Final Project Report

- Class: DS 5100
- Student Name: Addison Gambhir
- Student Net ID: ajg7pk
- This URL: a URL to the notebook source of this document

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/AddisonGambhir/ajg7pk_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 pandas as pd
import numpy as np
import random

class Die:
    """
    Fibricates a 'die' object that has specified faces and weights.

    Attributes:
        _die_data (pandas.DataFrame): This is the dataframe that stores
face values and weights.
    """
```

```
    def __init__(self, faces: np.ndarray):
        """
        Initializes the die object with specified 'faces'.

        Args:
            faces (numpy.ndarray): An array containing distinct face
values of the die.

        Raises:
            TypeError: If faces is not a NumPy array.
            ValueError: If faces do not have distinct values.
        """
        if not isinstance(faces, np.ndarray):
            raise TypeError("Input must be a NumPy array.")

        if len(faces) != len(np.unique(faces)):
            raise ValueError("Faces must have distinct values.")

        weights = np.ones_like(faces, dtype=float)
        data = {'Weights': weights}
        if np.issubdtype(faces.dtype, np.number):
            self._die_data = pd.DataFrame(data, index=faces)
        else:
            self._die_data = pd.DataFrame(data, index=faces.astype(str))

    def set_weight(self, face, weight):
        """
        This controls the weight for a specific face value.

        Arguments:
            face: The face value for which the weight is to be set.
            weight: The weight value assigned.

        Raises:
            IndexError: If the face value is invalid.
            TypeError: If the weight is not a numeric value.
        """
        if face not in self._die_data.index:
            raise IndexError("Invalid face value.")
        if not (isinstance(weight, (int, float)) or
np.issubdtype(type(weight), np.number)):
            raise TypeError("Weight must be a numeric value or castable
as numeric.")

        self._die_data.loc[face, 'Weights'] = weight

    def roll(self, num_rolls: int = 1) -> list:
        """
        This method simulates rolling of the die multiple times.

        Argument:
            num_rolls (int): The number of times the die is rolled.
```

```
        Returns:
            list: A list of outcomes from rolling the die.
        """
        outcomes = random.choices(self._die_data.index,
weights=self._die_data['Weights'], k=num_rolls)
        return outcomes

    def get_die_state(self) -> pd.DataFrame:
        """
        This will return a copy of the die's data frame.

        Returns:
            pandas.DataFrame: A copy of the die data frame.
        """
        return self._die_data.copy()


class Game:
    """
    The game class contains a game consisting of rolling multiple dice.

    Attributes:
        dice: A list of Die objects representing the dice in the game.
        play_data: Data frame that stores the results of the most recent
play.
    """
    def __init__(self, dice):
        """
        This initializes the Game class with a list of similar dice.
        This also checks if the list contains Die objects
        Argusment:
            dice (list): A list of already instantiated similar dice (Die
objects).
        """
        # Check if the list contains Die objects
        if not all(isinstance(die, Die) for die in dice):
            raise ValueError('All elements in the list must be instances
of the Die class.')

        # Check if all dice have the same faces
        faces = set(dice[0].get_die_state().index)
        if not all(set(die.get_die_state().index) == faces for die in
dice):
            raise ValueError('All dice must have the same faces.')

        self.dice = dice
        self.play_data = pd.DataFrame()  # Private data frame to store
play results

    def play(self, times):
        """
        This method simulates playing the game by rolling the dice a
```

```
specified number of times.

        Argsuments:
            times (int): The number of times instructed to roll the dice.
        """
        rolls = []
        for i in range(times):
            roll_result = []
            for die in self.dice:
                roll = die.roll()[0]
                roll_result.append(roll)
            rolls.append(roll_result)

        columns = [f"Die_{i}" for i in range(1, len(self.dice) + 1)]
        self.play_data = pd.DataFrame(rolls, columns=columns)

    def show_results(self, form='wide'):
        """
        This method displays the results of the most recent play.

