Unit 5: Probability and Random Sampling

Case Study: What is the probability of randomly drawing a full house in a deck of cards?

Information

In this section, we explore random sampling from the 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.

We will demonstrate the following key ideas:

- Simple random sampling
- Sampling with and without replacement
- Probabilities of outcomes
- Probablilities of events made up of multiple outcomes
- Multiplication rule for counting compound events

Computational skills:

- Using pandas to sample from a data frame
- Defining a simple function
- Iteration using a for loop

Imports

To start, we import the necessary package(s) and construct a simple data frame to be used for illustration.

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

1. How is probability used in the data science pipeline?

See the Unit 5 slides (section 1) for information on how probbility is used in the data science pipeline.

1.1 UIUC Course Artifical Dataframe Creation

Let's create the same UIUC course information dataframe that we used in the previous unit's lecture.

Initially create the dataframe.

In [2]:
```
courses = ['cs105', 'cs105', 'stat107', 'stat207', 'badm210', 'badm210', 'badm
210', 'adv307']
sections = ['B', 'A', 'A', 'A', 'A', 'C', 'B', 'A']
enrollments = [345, 201, 197, 53, 215, 197, 178, 37]
sectdf = pd.DataFrame({'course': courses,
                       'section': sections,
                       'enrolled': enrollments})
sectdf
```
Out[2]:

	course	section	enrolled
0	cs105	B	345
1	cs105	A	201
2	stat107	A	197
3	stat207	A	53
4	badm210	A	215
5	badm210	C	197
6	badm210	B	178
7	adv307	A	37

Then sort it.

Next, we'll sort the data in-place by course rubric and section.

In [3]:
```
sectdf.sort_values(by=['course', 'section'], inplace=True)
```
In [4]:
```
sectdf
```
Out[4]:

	course	section	enrolled
7	adv307	A	37
4	badm210	A	215
6	badm210	B	178
5	badm210	C	197
1	cs105	A	201
0	cs105	B	345
2	stat107	A	197
3	stat207	A	53

1.2 Population vs. Sample vs. Observation

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

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

	course	section	enrolled
7	adv307	A	37
4	badm210	A	215
6	badm210	B	178
5	badm210	C	197
1	cs105	A	201
0	cs105	B	345
2	stat107	A	197
3	stat207	A	53

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

```
In [6]: sectdf.iloc[0:3,:]
```

Out[6]:

	course	section	enrolled
7	adv307	A	37
4	badm210	A	215
6	badm210	B	178

An example of an **observation** from this population would be a single row/class.

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

Out[7]:

	course	section	enrolled
4	badm210	A	215

2. Experiment Related Definitions

See the Unit 5 (section 2) slides for experiment related definitions.

3. Types of Random Sampling

See Unit 5 (section 3) slides for more examples of random sampling with and without replacement.

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 sample are:

- **Without replacement** (replace=False) - items pulled from the data frame can only be seleted once in a given sample, currently 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** (replace=True) - items remain in the pool after being selected and therefore can 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.

3.1 Random Sampling Without Replacement

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

```
In [8]: sectdf.sample(2)
```

Out[8]:

	course	section	enrolled
3	stat207	A	53
6	badm210	B	178

```
In [9]: sectdf.sample(2)
```

Out[9]:

	course	section	enrolled
4	badm210	A	215
5	badm210	C	197

What if we try to randomly sample 10 without replacement? We'll get an error message because there aren't that many rows.

```
In [10]: # sectdf.sample(10)
```

3.2 Random Sampling With Replacement

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

```
In [11]: sectdf.sample(10, replace=True)
```

Out[11]:

	course	section	enrolled
7	adv307	A	37
1	cs105	A	201
0	cs105	B	345
4	badm210	A	215
5	badm210	C	197
0	cs105	B	345
7	adv307	A	37
2	stat107	A	197
7	adv307	A	37
5	badm210	C	197

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

Out[12]:

	course	section	enrolled
2	stat107	A	197
4	badm210	A	215

4. Two Different Definitions of Probability

4.1 Bayesian Definition of Probability

See Unit 5 (section 4.1) slides for definition.

4.2 Frequentist Definition of Probability

See Unit 5 (section 4.2) slides for definition.

