Final Project Report

- Class: DS 5100
- Student Name: Alexa Owen
- Student Net ID: amo9f
- This URL: https://github.com/aowen18/amo9f_ds5100_montecarlo/blob/main/montecarlo/FinalProjectTem

Instructions

Follow the instructions in the Final Project isntructions and put your work in this notebook.

Total points for each subsection under **Deliverables** and **Scenarios** are given in parentheses.

Breakdowns of points within subsections are specified within subsection instructions as bulleted lists.

This project is worth **50 points**.

Deliverables

The Monte Carlo Module (10)

- URL included, appropriately named (1).
- Includes all three specified classes (3).
- Includes at least all 12 specified methods (6; .5 each).

Put the URL to your GitHub repo here.

Repo URL: https://github.com/aowen18/amo9f_ds5100_montecarlo

Paste a copyy of your module here.

NOTE: Paste as text, not as code. Use triple backticks to wrap your code blocks.

```
import numpy as np
import pandas as pd
import random

class Die:
    '''
    A class representing a die.

    Attributes:
        _die_df (pd.DataFrame): A private data frame containing the
faces and their weights.
```

```
    '''
    def __init__(self, face):
        '''
        Initializes the die object with a set of faces and default
weights.

        Args:
            face: A numpy array of sides (int or str) of the die.

        Raises:
            TypeError: If faces argument is not a NumPy array.
            ValueError: If the faces array does not have distinct
values.
        '''
        if not isinstance(face, np.ndarray):
           raise TypeError("Input must be a numpy array")

        if len(face) > len(set(face)):
            raise ValueError("The sides must be unique values")

        weights = np.ones(len(face))

        self._die_df = pd.DataFrame(weights.T, index=face)
        self._die_df = self._die_df.rename( columns = {0:
'Weights'})

    def adj_weights(self, face, new_weight):
        '''
        Changes the weight of a single face in the die.

        Args:
            face: The face value to be changed.
            new_weight: The new weight to set for the specified
face.

        Raises:
            IndexError: If the face is not in the die array.
            TypeError: If the new_weight is not numeric (integer or
float) or castable as numeric.
        '''
        if face not in self._die_df.index:
            raise IndexError("The value is not in the array")

        if not isinstance(new_weight, (int,float)):
            raise TypeError("Weight must be an int or float value")

        self._die_df['Weights'][face] = new_weight
        return self._die_df

    def die_roll(self, roll = 1):
        '''
        Rolls the die one or more times.
```

```
        Args:
            roll: Number of times the die is rolled. Defaults to 1.

        Returns:
            list: A Python list of outcomes from the rolls.
        '''
        results = random.choices(self._die_df.index,
weights=self._die_df['Weights'], k=roll)
        return results

    def show_df(self):
        """
        Returns a copy of the private die data frame containing
faces and weights.

        Returns:
            pd.DataFrame: A copy of the private die data frame.
        """
        return self._die_df.copy()

class Game:
    '''
     A class representing a game with similar dice.

    Attributes:
        _dice_list (list): A list of similar dice (Die objects).
        _play_df (pd.DataFrame): A private data frame to store the
results of the game.
    '''
    def __init__(self, _dice_list):
        '''
        Initializes the game object with a list of similar dice.

        Args:
            _dice_list (list): A list of already instantiated
similar dice.
        '''
        self._dice_list = _dice_list
        self._play_df = None

    def play(self, roll):
        '''
        Plays the game by rolling all dice a given number of times.

        Args:
            roll (int): Number of times the dice should be rolled.

        Returns:
            pd.DataFrame: A data frame of results with roll number
as index and dice outcomes as columns.
        '''
        results = {f'Die_{i}': die.die_roll(roll) for i, die in
enumerate(self._dice_list)}
```

```
            self._play_df = pd.DataFrame(results)
            return self._play_df.copy()

    def recent_play(self, form='wide'):
            '''
            Shows the results of the most recent play.

            Args:
                form (str, optional): The form of the data frame to
return ('wide' or 'narrow'). Defaults to 'wide'.

            Returns:
                pd.DataFrame: A copy of the private play data frame in
the specified form.

            Raises:
                ValueError: If the user passes an invalid option for
narrow or wide.
            '''
            if form not in ['wide','narrow']:
                raise ValueError("The form must be 'wide' or
'narrow'.")

            if form == 'wide':
                return self._play_df.copy()

            else:
                nar_df =
self._play_df.melt(ignore_index=False).reset_index().set_index(['index',
'variable'])
                return nar_df.copy()


class Analyzer:
    '''
    A class representing an analyzer for game results.

    Attributes:
        game (Game): A game object whose results will be analyzed.
    '''
    def __init__(self, game):
            '''
            Initializes the analyzer object with a game.

            Args:
                game (Game): A game object.

            Raises:
                ValueError: If the passed value is not a Game object.
            '''
            if not isinstance(game, Game):
                raise ValueError("The game object was not found in
Game.")
```

```
        self.game = game

    def jackpot(self):
        '''
        Computes how many times the game resulted in a jackpot (all
faces are the same).

        Returns:
            jp_count: A count of the number of jackpots.
        '''
        match = self.game.recent_play()
        jp_count = match.apply(lambda row: row.nunique() == 1,
axis=1).sum()
        jp_count = jp_count.item()
        return jp_count

    def face_counts (self):
        '''
        Computes how many times a given face is rolled in each
event.

        Returns:
            pd.DataFrame: A data frame of results with roll number
as index, face values as columns, and count values in the cells.
        '''
        face = self.game.recent_play()
        face = face.apply(lambda row:
row.value_counts()).fillna(0).astype(int)
        return face

    def combo_count (self):
        '''
        Computes the distinct combinations of faces rolled, along
with their counts.

        Returns:
            pd.DataFrame: A data frame of results with distinct
combinations as MultiIndex and count as a column.
        '''
        combo = self.game.recent_play()
        combo = pd.DataFrame(np.sort(combo.values, axis = 1),
columns = combo.columns)
        combo_df = combo.value_counts().to_frame().rename(columns =
{0:'count'})
        return combo_df

    def perm_count (self):
        '''
        Computes the distinct permutations of faces rolled, along
with their counts.

        Returns:
```

```
            pd.DataFrame: A data frame of results with distinct
permutations as MultiIndex and count as a column.
        '''
        perm = self.game.recent_play()
        perm = perm.value_counts().to_frame().rename(columns =
{0:'count'})
        return perm
```

