Sampling in Python
by datacamp

population v sample
population is the complete dataset
*tricksy thing - most of the time in statistics we won't know what the whole population is like
sample is the subset of data that we are working with

Example - let's consider the relationship between cup points and flavor in an evaluation of coffee
pull our entire population whcih in this case is 1,338 coffee ratings
pts_vs_flavor_pop = coffee_ratings[['total_cup_point', 'flavor']]
lets pull a sample
pts_vs_flavor_samp = pts_vs_flavor_pop.sample(n=10)
set n to our desired sample size
sample method in pandas package
sample method returns a random subset of rows
**by default rows cannot appear multiple times, so we are guaranteed to have ten unique rows in our sample

Can also use the sample method on pandas Series by using subsetting
cup_poins_samp = coffee_ratings['total_cup_points'].sample(n=10)

Population parameter
is a calculation made on the population dataset
import numpy as np
np.mean(pts_vs_flavor_pop['total_cup_points'])
with pandas
pts_vs_flavor_pop['flavor'].mean()

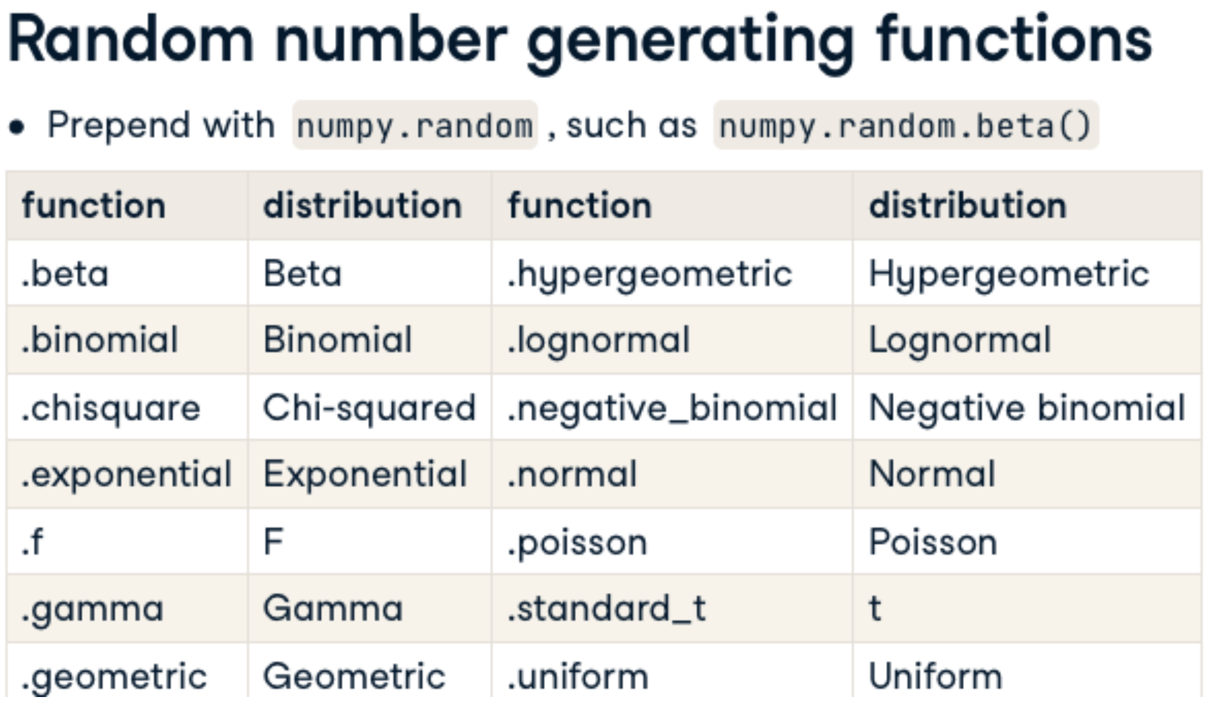
For a sample these parameters are called:
sample statistic or point estimate
np.mean(cup_points_samp)
pts_vs_flavor_samp['flavor'].mean()

Convenience sampling
collecting data by the easiest method
often prone to sample bias
sample bias
sample is not representative of population