        Args:
            Format options: Options include: 'wide' (default) or 'narrow'

        Returns:
            pandas.DataFrame: The play results data frame.

        Raises:
            ValueError: If an invalid option is provided for the format.
        """
        if form == 'wide':
            return self.play_data.copy()
        elif form == 'narrow':
            narrow_data = pd.DataFrame()
            for idx, col in enumerate(self.play_data.columns):
                rolls = self.play_data[col].tolist()
                roll_numbers = list(range(1, len(rolls) + 1))
                multi_index = pd.MultiIndex.from_tuples([(r, col) for r
in roll_numbers], names=['Roll', 'Die'])
                df = pd.DataFrame(rolls, index=multi_index, columns=
['Outcome'])
                narrow_data = pd.concat([narrow_data, df])
            return narrow_data
        else:
            raise ValueError("Invalid option for 'form'. Please choose
'wide' or 'narrow'.")

class Analyzer:
    """
    Analyzes the results of a single game and computes various
descriptive properties.

    Attributes:
        game (Game): A Game object whose results will be analyzed.
```

```
    """

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

        Args:
            game (Game): A Game object whose results will be analyzed.

        Raises:
            ValueError: If the input is not a Game object.
        """
        if not isinstance(game, Game):
            raise ValueError("Input must be a Game object.")

        self.game = game

    def jackpot(self):
        """
        Jackpot computes the number of jackpot occurrences in the game.

        Returns:
            int: The number of jackpots in the game.
        """
        jackpot_count = 0
        results = self.game.show_results('wide')
        for i, row in results.iterrows():
            if all(value == row.iloc[0] for value in row):
                jackpot_count += 1
        return jackpot_count

    def face_counts_per_roll(self):
        """
        Counts how many times a given face is rolled in each event.

        Returns: A data frame with an index of the roll number, face
values as columns,
        and count values in the cells (in the 'wide' format).
        """
        # Getting the distinct faces
        faces = self.game.show_results().melt().value.unique()

        # Initializing a DataFrame to store the counts
        counts_df = pd.DataFrame(columns=faces,
index=range(self.game.show_results().shape[0]))

        # Looping through the rolls and computing the counts
        for roll_number, row in self.game.show_results().iterrows():
            counts = row.value_counts()
            for face in faces:
                counts_df.loc[roll_number, face] = counts.get(face, 0)

        return counts_df
```

```
    def combo_count(self):
        """
        Determines unique combinations of faces rolled, along with their
counts.

        Returns:
            pandas DataFrame with distinct combinations and associated
counts.
        """
        results = self.game.show_results('narrow')
        combo_counts = results.groupby('Roll')
['Outcome'].apply(tuple).value_counts().reset_index()
        combo_counts.columns = ['Combo', 'Count']
        return combo_counts

    def permutation_count(self):
        """
        This method will compute unique permutations of faces rolled,
along with their counts.

        Returns:
            pandas DataFrame with distinct permutations and associated
counts.
        """
        results = self.game.show_results('narrow')
        permutation_counts = results.groupby('Roll')
['Outcome'].apply(list).value_counts().reset_index()
        permutation_counts.columns = ['Permutation', 'Count']
        return 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
from montecarlo import Die, Game, Analyzer
import sys


class TestDieGameAnalyzer(unittest.TestCase):

    # Test methods for Die class
    def test_die_creation(self):
        faces = np.array([1, 2, 3, 4, 5, 6])
        die = Die(faces)
```

```
        self.assertIsNotNone(die)

    def test_set_weight(self):
        faces = np.array([1, 2, 3, 4, 5, 6])
        die = Die(faces)
        die.set_weight(3, 5)
        self.assertEqual(die.get_die_state().loc[3, 'Weights'], 5)

