Metadata

Title: **Final Project Report**

Class: DS 5100

Date: 12/6/22

Student Name: Noah Edwards-Thro

Student Net ID: xjb6yb

This URL: https://github.com/nedwardsthro/MonteCarlo/blob/main/final-project-submission.ipynb (https://github.com/nedwardsthro/MonteCarlo/blob/main/final-project-submission.ipynb)

Github Repo URL: https://github.com/nedwardsthro/MonteCarlo (https://github.com/nedwardsthro/MonteCarlo)

Monte Carlo Module

In [2]:

```
import numpy as np
import pandas as pd

class Die:
    '''
    Purpose: Create a die to be rolled with different weights
    Inputs:
        - faces : List of faces for the die
    '''
    def __init__(self, faces):
        self.die = pd.DataFrame({
            'faces': faces,
            'weights': np.ones(len(faces))
        })

    def change_weight(self, face_value, new_weight):
        '''
        Purpose: Change the weight of a single face on the die
        Inputs:
            - faces : face name to be changed
            - new_weight: new weight of the face
        '''
        self.die.loc[self.die.faces == face_value, 'weights'] = new_weight

    def roll(self, nrolls = 1):
        '''
        Purpose: Roll a die based on the current weights of the die
        Inputs:
            - nrolls : number of rolls
        Outputs:
            - Result of the rolls
        '''
        self.my_probs = [i/sum(self.die.weights) for i in self.die.weights]
        results = []
        for i in range(nrolls):
            result = self.die.sample(weights=self.die.weights).values[0][0]
            results.append(result)
        return(results)

    def show(self):
        '''
        Purpose: Show the current die faces and weights
        Outputs:
            - Dataframe of the die faces and weights
        '''
        return(self.die)

class Game:
    '''
    Purpose: To roll a set of dice a certain number of times and store the result
    Inputs:
        - die_set: List of die objects
    '''
    def __init__(self, die_set):
        self.die_set = die_set

    def play(self, nrolls):
```

```
        '''
        Purpose: Roll the die set nrolls times and store the combinations
        Inputs:
            - nrolls: Number of times to roll each dice in the set
        '''
        self.result_df = pd.DataFrame()
        for i in range(0, len(self.die_set)):
            die = self.die_set[i]
            self.result_df[i] = die.roll(nrolls)
        self.result_df = self.result_df.rename_axis(columns = "Faces")
        self.result_df.index.rename('Roll_Number', inplace = True)

    def show(self, df_format = 'wide'):
        '''
        Purpose: Show the rolls of the die set in either a narrow or wide format
        Inputs:
            - df_format: "narrow" or "wide", how you want to df to be returned
        Outputs:
            - Dataframe with the results of the game
        '''
        if df_format == "wide":
            return(self.result_df)
        elif df_format == "narrow":
            df = self.result_df.stack().to_frame().reset_index().rename(columns =
{'Roll_Number':'Roll_Number', 'Faces':'Die_Number', 0:'Face'}).set_index(['Roll_Num
ber', 'Die_Number'])
            return(df)
        else:
            print("df_format must be 'narrow' or 'wide'")

class Analyzer:
    '''
    Purpose: Analyze a game class and find different combinations and face counts
    Inputs:
        - Game: a game class
    '''
    def __init__(self, Game):
        self.Game = Game

    def face_counts_per_roll(self):
        '''
        Purpose: Find the number of times each face appears in each roll
        '''
        self.face_counts = self.Game.show("wide").copy().apply(pd.Series.value_coun
ts, axis=1).fillna(0).rename_axis(columns = "Faces")

    def combo(self):
        '''
        Purpose: Find the different combinations of faces that appear in each roll
and count the number of times they appear
        '''
        df = self.Game.show("wide").copy()
        df['list'] = pd.Series(df.astype(str).values.tolist()).apply(lambda x: sort
ed(x))
        inter = df.sort_values("list")['list'].value_counts().to_frame().reset_inde
x().sort_values('index').reset_index(drop = True)
        self.combo = pd.DataFrame(inter['index'].tolist()).reset_index(drop = True)
```

```
        self.combo['Count'] = inter['list']
        self.combo = self.combo.set_index(self.combo.columns.difference(['Count'],
sort = False).tolist())

    def jackpot(self):
        '''
        Purpose: Count the number of times a roll returns all of the same faces
        Output:
            - An integer of the number of times all of the faces were the same
        '''
        self.jackpot = self.Game.show("wide").copy()
        self.jackpot['list'] = pd.Series(self.jackpot.astype(str).values.tolist()).
apply(lambda x: sorted(x))
        self.jackpot['uniques'] = self.jackpot['list'].explode().groupby('Roll_Numb
er').unique()
        self.jackpot['Jackpot'] = np.where(self.jackpot['uniques'].str.len() == 1,
True, False)
        self.jackpot = self.jackpot[self.jackpot['Jackpot'] == True]
        total_jackpots = sum(self.jackpot['Jackpot'])
        return(total_jackpots)
```

Test Module

In [3]:

```
import unittest
class TestCase(unittest.TestCase):

    def test_die_length(self):
        Test_Die = Die([1,2,3,4])
        Expected_Length = 4
        # Number of rows is the same as it should be
        self.assertEqual(len(Test_Die.die), Expected_Length)

    def test_change_weight(self):
        Test_Die = Die([1,2,3,4])
        Test_Die.change_weight(2,2)
        # Test if the weight was changed
        self.assertEqual(Test_Die.die[Test_Die.die['faces'] == 2]['weights'], 2)

    def test_roll_length(self):
        Test_Die = Die([1,2,3,4])
        roll = Test_Die.roll(3)
        # Test to see if the three rolls were produced
        self.assertEqual(len(roll), 3)

    def test_show_dataframe(self):
        Test_Die = Die([1,2,3,4])
        # Test to see if a dataframe is returned
        self.assertEqual(type(Test_Die.show()), type(pd.DataFrame()))

    def test_die_set_length(self):
        Test_Die = Die([1,2,3,4])
        Test_Game = Game([Test_Die, Test_Die, Test_Die])
        # Test to see if there are 3 die in the set
        self.assertEqual(len(Test_Game.die_set), 3)

    def test_game_play_length(self):
        Test_Die = Die([1,2,3,4])
        Test_Game = Game([Test_Die, Test_Die, Test_Die])
        Test_Game.play(10)
        # Test to see if there are 10 rolls
        self.assertEqual(len(Test_Game.result_df), 10)

    def test_game_show_dataframe(self):
        Test_Die = Die([1,2,3,4])
        Test_Game = Game([Test_Die, Test_Die, Test_Die])
        Test_Game.play(10)
        # Test to see if show returns a dataframe
        self.assertEqual(type(Test_Game.show()), type(pd.DataFrame()))

    def test_face_counts_per_roll_length(self):
        Test_Die = Die([1,2,3,4])
        Test_Game = Game([Test_Die, Test_Die, Test_Die])
        Test_Game.play(10)
        Test_Analyzer = Analyzer(Test_Game)
        Test_Analyzer.face_counts_per_roll()
        # Test to see if there are the right number of faces
        self.assertEqual(len(Test_Analyzer.face_counts.axes[1]), 4)

    def test_jackpot_type(self):
        Test_Die = Die([1,2,3,4])
```

```
        Test_Game = Game([Test_Die, Test_Die, Test_Die])
        Test_Game.play(10)
        Test_Analyzer = Analyzer(Test_Game)
        # Test to see if jackpot returns an integer
        self.assertEqual(type(Test_Analyzer.jackpot()), int)

    def test_combo_method(self):
        Test_Die = Die([1,2,3,4])
        Test_Game = Game([Test_Die, Test_Die, Test_Die])
        Test_Game.play(10)
        Test_Analyzer = Analyzer(Test_Game)
        Test_Analyzer.combo()
        # Test to see if the index is a MultiIndex
        self.assertEqual(type(Test_Analyzer.combo.index), pd.core.indexes.multi.Mul
tiIndex)
```

Test Results

test_change_weight (__main__.TestCase) ... ok test_combo_method (__main__.TestCase) ... ok test_die_length (__main__.TestCase) ... ok test_die_set_length (__main__.TestCase) ... ok test_face_counts_per_roll_length (__main__.TestCase) ... ok test_game_play_length (__main__.TestCase) ... ok test_game_show_dataframe (__main__.TestCase) ... ok test_jackpot_type (__main__.TestCase) ... ok test_roll_length (__main__.TestCase) ... ok test_show_dataframe (__main__.TestCase) ... ok ---------- Ran 10 tests in 0.276s OK

Scenario 1

In [4]:

```
Fair_Coin = Die(['Heads', 'Tails'])
Unfair_Coin = Die(['Heads', 'Tails'])
Unfair_Coin.change_weight("Heads", 5)
Unfair_Coin.show()
Fair_Coin.show()
```

Out[4]:

	faces	weights
0	Heads	1.0
1	Tails	1.0

In [5]:

```
Fair_Game = Game([Fair_Coin, Fair_Coin, Fair_Coin])
Fair_Game.play(1000)
Fair_Game.show("wide")
```

Out[5]:

Faces	0	1	2
Roll_Number			
0	Tails	Heads	Heads
1	Heads	Heads	Heads
2	Tails	Heads	Tails
3	Tails	Tails	Heads
4	Tails	Heads	Tails
...	...	...	...
995	Tails	Heads	Heads
996	Heads	Heads	Tails
997	Tails	Heads	Tails
998	Tails	Heads	Heads
999	Heads	Tails	Tails

1000 rows × 3 columns

In [6]:

```
Unfair_Game = Game([Unfair_Coin, Unfair_Coin, Unfair_Coin])
Unfair_Game.play(1000)
Unfair_Game.show("wide")
```

Out[6]:

Faces	0	1	2
Roll_Number			
0	Heads	Heads	Heads
1	Heads	Tails	Heads
2	Heads	Heads	Heads
3	Heads	Heads	Heads
4	Heads	Heads	Heads
...	...	...	...
995	Tails	Heads	Tails
996	Heads	Heads	Heads
997	Tails	Tails	Tails
998	Heads	Heads	Heads
999	Heads	Heads	Heads

1000 rows × 3 columns

In [7]:

```
F = Analyzer(Fair_Game)
F.jackpot()
UF = Analyzer(Unfair_Game)
UF.jackpot()
```

Out[7]:

```
568
```

Scenario 2

In [8]:

```
Fair_Dice = Die([1,2,3,4,5,6])
Fair_Dice.show()
Unfair_Dice1 = Die([1,2,3,4,5,6])
Unfair_Dice1.change_weight(6, 5)
Unfair_Dice1.show()
Unfair_Dice2 = Die([1,2,3,4,5,6])
Unfair_Dice2.change_weight(1, 5)
Unfair_Dice2.show()
```

Out[8]:

	faces	weights
0	1	5.0
1	2	1.0
2	3	1.0
3	4	1.0
4	5	1.0
5	6	1.0

In [9]:

```
Fair_Dice_Game = Game([Fair_Dice,Fair_Dice,Fair_Dice,Fair_Dice,Fair_Dice])
Fair_Dice_Game.play(10000)
Fair_Dice_Game.show()
Unfair_Dice_Game = Game([Unfair_Dice1,Unfair_Dice1,Unfair_Dice2,Fair_Dice,Fair_Dic
e])
Unfair_Dice_Game.play(10000)
Unfair_Dice_Game.show("wide")
```

Out[9]:

Faces	0	1	2	3	4
Roll_Number					
0	5.0	1.0	3.0	6.0	3.0
1	6.0	5.0	6.0	2.0	5.0
2	6.0	2.0	1.0	5.0	6.0
3	6.0	5.0	1.0	2.0	2.0
4	6.0	3.0	2.0	2.0	5.0
...	...	...	...	...	...
9995	6.0	6.0	1.0	6.0	6.0
9996	6.0	6.0	5.0	2.0	1.0
9997	6.0	6.0	1.0	2.0	3.0
9998	6.0	6.0	4.0	3.0	6.0
9999	6.0	6.0	6.0	2.0	1.0

10000 rows × 5 columns

In [10]:

```
FDA = Analyzer(Fair_Dice_Game)
UFDA = Analyzer(Unfair_Dice_Game)
FDA.jackpot()
UFDA.jackpot()
```

Out[10]:

10

In [11]:

```
FDA.combo()
UFDA.combo()
FDA.combo.sort_values('Count', ascending = False)
UFDA.combo.sort_values('Count', ascending = False)
```

Out[11]:

					Count
0	**1**	**2**	**3**	**4**	
1.0	**4.0**	**5.0**	**6.0**	**6.0**	220
	2.0	**5.0**	**6.0**	**6.0**	204
	3.0	**5.0**	**6.0**	**6.0**	202
		4.0	**6.0**	**6.0**	198
	2.0	**4.0**	**6.0**	**6.0**	189
...	**...**	**...**	**...**	**...**	...
2.0	**2.0**	**5.0**	**5.0**	**5.0**	1
	3.0	**3.0**	**3.0**	**3.0**	1
	4.0	**4.0**	**4.0**	**4.0**	1
3.0	**3.0**	**3.0**	**5.0**	**5.0**	1
1.0	**5.0**	**5.0**	**5.0**	**5.0**	1

241 rows × 1 columns

Scenario 3

In [12]:

```
letters = Die(["A", "B", "C", "D", "E", "F", "G", "H", "I", 'J', 'K',
               'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V',
               'W', 'X', 'Y', 'Z'])
letters.change_weight("A",8.4966)
letters.change_weight("B",2.0720)
letters.change_weight("C",4.5388)
letters.change_weight("D",3.3844)
letters.change_weight("E",11.1607)
letters.change_weight("F",1.8121)
letters.change_weight("G",2.4705)
letters.change_weight("H",3.0034)
letters.change_weight("I",7.5448)
letters.change_weight("J",0.1965)
letters.change_weight("K",1.1016)
letters.change_weight("L",5.4893)
letters.change_weight("M",3.0129)
letters.change_weight("N",6.6544)
letters.change_weight("O",7.1635)
letters.change_weight("P",3.1671)
letters.change_weight("Q",0.1962)
letters.change_weight("R",7.5809)
letters.change_weight("S",5.7351)
letters.change_weight("T",6.9509)
letters.change_weight("U",3.6308)
letters.change_weight("V",1.0074)
letters.change_weight("W",1.2899)
letters.change_weight("X",0.2902)
letters.change_weight("Y",1.7779)
letters.change_weight("Z",0.2722)

letters.show()
```

Out[12]:

	faces	weights
0	A	8.4966
1	B	2.0720
2	C	4.5388
3	D	3.3844
4	E	11.1607
5	F	1.8121
6	G	2.4705
7	H	3.0034
8	I	7.5448
9	J	0.1965
10	K	1.1016
11	L	5.4893
12	M	3.0129
13	N	6.6544
14	O	7.1635
15	P	3.1671
16	Q	0.1962
17	R	7.5809
18	S	5.7351
19	T	6.9509
20	U	3.6308
21	V	1.0074
22	W	1.2899
23	X	0.2902
24	Y	1.7779
25	Z	0.2722

In [13]:

```
words = Game([letters, letters, letters, letters, letters])
words.play(1000)
words_df = words.show()
```

In [14]:

```
np.random.seed(1)
sample1 = words_df.sample(10)
sample2 = words_df.sample(10)
sample3 = words_df.sample(10)
sample4 = words_df.sample(10)
sample5 = words_df.sample(10)
sample6 = words_df.sample(10)
sample7 = words_df.sample(10)
sample8 = words_df.sample(10)
sample9 = words_df.sample(10)
sample10 = words_df.sample(10)
```

0 of 100 five lettere combinations are words.

Directory Listing

In [16]:

```
!ls -lRF -o
```

```
total 88
-rw-r--r--  1 noahthro   1074 Dec  5 23:59 LICENSE
drwxr-xr-x@ 8 noahthro    256 Dec  6 00:01 MonteCarlo/
-rw-r--r--  1 noahthro    205 Dec  5 23:59 MonteCarlo.Rproj
-rw-r--r--  1 noahthro     12 Dec  5 23:59 README.md
drwxr-xr-x@ 4 noahthro    128 Dec  6 00:01 __pycache__/
-rw-r--r--  1 noahthro  28036 Dec  6 00:10 final-project-submission.ipy
nb
-rw-r--r--  1 noahthro    576 Dec  5 23:43 montecarlo_test_results.txt

./MonteCarlo:
total 88
-rw-r--r--  1 noahthro      0 Dec  5 23:33 __init__.py
-rw-r--r--@ 1 noahthro   4727 Dec  5 22:52 montecarlo.py
-rw-r--r--  1 noahthro  26612 Dec  5 23:28 montecarlo_demo.ipynb
-rw-r--r--  1 noahthro   2880 Dec  5 23:08 montecarlo_tests.py
-rw-r--r--  1 noahthro    236 Dec  5 23:37 setup.py

./__pycache__:
total 24
-rw-r--r--  1 noahthro  5760 Dec  5 22:52 montecarlo.cpython-38.pyc
-rw-r--r--  1 noahthro  2988 Dec  5 22:55 montecarlo_tests.cpython-38.p
yc
```

Installation Output Listing

In [17]:

```
!cd MonteCarlo/; pip install .
```

```
Processing /Users/noahthro/Desktop/UVA Work/UVA-MSDS/DS 5100/MonteCarl
o/MonteCarlo
Building wheels for collected packages: MonteCarlo
  Building wheel for MonteCarlo (setup.py) ... done
  Created wheel for MonteCarlo: filename=MonteCarlo-0.1-py3-none-any.wh
l size=1050 sha256=e9f0b8c8b0f0f6476f09953c9030f4af019d3ffc570501954ae6
fb898625e783
  Stored in directory: /private/var/folders/vh/g7bcwq5124s8ggtldg60zx6c
0000gn/T/pip-ephem-wheel-cache-tilgx_rf/wheels/fd/0d/cd/31a8b98c495dfd1
09cf484793e27dc1e5118d7e87c2da2ad0d
Successfully built MonteCarlo
Installing collected packages: MonteCarlo
  Attempting uninstall: MonteCarlo
    Found existing installation: MonteCarlo 0.1
    Uninstalling MonteCarlo-0.1:
      Successfully uninstalled MonteCarlo-0.1
Successfully installed MonteCarlo-0.1
```