Module 9

Date	@01/09/2023
Title	Sampling in Python

Comparing point estimates

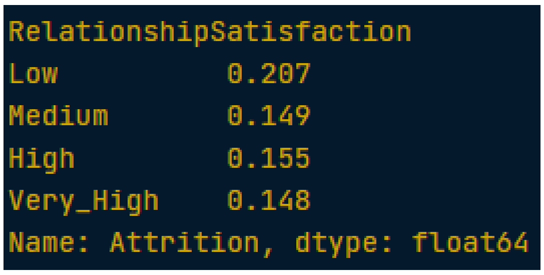
Now that you have three types of sample (simple, stratified, and cluster), you can compare point estimates from each sample to the population parameter. That is, you can calculate the same summary statistic on each sample and see how it compares to the summary statistic for the population.

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`.

`attrition_pop`, `attrition_srs`, `attrition_strat`, and `attrition_clust` are available; `pandas` is loaded with its usual alias.

- Group `attrition_pop` by `RelationshipSatisfaction` levels and calculate the mean of `Attrition` for each level.

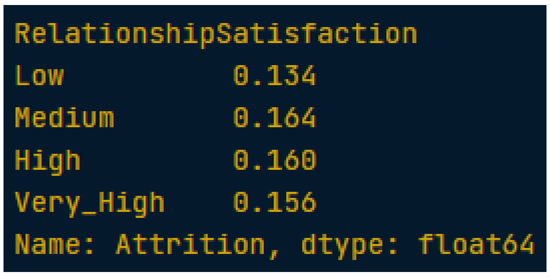
```
# Mean Attrition by RelationshipSatisfaction group
mean_attrition_pop = attrition_pop.groupby('RelationshipSatis

# Print the result
print(mean_attrition_pop)
```



```
RelationshipSatisfaction
Low           0.207
Medium        0.149
High          0.155
Very_High     0.148
Name: Attrition, dtype: float64
```

- Calculate the proportion of employee attrition for each relationship satisfaction group, this time on the simple random sample, `attrition_srs`.

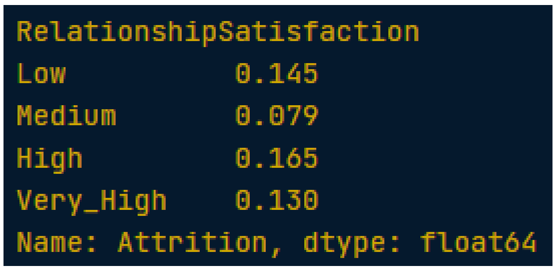
```
# Calculate the same thing for the simple random sample
mean_attrition_srs = attrition_srs.groupby('RelationshipSatis

# Print the result
print(mean_attrition_srs)
```



```
RelationshipSatisfaction
Low          0.134
Medium       0.164
High         0.160
Very_High    0.156
Name: Attrition, dtype: float64
```

- Calculate the proportion of employee attrition for each relationship satisfaction group, this time on the stratified sample, `attrition_strat`.

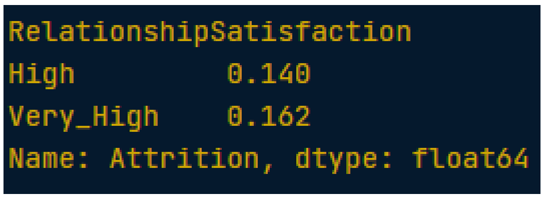
```
# Calculate the same thing for the stratified sample
mean_attrition_strat = attrition_strat.groupby('RelationshipS

# Print the result
print(mean_attrition_strat)
```



```
RelationshipSatisfaction
Low          0.145
Medium       0.079
High         0.165
Very_High    0.130
Name: Attrition, dtype: float64
```

- Calculate the proportion of employee attrition for each relationship satisfaction group, this time on the cluster sample, `attrition_clust`.

```
# Calculate the same thing for the cluster sample
mean_attrition_clust = attrition_clust.groupby('RelationshipS
# Print the result
print(mean_attrition_clust)
```



```
RelationshipSatisfaction
High          0.140
Very_High     0.162
Name: Attrition, dtype: float64
```

🐸 As you increase the sample size, the relative error decreases quickly at first, then more slowly as it drops to zero.

