Final Project Report

- Class: DS 5100
- Student Name: Sydney Mathiason
- Student Net ID: qex8sd
- This URL: https://github.com/sydneymathiason/qex8sd_ds5100_montecarlo/blob/main/FinalProjectTemplate.ipyn

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/sydneymathiason/qex8sd_ds5100_montecarlo/tree/main

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

class Die:
    """
    Represents a Die object that simulates a fair or weighted die.

    Attributes
    ----------
    faces : numpy.ndarray
        An array representing the possible faces of the die.
```

```
    Raises
    ------
    TypeError
        If the faces parameter is not a NumPy array.
    ValueError
        If the faces are not all unique.

    Note
    ----
    The Die object starts with equal weights for all faces.

    """
    def __init__(self, faces):
        """
        Create a Die object with specified faces and equal weights.

        Parameters
        ----------
        faces : numpy.ndarray
            An array representing the possible faces of the die.

        Raises
        ------
        TypeError
            If the faces parameter is not a NumPy array.
        ValueError
            If the faces are not all unique.
        """
        self.faces = faces
        if not isinstance(self.faces, np.ndarray):
            raise TypeError("The faces parameter must be a NumPy array.")
        if len(faces) != len(np.unique(faces)):
            raise ValueError("Faces are not all unique")

        self._die_state = pd.DataFrame({"weights": [1] * len(faces)},
index=faces)

    def change_weight(self, face, new_weight):
        """
        Change the weight of a specific die face.

        Parameters
        ----------
        face : int
            The face whose weight needs to be changed.
        new_weight : int or float
            The new weight for the specified face.

        Raises
        ------
        IndexError
            If the specified face is not a valid face of the die.
        TypeError
            If the new_weight parameter is not an int or float.
```

```
        """
        if face not in self._die_state.index:
            raise IndexError("Invalid face value.")

        if not (isinstance(new_weight, (int, float)) or
str(new_weight).isnumeric()):
            raise TypeError("Invalid weight type.")

        self._die_state.at[face, "weights"] = new_weight

    def roll_die(self, rolls=1):
        """
        Roll the die a given number of times and return the results in a list.

        Parameters
        ----------
        rolls : int, optional
            The number of times to roll the die. Default is 1.

        Returns
        -------
        list
            A list of outcomes obtained from rolling the die.
        """
        outcomes = np.random.choice(self._die_state.index, rolls,
p=self._die_state["weights"] / sum(self._die_state["weights"]))
        return outcomes.tolist()

    def show_state(self):
        """
        Show the current faces and weights of the die object.

        Returns
        -------
        pandas.DataFrame
            A DataFrame containing the faces and their corresponding weights.
        """
        return self._die_state.copy()



class Game:
    """
    Represents a game involving one or multiple dice.

    Parameters
    ----------
    list_of_die : list
        A list of Die objects used in the game.

    """

    def __init__(self, list_of_die):
        """
        Initialize a game object with a list of Die objects.
```

```
        Parameters
        ----------
        list_of_die : list
            A list of Die objects used in the game.
        """
        self._dice = list_of_die
        self._play_data = None

    def play(self, times):
        """
        Simulate playing the game by rolling the dice a given number of times.

        Parameters
        ----------
        times : int
            The number of times to roll the dice.
        """
        play_results = {i: die.roll_die(times) for i, die in
enumerate(self._dice)}
        self._play_data = pd.DataFrame(play_results)
        self._play_data.index.name = "roll_number"

    def show_result(self, dftype="wide"):
        """
        Return the results from playing the game in a specified format.

        Parameters
        ----------
        dftype : str, optional
            The format of the results. Options: "wide" (default) or "narrow".

        Returns
        -------
        pandas.DataFrame
            The results in the specified format.

        Raises
        ------
        ValueError
            If dftype is not "narrow" or "wide".
        """
        if dftype == "wide":
            out = self._play_data.copy()
        elif dftype == "narrow":
            out = pd.DataFrame(self._play_data.copy().stack())
        else:
            raise ValueError("dftype needs to be narrow or wide")

        return out


class Analyzer:
    """
    Represents an Analyzer class for analyzing results from a Game object.
```

```
    Attributes:
        game (Game): The Game object to analyze.

    """

    def __init__(self, game):
        """
        Create an Analyzer object with a Game object.

        Parameters
        ----------
        game : Game
            The Game object to be analyzed.

        Raises
        ------
        ValueError
            If the input is not a valid Game object.
        """
        if not isinstance(game, Game):
            raise ValueError("The input must be a Game object")
        self._game = game

    def jackpot(self):
        """
        Return the number of jackpots in the game results.

        Returns
        -------
        int
            The count of jackpots.
        """
        return
pd.DataFrame(self._game.show_result().eq(self._game.show_result().iloc[:, 0],
axis=0

).all(1).astype(int)).sum().item()

    def face_counts(self):
        """
        Return the face counts for all rolls in the game results.

        Returns
        -------
        pandas.DataFrame
            A DataFrame containing face counts for each roll.
        """
        return self._game.show_result().apply(pd.Series.value_counts,
axis=1).fillna(0).astype(int)

    def combo_count(self):
        """
        Return the combination counts of faces in the game results.
```

```
        Returns
        -------
        pandas.DataFrame
            A DataFrame containing combination counts.
        """
        df1 = self.face_counts()
        cols = df1.columns.to_list()
        mylist = []
        for i in range(len(df1)):
            newlist = []
            for col in cols:
                if df1.iloc[i][col] > 0:
                    for x in range(df1.iloc[i][col]):
                        newlist.append(col)

            mylist.append(newlist)
        return pd.DataFrame(pd.DataFrame(mylist,
columns=range(len(self._game._dice))

).groupby(list(range(len(self._game._dice)))).value_counts()).rename(columns=
{0: 'count'})

    def permutation_count(self):
        """
        Return the permutation counts of combinations in the game results.

        Returns
        -------
        pandas.DataFrame
            A DataFrame containing permutation counts.
        """
        permutations =
self._game.show_result().groupby(list(range(len(self._game._dice))))
        permutation_counts = permutations.value_counts()
        return pd.DataFrame({"count": permutation_counts})
```

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.montecarlo import Die, Game, Analyzer

class TestDie(unittest.TestCase):
    def test_d__init__(self):
        faces = np.array([1, 2, 3, 4, 5, 6])
```

```
        die = Die(faces)
        self.assertIsInstance(die, Die)

    def test_change_weight(self):
        faces = np.array([1, 2, 3, 4, 5, 6])
        die = Die(faces)
        die.change_weight(1, 0.5)
        self.assertEqual(die.show_state().loc[1, "weights"], 0.5)

    def test_roll(self):
        faces = np.array([1, 2, 3, 4, 5, 6])
        die = Die(faces)
        outcomes = die.roll_die(10)
        self.assertEqual(len(outcomes), 10)

    def test_show_state(self):
        faces = np.array([1, 2, 3, 4, 5, 6])
        die = Die(faces)
        self.assertIsInstance(die.show_state(), pd.DataFrame)

    def setUp(self):
        faces1 = np.array([1, 2, 3, 4, 5, 6])
        faces2 = np.array(["H", "T"])
        self.die1 = Die(faces1)
        self.die2 = Die(faces2)
        self.game1 = Game([self.die1, self.die1])
        self.game2 = Game([self.die1, self.die2, self.die2])
        self.analyzer1 = Analyzer(self.game1)
        self.analyzer2 = Analyzer(self.game2)

    def test_g__init__(self):
        self.game1.play(10)
        self.assertIsInstance(self.game1, Game)

        self.game2.play(10)
        self.assertIsInstance(self.game2, Game)

    def test_play(self):
        self.game1.play(10)
        self.assertEqual(self.game1._play_data.index.name, "roll_number")

        self.game2.play(10)
        self.assertEqual(self.game2._play_data.index.name, "roll_number")

    def test_show_results(self):
        self.game1.play(10)
        self.assertEqual(self.game1.show_result(dftype="wide").shape, (10, 2))

        self.game2.play(10)
        self.assertEqual(self.game2.show_result(dftype="narrow").shape, (30,
1))

    def test_a__init__(self):
        self.game1.play(100)
        self.assertIsInstance(self.analyzer1, Analyzer)
```

```
        self.game2.play(100)
        self.assertIsInstance(self.analyzer2, Analyzer)

    def test_jackpot(self):
        self.game1.play(100)
        self.assertIsInstance(self.analyzer1.jackpot(), int)

        self.game2.play(100)
        self.assertIsInstance(self.analyzer2.jackpot(), int)

    def test_face_counts_per_roll(self):
        self.game1.play(100)
        self.assertIsInstance(self.analyzer1.face_counts(), pd.DataFrame)

        self.game2.play(100)
        self.assertIsInstance(self.analyzer2.face_counts(), pd.DataFrame)

    def test_combo_count(self):
        self.game1.play(100)
        self.assertIsInstance(self.analyzer1.combo_count(), pd.DataFrame)

        self.game2.play(100)
        self.assertIsInstance(self.analyzer2.combo_count(), pd.DataFrame)

    def test_permutation_count(self):
        self.game1.play(100)
        self.assertIsInstance(self.analyzer1.permutation_count(),
pd.DataFrame)

        self.game2.play(100)
        self.assertIsInstance(self.analyzer2.permutation_count(),
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_a__init__ (__main__.TestDie) ... ok
test_change_weight (__main__.TestDie) ... ok
test_combo_count (__main__.TestDie) ... ok
test_d__init__ (__main__.TestDie) ... ok
test_face_counts_per_roll (__main__.TestDie) ... ok
test_g__init__ (__main__.TestDie) ... ok
test_jackpot (__main__.TestDie) ... ok
test_permutation_count (__main__.TestDie) ... ok
test_play (__main__.TestDie) ... ok
```

```
test_roll (__main__.TestDie) ... ok
test_show_results (__main__.TestDie) ... ok
test_show_state (__main__.TestDie) ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.237s

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 montecarlo.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 [2]:
```
help(montecarlo.montecarlo)
```