<u>**Experiment 1**</u>**:**

First collect a random sample of size n=1000 WITH REPLACEMENT.

Let's go wild and select 1000 rows with replacement! Instead of showing them we'll load them into a new dataframe.

```
In [13]: #Run this multiple times, why is the dataframe the same everytime?
         bigsample = sectdf.sample(1000, replace=True, random_state=12347)  # save the
          seed for reproducibility.
         bigsample
```

Out[13]:

	course	section	enrolled
7	adv307	A	37
3	stat207	A	53
2	stat107	A	197
2	stat107	A	197
1	cs105	A	201
...	...	...	...
2	stat107	A	197
3	stat207	A	53
7	adv307	A	37
5	badm210	C	197
3	stat207	A	53

1000 rows × 3 columns

```
In [14]: bigsample.shape
```

```
Out[14]: (1000, 3)
```

```
In [15]: bigsample.head()
```

Out[15]:

	course	section	enrolled
7	adv307	A	37
3	stat207	A	53
2	stat107	A	197
2	stat107	A	197
1	cs105	A	201

What proportion of classes in our random sample were actually STAT207?

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

```
In [16]: bigsample["course"].value_counts()
```

```
Out[16]: badm210    365
         cs105      245
         stat207    134
         adv307     131
         stat107    125
         Name: course, dtype: int64
```

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

```
In [17]: SampleProportions = bigsample["course"].value_counts()/1000
```

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

```
In [18]: SampleProportions.sort_index(inplace=True)   # sort by the index instead of th
         e values
         SampleProportions
```

```
Out[18]: adv307     0.131
         badm210    0.365
         cs105      0.245
         stat107    0.125
         stat207    0.134
         Name: course, dtype: float64
```

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

```
In [19]: (134+125)/1000
```

```
Out[19]: 0.259
```

<u>Experiment 2</u>:

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.

```
In [20]: n = 1000000

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

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

```
n= 1000000
Sample Proportions:
------------------
adv307     0.125076
badm210    0.374969
cs105      0.249760
stat107    0.125526
stat207    0.124669
Name: course, dtype: float64
```

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

```
In [21]: 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:
---------------------------------------
adv307     0.125
badm210    0.375
cs105      0.250
stat107    0.125
stat207    0.125
Name: course, dtype: float64
```

4.3 Law of Large Numbers:

See Unit 5 (section 4.3) slides for picture of this effect.

To 3 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).

5. How to Calculate the Probability of *Certain Types of Events.*

See section 5 in the Unit 5 lecture slides.

5.1. Event Type 1: How to Calculate the Probability of a Simple Event that Follows a Uniform Probability Model

Experiment 3:

Suppose we also want to know how many times was each index (0-7) in the original dataframe was selected (ie. each row of the dataframe).

```
In [22]: biggersample = sectdf.sample(1000000, replace=True, random_state=12347)  # save the seed for reproducibility.
biggersample
```

Out[22]:

	course	section	enrolled
7	adv307	A	37
3	stat207	A	53
2	stat107	A	197
2	stat107	A	197
1	cs105	A	201
...	...	...	...
7	adv307	A	37
6	badm210	B	178
0	cs105	B	345
5	badm210	C	197
2	stat107	A	197

1000000 rows × 3 columns

```
In [23]: biggersample.index.value_counts()
```

```
Out[23]: 2    125526
5    125347
1    125267
7    125076
4    125046
3    124669
6    124576
0    124493
dtype: int64
```

In [24]:
```
biggersample.index.value_counts()/1000000
```
Out[24]:
```
2    0.125526
5    0.125347
1    0.125267
7    0.125076
4    0.125046
3    0.124669
6    0.124576
0    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 probabilities are all the same, and thus **uniform.**

If we select 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.

5.2. Event Type 2: How to Calculate the Probability of a Compound Event of Simple Events that Follow 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 proportions 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 $\frac{k}{n}$.

Example 1. What is the probability of selecting a "stat" course if we choose one row at random from the following data frame?

In [25]:
```
sectdf
```
Out[25]:

	course	section	enrolled
7	adv307	A	37
4	badm210	A	215
6	badm210	B	178
5	badm210	C	197
1	cs105	A	201
0	cs105	B	345
2	stat107	A	197
3	stat207	A	53

We see that there are 8 rows, and 2 of them correspond to "stat" courses. So the probability = $2/8$

Example 2. if we randomly select a course from this list, what is the probability that the course enrollment is more than 200 students? It might be helpful to sort by enrollment.

```
In [26]: sectdf.sort_values(by='enrolled', ascending=False)
```

Out[26]:

	course	section	enrolled
0	cs105	B	345
4	badm210	A	215
1	cs105	A	201
5	badm210	C	197
2	stat107	A	197
6	badm210	B	178
3	stat207	A	53
7	adv307	A	37

Pr(Enrolled > 200) = $3/8$

5.3. Event Type 3: How to Calculate the Probability of a Two Independent Events (or Two Observation Outcomes Sampling with Replacement)

See Unit 5 (Section 5.3) slides for definitions and information.

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.

5.3.1: *Calculating* the probability that both courses 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 x 8 = 64 possible samples of two course sections. In order to get two stat courses we have to select stat courses both times, and we have 2 different stat courses available, so there are 2 x 2 = 4 ways to select them. By the uniform probability rule, the probability of getting two stat courses is $\frac{4}{64}$ = $\frac{1}{16}$ = 0.0625.

5.3.2: *Approximating* the probability that both courses randomly drawn courses (drawn with replacement) are statistics courses *with a simulation*.

Now let's see if we can approximate this probability instead by designing a simulation that does the following.

1. Repeatedly randomly samples (with replacement) two classes from the sectdf dataframe (ie. the population).
2. And each time checks whether the two drawn are BOTH statistics class or not.

Below could be thought of as a *single* trial of this simulation.

```
In [27]: my_sample = sectdf.sample(2, replace=True).sort_values(by='course')['course']
         my_sample
```

```
Out[27]: 4    badm210
         2    stat107
         Name: course, dtype: object
```

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

```
In [28]: #Does the sorted sample equal the two statistics classes we're looking for in
          both positions?
         (my_sample == 'stat107') | (my_sample == 'stat207')

Out[28]: 4    False
         2     True
         Name: course, dtype: bool
```

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

```
In [29]: #Is every entry in a True?
         all((my_sample == 'stat107') | (my_sample == 'stat207'))

Out[29]: False
```

Now let's repeat this conduct this simulation again. This time using 20 trials (ie. 20 random samples of size 2 drawn from sectdf with replacement).

```
In [48]: # With replacement

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

         #This is the number of trials(ie. 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 tim
         es.
         for i in range(num_trials):
             print('Trial:',i)

             #Collects the sample here.
             my_sample = sectdf.sample(2, replace=True).sort_values(by='course')['cours
         e']
             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/n
         um_trials)
```

```
Trial: 0
6    badm210
1      cs105
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 1
7     adv307
2    stat107
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 2
5    badm210
2    stat107
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 3
6    badm210
0      cs105
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 4
6    badm210
1      cs105
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 5
1    cs105
0    cs105
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 6
5    badm210
3    stat207
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 7
4    badm210
5    badm210
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 8
7    adv307
1     cs105
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 9
5    badm210
0      cs105
Name: course, dtype: object
False
Current Value of both_stat:  0
--------------------------------
Trial: 10
2    stat107
3    stat207
Name: course, dtype: object
```

```
True
Current Value of both_stat:  1
--------------------------------
Trial: 11
5    badm210
0      cs105
Name: course, dtype: object
False
Current Value of both_stat:  1
--------------------------------
Trial: 12
5    badm210
3    stat207
Name: course, dtype: object
False
Current Value of both_stat:  1
--------------------------------
Trial: 13
5    badm210
1      cs105
Name: course, dtype: object
False
Current Value of both_stat:  1
--------------------------------
Trial: 14
6    badm210
4    badm210
Name: course, dtype: object
False
Current Value of both_stat:  1
--------------------------------
Trial: 15
2    stat107
3    stat207
Name: course, dtype: object
True
Current Value of both_stat:  2
--------------------------------
Trial: 16
0    cs105
1    cs105
Name: course, dtype: object
False
Current Value of both_stat:  2
--------------------------------
Trial: 17
5    badm210
3    stat207
Name: course, dtype: object
False
Current Value of both_stat:  2
--------------------------------
Trial: 18
4    badm210
2    stat107
Name: course, dtype: object
False
Current Value of both_stat:  2
--------------------------------
Trial: 19
4    badm210
3    stat207
Name: course, dtype: object
False
Current Value of both_stat:  2
--------------------------------
Probability estimate based on  20 iterations: 0.1
```

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 simulatoin process, which would otherwise be extremely laborious.

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

- All we need to change is the num_trials=1000
- We will delete the **print** statements in the code above so the output does not take up too much space.

In [31]:
```
# With replacement

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

#This is the number of trials(ie. 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 tim
es.
for i in range(num_trials):

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

    #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/n
um_trials)
```

```
Probability estimate based on  1000 iterations: 0.055
```

Now let's repeat this simulation with n=10000 simulation.

- All we need to change is the num_trials=10000

In [32]:
```
# With replacement

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

#This is the number of trials(ie. 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 tim
es.
for i in range(num_trials):

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

    #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/n
um_trials)
```

```
Probability estimate based on  10000 iterations: 0.0606
```

5.4. Event Type 4: How to Calculate the Probability of a Two Independent Events (or Two Observation Outcomes Sampling with Replacement)

See Unit 5 (Section 5.4) slides for definitions and information.

Now suppose we want to randomly sample two classes from the sectdf dataframe (ie. population), this time doing so **without replacement.*)

5.4.1: *Calculating* the probability that both courses randomly drawn courses (drawn without replacement) are statistics courses.

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 8 x 7 = 56 possible (first row, second row) selections.

BUT, it is the same sample (course schedule) if we select cs105A first and badm210C second as if we selected badm210C first and cs105A second. So the total number of *samples of size two* is 8 x 7 / 2 = 28.

Out of these, how many consist of two stat courses? we have 2 choices for the first draw, 1 choice for the second draw, so 2 possible ordered draws. But the two order samples give us the same set, {stat107A, stat207A}, so there is only one possible set like this. Thus the probability of picking two stat clases is $\frac{1}{28} \approx 0.0357$.

5.4.2: *Approximating* the probability that both courses randomly drawn courses (drawn without replacement) are statistics courses *with a simulation*.

Now let's see if we can approximate this probability instead by designing a simulation that does the following.

1. Repeatedly randomly samples (without replacement) two classes from the sectdf dataframe (ie. the population).
2. And each time checks whether the two drawn are BOTH statistics class or not.

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 [33]:
```
# WITHOUT REPLACEMENT SIMULATION

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

#This is the number of trials(ie. 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 tim
es.
for i in range(num_trials):

    #Collects 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/n
um_trials)
```

```
Probability estimate based on  10000 iterations: 0.0339
```

6. Using Combinatorics Equations to Help Us Quickly Count Events

Go to the Unit 5 (section 6 slides) for explanations on this.

<u>Supplementary Information</u>: General expressions for the number of possible samples with and without replacement

Generalizing from the previous examples, we have several general formulas for the number of ways to draw samples and subsample. 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

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

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

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

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

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

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

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

7. Using the Multiplication Rule to also Help Us Quickly Count Events

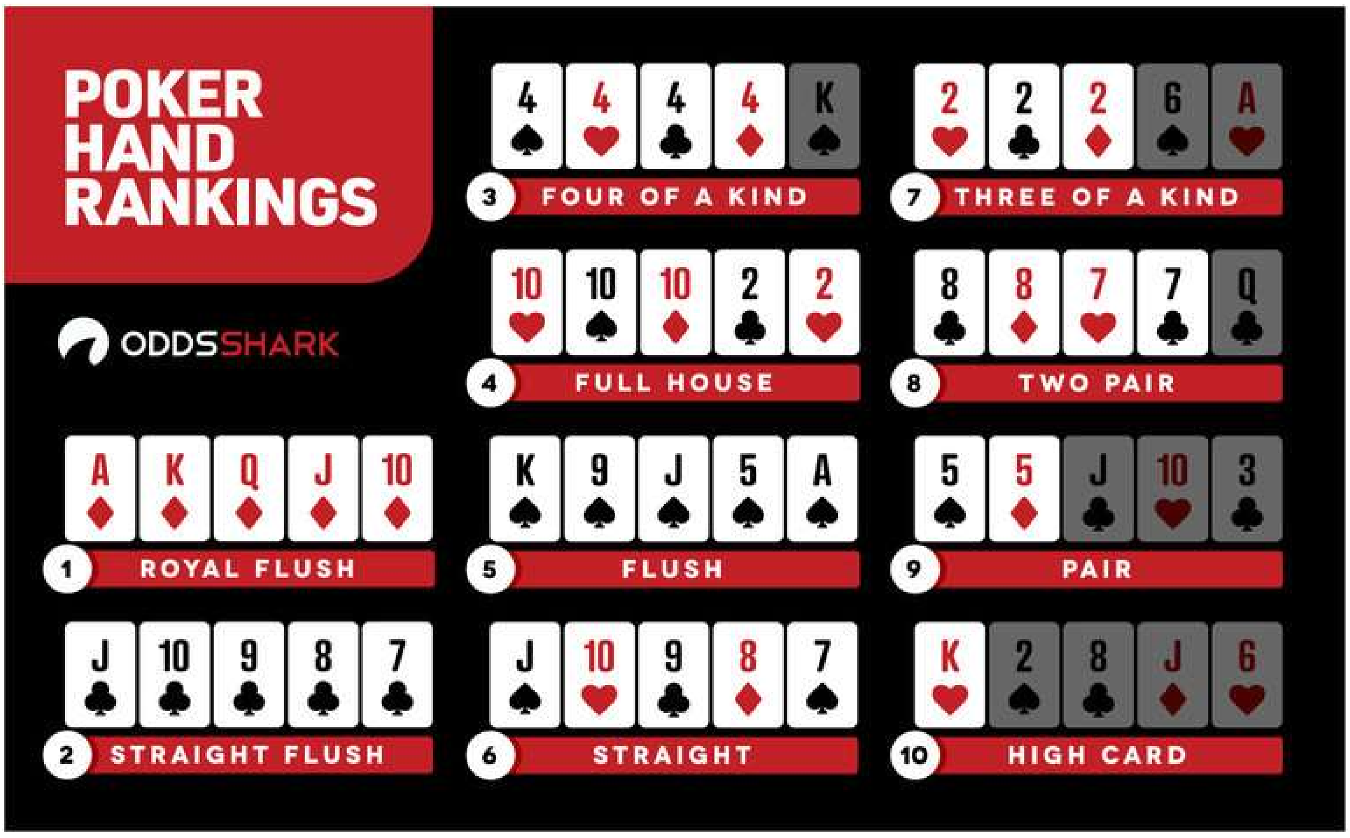
Go to the Unit 5 (section 7 slides) for explanations on this.

<u>Supplementary Information</u>:

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 numbers of ways to make each of the individual choices (i.e., $k_1 * k_2 * k_3 * \cdots$).

8. <u>Case Study</u>: *Calculate* the probability of randomly drawing a full house from a standard 52 card deck of playing cards.

See Unit 5 (section 8) slides for how to do this.



(Image source: https://www.oddsshark.com/ (https://www.oddsshark.com/))

<u>Supplementary Information:</u> Calculating the probabilities using combinatorics

Combinatorial (counting) methods are often useful for determining the probabilities of events made up of multiple outcomes when the basic outcoems 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 houses are all possibel 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?

Example 1: How many possible ordered sequences of 5 cards?

$$52 * 51 * 50 * 49 * 48 = 52!/47!$$

Example 2: How many possible poker hands? First notice that the number of ways to select an ordered sequence of 5 cards = number of poker hands $\times$ number of ways of reordering the five cards. Thus, using combinatorial notation,

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

and the number of possible poker hands is

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

Example 3: How many possible full houses? This is a bit more complicated. There are 13 choices for the face value triplet, and then 12 choices for the face value duo. There are then $\binom{4}{3}$ ways to select three of the four cards of the first face value, and $\binom{4}{2}$ ways to choose the of the 4 cards of the second face value. Putting this together, the number of possible full houses is

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

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

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

In [34]:
```
# 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
```

9. <u>Case Study </u>: Approximate the probability of randomly drawing a full house from a standard 52 card deck of playing cards with a simulation.

Load dataframe of card information.

Let's load a data frame detailing the following information about each card in the standard 52 deck:

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

```
In [35]: cards=pd.read_csv('cards.csv')
         cards
```

Out[35]:

	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
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

	color	suit	face
47	red	heart	9
48	red	heart	10
49	red	heart	J
50	red	heart	Q
51	red	heart	K

Below could be thought of as a single trial of the simulation.

When drawing a "hand" (ie. 5 cards) in poker, we sample without replacement.

In [36]:
```
# single poker hand
cards.sample(5)
```

Out[36]:

	color	suit	face
30	red	diamond	5
12	black	club	K
35	red	diamond	10
39	red	heart	A
49	red	heart	J

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

Let's first make a test case of a full house and see what it looks like when we use the .value_counts() function

In [37]:
```
test_case_1 = pd.DataFrame({'face': ['2','2','2','3','3'],
                            'suit': ['heart', 'club', 'spade', 'club', 'diamon
d']})
test_case_1
```

Out[37]:

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

In [38]:
```
test_case_1['face'].value_counts()
```

Out[38]:
```
2    3
3    2
Name: face, dtype: int64
```

Let's build a function to test whether a given hand is a full house, that uses the value_counts() function.

In [39]:
```python
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 it against several test cases.

In [40]:
```python
isfullhouse(test_case_1)
```

Out[40]:
```
True
```

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

In [41]:
```python
test_case_2 = pd.DataFrame({'face': ['2','2','2','2','A'],
                            'suit': ['heart', 'club', 'spade', 'club', 'diamon
d']})
test_case_2
```

Out[41]:

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

In [42]:
```python
isfullhouse(test_case_2)
```

Out[42]:
```
False
```

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

In [43]:
```python
test_case_3 = pd.DataFrame({'face': ['2','2','2','3','A', 'K'],
                            'suit': ['heart', 'club', 'spade', 'club', 'diamon
d', 'heart']})
test_case_3
```

Out[43]:

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

In [44]:
```python
isfullhouse(test_case_3)
```

Out[44]:
```
'Not a poker hand'
```

Now let's conduct the simulation that will do the following.

1. Simulate drawing 50000 hands (ie. 5 cards) from the 52 card deck. Each trial is randomly sampled without replacement. (After the 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 50000 hands that were a full house.

In [45]:
```python
## Draw poker hands at random
## Estimate probability of 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):
    #Draws the sample of 5 cards from the 'cards' dataframe.
    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 (ie. a full house).
    #0 is added to the fh_count tally if it is False (ie. not a full house).
    fh_count += isfullhouse(new_hand)
```

In [46]:
```python
print(fh_count)
print(50000)
print('Probability estimate based on ', n_iterations, 'iterations:', fh_count/
n_iterations)
```

```
60
50000
Probability estimate based on  50000 iterations: 0.0012
```

10. Case Study: We toss a coin 10 times. One possible sequence of heads and tails with 6 tails is THTTTHTTHH. How many possible sequences of heads and tails are there with exactly 6 tails?

See the Unit 5 (Section 10) slides for more detailed information.

Supplementary Information :

NUMERATOR: We toss a coin 10 times. One possible sequence of heads and tails with 6 tails is THTTTHTTHH. 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 ways to do this is

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

DENOMINATOR: What is the probability of getting 6 tails in 10 tosses of a fair coin?

Answer: We already know there are 210 ways to get 6 tails. All possible sequences are equally likely if we toss the coin independently. How many possible sequences of heads and tails are there?

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

$$2*2*2*\cdots*2 = 2^{10} = 1024.$$

PUTTING IT ALL TOGETHER:

Therefore, the probability of exactly 6 tails out of 10 tosses is

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

In [47]:
```
TenChoose6=10*9*8*7*6*5/(6*5*4*3*2)
TwoTo10th=2**10
(TenChoose6, TwoTo10th, TenChoose6/TwoTo10th)
```

Out[47]:
```
(210.0, 1024, 0.205078125)
```

STAT 207, Victoria Ellison and Douglas Simpson, University of Illinois at Urbana-Champaign

In [ ]:
```

```