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.

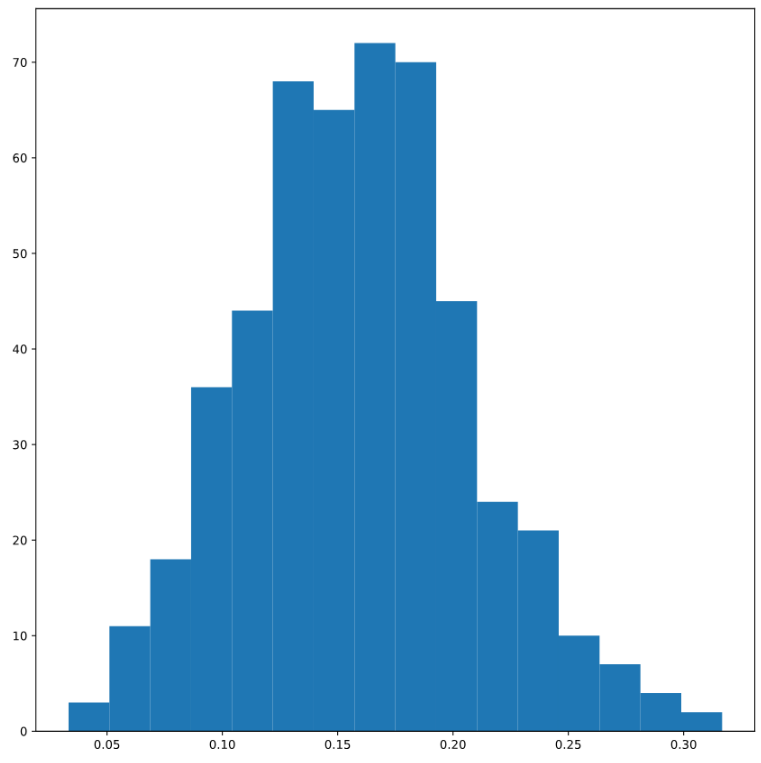
- Replicate the provided code so that it runs `500` times. Assign the resulting list of sample means to `mean_attritions`.

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

# Print out the first few entries of the list
print(mean_attritions[0:5])
```

```
[0.23333333333333334, 0.13333333333333333, 0.18333333333333332, 0.16666666666666666, 0.1]
```

- Draw a histogram of the `mean_attritions` list with 16 bins.

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

# Create a histogram of the 500 sample means
plt.hist(mean_attritions, bins=16)
plt.show()
```

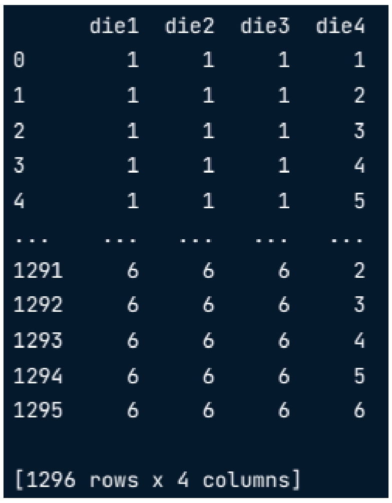


🐸 As sample size increases, on average each sample mean has a lower relative error compared to the population mean, thus reducing the range of the distribution.