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

    def test_get_die_state(self):
        faces = np.array([1, 2, 3, 4, 5, 6])
        die = Die(faces)
        state = die.get_die_state()
        self.assertTrue((state.index == faces).all())

    # Test methods for Game class
    def test_game_creation(self):
        die1 = Die(np.array([1, 2, 3, 4, 5, 6]))
        die2 = Die(np.array([1, 2, 3, 4, 5, 6]))
        game = Game([die1, die2])
        self.assertIsNotNone(game)

    def test_game_play(self):
        die1 = Die(np.array([1, 2, 3, 4, 5, 6]))
        die2 = Die(np.array([1, 2, 3, 4, 5, 6]))
        game = Game([die1, die2])
        game.play(5)
        self.assertEqual(game.play_data.shape[0], 5)

    def test_show_results(self):
        die1 = Die(np.array([1, 2, 3, 4, 5, 6]))
        die2 = Die(np.array([1, 2, 3, 4, 5, 6]))
        game = Game([die1, die2])
        game.play(5)
        results = game.show_results('wide')
        self.assertEqual(results.shape[0], 5)

    # Test methods for Analyzer class
    def test_analyzer_creation(self):
        die1 = Die(np.array([1, 2, 3, 4, 5, 6]))
        die2 = Die(np.array([1, 2, 3, 4, 5, 6]))
        game = Game([die1, die2])
        analyzer = Analyzer(game)
        self.assertIsNotNone(analyzer)

    def test_analyzer_jackpot(self):
        die1 = Die(np.array([1, 2, 3]))
        die2 = Die(np.array([1, 2, 3]))
```

```
        game = Game([die1, die2])
        game.play(10)
        analyzer = Analyzer(game)
        self.assertTrue(analyzer.jackpot() >= 0)

    def test_analyzer_face_counts_per_roll(self):
        die1 = Die(np.array([1, 2, 3]))
        die2 = Die(np.array([1, 2, 3]))
        game = Game([die1, die2])
        game.play(10)
        analyzer = Analyzer(game)
        counts = analyzer.face_counts_per_roll()
        self.assertEqual(counts.shape[0], 10)

    def test_analyzer_combo_count(self):
        die1 = Die(np.array([1, 2, 3]))
        die2 = Die(np.array([1, 2, 3]))
        game = Game([die1, die2])
        game.play(10)
        analyzer = Analyzer(game)
        combos = analyzer.combo_count()
        self.assertTrue(combos.shape[0] > 0)

    def test_analyzer_permutation_count(self):
        die1 = Die(np.array([1, 2, 3]))
        die2 = Die(np.array([1, 2, 3]))
        game = Game([die1, die2])
        game.play(10)
        analyzer = Analyzer(game)
        permutations = analyzer.permutation_count()
        self.assertTrue(permutations.shape[0] > 0)

# Execute the tests
#if __name__ == '__main__':
#    unittest.main()
#this command didn't push the results of the test to the .txt file so I
had to check stack overflow

if __name__ == '__main__':

unittest.TextTestRunner(stream=sys.stdout).run(unittest.TestLoader().loadTestsFr
```

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).

```
............
----------------------------------------------------------------------
Ran 12 tests in 0.053s

OK
```

Import (1)

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

- Module successuflly imported (1).

In [1]:
```
from montecarlo.montecarlo import Die,Game,Analyzer