```
Help on module montecarlo.montecarlo in montecarlo:

NAME
    montecarlo.montecarlo

CLASSES
    builtins.object
        Analyzer
        Die
        Game

    class Analyzer(builtins.object)
     |  Analyzer(game)
     |
     |  Represents an Analyzer class for analyzing results from a Game object.
     |
     |  Attributes:
     |      game (Game): The Game object to analyze.
     |
     |  Methods defined here:
     |
     |  __init__(self, game)
     |      Create an Analyzer object with a Game object.
     |
     |      Parameters
     |      ----------
     |      game : Game
     |          The Game object to be analyzed.
     |
     |      Raises
     |      ------
     |      ValueError
     |          If the input is not a valid Game object.
     |
     |  combo_count(self)
     |      Return the combination counts of faces in the game results.
     |
     |      Returns
     |      -------
     |      pandas.DataFrame
     |          A DataFrame containing combination counts.
     |
     |  face_counts(self)
     |      Return the face counts for all rolls in the game results.
     |
     |      Returns
     |      -------
     |      pandas.DataFrame
     |          A DataFrame containing face counts for each roll.
     |
     |  jackpot(self)
     |      Return the number of jackpots in the game results.
     |
     |      Returns
     |      -------
     |      int
     |          The count of jackpots.
     |
     |  permutation_count(self)
     |      Return the permutation counts of combinations in the game results.
     |
     |      Returns
```

```
 |      -------
 |      pandas.DataFrame
 |          A DataFrame containing permutation counts.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Die(builtins.object)
 |  Die(faces)
 |
 |  Represents a Die object that simulates a fair or weighted die.
 |
 |  Attributes
 |  ----------
 |  faces : numpy.ndarray
 |      An array representing the possible faces of the die.
 |
 |  Raises
 |  ------
 |  TypeError
 |      If the faces parameter is not a NumPy array.
 |  ValueError
 |      If the faces are not all unique.
 |
 |  Note
 |  ----
 |  The Die object starts with equal weights for all faces.
 |
 |  Methods defined here:
 |
 |  __init__(self, faces)
 |      Create a Die object with specified faces and equal weights.
 |
 |      Parameters
 |      ----------
 |      faces : numpy.ndarray
 |          An array representing the possible faces of the die.
 |
 |      Raises
 |      ------
 |      TypeError
 |          If the faces parameter is not a NumPy array.
 |      ValueError
 |          If the faces are not all unique.
 |
 |  change_weight(self, face, new_weight)
 |      Change the weight of a specific die face.
 |
 |      Parameters
 |      ----------
 |      face : int
 |          The face whose weight needs to be changed.
 |      new_weight : int or float
 |          The new weight for the specified face.
 |
 |      Raises
```

```
 |      ------
 |      IndexError
 |          If the specified face is not a valid face of the die.
 |      TypeError
 |          If the new_weight parameter is not an int or float.
 |
 |  roll_die(self, rolls=1)
 |      Roll the die a given number of times and return the results in a list.
 |
 |      Parameters
 |      ----------
 |      rolls : int, optional
 |          The number of times to roll the die. Default is 1.
 |
 |      Returns
 |      -------
 |      list
 |          A list of outcomes obtained from rolling the die.
 |
 |  show_state(self)
 |      Show the current faces and weights of the die object.
 |
 |      Returns
 |      -------
 |      pandas.DataFrame
 |          A DataFrame containing the faces and their corresponding weights.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

class Game(builtins.object)
 |  Game(list_of_die)
 |
 |  Represents a game involving one or multiple dice.
 |
 |  Parameters
 |  ----------
 |  list_of_die : list
 |      A list of Die objects used in the game.
 |
 |  Methods defined here:
 |
 |  __init__(self, list_of_die)
 |      Initialize a game object with a list of Die objects.
 |
 |      Parameters
 |      ----------
 |      list_of_die : list
 |          A list of Die objects used in the game.
 |
 |  play(self, times)
 |      Simulate playing the game by rolling the dice a given number of times.
 |
 |      Parameters
 |      ----------
 |      times : int
```

```
 |          The number of times to roll the dice.
 |
 |  show_result(self, dftype='wide')
 |      Return the results from playing the game in a specified format.
 |
 |      Parameters
 |      ----------
 |      dftype : str, optional
 |          The format of the results. Options: "wide" (default) or "narrow".
 |
 |      Returns
 |      -------
 |      pandas.DataFrame
 |          The results in the specified format.
 |
 |      Raises
 |      ------
 |      ValueError
 |          If dftype is not "narrow" or "wide".
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

FILE
    /Users/sydneymathiason/Documents/MSDS/Summer/DS5100/qex8sd_ds5100_montecarlo/montecarlo/montecarlo.py
```

`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/sydneymathiason/qex8sd_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) sydneymathiason@Sydneys-MacBook-Pro qex8sd_ds5100_montecarlo % pip install -e .
Obtaining file:///Users/sydneymathiason/Documents/MSDS/Summer/DS5100/qex8sd_ds5100_montecarlo
  Preparing metadata (setup.py) ... done
Installing collected packages: montecarlo
  Running setup.py develop for montecarlo
Successfully installed montecarlo-1.0.0
```

Scenarios

Use code blocks to perform the tasks for each scenario.

Be sure the outputs are visible before submitting.

In [3]:
```
import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
import seaborn as sns
from montecarlo.montecarlo import *
```

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 [4]:
```
faces = np.array(["H", "T"])
die1 = Die(faces)
die2 = Die(faces)
die2.change_weight("H", 5)
```

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

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

In [5]:
```
rolls = 1000
game1 = Game([die1, die1])
game1.play(rolls)
```

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 [6]:
```
game2 = Game([die1, die2, die2])
game2.play(rolls)
```

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 [7]:
```
A1 = Analyzer(game1)
A2 = Analyzer(game2)
```

In [8]:
```
jackpot1 = A1.jackpot()
jackpot1
```

Out[8]:
```
502
```

In [9]:
```
jackpot2 = A2.jackpot()
jackpot2
```

Out[9]:
```
343
```

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 [10]:
```
freq1 = jackpot1/rolls
freq2 = jackpot2/rolls
freq1, freq2
```

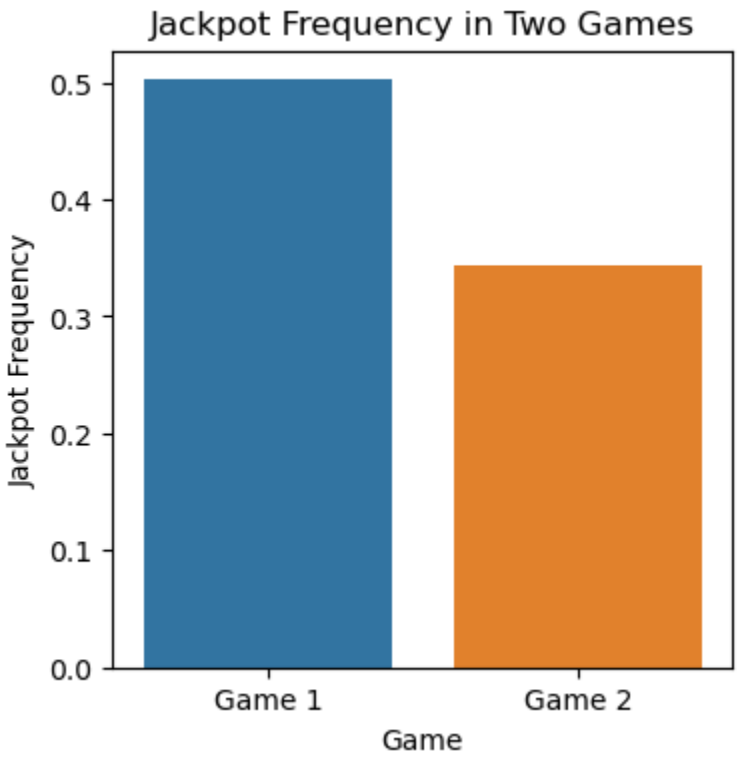
Out[10]:
```
(0.502, 0.343)
```

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

- Bar chart plotted and correct (1).

In [11]:
```
bar_dict = {"Game":["Game 1", "Game 2"], "Jackpot Frequency":[freq1, freq2]}
bar_df = pd.DataFrame(bar_dict)
plt.figure(figsize=(4,4))
sns.barplot(x="Game", y="Jackpot Frequency", data=bar_df)
plt.title("Jackpot Frequency in Two Games")
```

Out[11]:
```
Text(0.5, 1.0, 'Jackpot Frequency in Two Games')
```



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 [12]:
```
die1 = Die(np.array([1,2,3,4,5,6]))
die2 = Die(np.array([1,2,3,4,5,6]))
die3 = Die(np.array([1,2,3,4,5,6]))
```

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 [13]:
```
die1.change_weight(6, 5)
```

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 [14]:
```
die2.change_weight(1, 5)
```

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

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

In [15]:
```
rolls = 10000
game1 = Game([die3, die3, die3, die3, die3])
game1.play(rolls)
```

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 [16]:
```
game2 = Game([die1, die2, die3, die3, die3])
game2.play(rolls)
```

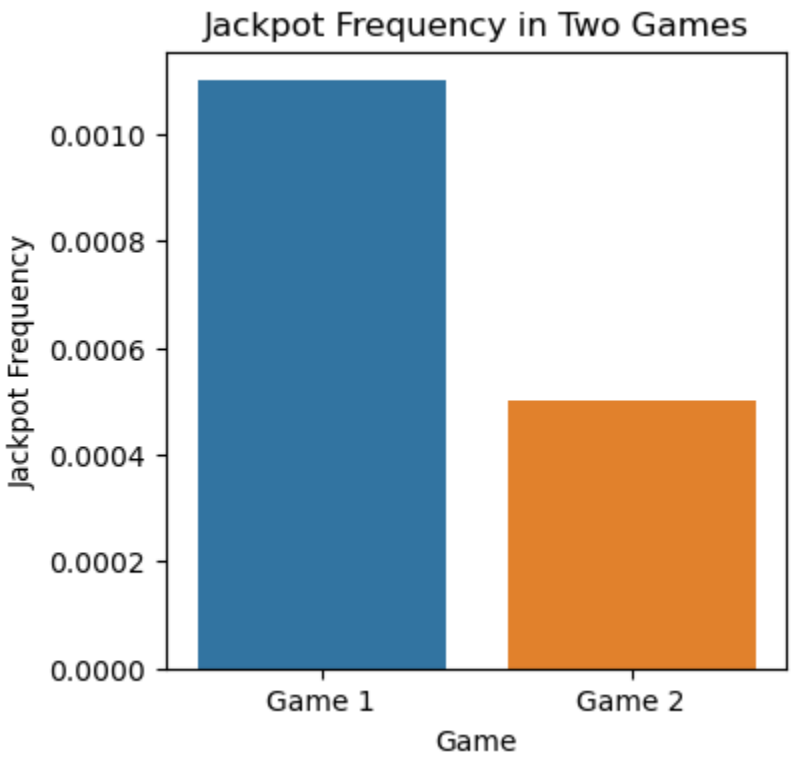
Task 6. For each game, use an Analyzer object to determine the relative frequency of jackpots and show your results, comparing the two relative frequencies, in a simple bar chart.

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

In [17]:
```
A1 = Analyzer(game1)
jackpot1 = A1.jackpot()
freq1 = jackpot1/rolls
A2 = Analyzer(game2)
jackpot2 = A2.jackpot()
freq2 = jackpot2/rolls
```

In [18]:
```
bar_dict = {"Game":["Game 1", "Game 2"], "Jackpot Frequency":[freq1, freq2]}
bar_df = pd.DataFrame(bar_dict)
plt.figure(figsize=(4,4))
sns.barplot(x="Game", y="Jackpot Frequency", data=bar_df)
plt.title("Jackpot Frequency in Two Games")
```

Out[18]:
```
Text(0.5, 1.0, 'Jackpot Frequency in Two Games')
```



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 [19]:
```
df = pd.read_csv('english_letters.txt', header=None, delimiter = " ")
letters = df[0].to_numpy()
die = Die(letters)
for i in letters:
    die.change_weight(i, df[df[0]==i][1].item())
die.show_state()
```

Out[19]:

	weights
E	529117365
T	390965105
A	374061888
O	326627740
I	320410057
N	313720540
S	294300210
R	277000841
H	216768975
L	183996130
D	169330528
C	138416451
U	117295780
M	110504544
F	95422055
G	91258980
P	90376747
W	79843664
Y	75294515
B	70195826
V	46337161
K	35373464
J	9613410
X	8369915
Z	4975847
Q	4550166

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

- Game play method properly called (1).

In [20]:
```
game = Game([die, die, die, die])
game.play(1000)
```

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 [21]:
```
words = pd.read_csv('scrabble_words.txt', header=None)
four = set(words[words[0].apply(lambda x: len(str(x))==4)][0].to_list())
len(four)
```

Out[21]:
```
5637
```

In [22]:
```
A = Analyzer(game)
perm4 = A.permutation_count()
```

In [23]:
```
perm4['word'] = perm4.index.get_level_values(0)+perm4.index.get_level_values(1)\
+perm4.index.get_level_values(2)+perm4.index.get_level_values(3)
word = set(perm4["word"].to_list())
```

In [24]:
```
print(four.intersection(word))
```

```
{'LENT', 'FIER', 'TOSA', 'WHIR', 'YOGA', 'OOTS', 'LUCE', 'COIR', 'DEAR', 'NACH', 'SOUR',
'LEES', 'METE', 'ROTL', 'RAID', 'GALS', 'DEAN', 'EINA', 'ONES', 'PERE', 'LATS', 'ALBS',
'NESH', 'PESO', 'HALE', 'HARE', 'NOSH', 'ROMP', 'HIOI', 'NEWT', 'NOWS', 'WOOS', 'REAL',
'LARN', 'DAWD', 'EASE', 'RHEA', 'MEES', 'CASE', 'BIER', 'DURA', 'DAUD', 'TAMP', 'HASH',
'CARN', 'CIDS', 'ANTI', 'NARK', 'LASS', 'MUNT', 'TOGE', 'DIGS', 'HENS', 'SEEN', 'RENT',
'TEME', 'AILS', 'RITT', 'RILE', 'LEYS', 'REEL', 'RAUN', 'ARTI'}
```

In [25]:
```
len(four.intersection(word))
```

Out[25]:
```
63
```

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

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

In [26]:
```
five = set(words[words[0].apply(lambda x: len(str(x))==5)][0].to_list())
len(five)
```

Out[26]:
```
12972
```

In [27]:
```
game1 = Game([die, die, die, die, die])
game1.play(1000)
A1 = Analyzer(game1)
perm5 = A1.permutation_count()
perm5
```

Out[27]:

					count
0	1	2	3	4	
A	A	D	C	A	1
		H	A	E	1
		L	T	P	1
		R	M	U	1
		T	A	C	1
...	...	...	...	...	...
Y	O	D	L	L	1
	R	R	A	E	1
	S	E	D	E	1
	W	T	D	I	1
	Y	N	U	A	1

998 rows × 1 columns

In [28]:
```
perm5['word'] = perm5.index.get_level_values(0)+perm5.index.get_level_values(1) \
+perm5.index.get_level_values(2)+perm5.index.get_level_values(3)+perm5.index.get_level_v
word = set(perm5["word"].to_list())
perm5
```

Out[28]:

					count	word
0	1	2	3	4		
A	A	D	C	A	1	AADCA
		H	A	E	1	AAHAE
		L	T	P	1	AALTP
		R	M	U	1	AARMU
		T	A	C	1	AATAC
...	...	...	...	...	...	...
Y	O	D	L	L	1	YODLL
	R	R	A	E	1	YRRAE
	S	E	D	E	1	YSEDE
	W	T	D	I	1	YWTDI
	Y	N	U	A	1	YYNUA

998 rows × 2 columns

In [29]:
```
five.intersection(word)
```

Out[29]:
```
{'BENNI', 'CHANT', 'CHARS', 'LIGAN', 'PLAIT', 'SNARE', 'TATIE', 'TYNED'}
```

In [30]:
```
len(five.intersection(word))
```

Out[30]: 8

4 letter words generate a higher percentage of English words

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.