Approximate sampling distributions

4 dice

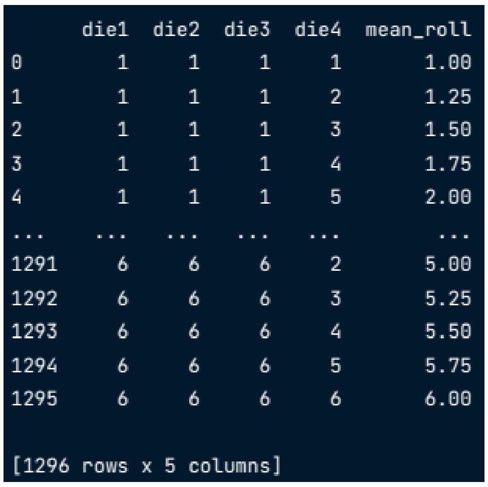
```
dice = expand_grid(
 {'die1': [1, 2, 3, 4, 5, 6],
  'die2': [1, 2, 3, 4, 5, 6],
  'die3': [1, 2, 3, 4, 5, 6],
  'die4': [1, 2, 3, 4, 5, 6]
 }
)
```



```
      die1  die2  die3  die4
0        1     1     1     1
1        1     1     1     2
2        1     1     1     3
3        1     1     1     4
4        1     1     1     5
...    ...   ...   ...   ...
1291     6     6     6     2
1292     6     6     6     3
1293     6     6     6     4
1294     6     6     6     5
1295     6     6     6     6

[1296 rows x 4 columns]
```

Mean roll

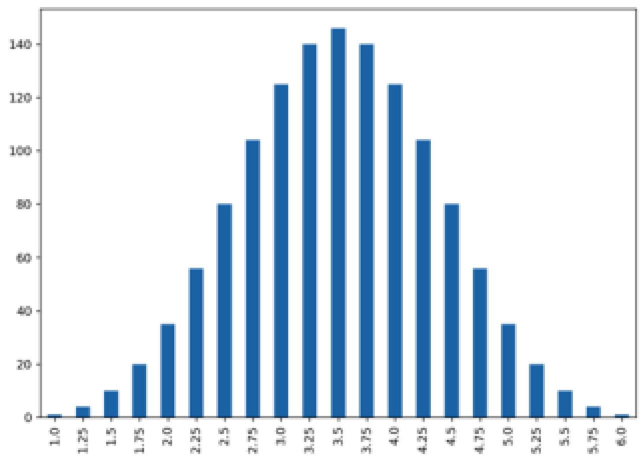
```
dice['mean_roll'] = (dice['die1'] +
                                         dice['die2'] +
                                         dice['die3'] +
                                         dice['die4']) / 4
print(dice)
```



```
      die1  die2  die3  die4  mean_roll
0        1     1     1     1       1.00
1        1     1     1     2       1.25
2        1     1     1     3       1.50
3        1     1     1     4       1.75
4        1     1     1     5       2.00
...    ...   ...   ...   ...        ...
1291     6     6     6     2       5.00
1292     6     6     6     3       5.25
1293     6     6     6     4       5.50
1294     6     6     6     5       5.75
1295     6     6     6     6       6.00

[1296 rows x 5 columns]
```

Exact sampling distribution

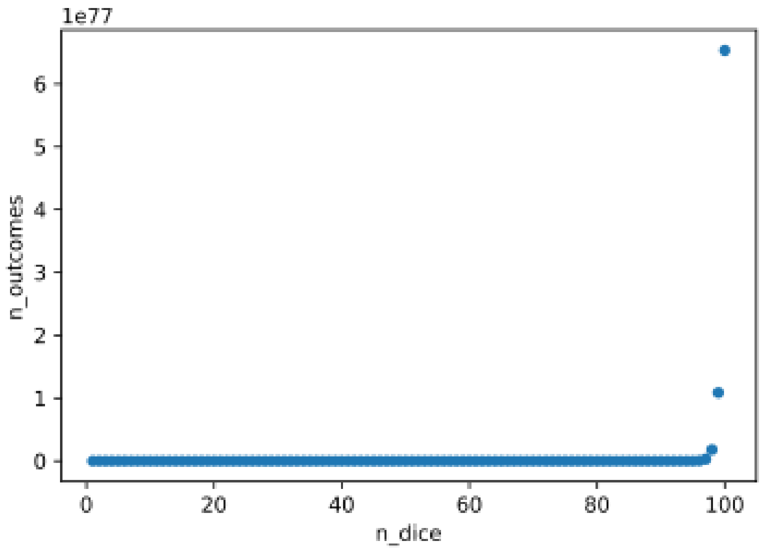
```
dice['mean_roll'] = dice['mean_roll'].astype('category')
dice['mean_roll'].value_counts(sort=False).plot(kind="bar")
```



