Final Project Report

- Class: DS 5100
- Student Name: Thomas Burrell
- Student Net ID: tmb9ccd
- This URL: https://github.com/thomasbva/tmb9ccd_ds5100_montecarlo/blob/main/finalprojreport.pdf

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/thomasbva/tmb9ccd_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 pandas as pd
import numpy as np
import random
from itertools import product
from itertools import combinations_with_replacement

class Die:
    """
    A class representing a n-sided die.

    This class provides functionality to create a dataframe that
displays the faces
    of a standard six-sided die along with their corresponding
weights, and allows
    the user to roll the die to get a random outcome.


    Methods:
    -------
    change_weight(face, new_weight)
        Changes the weights of faces of a n-sided die. The user can
determine
        which faces to change.

        Takes two parameters, face, which is a string or integer and
weight which is
        a numeric value.


     roll_die(rolls)
         Rolls the die and returns a random outcome.

         Takes one parameter, the number of rolls a user wants to
have.


    die_current_state()
        Return the current state of the die, which is the faces and
corresponding
        weights.

    """

    face_weight_df = pd.DataFrame({})

    def __init__(self, faces):
        '''Takes in faces of the die. Initializer method for the Die
class.'''

        self.faces = faces
        self.weights = np.ones(len(faces))

        if not type(faces) is np.ndarray:
            raise TypeError("Only numpy arrays are allowed.")

        if not len(faces) == len(np.unique(faces)):
            raise ValueError("Values of numpy array are not
distinct.")

        self.face_weight_df =
pd.DataFrame({'weights':np.ones(len(faces))}, index=faces)



    def change_weight(self, face, new_weight):
        '''Method to change the weights of dice. Parameters are the
face you want to change, and the
            new weight of the face.'''

        try:
            if face in self.face_weight_df.index:
                self.face_weight_df.loc[face] = new_weight
            else:
                raise IndexError("The face you are trying to replace
does not exist.")
        except:
            raise TypeError("Enter a valid data type")

    def roll_die(self, rolls = 1):
        '''Method to roll the dice. Default number of rolls is 1, but
user can specify how many are needed.'''

        roll_outcome = random.choices(self.face_weight_df.index,
weights=self.face_weight_df['weights'], k=rolls)
        return roll_outcome

    def die_current_state(self):
        '''Method to get the current state of the die. Returns
dataframe of faces and weights.'''

        return self.face_weight_df




class Game:
    """
    A class representing a game of n amount of dice.

    This class provides functionality to play a game of rolling n-
sided dice,
    with random outcomes. It uses Die objects created from the Die
class.

    Methods:
    -------
    play(die_rolls)
        Rolls n-sided die a specified number of times. This value is
an integer.

        Takes one parameters, die rolls. This is how many times we
want to roll
        the dice previously specified in the Die object.
```

```
     results_recent_play(wide=True)
         Returns the results of the most recent play, which shows the
dice number in
         the columns and the roll number as the rows. The cells
represent the outcome,
         which is the face that was rolled.

         Takes one parameter, wide, which returns a wide form
dataframe if left to default.
         It returns a narrow dataframe if wide=False.


    get_faces()
        Return the faces of the dice.

    """

    df_die = pd.DataFrame({})

    def __init__(self, list_of_die_obj):
        '''Takes in Die object. Initializer method for the Game
class.'''

        self.list_of_die_obj = list_of_die_obj

    def play(self, die_rolls):
        '''Takes in how many rolls of dice are wanted. Takes in an
integer.'''

        dde = []
        for dice in self.list_of_die_obj:
            dde.append(dice.roll_die(rolls=die_rolls))
        self.df_die = pd.DataFrame(dde)
        self.df_die = self.df_die.transpose()
        self.df_die.index += 1
        self.df_die.columns += 1

    def results_recent_play(self, wide=True):
        '''Returns the recent results of the dice roll in wide format
my default.'''

        if wide == True:
            return self.df_die
        elif wide == False:
            return self.df_die.transpose().unstack()
        else:
            raise ValueError('The wide parameter should be either
True or False.')

    def get_faces(self):
        '''Returns faces of the dice.'''
        return
list(self.list_of_die_obj[0].die_current_state().index)



class Analyzer:
    """
    A class representing analysis of a Game object.

    This class provides functionality to determine if a game resulted
in a jackpot,
    the specific combination and permutation that was rolled.

    Methods:
    -------
    jackpot()
        A jackpot is a result in which all faces are the same, e.g.
all ones for a six-sided die.

        Returns how many times the game resulted in a jackpot as an
integer.


     get_faces_analyzer()
         Return the faces of the dice.



    face_counts_per_roll()
        Computes how many times a given face is rolled in each event.
Returns a data frame of results.
        The data frame has an index of the roll number, face values
as columns, and count values in the cells.

    permutation_count()
        Computes the distinct permutations of faces rolled, along
with their counts.
        Returns a data frame of results. The data frame has an
MultiIndex of distinct permutations and a
        column for the associated counts.

    combo_count()
        Computes the distinct combinations of faces rolled, along
with their counts.
        Returns a data frame of results. The data frame has an
MultiIndex of distinct combinations and a
        column for the associated counts.
    """

    def __init__(self, game_obj):
        '''Initializer method that takes in a game object.'''

        self.game_obj = game_obj
        if type(game_obj) != Game:
            raise ValueError("The passed value is not a game 
object.")

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

        try:
            count_jackpot = 0

            for i in self.game_obj.results_recent_play().transpose():
                if 
len(set(self.game_obj.results_recent_play().iloc[i]))==1:
                    count_jackpot += 1
        except IndexError:
            pass
        return count_jackpot

    def get_faces_analyzer(self):
        '''Get the faces of the game object.'''

        return self.game_obj.get_faces()




    def face_counts_per_roll(self):
        '''Get the face counts per roll of the game object.'''

        faces_df = pd.DataFrame(self.game_obj.get_faces())
        faces_df.columns = ['john']

        yuh = self.game_obj.results_recent_play().transpose()
        yuh.rename(columns={1:'john'}, inplace=True)

        emp = []
        for i in yuh:
            ddd = pd.concat([faces_df, yuh[i]])
            eee = ddd['john'].fillna(ddd[0])
            emp.append(eee.value_counts()-1)

        final = pd.DataFrame(emp).reset_index(drop=True)
        final.index += 1

        final = final.reindex(sorted(final.columns), axis=1)

        self.final = final

        return final


    def permutation_count(self):
        '''Method to compute the distinct permutations of faces
```

```
rolled, along with their counts.'''


        def distinct_combinations(values, n):
            return set(product(values, repeat=n))

        # Compute distinct combinations
        combinations =
distinct_combinations(self.game_obj.get_faces(),
len(self.game_obj.results_recent_play()))

        # Create a DataFrame from the distinct combinations
        columns = [f'Die{i+1}' for i in
range(len(self.game_obj.results_recent_play()))]
        combos = pd.DataFrame(list(combinations), columns=columns)

        combos = combos.set_index(columns)
        combos['combos'] = combos.index
        combos['combos'].astype(str)

        outcome = self.game_obj.results_recent_play()
        outcome = outcome.transpose()
        outcome_tuples = outcome.apply(lambda row: tuple(row),
axis=1)


        final_combo = pd.concat([combos,outcome_tuples])
        final_combo['Col1'] = final_combo['combos'].astype(str)
        final_combo['Col2'] = final_combo[0].astype(str)
        final_combo = final_combo[['Col1', 'Col2']]
        final_combo = final_combo.replace('nan', '')
        final_combo['Col3'] = final_combo['Col1'] +
final_combo['Col2']
        final_combo = final_combo['Col3'].value_counts()-1
        final_combo = final_combo.sort_index()
        #final_combo = final_combo.to_frame()
        combos = combos.sort_index()
        combos['value_counts'] = list(final_combo)
        combos = combos.drop(columns=['combos'])

        return combos



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

        def distinct_combinations(lst, input_list_length):
            unique_combinations = set()

            for combo in combinations_with_replacement(lst,
input_list_length):
                unique_combinations.add(tuple(sorted(combo)))
            return [list(combo) for combo in unique_combinations]

        combinations =
distinct_combinations(self.game_obj.get_faces(),
len(self.game_obj.results_recent_play()))


        columns = [f'Die{i+1}' for i in
range(len(self.game_obj.results_recent_play()))]
        combos = pd.DataFrame(list(combinations), columns=columns)

        combos = combos.set_index(columns)
        combos['combos'] = combos.index
        combos['combos'].astype(str)

        outcome = self.game_obj.results_recent_play()
        outcome = outcome.transpose()
        outcome_tuples = outcome.apply(lambda row: tuple(row),
axis=1)

        sorted_tuples = []

        for plop in range(len(outcome_tuples)):

sorted_tuples.append(tuple(sorted(outcome_tuples.iloc[plop])))

        outcome_tuples = pd.Series(sorted_tuples)
        final_combo = pd.concat([combos,outcome_tuples])
        final_combo['Col1'] = final_combo['combos'].astype(str)
        final_combo['Col2'] = final_combo[0].astype(str)
        final_combo = final_combo[['Col1', 'Col2']]
        final_combo = final_combo.replace('nan', '')
        final_combo['Col3'] = final_combo['Col1'] +
final_combo['Col2']
        final_combo = final_combo['Col3'].value_counts()-1
        final_combo = final_combo.sort_index()
        combos = combos.sort_index()
        combos['value_counts'] = list(final_combo)
        combos = combos.drop(columns=['combos'])


        return combos
```

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
from montecarlo import Die
from montecarlo import Game
from montecarlo import Analyzer
import pandas as pd
import numpy as np
import random
from itertools import product
from itertools import combinations_with_replacement



class montecarloTestSuite(unittest.TestCase):

    def test_DIECLASS_default_weight(self):
        # add a book and test if it is in `book_list`.

        die1 = Die(np.array(['H', 'T']))
        self.assertEqual(die1.die_current_state()['weights'][0],
[1.0])

    def test_DIECLASS_change_weight(self):
        die1 = Die(np.array(['H', 'T']))

        die1.change_weight('H', 2.0)
        self.assertEqual(die1.die_current_state()['weights'][0],
[2.0])

    def test_DIECLASS_rolldie_length(self):
        die1 = Die(np.array(['H', 'T']))

        self.assertEqual(len(die1.roll_die(4)), 4)

    def test_DIECLASS_rolldie_datatype(self):
        die1 = Die(np.array(['H', 'T']))

        self.assertEqual(type(die1.roll_die(4)), list)

    def test_DIECLASS_currentstate(self):
        die1 = Die(np.array(['H', 'T']))

        self.assertEqual(type(die1.die_current_state()),
pd.DataFrame)



    def test_GAMECLASS_play(self):
        die1 = Die(np.array(['H', 'T']))
        list_of_die = [die1]

        game_obj = Game(list_of_die)
        game_obj.play(4)

        self.assertEqual(len(game_obj.results_recent_play()), 4)

    def test_GAMECLASS_resultsrecentplay(self):
        die1 = Die(np.array(['H', 'T']))
        list_of_die = [die1]

        game_obj = Game(list_of_die)
        game_obj.play(4)

        self.assertEqual(type(game_obj.results_recent_play()),
pd.DataFrame)

    def test_GAMECLASS_getfaces(self):
        die1 = Die(np.array(['H', 'T']))
        list_of_die = [die1]

        game_obj = Game(list_of_die)

        self.assertEqual(game_obj.get_faces(),['H', 'T'])



    def test_ANALYZERCLASS_jackpot(self):
        die1 = Die(np.array(['H', 'T']))
        list_of_die = [die1]

        game_obj = Game(list_of_die)
        game_obj.play(2)

        analyze_obj = Analyzer(game_obj)


        self.assertEqual(type(analyze_obj.jackpot()), int)

    def test_ANALYZERCLASS_get_faces_analyzer(self):
        die1 = Die(np.array(['H', 'T']))
        list_of_die = [die1]

        game_obj = Game(list_of_die)
        game_obj.play(2)

        analyze_obj = Analyzer(game_obj)


        self.assertEqual(analyze_obj.get_faces_analyzer(),['H', 'T'])


    def test_ANALYZERCLASS_face_counts_per_roll(self):
        die1 = Die(np.array(['H', 'T']))
        list_of_die = [die1]
        game_obj = Game(list_of_die)
        game_obj.play(2)

        analyze_obj = Analyzer(game_obj)

        self.assertEqual(type(analyze_obj.face_counts_per_roll()),
pd.DataFrame)

    def test_ANALYZERCLASS_permutation_count(self):
        die1 = Die(np.array(['H', 'T']))
        list_of_die = [die1]

        game_obj = Game(list_of_die)
        game_obj.play(2)

        game_obj.get_faces()
        game_obj.results_recent_play()

        analyze_obj = Analyzer(game_obj)
        analyze_obj.face_counts_per_roll()

        self.assertEqual(type(analyze_obj.permutation_count()),
pd.DataFrame)

    def test_ANALYZERCLASS_combo_count(self):
        die1 = Die(np.array(['H', 'T']))
        list_of_die = [die1]

        game_obj = Game(list_of_die)
        game_obj.play(2)

        game_obj.get_faces()
        game_obj.results_recent_play()

        analyze_obj = Analyzer(game_obj)
        analyze_obj.face_counts_per_roll()

        self.assertEqual(type(analyze_obj.combo_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_ANALYZERCLASS_combo_count (__main__.montecarloTestSuite) ... ok
test_ANALYZERCLASS_face_counts_per_roll
(__main__.montecarloTestSuite) ... ok
test_ANALYZERCLASS_get_faces_analyzer (__main__.montecarloTestSuite)
... ok
test_ANALYZERCLASS_jackpot (__main__.montecarloTestSuite) ... ok
test_ANALYZERCLASS_permutation_count (__main__.montecarloTestSuite)
... ok
test_DIECLASS_change_weight (__main__.montecarloTestSuite) ... ok
test_DIECLASS_currentstate (__main__.montecarloTestSuite) ... ok
test_DIECLASS_default_weight (__main__.montecarloTestSuite) ... ok
test_DIECLASS_rolldie_datatype (__main__.montecarloTestSuite) ... ok
test_DIECLASS_rolldie_length (__main__.montecarloTestSuite) ... ok
test_GAMECLASS_getfaces (__main__.montecarloTestSuite) ... ok
test_GAMECLASS_play (__main__.montecarloTestSuite) ... ok
test_GAMECLASS_resultsrecentplay (__main__.montecarloTestSuite) ...
ok

---------------------------------------------------------------------
-
Ran 13 tests in 0.041s

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
```

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:

NAME
    montecarlo - Created on Thu Aug  3 08:09:42 2023

DESCRIPTION
    @author: treyb

CLASSES
    builtins.object
        Analyzer
        Die
        Game

    class Analyzer(builtins.object)
     |  Analyzer(game_obj)
     |
     |  A class representing analysis of a Game object.
     |
     |  This class provides functionality to determine if a game resulted in a jackpo
t,
     |  the specific combination and permutation that was rolled.
     |
     |  Methods:
     |  -------
     |  jackpot()
     |      A jackpot is a result in which all faces are the same, e.g. all ones for
a six-sided die.
     |
     |      Returns how many times the game resulted in a jackpot as an integer.
     |
     |
     |   get_faces_analyzer()
     |       Return the faces of the dice.
     |
     |
     |  face_counts_per_roll()
     |      Computes how many times a given face is rolled in each event. Returns a d
ata frame of results.
     |      The data frame has an index of the roll number, face values as columns, a
nd count values in the cells.
     |
     |  permutation_count()
     |      Computes the distinct permutations of faces rolled, along with their coun
ts.
     |      Returns a data frame of results. The data frame has an MultiIndex of dist
inct permutations and a
     |      column for the associated counts.
     |
     |  combo_count()
     |      Computes the distinct combinations of faces rolled, along with their coun
ts.
     |      Returns a data frame of results. The data frame has an MultiIndex of dist
inct combinations and a
     |      column for the associated counts.
     |
     |  Methods defined here:
     |
     |  __init__(self, game_obj)
     |      Initializer method that takes in a game object.
     |  
     |  combo_count(self)
     |      Method to compute the distinct combinations of faces rolled, along with t
heir counts.
     |  
     |  face_counts_per_roll(self)
     |      Get the face counts per roll of the game object.
     |  
     |  get_faces_analyzer(self)
     |      Get the faces of the game object.
     |  
     |  jackpot(self)
     |      A jackpot is a result in which all faces are the same. Computes how many
times the game resulted in a jackpot.
     |  
     |  permutation_count(self)
     |      Method to compute the distinct permutations of faces rolled, along with t
heir 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)
     |  
     |  A class representing a n-sided die.
     |  
     |  This class provides functionality to create a dataframe that displays the fac
es
     |  of a standard six-sided die along with their corresponding weights, and allow
s
     |  the user to roll the die to get a random outcome.
     |  
     |  
     |  Methods:
     |  -------
     |  change_weight(face, new_weight)
     |      Changes the weights of faces of a n-sided die. The user can determine
     |      which faces to change.
     |  
     |      Takes two parameters, face, which is a string or integer and weight which
is
     |      a numeric value.
     |  
     |  
     |   roll_die(rolls)
     |       Rolls the die and returns a random outcome.
     |  
     |       Takes one parameter, the number of rolls a user wants to have.
     |  
     |  
     |  die_current_state()
     |      Return the current state of the die, which is the faces and corresponding
     |      weights.
     |
     |  Methods defined here:
     |
     |  __init__(self, faces)
     |      Takes in faces of the die. Initializer method for the Die class.
     |
     |  change_weight(self, face, new_weight)
     |      Method to change the weights of dice. Parameters are the face you want to
change, and the
     |      new weight of the face.
     |
     |  die_current_state(self)
     |      Method to get the current state of the die. Returns dataframe of faces an
d weights.
     |
     |  roll_die(self, rolls=1)
     |      Method to roll the dice. Default number of rolls is 1, but user can speci
fy how many are needed.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  face_weight_df = Empty DataFrame
     |  Columns: []
     |  Index: []

    class Game(builtins.object)
     |  Game(list_of_die_obj)
     |
     |  A class representing a game of n amount of dice.
     |
     |  This class provides functionality to play a game of rolling n-sided dice,
     |  with random outcomes. It uses Die objects created from the Die class.
     |
     |  Methods:
     |  -------
     |  play(die_rolls)
     |      Rolls n-sided die a specified number of times. This value is an integer.
     |
     |      Takes one parameters, die rolls. This is how many times we want to roll
     |      the dice previously specified in the Die object.
     |
     |
     |   results_recent_play(wide=True)
     |       Returns the results of the most recent play, which shows the dice number
in
     |       the columns and the roll number as the rows. The cells represent the out
come,
     |       which is the face that was rolled.
     |
     |       Takes one parameter, wide, which returns a wide form dataframe if left t
```

```
o default.
     |       It returns a narrow dataframe if wide=False.
     |
     |
     |  get_faces()
     |      Return the faces of the dice.
     |
     |  Methods defined here:
     |
     |  __init__(self, list_of_die_obj)
     |      Takes in Die object. Initializer method for the Game class.
     |
     |  get_faces(self)
     |      Returns faces of the dice.
     |
     |  play(self, die_rolls)
     |      Takes in how many rolls of dice are wanted. Takes in an integer.
     |
     |  results_recent_play(self, wide=True)
     |      Returns the recent results of the dice roll in wide format my default.
     |
     |  ----------------------------------------------------------------------
     |  Data descriptors defined here:
     |
     |  __dict__
     |      dictionary for instance variables (if defined)
     |
     |  __weakref__
     |      list of weak references to the object (if defined)
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  df_die = Empty DataFrame
     |  Columns: []
     |  Index: []

FILE
    c:\users\treyb\box\msds\ds 5100 (programming)\final project\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/thomasbva/tmb9ccd_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) C:\Users\treyb>pip install montecarlo Collecting montecarlo Downloading montecarlo-0.1.17.tar.gz (1.3 kB)
Building wheels for collected packages: montecarlo Building wheel for montecarlo (setup.py) ... done Created
wheel for montecarlo: filename=montecarlo-0.1.17-py3-none-any.whl size=1881
sha256=44e7dc3d07e8ad678f50b2fffda740f5b4006db08d0e8eac41e37cd176689726 Stored in directory:
c:\users\treyb\appdata\local\pip\cache\wheels\ea\60\c6\9de9b2f21cd9b2fcf3fef492ffb56de76b3bb1dc79dc508ac6
Successfully built montecarlo Installing collected packages: montecarlo Successfully installed montecarlo-0.1.17
```

Scenarios

Use code blocks to perform the tasks for each scenario.

Be sure the outputs are visible before submitting.

Scenario 1: A 2-headed Coin (9)

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

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

In [3]:

```
from montecarlo import Die
from montecarlo import Game
from montecarlo import Analyzer
import pandas as pd
import numpy as np
import random
from itertools import product
from itertools import combinations_with_replacement
import matplotlib.pyplot as plt
```

In [4]:

```
coin1 = Die(np.array(['H', 'T']))
coin2 = Die(np.array(['H', 'T']))
coin2.change_weight('H', 5)

coin1.die_current_state()
```

Out[4]:

	weights
H	1.0
T	1.0

In [5]:

```
coin2.die_current_state()
```

Out[5]:

	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 [6]:

```
faircoin1 = Die(np.array(['H', 'T']))
faircoin2 = Die(np.array(['H', 'T']))
list_of_fair_coins = [faircoin1, faircoin2]
game_obj_fair = Game(list_of_fair_coins)
game_obj_fair.play(1000)
```

In [7]:

```
game_obj_fair.results_recent_play()
```

Out[7]:

	1	2
1	H	H
2	H	H
3	H	H
4	T	H
5	T	H
...	...	...
996	H	H
997	H	T
998	T	T
999	H	T
1000	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 [8]:

```
fair = Die(np.array(['H', 'T']))
unfair = Die(np.array(['H', 'T']))
unfair.change_weight('H', 5)
list_of_mixed_coins = [unfair, unfair, fair]
```

```
game_obj_mixed = Game(list_of_mixed_coins)
game_obj_mixed.play(1000)
```

In [9]:

```
game_obj_mixed.results_recent_play()
```

Out[9]:

	1	2	3
1	H	T	T
2	H	H	T
3	H	H	T
4	H	H	T
5	H	H	H
...	...	...	...
996	H	H	T
997	H	H	T
998	H	H	T
999	T	T	H
1000	H	T	T

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 [10]:

```
analyzer_obj_mixed = Analyzer(game_obj_mixed)
analyzer_obj_mixed.jackpot()
```

Out[10]:

```
361
```

In [11]:

```
analyzer_obj_fair = Analyzer(game_obj_fair)
analyzer_obj_fair.jackpot()
```

Out[11]:

```
490
```

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 [12]:

```
mixed_rel_freq = analyzer_obj_mixed.jackpot()/1000
mixed_rel_freq
```

Out[12]:

```
0.361
```

```
In [13]: fair_rel_freq = analyzer_obj_fair.jackpot()/1000
         fair_rel_freq
```

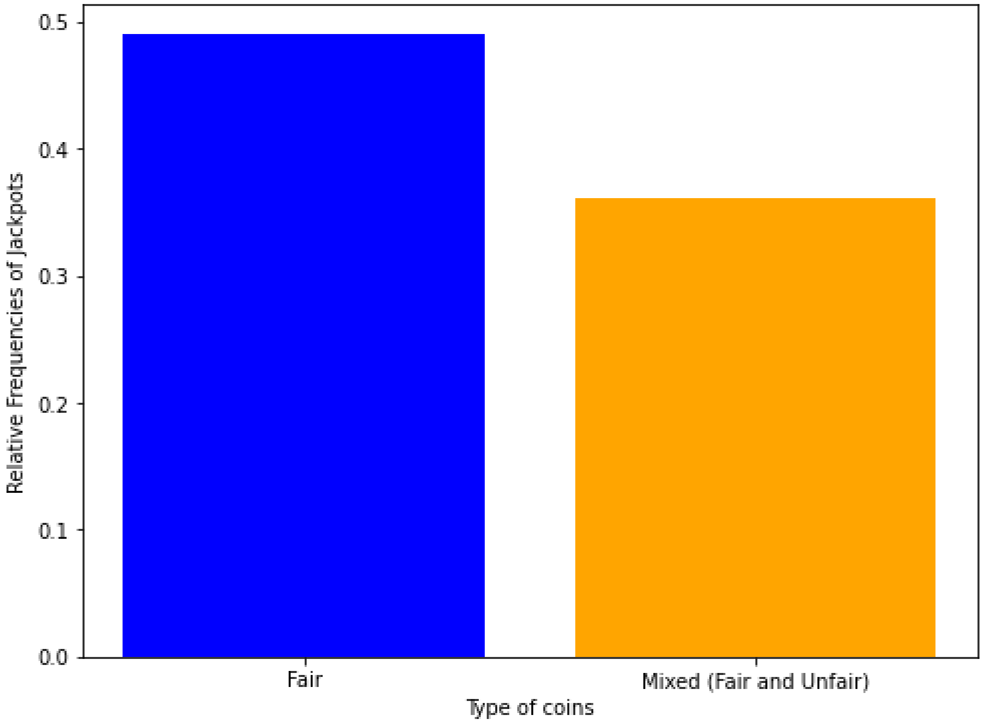
```
Out[13]: 0.49
```

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

- Bar chart plotted and correct (1).

```
In [14]: plt.figure(figsize=(8, 6))
         plt.bar(['Fair', 'Mixed (Fair and Unfair)'], [fair_rel_freq, mixed_rel_freq], color=['
         plt.xlabel('Type of coins')
         plt.ylabel('Relative Frequencies of Jackpots')
```

```
Out[14]: Text(0, 0.5, 'Relative Frequencies of Jackpots')
```



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 [15]: 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]))
         die1.die_current_state()
```

Out[15]:

	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 [16]:
```
die1.change_weight(6, 5)
die1.die_current_state()
```

Out[16]:

	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 [17]:
```
die2.change_weight(1, 5)
die2.die_current_state()
```

Out[17]:

	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 [18]:

```
list_of_fair_die = [die3,die3,die3,die3,die3]
game_obj_scen2_task4 = Game(list_of_fair_die)
game_obj_scen2_task4.play(10000)
game_obj_scen2_task4.results_recent_play()
```

Out[18]:

	1	2	3	4	5
1	6	2	1	2	5
2	3	4	1	2	3
3	6	1	3	4	3
4	3	4	5	4	5
5	5	2	1	1	5
...	...	...	...	...	...
9996	4	1	6	1	1
9997	2	3	5	1	4
9998	1	3	3	4	3
9999	2	3	1	5	1
10000	4	1	2	5	6

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 [19]:

```
list_of_die = [die1,die2,die3,die3,die3]
game_obj_scen2_task5 = Game(list_of_die)
game_obj_scen2_task5.play(10000)
game_obj_scen2_task5.results_recent_play()
```

Out[19]:

	1	2	3	4	5
1	6	5	5	5	4
2	6	1	3	4	6
3	2	3	1	4	3
4	6	5	4	3	3
5	6	1	1	5	6
...	...	...	...	...	...
9996	6	1	5	3	4
9997	1	2	3	1	2
9998	6	1	6	6	2
9999	4	6	1	2	1
10000	6	6	4	2	5

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.

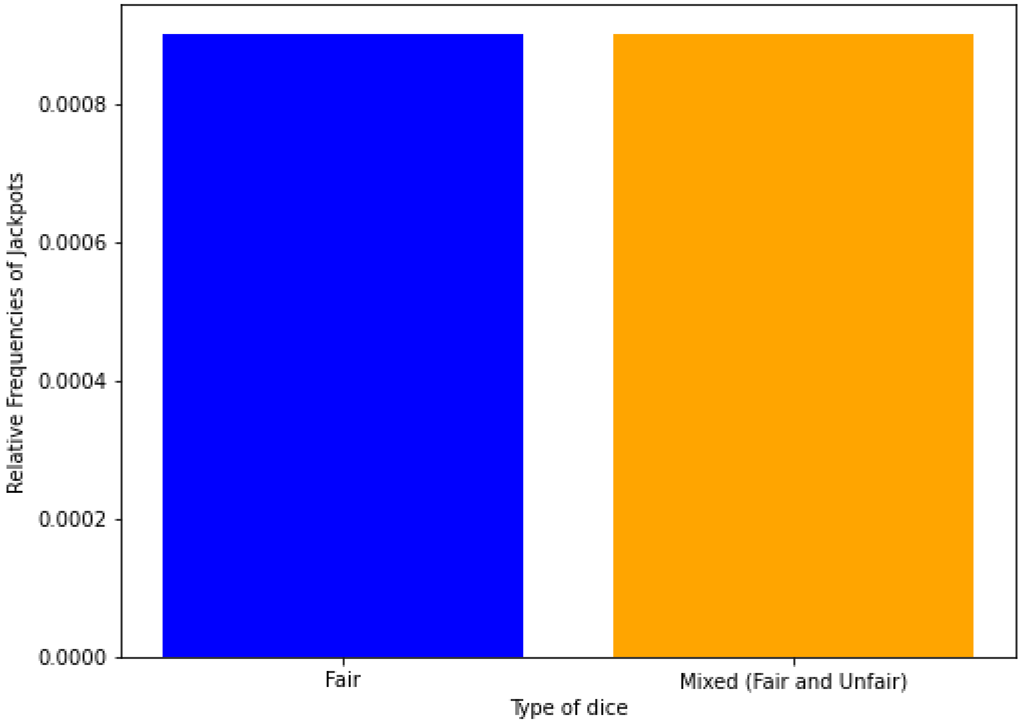
- Jackpot methods called (1).
- Graph produced (1).

In [20]:

```
analyzer_obj_fair = Analyzer(game_obj_scen2_task4)
analyzer_obj_fair.jackpot()

analyzer_obj_unfair = Analyzer(game_obj_scen2_task5)
analyzer_obj_unfair.jackpot()

freq1 = analyzer_obj_fair.jackpot()/10000
freq2 = analyzer_obj_unfair.jackpot()/10000

plt.figure(figsize=(8, 6))
plt.bar(['Fair', 'Mixed (Fair and Unfair)'], [freq1, freq2], color=['blue', 'orange']
plt.xlabel('Type of dice')
plt.ylabel('Relative Frequencies of Jackpots')
```

Out[20]:

```
Text(0, 0.5, 'Relative Frequencies of Jackpots')
```



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 [21]:
```
weights = [529117365, 390965105, 374061888, 326627740, 320410057, 313720540, 29430021
           277000841, 216768975, 183996130, 169330528, 138416451, 117295780, 110504544
           95422055, 91258980, 90376747, 79843664, 75294515, 70195826, 46337161, \
           35373464, 9613410, 8369915, 4975847, 4550166]

die1 = Die(np.array(['E', 'T', 'A', 'O', 'I', 'N', 'S', 'R', 'H', 'L', 'D', 'C', 'U',
           'G', 'P', 'W', 'Y', 'B', 'V', 'K', 'J', 'X', 'Z', 'Q']))

die1.die_current_state()['weights'] = weights
die1.die_current_state()
```

Out[21]:

	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 [ ]:

```
list_of_letters = [die1, die1, die1, die1]
game_obj = Game(list_of_letters)
game_obj.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 [ ]:

```
task3 = Analyzer(game_obj)
perms = task3.permutation_count()
```

This produces 44 words

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 [ ]:

```
list_of_letters = [die1, die1, die1, die1, die1]
game_obj = Game(list_of_letters)
game_obj.play(1000)

task4 = Analyzer(game_obj)
perms = task4.permutation_count()
```

This produces 20 words

4 letter words are more common than 5 letter words when randomly selecting letters with weights.

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.