**randomness is one way to help avoid this

Visualizing selection bias
coffee_ratings['total_cup_points'].hist(bins=np.arange(59, 93, 2))
plt.show()
#we looked at the dataset and subjectively chose a low rating of 59 and a high of 91
#the 2 creates bins of width 2

Pseudo-random number generation
randomness really mean
If we want to choose data points at random from a population, we shoudn't be able to predict which data points would be selected ahead of time in some systematic way.
we use pseudo-random number generation because it is cheap and fast
next 'random' number is actually calculated from previous 'random' number
the first 'random' number is calculated from a 'seed' value
**same seed value yields the same random numbers



Random number generating functions

- Prepend with `numpy.random`, such as `numpy.random.beta()`

function	distribution	function	distribution
.beta	Beta	.hypergeometric	Hypergeometric
.binomial	Binomial	.lognormal	Lognormal
.chisquare	Chi-squared	.negative_binomial	Negative binomial
.exponential	Exponential	.normal	Normal
.f	F	.poisson	Poisson
.gamma	Gamma	.standard_t	t
.geometric	Geometric	.uniform	Uniform

Visualizing random numbers
randoms = np.random.beta(a=2, b=2, size=5000)
plt.hist(randoms, bins=np.arange(0,1, 0.05)
plt.show()
#a,b arguments to the beta function specify distribution parameters

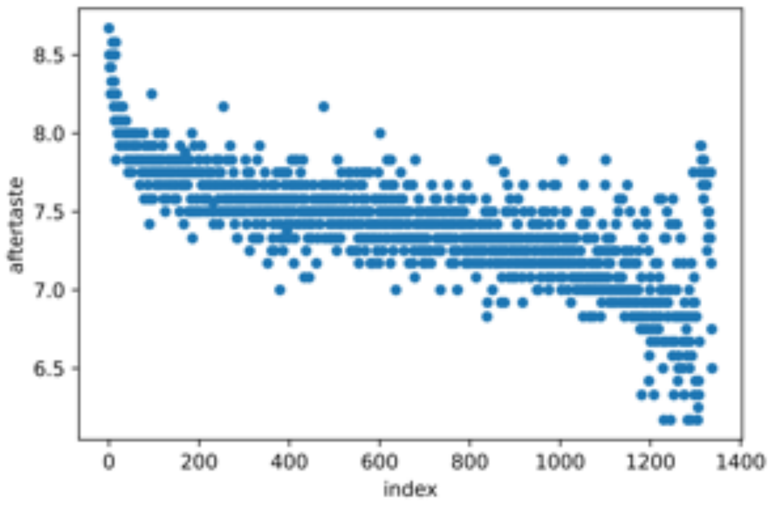
Setting a random seed
np.random.seed(any integer)

another example of random number generator function
np.random.normal(loc=2, scale=1.5, size=2)
this one generates pseudo-random numbers from the normal distribution
loc and scale arguments set the mean and standard deviation of the distribution

Simple random and systematic sampling
simple random sampling is like a raffle or lottery
also called SRS sometimes
coffee_ratings.sample(n=t, random_state=190000113)
random_state allows us to set a seed within the sample request

systematic sampling
samples the population at regular intervals
example would be taking every fifth sample
harder but still possible to do with pandas
example - sampling 5 coffees from our coffee set
sample_size = 5
pop_size = len(coffee_ratings)
print(pop_size)
output > 1338
interval = pop_size // sample_size
#// represents integer division, like standard division but discards any fractional part
interval = 1338 // 5 = 267 instead of 267.6
to select every 267th coffee
coffee_ratings.iloc[::interval]
pass :: to tell panda to select every 267th coffee till the end of the DF

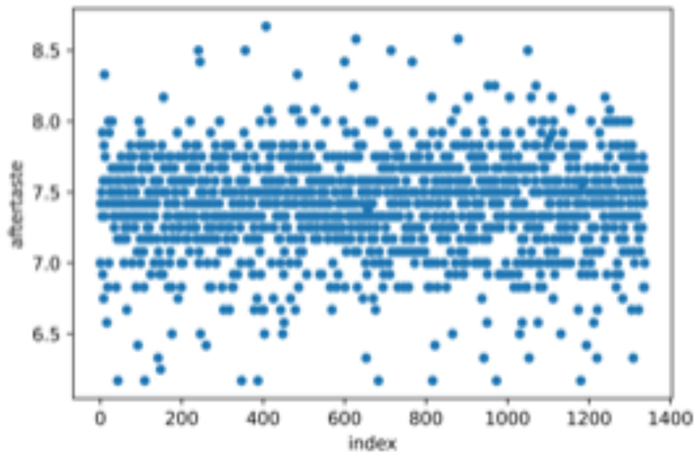
The trouble with systematic sampling
can accidentally introduce bias into the statistics that we calculate
important to visualize data prior to attempt to avoid these types of bias
example in the coffee set
coffee_ratings_with_id = coffee_ratings.reset_index()
coffee_ratings_with_id.plot(x=‘index’, y=‘aftertaste’, kind=’scatter’)
plt.show()