The number of outcomes increases fast

```
n_dice = list(range(1, 101))
n_outcomes = []
for n in n_dice:
        n_outcomes.append(6**n)

outcomes = pd.DataFrame(
{"n_dice": n_dice,
"n_outcomes": n_outcomes})
```

```
outcomes.plot(x="n_dice", y="n_outcomes", kind="scatter")
plt.show()
```



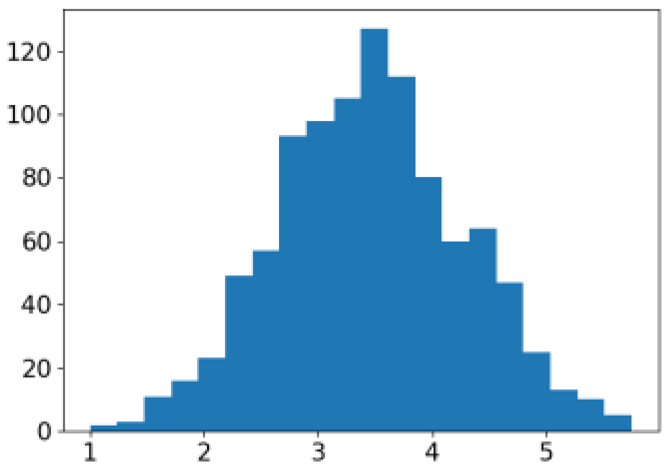
Simulating the mean of four dice rolls

```
import numpy as np
sample_means_1000 = []
for i in range(1000):
    sample_means_1000.append(
        np.random.choice(list(range(1, 7)), size=4, replace=T

print(sample_means_1000)
```

```
[3.25, 3.25, 1.75, 2.0, 2.0, 1.0, 1.0, 2.75, 2.75, 2.5, 3.0, 2.0, 2.75,
...
1.25, 2.0, 2.5, 2.5, 3.75, 1.5, 1.75, 2.25, 2.0, 1.5, 3.25, 3.0, 3.5]
```

Approximate sampling distribution

```
plt.hist(sample_means_1000, bins=20)
```

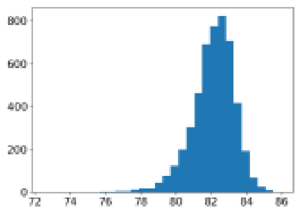


🐸 The exact sampling distribution can only be calculated if you know what the population is and if the problems are small and simple enough to compute. Otherwise, the approximate sampling distribution must be used.

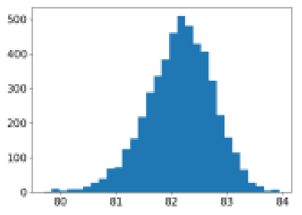
Standard errors and the Central Limit Theorem

Sampling distribution of mean cup points

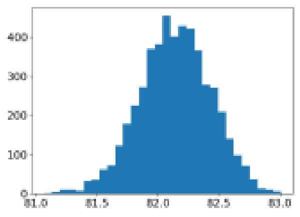
Sample size: 5



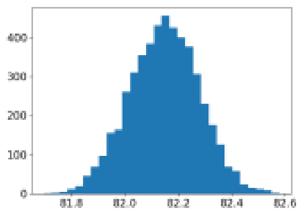
Sample size: 20



Sample size: 80



Sample size: 320



Consequences of the central limit theorem

🐸 Averages of independent samples have approximately normal distributions

As the sample size increases,

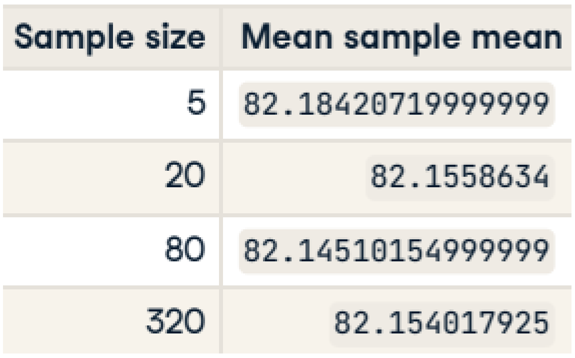
- The distribution of the averages gets closer to being normally distributed
- The width of the sampling distribution gets narrower

Population & sampling distribution means

```
coffee_ratings['total_cup_points'].mean()
```

> 82.15120328849028

Use np.mean() on each approximate sampling distribution:



Sample size	Mean sample mean
5	82.18420719999999
20	82.1558634
80	82.14510154999999
320	82.154017925

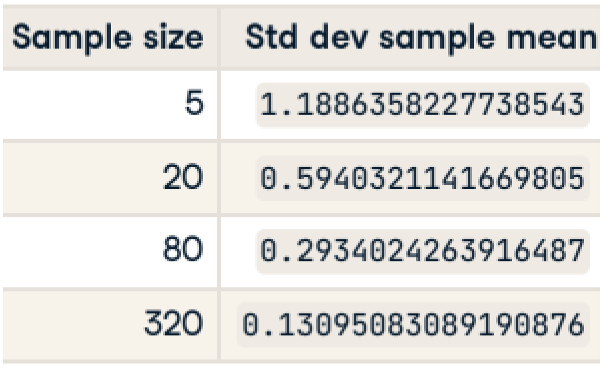
Population & sampling distribution standard deviations

```
coffee_ratings['total_cup_points'].std(ddof=0)
```

> 2.685858187306438

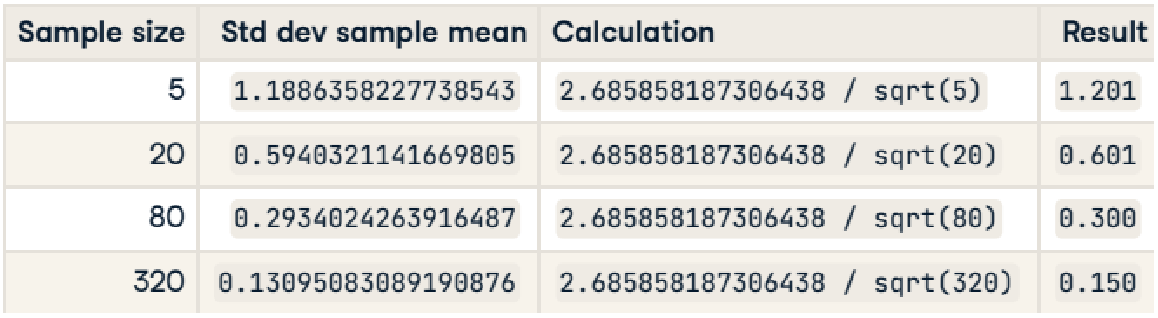
🐸 Specify ddof=0 when calling .std() on populations

🐸 Specify ddof=1 when calling np.std() on samples or sampling distributions



Sample size	Std dev sample mean
5	`1.1886358227738543`
20	`0.5940321141669805`
80	`0.2934024263916487`
320	`0.13095083089190876`

Population mean over square root sample size



Sample size	Std dev sample mean	Calculation	Result
5	`1.1886358227738543`	`2.685858187306438 / sqrt(5)`	`1.201`
20	`0.5940321141669805`	`2.685858187306438 / sqrt(20)`	`0.601`
80	`0.2934024263916487`	`2.685858187306438 / sqrt(80)`	`0.300`
320	`0.13095083089190876`	`2.685858187306438 / sqrt(320)`	`0.150`

Standard error

- Standard deviation of the sampling distribution
- Important tool in understanding sampling variability

🐸 If the sample is not closely representative of the population, then the mean of the bootstrap distribution will not be representative of the population mean. This is less of a problem for standard errors.

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`.

Calculate the mean `popularity` in 4 ways:

- Population: from `spotify_population`, 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.

```
# Calculate the population mean popularity
pop_mean = spotify_population['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, 54.8686, 54.83586444000001, 54.86607339999999]
```

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

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.

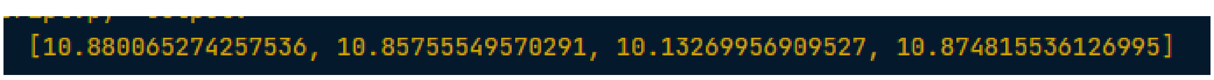
Recall that the sample size is `5000`.

