Randomness

Readings

- **Computational and Inferential Thinking: The Foundations of Data Science**
  - **Chap 9. Randomness**

- Link for the textbook: https://inferentialthinking.com

Comparison Operators

- The result of a comparison expression is a `bool` value
- Assignment statements
  - x = 2 y = 3
- Comparison expressions
  - x > 1 x > y y >= 3
  - x == y x != 2 2 < x < 5
- Comparing strings
  - ‘Dog’ > ‘Catastrophe’ > ‘Cat’
    - alphabetical order, e.g., ‘a’ < ‘b’
    - string length, e.g., ‘abc’ < ‘abcd’

Aggregating Comparisons

- Summing an array or list of bool values will count the True values only.

```
1    + 0     + 1          ==
True + False + True       ==
sum([1    , 0    , 1   ]) ==
sum([True, False, True])  ==
```

Comparing an Array and a Value

- comparison applies to each element of the array

```
tosses = np.array(['Tails', 'Heads', 'Tails', 'Heads', 'Heads'])
tosses == 'Tails'
```

→ `array([ True, False, True, False, False])`

- the result array can be aggreated

```
np.count_nonzero(tosses == 'Tails’)
```

```
sum(tosses == 'Tails')
```

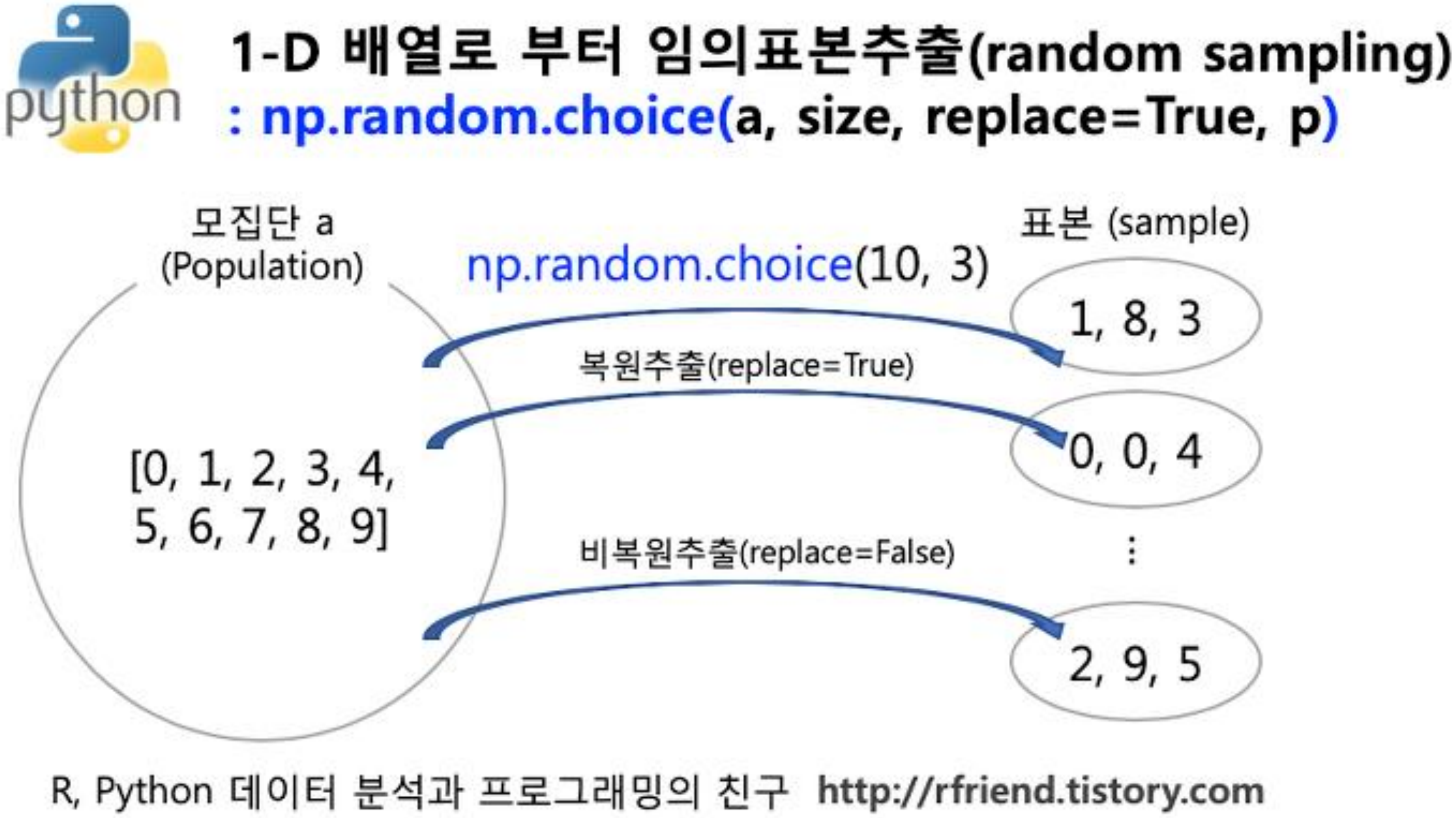
→ 2

Random Selection

- `np.random.choice(some_array, sample_size)`
  - Selects uniformly at random
  - with replacement
  - from an array,
  - a specified number of times

- `np.random.choice(a, size=None, replace = True, p=None)`
  - a: 1-D array or int
  - size: (optional) output shape. e.g., (m, n, k) → m * n * k samples are drawn
  - replace: (optional) sample with(True)/without(False) replacement
  - p: (optional) probabilities associated with each entry in a,
    - None: uniform distribution

sample with replacement and without replacement



Uniform Random Sample demo