The plot clearly shows a bias as the samples go from 0 to 1300
*systematic sampling is only safe if we don't see a pattern in this scatter plot

Making systematic sampling safe
we can randomize the row order before sampling
shuffled = coffee_ratings.sample(frac=1)
frac argument lets us specify the proportion of the dataset to return in the sample rather than the absolute number of rows than n specifies
**setting frac=1, randomly samples the whole dataset

Next, we need the indices to be reset so that they go in order from zero again
dropt argument set to True clears the previous row indexes
then chaining them to another reset_index call creates a column containing these new indexes
shuffled = shuffled.reset_index(drop =True).reset_index()
shuffled.plot(x='index', y='aftertaste', kind='scatter')
plt.show()



***biggest key point is this whole process is essentially the same as simple random sampling

Stratified and weighted random sampling
stratified sampling
a technique that allow us to sample a population that contains subgroups
example - working with coffees via their country
to see how the data is spersed (often useful to know where the most data is)
top_counts = coffee_ratings['country_of_origin'].value_counts()
top_counts.head(6)
then filter for the 6 top countries
top_counted_countries = ['Mexico', 'Columbia', 'Guatemala', 'Brazil', 'Taiwan', 'United States (Hwaii)']
top_counted_subset = coffee_ratings['country_of_origin'].isin(top_counted_countries)
coffee_ratings_top = coffee_ratings[top_counted_subset]
now take a percent of the simple random sample
do this using the frac argument 0.1 = 10%
coffee_ratings_samp = coffee_ratings_top.sample(frac=0.1, random_state=2021)
look at the counts now
coffee_ratings_samp['country_of_origin'].value_counts(normalize=True)
set normalize argument to True to convert the counts into a proportion
this proportion shows what proportion of coffees in the sample came from each country
**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

```
coffee_ratings_strat = coffee_ratings_top.groupby('country_of_origin') \
        .sample(frac=0.1, random_state=2021)
```

*just practicing with the '\', it is there simple to sparse out code to make it more readable
**now the proportions of each country in the stratified sample are much closer to those in the population
*a variation of stratified sampling is to sample equal counts from each group
to do this:
coffee_ratings_eq = coffee_ratings_top.groupby('country_of_origin').sample(n=15, random_state=2021)
the difference is using 'n' instead of 'frac'

Weighted random sampling
specifies weights to adjust the relative probability of a row being sampled
example
say we wanted to have a higher proportion of Taiwanese coffees in the sample than in the population

```
import numpy as np
coffee_ratings_weight = coffee_ratings_top
condtion = coffee_ratings_weight[‘country_of_origin’] == ’Taiwan’
coffee_ratings_weight[‘weight’] = np.where(condition, 2, 1)
```

**where function - here we have set a weight of 2 to 1 when ‘condition’ is met
**here this will weight Taiwanese coffee 2 to 1
ie Taiwanese coffee has two times the chance of being picked compared to the other coffees
now we have to use the ‘weights’ argument in our ‘sample’ method

```
coffee_ratings_weight = coffee_ratings_weight.sample(frac=0.1, weights=‘weight’)
```

**this sort of weighted sampling is common in political polling, where we need to correct for under or over representation of demographic groups

Example

```
# Proportion of employees by Education level
education_counts_pop = attrition_pop['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_pop.groupby('Education')\
    .sample(frac=0.4, random_state=2022)

# 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)
```

Weighted example

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

# Sample 400 employees weighted by YearsAtCompany
attrition_weight = attrition_pop.sample(n=400, weights="YearsAtCompany")