Calculate the standard deviation of `popularity` in 4 ways.

- Population: from `spotify_population`, 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.

```
# Calculate the population std dev popularity
pop_sd = spotify_population['popularity'].std(ddof=0)

# Calculate the original sample std dev popularity
samp_sd = spotify_sample['popularity'].std()

# Calculate the sampling dist'n estimate of std dev popularit
samp_distn_sd = np.std(sampling_distribution, ddof=1) * np.sq

# Calculate the bootstrap dist'n estimate of std dev populari
boot_distn_sd = np.std(bootstrap_distribution, ddof=1) * np.s
```

```
# Print the standard deviations
print([pop_sd, samp_sd, samp_distn_sd, boot_distn_sd])
```



```
[10.880065274257536, 10.85755549570291, 10.13269956909527, 10.874815536126995]
```

This is an important property of the bootstrap distribution. When you don't have all the values from the population or the ability to sample multiple times, you can use bootstrapping to get a good estimate of the population standard deviation.

Confidence intervals

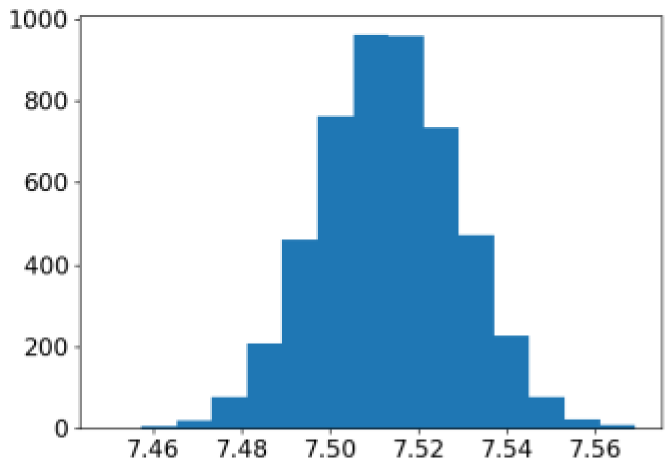
- "Values within one standard deviation of the mean" includes a large number of values from
  each of these distributions
- We'll define a related concept called a confidence interval

Predicting the weather

- Rapid City, South Dakota in the United States has the least predictable weather
- Our job is to predict the high temperature there tomorrow

Bootstrap distribution of mean flavor

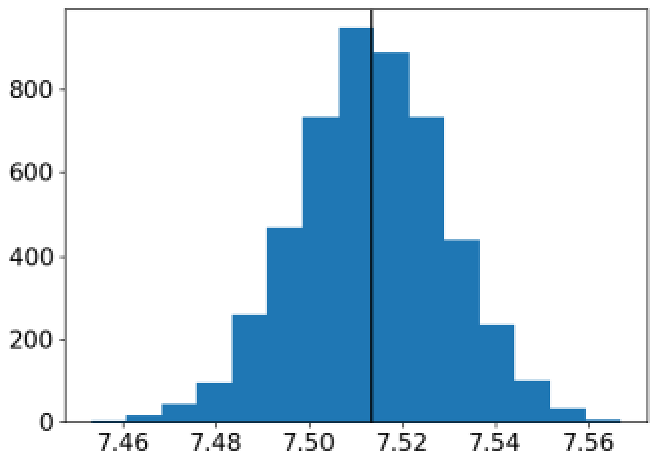
```
import matplotlib.pyplot as plt
plt.hist(coffee_boot_distn, bins=15)
plt.show()
```



