Case Study 4: Probability and Random Sampling

In this section, we explore random sampling from data, both with and without replacement. It is natural at this point to begin introducing ideas of probability, random variables, and distributions. We will need these concepts as we move from exploratory data analysis to principled statistical modeling and analysis.

This case study will demonstrate the following concepts:

- simple random sampling
- sampling with and without replacement
- probability of outcomes
- probabilities of events made up of multiple outcomes
- multiplication rule for counting compound events

and the following computational skills:

- using pandas to sample from a data frame
- defining a simple function
- iteration using a `for` loop

We will build these skills to be able to answer questions such as: **What is the probability of randomly drawing a full house in a deck of cards?**

Imports

We begin by importing the necessary Python package.

In [1]:
```
import pandas as pd
```

Courses Data

We construct a simple data frame to be used for illustration purposes. We'll create a similar UIUC course information data frame to that from Case Study 2.

In [2]:
```
courses = ['cs105', 'cs105', 'stat107', 'stat207', 'stat207',
           'badm210', 'badm210', 'ansc307']
sections = ['B', 'A', 'A', 'A', 'B', 'A', 'B', 'A']
enrollments = [134, 345, 343, 103, 123, 172, 216, 55]
sectdf = pd.DataFrame({'course': courses,
                       'section': sections,
                       'enrolled': enrollments})
sectdf
```

Out[2]:

	course	section	enrolled
0	cs105	B	134
1	cs105	A	345
2	stat107	A	343
3	stat207	A	103

	course	section	enrolled
4	stat207	B	123
5	badm210	A	172
6	badm210	B	216
7	ansc307	A	55

In [3]:
```
sectdf.shape
```

Out[3]:
```
(8, 3)
```

Now, we'll sort the data by course rubric (the shorthand code for the course, stored in the course variable) and section and save the results in place.

In [4]:
```
sectdf.sort_values(by = ['course', 'section'], inplace = True)
sectdf
```

Out[4]:

	course	section	enrolled
7	ansc307	A	55
5	badm210	A	172
6	badm210	B	216
1	cs105	A	345
0	cs105	B	134
2	stat107	A	343
3	stat207	A	103
4	stat207	B	123

1. Populations, Samples, and Observations

In this case study, the sectdf data frame will contain our **population** of UIUC courses that we're interested in.

In [5]:
```
sectdf
```

Out[5]:

	course	section	enrolled
7	ansc307	A	55
5	badm210	A	172
6	badm210	B	216
1	cs105	A	345
0	cs105	B	134
2	stat107	A	343
3	stat207	A	103
4	stat207	B	123

An example of a **sample** from this population might be a subset of 3 of these courses. The sample is **not a random sample** because we selected the rows that we wanted ahead of time.

In [6]:
```
sectdf.iloc[:3,:]
```

Out[6]:

	course	section	enrolled
7	ansc307	A	55
5	badm210	A	172
6	badm210	B	216

An example of an **observation** from this population would be a single row or observational unit. For this example, a row corresponds to a single section of a class. This would be the course meeting that you would enroll in.

In [7]:
```
sectdf.iloc[[1],:]
```

Out[7]:

	course	section	enrolled
5	badm210	A	172

2. Types of Random Sampling

Random sampling with and without replacement

The pandas `DataFrame.sample()` function provides a way to randomly sample from the rows of the data frame. Two major types of samples are:

- **Without replacement**, argument (replace=False), observations from the data frame can only be selected once in a given sample. Once an observation has been included in a sample, it is not returned to the pool of possible observations. This is the default setting for DataFrame.sample(). Random sampling without replacement is typical in surveys from large populations where we desire a manageable representative sample.
- **With replacement**, argument (replace=True), observations from the data frame can be selected multiple times in a given sample. After an observation has been included in a sample, it is returned to the pool of possible observations for subsequent draws and can therefore be selected multiple times. Random sampling with replacement is often used as a way to think about long run relative frequencies of events repeating an experiment under the same conditions.

For further information about other DataFrame.sample options, see pandas.DataFrame.sample documentation (https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.sample.html).

For reproducibility, DataFrame.sample has an option to set the random seed (e.g. random_state=12347) so that the random generator produces the same result each time it is run in the document.

Random Sampling Without Replacement

Here, we randomly sample 2 rows without replacement. We can run this multiple times and see that we get different samples of 2 items (courses) each time.

In [8]:

```
sectdf.sample(2)
```

Out[8]:

	course	section	enrolled
2	stat107	A	343
6	badm210	B	216

In [9]:

```
sectdf.sample(2)
```

Out[9]:

	course	section	enrolled
2	stat107	A	343
7	ansc307	A	55

What if we try to randomly sample 10 without replacement?

In [10]:

```
sectdf.sample(10)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/var/folders/8n/vcl88k4x7qnddbqbz4ljfx780000gn/T/ipykernel_86323/1686137758.py in <module>
----> 1 sectdf.sample(10)

~/miniconda3/lib/python3.8/site-packages/pandas/core/generic.py in sample(self, n, frac, r
eplace, weights, random_state, axis, ignore_index)
   5363             )
   5364 
-> 5365         locs = rs.choice(axis_length, size=n, replace=replace, p=weights)
   5366         result = self.take(locs, axis=axis)
   5367         if ignore_index:

mtrand.pyx in numpy.random.mtrand.RandomState.choice()

ValueError: Cannot take a larger sample than population when 'replace=False'
```

We get an error message because there aren't that many rows. After we sample the first 8 courses, no courses remain in the pool of possible courses to be drawn from.

Random Sampling With Replacement

Here, we sample 10 rows **with replacement:**

In [11]:

```
sectdf.sample(10, replace=True)
```

Out[11]:

	course	section	enrolled
6	badm210	B	216
7	ansc307	A	55
0	cs105	B	134
2	stat107	A	343
6	badm210	B	216
2	stat107	A	343

	course	section	enrolled
1	cs105	A	345
0	cs105	B	134
5	badm210	A	172
2	stat107	A	343

In [12]:
```
sectdf.sample(2, replace=True)
```

Out[12]:

	course	section	enrolled
2	stat107	A	343
4	stat207	B	123

3. Probability Experiments

Experiment 1: Random Sampling With Replacement

Let's go wild and select 1000 rows with replacement! Instead of displaying them, which could create a lengthy document, we'll save them in a new data frame.

In [13]:
```
bigsample = sectdf.sample(1000, replace=True, random_state=12347)
bigsample
```

Out[13]:

	course	section	enrolled
7	ansc307	A	55
4	stat207	B	123
3	stat207	A	103
3	stat207	A	103
0	cs105	B	134
...	...	...	...
3	stat207	A	103
4	stat207	B	123
7	ansc307	A	55
1	cs105	A	345
4	stat207	B	123

1000 rows × 3 columns

In [14]:
```
bigsample = sectdf.sample(1000, replace=True, random_state=12347)
bigsample
```

Out[14]:

	course	section	enrolled
7	ansc307	A	55
4	stat207	B	123

	course	section	enrolled
3	stat207	A	103
3	stat207	A	103
0	cs105	B	134
...	...	...	...
3	stat207	A	103
4	stat207	B	123
7	ansc307	A	55
1	cs105	A	345
4	stat207	B	123

1000 rows × 3 columns

In [15]:

```
bigsample = sectdf.sample(1000, replace=True, random_state=12347)
bigsample
```

Out[15]:

	course	section	enrolled
7	ansc307	A	55
4	stat207	B	123
3	stat207	A	103
3	stat207	A	103
0	cs105	B	134
...	...	...	...
3	stat207	A	103
4	stat207	B	123
7	ansc307	A	55
1	cs105	A	345
4	stat207	B	123

1000 rows × 3 columns

If we run the cell above multiple times, we see the same resulting data frame. Why does this occur?

The random state argument means that we start at the same location when generating our random sample. This is still a random sample, but it is reproducible. I can send you this code, and you'll get the same result. If I remove that argument, I get the following output, which is different. If you ran this same code, you'd likely get a different result, too!

In [16]:

```
bigsample = sectdf.sample(1000, replace=True)
bigsample
```

Out[16]:

	course	section	enrolled
0	cs105	B	134
5	badm210	A	172

	course	section	enrolled
7	ansc307	A	55
3	stat207	A	103
0	cs105	B	134
...	...	...	...
6	badm210	B	216
4	stat207	B	123
5	badm210	A	172
1	cs105	A	345
6	badm210	B	216

1000 rows × 3 columns

Let's return to the original, reproducible version of this data.

In [17]:
```
bigsample = sectdf.sample(1000, replace=True, random_state=12347)
bigsample
```

Out[17]:

	course	section	enrolled
7	ansc307	A	55
4	stat207	B	123
3	stat207	A	103
3	stat207	A	103
0	cs105	B	134
...	...	...	...
3	stat207	A	103
4	stat207	B	123
7	ansc307	A	55
1	cs105	A	345
4	stat207	B	123

1000 rows × 3 columns

In [18]:
```
bigsample.shape
```

Out[18]:
```
(1000, 3)
```

In [19]:
```
bigsample.head()
```

Out[19]:

	course	section	enrolled
7	ansc307	A	55
4	stat207	B	123
3	stat207	A	103

	course	section	enrolled
3	stat207	A	103
0	cs105	B	134

In this big sample, we can ask: How many times were each of the courses selected? Recall the `.value_counts()` function.

Using this, we can answer: **What proportion of classes in our random sample were actually STAT207?**

In [20]:
```
bigsample['course'].value_counts()
```

Out[20]:
```
stat207    259
cs105      247
badm210    239
ansc307    131
stat107    124
Name: course, dtype: int64
```

To understand the relative frequencies better, let's normalize by n=1000.

In [21]:
```
SampleProportions = bigsample['course'].value_counts()/1000  #OR
SampleProportions = bigsample['course'].value_counts(normalize=True)
```

And then let's sort this series by the index, so it is in alphabetical order.

In [22]:
```
SampleProportions.sort_index(inplace=True)  # sort by the index instead of values
SampleProportions
```

Out[22]:
```
ansc307    0.131
badm210    0.239
cs105      0.247
stat107    0.124
stat207    0.259
Name: course, dtype: float64
```

What proportion of classes in our random sample were actually a statistics class?

In [23]:
```
(259+124)/1000
```

Out[23]:
```
0.383
```

Experiment 2: Random Sampling With Replacement + Larger Sample Size

Let's go really wild and sample with replacement **a million times**! We'll code in the sample size so we only have to change one number (n) to modify the experiment, if we wanted to repeat this with another sample size.

In [24]:
```
n = 1000000

SampleProportions = sectdf.sample(n, replace=True,
                                  random_state=12347)['course'].value_counts()/n
SampleProportions.sort_index(inplace=True)

print('n =', n)
print('Sample Proportions:')
print('--------------------')
print(SampleProportions)
```

```
n = 1000000
Sample Proportions:
-------------------
ansc307    0.125076
badm210    0.249622
cs105      0.250614
stat107    0.124493
stat207    0.250195
Name: course, dtype: float64
```

It is enlightening to compare these "big sample" proportions with the corresponding proportions in the original data frame.

In [25]:
```
Proportions = sectdf['course'].value_counts()/sectdf['course'].size
Proportions.sort_index(inplace=True)
print('Proportions in the Original Data Frame:')
print('---------------------------------------')
print(Proportions)
```

```
Proportions in the Original Data Frame:
---------------------------------------
ansc307    0.125
badm210    0.250
cs105      0.250
stat107    0.125
stat207    0.250
Name: course, dtype: float64
```

Law of Large Numbers

To 2 significant digits, the proportions are the same! This is an example of what is often called the **law of large numbers**. In probability, if we **randomly sample with replacement n times** under the same conditions, then the proportion of times a particular outcome occurs gets closer and closer to the probability of that outcome (as n gets larger and larger).

4. Calculating Probabilities of Certain Types of Events

Experiment 3: Probability for a Simple Event that Follows a Uniform Probability Model

Suppose we also want to know how many times each index (0-7) in the original data frame was selected, i.e. each row of the data frame.

In [26]:
```
biggersample = sectdf.sample(n, replace=True, random_state=12347)
biggersample
```

Out[26]:

	course	section	enrolled
7	ansc307	A	55
4	stat207	B	123
3	stat207	A	103
3	stat207	A	103
0	cs105	B	134

	course	section	enrolled
...	...	...	...
7	ansc307	A	55
6	badm210	B	216
2	stat107	A	343
1	cs105	A	345
3	stat207	A	103

1000000 rows × 3 columns

In [27]:
```
biggersample.index.value_counts()
```

Out[27]:
```
3    125526
1    125347
0    125267
7    125076
5    125046
4    124669
6    124576
2    124493
dtype: int64
```

In [28]:
```
biggersample.index.value_counts(normalize=True)
```

Out[28]:
```
3    0.125526
1    0.125347
0    0.125267
7    0.125076
5    0.125046
4    0.124669
6    0.124576
2    0.124493
dtype: float64
```

More information:

In the original data frame from which we sampled, there were 8 rows, each of which had the same chance of being selected. In this case, we say that the row proabilities are all the same, and thus **uniform**.

If we selecte one row at random, this implies that each row has a 1/8 = 0.125 chance of being selected.

Uniform probability rule: If we make a random draw from a set of n possible choices, and each choice has *the same probability of selection*, then each outcome has probability $\frac{1}{n}$ of occurring.

Experiment 4: Probability for a Compound Event of Simple Events that Follows a Uniform Probability Model

Notice that, in our example, the course selections themselves are *not* uniformly distributed, because they appear in different numbers of rows. Instead, their probabilities are given by the proportion of times they appear in the original data frame. Courses that appear only once in the list have probability 1/8 of being selected. Courses that appear more than once have higher probabilities of selection. This observation leads to our second rule about uniform probability distributions.

Rule for calculating event probabilities from uniform probability distributions: If an event of interest includes k of the n possible choices in a random draw from a set, then the probability of the event is $\frac{k}{n}$.

What is the probability of selecting a statistics (stat) course if we choose one row at random from the following data frame?

In [29]:
```
sectdf
```

Out[29]:

	course	section	enrolled
7	ansc307	A	55
5	badm210	A	172
6	badm210	B	216
1	cs105	A	345
0	cs105	B	134
2	stat107	A	343
3	stat207	A	103
4	stat207	B	123

We see that there are 8 rows, and 3 of them correspond to stat courses. So the probability is 3/8.

If we randomly select a course from this list, what is the probability that the course enrollment is more than 300 students?

It might be helpful to sort by enrollment.

In [30]:
```
sectdf.sort_values(by='enrolled', ascending=False)
```

Out[30]:

	course	section	enrolled
1	cs105	A	345
2	stat107	A	343
6	badm210	B	216
5	badm210	A	172
0	cs105	B	134
4	stat207	B	123
3	stat207	A	103
7	ansc307	A	55

Two courses have enrollment larger than 300 students.

So, $\mathbb{P}(\text{enrolled} > 200) = 2/8$.

Calculating Probabilities of Two Independent Events With Replacement

It gets a bit more complicated when we think about events involving multiple draws. As an example, let's consider the probability of selecting two 'stat' courses. A useful way to think about this is to first imagine how many possible 2-row draws there are. These should all be equally likely. Then we need to determine how many of them consist of two 'stat' courses.

Actual Probability Calculation

The first part of this process is calculating the probability that both randomly drawn courses (drawn with replacement) are statistics courses).

With replacement: This case is easier. There are 8 possible choices for each draw, and the first draw has no effect on the second draw. Therefore, there are $8 \times 8 = 64$ possible samples of two course sections. In order to get two stat courses, we have to select stat courses both times. We have 3 different stat courses available, so there are $3 \times 3 = 9$ ways to select them. By the uniform probability rule, the probability of getting two stat courses is $\frac{9}{64} = 0.140625$.

This is the true probability of drawing two stat courses in a sample of two courses with replacement.

Approximate Probability by Simulation

While we were able to find the actual probability above, a few things may happen:

- you aren't sure how to calculate the probability,
- you want to confirm your probability calculation, or
- you have a more complicated probability calculation that is either challenging or impossible to calculate by hand.

In those instances, you may want to instead *approximate* the probability using a *simulation*.

As an example, we will see if we can approximate the probability that two stats courses are drawn with replacement by designing a simulation that:

1. Randomly samples (with replacement) two classes from the sectdf data frame (i.e., the population) repeatedly.
2. Each time, checks whether the two drawn are BOTH statistics classes.

To start, we can perform a **single** trial of this simulation.

In [31]:
```
my_sample = sectdf.sample(2, replace=True).sort_values(by='course')['course']
my_sample
```

Out[31]:
```
0      cs105
3    stat207
Name: course, dtype: object
```

The condition below **individually** checks whether each class in the current trial (i.e. sample) is a statistics course (i.e. is either 'stat107' or 'stat207').

The `|` in Python stands for "OR", so we are checking if either condition is true.

In [32]:
```
(my_sample == 'stat107') | (my_sample == 'stat207')
```

Out[32]:
```
0    False
3     True
Name: course, dtype: bool
```

The condition below checks whether **ALL** of the values in the condition above are True (i.e. both drawn courses are a statistics course).

```
In [33]: all((my_sample == 'stat107') | (my_sample == 'stat207'))
```

```
Out[33]: False
```

Now that we can report whether both courses selected are statistics courses from one trial, let's conduct this simulation again. This time, we will repeat the process for 20 trials (i.e. 20 random samples of size 2 drawn from sectdf with replacement).

```
In [34]:
# With replacement

# This will keep a tally of which trials (i.e. samples) have both classes
# as statistics classes.
# We will update this as we go through the simulation.
both_stat = 0

# This is the number of trials (i.e. samples we will collect)
num_trials = 20

# This is the beginning of a for-loop
# It says we will execute the code that is indented to the right num_trials times
for i in range(num_trials):
    print('Trial:',i)

    # Collect the sample here
    my_sample = sectdf.sample(2, replace=True).sort_values(by='course')['course']
    print(my_sample)

    # Checks whether both classes in the sample are statistics classes
    # If True, a 1 is added to the both_stat tally
    # If False, a 0 is added to the both_stat tally
    print(all((my_sample == 'stat107') | (my_sample == 'stat207')))
    both_stat += all((my_sample == 'stat107') | (my_sample == 'stat207'))
    print('Current Value of both_stat: ', both_stat)
    print('---------------------------------')

print('Probability estimate based on', num_trials, 'iterations:', both_stat/num_trials)
```

```
Trial: 0
7    ansc307
2    stat107
Name: course, dtype: object
False
Current Value of both_stat:  0
---------------------------------
Trial: 1
1      cs105
3    stat207
Name: course, dtype: object
False
Current Value of both_stat:  0
---------------------------------
Trial: 2
6    badm210
4    stat207
Name: course, dtype: object
False
Current Value of both_stat:  0
---------------------------------
Trial: 3
3    stat207
3    stat207
Name: course, dtype: object
```

```
True
Current Value of both_stat:  1
---------------------------------
Trial: 4
4    stat207
4    stat207
Name: course, dtype: object
True
Current Value of both_stat:  2
---------------------------------
Trial: 5
6    badm210
4    stat207
Name: course, dtype: object
False
Current Value of both_stat:  2
---------------------------------
Trial: 6
7    ansc307
3    stat207
Name: course, dtype: object
False
Current Value of both_stat:  2
---------------------------------
Trial: 7
3    stat207
3    stat207
Name: course, dtype: object
True
Current Value of both_stat:  3
---------------------------------
Trial: 8
1      cs105
2    stat107
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 9
7    ansc307
3    stat207
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 10
7    ansc307
6    badm210
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 11
7    ansc307
6    badm210
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 12
6    badm210
1      cs105
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
```

```
Trial: 13
7    ansc307
7    ansc307
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 14
7    ansc307
2    stat107
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 15
5    badm210
0      cs105
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 16
7    ansc307
3    stat207
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 17
1      cs105
2    stat107
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 18
0    cs105
1    cs105
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Trial: 19
6    badm210
4    stat207
Name: course, dtype: object
False
Current Value of both_stat:  3
---------------------------------
Probability estimate based on 20 iterations: 0.15
```

Remarks on the syntax:

- The operation `+=` is new here. It allows adding to an existing sum efficiently within a loop. In the example, `both_stat += ...` is equivalent to `both_stat = both_stat + ...`
- The function `all()` is also new. It takes the value True if all elements are True and False if any element is False. It is an implementation of the boolean "and" operator for multiple boolean elements.
- The for-loop is an important flow control operation that we will cover in the next chapter. Here we see it in action as a way to automate the simulation process, which would otherwise be extremely laborious.

Now, let's repeat this simulation with **n=1000** simulations.

- All we need to change is the **`num_trials`** argument, setting it equal to 1000.

- We will also delete the **`print`** statements for each trial in the code above, so that there is not *pages* of output.

In [35]:
```
# With replacement

# This will keep a tally of which trials (i.e. samples) have both classes
# as statistics classes.
# We will update this as we go through the simulation.
both_stat = 0

# This is the number of trials (i.e. samples we will collect)
num_trials = 1000

# This is the beginning of a for-loop
# It says we will execute the code that is indented to the right num_trials times
for i in range(num_trials):

    # Collect the sample here
    my_sample = sectdf.sample(2, replace=True).sort_values(by='course')['course']

    # Checks whether both classes in the sample are statistics classes
    # If True, a 1 is added to the both_stat tally
    # If False, a 0 is added to the both_stat tally
    both_stat += all((my_sample == 'stat107') | (my_sample == 'stat207'))

print('Probability estimate based on', num_trials, 'iterations:', both_stat/num_trials)
```

```
Probability estimate based on 1000 iterations: 0.146
```

If we want to repeat this simulation but with **n = 10000** simulations, all we need to change is **`num_trials=10000`**

In [36]:
```
# With replacement

# This will keep a tally of which trials (i.e. samples) have both classes
# as statistics classes.
# We will update this as we go through the simulation.
both_stat = 0

# This is the number of trials (i.e. samples we will collect)
num_trials = 10000

# This is the beginning of a for-loop
# It says we will execute the code that is indented to the right num_trials times
for i in range(num_trials):

    # Collect the sample here
    my_sample = sectdf.sample(2, replace=True).sort_values(by='course')['course']

    # Checks whether both classes in the sample are statistics classes
    # If True, a 1 is added to the both_stat tally
    # If False, a 0 is added to the both_stat tally
    both_stat += all((my_sample == 'stat107') | (my_sample == 'stat207'))

print('Probability estimate based on', num_trials, 'iterations:', both_stat/num_trials)
```

```
Probability estimate based on 10000 iterations: 0.1397
```

Calculating Probabilities of Two Independent Events Without Replacement

Now, suppose we want to randomly sample two classes from the sectdf data frame (i.e. population), but this time doing so **without replacement**.

We will again want to answer the question: "What is the probability that both courses are statistics courses?"

Actual Probability Calculation

Without replacement: if we select two course sections at random without replacement, then we have:

- 8 possible choices for the first row selected
- 7 possible choices for the second row selected.

Thus, there are $8 \times 7 = 56$ possible (first row, second row) selections.

BUT, it is the sample sample (course schedule) if we select cs105A first and badm210B second as if we selected badm210B first and cs105A second. So the total number of *samples of size two* is $8 \times 7/2 = 28$.

OUt of these, how many consist of two stat courses? We have three choices for the first draw, and two choices for the second draw, so 6 possible ordered draws. But two order samples give us the same set {stat107A, stat207B} can be from (stat107A, stat207B) or (stat207B, stat107A), so there are only three possible sets. Thus, the probability of picking two stat classes is $\frac{3}{28} \approx 0.1071$.

Approximate Probability Calculation with Simulations

Now, let's see if we can approximate this probability by designing a simulation instead.

1. Randomly sample (without replacement) two classes from the sectdf data frame (i.e. the population) repeatedly.
2. Each time, check whether the two drawn are BOTH statistics classes.

We can do this by making the following modification to our code for the simulation designed for "with replacement" sampling:

- Delete the `replace=True` in the **`sample()`** function, so now it will sample from sectdf **without replacement**.

In [37]:
```
# Without replacement simulation

# This will keep a tally of which trials (i.e. samples) have both classes
# as statistics classes.
# We will update this as we go through the simulation.
both_stat = 0

# This is the number of trials (i.e. samples we will collect)
num_trials = 10000

# This is the beginning of a for-loop
# It says we will execute the code that is indented to the right num_trials times
for i in range(num_trials):

    # Collect the sample here
    my_sample = sectdf.sample(2).sort_values(by='course')['course']

    # Checks whether both classes in the sample are statistics classes
    # If True, a 1 is added to the both_stat tally
    # If False, a 0 is added to the both_stat tally
    both_stat += all((my_sample == 'stat107') | (my_sample == 'stat207'))

print('Probability estimate based on', num_trials, 'iterations:', both_stat/num_trials)
```

```
Probability estimate based on 10000 iterations: 0.1153
```

5. Probability Calculation Shortcuts with Combinatorics

We can use combinatorics to help us calculate probabilities more quickly. We used them above for our actual probability calculations. Below, we will formalize these statements.

General Counting Rules

Generalizing from the previous examples, we have several general formulas for the number of ways to draw samples and subsamples. Assume n and k are positive integers. For sampling without replacement assume $k \leq n$. For sampling with replacement this is not necessary.

- The number of ways to select a set of k items **with replacement** if we keep track of the order of selection is

$$n^k = \underbrace{n \times n \times \cdots \times n}_{k \text{ times}}$$

- The number of ways to reorder (permute) n items is

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

- The number of ways sample k **ordered** items out of n **without replacement** where we keep track of the order of selection:

$$\underbrace{n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)}_{k \text{ terms}} = \frac{n!}{(n-k)!}$$

- The number of ways to sample k **unordered** items **without replacement** from n items:

$$\binom{n}{k} = \frac{n \times (n-1) \times (n-2) \times \cdots \times (n-k+1)}{k \times (k-1) \times \cdots \times 2 \times 1} = \frac{n!}{k!(n-k)!}$$

Multiplication Rule

If an event is made up of a series of choices with k_1 ways to make the first choice, k_2 ways to make the second choice, and so on, then the total number of combinations of choices is the product of the number of ways to make each of the individual choices (i.e., $k_1 \times k_2 \times k_3 \times \cdots$).

6. Case Study: Playing Cards

Calculate the probability of randomly drawing a full house (3 cards of one number and 2 cards of another number) from a standard 52 card deck of playing cards.

Actual Probability Calculation with Combinatorics

Combinatorial (counting) methods are often useful for determining the probabilities of events made up of multiple outcomes when the basic outcomes are sampled with equal probabilities. As an example, if 5 cards are dealt at random and without replacement from a 52 card deck, then all possible 5 card hands are equally likely. The probability of a given type of hand is then the ratio of the number of possible hands of that type over the total number of possible 5 card hands of any type.

Let's consider a full house, which is a compound event. The possible full hourses are all possible sets of three cards of one face value and two cards of another face value.

We need to figure out:

- How many possible sets of 5 cards are there?
- How many possible full house combinations are there?

How many possible ordered sequences of 5 cards?

$$52 \times 51 \times 50 \times 49 \times 48 = 52!/47!$$

How many possible poker hands?

The order of the cards as they are dealt is not important for the final poker hand. So the ordered sequence calculated above does not enumerate the total number of possible poker hands.

Instead, the ordered sequence of 5 cards we calculated above is the number of poker hands $\times$ the number of ways of reordering the five cards. Thus, using combinatorial notation,

$$52 \times 51 \times 50 \times 49 \times 48 = 5 \times 4 \times 3 \times 2 \times 1 \times \binom{52}{5}$$

.

The number of possible poker hands (unordered) is therefore

$$\binom{52}{5} = \frac{52 \times 51 \times 50 \times 49 \times 48}{5 \times 4 \times 3 \times 2 \times 1} = \frac{52!}{5!47!}$$

How many possible full houses?

This is a bit more complicated. We will separate this calculation into more manageable pieces:

- There are 13 choices for the face value of the triplet
- There are then 12 choices for the face value of the pair (after the face value of the triplet has been set).
- There are $\binom{4}{3}$ ways to select the three cards for the triplet based on the card suits.
- There are $\binom{4}{2}$ ways to select the two cards for the pair based on the card suits.

Putting this together, the number of possible full houses is

$$13 \times 12 \times \binom{4}{3} \times \binom{4}{2} = 13 \times 12 \times 4 \times 6$$

Because each possible poker hand is equally likely, this means that the probability of a full house is

$$\frac{\#\text{ possible full houses}}{\#\text{ possible poker hands}} = \frac{13 \times 12 \times 4 \times 6}{\binom{52}{5}}$$

In [38]:

```
# Probability of full house
print('Probability of full house = ',
      13*12*4*6*5*4*3*2/(52*51*50*49*48))
```

```
Probability of full house =  0.0014405762304921968
```

Approximate Probability Calculation by Simulation

Now, we will instead approximate this same probability using simulation.

We will load a data frame of card information. This data frame contains the following information about each card (observational unit) in the standard 52 deck:

- color (black or red)
- suit (club, diamond, spade, or heart), and
- value (King, Queen, Jack, 10, 9, 8, 7, 6, 5, 4, 3, 2, or Ace).

In [39]:

```
cards = pd.read_csv('cards.csv')
cards
```

Out[39]:

	color	suit	face
0	black	club	A
1	black	club	2
2	black	club	3
3	black	club	4
4	black	club	5
5	black	club	6
6	black	club	7
7	black	club	8
8	black	club	9
9	black	club	10
10	black	club	J
11	black	club	Q
12	black	club	K
13	black	spade	A
14	black	spade	2
15	black	spade	3
16	black	spade	4
17	black	spade	5
18	black	spade	6
19	black	spade	7
20	black	spade	8
21	black	spade	9
22	black	spade	10
23	black	spade	J
24	black	spade	Q
25	black	spade	K

	color	suit	face
26	red	diamond	A
27	red	diamond	2
28	red	diamond	3
29	red	diamond	4
30	red	diamond	5
31	red	diamond	6
32	red	diamond	7
33	red	diamond	8
34	red	diamond	9
35	red	diamond	10
36	red	diamond	J
37	red	diamond	Q
38	red	diamond	K
39	red	heart	A
40	red	heart	2
41	red	heart	3
42	red	heart	4
43	red	heart	5
44	red	heart	6
45	red	heart	7
46	red	heart	8
47	red	heart	9
48	red	heart	10
49	red	heart	J
50	red	heart	Q
51	red	heart	K

In [40]:
```
cards.shape
```

Out[40]:
```
(52, 3)
```

A single trial of the simulation

First, we will see how to draw a single "hand" (i.e. 5 cards) in poker. To do this, we sample without replacement.

In [41]:
```
# single poker hand
cards.sample(5)
```

Out[41]:

	color	suit	face
4	black	club	5

	color	suit	face
51	red	heart	K
42	red	heart	4
18	black	spade	6
23	black	spade	J

What does a full hourse look like when we use the `.value_counts()` function?

Let's first make a test case of a full house and see what that looks like with the `.value_counts()` function.

In [42]:
```
test_case_1 = pd.DataFrame({'face': ['2','2','2','3','3'],
                            'suit': ['heart', 'club', 'spade', 'club', 'diamond']})
test_case_1
```

Out[42]:

	face	suit
0	2	heart
1	2	club
2	2	spade
3	3	club
4	3	diamond

In [43]:
```
test_case_1['face'].value_counts()
```

Out[43]:
```
2    3
3    2
Name: face, dtype: int64
```

Now, let's build a function to test whether a given hand is a full house. We'll use the `.value_counts()` function here.

In [44]:
```
def isfullhouse(df, var='face'):
    # First checks if you have 5 cards in the hand.
    if df[var].shape[0] != 5:
        return 'Not a poker hand'
    else:
        counts = df[var].value_counts()
        if counts.min() == 2 and counts.max() == 3:
            return True
        else:
            return False
```

Let's check this function against several test cases:

In [45]:
```
isfullhouse(test_case_1)
```

Out[45]:
```
True
```

test_case_1 was a full house. The function is able to correctly identify full house hands.

In [46]:
```
test_case_2 = pd.DataFrame({'face': ['2','2','2','2','A'],
                                   'suit': ['heart','club','spade','diamond', 'club']})
test_case_2
```

Out[46]:

	face	suit
0	2	heart
1	2	club
2	2	spade
3	2	diamond
4	A	club

In [47]:

```
isfullhouse(test_case_2)
```

Out[47]:

```
False
```

test_case_2 was not a full house. It looks like the function is able to correctly identify non-full house hands.

In [48]:

```
test_case_3 = pd.DataFrame({'face': ['2', '2', '2', '3', 'A', 'K'],
                            'suit': ['heart', 'club', 'spade', 'club', 'diamond', 'heart']
test_case_3
```

Out[48]:

	face	suit
0	2	heart
1	2	club
2	2	spade
3	3	club
4	A	diamond
5	K	heart

In [49]:

```
isfullhouse(test_case_3)
```

Out[49]:

```
'Not a poker hand'
```

test_case_3 was not a poker hand, as it had 6 cards. The function is able to correctly identify a set of cards that is not valid.

Repeated Trials of the Simulation

Let's generate a simulation that:

1. Simulates drawing 50,000 hands (i.e. 5 cards) from the 52 card deck. Each trial is randomly sampled without replacement. After each trial, the cards are put back in the deck.
2. Tests whether the trial hand is a full house.
3. Calculates the proportion of the 50,000 hands that were a full house.

In [50]:

```
## Draw poker hands at random
## Estimate the probability of a full house

# This will keep a tally of the hands that are full houses.
# We will update this as we iterate through the for-loop.
```

```
fh_count = 0

# Number of trials/hands in the simulation
n_iterations = 50000

# This will iterate through n_iterations
for i in range(n_iterations):
    # Draw the sample of 5 cards from the 'cards' data frame
    new_hand = cards.sample(5)

    # Tests whether the randomly sampled hand is a full house
    # 1 is added to the fh_count tally if it is True (a full house)
    # 0 is added to the fh_count tally if it is False (not a full house)
    fh_count += isfullhouse(new_hand)
```

In [51]:

```
print(fh_count)
print(n_iterations)
print('Probability estimate based on', n_iterations, 'iterations:', fh_count/n_iterations)
```

```
66
50000
Probability estimate based on 50000 iterations: 0.00132
```

7. Case Study: Coin Flips

Suppose we toss a fair coin 10 times. One possible sequence of head and tails with 6 tails is THTTTHTTHH. How many possible sequences of heads and tails are there with exactly 6 tails?

We can solve this using combinatorics.

DENOMINATOR: Number of Possible Outcomes

We toss a coin 10 times. How many possible outcomes are there?

ANSWER: Each of the 10 positions in the sequence has 2 possible choices, heads or tails. Therefore, the number of possible sequences of heads and tails is:

$$2 \times 2 \times 2 \times \cdots \times 2 = 2^{10} = 1024$$

NUMERATOR: Number of Possible Outcomes with 6 tails

How many possible sequences of heads and tails are there with exactly 6 tails?

ANSWER: There are 10 positions available in the sequence. Choose 6 of them for the tails. The number of ways to do this is:

$$\binom{10}{6} = \frac{10 \times 9 \times 8 \times 7 \times 6 \times 5}{6 \times 5 \times 4 \times 3 \times 2 \times 1} = 210$$

Putting it all together

All possible sequences are equally likely if we toss a fair coin independently. Therefore, the probability of exactly 6 tails out of 10 tosses is

$$\frac{\binom{10}{6}}{2^{10}} = \frac{210}{1024} = 0.2051$$

In [52]:
```
TenChoose6 = 10*9*8*7*6*5/(6*5*4*3*2)
TwoTo10th = 2**10
(TenChoose6, TwoTo10th, TenChoose6/TwoTo10th)
```

Out[52]:
```
(210.0, 1024, 0.205078125)
```