- Run the code below multiple times and see the results

```
two_groups = np.array(['treatment', 'control'])
np.random.choice(two_groups)
```

- Repeat the sampling 10 times

```
np.random.choice(two_groups, 10)
```

Example: Betting on a Die

- Suppose you bet on a roll of a fair die. The rules of the game
  - If the die shows 1 spot or 2 spots, I lose a dollar.
  - If the die shows 3 spots or 4 spots, I neither lose money nor gain money.
  - If the die shows 5 spots or 6 spots, I gain a dollar.
- Define a function one_bet(x), x = # on the die

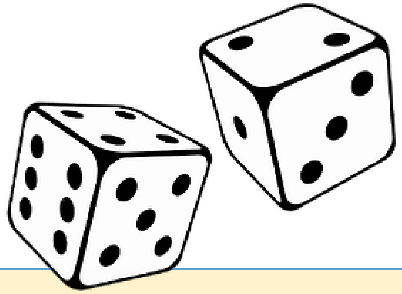
```
def one_bet(x):
      """Returns my net gain if the die shows x spots"""
      if x <= 2:
            return -1
      elif x <= 4:
            return 0
      elif x <= 6:
            return 1
```

Example: Betting on a Die (cont.)

- one roll of a fair die

```
np.random.choice(np.arange(1, 7))
```

- Simulating betting on a die



```
one_bet(np.random.choice(np.arange(1, 7)))
```

Iteration

- What if you want to see the results of 300 rolls of the die?
- First embed the random sample code into the function

```
def bet_on_one_roll():
        """Returns my net gain on one bet"""
        # roll a die once and   record the number of spots
        x = np.random.choice(np.arange(1, 7))
        if x <= 2:
                return -1
        elif x <= 4:
                return 0
        elif x <= 6:
                return 1
```

Iteration (cont.)

1. make an empty array for outcomes
2. iterating the bet n times
3. augmenting the outcome array within the for loop

```
outcomes = np.array([])

for i in np.arange(300):
      outcome_of_bet = bet_on_one_roll()
      outcomes = np.append(outcomes, outcome_of_bet)
```

```
len(outcomes)
```