# e.g. 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(Die)
help(Game)
help(Analyzer)
```

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

class Die(builtins.object)
 |  Die(faces: numpy.ndarray)
 |
 |  Fibricates a 'die' object that has specified faces and weights.
 |
 |  Attributes:
 |      _die_data (pandas.DataFrame): This is the dataframe that stores face values and
weights.
 |
 |  Methods defined here:
 |
 |  __init__(self, faces: numpy.ndarray)
 |      Initializes the die object with specified 'faces'.
 |
 |      Args:
 |          faces (numpy.ndarray): An array containing distinct face values of the die.
 |
 |      Raises:
 |          TypeError: If faces is not a NumPy array.
 |          ValueError: If faces do not have distinct values.
 |
 |  get_die_state(self) -> pandas.core.frame.DataFrame
 |      This will return a copy of the die's data frame.
 |
 |      Returns:
 |          pandas.DataFrame: A copy of the die data frame.
 |
```

```
 |  roll(self, num_rolls: int = 1) -> list
 |      This method simulates rolling of the die multiple times.
 |
 |      Argument:
 |          num_rolls (int): The number of times the die is rolled.
 |
 |      Returns:
 |          list: A list of outcomes from rolling the die.
 |
 |  set_weight(self, face, weight)
 |      This controls the weight for a specific face value.
 |
 |      Arguments:
 |          face: The face value for which the weight is to be set.
 |          weight: The weight value assigned.
 |
 |      Raises:
 |          IndexError: If the face value is invalid.
 |          TypeError: If the weight is not a numeric value.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

Help on class Game in module montecarlo.montecarlo:

class Game(builtins.object)
 |  Game(dice)
 |
 |  The game class contains a game consisting of rolling multiple dice.
 |
 |  Attributes:
 |      dice: A list of Die objects representing the dice in the game.
 |      play_data: Data frame that stores the results of the most recent play.
 |
 |  Methods defined here:
 |
 |  __init__(self, dice)
 |      This initializes the Game class with a list of similar dice.
 |      This also checks if the list contains Die objects
 |      Argusment:
 |          dice (list): A list of already instantiated similar dice (Die objects).
 |
 |  play(self, times)
 |      This method simulates playing the game by rolling the dice a specified number of
times.
 |
 |      Argsuments:
 |          times (int): The number of times instructed to roll the dice.
 |
 |  show_results(self, form='wide')
 |      This method displays the results of the most recent play.
 |
 |      Args:
 |          Format options: Options include: 'wide' (default) or 'narrow'
 |
 |      Returns:
 |          pandas.DataFrame: The play results data frame.
 |
 |      Raises:
```

```
 |          ValueError: If an invalid option is provided for the format.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

Help on class Analyzer in module montecarlo.montecarlo:

class Analyzer(builtins.object)
 |  Analyzer(game)
 |
 |  Analyzes the results of a single game and computes various descriptive properties.
 |
 |  Attributes:
 |      game (Game): A Game object whose results will be analyzed.
 |
 |  Methods defined here:
 |
 |  __init__(self, game)
 |      Initializes an Analyzer object with a Game object.
 |
 |      Args:
 |          game (Game): A Game object whose results will be analyzed.
 |
 |      Raises:
 |          ValueError: If the input is not a Game object.
 |
 |  combo_count(self)
 |      Determines unique combinations of faces rolled, along with their counts.
 |
 |      Returns:
 |          pandas DataFrame with distinct combinations and associated counts.
 |
 |  face_counts_per_roll(self)
 |      Counts how many times a given face is rolled in each event.
 |
 |      Returns: A data frame with an index of the roll number, face values as columns,
 |      and count values in the cells (in the 'wide' format).
 |
 |  jackpot(self)
 |      Jackpot computes the number of jackpot occurrences in the game.
 |
 |      Returns:
 |          int: The number of jackpots in the game.
 |
 |  permutation_count(self)
 |      This method will compute unique permutations of faces rolled, along with their c
ounts.
 |
 |      Returns:
 |          pandas DataFrame with distinct permutations and associated counts.
 |
 |  ----------------------------------------------------------------------
 |  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/AddisonGambhir/ajg7pk_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).

```
# Command to install:$

bash-4.2$pip3 install -e .

