Coin, Dice, and Card

1 Overview

For this challenge, we want to simulate a game with the following win condition:

- Roll of 6 on a 6 sided die,
- Flip of heads on a coin,
- Either the Ace of Diamonds or the Jack of Spades from a 52 card deck.

We continue playing until this win condition is met. One play through of the game consists of:

1. Rolling the dice,
2. Flipping the coin,
3. Shuffling the deck, picking a card, and returning the card to the deck

We also want to log the outcome of all plays until the win condition is met. Keyboard Interrupt error handling is also supported. Efficiency, readability and modularity were also designed into the program.

2 File Structure and Program Overview

To maintain modularity, the components are broken up into individual files and classes, shown in Figure 1

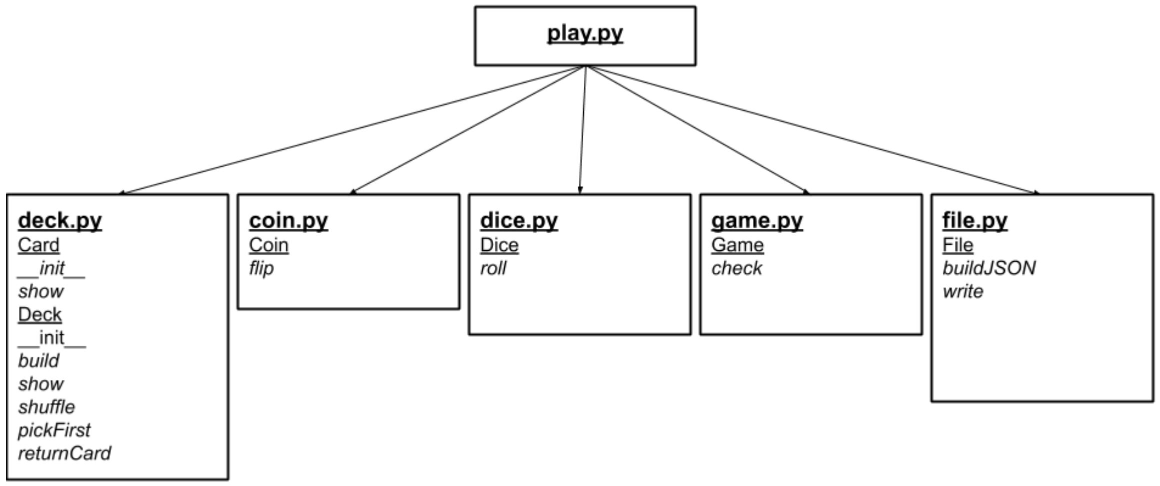

Figure 1: File Structure with Classes and Methods

This file structure allows for individual components to be isolated and independent of each other. The play file controls the instantiation, calling, and implementation of all other classes. Any changes to individual classes, such as increasing the number of sides on the dice, can be easily made without having to reflect the change in multiple other files. It also provides opportunities for unit testing the individual components and feature testing the entire play program. Further explanation of the methods and classes can be found in section 3.

To play a game, run the play.py script. The script plays with the flow outline in figure 2

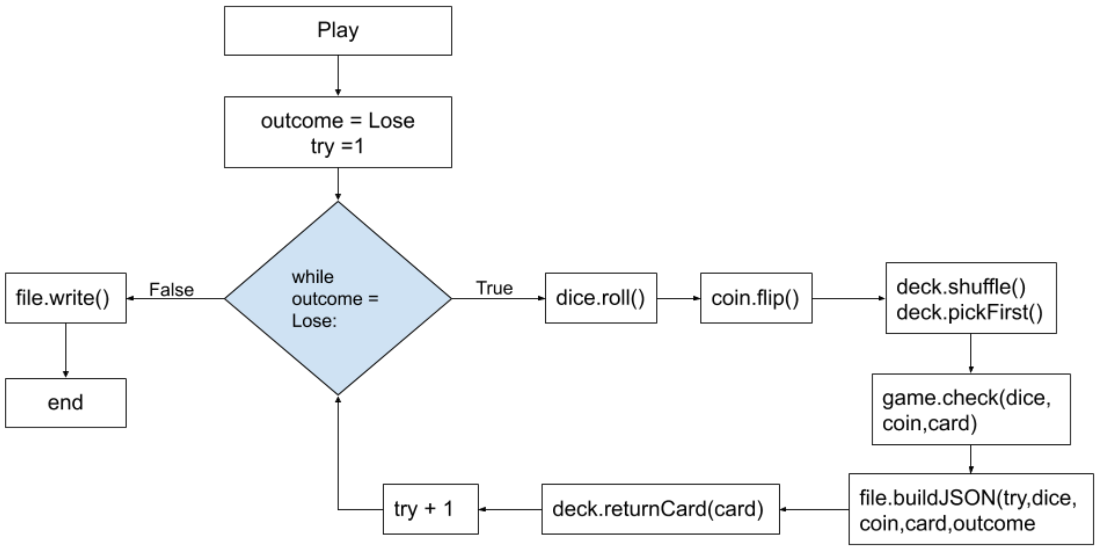

Figure 2: Flow Diagram for Play Script

Note that the deck is reshuffled for each subsequent play before choosing a card.

3 Classes and Methods

Raw code for all classes and relevant files can be found in the A section of the appendix.

3.1 Play

The play script simply instantiates the component classes and calls specific methods while the outcome remains false. The flow diagram in 2 gives a broad overview of the script's purpose. The index `i` is used to track which how many times the game has been played.

The script also supports `KeyboardInterrupt` error handling and records which try the interrupt happened.

3.2 Coin

The coin class contains only one `flip()` method, which returns a string value of heads or tails. Using the random number generator, the method selects a random element of the list `outcomes[]`. This allows the possible outcomes to be adjusted in the future.

3.3 Dice

The dice class also contains only one `roll()` method which returns an integer. The the number of possibilities is between 1 and `sides`, which dictates how many sides the dice has.

3.4 Deck

The deck file contains the `Card` and `Deck` class. The `Card` class object has `suit` and `value` attributes. The possible suits and values are defined by the `suits[]` and `values[]` lists and are clubs, diamonds, hearts, and spades and Ace through King, respectively.

3.4.1 Card Class

The `Card` class takes the suit and value of the card object and assigns them, respectively. `Card`'s only method is `show()`, which formats the suit and value of the card and returns it as a string.

3.4.2 Deck Class

The `Deck` constructor initializes an empty `cards[]` list to take individual `Card` objects. The constructor then calls the `build()` method, which iterates through suits and values to generate 52 unique `Card` objects, which are then stored in the `cards[]` list, building a deck. The other available methods to a `Deck` object are:

- The `shuffle()` method randomizes the order of the `Card` objects in the deck and does not return anything.

- The `pickFirst()` method pops the first `Card` object from the deck and returns the object.
- The `returnCard(Card_Object)` method appends a `Card` object to the deck.

3.5 Game

The game file determines the winning conditions of the game. The current winning condition must meet the following criteria:

- Roll of 6,
- Coin flip of heads,
- Either the Ace of Diamonds or the Jack of Spades.

The `Game` class contains a `check(coinFlip, diceRoll, cardPick)` method, which takes the coin, dice, and card outcomes as arguments and compares them against the winning condition. It returns a string result of `Win` or `Lose`, respectively.

3.6 File

The `file` file handles writing to the output file. In it, the results of each are stored in a dictionary called `resultsJSON{[{}]}`. All results until the winning try are stored and appended to this dictionary in the `buildJSON(tryNumber, diceRoll, coinFlip, cardPick, outcome)` method of the `File` class. Upon a win condition, the resulting object is then written to `results.txt` using the `write()` method. This method is only called once and is separate to avoid opening and writing to the file for each individual try.

4 Testing, Design Decisions, and Improvements

In this section I want to discuss unit tests and some of my design design decisions, as well as improvements in the code that may mitigate scaling issues.

4.1 Testing Probabilities

For testing, I focused on the probabilities of each component. I designed and implemented a Monte Carlo Simulation unit test for the coin flip, dice roll, and card draw. The unit test can be found in the Appendix. The probabilities for any given outcome is $\frac{1}{2}$ for the coin, $\frac{1}{6}$ for the dice, and $\frac{1}{52}$ for any given card. To test these, I ran the flip and roll for 10,000 times 10 times for the coin and dice, respectively. I then recorded how many times the result was heads or tails and 1-6 and compared. After this test, I came out with the following results:

```
Run 1{'tails': 4933, 'heads': 5067}
 {'1': 1633, '3': 1615, '2': 1690, '5': 1683, '4': 1710, '6': 1669}

Run 2{'tails': 5023, 'heads': 4977}
 {'1': 1688, '3': 1677, '2': 1677, '5': 1695, '4': 1609, '6': 1654}

Run 3{'tails': 5009, 'heads': 4991}
{'1': 1709, '3': 1625, '2': 1687, '5': 1615, '4': 1690, '6': 1674}

Run 4{'tails': 4981, 'heads': 5019}
{'1': 1627, '3': 1664, '2': 1725, '5': 1629, '4': 1674, '6': 1681}

Run 5{'tails': 5003, 'heads': 4997}
 {'1': 1628, '3': 1700, '2': 1673, '5': 1679, '4': 1726, '6': 1594}

Run 6{'tails': 4863, 'heads': 5137}
 {'1': 1648, '3': 1639, '2': 1675, '5': 1677, '4': 1674, '6': 1687}

Run 7{'tails': 4871, 'heads': 5129}
{'1': 1627, '3': 1656, '2': 1676, '5': 1672, '4': 1702, '6': 1667}

Run 8{'tails': 5014, 'heads': 4986}
{'1': 1607, '3': 1674, '2': 1713, '5': 1680, '4': 1654, '6': 1672}

Run 9{'tails': 4940, 'heads': 5060}
 {'1': 1639, '3': 1622, '2': 1659, '5': 1688, '4': 1695, '6': 1697}

Run 10{'tails': 4975, 'heads': 5025}
{'1': 1687, '3': 1648, '2': 1674, '5': 1702, '4': 1676, '6': 1613}
```

Though not perfect, the distribution is very close to random. I test the deck by shuffling, picking the first card, and returning the card to the bottom of the pile and repeating. However the test results are difficult to manually verify and would require subsequent analysis to verify the distribution.

4.2 Design Decision

4.2.1 Using JSON vs Raw Text

The attempt logging uses a JSON, key-value format, rather than a CSV raw text. I made this decision with data manipulation in mind. While they are larger files, JSON allows us to iterate over, manipulate, and work with data much easier than a straight CSV or text file. Using the `json.loads()` method, we would be able to easily read the logs and import them as a JSON object.

4.3 Improvements

4.3.1 Testing

The test method is merely testing the individual components. A more robust test suite would include testing the win outcomes in a similar manner. The overall probability of a win is:

$$\frac{1}{2} \times \frac{1}{6} \times \frac{2}{52} = \frac{1}{312} \tag{1}$$

By using a Monte Carlo Simulation, we can effectively run the `play` script many, many times and record the winning try. By averaging a large number of these tries, we expect to approach an average of 312 tries per run.
The unit tests may also be more effective to show distribution if they displayed the distribution by percent or approximate fraction.

4.3.2 Result Logging

Currently the `File` class stores all results in memory until the win condition is met. While this works for small logs, such as this, it is not very scalable. In the case where a win scenario is extremely unlikely, the JSON object may outgrow the computer's memory. If this were of concern, we could implement writing the logs more frequently to clear memory. While this is slower, we would be able to log much, much bigger logs with more detail.

4.3.3 Error Handling

Though unlikely, built in error handling which breaks the program in the event that a response is not received from one of the components may be an area of further improvement. In the event that a response is not received, the program currently runs the risk of never exiting the `while` loop in play and continuing to test indefinitely, while never being able to achieve the win condition.

A Appendix for Raw Code

A.1 play.py

```
from dice import Dice
from coin import Coin
from deck import Deck, Card
from game import Game
from file import File
```

```
import json

#Initializing class instances required and try counter,
#useful for debugging and statistic calculations
i=1
coin = Coin()
dice = Dice()
deck = Deck()
game = Game()
file = File()

outcome = "Lose"
#Keep playing until win
while outcome == "Lose":
        try:
                #Retrieving coin flip and dice roll value
                coinFlip = coin.flip()
                diceRoll = dice.roll()

                #Shuffling the deck and picking the first card
                deck.shuffle()
                cardPick = deck.pickFirst()

                #Passing the values to the game class for checking outcome
                outcome = game.check(coinFlip,diceRoll, cardPick.show())

                #Building the JSON object to write to output file
                file.buildJSON(i,coinFlip,diceRoll, cardPick.show(),outcome)

                #Returning the top card to the bottom of the pile
                deck.returnCard(cardPick)
                i+=1
        except KeyboardInterrupt:
                file.keyboardInterrupt(i)
                file.write()
                raise

#Writing the full JSON object to the file
file.write()
```

A.2 coin.py

```
import random

outcomes = ["heads", "tails"]

class Coin():
        #Random heads/tails output
        def flip(self):
                outcome = outcomes[random.randint(0,1)]
                return outcome
```

A.3 dice.py

```
import random

outcomes = ["heads", "tails"]

class Coin():
        #Random heads/tails output
        def flip(self):
                outcome = outcomes[random.randint(0,1)]
                return outcome
```

A.4 deck.py

```
import random

#Determining suits and values of cards in the deck
suits = ['clubs', 'diamonds', 'hearts', 'spades']
values = ['A', '2', '3', '4', '5', '6', '7', '8', '9', '10' ,'J', 'Q', 'K']

class Card:
        def __init__(self,suit,value):
                self.suit = suit
                self.value = value

        #Stringifies the card object to use for comparison and recording
        def show(self):
                return ("{} of {}".format(self.value, self.suit))

class Deck(Card):
        def __init__(self):
                self.cards = []
                self.build()

        #Building a deck of 52 Cards
        def build(self):
                for suit in suits:
                        for value in values:
                                self.cards.append(Card(suit,value))

        def shuffle(self):
                cards = random.shuffle(self.cards)

        def pickFirst(self):
                return self.cards.pop(0)

        def returnCard(self, pick):
                self.cards.append(pick)
```

A.5 game.py

```
winning_flip = "heads"
winning_roll = 6
winning_cards = ["J of spades", "A of diamonds"]

class Game():
        def check(self, coinFlip, diceRoll, cardPick):
                #Win condition is a 6 die roll, heads coin flip,
                #and card is either J of spades or A of Diamonds
                if      coinFlip == winning_flip and \
                        diceRoll == winning_roll and \
                        (cardPick in winning_cards):
                        return "Win"
                else:
                        return "Lose"
```

A.6 file.py

```
import json

#File Object Handler Init
resultsJSON = {}
resultsJSON['run'] = []
```

```
class File:
        #Buidling JSON object to dump into the text file
        def buildJSON(self, tryNumber, diceRoll, coinFlip, cardPick, outcome):
                outcomeDict = {
                        'try': tryNumber,
                        'dice': diceRoll,
                        'coin': coinFlip,
                        'card': cardPick,
                        'result': outcome
                }

                resultsJSON['run'].append(outcomeDict)

        def keyboardInterrupt(self,tryNumber):

                interruptMsg = {
                        'try': tryNumber,
                        'message': "Keyboard Interrupt on Try: " + \
                        str(tryNumber)
                }

                resultsJSON['run'].append(interruptMsg)

        #Writing the full JSON Object to results.txt
        def write(self):
                with open('results.txt',"w+") as result_file:
                        json.dump(resultsJSON, result_file)
```

A.7 test.py

```
from dice import Dice
from coin import Coin
from deck import Deck, Card
from game import Game
import json

coin = Coin()
dice = Dice()
deck = Deck()
game = Game()
```

```
iterations = 10000

open("test_results.txt","a+")
open("card_results.txt","a+")

coinFlips = {
        'heads': 0,
        'tails': 0
}

diceRolls = {
        '1': 0,
        '2': 0,
        '3': 0,
        '4': 0,
        '5': 0,
        '6': 0
}
cardPicks = {}

for card in deck.cards:
        cardPicks[card.show()] = 0

#Running through the outcomes 10 x 10k
for x in range(0,10):
        for i in range(0,iterations):
                coinFlip = coin.flip()
                coinFlips[coinFlip] += 1

                diceRoll = dice.roll()
                diceRolls[str(diceRoll)] += 1

                deck.shuffle()
                cardPick = deck.pickFirst()
                cardPicks[cardPick.show()] += 1
                deck.returnCard(cardPick)

        with open('test_results.txt',"a") as result_file:
                result_file.write('Run ' + str(x+1) + str(coinFlips) + \
                ' '+ str(diceRolls) +'\n')
        with open('card_results.txt',"a") as card_results_file:
                card_results_file.write('Run '+ str(x+1) + str(cardPicks))
```

```
#Resetting the count of outcomes
coinFlips = {
'heads': 0,
'tails': 0
}

diceRolls = {
'1': 0,
'2': 0,
'3': 0,
'4': 0,
'5': 0,
'6': 0
}
for card in deck.cards:
        cardPicks[card.show()] = 0
```