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

Cluster sampling

-use SRS to pick some subgroups
-use SRS on only those subgroups
in contrast stratified sampling
-splits the population into subgroups
-then uses SRS on every subgroup
clustering is often used to cut down on costs
example - coffee dataset
first stage of cluster sampling is to randomly cut down the number of varieties (or unique values)
import random
#create a list
varieties_pop = list(coffee_ratings[‘variety’].unique())
varieties_samp = random.sample(varieties_pop, k=3)
we specify how many unique varieties that we want in this example with the ‘k’ argument
stage 2 - perform SRS on each of the three varieties
#filter dataset for rows where the variety is one of the three selected using the .isin() method
variety_condition = coffee_ratings[‘variety’].isin(varieties_samp)
coffee_ratings_cluster = coffee_ratings[variety_condition]
#then ensure the isin() method removes levels with zero rows, we apply the .cat.remove_unused_categories() on the focal Series
**this is important or otherwise an error may occur when sampling by variety level
coffee_ratings_cluster[‘variety’] =
coffee_ratings_cluster[‘variety’].cat.remove_unused_categories()
then groupby and sample as we’ve done in previous examples
coffee_ratings_cluster.groupby(‘variety’).sample(n=5, random_state=2021)
**we randomly sampled the subgroups to include, then we randomly sampled rows from those subgroups

Example
# Create a list of unique JobRole values
job_roles_pop = list(attrition_pop['JobRole'].unique())

# Randomly sample four JobRole values
job_roles_samp = random.sample(job_roles_pop, k=4)

# Filter for rows where JobRole is in job_roles_samp
jobrole_condition = attrition_pop['JobRole'].isin(job_roles_samp)
***we then apply the ‘jobrole_condition’ as a filter on the ‘attrition_pop’ DF using boolen indexing
***this filters out the rows where the ‘JobRole’ is not in the sampled list
attrition_filtered = attrition_pop[jobrole_condition]

```
# 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)
```

Another example

```
# Create a list of unique RelationshipSatisfaction values
satisfaction_unique = list(attrition_pop['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_pop['RelationshipSatisfaction'].isin(satisfaction_samp)
attrition_clust_prep = attrition_pop[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_pop
attrition_clust =
attrition_clust_prep.groupby('RelationshipSatisfaction').sample(n=367,
random_state=2022)
```

Relative error of point estimates
ie how the size of the sampe affects the accuracy of the point estimates
len function pulls the number of rows in the sample ie the number of observations in the sample
**larger sample sizes usually give us more accurate results

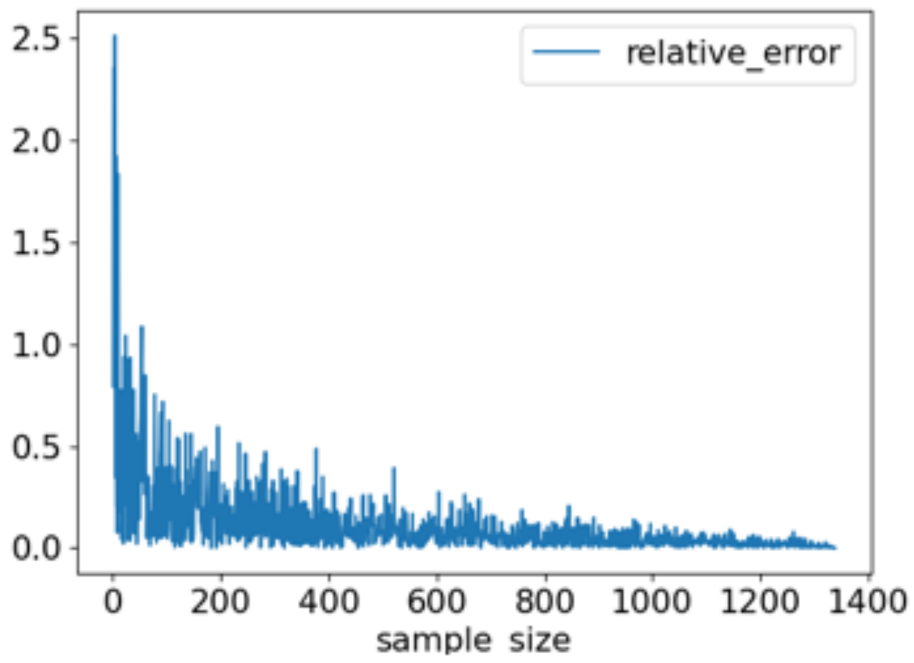
Relative error
is the absolute (ignore minus signs) difference between the population and a sample mean
population parameter:

```
population_mean = coffee_ratings[‘total_cup_points’].mean()
```

point estimate:

sample_mean = coffee_ratings.sample(n=sample_size)[‘total_cup_points’].mean()
relative error as a percentage:
rel_error_pct = 100 * abs(population_mean - sample_mean) / population_mean