# Terminal Output:

Defaulting to user installation because normal site-packages is not
writeable
Obtaining file:///sfs/qumulo/qhome/ajg7pk/Documents/MSDS/ds5100-
ajg7pk/lessons/M15
Installing collected packages: montecarlo
  Running setup.py develop for montecarlo
Successfully installed montecarlo
```

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 [3]:
```
import numpy as np
import pandas as pd

#fair coin
fair_faces = np.array(['H', 'T'])
fair_coin = Die(fair_faces)
print("Fair coin Info:", fair_coin.get_die_state())

#unfair coin
unfair_faces = np.array(['H', 'T'])
unfair_coin = Die(unfair_faces)
unfair_coin.set_weight('H', 5)
print("Unfair coin Info:", unfair_coin.get_die_state())
```

```
Fair coin Info:    Weights
H      1.0
T      1.0
Unfair coin Info:    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 [4]:
```
# creating fair coins
faces = np.array(['H', 'T'])
fair_coin = Die(faces)

unfair_coin = Die(faces)
unfair_coin.set_weight(face='H', weight=5)

game = Game([fair_coin, unfair_coin])

game.play(1000)

results_df = game.show_results()

print("Results of the game with 1000 flips:")
results_df
```

```
Results of the game with 1000 flips:
```

Out[4]:

	Die_1	Die_2
0	H	H
1	H	H
2	H	H
3	H	T
4	H	H
...	...	...
995	H	H

	Die_1	Die_2
996	H	T
997	T	H
998	T	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 [5]:
```
#from die_game import Die, Game
import numpy as np

faces = np.array(['H', 'T'])
fair_coin = Die(faces)

unfair_coin = Die(faces)
unfair_coin.set_weight(face='H', weight=5)

new_game = Game([unfair_coin, unfair_coin, fair_coin])
new_game.play(1000)

new_results_df = new_game.show_results()
new_results_df
```

Out[5]:

	Die_1	Die_2	Die_3
0	H	H	T
1	H	H	H
2	T	H	T
3	H	T	T
4	H	T	T
...	...	...	...
995	H	H	H
996	H	H	H
997	H	H	T
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 [6]:
```
analyzer_results = Analyzer(game)  # Game results from task 2
analyzer_new_results = Analyzer(new_game)  # Game results from task 3

raw_frequency_results = analyzer_results.jackpot()
raw_frequency_new_results = analyzer_new_results.jackpot()

# Print raw frequencies
print("Raw frequency of jackpots for task 2:", raw_frequency_results)
print("Raw frequency of jackpots for task3:", raw_frequency_new_results)
```

```
Raw frequency of jackpots for task 2: 513
Raw frequency of jackpots for task3: 369
```

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 [7]:
```
total_rolls = 1000  # Since each game consists of 1000 flips

# Compute relative frequencies for both games
relative_frequency_results = raw_frequency_results / total_rolls
relative_frequency_new_results = raw_frequency_new_results / total_rolls


print("Relative frequency of jackpots for the fair game:", relative_frequency_results)
print("Relative frequency of jackpots for the unfair game:", relative_frequency_new_res
```

```
Relative frequency of jackpots for the fair game: 0.513
Relative frequency of jackpots for the unfair game: 0.369
```

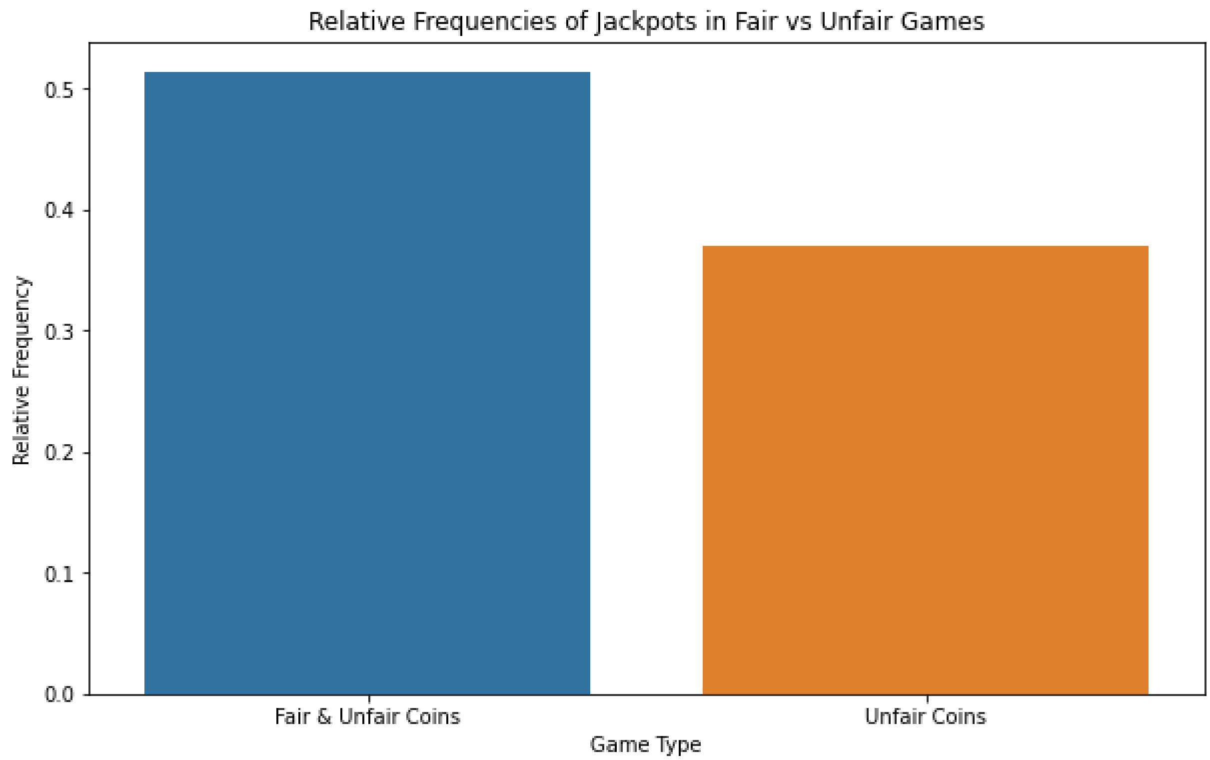
Task 6. Show your results, comparing the two relative frequencies, in a simple bar chart.

- Bar chart plotted and correct (1).

In [8]:
```
import seaborn as sns
import matplotlib.pyplot as plt

relative_frequencies = pd.DataFrame({
    'Game': ['Fair & Unfair Coins', 'Unfair Coins'],
    'Relative Frequency': [relative_frequency_results, relative_frequency_new_results]
})

# Plot the bar chart
plt.figure(figsize=(10, 6))
sns.barplot(x='Game', y='Relative Frequency', data=relative_frequencies)
plt.title('Relative Frequencies of Jackpots in Fair vs Unfair Games')
plt.ylabel('Relative Frequency')
plt.xlabel('Game Type')
plt.show()
```



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 [9]:
```
faces=np.array([1,2,3,4,5,6])
die1 = Die(faces)
die2 = Die(faces)
die3 = Die(faces)
print("die1 Info:", die1.get_die_state())
print("die2 Info:", die2.get_die_state())
print("die3 Info:", die3.get_die_state())
```

```
die1 Info:    Weights
1       1.0
2       1.0
3       1.0
4       1.0
5       1.0
6       1.0
die2 Info:    Weights
1       1.0
2       1.0
3       1.0
4       1.0
5       1.0
6       1.0
die3 Info:    Weights
1       1.0
2       1.0
3       1.0
4       1.0
5       1.0
6       1.0
```

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 [10]:
```
die1.set_weight(face=6, weight=5)
die1.get_die_state()
```

Out[10]:

	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 [11]:
```
die2.set_weight(face=1, weight=5)
die2.get_die_state()
```

Out[11]:

	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 [12]:
```
#5 dice
faces = np.array([1, 2, 3, 4, 5, 6])
fair_dice = [Die(faces) for i in range(5)]

game_with_fair_dice = Game(fair_dice)
game_with_fair_dice.play(10000)

print("Game Results:")
game_with_fair_dice.show_results()
```

```
Game Results:
```

Out[12]:

	Die_1	Die_2	Die_3	Die_4	Die_5
0	1	6	5	4	3
1	1	1	6	5	3
2	5	4	1	1	6
3	2	6	6	1	2
4	1	2	1	1	1
...	...	...	...	...	...
9995	5	3	3	1	2
9996	5	2	5	3	5
9997	4	6	5	6	6
9998	6	1	3	6	1
9999	3	6	2	1	4

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 [13]:

```
#dice
fair_dice_for_mixed_game = [Die(faces) for i in range(3)]
mixed_dice = fair_dice_for_mixed_game + [die1, die2]


#game
game_with_mixed_dice = Game(mixed_dice)
game_with_mixed_dice.play(10000)
results_df_mixed_dice = game_with_mixed_dice.show_results()
print("Game results:")

results_df_mixed_dice
```

```
Game results:
```

Out[13]:

	Die_1	Die_2	Die_3	Die_4	Die_5
0	2	2	6	6	5
1	6	6	2	6	1
2	4	4	4	1	1
3	5	1	5	6	4
4	2	2	5	6	4
...	...	...	...	...	...
9995	3	2	3	4	4

	Die_1	Die_2	Die_3	Die_4	Die_5
9996	3	2	4	3	1
9997	4	3	1	4	1
9998	6	5	1	6	1
9999	1	4	4	1	1

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.

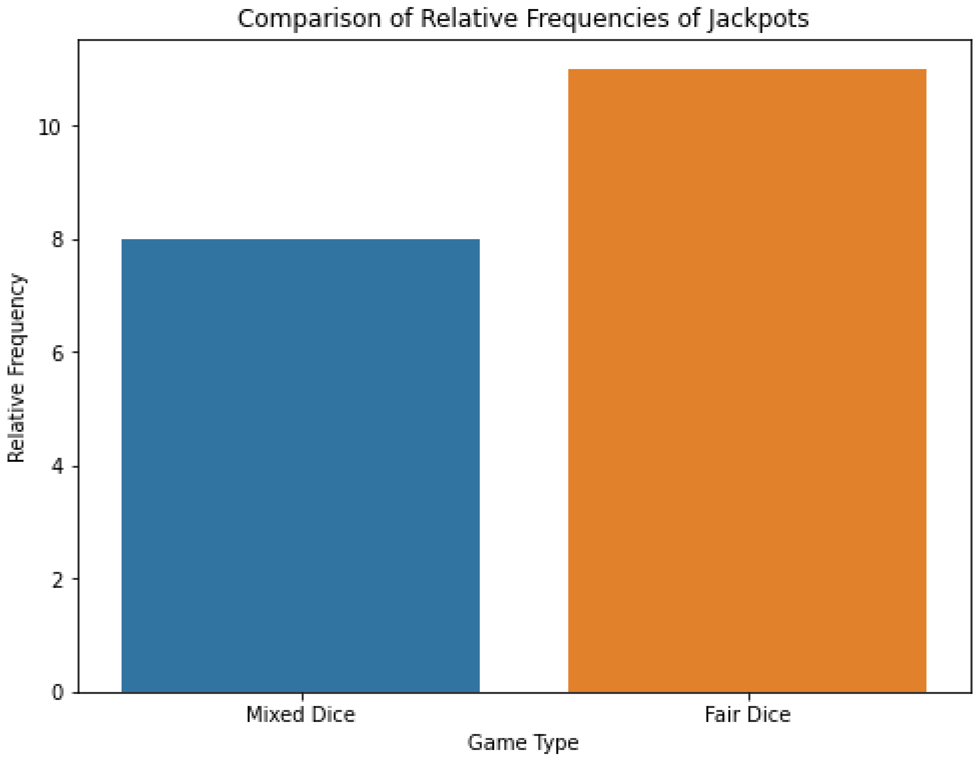
- Jackpot methods called (1).
- Graph produced (1).