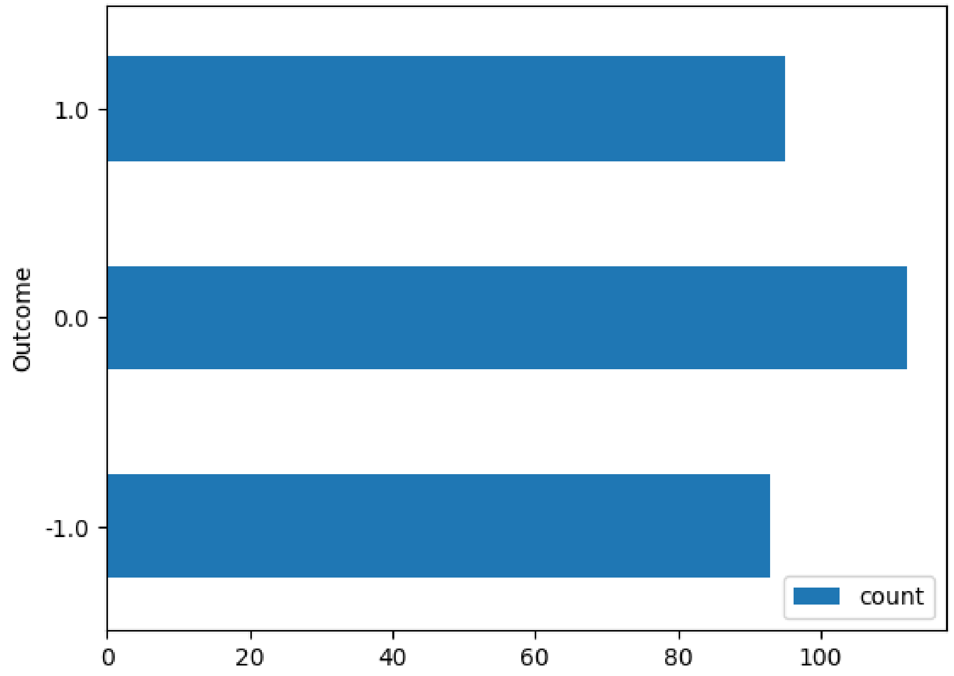
Augmenting arrays

- **`np.append(array_1, value)`**
  - new array with `value` appended to `array_1`
  - `value` has to be of the same type as elements of `array_1`
- **`np.append(array_1, array_2)`**
  - new array with `array_2` appended to `array_1`
  - `array_2` elements must have the same type as `array_1` elements

Iteration (cont.)

- Visualize the outcome

```
outcome_table = pd.DataFrame({'Outcome': outcomes})
outcome_table = outcome_table.groupby('Outcome')['Outcome'].count()
outcome_table = outcome_table.reset_index(name='count')
fig = outcome_table.plot.barh(x='Outcome', y='count')
```



Simulation

- the process of using a computer to mimic physical experiments
  - those experiments will almost invariably involve chance (at least in this class)

- The process (Number of Heads in 100 Tosses)
  1. What to Simulate: # coin toss
  2. Simulating One Value: one set of 100 tosses
  3. Number of Repetitions: e.g., a loop for 20,000 repetitions
  4. Simulating Multiple Values: augmenting outcome array

Simulation demo

- What to Simulate: coin toss

```
coin = np.array(['Heads', 'Tails'])
np.random.choice(coin, 10)
```

Simulation demo (cont.)

- Simulating One Value: one set of 100 tosses

```
outcomes = np.random.choice(coin, 100)
num_heads = np.count_nonzero(outcomes == 'Heads')
num_heads
```

```
def one_simulated_value():
      outcomes = np.random.choice(coin, 100)
      return np.count_nonzero(outcomes == 'Heads')
```

Simulation demo (cont.)

- Number of Repetitions: e.g., a loop for 20,000 repetitions

```
num_repetitions = 20000 # number of repetitions

# repeat the process num_repetitions times
for i in np.arange(num_repetitions):
      # simulate one value using the function defined
      new_value = one_simulated_value()
```

Simulation demo (cont.)

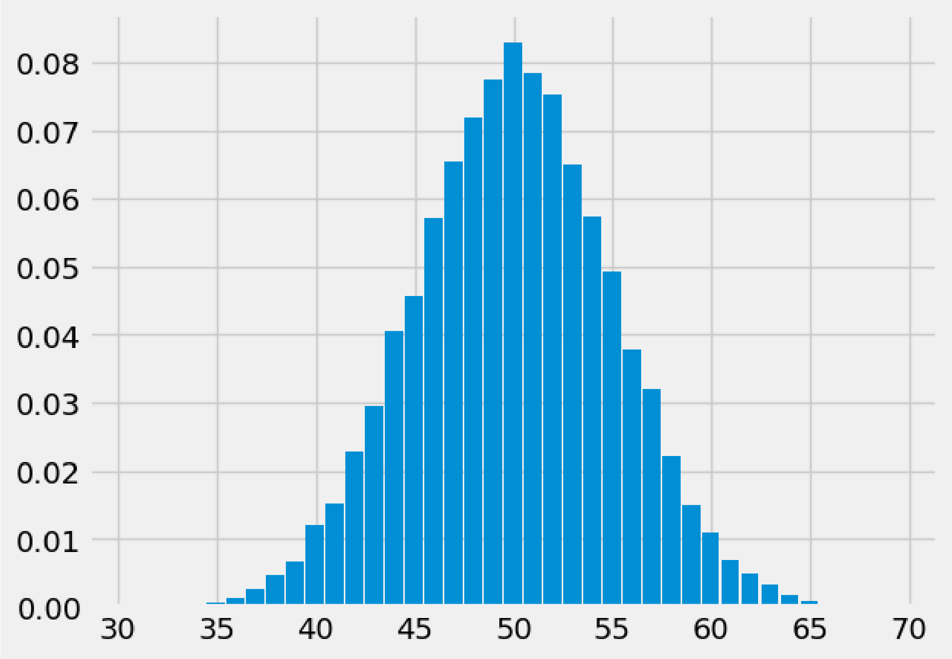
- Simulating Multiple Values: augmenting outcome array

```
num_repetitions = 20000 # number of repetitions

heads = np.array([])

# repeat the process num_repetitions times
for i in np.arange(num_repetitions):
      # simulate one value using the function defined
      new_value = one_simulated_value()
      # augment the collection array with the simulated value
      heads = np.append(heads, new_value)
```

Simulation demo (cont.)

- Draw distribution (histogram)

```
rst_pd = pd.DataFrame(
      {'Repetition': np.arange(1, num_repetitions + 1),
      'Number of Heads': heads}
)
fig = rst_pd['Number of Heads'].hist(bins=np.arange(30.5, 69.6, 1),
density = True, width=.9)
```



Chace: Basics

- possibility of something happening
- **Lowest value**: 0
  - Chance of event that is impossible
- **Highest value**: 1 (or 100%)
  - Chance of event that is certain
- **Complement**: If an event has chance 70%, then the chance that it doesn't happen is
  - 100% - 70% = 30%
  - 1 - 0.7 = 0.3
  - i.e., P(~A) = 1 – P(A)

Equally Likely Outcomes

- **Assuming** all outcomes are equally likely, the chance of an event A is:

- $P(A) = \dfrac{number\ of\ outcomes\ that\ make\ A\ happen}{total\ number\ of\ outcomes}$

- e.g., the chance that the die shows an even numbers is
  - $P(A) = \dfrac{\#\{2,4,6\}}{\#\{1,2,3,4,5,6\}} = \dfrac{3}{6}$

A Question

- I have three cards: **ace of hearts**, **king of diamonds**, and **queen of spades**.
- I shuffle them and draw two cards *at random without replacement.*

- What is the chance that I get the Queen followed by the King?

Multiplication Rule

- Chance that two events *A* and *B* both happen

  = P(*A* happens) x P(*B* happens given that *A* has happened)

  - The answer is *less than or equal to* each of the two chances being multiplied
  - The more conditions you have to satisfy,
    the less likely you are to satisfy them all

Another Qeustion

- I have three cards: **ace of hearts**, **king of diamonds**, and **queen of spades**.
- I shuffle them and draw two cards *at random without replacement*

- What is the chance that one of the cards I draw is a King and the other is Queen?

Addition Rule

- If event *A* can happen in *exactly one* of two ways, then

$$P(A) = P(\text{first way}) + P(\text{second way})$$

  - The answer is *greater than or equal to* the chance of each individual way

Complement: At Least One Head

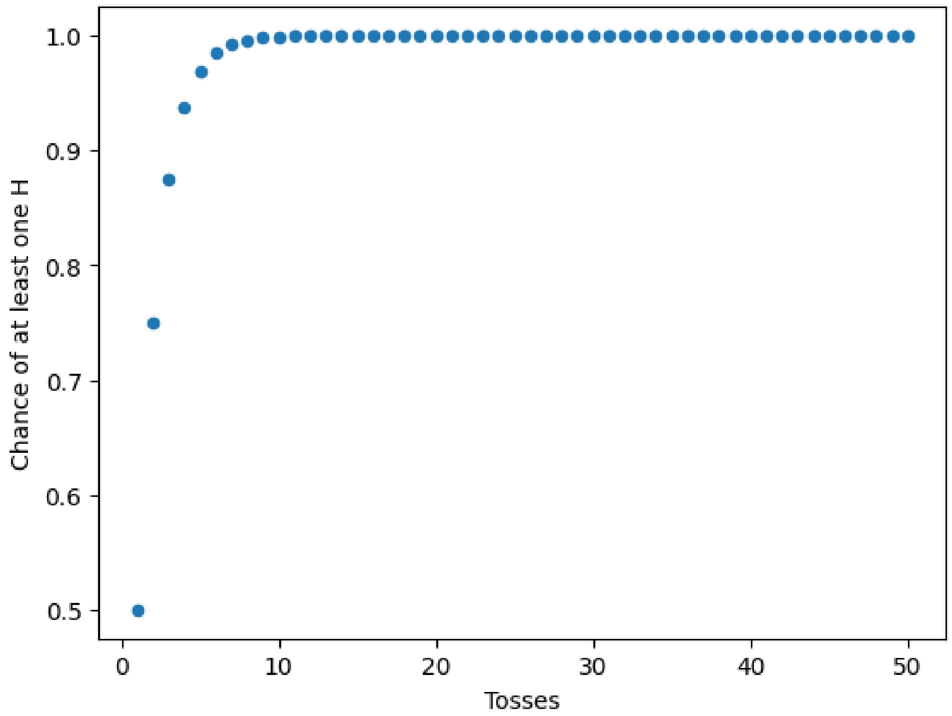
- In 3 tosses:
  - Any outcome *except* TTT
  - P(TTT) = (1/2) x (1/2) x (1/2) = 1/8
  - P(at least one head) = 1 - P(TTT) = 1 - (1/8) = 87.5%

- In 10 tosses:
  - 1 - (1/2)**10 ≅ 99.9%

Complement demo

```
tosses = np.arange(1, 51, 1)
results = pd.DataFrame({'Tosses': tosses,
                        'Chance of at least one H': 1 - (1/2)**tosses})
results.head(10)
```

	Tosses	Chance of at least one H
0	1	0.500000
1	2	0.750000
2	3	0.875000
3	4	0.937500
4	5	0.968750
5	6	0.984375
6	7	0.992188
7	8	0.996094
8	9	0.998047
9	10	0.999023



Problem-Solving Method

- Here's a method that works widely:
  Ask yourself what event must happen on the first trial.
  - If there's a clear answer (e.g. "not a six") whose probability you know, you can most likely use the **multiplication rule**.
  - If there's no clear answer (e.g. "could be K or Q, but then the next one would have to be Q or K …"), list all the **distinct ways** your event could occur and **add up their chances.**
  - If the list above is long and complicated, look at the **complement**.
    If the complement is simpler (e.g. the complement of "at least one" is "none"), you can find its chance and subtract that from 1.

Discussion Question

- A population has 100 people, including Rick and Morty. We sample two people at random without replacement.

- P(both Rick and Morty are in the sample)

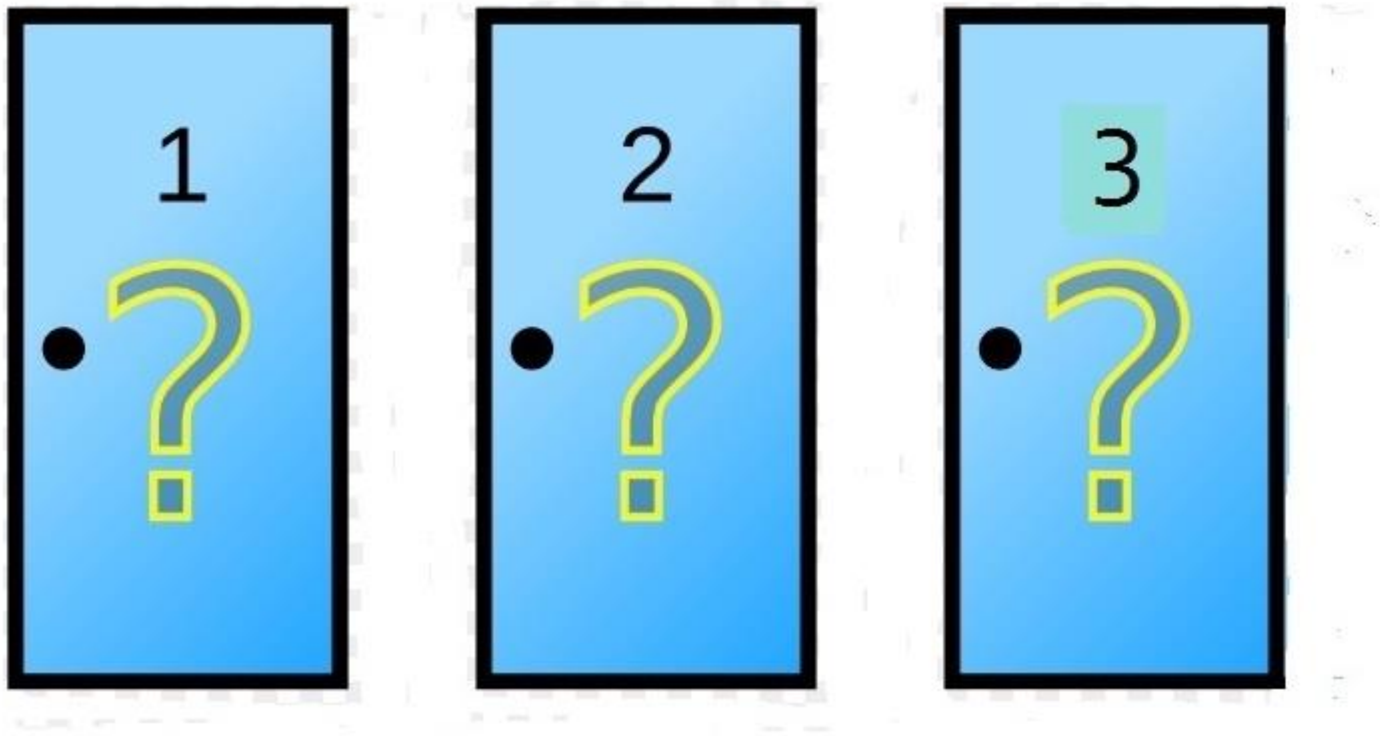
  $= P(\text{first Rick, then Morty}) + P(\text{first Morty, then Rick})$

  $= (1/100) * (1/99) \quad + \quad (1/100) * (1/99) \qquad = 0.0002$

- P(neither Rick nor Morty is in the sample)

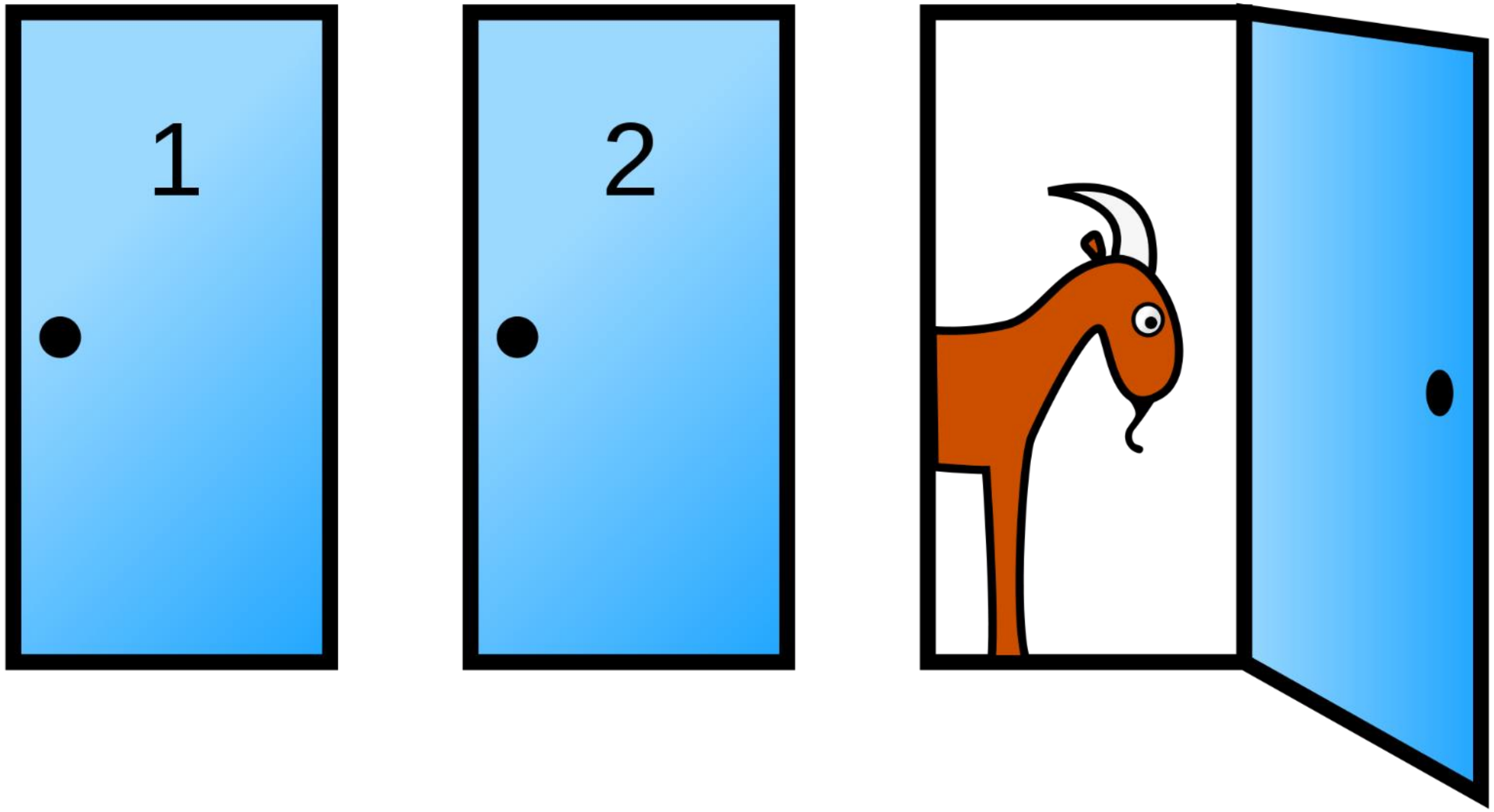
  $= (98/100)*(97/99) \qquad = 0.9602$

The Monty Hall Problem



https://probabilityandstats.files.wordpress.com/2017/05/monty-hall-pic-1.jpg

Stay or Switch?



https://en.wikipedia.org/wiki/Monty_Hall_problem

Equally Likely Outcomes

Monty Hall Simulation

- What to Simulate → what's behind all three doors
  - contestant's first pick
  - Monty opened
  - remaining door
- Simulating One Play
  - setting up arrays

```
goats = np.array(['first goat', 'second goat']) # distinct goats
hidden_behind_doors = np.append(goats, 'car') # items behind the doors
```

Monty Hall Simulation

- Simulating One Value → one monty hall game play
  - generate [contestant's guess, Monty reveals, remained]

```
# for choosing a goat behind the unopened door
def other_goat(x):
        if x == 'first goat':
                return 'second goat'
        elif x == 'second goat':
                return 'first goat'
```

```
def monty_hall_game():
    contestant_guess = np.random.choice(hidden_behind_doors)
    if contestant_guess == 'first goat':
        return [contestant_guess, 'second goat', 'car']
    if contestant_guess == 'second goat':
        return [contestant_guess, 'first goat', 'car']
    if contestant_guess == 'car':
        revealed = np.random.choice(goats)
        return [contestant_guess, revealed, other_goat(revealed)]
```

Monty Hall Simulation

- Repeating the game multiple times and collecting the simulated results

```
# empty collection table
games= pd.DataFrame(columns=['Guess', 'Revealed', 'Remaining'])

# Play the game 10000 times and
# record the results in the table games

for i in np.arange(10000):
      games.loc[i] = monty_hall_game()
```

Monty Hall Simulation

- Simulation result

```
games.head()
```

	Guess	Revealed	Remaining
0	second goat	first goat	car
1	car	second goat	first goat
2	car	second goat	first goat
3	car	first goat	second goat
4	car	first goat	second goat

- Grouping on items for the initial pick and remaining door

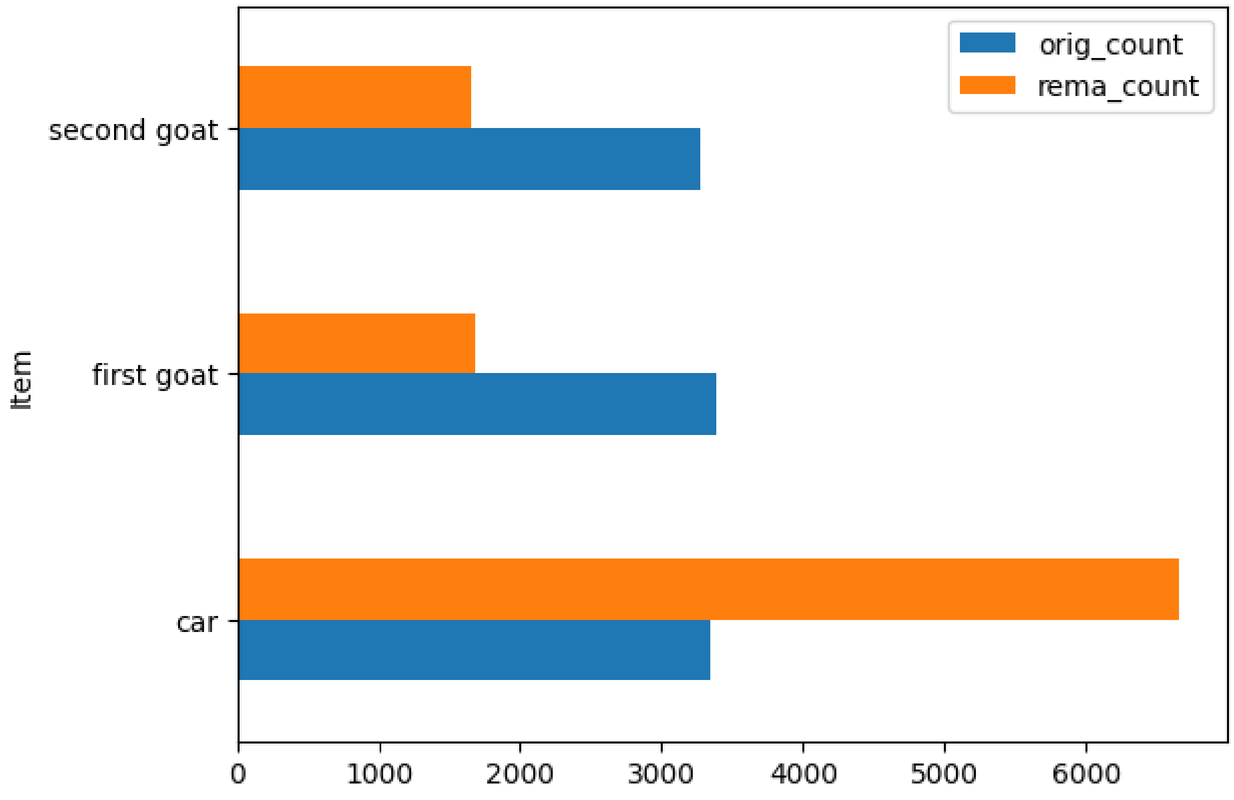
```
original_choice =\
games.groupby('Guess')['Guess'].count().reset_index(name='orig_count')

remaining_door =\
games.groupby('Remaining')['Remaining'].count().reset_index(name='rema_count')

joined = original_choice.join(remaining_door.set_index('Remaining'), on='Guess')
```

	Guess	orig_count	rema_count
0	car	3345	6655
1	first goat	3382	1683
2	second goat	3273	1662

Monty Hall Simulation

- Visualize the distribution

```
fig = joined.set_index('Guess').plot.barh(ylabel = 'Item')
```



Switch!

Q&A