Final Project Report

- Class: DS 5100
- Student Name: Wyatt Priddy
- Student Net ID: tds8tv
- This URL: https://github.com/wpriddy/tds8tv_ds5100_montecarlo/blob/master/montecarlo_demo.ipynb

Instructions

Follow the instructions in the Final Project instructions 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/wpriddy/tds8tv_ds5100_montecarlo

Paste a copy 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
from collections import Counter

class die:
    """Die Class

    Creates and allows modification of a Die with N faces and corresponding W
weight

    """
    def __init__(self, faces: np.array):
        """Initializes Die Class with user prescribed faces and initial weights of
1.0:

        params:
            faces: numpy array of strings or numeric types"""
```

```
        # Check that faces are numpy array
        if not isinstance(faces, np.ndarray):
            raise TypeError(f'Faces must be numpy array not {type(faces)}')

        # Check that all sides are unique
        if len(faces) != len(set(faces)):
            # if not raise value error
            raise ValueError(f'Face values must be distinct. {len(faces) -
len(set(faces))} are duplicated')

        # Create private dataframe
        self._parameters = pd.DataFrame({'face': faces})

        # Assign weights = 1
        self._parameters['weights'] = 1.0

    def change_side_weight(self, face: (int, str), weight: int):
        """Takes two arguments: the face value to be changed and the new
    weight.

        params:
            face: string or int value of existing face on die
            weight: new weight to be added die"""

        # Check that face exists before updating
        if face not in self._parameters.face.unique():
            raise IndexError(f'Invalid Face name "{face}"')

        # Check that weight is correct type before updating
        try:
            float(weight)
        except ValueError:
            raise TypeError(f'Invalid weight type {type(weight)}')

        # updates private dataframe with weight
        self._parameters.loc[self._parameters.face == face, 'weights'] = weight

        return self

    def roll_the_dice(self, rolls: int = 1):
        """Takes a parameter of how many times the die is to be rolled;
    defaults to 1. Returns a Python list of outcomes.

        params:
            rolls: int indicating number of rolls"""

        # Make sure rolls is integer and greater than 0
        if not isinstance(rolls, int) or rolls <= 0:
            raise Exception('Rolls must be integer and greater than 0')

        # Instantiate empty list to store results
        results = []

        # Roll die each time
        for roll in range(rolls):
            # Sample DataFrame for Dice Roll
            results.append(self._parameters.sample(n=1, replace=True,
weights='weights').squeeze())
```

```
        # return list
        return results

    def show_die(self):
        """Shows existing state of Dice"""

        return self._parameters

    def __str__(self):
        """Creates REPR to type check class"""

        return 'Die Class'

class game:
    """Game Class

    Creates Game and Allows for Play by rolling dice and storing results of game
play
    """
    def __init__(self, list_of_die: list):
        """Initializes Game Object from list of die

        params:
            list_of_die: list containing one or more die object"""

        #Make sure all objects in list are Die Class
        if not all(i.__str__() == 'Die Class' for i in list_of_die):
            raise TypeError('Not all objects in list are die class')

        # Save to self
        self.list_of_die = list_of_die

    def play(self, number_of_die_rolls: int):
        """Takes an integer parameter to specify how many times the dice should
    be rolled. Saves the result of the play to a private data frame

        params:
            number_of_die_rolls: int

        """
        # Make sure rolls numbers are int
        if not isinstance(number_of_die_rolls, int):
            raise TypeError(f'number_of_die_rolls must be int not
{type(number_of_die_rolls)}')

        # save to self
        self.die_rolls = number_of_die_rolls

        # instantiate empty play dict
        self.play_dict = {}

        # roll the dice
        for enum, die in enumerate(self.list_of_die, start=1):
            # roll of each dice
            outcomes = {enum: [roll['face'] for roll in die.roll_the_dice(rolls =
self.die_rolls)]}
```

```
            # Update Dictionary with outcomes
            self.play_dict = {**outcomes, **self.play_dict}

        # Create DataFrame from Dictionary of Outcomes
        self.game_results = pd.DataFrame(self.play_dict)
        # Sort by Die Num Ascending
        self.game_results = self.game_results[sorted(self.game_results.columns)]

        return self

    def most_recent_play(self, shape: str = 'wide'):
        """Shows the results of the most recent play in either wide or narrow
dataframe format. Default = Wide

        params:
            shape: 'wide' or 'narrow'

        """
        # Raise Exception if incorrect parameters
        if shape.lower() not in (arg_constraints := ['narrow', 'wide']):
            raise ValueError(f'Shape not in {arg_constraints}')

        # If wide
        if shape == 'wide':
            # Return original game results
            return self.game_results

        # Otherwise it has to be narrow
        else:
            # Stack Results
            self.game_results_narrow = pd.DataFrame.from_dict(self.play_dict,
orient="index").stack().to_frame(name='results')
            # Sort Index Accordingly
            self.game_results_narrow =
self.game_results_narrow.swaplevel(axis=0).sort_index()
            # Return manipulated frame
            return self.game_results_narrow

    def __str__(self):
        """Creates REPR to type check class"""

        return 'Game Class'

class analyzer:
    """Analyzer Class
        Stores results of game play and calculates summary statistics
    """
    def __init__(self, game):
        """Initializes Analyzer Object For Specific Game"""

        # check params is game class
        if game.__str__() != 'Game Class':

            raise ValueError(f'game should be "Game Class" not "{type(game)}"')

        # save to self
        self.game = game
```

```
    def jackpot(self):
        """Analyzes results to see number of times 'jackpot', or all faces being
the same for rolled dices,
        happens within a specific game
        """
        # Set initially to 0
        self.num_jackpots = 0

        # Iterate through each play to see if jackpot happened
        for index in self.game.game_results.index:
            if len(set(self.game.game_results.iloc[index].values)) == 1:
                self.num_jackpots += 1

        return self.num_jackpots

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

          For example, if a roll of five dice has all sixes, then the
        counts for this roll would be 5 for the face value ‘6’ and 0
        for the other faces.

         Returns a data frame of results."""
        # initializes face counts list
        face_counts = []
        # Iterates throught game results
        for index in self.game.game_results.index:

face_counts.append(dict(self.game.game_results.iloc[index].value_counts()))

        # Creates dataframe of results
        self.face_counts = pd.DataFrame(face_counts).fillna(0)

        return self.face_counts


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

        Returns a data frame of results.
        """
        # Get list to store unique combo counts
        values = []
        # Iterate through game results
        for index in self.game.game_results.index:
            values.append(tuple(set(self.game.game_results.iloc[index])))

        # Get count of values
        counter_results = Counter(values)
        # Create DataFrame from Values
        df = pd.DataFrame.from_dict(counter_results, orient='index').reset_index()
        # Rename column
        df = df.rename(columns={0:'count'})
        # Manual updating of dataframe
        df['index'] = df['index'].astype(str)
        df['index'] = df['index'].str.replace('(', '')
```

```
        df['index'] = df['index'].str.replace(')', '')
        # Exand index for multi-index
        df[[*range(df['index'][0].count(',')+1)]] = df['index'].str.split(',',
expand=True)
        # Set Multi-Index
        df = df.set_index([i for i in df.columns if type(i) == int])
        # Drop original transforming columns
        df.drop(columns=['index'], inplace=True)
        self.combo_results = df
        return self.combo_results

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

        Returns a data frame of results.
        """

        # Create counter column
        self.game.game_results['count'] = 1
        # Group By Combos
        self.permutation_results = self.game.game_results.groupby(by=[i for i in
self.game.game_results.columns if 'count' != i]).agg({'count': 'sum'})

        return self.permutation_results
```

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).

```
from montecarlo import die, game, analyzer
import unittest
import numpy as np
import pandas as pd

# Initialize a Die
test_die = die(np.array(['heads', 'tails']))

# Initialize a Game
test_game = game([test_die, test_die])

# Initialize the Analyzer
test_analyzer = analyzer(test_game.play(10))

class montecarlo_test_suite(unittest.TestCase):

    def test_die_initializer(self):
        # Test that initializing die class creates private pd.DataFrame
        self.assertEqual(type(test_die._parameters), pd.DataFrame)

    def test_die_change_side_weight(self):
        # Test that changing the side weight updates correctly
```

```
        test_die.change_side_weight('heads', 33)

        self.assertTrue(test_die._parameters[test_die._parameters.face == 'heads']
['weights'][0] == 33, 'Face weight did not update')

    def test_die_roll_output(self):
        # Test die roll outputs list
        self.assertEqual(type(test_die.roll_the_dice(3)), list, "Dice Roll did not
return list")

    def test_die_show_the_dice(self):
        #Test that show the dice returns pd.DataFrame
        self.assertEqual(type(test_die.show_die()), pd.DataFrame, 'Show Dice did
not return pandas DataFrame')

    def test_game_initializer(self):
        # Test that initializer stores list of die
        self.assertEqual(test_game.list_of_die, [test_die, test_die], 'Game
initializer did not store die')

    def test_game_play(self):
        # test game play saves results to pd.DataFrame
        test_game.play(3)
        self.assertIsInstance(test_game.game_results, pd.DataFrame, 'Game results
are not pandas DataFrame')

    def test_game_results_narrow(self):
        # Test that narrow dataframe return only 1 column
        self.assertTrue(len(test_game.most_recent_play(shape='narrow').columns) ==
1, 'Narrow function returned more than one column')

    def test_analyzer_initializer(self):
        # Test that the analyzer won't initialize without a game object
        self.assertRaises(ValueError, analyzer, "won't work")

    def test_analyzer_jackpot(self):
        # Test that jackpot returns an int
        self.assertIsInstance(test_analyzer.jackpot(), int, 'Jackpot is not
integer')

    def test_analyzer_face_counts_per_roll(self):
        #Test that Face Counts returns pd.DataFrame
        self.assertIsInstance(test_analyzer.face_counts_per_roll(), pd.DataFrame,
'Face counts is not pd.DataFrame')

    def test_analyzer_combo_counts(self):
        #Test that Combo Counts returns pd.DataFrame with multi-index
        self.assertIsInstance(test_analyzer.combo_counts().index,
pd.core.indexes.multi.MultiIndex, 'Combo Counts is not multi-index pd.DataFrame')

    def test_analyzer_permutation_counts(self):
        #Test that Permutation Counts returns pd.DataFrame with multi-index
        self.assertIsInstance(test_analyzer.permutation_counts().index,
pd.core.indexes.multi.MultiIndex, 'Combo Counts is not multi-index 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_analyzer_combo_counts (__main__.montecarlo_test_suite) ... ok
test_analyzer_face_counts_per_roll (__main__.montecarlo_test_suite) ... ok
test_analyzer_initializer (__main__.montecarlo_test_suite) ... ok
test_analyzer_jackpot (__main__.montecarlo_test_suite) ... ok
test_analyzer_permutation_counts (__main__.montecarlo_test_suite) ... ok
test_die_change_side_weight (__main__.montecarlo_test_suite) ... ok
test_die_initializer (__main__.montecarlo_test_suite) ... ok
test_die_roll_output (__main__.montecarlo_test_suite) ... ok
test_die_show_the_dice (__main__.montecarlo_test_suite) ... ok
test_game_initializer (__main__.montecarlo_test_suite) ... ok
test_game_play (__main__.montecarlo_test_suite) ... ok
test_game_results_narrow (__main__.montecarlo_test_suite) ... ok

----------------------------------------------------------------------
Ran 12 tests in 0.028s

OK
```

Import (1)

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

- Module successuflly imported (1).

In [25]:

```
# import custom module
from montecarlo import montecarlo
# Also import pandas for later visualizations
import pandas as pd
```

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 [26]:

```
help(montecarlo)
```

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

NAME
    montecarlo.montecarlo

CLASSES
    builtins.object
        analyzer
        die
```

```
        game

    class analyzer(builtins.object)
     |  analyzer(game)
     |
     |  Analyzer Class
     |  Stores results of game play and calculates summary statistics
     |
     |  Methods defined here:
     |
     |  __init__(self, game)
     |      Initializes Analyzer Object For Specific Game
     |
     |  combo_counts(self)
     |      Computes the distinct combinations of faces rolled, along with their
     |      counts.
     |
     |          Returns a data frame of results.
     |
     |  face_counts_per_roll(self)
     |      Computes how many times a given face is rolled in each event.
     |
     |        For example, if a roll of five dice has all sixes, then the
     |      counts for this roll would be 5 for the face value ‘6’ and 0
     |      for the other faces.
     |
     |       Returns a data frame of results.
     |
     |  jackpot(self)
     |      Analyzes results to see number of times 'jackpot', or all faces being the same
for rolled dices,
     |      happens within a specific game
     |
     |  permutation_counts(self)
     |      Computes the distinct permutations of faces rolled, along with their
     |      counts.
     |
     |          Returns a data frame of results.
     |
     |  ----------------------------------------------------------------------
     |  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: <built-in function array>)
     |
     |  Die Class
     |
     |  Creates and allows modification of a Die with N faces and corresponding W weight
     |
     |  Methods defined here:
     |
     |  __init__(self, faces: <built-in function array>)
     |      Initializes Die Class with user prescribed faces and initial weights of 1.0:
     |
     |      params:
     |          faces: numpy array of strings or numeric types
     |
     |  __str__(self)
     |      Creates REPR to type check class
     |
```

```
 |  change_side_weight(self, face: (<class 'int'>, <class 'str'>), weight: int)
 |      Takes two arguments: the face value to be changed and the new
 |      weight.
 |
 |          params:
 |              face: string or int value of existing face on die
 |              weight: new weight to be added die
 |
 |  roll_the_dice(self, rolls: int = 1)
 |      Takes a parameter of how many times the die is to be rolled;
 |      defaults to 1. Returns a Python list of outcomes.
 |
 |          params:
 |              rolls: int indicating number of rolls
 |
 |  show_die(self)
 |      Shows existing state of Dice
 |
 |  ----------------------------------------------------------------------
 |  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: list)
 |
 |  Game Class
 |
 |  Creates Game and Allows for Play by rolling dice and storing results of game play
 |
 |  Methods defined here:
 |
 |  __init__(self, list_of_die: list)
 |      Initializes Game Object from list of die
 |
 |      params:
 |          list_of_die: list containing one or more die object
 |
 |  __str__(self)
 |      Creates REPR to type check class
 |
 |  most_recent_play(self, shape: str = 'wide')
 |      Shows the results of the most recent play in either wide or narrow dataframe f
ormat. Default = Wide
 |
 |      params:
 |          shape: 'wide' or 'narrow'
 |
 |  play(self, number_of_die_rolls: int)
 |      Takes an integer parameter to specify how many times the dice should
 |      be rolled. Saves the result of the play to a private data frame
 |
 |          params:
 |              number_of_die_rolls: int
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
```

```
     |      list of weak references to the object (if defined)

FILE
    c:\users\wyatt\onedrive\documents\repos\tds8tv_ds5100_montecarlo\montecarlo\montecarl
o.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/wpriddy/tds8tv_ds5100_montecarlo/blob/master/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).

```
bash-4.2$pip install -e .
Defaulting to user installation because normal site-packages is not writeable
Obtaining file:///sfs/qumulo/qhome/tds8tv/Documents/MSDS/DS5100/final
Installing collected packages: montecarlo
  Running setup.py develop for montecarlo
Successfully installed montecarlo
bash-4.2$
```

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 [27]:
```
import numpy as np

# Create Fair Coin
fair_coin = montecarlo.die(np.array(['H', 'T']))
```

```
# Create Unfair Coin
unfair_coin = montecarlo.die(np.array(['H', 'T']))
# Change Weight of Unfair Coin
unfair_coin.change_side_weight(face='H', weight=5);
```

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

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

In [28]:
```
# Store number of flips
flips = 1000

# Instantiate and Play Fair Game
fair_game = montecarlo.game([fair_coin, fair_coin])
fair_game.play(flips);
```

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 correctly and without error (1).

In [29]:
```
# Instantiate and Play Game with 2 Unfair Die
unfair_game = montecarlo.game([unfair_coin, unfair_coin, fair_coin])
unfair_game.play(flips);
```

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 [43]:
```
# Instantiate Analyzer Classes
fair_game_analyzer = montecarlo.analyzer(fair_game)
unfair_game_analyzer = montecarlo.analyzer(unfair_game)

#Show Fair Game Jackpots
print('Fair Game Jackpots:', fair_game_analyzer.jackpot())
#Show Unfair Game Jackpots
print('Unfair Game Jackpots:', unfair_game_analyzer.jackpot())
```

```
Fair Game Jackpots: 517
Unfair Game Jackpots: 370
```

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

- Both relative frequencies computed (1).

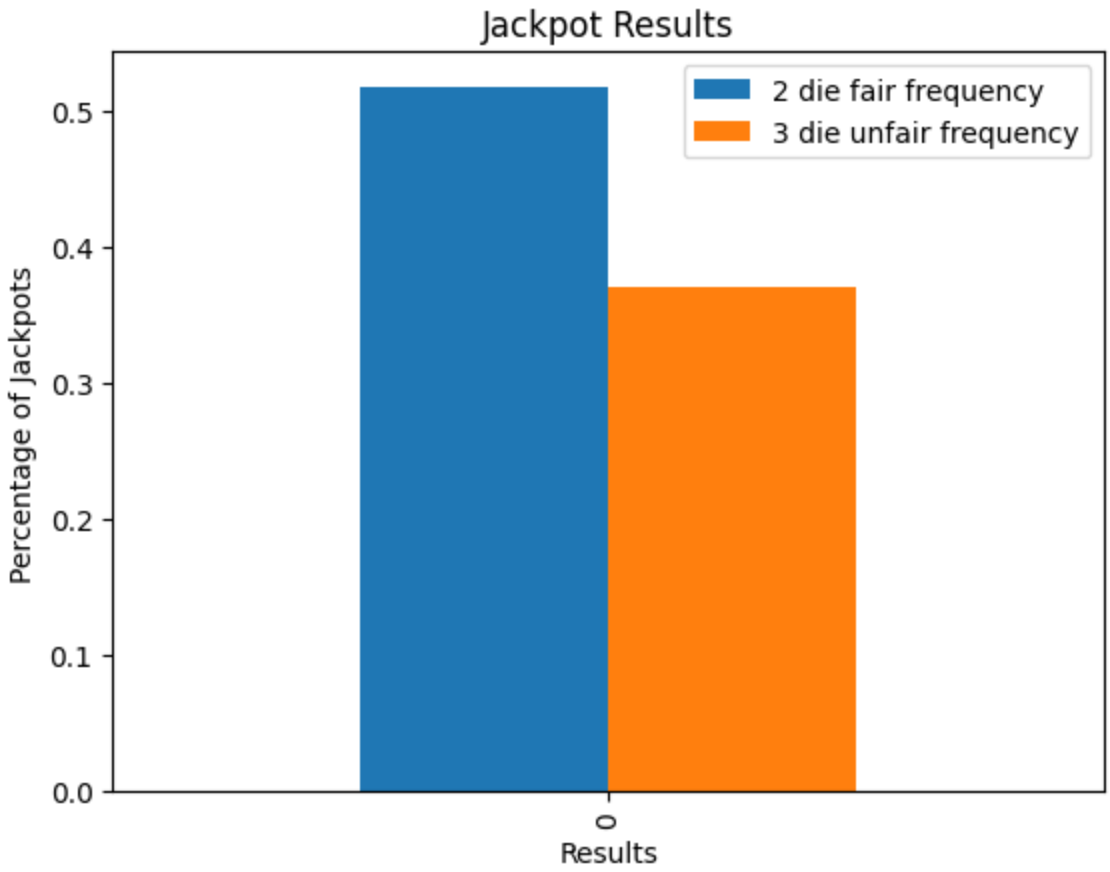
In [31]:
```
# Get Percentage of Flips that Resulted in Jackpots
fair_game_frequency = fair_game_analyzer.jackpot() / flips
unfair_game_frequency = unfair_game_analyzer.jackpot() / flips
# Output Results
print(f"{fair_game_frequency:.1%} of the fair game results as jackpots")
print(f"{unfair_game_frequency:.1%} of the unfair game results as jackpots")
```

```
51.7% of the fair game results as jackpots
37.0% of the unfair game results as jackpots
```

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

- Bar chart plotted and correct (1).

In [32]:
```
pd.DataFrame({'2 die fair frequency': [fair_game_frequency], '3 die unfair frequency': [ur
```



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 [33]:
```
# Instantiate Three Die
die_one = montecarlo.die(np.array([i for i in range(1,7)]))
die_two = montecarlo.die(np.array([i for i in range(1,7)]))
die_three = montecarlo.die(np.array([i for i in range(1,7)]))
```

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 [34]:
```
# Die one is unfair die
die_one.change_side_weight(face=6, weight=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 [35]:

```
# Die two is also unfair
die_two.change_side_weight(face=1, weight=5);
```

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

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

In [36]:
```
# Instantiate Game with Fair Die
five_die_game = montecarlo.game([die_three for _ in range(5)])
five_die_game.play(10000);
```

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 [37]:
```
# Instantiate Game with the 2 Unfair Dice and 3 Fair Dice
five_die_unfair_game = montecarlo.game([die_one, die_two, *[die_three for _ in range(3)]])
five_die_unfair_game.play(10000);
```

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.

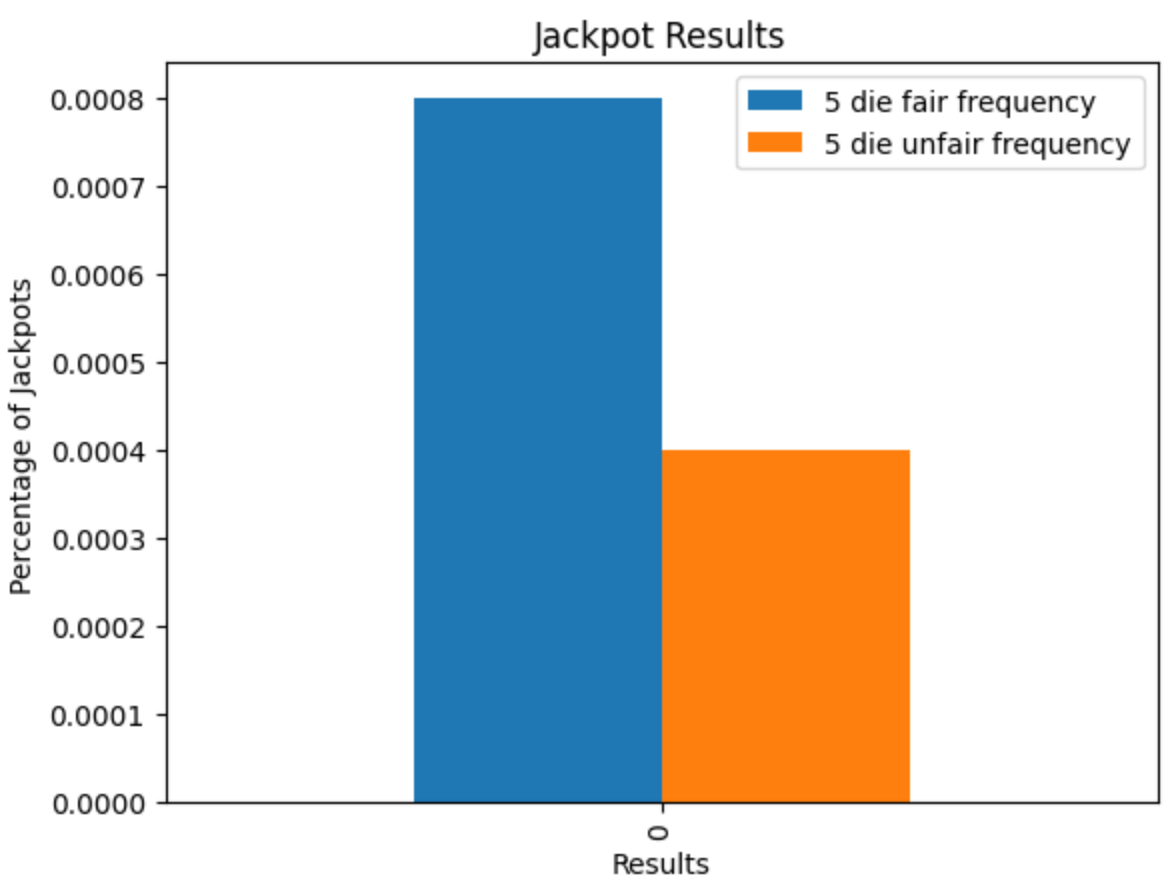
- Jackpot methods called (1).
- Graph produced (1).

In [38]:
```
# Initialize 5 Die Fair Analyzer
five_die_fair_analyzer = montecarlo.analyzer(game=five_die_game)
# Initialize 5 Die Unfair Analyzer
five_die_unfair_analyzer = montecarlo.analyzer(game=five_die_unfair_game)

#Show Fair Game Jackpots
print('5 Die Fair Game Jackpots:', five_die_fair_analyzer.jackpot())
#Show Unfair Game Jackpots
print('5 Die Unfair Game Jackpots:', five_die_unfair_analyzer.jackpot())

# Created Bar Chart Graphic
pd.DataFrame({'5 die fair frequency': [five_die_fair_analyzer.jackpot() / 10000], '5 die u
```

```
5 Die Fair Game Jackpots: 8
5 Die Unfair Game Jackpots: 4
```



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 [39]:
```
# Read in English Letters Data Frame and Manipulate
english_letters = pd.read_csv(open('data/english_letters.txt', 'r'), sep=' ', header=None
                              ).sort_values(by=0).reset_index(drop=True).rename(columns={0:

# Create a Die with each letter of the alphabet
english_die = montecarlo.die(np.array([letter for letter in english_letters['Letter']]))

# Change Weights to Corresponding weight in english_letters DataFrame
for row in english_letters.index:
    english_die.change_side_weight(face=english_letters.iloc[row]['Letter'], weight=englis
```

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

- Game play method properly called (1).

In [40]:
```
# Instantiate 4 die with english letters and roll 1000 times
english_game = montecarlo.game([english_die for _ in range(4)])
english_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 [41]:
```
# Read in Scrabble Words
scrabble_words = open('data/scrabble_words.txt', 'r').read().split('\n')

# Create Analyzer Class
english_analyzer = montecarlo.analyzer(english_game)

# Save Words from Multi-Index Resulting from permutation method into List for Comparing ag
permutation_words = [''.join(i).upper() for i in english_analyzer.permutation_counts().inc

# Find number of permutation words in Scrabble Words
print('Number of words found:', sum([1 for word in permutation_words if word in scrabble_w
```

```
Number of words found: 42
```

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 [42]:
```
# Instantiate 5 die with english letters and roll 1000 times
english_game_five_die = montecarlo.game([english_die for _ in range(5)])
english_game_five_die.play(1000)

# Create Analyzer Class
english_analyzer_five_die = montecarlo.analyzer(english_game_five_die)

# Save Words from Multi-Index Resulting from permutation method into List for Comparing ag
permutation_words_five_die = [''.join(i).upper() for i in english_analyzer_five_die.permut

# Find number of permutation words in Scrabble Words
print('Number of words found:', sum([1 for word in permutation_words_five_die if word in s
```

```
Number of words found: 3
```

Analysis of Word Generation

The four die game generates more words that are eligible for scrabble than the the 5 die game. As you add the extra die (or new letter), the permutations become exponentially less likely to generate a legitimate word. Therefore, less die are ideal for producing more 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.