In [14]:

```
#analyzing games
analyzer_mixed_game = Analyzer(game_with_mixed_dice)
analyzer_fair_game = Analyzer(game_with_fair_dice)
relative_frequency_mixed = analyzer_mixed_game.jackpot()
relative_frequency_fair = analyzer_fair_game.jackpot()

#making data
df1 = pd.DataFrame({
    'Game': ['Mixed Dice', 'Fair Dice'],
    'Relative Frequency': [relative_frequency_mixed, relative_frequency_fair]
})

#plot
plt.figure(figsize=(8, 6))
sns.barplot(x='Game', y='Relative Frequency', data=df1)
plt.title('Comparison of Relative Frequencies of Jackpots')
plt.ylabel('Relative Frequency')
plt.xlabel('Game Type')
plt.show()
```



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 [15]:
```
my_txt = 'english_letters.txt'
letter_freq_data = pd.read_csv(my_txt, sep=" ", header=None)
letter_freq_data.columns = ['Letter', 'Weight']

# faces und weights
faces = letter_freq_data['Letter'].values
weights = letter_freq_data['Weight'].values

letter_die = Die(faces)
for face, weight in zip(faces, weights):
    letter_die.set_weight(face, weight)

print("Letter die info:")
letter_die.get_die_state()
```

```
Letter die info:
```

Out[15]:

	Weights
E	529117365.0
T	390965105.0
A	374061888.0
O	326627740.0

	Weights
I	320410057.0
N	313720540.0
S	294300210.0
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 [16]:
```
letter_dice = [Die(faces) for i in range(4)]

for die in letter_dice:
    for face, weight in zip(faces, weights):
        die.set_weight(face, weight)

game_4d = Game(letter_dice)
game_4d.play(1000)

# Show the results
print("Game results:")
game_4d.show_results()
```

```
Game results:
```

Out[16]:

	Die_1	Die_2	Die_3	Die_4
0	C	I	A	T
1	H	T	V	E
2	R	R	I	U
3	L	H	E	E
4	H	T	A	E
...	...	...	...	...
995	W	O	R	E
996	B	N	T	R
997	E	V	L	E
998	E	T	O	N
999	C	A	P	E

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 [17]:

```
with open('scrabble_words.txt', 'r') as file:
    vocabulary = set(word.strip().upper() for word in file.readlines())

analyzer_letter = Analyzer(game_4d)

#counting permutations
permutation_counts = analyzer_letter.permutation_count()

# checking how many permutations are english word
real_words = [perm for perm in permutation_counts['Permutation'] if ''.join(perm) in vo
real_word_count = len(real_words)

print("Number of permutations that are actual English words:", {real_word_count})
```

```
Number of permutations that are actual English words: {45}
```

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 [18]:

```
def play_game_and_count_words(num_dice, vocabulary):
    dice = [letter_die] * num_dice

    game_letter = Game(dice)

    game_letter.play(1000)
```

```
    analyzer = Analyzer(game_letter)
    permutations = analyzer.permutation_count()

    count_actual_words = sum(1 for permutation in permutations['Permutation'] if ''.joi

    return count_actual_words

count_4_dice = play_game_and_count_words(4, vocabulary)

count_5_dice = play_game_and_count_words(5, vocabulary)

print('Words produced from 4 dice:',count_4_dice)
print('Words produced from 5 dice:',count_5_dice)

print("4 dice produce more actual words.")
```

```
Words produced from 4 dice: 68
Words produced from 5 dice: 7
4 dice produce more actual 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.

In [ ]: