54.52, 54.63, 55.792, 54.088, 54.484, 55.126, 53.67, 54.53, 53.944, 54.534, 54.462, 53.472, 53.686, 53.94, 54.864, 54.84, 54.4, 53.772, 54.538, 54.764, 54.112, 54.556, 53.602, 54.382, 54.742, 54.076, 53.578, 53.392, 54.732, 54.356, 54.478, 55.644, 54.736, 53.678, 54.198, 54.474, 54.534, 54.408, 55.43, 53.392, 54.614, 53.84, 54.446, 54.652, 54.792, 54.184, 54.406, 55.082, 54.072, 55.374, 55.102, 55.3, 54.758, 54.454, 54.962, 55.098, 54.08, 54.61, 53.272, 54.81, 53.612, 53.88, 54.546, 54.706, 54.084, 55.584, 54.42, 53.496, 54.272, 53.972, 54.472, 53.954, 54.904, 54.758, 53.646, 54.094, 54.612, 55.012, 53.916, 54.564, 54.504, 55.492, 54.268, 54.738, 53.704, 53.65, 54.1, 53.82, 54.428, 54.92, 54.194, 55.196, 54.93, 53.398, 54.396, 54.284, 54.17, 54.274, 53.822, 55.142, 53.498, 54.342, 54.184, 53.538, 53.968, 53.044, 54.134, 54.852, 53.446, 54.858, 54.9, 54.414, 54.496, 54.412, 53.496, 55.318, 53.826, 54.778, 54.132, 54.652, 54.05, 54.034, 54.958, 55.056, 54.794, 54.306, 53.892, 54.418, 53.884, 54.242, 53.654, 54.228, 54.07, 53.68, 53.444, 53.716, 54.008, 53.742, 54.278, 54.39, 55.258, 54.824, 53.672, 54.856, 55.178, 53.932, 54.682, 54.576, 54.538, 53.994, 53.576, 55.01, 54.31, 54.936, 54.212, 53.71, 54.358, 54.648, 53.292, 54.67, 53.88, 54.374, 54.084, 54.08, 54.312, 54.274, 54.372, 53.776, 54.954, 54.324, 54.458, 54.49, 53.95, 54.894, 53.754, 53.45, 54.654, 54.552, 54.532, 53.646, 53.52, 54.344, 53.688, 53.804, 53.718, 53.722, 54.048, 54.124, 53.936, 54.486, 54.228, 54.002, 53.84, 54.538, 54.036, 53.492, 54.776, 54.058, 54.49, 53.83, 54.122, 54.94, 54.034, 53.952, 54.056, 54.902, 53.34, 54.164, 54.292, 54.204, 52.996, 53.91, 54.142, 54.002, 53.27, 54.404, 54.02, 54.202, 55.318, 55.09, 55.052, 54.238, 55.38, 53.586, 54.784, 53.81, 53.854, 53.854, 54.244, 54.038, 54.532, 53.94, 54.058, 54.676, 54.944, 54.984, 54.436, 53.382, 54.806, 54.482, 54.71, 54.22, 53.374, 54.296, 53.62, 54.716, 54.82, 54.608, 53.878, 54.576, 54.688, 54.166, 54.106, 54.55, 55.11, 53.926, 53.942, 53.77, 53.874, 55.186, 53.808, 54.414, 54.254, 54.628, 54.628, 54.326, 55.222, 54.142, 55.074, 54.294, 54.81, 54.364, 54.126, 55.292, 54.728, 53.946, 54.072, 52.586, 53.646, 54.24, 54.384, 54.446, 54.484, 54.508, 55.234, 54.514, 54.064, 54.016, 54.246, 54.15, 54.372, 53.73, 54.278, 54.998, 54.052, 54.672, 54.646, 54.366, 53.512, 54.592, 53.856, 54.46, 54.984, 53.478, 53.656, 54.26, 53.018, 54.31, 55.194, 55.226, 55.254, 53.892, 54.218, 54.056, 53.76, 53.97, 53.614, 54.168, 54.74, 54.57, 53.812, 53.958, 54.458, 53.198, 53.5, 54.788, 55.2, 53.856, 53.674, 54.102, 54.422, 53.834, 53.998, 53.264, 53.962, 53.718, 54.442, 54.18, 53.882, 54.502, 54.212, 54.444, 54.838, 54.782, 54.44, 54.9, 55.26, 53.916, 54.104, 53.61, 54.44, 53.734, 54.534, 54.608, 55.258, 53.65, 54.81, 53.88, 53.908, 54.874, 53.998, 54.108, 54.85, 53.784, 54.448, 53.688, 54.368, 54.354, 54.892, 54.052, 54.99, 54.334, 54.838, 53.898, 53.624, 52.768, 53.622, 53.886, 54.084, 54.576, 54.156, 54.348, 54.07, 54.136, 55.022, 53.586, 53.88, 54.628, 55.606, 53.59, 53.894, 54.758, 54.372, 54.982, 53.824, 55.046, 54.62, 54.312, 53.812, 53.986, 55.03, 54.464, 54.916, 53.138, 54.9, 54.694, 55.246, 55.086, 54.328, 53.92, 54.216, 54.712, 54.176, 53.806, 54.986, 54.022, 53.76, 54.114, 55.064, 54.518, 54.314, 55.122, 53.744, 54.328, 54.238, 54.434, 54.45, 53.92, 55.232, 54.854, 54.366, 53.138, 53.764, 54.6, 54.13, 54.234, 54.194, 54.768, 54.808, 53.062, 54.032, 54.98, 54.178, 54.972, 54.268, 53.838, 54.612, 54.348, 53.434, 53.474, 54.662, 53.688, 55.072, 53.946, 55.1, 53.466, 54.286, 54.808, 54.206, 53.842, 55.004, 53.778, 53.912, 53.952, 54.37, 54.836, 54.344, 54.902, 54.18, 54.64, 54.622, 55.028, 53.624, 53.642, 54.394, 53.816, 53.028, 54.454, 53.75, 54.23, 54.428, 54.284, 54.636, 53.654, 53.624, 54.2, 54.418, 54.44, 54.684, 54.762, 54.738, 54.838, 53.868, 54.1, 54.518, 54.07, 54.684, 54.224, 54.128, 54.752, 53.686, 53.912, 54.758, 53.474, 54.764, 53.788, 54.538, 54.358, 54.584, 54.43, 55.458, 54.132, 54.072, 55.01, 54.236, 54.018, 54.39, 53.838, 53.76, 54.084, 54.526, 54.836, 54.138, 53.576, 53.326, 53.474, 53.912, 54.098, 53.262, 53.738, 53.876, 54.574, 53.758, 54.224, 54.438, 55.008, 54.738, 53.99, 54.134, 54.628, 54.654, 53.46, 54.194, 53.852, 54.608, 54.8, 54.66, 53.638, 53.152, 54.878, 54.04, 54.404, 53.65, 54.044, 54.29, 54.528, 54.25, 54.192, 54.104, 53.474, 54.458, 53.652, 54.764, 54.538, 54.268, 54.838, 54.24, 54.708, 54.104, 54.372, 53.88, 53.86, 54.85, 54.76, 54.828, 53.902, 53.718, 53.55, 54.502, 54.596, 54.424, 54.456, 54.642, 54.632, 55.012, 54.494, 54.7, 54.044, 53.77, 54.002, 54.708, 54.08, 54.258, 53.834, 54.642, 54.654, 54.238, 53.742, 54.06, 53.792, 54.136, 54.376, 54.628, 54.164, 54.62, 53.868, 54.078, 54.712, 54.532, 53.998, 53.946, 54.682, 54.346, 54.002, 54.406, 54.62, 54.348, 54.072, 54.65, 54.432, 54.562, 55.122, 54.902, 54.316, 54.272, 54.226, 54.304, 54.164, 53.756, 53.706, 54.38, 54.422, 54.99, 53.926, 55.214, 54.226, 54.966, 54.582, 54.454, 53.716, 54.142, 54.552, 54.274, 54.192, 54.404, 54.424, 54.434, 53.254, 54.624, 54.208, 54.298, 54.118, 54.266, 54.14, 54.792, 53.972, 54.194, 53.776, 54.534, 54.092, 54.616, 54.762, 54.57, 53.704, 54.484, 53.828, 54.162, 54.596, 54.22, 53.978, 54.094, 54.788, 53.798, 54.98, 54.694, 54.23, 53.964, 54.23, 55.036, 53.886, 54.55, 53.912, 54.238, 54.548, 54.458, 54.516, 54.898, 54.644, 54.286, 54.32, 53.94, 55.36, 53.706, 54.206, 54.382, 53.598, 55.254, 54.65, 54.256, 55.22, 54.562, 53.918, 54.484, 54.762, 54.598, 54.956, 54.492, 54.17, 54.024, 54.39, 54.218, 53.944, 54.726, 54.61, 54.884, 54.47, 54.39, 53.87, 54.672, 54.778, 55.142, 55.902, 54.98, 54.382, 53.814, 54.228, 55.34, 55.434, 53.844, 54.584, 54.422, 54.438, 54.462, 53.34, 54.378, 53.162, 54.196, 54.286, 55.116, 54.51, 55.532, 55.376, 53.396, 53.364, 53.754, 53.772, 54.478, 54.356, 54.024, 54.33, 54.446, 53.672, 54.044, 54.676, 53.994, 53.846, 54.598, 53.858, 54.088, 53.658, 53.896, 54.138, 54.706, 53.778, 53.388, 54.142, 53.478, 55.036, 54.746, 54.28, 53.852, 55.466, 53.63, 54.076, 54.274, 54.756, 54.194, 53.966, 53.862, 54.422, 54.422, 53.576, 53.776, 54.888, 53.936, 53.79, 54.56, 54.144, 55.04, 55.042, 54.068, 54.038, 54.572, 53.934, 53.85, 54.188, 54.026, 53.888, 54.24, 54.284, 54.25, 54.628, 53.868, 53.818, 55.006, 54.098, 54.584, 54.462, 54.362, 53.942, 54.664, 53.766, 54.45, 54.256, 53.998, 54.314, 53.952, 54.112, 53.676, 54.698, 54.008, 53.74, 54.792, 53.558, 54.674, 54.372, 54.47, 54.506, 53.702, 54.31, 54.034, 55.04, 53.886, 54.322, 53.716, 53.938, 54.37, 54.304, 55.218, 54.24, 54.556, 54.512, 54.53, 54.112, 54.318, 54.766, 54.184, 53.912, 54.372, 53.758, 55.232, 55.056, 54.22, 54.634, 54.18, 54.148, 54.47, 54.488, 55.186, 54.582, 55.144, 54.424, 54.006, 54.446, 54.354, 54.632, 54.362, 54.556, 54.552, 53.99, 54.972, 54.138, 54.308, 53.69, 54.41, 54.79, 54.46, 54.638, 53.972, 54.192, 53.856, 54.596, 54.102, 53.662, 54.222, 54.33, 54.224, 54.344, 54.362, 55.358, 54.366, 54.648, 54.488, 53.534, 54.93, 54.018, 54.32, 54.896, 54.366, 54.804, 53.996, 54.862, 54.866, 55.318, 54.262, 54.104, 54.79, 54.64, 54.84, 54.784, 53.966, 54.636, 54.472, 54.71, 53.212, 53.776, 54.448, 54.366, 54.194, 54.186, 53.582, 54.302, 53.736, 54.136, 53.878, 54.642, 54.802, 54.454, 54.1, 53.652, 54.172, 54.476, 54.218, 54.342, 53.672, 54.048, 55.008, 54.036, 53.77, 55.11, 54.21, 53.588, 54.586, 54.066, 54.392, 54.726, 54.392, 54.968, 53.646, 53.964, 53.902, 53.942, 54.612, 54.124, 53.928, 54.386, 54.016, 54.026, 54.652, 54.554, 54.386, 55.078, 54.464, 54.754, 54.206, 53.91, 54.204, 53.37, 54.66, 54.234, 54.248, 54.066, 54.544, 54.516, 54.738, 53.992, 54.934, 53.93, 54.29, 54.02, 54.474, 53.898, 53.128, 53.856, 54.198, 53.838, 54.586, 54.088, 53.996, 53.762, 53.972, 54.458, 54.306, 54.484, 53.964, 54.684, 54.236, 54.074, 53.988, 54.222, 54.244, 54.526, 54.972, 54.394, 54.49, 53.874, 54.376, 53.998, 54.778, 54.302, 53.428, 53.81, 54.052, 54.872, 54.046, 53.348, 54.254, 54.702, 54.352, 53.436]

|  |
| --- |
|  Note |
| *The sampling distribution and bootstrap distribution are closely related, and so is the code to generate them.* |

### 4.5 Exercise 4.2.2

#### Compare sampling and bootstrap means

To make calculation easier, distributions similar to those calculated from the previous exercise have been included, this time using a sample size of 5000.

spotify\_population, spotify\_sample, sampling\_distribution, and bootstrap\_distribution are available; pandas and numpy are loaded with their usual aliases.

#### Instructions

1. Calculate the mean popularity in 4 ways:
* Population: from spotify, take the mean of popularity.
* Sample: from spotify\_sample, take the mean of popularity.
* Sampling distribution: from sampling\_distribution, take its mean.
* Bootstrap distribution: from `bootstrap\_distribution, take its mean.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course array
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

spotify\_sample = spotify.sample(n=500)

mean\_popularity\_2000\_samp = []

# Generate a sampling distribution of 2000 replicates
for i in range(2000):
 mean\_popularity\_2000\_samp.append(
 # Sample 500 rows and calculate the mean popularity
 spotify.sample(n=500)['popularity'].mean()
 )

# The sampling distribution results
sampling\_distribution = mean\_popularity\_2000\_samp

mean\_popularity\_2000\_boot = []

# Generate a bootstrap distribution of 2000 replicates
for i in range(2000):
 mean\_popularity\_2000\_boot.append(
 # Resample 500 rows and calculate the mean popularity
 np.mean(spotify\_sample.sample(frac=1, replace=True)['popularity'])
 )

# The bootstrap distribution results
bootstrap\_distribution = mean\_popularity\_2000\_boot

# Calculate the population mean popularity
pop\_mean = spotify['popularity'].mean()

# Calculate the original sample mean popularity
samp\_mean = spotify\_sample['popularity'].mean()

# Calculate the sampling dist'n estimate of mean popularity
samp\_distn\_mean = np.mean(sampling\_distribution)

# Calculate the bootstrap dist'n estimate of mean popularity
boot\_distn\_mean = np.mean(bootstrap\_distribution)

# Print the means
print([pop\_mean, samp\_mean, samp\_distn\_mean, boot\_distn\_mean])

[54.837142308430955, 54.972, 54.81951, 54.97457400000001]

|  |
| --- |
|  Note |
| *The sampling distribution mean can be used to estimate the population mean, but that is not the case with the bootstrap distribution.* |

### 4.6 Exercise 4.2.3

#### Compare sampling and bootstrap standard deviations

In the same way that you looked at how the sampling distribution and bootstrap distribution could be used to estimate the population mean, you’ll now take a look at how they can be used to estimate variation, or more specifically, the standard deviation, in the population.

Recall that the sample size is 5000.

#### Instructions

Calculate the standard deviation of popularity in 4 ways. - Population: from spotify, take the standard deviation of popularity. - Original sample: from spotify\_sample, take the standard deviation of popularity. - Sampling distribution: from sampling\_distribution, take its standard deviation and multiply by the square root of the sample size (5000). - Bootstrap distribution: from bootstrap\_distribution, take its standard deviation and multiply by the square root of the sample size.

# Importing libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import random

# Importing the course array
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

spotify\_sample = spotify.sample(n=5000, random\_state=2022)

mean\_popularity\_2000\_samp = []

# Generate a sampling distribution of 2000 replicates
for i in range(2000):
 mean\_popularity\_2000\_samp.append(
 # Sample 500 rows and calculate the mean popularity
 spotify.sample(n=5000)['popularity'].mean()
 )

# The sampling distribution results
sampling\_distribution = mean\_popularity\_2000\_samp

mean\_popularity\_2000\_boot = []

# Generate a bootstrap distribution of 2000 replicates
for i in range(2000):
 mean\_popularity\_2000\_boot.append(
 # Resample 500 rows and calculate the mean popularity
 np.mean(spotify\_sample.sample(frac=1, replace=True)['popularity'])
 )

# The bootstrap distribution results
bootstrap\_distribution = mean\_popularity\_2000\_boot

# Calculate the population std dev popularity
pop\_sd = spotify['popularity'].std(ddof=0)

# Calculate the original sample std dev popularity
samp\_sd = spotify\_sample['popularity'].std(ddof=1)

# Calculate the sampling dist'n estimate of std dev popularity
samp\_distn\_sd = np.std(sampling\_distribution, ddof=1) \* np.sqrt(5000)

# Calculate the bootstrap dist'n estimate of std dev popularity
boot\_distn\_sd = np.std(bootstrap\_distribution, ddof=1) \* np.sqrt(5000)

# Print the standard deviations
print([pop\_sd, samp\_sd, samp\_distn\_sd, boot\_distn\_sd])

[10.880065274257536, 10.975581356685552, 10.058796581128894, 10.764135231982]

### 4.7 Chapter 4.3: Confidence intervals

In the last few exercises, you looked at relationships between the sampling distribution and the bootstrap distribution.

One way to quantify these distributions is the idea of “values within one standard deviation of the mean”, which gives a good sense of where most of the values in a distribution lie. In this final lesson, we’ll formalize the idea of values close to a statistic by defining the term “confidence interval”.

#### Predicting the weather

Consider meteorologists predicting weather in one of the world’s most unpredictable regions - the northern Great Plains of the US and Canada. Rapid City, South Dakota was ranked as the least predictable of the 120 US cities with a National Weather Service forecast office. Suppose we’ve taken a job as a meteorologist at a news station in Rapid City. Our job is to predict tomorrow’s high temperature.

#### Our weather prediction

We analyze the weather data using the best forecasting tools available to us and predict a high temperature of 47 degrees Fahrenheit. In this case, 47 degrees is our point estimate. Since the weather is variable, and many South Dakotans will plan their day tomorrow based on our forecast, we’d instead like to present a range of plausible values for the high temperature. On our weather show, we report that the high temperature will be between forty and fifty-four degrees tomorrow.

#### We just reported a confidence interval!

This prediction of forty to fifty-four degrees can be thought of as a confidence interval for the unknown quantity of tomorrow’s high temperature. Although we can’t be sure of the exact temperature, we are confident that it will be in that range. These results are often written as the point estimate followed by the confidence interval’s lower and upper bounds in parentheses or square brackets. When the confidence interval is symmetric around the point estimate, we can represent it as the point estimate plus or minus the margin of error, in this case, seven degrees.

#### Bootstrap distribution of mean flavor

Here’s the bootstrap distribution of the mean flavor from the coffee dataset.

#### Mean of the resamples

We can calculate the mean of these resampled mean flavors.

#### Mean plus or minus one standard deviation

If we create a confidence interval by adding and subtracting one standard deviation from the mean, we see that there are lots of values in the bootstrap distribution outside of this one standard deviation confidence interval.

#### Quantile method for confidence intervals

If we want to include ninety-five percent of the values in the confidence interval, we can use quantiles. Recall that quantiles split distributions into sections containing a particular proportion of the total data. To get the middle ninety-five percent of values, we go from the point-zero-two-five quantile to the point-nine-seven-five quantile since the difference between those two numbers is point-nine-five. To calculate the lower and upper bounds for this confidence interval, we call quantile from NumPy, passing the distribution values and the quantile values to use. The confidence interval is from around seven-point-four-eight to seven-point-five-four.

#### Inverse cumulative distribution function

There is a second method to calculate confidence intervals. To understand it, we need to be familiar with the normal distribution’s inverse cumulative distribution function. The bell curve we’ve seen before is the probability density function or PDF. Using calculus, if we integrate this, we get the cumulative distribution function or CDF. If we flip the x and y axes, we get the inverse CDF. We can use scipy.stats and call norm.ppf to get the inverse CDF. It takes a quantile between zero and one and returns the values of the normal distribution for that quantile. The parameters of loc and scale are set to 0 and 1 by default, corresponding to the standard normal distribution. Notice that the values corresponding to point-zero-two-five and point-nine-seven-five are about minus and plus two for the standard normal distribution.

#### Standard error method for confidence interval

This second method for calculating a confidence interval is called the standard error method. First, we calculate the point estimate, which is the mean of the bootstrap distribution, and the standard error, which is estimated by the standard deviation of the bootstrap distribution. Then we call norm.ppf to get the inverse CDF of the normal distribution with the same mean and standard deviation as the bootstrap distribution. Again, the confidence interval is from seven-point-four-eight to seven-point-five-four, though the numbers differ slightly from last time since our bootstrap distribution isn’t perfectly normal.

### 4.8 Exercise 4.3.1

#### 4.8.1 Calculating confidence intervals

You have learned about two methods for calculating confidence intervals: *the quantile method* and *the standard error method*. The standard error method involves using the inverse cumulative distribution function (inverse CDF) of the normal distribution to calculate confidence intervals. In this exercise, you’ll perform these two methods on the Spotify data.

#### 4.8.2 Instructions

1. Generate a 95% confidence interval using the quantile method on the bootstrap distribution, setting the 0.025 quantile as lower\_quant and the 0.975 quantile as upper\_quant.
2. Generate a 95% confidence interval using the standard error method from the bootstrap distribution.
* Calculate point\_estimate as the mean of bootstrap\_distribution, and standard\_error as the standard deviation of bootstrap\_distribution.
* Calculate lower\_se as the 0.025 quantile of an inv. CDF from a normal distribution with mean point\_estimate and standard deviation standard\_error.
* Calculate upper\_se as the 0.975 quantile of that same inv. CDF.

# Importing libraries
import pandas as pd
import numpy as np
from scipy.stats import norm

# Importing the course array
spotify = pd.read\_feather("datasets/spotify\_2000\_2020.feather")

spotify\_sample = spotify.sample(n=5000, random\_state=2022)

mean\_popularity\_2000\_boot = []

# Generate a bootstrap distribution of 2000 replicates
for i in range(2000):
 mean\_popularity\_2000\_boot.append(
 # Resample 500 rows and calculate the mean popularity
 np.mean(spotify\_sample.sample(frac=1, replace=True)['popularity'])
 )

# The bootstrap distribution results
bootstrap\_distribution = mean\_popularity\_2000\_boot

# Generate a 95% confidence interval using the quantile method
lower\_quant = np.quantile(bootstrap\_distribution, 0.025)
upper\_quant = np.quantile(bootstrap\_distribution, 0.975)

# Print quantile method confidence interval
print((lower\_quant, upper\_quant))

# Find the mean and std dev of the bootstrap distribution
point\_estimate = np.mean(bootstrap\_distribution)
standard\_error = np.std(bootstrap\_distribution, ddof=1)

# Find the lower limit of the confidence interval
lower\_se = norm.ppf(0.025, loc=point\_estimate, scale=standard\_error)

# Find the upper limit of the confidence interval
upper\_se = norm.ppf(0.975, loc=point\_estimate, scale=standard\_error)

# Print standard error method confidence interval
print((lower\_se, upper\_se))

(54.480395, 55.097075000000004)
(54.471827614788566, 55.08400298521144)

## 5 Reference

Sampling in Python in Intermediate Python Course for Associate Data Scientist in Python Carrer Track in DataCamp Inc by James Chapman.