Mean of the resamples

```
import numpy as np
np.mean(coffee_boot_distn)
```

> 7.513452892



Mean plus or minus one standard deviation

```
np.mean(coffee_boot_distn)
```

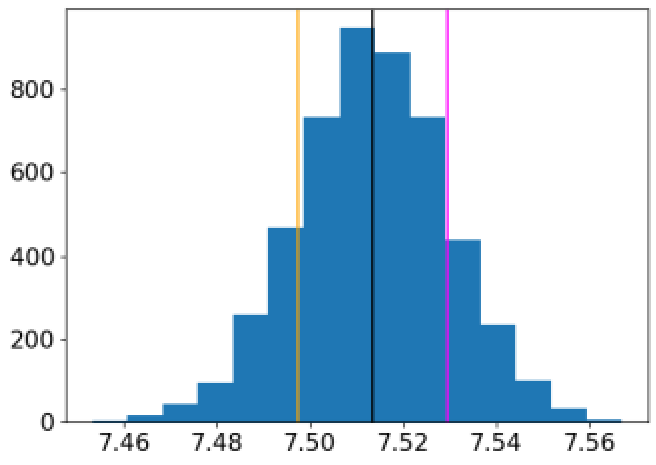
> 7.513452892

```
np.mean(coffee_boot_distn) - np.std(coffee_boot_distn, ddof=1
```

> 7.497385709174466

```
np.mean(coffee_boot_distn) + np.std(coffee_boot_distn, ddof=1
```

> 7.529520074825534



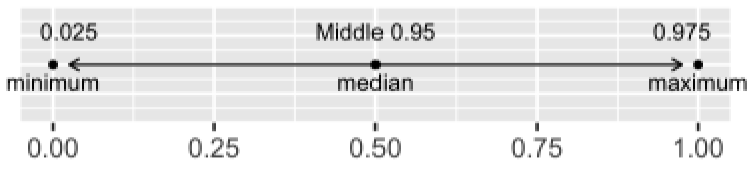
Quantile method for confidence intervals

```
np.quantile(coffee_boot_distn, 0.025)
```

> 7.4817195

```
np.quantile(coffee_boot_distn, 0.975)
```

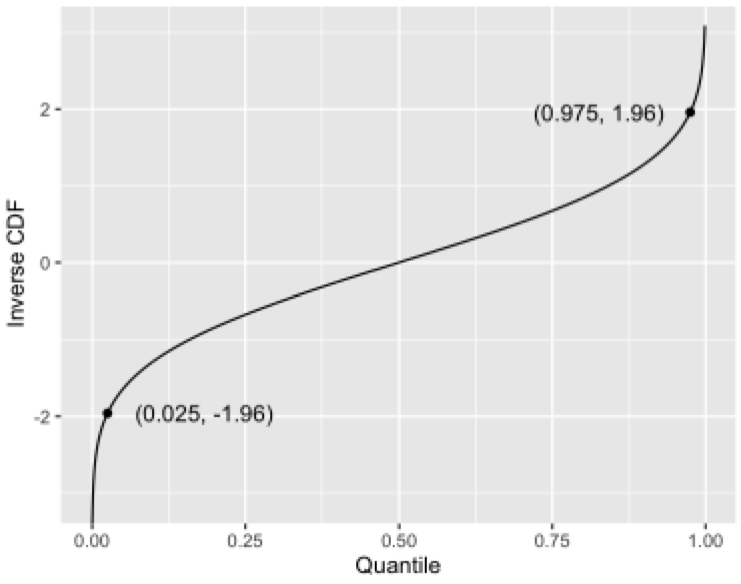
> 7.5448805



Inverse cumulative distribution function

PDF: The bell curve
CDF: integrate to get area under bell curve
Inv. CDF: flip x and y axes
Implemented in Python with

```
from scipy.stats import norm
norm.ppf(quantile, loc=0, scale=1)
```



Standard error method for confidence interval

```
point_estimate = np.mean(coffee_boot_distn)
```

> 7.513452892

```
std_error = np.std(coffee_boot_distn, ddof=1)
```

```
0.016067182825533724
```

```
from scipy.stats import norm
lower = norm.ppf(0.025, loc=point_estimate, scale=std_error)
upper = norm.ppf(0.975, loc=point_estimate, scale=std_error)
print((lower, upper))
```

```
(7.481961792328933, 7.544943991671067)
```

🐸 Confidence intervals account for uncertainty in our estimate of a population parameter by providing a range of possible values. We are confident that the true value lies somewhere in the interval specified by that range.