Final Project Report

- Class: DS 5100
- Student Name: Darreion Bailey
- Student Net ID: rzu5vw
- 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/darreion/rzu5vw_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 class Die: """ Represents a die with multiple faces and weights, which can be rolled to randomly select a face. """ def __init__(self, faces): """ Initializes the die with given faces and default weights. """ if not isinstance(faces, np.ndarray): raise TypeError("Faces must be a numpy array.") if len(faces) != len(set(faces)): raise ValueError("Faces must be distinct.") self._df = pd.DataFrame({ 'face': faces, 'weight': np.ones(len(faces)) }).set_index('face') def change_weight(self, face, weight): """ Changes the weight of a specific face. """ if face not in self._df.index: raise IndexError("Face not found in the die.") if not isinstance(weight, (int, float)): raise TypeError("Weight must be a numeric type.") self._df.loc[face, 'weight'] = weight def roll(self, num_rolls=1): """ Rolls the die a specified number of times. """ return self._df.sample(n=num_rolls, replace=True, weights='weight').index.tolist() def show(self): """ Shows the current
```

state of the die. """ return self._df.copy() class Game: def __init__(self, dice): self._dice = dice self._results = pd.DataFrame() def play(self, num_rolls): results = {} for i, die in enumerate(self._dice): rolls = [die.roll()[0] for _ in range(num_rolls)] results[f"die_{i}"] = rolls self._results = pd.DataFrame(results) self._results.index.name = 'roll' def show(self, form='wide'): if form not in ['wide', 'narrow']: raise ValueError("Form must be 'wide' or 'narrow'.") if form == 'narrow': return self._results.stack().reset_index(name='outcome').rename(columns={'level_0': 'roll', 'level_1': 'die'}) return self._results.copy() class Analyzer: def __init__(self, game): if not isinstance(game, Game): raise ValueError("Provided object must be a Game instance.") self._game = game def jackpot(self): if self._game._results.empty: return 0 return self._game._results.apply(lambda row: row.nunique() == 1, axis=1).sum() def face_counts_per_roll(self): if self._game._results.empty: print("Results DataFrame is empty.") return pd.DataFrame() try: if not all(len(col) == len(self._game._results.index) for col in self._game._results.values.T): print("Columns vary in length.") return pd.DataFrame() return pd.crosstab(index=self._game._results.index, columns=self._game._results.values.flatten()) except Exception as e: print(f"Error in face_counts_per_roll: {e}") return pd.DataFrame() def combo_count(self): if self._game._results.empty: return pd.DataFrame() return self._game._results.apply(lambda row: tuple(sorted(row)), axis=1).value_counts().to_frame('counts') def permutation_count(self): if self._game._results.empty: return pd.DataFrame() return self._game._results.apply(lambda row: tuple(row), axis=1).value_counts().to_frame('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_init(self): """Test Die initialization""" faces = np.array(['A', 'B', 'C']) die = Die(faces) self.assertEqual(len(die.show()), 3) def test_init_with_non_numpy_array(self): """Test Die initialization shows error if faces are not a numpy array.""" with self.assertRaises(TypeError): Die(['A', 'B', 'C']) def test_init_with_non_unique_faces(self): """Test Die initialization shows error if faces are not unique.""" faces = np.array(['A', 'A', 'B']) with self.assertRaises(ValueError): Die(faces) def test_change_weight_invalid_face(self): """Test changing weight of a face not existing in the die.""" faces = np.array(['A', 'B', 'C']) die = Die(faces) with self.assertRaises(IndexError): die.change_weight('D', 2) def test_change_weight_invalid_weight_type(self): """Test changing weight with a non-numeric weight.""" faces = np.array(['A', 'B', 'C']) die = Die(faces) with self.assertRaises(TypeError): die.change_weight('A', 'heavy') def test_roll(self): """Test rolling the die produces results within expected faces.""" faces = np.array(['A', 'B', 'C']) die = Die(faces) results = die.roll(10) for result in results: self.assertIn(result, faces) def test_show(self): """Test show method returns DataFrame of correct format.""" faces = np.array(['A', 'B', 'C']) die = Die(faces) df = die.show() self.assertTrue('weight' in df.columns) class TestGame(unittest.TestCase): def test_game_play(self): """Test playing a game produces the correct number of results.""" faces = np.array(['1', '2', '3']) die1 = Die(faces) die2 = Die(faces) game = Game([die1, die2]) game.play(5) results = game.show() self.assertEqual(len(results), 5) def test_game_results_format(self): """Test the format of the game results in wide form.""" faces = np.array(['1', '2', '3']) die1 = Die(faces) die2 = Die(faces) game = Game([die1, die2]) game.play(3) results = game.show('wide') self.assertEqual(len(results.columns), 2) def test_invalid_form(self): """Test show method with invalid form raises ValueError.""" faces = np.array(['1', '2', '3']) die1 = Die(faces) die2 = Die(faces) game = Game([die1, die2]) game.play(3) with self.assertRaises(ValueError): game.show('invalid') class TestAnalyzer(unittest.TestCase): def setUp(self): die = Die(np.array([1, 2, 3, 4, 5, 6])) game = Game([die]) self.analyzer = Analyzer(game) # This is critical def test_jackpot_count(self): jackpots = self.analyzer.jackpot() print(f"Jackpot count returned: {jackpots}") self.assertIsInstance(jackpots, int) def test_combo_count(self): """Test combo count returns DataFrame with expected indices.""" faces = np.array(['1', '2', '3']) die1 = Die(faces) die2 = Die(faces) game =
```