Plotting relative error vs sample size
import matplotlib.pylplot as plt
errors.plot(x=‘sample_size’, y=‘relative_error’, kind=‘line’)
plt.show()



really noisy, particularly for small samples
amplitude and noise decrease with more samples
**relative error decreases to zero when the sample size = population

example
# Generate a simple random sample of 100 rows, with seed 2022
attrition_srs100 = attrition_pop.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)

Creating a sampling distribution
using a for loop to run the same code many times
**this example samples 30 random coffees from the coffee_ratings dataset, calculates the mean, then puts the result (called a replicate) into the open list, does this a 1000 times

```
mean_cup_points_1000 = [ ]
for i in range(1000)
        mean_cup_points_1000.append(coffee_ratings.sample(n=30)
[‘total_cup_points’].mean())
print(mean_cup_points_1000)
```

output > a thousand separate sample means

Visualize the distribution
**again, often the best plot to visualize a distribution is a histogram

```
plt.hist(mean_cup_points_1000, bins=30)
plt.show()
```

this example creates a nice bell shape curve (normal distribution)
**a distribution of replicates of point estimates is called a ‘sampling distribution’
decreasing the sample size increases the range of the x-values on the histogram
inverse increasing the sample size decreases the range of the x-values

*increasing the number of replicates didn’t affect the relative error of the sample means but it did resultin a more consistent shape to the distribution

Approximate sampling distributions
**we can generate all possible combinations of rolls using the pandas expand_grid function
uses the itertools package
example with four 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]})
```

this is 6^4 or 1296 dice roll combinations
now consider the ‘mean roll’
add a column to the DataFrame

```
dice[‘mean_roll’] = (dice[‘die1’] + dice[‘die2’] + dice[‘die3’] + dice[‘die4’]) / 4
```

**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
**we are interested in the counts of each value so this is better suited as a data type category

```
dice[‘mean_roll’] = dice[‘mean_roll’].astype(‘category’)
dice[‘mean_roll’].value_counts(sort=False).plot(kind=‘bar’)
```
sort argument to False ensures the x-axis ranges from one to six instead of sorting the bars by frequency
*interesting fact, with just a 100 die we can get a number of possible outcomes equivalent to the number of atoms in the universe
this obviously shows how quickly we can enter the computationally impossible
**what this means is that we have to rely on approximations

Simulating - example simulating the mean of four dice rolls
```
sample_means_ 1000 = [ ]
for i in range(1000):
        sample_means_1000. append(
                        np.random.choice(list(range(1,7)), size=4, replace=True).mean())
```
the choice method allow you to randomly select elements from an array based on specified probabilities
first argument is the sequence or array from where the randome samples are to be drawn from
size argument specifies the shape of the output (in our example we are using 4 die)
replace argument to True states each element can be sampled more than once
*can add ‘p’ argument at end to give individualized weights/probabilities
this simulation gives us an approximate sampling distribution

Example variant and all together
```
# Expand a grid representing 5 8-sided dice
dice = expand_grid(
  {'die1': [1, 2, 3, 4, 5, 6, 7, 8],
   'die2': [1, 2, 3, 4, 5, 6, 7, 8],
   'die3': [1, 2, 3, 4, 5, 6, 7, 8],
   'die4': [1, 2, 3, 4, 5, 6, 7, 8],
   'die5': [1, 2, 3, 4, 5, 6, 7, 8]
  })

# 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')

# Draw a bar plot of mean_roll
dice['mean_roll'].value_counts(sort=False).plot(kind='bar')
plt.show()
```

Central Limit Theorem
**means of independent samples have normal distributions
as the sample size increases we see the distribution becoming more Gaussian (ie Normal) and the width of the sampling distribution gets narrower

Population and sampling distribution means and standard deviations
in our coffee_ratings example we took four separate sample means (roughly 5, 20, 80, and 320)
creating four separate distributions
however all four sample means are relatively close to the population mean
population std for this example is ~2.7
sample std's are relatively smaller (roughly 1.2, .6, .3, .13)
**std decreases as the sample size increases
why is this?
***first remember when calculating std with pandas, you have to set ddof argument to 0 when calculating population std
and ddof to 1 (sample std is pandas default) when calculating std for a sample
another consequence of the CLT is that if we divide the population std (in this case 2.7) by the square root of the sample size we will get an estimate of the std of the sampling distribution for that sample size
this isn't exact because of the randomness involved in the sampling process, but its pretty close
***standard error is the std of the sampling distribution
useful in estimating population std to setting expectations on what level of variability we would expect from the sampling process

Introduction to bootstrapping
example - focus on sampling with resampling
coffee_focus = coffee_ratings[['variety', 'country_of_origin', 'flavor']]
#to see which rows ended up in the sample, add a row index column called index with the reset_index method
coffee_focus = coffee_focus.reset_index()
#call sample() but this time with argument replace=True
coffee_resamp = coffee_focus.sample(frac=1, replace=True)
#count the values of the index column to see how many times each coffee ended up in the resampled dataset
coffee_resamp['index'].value_counts()
**this means that some coffees were used multiple times while others were not used at all
num_unique_coffees = len(coffee_resamp.drop_duplicates(subset='index'))
#find out how many coffees weren't sampled at all
len(coffee_ratings) - num_unique_coffees

Bootstrapping
we use resampling for a technique called bootstrapping
bootstrapping can be seen in some ways as the opposite of sampling from a population
with regular 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
when to use bootstrapping:
to try to understand the variability due to sampling
theoretically allows us to develop an understanding of sampling variability using a single sample
important in cases where we aren't able to sample the population multiple times to create a sampling distribution

Bootstrapping process

1. make a resample of the same size as the original sample
2. calculate the statistic of interest for this bootstrap sample
3. repeat steps 1 and 2 many times

**the resulting statistics are bootstrap statistics, and they form a bootstrap distribution

For loop this bootstrapping son of a gun

```
mean_flavors_1000 = []
for i in range(1000):
        mean_flavors_1000.append(np.mean(coffee_sample(frac=1, replace=True)
['flavor']))
plt.hist(mean_flavors_1000,)
plt.show()
```

example

```
# 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'])
    )

# Draw a histogram of the resample means
plt.hist(mean_danceability_1000)
plt.show()
```

Comparing sampling and bootstrap distributions
remember in the bootstrap distribution that each value is an estimate of the mean flavor score
recall that each of these values corresoponds to one potential sample mean from the thoeretical population
taking the mean of those means gives us a guess at the population mean
this mean will often be very close to the true population mean, but there is a key difference:
***key key element the bootstrop mean is usually almost identical to its original sample mean
**this can be a bad thing if the original sample wasn't closely representative of the population
this means that our bootstrap distribution mean will not be a good estimate of our true population mean
**bootstrapping cannot correct potential biases due to differences between the sample and the population

calculating the std of our bootstrapping distribution vs our sample we get an entirely different number
why?
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 std of the sampling distribution of that statistic
std of the bootstrap means can be used as a way to estimate this measure of uncertainty
if we multiply that SE by the sqrt of the sample size, we get an estimate of the std in the original population
standard_error = np.std(bootstrap_distn, ddof=1)
standard error is the std of the statistic of interest
standard_error * np.sqrt(samplesize)
standard error times square root of sample size estimates the population standard deviation

estimated standard error > std of the bootstrap distribution for a sample statistic
again pop std approx= SE x sqrt(samplesize)

***although bootstrapping was poor at estimating the population mean, it is generally great for estimating the population std

example
# 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 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])
```

Confidence intervals
values within on standard deviation of the mean
this gives a good sense of where most of the values in a distribution lie
rough example - weather prediction
we believe tomorrow will be approx 47F
we report a range of 40-54, this is a confidence interval
written as 47F (40F,54F) or 47F [40F, 54F] or 47 +-7F
47F is the 'point estimate'
with our estimate we have determined a margin of error of 7F

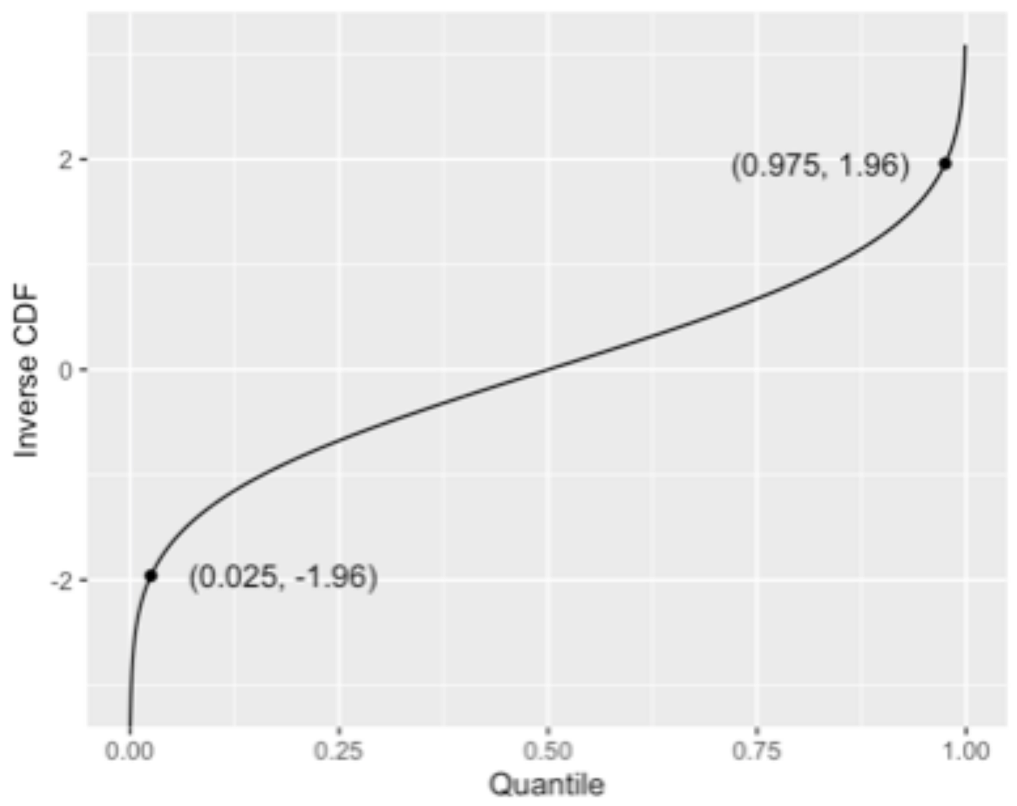
example
get mean then +- std

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

Quantile method for confidence intervals
quantiles split distributions into sections containing a proportion of the total data
we want 95% of values
go from 0.025 quantile to 0.975 quantile
to calculate the lower and upper bounds for this CI:

```
np.quantile(coffee_boot_distn, 0.025)
np.quantile(coffee_boot_distn, 0.975)
```

Second method to calculate confidence intervals
normal distribution's inverse cummulative distribution function
normal distribution's bell curve is the probability density function (PDF)
using calculus we can integrate this to get area under bell curve
this forms the cumulative distribution function (CDF)
by flipping the x and y axes, we get the inverse CDF



from scipy.stats import norm
norm.ppf(quantile, loc=0, scale=1)
takes a quantile between 0 and 1 and returns the values of the normal distribution for that quantile
default parameter of loc is 0 and scale is 1 which corresponds to the standard normal distribution
**notice in the above graph, the values corresponding to 0.025 and 0.975 are about minus and plus 2 for the standard normal distribution

this second method is called the standard error method for CI
#first calculate the point estimate, which is the mean of the bootstrap distribution
point_estimate = np.mean(coffee_boot_distn)
#second calculate the standard error, which is estimated by the std of the bootstrap distribution
std_error = np.std(coffee_boot_distn, ddof=1)
#then we call .ppf to get the inverse CDF of the normal distribution with the same mean and std as the bootstrap distribution
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))

Per lecturer, the most important things

-std of a bootstrap distribution statistic is a good approximation of the standard error of the sampling distribution
-the normal distribution tends to be a good approximation for bootstrap distributions