Unitest Module (2)

Paste a copy of your test module below.

NOTE: Paste as text, not as code. Use triple backticks to wrap your code blocks.

- All methods have at least one test method (1).
- Each method employs one of Unittest's Assert methods (1).

```
import unittest
import numpy as np
import pandas as pd
from montecarlo import Die, Game, Analyzer


class MonteCarloTestCase(unittest.TestCase):
    def test_1_die_init(self):
        d = Die(np.array([1, 2, 3, 4, 5, 6]))

        self.assertTrue(isinstance(d, Die))

    def test_2_adj_weights(self):
        d = Die(np.array([1,2,3,4,5,6]))

        d.adj_weights(4, 10)

        self.assertEqual(d._die_df.index[3], 4)
        self.assertEqual(d._die_df.at[d._die_df.index[3],
'Weights'], 10)
        self.assertTrue(type(d._die_df) == pd.DataFrame)

    def test_3_die_roll(self):
        d = Die(np.array([1,2,3,4,5,6]))
        d.adj_weights(4, 10)

        result = d.die_roll(10)

        self.assertTrue(type(result) == list)

    def test_4_show_df(self):
        d = Die(np.array([1, 2, 3, 4, 5, 6]))
        d.adj_weights(4, 10)
        d.die_roll(10)
```

```
        df = d.show_df()

        self.assertTrue(type(df) == pd.DataFrame)

    def test_5_game_init(self):
        die1 = Die(np.array([1, 2, 3, 4, 5, 6]))
        die2 = Die(np.array([1, 2, 3, 4, 5, 6]))

        g = Game([die1, die2])

        self.assertTrue(isinstance(g, Game))

    def test_6_play(self):
        d = Die(np.array([1, 2, 3, 4, 5, 6]))
        d.adj_weights(4, 10)
        d.die_roll(10)
        g = Game([d, d, d])

        g = g.play(5)

        self.assertTrue(type(g) == pd.DataFrame)

    def test_7_recent_play(self):
        d = Die(np.array([1, 2, 3, 4, 5, 6]))
        d.adj_weights(4, 10)
        d.die_roll(10)
        g = Game([d, d, d])
        g.play(5)

        wide_df = g.recent_play()
        nar_df = g.recent_play('narrow')

        self.assertTrue(type(wide_df) == pd.DataFrame)
        self.assertTrue(type(nar_df) == pd.DataFrame)

    def test_8_analyzer_init(self):
        faces = np.array([1, 2, 3, 4, 5, 6])
        die1 = Die(faces)
        die2 = Die(faces)
        g = Game([die1, die2])

        a = Analyzer(g)

        self.assertTrue(isinstance(a, Analyzer))

    def test_9_jackpot(self):
        d = Die(np.array([1, 2, 3, 4, 5, 6]))
        d.adj_weights(4, 10)
        d.die_roll(10)
        g = Game([d, d, d])
        g.play(5)
        g.recent_play()
        a = Analyzer(g)
```

```
        jkpot = a.jackpot()

        self.assertTrue(type(jkpot) == int)

    def test_10_face_counts(self):
        d = Die(np.array([1, 2, 3, 4, 5, 6]))
        d.adj_weights(4, 10)
        d.die_roll(10)
        g = Game([d, d, d])
        g.play(5)
        g.recent_play()
        a = Analyzer(g)

        face = a.face_counts()

        self.assertTrue(type(face) == pd.DataFrame)

    def test_11_combo_count(self):
        d = Die(np.array([1, 2, 3, 4, 5, 6]))
        d.adj_weights(4, 10)
        d.die_roll(10)
        g = Game([d, d, d])
        g.play(5)
        g.recent_play()
        a = Analyzer(g)

        combo = a.combo_count()

        self.assertTrue(type(combo) == pd.DataFrame)

    def test_12_perm_count(self):
        d = Die(np.array([1, 2, 3, 4, 5, 6]))
        d.adj_weights(4, 10)
        d.die_roll(10)
        g = Game([d, d, d])
        g.play(5)
        g.recent_play()
        a = Analyzer(g)

        perm = a.perm_count()

        self.assertTrue(type(perm) == pd.DataFrame)

if __name__ == '__main__':
    unittest.main(verbosity=3)
```

Unittest Results (3)

Put a copy of the results of running your tests from the command line here.

Again, paste as text using triple backticks.

- All 12 specified methods return OK (3; .25 each).

```
test_10_face_counts (__main__.MonteCarloTestCase) ... ok
test_11_combo_count (__main__.MonteCarloTestCase) ... ok
test_12_perm_count (__main__.MonteCarloTestCase) ... ok
test_1_die_init (__main__.MonteCarloTestCase) ... ok
test_2_adj_weights (__main__.MonteCarloTestCase) ... ok
test_3_die_roll (__main__.MonteCarloTestCase) ... ok
test_4_show_df (__main__.MonteCarloTestCase) ... ok
test_5_game_init (__main__.MonteCarloTestCase) ... ok
test_6_play (__main__.MonteCarloTestCase) ... ok
test_7_recent_play (__main__.MonteCarloTestCase) ... ok
test_8_analyzer_init (__main__.MonteCarloTestCase) ... ok
test_9_jackpot (__main__.MonteCarloTestCase) ... ok

----------------------------------------------------------------------
---
Ran 12 tests in 0.042s

OK
```

Import (1)

Import your module here. This import should refer to the code in your package directory.

- Module successuflly imported (1).

In [1]:
```
import os
os.chdir('/Users/alexaowen/Documents/Grad School/Summer 2023/DS 5001/amo9f_ds510
```

In [5]:
```
from Montecarlo import montecarlo
```

Help Docs (4)

Show your docstring documentation by applying `help()` to your imported module.

- All methods have a docstring (3; .25 each).
- All classes have a docstring (1; .33 each).

In [10]:
```
help (Die)
```

```
Help on class Die in module montecarlo:

class Die(builtins.object)
 |  Die(face)
 |
 |  A class representing a die.
 |
 |  Methods defined here:
 |
```

```
 |  __init__(self, face)
 |      Initializes the die object with a set of faces and default weights.
 |
 |      Args:
 |          face: A numpy array of sides (int or str) of the die.
 |
 |      Raises:
 |          TypeError: If faces argument is not a NumPy array.
 |          ValueError: If the faces array does not have distinct values.
 |
 |  adj_weights(self, face, new_weight)
 |      Changes the weight of a single face in the die.
 |
 |      Args:
 |          face: The face value to be changed.
 |          new_weight: The new weight to set for the specified face.
 |
 |      Raises:
 |          IndexError: If the face is not in the die array.
 |          TypeError: If the new_weight is not numeric (integer or float) or ca
stable as numeric.
 |
 |  die_roll(self, roll=1)
 |      Rolls the die one or more times.
 |
 |      Args:
 |          roll: Number of times the die is rolled. Defaults to 1.
 |
 |      Returns:
 |          list: A Python list of outcomes from the rolls.
 |
 |  show_df(self)
 |      Returns a copy of the private die data frame containing faces and weight
s.
 |
 |      Returns:
 |          pd.DataFrame: A die data frame.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

In [11]:

```
help (Game)
```

```
Help on class Game in module montecarlo:

class Game(builtins.object)
 |  Game(_dice_list)
 |
 |  A class representing a game with similar dice.
 |
 |  Methods defined here:
 |
 |  __init__(self, _dice_list)
 |      Initializes the game object with a list of similar dice.
 |
 |      Args:
 |          _dice_list (list): A list of already instantiated similar dice.
```

```
 |
 |  play(self, roll)
 |      Plays the game by rolling all dice a given number of times.
 |
 |      Args:
 |          roll (int): Number of times the dice should be rolled.
 |
 |      Returns:
 |          pd.DataFrame: A data frame of results with roll number as index and
dice outcomes as columns.
 |
 |  recent_play(self, form='wide')
 |      Shows the results of the most recent play.
 |
 |      Args:
 |          form (str, optional): The form of the data frame to return ('wide' o
r 'narrow'). Defaults to 'wide'.
 |
 |      Returns:
 |          pd.DataFrame: A play data frame in the specified form.
 |
 |      Raises:
 |          ValueError: If the user passes an invalid option for narrow or wide.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

In [12]:

```
help (Analyzer)
```

```
Help on class Analyzer in module montecarlo:

class Analyzer(builtins.object)
 |  Analyzer(game)
 |
 |  A class representing an analyzer for game results.
 |
 |  Attributes:
 |      game (Game): A game object whose results will be analyzed.
 |
 |  Methods defined here:
 |
 |  __init__(self, game)
 |      Initializes the analyzer object with a game.
 |
 |      Args:
 |          game (Game): A game object.
 |
 |      Raises:
 |          ValueError: If the passed value is not a Game object.
 |
 |  combo_count(self)
 |      Computes the distinct combinations of faces rolled, along with their cou
nts.
 |
 |      Returns:
 |          pd.DataFrame: A data frame of results with distinct combinations as
MultiIndex and count as a column.
```

```
 |
 |  face_counts(self)
 |      Computes how many times a given face is rolled in each event.
 |
 |      Returns:
 |          pd.DataFrame: A data frame of results with roll number as index, fac
e values as columns, and count values in the cells.
 |
 |  jackpot(self)
 |      Computes how many times the game resulted in a jackpot (all faces are th
e same).
 |
 |      Returns:
 |          jp_count: A count of the number of jackpots.
 |
 |  perm_count(self)
 |      Computes the distinct permutations of faces rolled, along with their cou
nts.
 |
 |      Returns:
 |          pd.DataFrame: A data frame of results with distinct permutations as
MultiIndex and count as a column.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

README.md File (3)

Provide link to the README.md file of your project's repo.

- Metadata section or info present (1).
- Synopsis section showing how each class is called (1). (All must be included.)
- API section listing all classes and methods (1). (All must be included.)

URL: https://github.com/aowen18/amo9f_ds5100_montecarlo/blob/main/README.md

Successful installation (2)

Put a screenshot or paste a copy of a terminal session where you successfully install your module with pip.

If pasting text, use a preformatted text block to show the results.

- Installed with `pip` (1).
- Successfully installed message appears (1).

```
(base) alexaowen@Alexas-MacBook-Pro ~ % pip install montecarlo
Collecting montecarlo
  Using cached montecarlo-0.1.17-py3-none-any.whl
```

```
Installing collected packages: montecarlo
Successfully installed montecarlo-0.1.17
```

Scenarios

Use code blocks to perform the tasks for each scenario.

Be sure the outputs are visible before submitting.

Scenario 1: A 2-headed Coin (9)

Task 1. Create a fair coin (with faces H and T) and one unfair coin in which one of the faces has a weight of 5 and the others 1.

- Fair coin created (1).
- Unfair coin created with weight as specified (1).

In [20]:
```
import numpy as np
import pandas as pd
import random
```

In [21]:
```
coin = np.array(['H', 'T'])
d1= Die(coin)
d1.show_df()
```

Out[21]:

	Weights
H	1.0
T	1.0

In [22]:
```
d2 = Die (coin)
d2.adj_weights ('H', 5)
d2.show_df()
```

Out[22]:

	Weights
H	5.0
T	1.0

Task 2. Play a game of 1000 flips with two fair dice.

- Play method called correclty and without error (1).

In [23]:
```
g1 = Game([d1,d1])
g1.play(1000)
g1.recent_play()
```

Out[23]:

	Die_0	Die_1
0	T	H
1	H	H
2	H	H
3	T	T
4	T	T
...	...	...
995	H	H
996	H	H
997	T	T
998	H	T
999	H	H

1000 rows × 2 columns

Task 3. Play another game (using a new Game object) of 1000 flips, this time using two unfair dice and one fair die. For the second unfair die, you can use the same die object twice in the list of dice you pass to the Game object.

- New game object created (1).
- Play method called correclty and without error (1).

In [26]:

```
g2 = Game([d1,d2,d2])
g2.play(1000)
g2.recent_play()
```

Out[26]:

	Die_0	Die_1	Die_2
0	T	H	H
1	T	T	H
2	T	H	H
3	H	H	H
4	H	H	H
...	...	...	...
995	H	T	H
996	T	H	H
997	H	H	H
998	H	H	T
999	T	H	H

1000 rows × 3 columns

Task 4. For each game, use an Analyzer object to determine the raw frequency of jackpots — i.e. getting either all Hs or all Ts.

- Analyzer objecs instantiated for both games (1).
- Raw frequencies reported for both (1).

In [27]:
```
a1 = Analyzer(g1)
jp_count = a1.jackpot()
print(jp_count)
```

```
504
```

In [28]:
```
a2 = Analyzer(g2)
jp_count2 = a2.jackpot()
print(jp_count2)
```

```
376
```

Task 5. For each analyzer, compute relative frequency as the number of jackpots over the total number of rolls.

- Both relative frequencies computed (1).

In [31]:
```
jp_freq = jp_count/len(g1.recent_play())*100
print('Jackpot Frequency:', jp_freq,'%')
```

```
Jackpot Frequency: 50.4 %
```

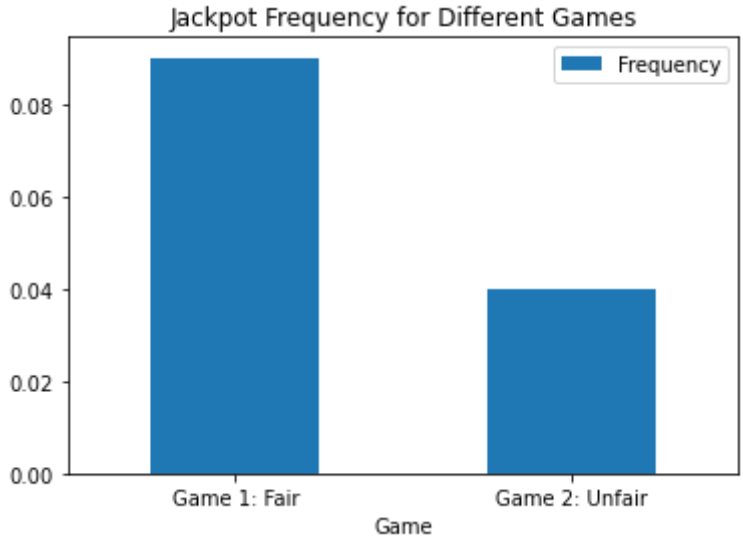
In [32]:
```
jp_freq2 = a2.jackpot()/len(g2.recent_play())*100
print('Jackpot Frequency:',jp_freq2,'%')
```

```
Jackpot Frequency: 37.6 %
```

Task 6. Show your results, comparing the two relative frequencies, in a simple bar chart.

- Bar chart plotted and correct (1).

In [61]:
```
df = pd.DataFrame({'Game':['Game 1: Fair', 'Game 2: Unfair'], 'Frequency':[jp_fr
ax = df.plot.bar(x='Game', y='Frequency', title = 'Jackpot Frequency for Differe
```



Scenario 2: A 6-sided Die (9)

Task 1. Create three dice, each with six sides having the faces 1 through 6.

- Three die objects created (1).

In [36]:
```
die = np.array([1,2,3,4,5,6])
d1 = Die(die)
d2 = Die(die)
d3 = Die(die)
```

Task 2. Convert one of the dice to an unfair one by weighting the face 6 five times more than the other weights (i.e. it has weight of 5 and the others a weight of 1 each).

- Unfair die created with proper call to weight change method (1).

In [37]:
```
d1.adj_weights (6, 5)
d1.show_df()
```

Out[37]:

	Weights
1	1.0
2	1.0
3	1.0
4	1.0
5	1.0
6	5.0

Task 3. Convert another of the dice to be unfair by weighting the face 1 five times more than the others.

- Unfair die created with proper call to weight change method (1).

In [38]:
```
d2.adj_weights (1, 5)
d2.show_df()
```

Out[38]:

	Weights
1	5.0
2	1.0
3	1.0
4	1.0
5	1.0
6	1.0

Task 4. Play a game of 10000 rolls with 5 fair dice.

- Game class properly instantiated (1).
- Play method called properly (1).

In [39]:
```
g1 = Game ([d3,d3,d3,d3,d3])
g1.play(10000)
g1.recent_play()
```

Out[39]:

	Die_0	Die_1	Die_2	Die_3	Die_4
0	2	1	6	3	1
1	6	6	1	5	2
2	4	1	3	6	5
3	4	4	2	2	6
4	1	5	6	1	1
...	...	...	...	...	...
9995	4	5	1	2	5
9996	6	6	3	1	1
9997	5	5	4	5	1
9998	3	2	2	5	2
9999	5	5	5	2	1

10000 rows × 5 columns

Task 5. Play another game of 10000 rolls, this time with 2 unfair dice, one as defined in steps #2 and #3 respectively, and 3 fair dice.

- Game class properly instantiated (1).
- Play method called properly (1).

In [40]:
```
g2 = Game ([d1,d2,d3,d3,d3])
g2.play(10000)
g2.recent_play()
```

Out[40]:

	Die_0	Die_1	Die_2	Die_3	Die_4
0	6	6	5	3	5
1	6	3	5	4	6
2	3	1	6	1	6
3	3	4	2	2	2
4	1	1	5	6	2
...	...	...	...	...	...
9995	2	2	1	2	6
9996	6	4	3	2	2
9997	4	1	2	5	2
9998	3	1	5	6	2
9999	5	1	4	3	4

10000 rows × 5 columns

Task 6. For each game, use an Analyzer object to determine the relative frequency of jackpots and show your results, comparing the two relative frequencies, in a simple bar chart.

- Jackpot methods called (1).
- Graph produced (1).

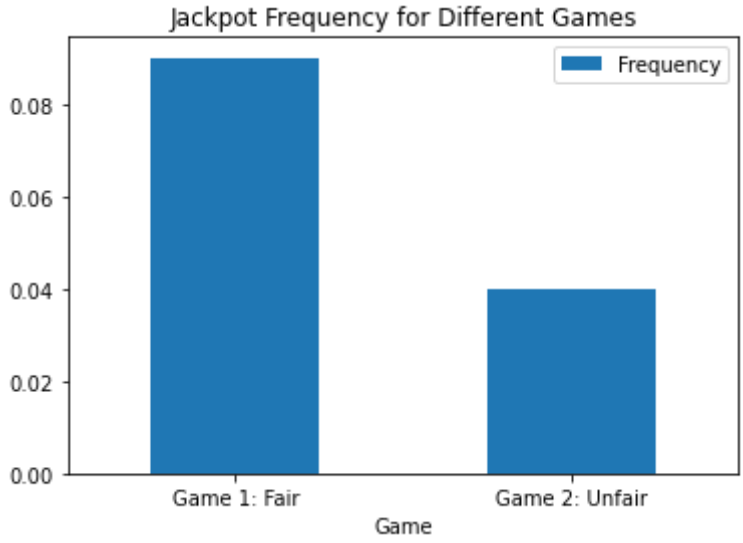
In [42]:
```
a1 = Analyzer(g1)
print(a1.jackpot())
jp_freq = a1.jackpot()/len(g1.recent_play())*100
print('Jackpot Frequency:', jp_freq,'%')
```

```
9
Jackpot Frequency: 0.09 %
```

In [43]:
```
a2 = Analyzer(g2)
print(a2.jackpot())
jp_freq2 = a2.jackpot()/len(g2.recent_play())*100
print('Jackpot Frequency:',jp_freq2,'%')
```

```
4
Jackpot Frequency: 0.04 %
```

In [62]:
```
df = pd.DataFrame({'Game':['Game 1: Fair', 'Game 2: Unfair'], 'Frequency':[jp_fr
ax = df.plot.bar(x='Game', y='Frequency', title = 'Jackpot Frequency for Differe
```



Scenario 3: Letters of the Alphabet (7)

Task 1. Create a "die" of letters from A to Z with weights based on their frequency of usage as found in the data file `english_letters.txt` . Use the frequencies (i.e. raw counts) as weights.

- Die correctly instantiated with source file data (1).
- Weights properly applied using weight setting method (1).

In [45]:
```
weights = pd.read_csv('english_letters.txt', sep=" ", header= None)
scrabble = pd.read_csv('scrabble_words.txt', header= None)
scrabble = scrabble.rename(columns = {0: 'words'})
```

In [46]:
```
letters = weights[0].tolist()
weights = weights[1].tolist()

d= Die(np.array(letters))

for i, face in enumerate(letters):
    d.adj_weights(face, weights[i])

d.show_df()
```

Out[46]:

	Weights
E	529117365.0
T	390965105.0
A	374061888.0
O	326627740.0
I	320410057.0
N	313720540.0
S	294300210.0

	Weights
R	277000841.0
H	216768975.0
L	183996130.0
D	169330528.0
C	138416451.0
U	117295780.0
M	110504544.0
F	95422055.0
G	91258980.0
P	90376747.0
W	79843664.0
Y	75294515.0
B	70195826.0
V	46337161.0
K	35373464.0
J	9613410.0
X	8369915.0
Z	4975847.0
Q	4550166.0

Task 2. Play a game involving 4 of these dice with 1000 rolls.

- Game play method properly called (1).

In [47]:
```
g1 = Game ([d,d,d,d])
g1.play(1000)
g1.recent_play()
```

Out[47]:

	Die_0	Die_1	Die_2	Die_3
0	A	N	I	T
1	A	A	M	E
2	R	I	E	S
3	V	G	O	I
4	C	G	I	D
...	...	...	...	...
995	L	H	A	N
996	D	O	L	W
997	L	U	A	L

	Die_0	Die_1	Die_2	Die_3
998	L	A	L	T
999	E	S	I	R

1000 rows × 4 columns

Task 3. Determine how many permutations in your results are actual English words, based on the vocabulary found in `scrabble_words.txt` .

- Use permutation method (1).
- Get count as difference between permutations and vocabulary (1).

In [48]:
```
a1 = Analyzer (g1)
perm1 = a1.perm_count()
perm1
```

Out[48]:

				count
Die_0	**Die_1**	**Die_2**	**Die_3**	
R	**S**	**E**	**A**	2
E	**T**	**T**	**E**	2
	L	**O**	**A**	2
	U	**A**	**E**	2
T	**S**	**E**	**E**	2
...	**...**	**...**	**...**	...
G	**N**	**E**	**I**	1
		H	**H**	1
		S	**T**	1
			Y	1
Y	**U**	**T**	**A**	1

992 rows × 1 columns

In [50]:
```
l1 = []
for e in list(perm1.index):
    l1.append(''.join(e))

df1 = pd.DataFrame({'perm':l1})
df1
```

Out[50]:

	perm
0	RSEA
1	ETTE
2	ELOA

	perm
3	EUAE
4	TSEE
...	...
987	GNEI
988	GNHH
989	GNST
990	GNSY
991	YUTA

992 rows × 1 columns

In [53]:
```
words_both1 = df1.perm.isin(scrabble.words).sum()
print ('Words in Common:', words_both1)
```

```
Words in Common: 56
```

Task 4. Repeat steps #2 and #3, this time with 5 dice. How many actual words does this produce? Which produces more?

- Successfully repreats steps (1).
- Identifies parameter with most found words (1).

In [54]:
```
g2 = Game ([d,d,d,d,d])
g2.play(1000)
g2.recent_play()
```

Out[54]:

	Die_0	Die_1	Die_2	Die_3	Die_4
0	I	C	T	Y	H
1	U	R	D	N	A
2	A	O	N	I	O
3	I	L	E	R	A
4	S	N	H	S	R
...	...	...	...	...	...
995	S	N	N	O	F
996	O	R	M	T	E
997	F	R	H	P	S
998	H	H	A	A	E
999	A	O	R	S	R

1000 rows × 5 columns

In [55]:
```
a2 = Analyzer (g2)
perm2 = a2.perm_count()
perm2
```

Out[55]:

					count
Die_0	Die_1	Die_2	Die_3	Die_4	
A	A	D	H	U	1
O	R	L	F	T	1
	N	I	N	D	1
		N	S	U	1
		O	B	L	1
...	...	...	...	...	...
H	E	E	C	T	1
		G	H	I	1
		M	A	R	1
		O	A	A	1
Z	N	R	S	D	1

1000 rows × 1 columns

In [56]:
```
l2 = []
for e in list(perm2.index):
    l2.append(''.join(e))

df2 = pd.DataFrame({'perm':l2})
df2
```

Out[56]:

	perm
0	AADHU
1	ORLFT
2	ONIND
3	ONNSU
4	ONOBL
...	...
995	HEECT
996	HEGHI
997	HEMAR
998	HEOAA
999	ZNRSD

1000 rows × 1 columns

In [57]:
```
words_both2 = df2.perm.isin(scrabble.words).sum()
print ('Words in Common:', words_both2)
```

```
Words in Common: 7
```

Submission

When finished completing the above tasks, save this file to your local repo (and within your project), and them push it to your GitHub repo.

Then convert this file to a PDF and submit it to GradeScope according to the assignment instructions in Canvas.