Game([die1, die2]) game.play(100) analyzer = Analyzer(game) combo_df = analyzer.combo_count() self.assertTrue(isinstance(combo_df, pd.DataFrame)) def test_face_counts_per_roll(self): """Test face counts per roll returns correct DataFrame format.""" faces = np.array(['1', '2', '3']) die1 = Die(faces) die2 = Die(faces) game = Game([die1, die2]) game.play(10) analyzer = Analyzer(game) count_df = analyzer.face_counts_per_roll() self.assertTrue(isinstance(count_df, pd.DataFrame)) def test_permutation_count(self): """Test permutation count returns DataFrame with expected""" faces = np.array(['1', '2', '3']) die1 = Die(faces) die2 = Die(faces) game = Game([die1, die2]) game.play(100) analyzer = Analyzer(game) perm_df = analyzer.permutation_count() self.assertTrue(isinstance(perm_df, pd.DataFrame)) if __name__ == '__main__': unittest.main() ```

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).

-bash-4.4$python3 -m unittest -v montecarlo_unittest.py test_combo_count (montecarlo_unittest.TestAnalyzer) Test combo count returns DataFrame with expected indices. ... ok test_face_counts_per_roll (montecarlo_unittest.TestAnalyzer) Test face counts per roll returns correct DataFrame format. ... Error in face_counts_per_roll: arrays must all be same length ok test_jackpot_count (montecarlo_unittest.TestAnalyzer) ... Jackpot count returned: 0 ok test_permutation_count (montecarlo_unittest.TestAnalyzer) Test permutation count returns DataFrame with expected ... ok test_change_weight_invalid_face (montecarlo_unittest.TestDie) Test changing weight of a face not existing in the die. ... ok test_change_weight_invalid_weight_type (montecarlo_unittest.TestDie) Test changing weight with a non-numeric weight. ... ok test_init (montecarlo_unittest.TestDie) Test Die initialization with valid inputs. ... ok test_init_with_non_numpy_array (montecarlo_unittest.TestDie) Test Die initialization raises TypeError if faces are not a numpy array. ... ok test_init_with_non_unique_faces (montecarlo_unittest.TestDie) Test Die initialization raises ValueError if faces are not unique. ... ok test_roll (montecarlo_unittest.TestDie) Test rolling the die produces results within expected faces. ... ok test_show (montecarlo_unittest.TestDie) Test show method returns DataFrame of correct format. ... ok test_game_play (montecarlo_unittest.TestGame) Test playing a game produces the correct number of results. ... ok test_game_results_format (montecarlo_unittest.TestGame) Test the format of the game results in wide form. ... ok test_invalid_form (montecarlo_unittest.TestGame) Test show method with invalid form raises ValueError. ... ok -- Ran 14 tests in 0.724s 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 as 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)
```

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

NAME
    Montecarlo.montecarlo

CLASSES
    builtins.object
        Analyzer
        Die
        Game

    class Analyzer(builtins.object)
     |  Analyzer(game)
     |
     |  Methods defined here:
     |
     |  __init__(self, game)
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  combo_count(self)
     |
     |  face_counts_per_roll(self)
     |
     |  jackpot(self)
     |
     |  permutation_count(self)
     |
     |  ----------------------------------------------------------------------
     |  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, weights=None)
     |
     |  Represents a die with multiple faces and weights, which can be rolled to ran
domly select a face.
     |
     |  Methods defined here:
     |
     |  __init__(self, faces, weights=None)
     |      Initialize self.  See help(type(self)) for accurate signature.
     |
     |  change_weight(self, face, weight)
     |      Changes the weight of a specific face.
     |
     |  roll(self, num_rolls=1)
     |      Rolls the die a specified number of times.
     |
     |  show(self)
     |      Shows the current state of the die.
     |
```

```
 |  ----------------------------------------------------------------------
 |  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(dice)
 |
 |  Methods defined here:
 |
 |  __init__(self, dice)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  play(self, num_rolls)
 |
 |  show(self, form='wide')
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)

FILE
    /sfs/qumulo/qhome/rzu5vw/Documents/MSDS/DS5100/DS5100-darreion/lessons/M10/Monte
carlo/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/darreion/rzu5vw_ds5100_montecarlo/edit/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).

Pasted code

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
from Montecarlo.montecarlo import Die
from Montecarlo.montecarlo import Game
from Montecarlo.montecarlo import Analyzer
fair_coin = Die(np.array(['H', 'T']))

unfair_coin = Die(np.array(['H', 'T']), np.array([5, 1]))
```

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

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

In [4]:
```
game1 = Game([fair_coin, fair_coin])
game1.play(1000)
```

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]:
```
game2 = Game([unfair_coin, unfair_coin, fair_coin])
game2.play(1000)
```

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]:
```
analyzer1 = Analyzer(game1)
jackpots1 = analyzer1.jackpot()

analyzer2 = Analyzer(game2)
jackpots2 = analyzer2.jackpot()
```

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

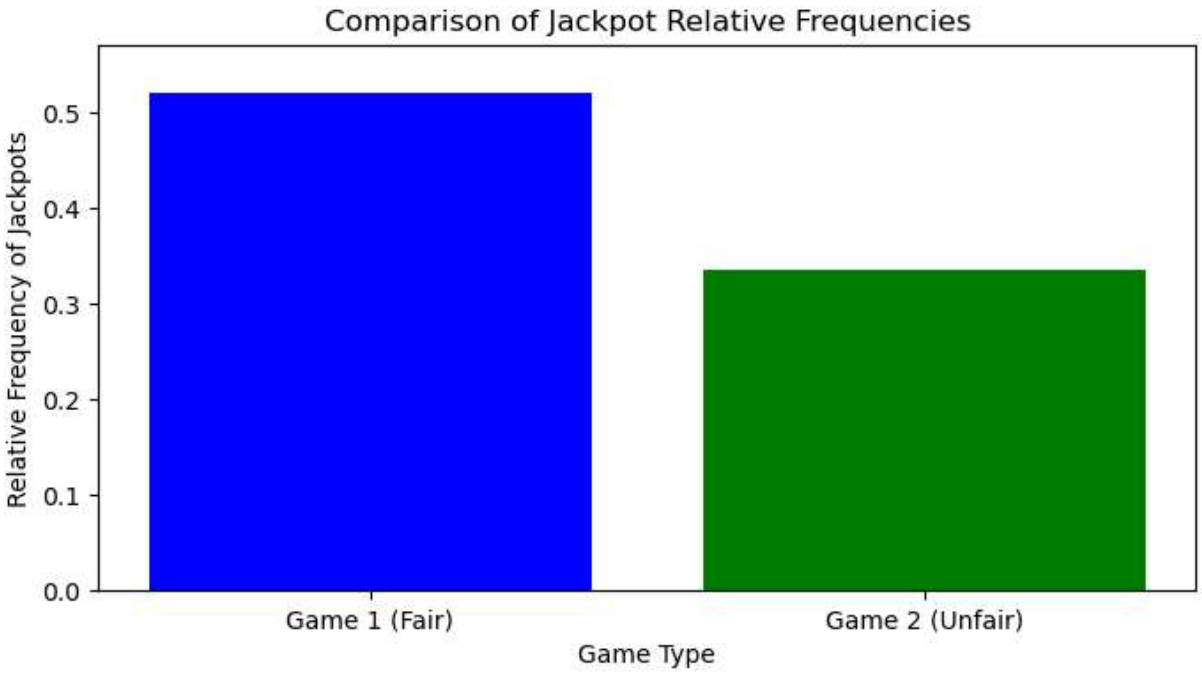
- Both relative frequencies computed (1).

In [7]:
```
relative_frequency1 = jackpots1 / 1000
relative_frequency2 = jackpots2 / 1000
```

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

- Bar chart plotted and correct (1).

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


labels = ['Game 1 (Fair)', 'Game 2 (Unfair)']
frequencies = [relative_frequency1, relative_frequency2]

plt.figure(figsize=(8, 4))
plt.bar(labels, frequencies, color=['blue', 'green'])
plt.xlabel('Game Type')
plt.ylabel('Relative Frequency of Jackpots')
plt.title('Comparison of Jackpot Relative Frequencies')
plt.ylim(0, max(frequencies) + 0.05)
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]:
```
dice_faces = np.array([1, 2, 3, 4, 5, 6])
die1 = Die(dice_faces)
die2 = Die(dice_faces)
die3 = Die(dice_faces)
```

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.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 [11]:
```
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 [12]:
```
game1 = Game([die3, die3, die3, die3, die3])  # Using the third die which is still
game1.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.

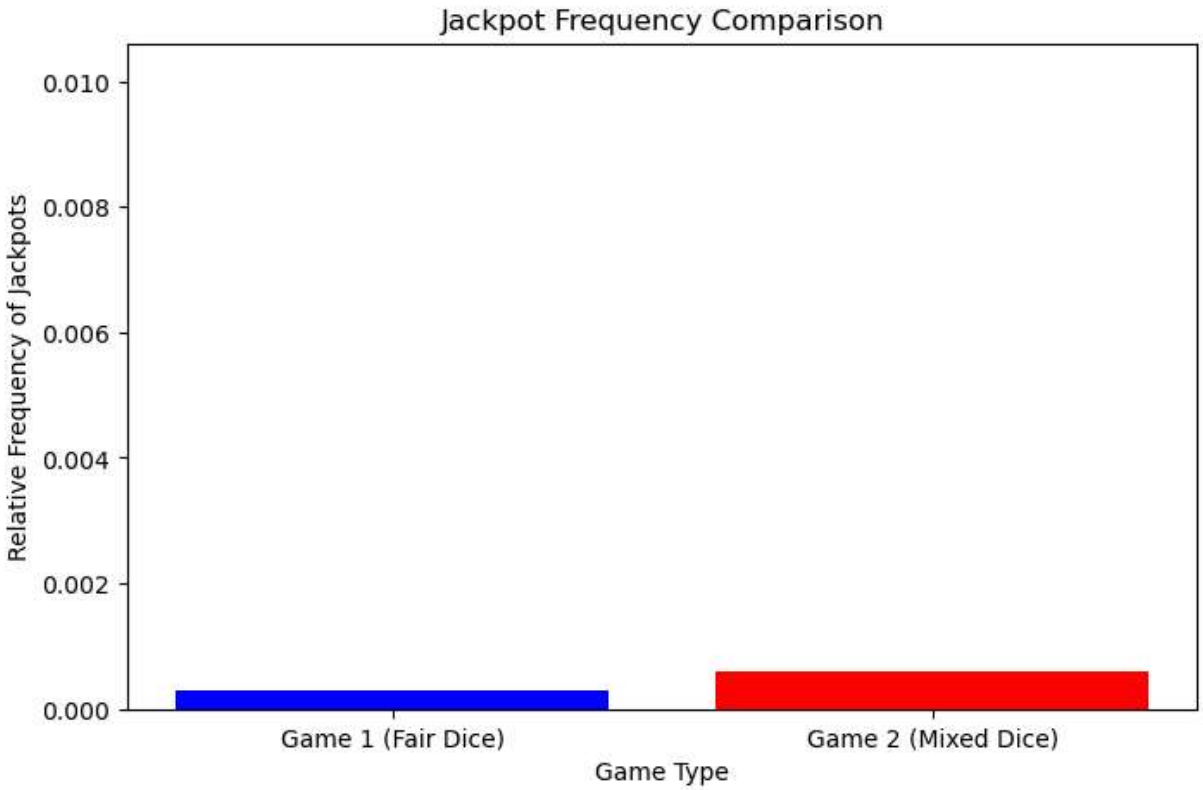
- Game class properly instantiated (1).
- Play method called properly (1).

In [13]:
```
game2 = Game([die1, die1, die2, die3, die3])
game2.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 [14]:
```
analyzer1 = Analyzer(game1)
analyzer2 = Analyzer(game2)

jackpots1 = analyzer1.jackpot()
jackpots2 = analyzer2.jackpot()

relative_frequency1 = jackpots1 / 10000
relative_frequency2 = jackpots2 / 10000

import matplotlib.pyplot as plt

labels = ['Game 1 (Fair Dice)', 'Game 2 (Mixed Dice)']
frequencies = [relative_frequency1, relative_frequency2]

plt.figure(figsize=(8, 5))
plt.bar(labels, frequencies, color=['blue', 'red'])
plt.xlabel('Game Type')
plt.ylabel('Relative Frequency of Jackpots')
plt.title('Jackpot Frequency Comparison')
plt.ylim(0, max(frequencies) + 0.01)
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]:
```
import pandas as pd
import numpy as np

letter_data = pd.read_csv('english_letters.txt', delim_whitespace=True, header=None


letters = letter_data['letter'].values
frequencies = letter_data['frequency'].values
alphabet_die = Die(letters, frequencies)
```

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

- Game play method properly called (1).

In [16]:
```
game4 = Game([alphabet_die] * 4)
game4.play(1000)
```

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

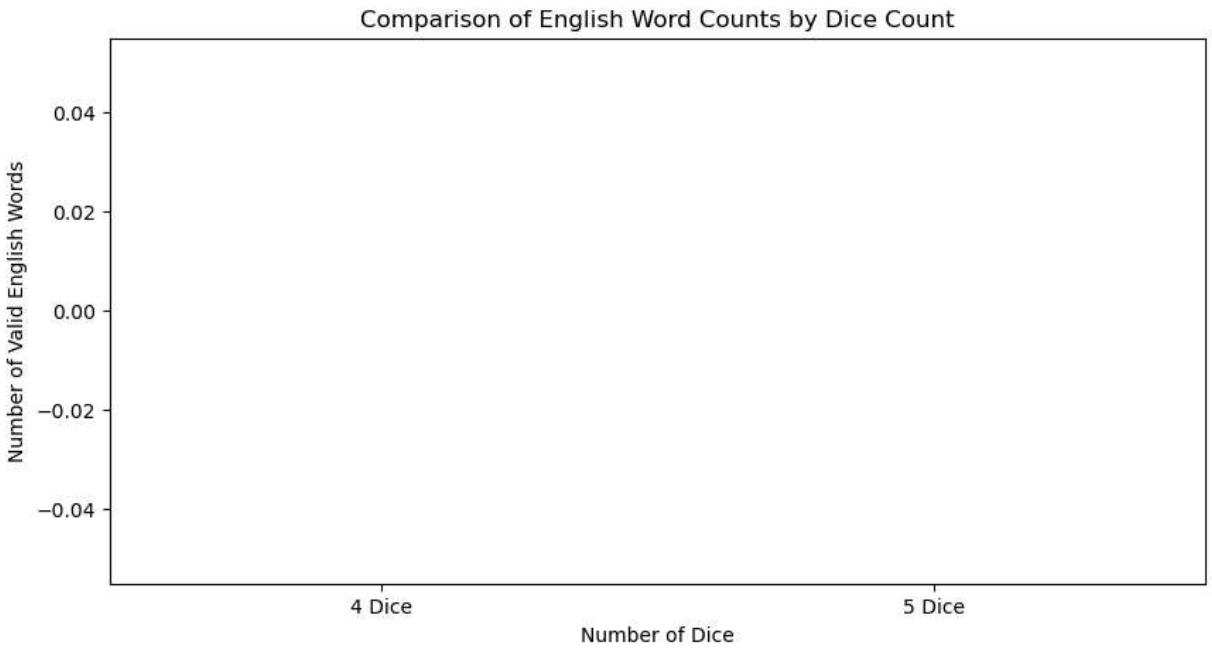
- Use permutation method (1).
- Get count as difference between permutations and vocabulary (1).

In [17]:
```
with open('scrabble_words.txt', 'r') as file:
    english_words = set(word.strip().upper() for word in file.readlines())


analyzer4 = Analyzer(game4)



valid_words4 = analyzer4.permutation_count()


valid_words4['is_word'] = valid_words4.index.isin(english_words)


count_valid_words4 = valid_words4[valid_words4['is_word']]['counts'].sum()
```

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 [18]:
```
game5 = Game([alphabet_die] * 5)
game5.play(1000)


analyzer5 = Analyzer(game5)
valid_words5 = analyzer5.permutation_count()
valid_words5['is_word'] = valid_words5.index.isin(english_words)
count_valid_words5 = valid_words5[valid_words5['is_word']]['counts'].sum()


import matplotlib.pyplot as plt


results = [count_valid_words4, count_valid_words5]
labels = ['4 Dice', '5 Dice']

plt.figure(figsize=(10, 5))
plt.bar(labels, results, color=['blue', 'green'])
plt.xlabel('Number of Dice')
plt.ylabel('Number of Valid English Words')
plt.title('Comparison of English Word Counts by Dice Count')
plt.show()
```



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 [ ]: