ProjectIDK

PlayingWithCards – Version 1

February 11, 2017 *NickLeave a comment*
After a good three months, the ChiPy Mentorship program's current session has come to an end and while there is a lot more that can be done on this project, I feel like this is a good time for a detailed writeup on my project's current state. It's nowhere near what I wanted it to be, but I do have a frame in which to build additional details. Having gone with the repository name PlayingWithCards for so long, I think I'll stick with that for my project's name.

https://github.com/nickhattwick/PlayingWithCards (https://github.com/nickhattwick/PlayingWithCards)

Overview

Currently, the program plays a version of Magic with forty card decks consisting of type-less lands and effect-less creature. With a simplified game structure, players summon creatures to attack and block, trying to take their opponents life points from twenty to zero while defending their own. Players can play against an autopilot with set moves or a learning computer player.

The next layer is the problem is the logging system which logs every move made, and parses the information keeping a growing record of moves (currently creatures summoned) and how often they led to a win or a loss. Using this log, the learning computer player calculates which cards to play during the game, while also updating the log and its strategy.

Rules of the Game

1. Each player starts with a 40 card deck made up of lands and creatures, 20 life points, and an empty field, hand and discard pile.
2. Both players shuffle and draw 7 cards from their respective decks.
3. A player is randomly selected to go first and players take turns making moves
4. At the beginning of each players turn, they draw a card and untap all cards they control. During each player's turn they can play one land card and tap each untapped land they have to produce 1 Mana each.
5. During their turn, players can summon creatures by spending Mana equal to the creature's cost.

6. Players can attack their opponent with the creatures they summon, and their opponent can block with one of their creatures. In that case, the creature with lower power will be destroyed (in case of a tie, both are destroyed).
7. Attacking or blocking require creatures to become tapped, making them unable to attack or block again until they are untapped.
8. If a creature's attack is not blocked, the player it's attacking loses life equal to its power.
9. If a players life points hit zero, they lose.

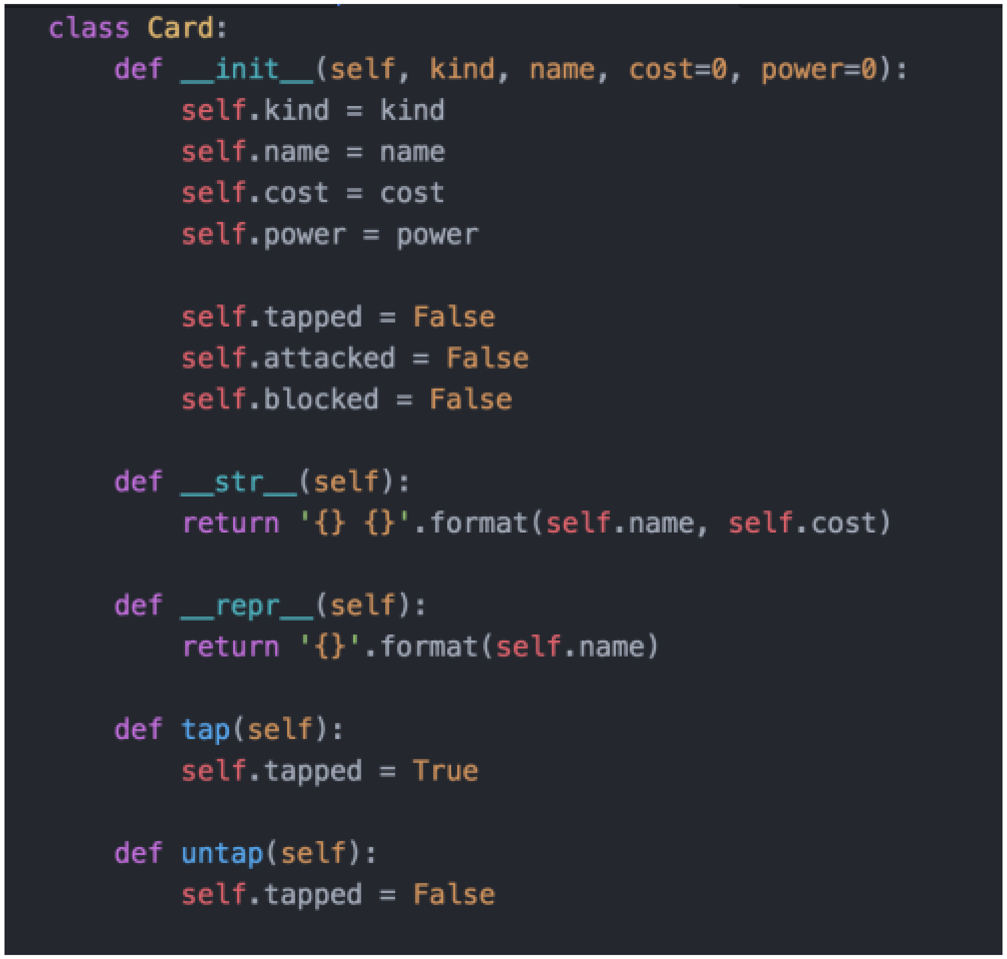
Game Structure

Before getting into the specifics of how the game works, here's a quick rundown of how the game is organized. Each player has a deck of cards, which is a list of Card objects. These decks are imported into a Board class which contains the game logic for one player and the functions needed to move cards around the players zones. Each Player (another class object) is assigned a Board and the Player class contains functions to control the board as well and to handle the attacking/blocking logic which require input from both players. The Player class is divided into subclasses that allow a human or computer to have different ways of inputting their moves. Now, onto a more detailed version of the game, starting at the end of the import chain with the cards, themselves.

The Cards (https://github.com/nickhattwick/PlayingWithCards/blob/master/card.py)

As a card game, some of the most important factors are the cards themselves and how they're programmed. At present, there are six different cards in the games, which are roughly the same as in the initial prototype of the game. These are five creatures with powers ranging from two to six and the land cards used to to play them. Initially these were represented as the string "l" and integers 2 through 6, with one number representing everything about the creature cards. While it created a playable game temporarily, if I wanted to represent specific cards, this wouldn't work.

In order to better represent cards, I started with creating a "Card" class.

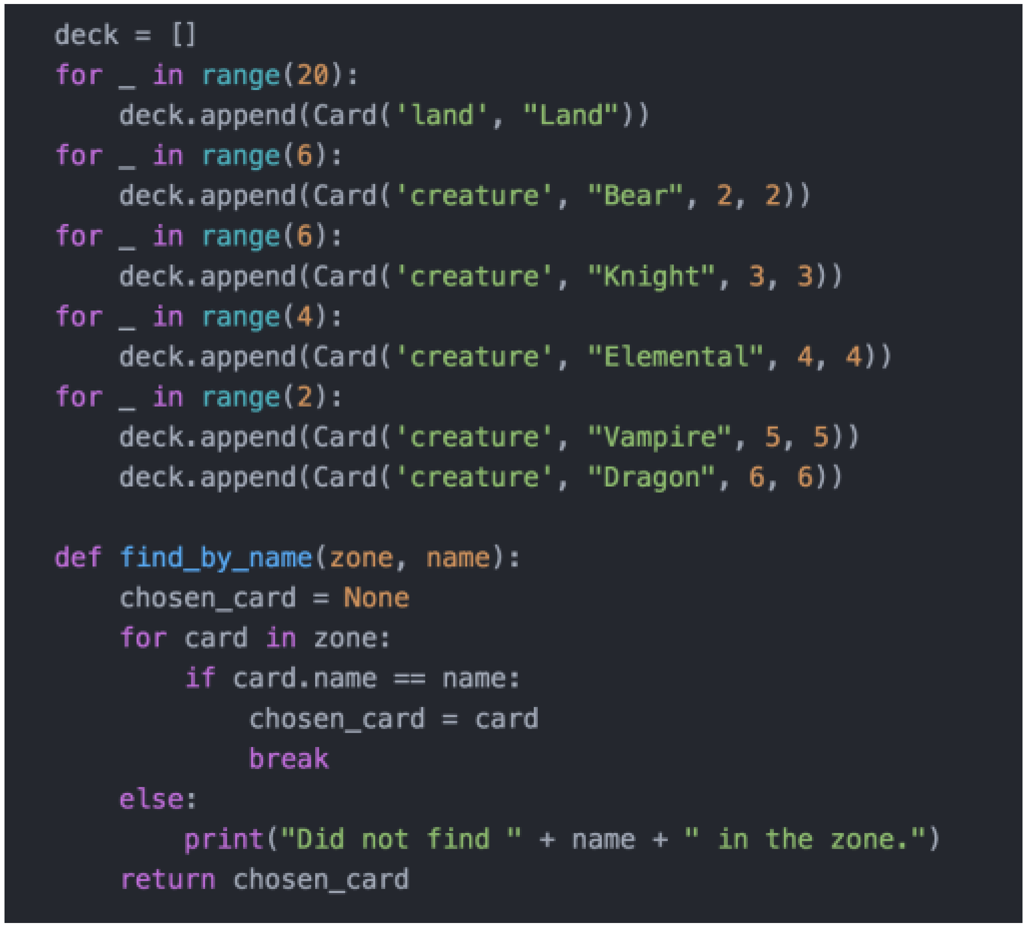


```python
class Card:
    def __init__(self, kind, name, cost=0, power=0):
        self.kind = kind
        self.name = name
        self.cost = cost
        self.power = power

        self.tapped = False
        self.attacked = False
        self.blocked = False

    def __str__(self):
        return '{} {}'.format(self.name, self.cost)

    def __repr__(self):
        return '{}'.format(self.name)

    def tap(self):
        self.tapped = True

    def untap(self):
        self.tapped = False
```

In its current version, all cards are represented with this one class. Every card will have a “kind”, which are either “land” or “creature” at the moment, and will get more detailed with subtypes as the game gets closer to Magic. Every card will also have an individual card name. At the moment, only creatures have cost and power, so presently, those values default to zero. This works presently, though when I add more kinds of cards, I will likely add subclasses to different types of cards to better match the values they do have.

The remaining three values in the _init_ function are different states a card can be in. Both lands and creatures have a tapped state, and the tap and untap methods allow for easy switching of the modes. Only creatures have attacked/blocked states, which are more references for the logging function rather than the game itself. Although as complicated as Magic can get, I’m fairly certain there are cards that will require me to keep those states. Finally, the _str_ and _repr_ function allow the game to print and return the objects as relevant information a person can understand rather than the object’s place in memory.

A deck is a list of these card objects, and currently all players use the same forty card deck.



```python
deck = []
for _ in range(20):
    deck.append(Card('land', "Land"))
for _ in range(6):
    deck.append(Card('creature', "Bear", 2, 2))
for _ in range(6):
    deck.append(Card('creature', "Knight", 3, 3))
for _ in range(4):
    deck.append(Card('creature', "Elemental", 4, 4))
for _ in range(2):
    deck.append(Card('creature', "Vampire", 5, 5))
    deck.append(Card('creature', "Dragon", 6, 6))

def find_by_name(zone, name):
    chosen_card = None
    for card in zone:
        if card.name == name:
            chosen_card = card
            break
    else:
        print("Did not find " + name + " in the zone.")
    return chosen_card
```

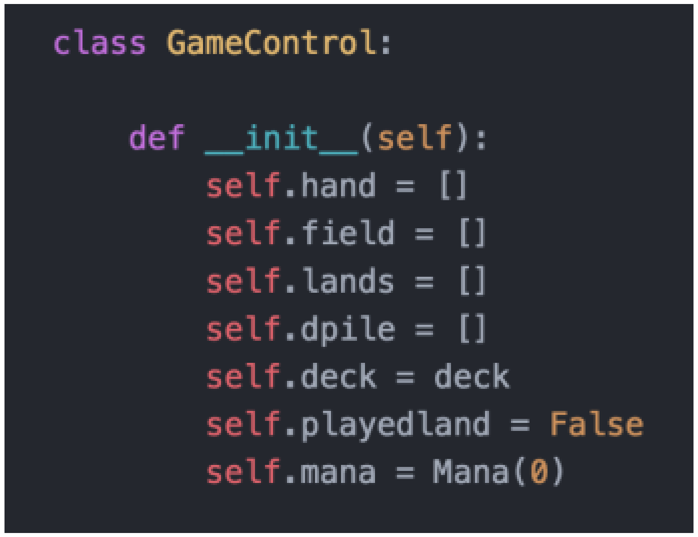
This next part creates a deck of six different cards that are improved versions of the origin one character cards. Each card is given a name such as Bear or Vampire and a cost to play and power corresponding to its number. While the cost and power were taken care of by one number in the previous version, as the game gets more complex, there will be cards with special effects and ones where the cost and power are different. The addition of names also helps make logging easier as well, as well the ability to move cards between game zones of check if a card is in a specific place. The find_by_name function allows the game to look up a card by its name, marking it to be used by another function and checking that it exists and that everything is running as it should. This function is very import and used all over the program.

Mana (https://github.com/nickhattwick/PlayingWithCards/blob/master/mana.py)

In order to play creatures you tap lands to get Mana, a sort of currency used to play creatures. In its present state, Mana is a class with with an amount value that can be increased or decreased and a _repr_ function that represents it as its amount value. The state of mana.py is rather small right now since at the moment its just a number. The file will become more important in the future as the full game has different colors of Mana which are required to meet more specific costs. Additionally, types of Mana can have specific effects or restraints based on the effect of the card used to produce it. While it fills a very small role right now, it does set a class that can be expanded on to increase Mana capabilities later without messing much with the main functionality of the program.

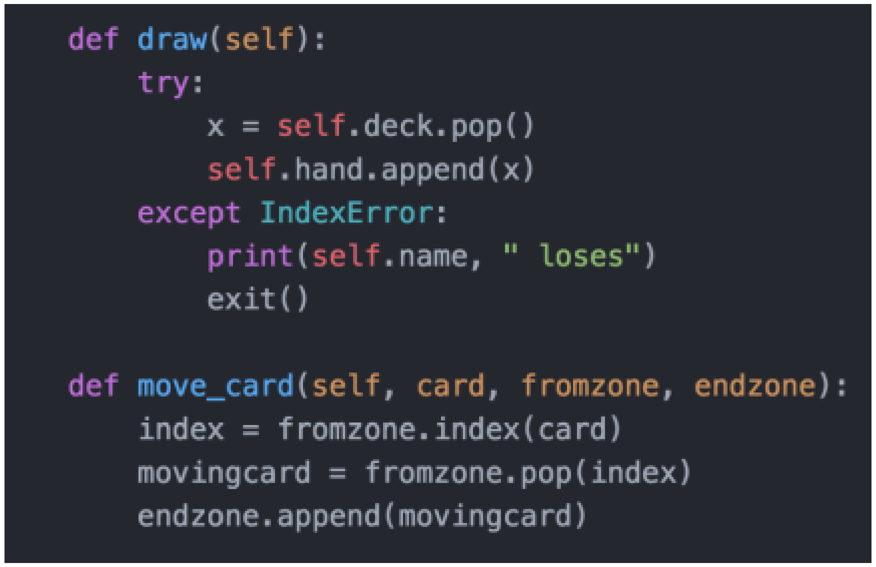
The Board (https://github.com/nickhattwick/PlayingWithCards/blob/master/board.py)

Board.py is a program created to take care of all of the game logic in my program. It contains the code that imports the cards and Mana and controls how cards move around different zones and what state they are in. This is also the file that creates those zones.



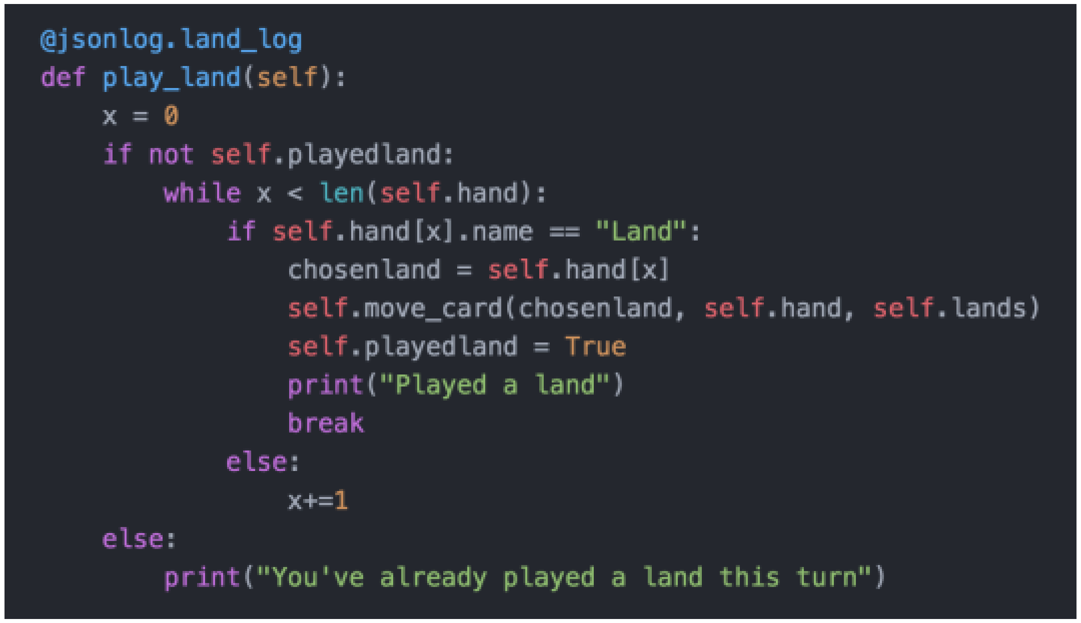
```python
class GameControl:

    def __init__(self):
        self.hand = []
        self.field = []
        self.lands = []
        self.dpile = []
        self.deck = deck
        self.playedland = False
        self.mana = Mana(0)
```

The GameControl class represents all the different zones that cards can move between. Zones are represented by lists where the card objects are stored and consist of the hand, the field, the lands, the discard pile and the deck. The two main functions that move cards around are draw and move_card.



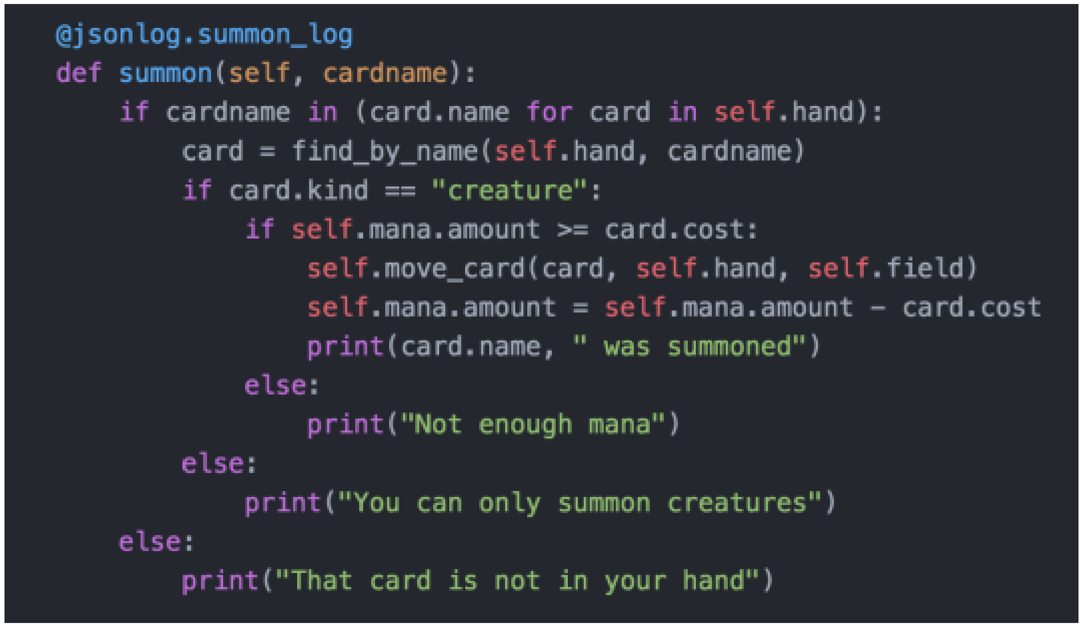
```
def draw(self):
    try:
        x = self.deck.pop()
        self.hand.append(x)
    except IndexError:
        print(self.name, " loses")
        exit()

def move_card(self, card, fromzone, endzone):
    index = fromzone.index(card)
    movingcard = fromzone.pop(index)
    endzone.append(movingcard)
```

Draw takes care of moving unknown cards from the deck into the hand and making a player who tries to draw from an empty deck lose. Move_card on the other hand takes in three variables, the card its moving, where its starting point is, and where its supposed to end up. Then, as its name suggests, it moves the card. This function allows cards to move between any of the zones and is used in this file to play land by moving land from the hand to land zone



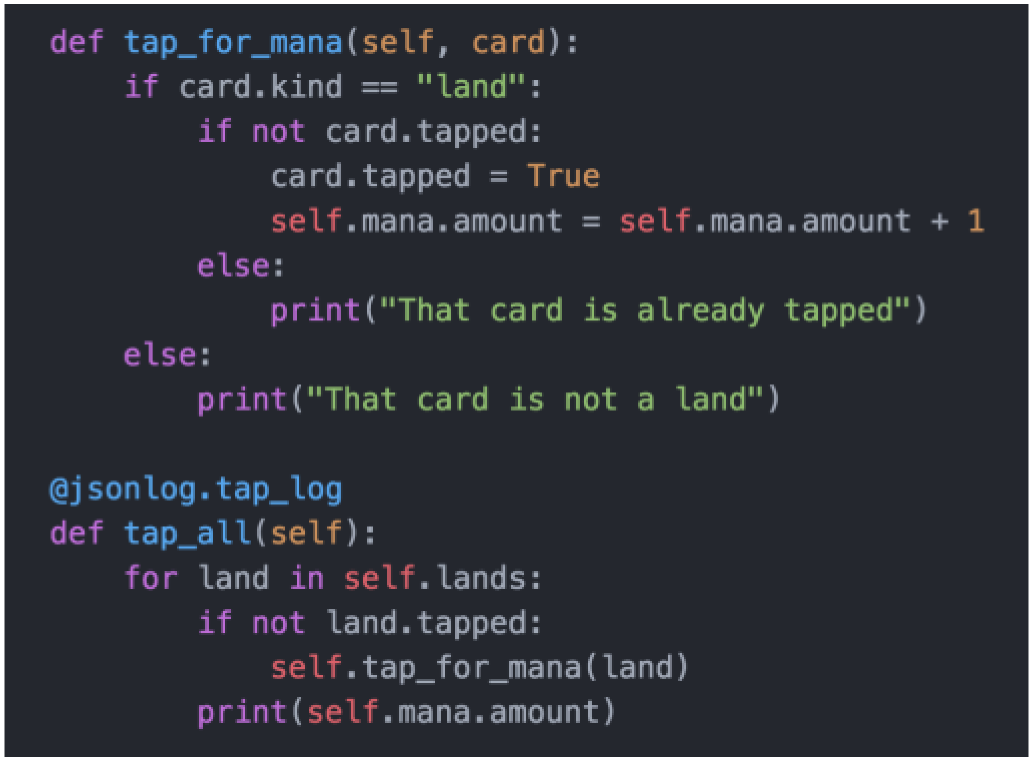
```
@jsonlog.land_log
def play_land(self):
    x = 0
    if not self.playedland:
        while x < len(self.hand):
            if self.hand[x].name == "Land":
                chosenland = self.hand[x]
                self.move_card(chosenland, self.hand, self.lands)
                self.playedland = True
                print("Played a land")
                break
            else:
                x+=1
    else:
        print("You've already played a land this turn")
```

and to summon creatures, by moving them from the hand to the field.



```
@jsonlog.summon_log
def summon(self, cardname):
    if cardname in (card.name for card in self.hand):
        card = find_by_name(self.hand, cardname)
        if card.kind == "creature":
            if self.mana.amount >= card.cost:
                self.move_card(card, self.hand, self.field)
                self.mana.amount = self.mana.amount - card.cost
                print(card.name, " was summoned")
            else:
                print("Not enough mana")
        else:
            print("You can only summon creatures")
    else:
        print("That card is not in your hand")
```

The tap_for_mana function allows for lands to tap individually, although with the game being simplified, most of the the time, tap_all function is what will be called to produce all Mana possible at once. As the game improves, the tap functions will change as the game becomes closer to Magic.

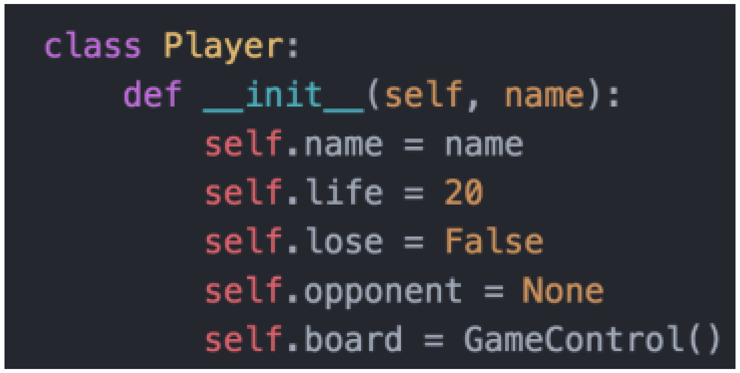


```python
def tap_for_mana(self, card):
    if card.kind == "land":
        if not card.tapped:
            card.tapped = True
            self.mana.amount = self.mana.amount + 1
        else:
            print("That card is already tapped")
    else:
        print("That card is not a land")

@jsonlog.tap_log
def tap_all(self):
    for land in self.lands:
        if not land.tapped:
            self.tap_for_mana(land)
        print(self.mana.amount)
```

I'll get to that @jsonlog in a bit. It's worth noting that there's also an untap_all function in here, which does untaps all of the player's cards.

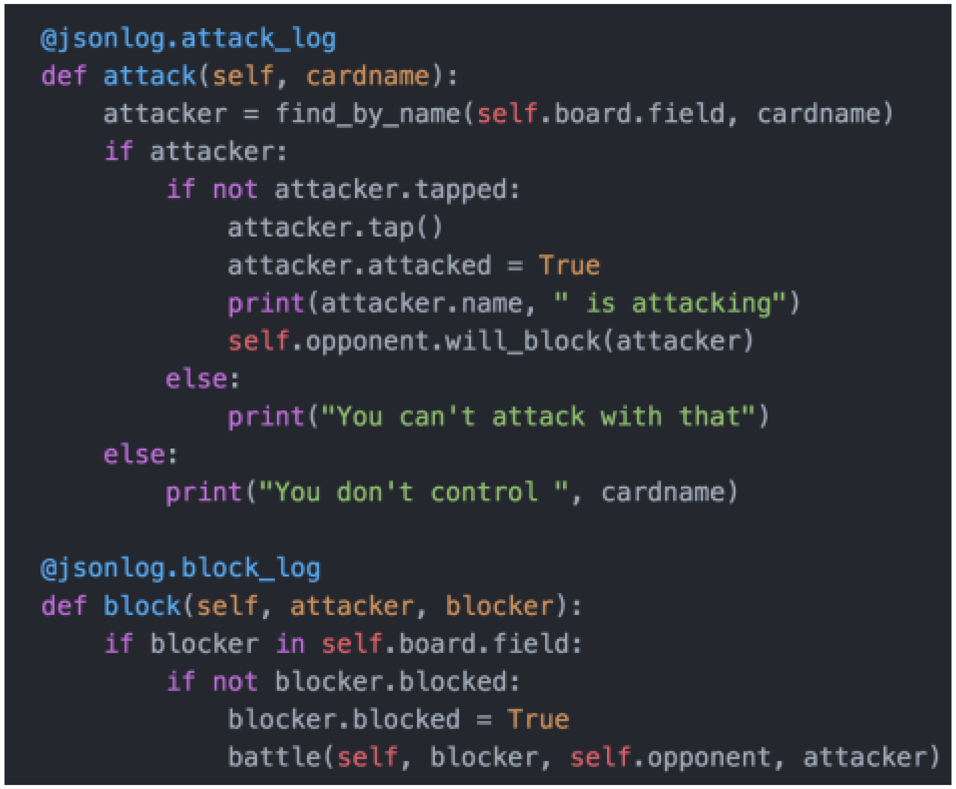
The Player Class (https://github.com/nickhattwick/PlayingWithCards/blob/master/player.py)

Now, let's talk about the "players" as they pertain to the program. Originally, the game was only playable between one human and one computer, but by containing most of the game logic in GameControl, it can exist without being tied to a player, allowing for a lot more flexibility. Since a GameControl object represents the board and logic for one specific player, GameControls can be given to players when the game begins based on who the players are. In order to represent them, a basic Player class was created and initiated with various values that are true for all players as well as references to some functions which exist in different forms across the different types of players .



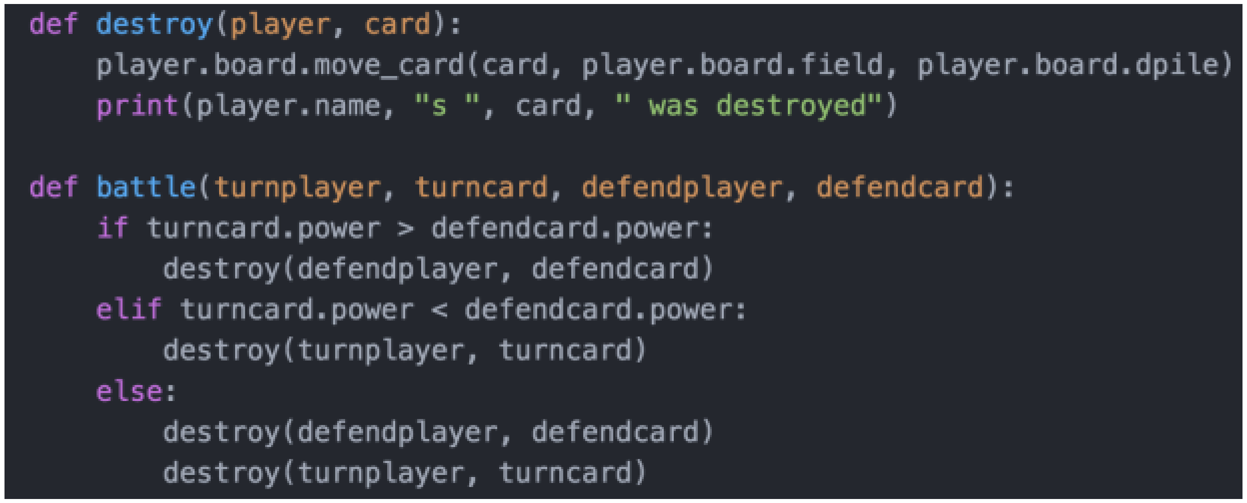
```python
class Player:
    def __init__(self, name):
        self.name = name
        self.life = 20
        self.lose = False
        self.opponent = None
        self.board = GameControl()
```

As you can see, when a player is created it will be given its own GameControl object. While I mentioned that the board contained the game logic, there is an exception that wouldn’t fit nicely. As a GameControl represents one player’s zones and card movements, it can handle most logic except for parts where the two players interact. Because of this, the attacking and blocking logic is currently handled in the Player class.



```python
@jsonlog.attack_log
def attack(self, cardname):
    attacker = find_by_name(self.board.field, cardname)
    if attacker:
        if not attacker.tapped:
            attacker.tap()
            attacker.attacked = True
            print(attacker.name, " is attacking")
            self.opponent.will_block(attacker)
        else:
            print("You can't attack with that")
    else:
        print("You don't control ", cardname)

@jsonlog.block_log
def block(self, attacker, blocker):
    if blocker in self.board.field:
        if not blocker.blocked:
            blocker.blocked = True
            battle(self, blocker, self.opponent, attacker)
```

These functions still call on the board to move the cards, but the logic of attacking and blocking relies on a lot of specifics from the player, due to how many choices need to be made in the process. If one player attacks and the other player blocks, a battle occurs, taken care of by the imported battle program from battle.py and determines which card or cards will be destroyed, calling back to board.py when resolved.



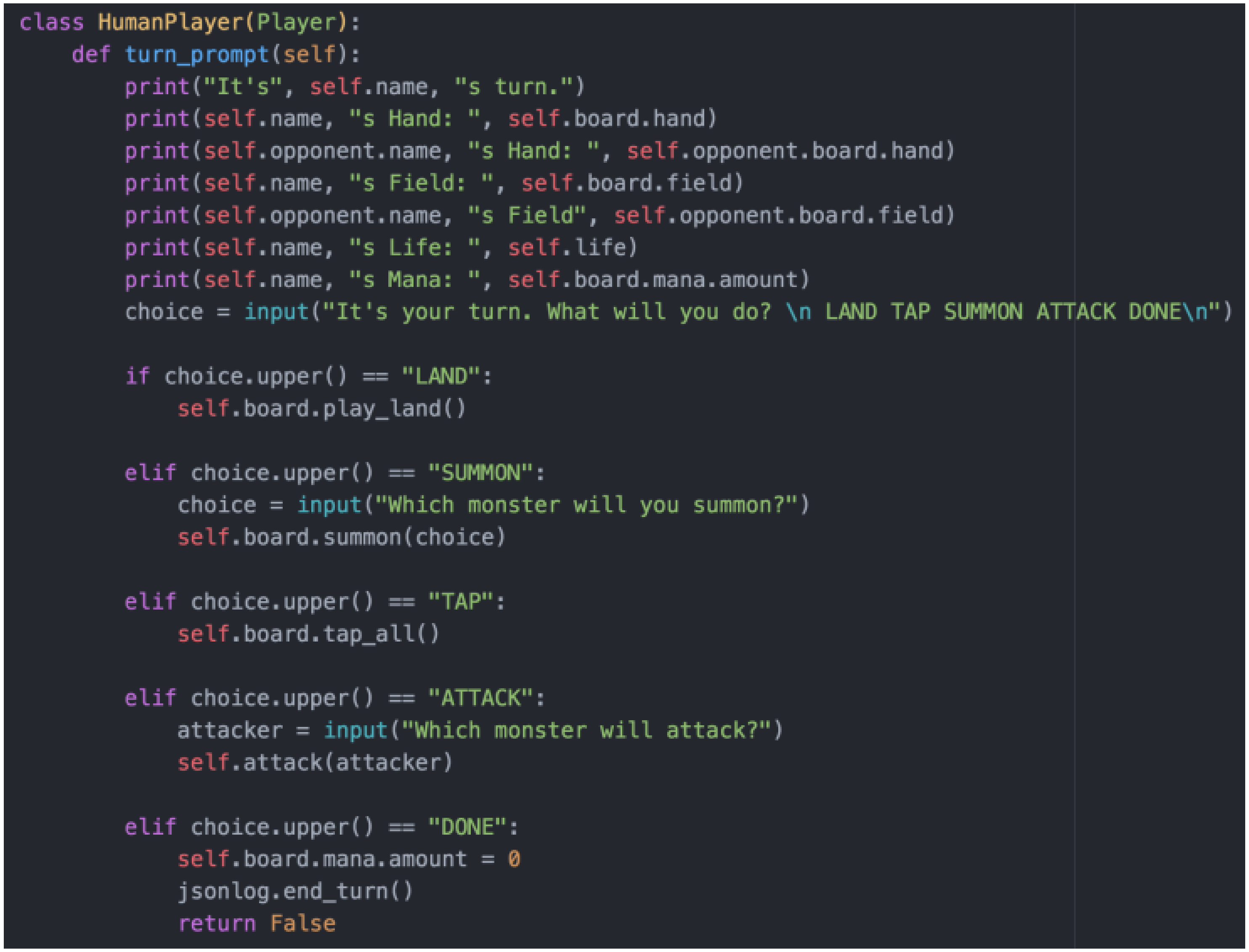
```
def destroy(player, card):
    player.board.move_card(card, player.board.field, player.board.dpile)
    print(player.name, "s ", card, " was destroyed")

def battle(turnplayer, turncard, defendplayer, defendcard):
    if turncard.power > defendcard.power:
        destroy(defendplayer, defendcard)
    elif turncard.power < defendcard.power:
        destroy(turnplayer, turncard)
    else:
        destroy(defendplayer, defendcard)
        destroy(turnplayer, turncard)
```

Again, this function is separated as when the game becomes more complete, the factors involved in battle will also become more complicated. One function to destroy a card which is made to reference one card card movement by passing specific arguments into board’s move_card argument, in this case from a players field to their discard pile. In a future version this function will likely be moved back to board as well as there will be more ways for cards to be destroyed than just the battle function. Additionally, the battle function will become more detailed, but for now it compares two creatures powers and destroys the one with less power. In case of a tie, both are destroyed.

Lastly, while the board object may actually play or tap cards, it’s up for the player to designate the cards that will be played or tapped. However, a single function to determine when and which arguments are passed into the board wouldn’t work because a human would interact differently with the game than a computer. For this, there are subclasses to represent the different types of players and there interactions.

The Human Player

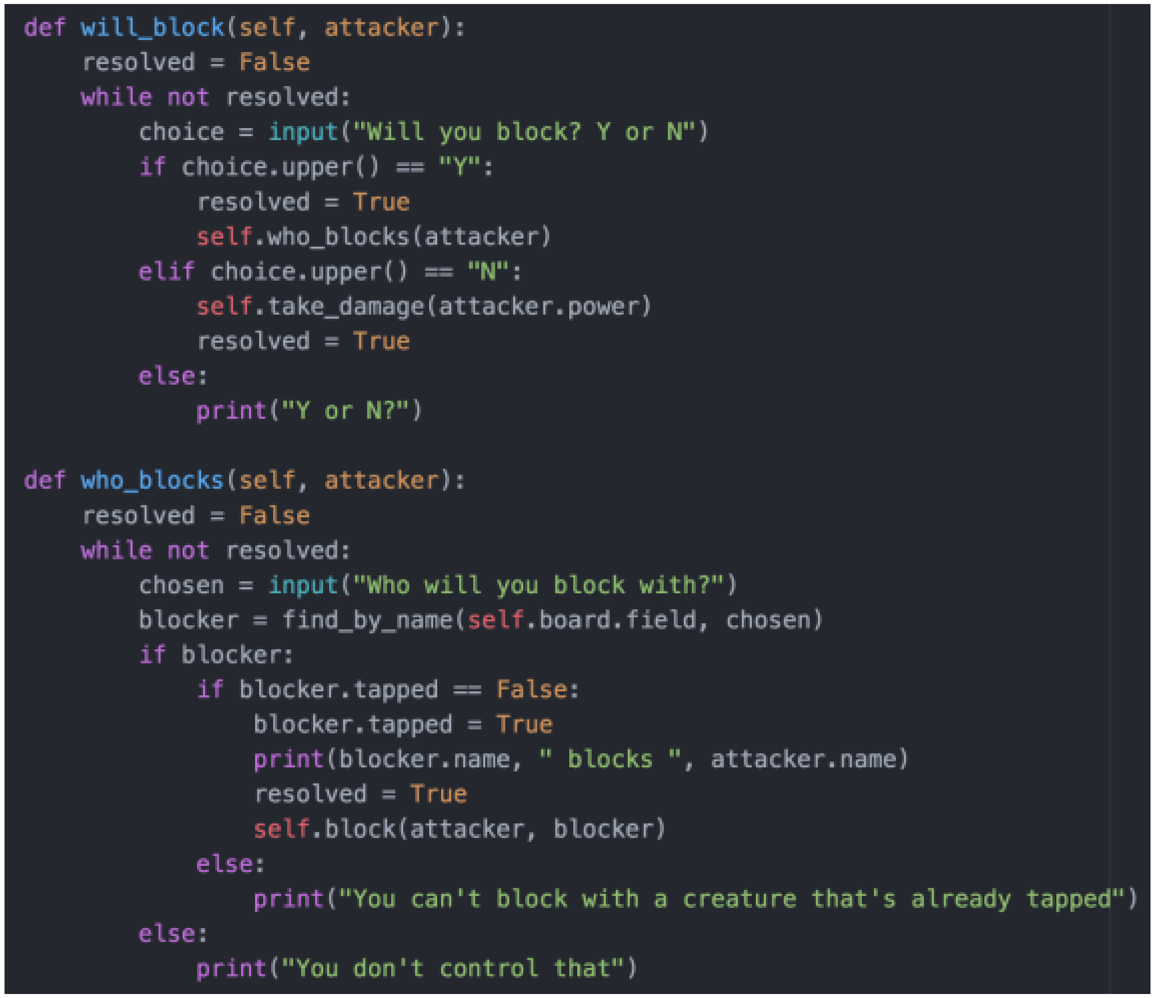
The different players have different turn_prompt functions that will be called to determine how the player’s turn will be run. In the case of the human player, it needs print information based on the game state and give prompts for the player to respond to. After printing information, a “choice” variable will take an input from the player after asking what type of move they want to make.



```python
class HumanPlayer(Player):
    def turn_prompt(self):
        print("It's", self.name, "s turn.")
        print(self.name, "s Hand: ", self.board.hand)
        print(self.opponent.name, "s Hand: ", self.opponent.board.hand)
        print(self.name, "s Field: ", self.board.field)
        print(self.opponent.name, "s Field", self.opponent.board.field)
        print(self.name, "s Life: ", self.life)
        print(self.name, "s Mana: ", self.board.mana.amount)
        choice = input("It's your turn. What will you do? \n LAND TAP SUMMON ATTACK DONE\n")

        if choice.upper() == "LAND":
            self.board.play_land()

        elif choice.upper() == "SUMMON":
            choice = input("Which monster will you summon?")
            self.board.summon(choice)

        elif choice.upper() == "TAP":
            self.board.tap_all()

        elif choice.upper() == "ATTACK":
            attacker = input("Which monster will attack?")
            self.attack(attacker)

        elif choice.upper() == "DONE":
            self.board.mana.amount = 0
            jsonlog.end_turn()
            return False
```

There are four types of moves the human player will be able to choose from. “LAND” will play a land, “SUMMON” will prompt the player to input the card they want to summon calling the board’s summon function on that card, “TAP” will call tap_all for the player and tap all the player’s lands to generate Mana, and “ATTACK” prompts the player for to input which card they want to attack with. Until a player types “DONE”, the program will keep prompting them to inputting moves, with the board file handling the logic to make the move happen and check if a card is valid or not or if the player has already played their land for the turn. After “DONE” is called, the player’s turn will end, allowing the opponent to make their move.

During the opponent's turn, the player's block functions will handle everything related, passing in the blocker and making sure the block is valid, if not, passing back the question of whether or not to block. First, it checks whether they want to block the attacking creature, then which creature they will block with, and then lets the battle function take care of the battle, if it occurs.

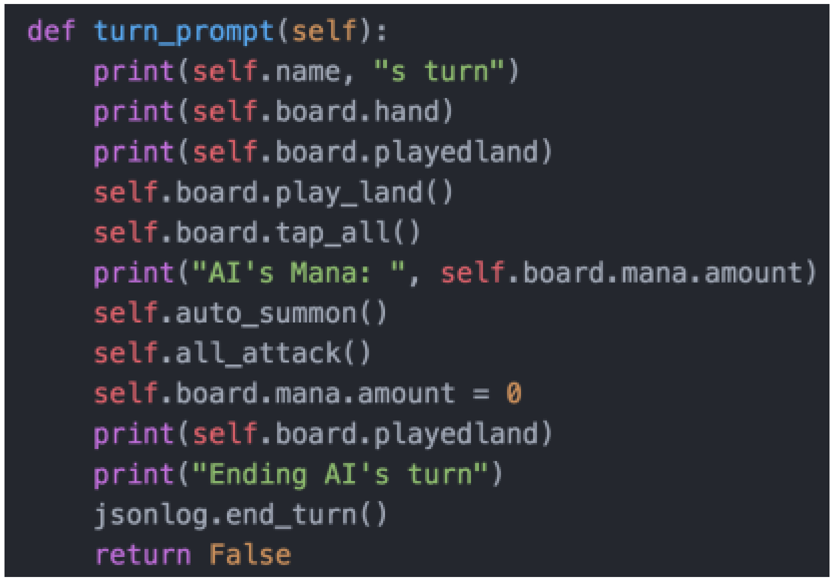


```
def will_block(self, attacker):
    resolved = False
    while not resolved:
        choice = input("Will you block? Y or N")
        if choice.upper() == "Y":
            resolved = True
            self.who_blocks(attacker)
        elif choice.upper() == "N":
            self.take_damage(attacker.power)
            resolved = True
        else:
            print("Y or N?")

def who_blocks(self, attacker):
    resolved = False
    while not resolved:
        chosen = input("Who will you block with?")
        blocker = find_by_name(self.board.field, chosen)
        if blocker:
            if blocker.tapped == False:
                blocker.tapped = True
                print(blocker.name, " blocks ", attacker.name)
                resolved = True
                self.block(attacker, blocker)
            else:
                print("You can't block with a creature that's already tapped")
        else:
            print("You don't control that")
```

Additional game details are printed throughout the program to let players know who wins or what the life totals are, although the above functions are everything specific to the human player.

The AutoPilot

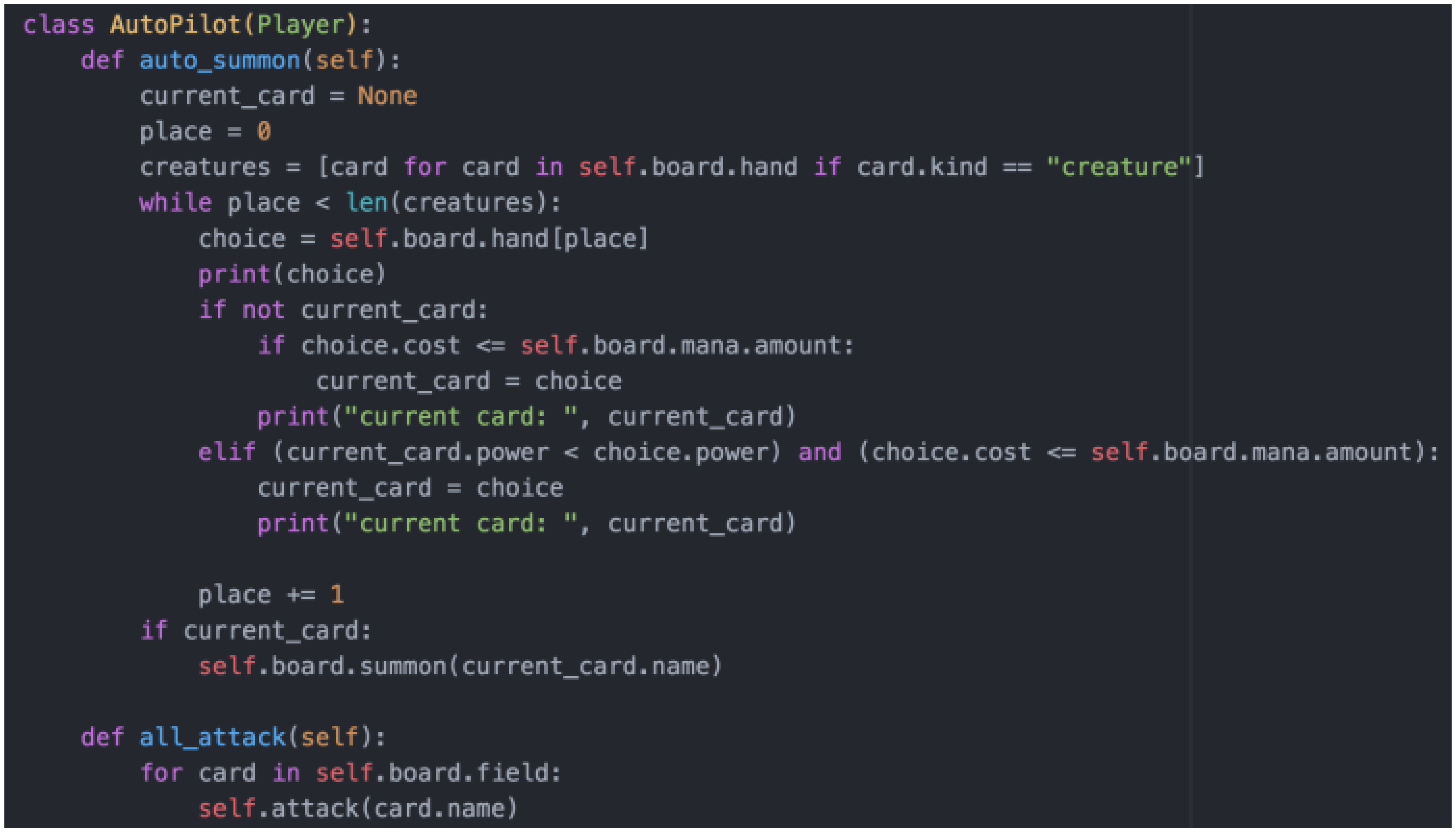
The AutoPilot is the opposing player controlled by the computer, which has fixed logic for its moves. It is not the best strategy for the game, but is made to have a computer player that puts up resistance and which will win if it goes unchallenged. The human player can play against it, and it will give a learning computer player to learn if it wants to win. The leaning player's turn_prompt is a list of different types of moves to call.



```
def turn_prompt(self):
    print(self.name, "s turn")
    print(self.board.hand)
    print(self.board.playedland)
    self.board.play_land()
    self.board.tap_all()
    print("AI's Mana: ", self.board.mana.amount)
    self.auto_summon()
    self.all_attack()
    self.board.mana.amount = 0
    print(self.board.playedland)
    print("Ending AI's turn")
    jsonlog.end_turn()
    return False
```

First it plays a land, then taps all its mana, and then runs its summon function and attacks with everything, before ending its turn.

The AutoPilot also has two other functions that make it work; auto_summon and all_attack. While all_attack is a “for loop” that just has all creatures the player controls attack one at a time, the AutoPilot’s auto_summon function has the decision making for which creature the player wants to summon. In this case, it’s a loop that plays the strongest card the player has the Mana to play.



```python
class AutoPilot(Player):
    def auto_summon(self):
        current_card = None
        place = 0
        creatures = [card for card in self.board.hand if card.kind == "creature"]
        while place < len(creatures):
            choice = self.board.hand[place]
            print(choice)
            if not current_card:
                if choice.cost <= self.board.mana.amount:
                    current_card = choice
                print("current card: ", current_card)
            elif (current_card.power < choice.power) and (choice.cost <= self.board.mana.amount):
                current_card = choice
                print("current card: ", current_card)

            place += 1
        if current_card:
            self.board.summon(current_card.name)

    def all_attack(self):
        for card in self.board.field:
            self.attack(card.name)
```

While a better attack/block strategy will need to be implemented later, the all out attack strategy means that the computer will never block, so the computer's will_block function will just call the player's take_damage function.

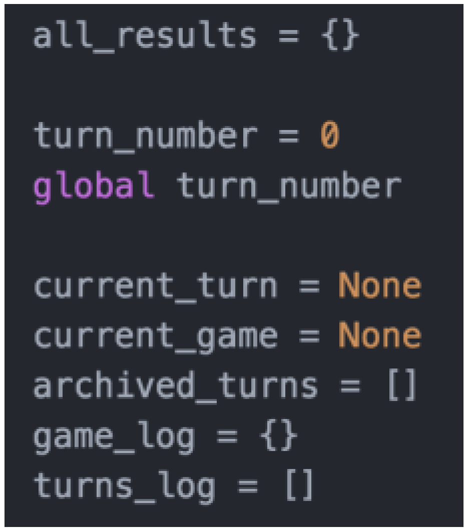
The last of the players is called StillLearning, and is the computer player that uses an updating strategy. It presently uses AutoPilot as the parent class and changes the auto_summon function to value cards differently. But before getting into that, let's talk about the logging system the game uses.

The Logging System

The structure of the current logging system is one file, jsonlog.py, is run along with the game, called through decorators, which keeps a log of every single move made through the course of a game and stores it and stores them in results.json. After a game is finished, running parsing.py will take all of the summons from results.json and create/update a dictionary for each card with the amount of times that card was played by the winning player and each time it was used by the losing player and store it in parsed.json. The last part of the logging cycle is the player that uses it. StillLearning will use the ratios in parsed.json to determine which cards it plays.

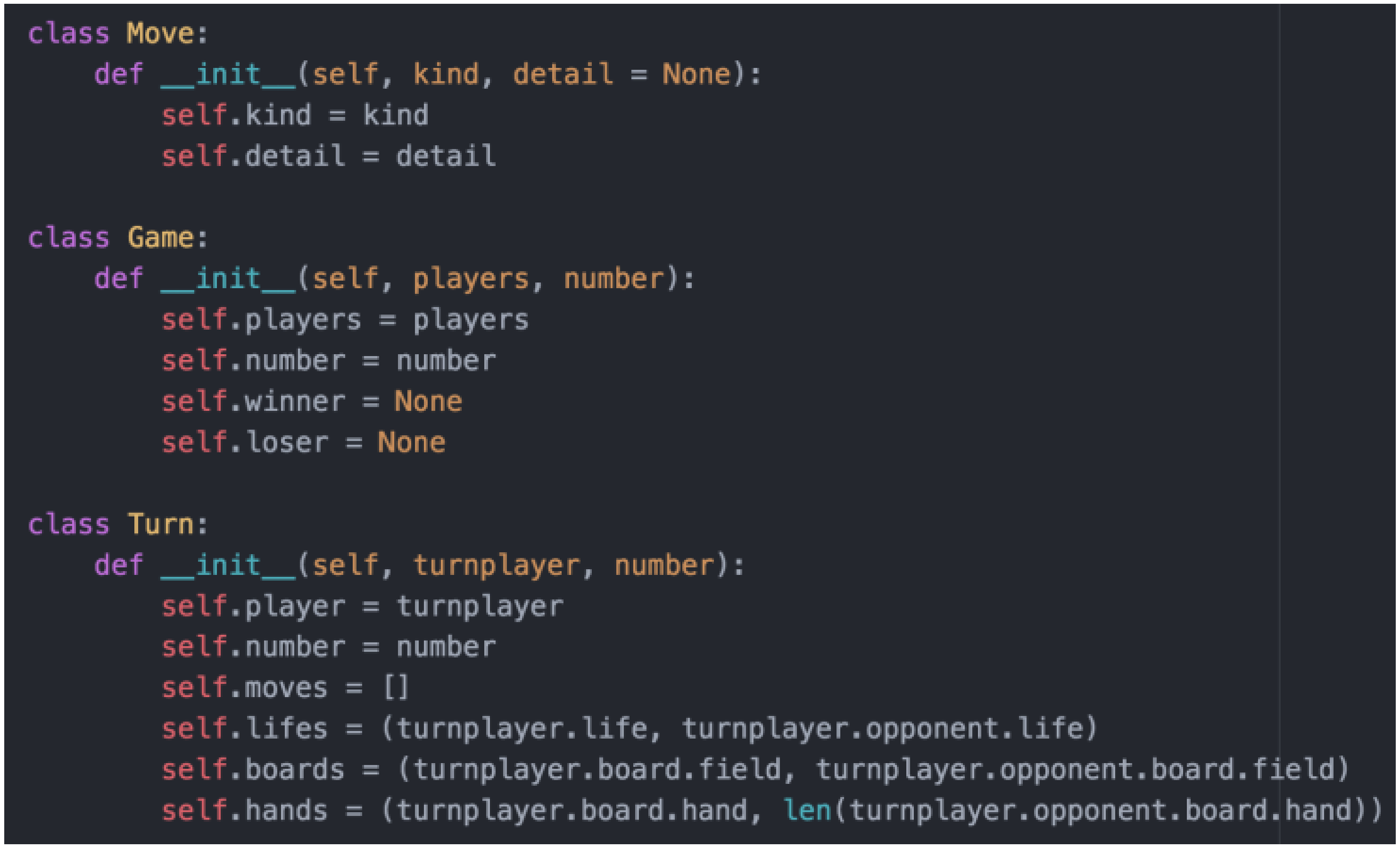
The Game Log (https://github.com/nickhattwick/PlayingWithCards/blob/master/jsonlog.py)

First, let's talk about the initial logging file. Jsonlog is made to collect a log of everything in a game. Initially, the program creates some blank variables to be edited as games are played. The turn number will be increased as part of the program as a way of incorporating the turn number into the game.



```
all_results = {}

turn_number = 0
global turn_number

current_turn = None
current_game = None
archived_turns = []
game_log = {}
turns_log = []
```

The information that's logged will also be separated into three different classes. One that creates moves, one that creates turns, and one that creates games.

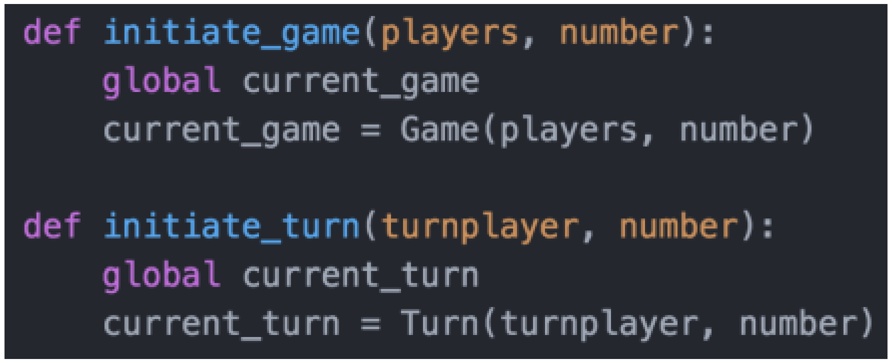


```python
class Move:
    def __init__(self, kind, detail = None):
        self.kind = kind
        self.detail = detail

class Game:
    def __init__(self, players, number):
        self.players = players
        self.number = number
        self.winner = None
        self.loser = None

class Turn:
    def __init__(self, turnplayer, number):
        self.player = turnplayer
        self.number = number
        self.moves = []
        self.lifes = (turnplayer.life, turnplayer.opponent.life)
        self.boards = (turnplayer.board.field, turnplayer.opponent.board.field)
        self.hands = (turnplayer.board.hand, len(turnplayer.opponent.board.hand))
```

The Moves class takes two arguments; a kind and a detail. Kind represents what category of move it is (land, tap, summon, attack, etc.) while the detail would be an optional variable if more is needed to describe the move. For instance, the card that was summoned or the card that attacked would be the detail for the “attack” or “summon” kinds, while the amount of Mana generated would be the detail for “tap”. The Game takes in a list of players in the game and a game number (currently not being implemented) as well as winner and loser values to be added after the game’s conclusion.

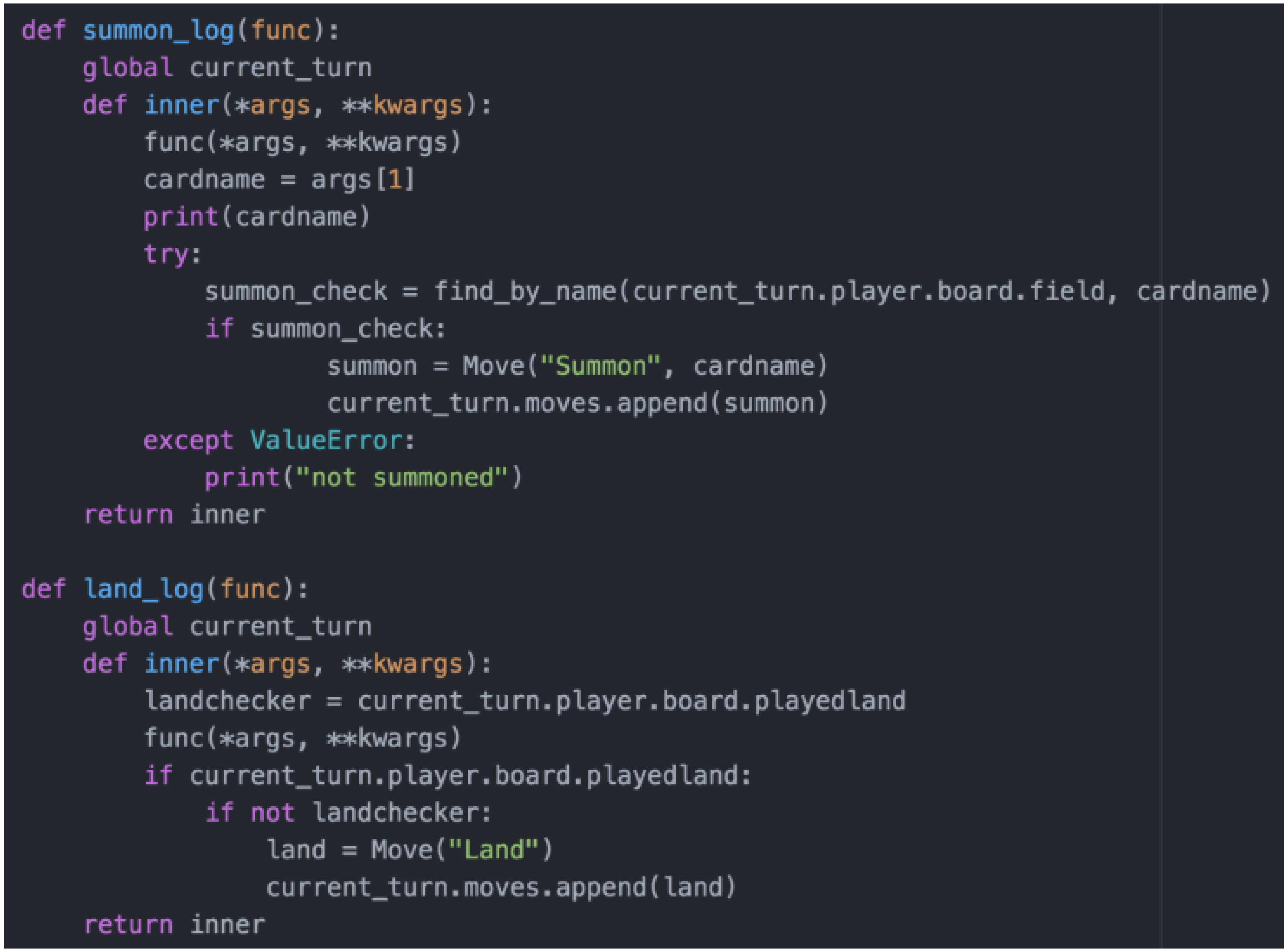
The largest of the class objects is the Turn object which starts with a player and a turn number, as well as the life totals, board states and knowledge of the player’s hand and number of cards in the opponent’s hand. The goal is for the turn log to log all information the player has access to, in order to see what the state the game was when a move was made. A list of moves is also added to the turn object as a way of sorting moves by player and turn.

Games and turns are created with help from the initiate_game and initiate_turn functions respectively.



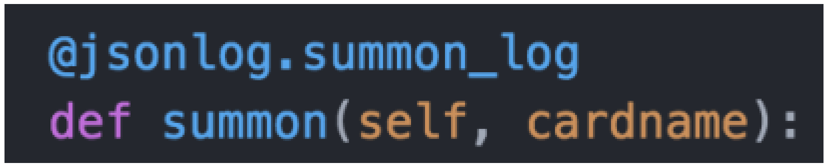
```python
def initiate_game(players, number):
    global current_game
    current_game = Game(players, number)

def initiate_turn(turnplayer, number):
    global current_turn
    current_turn = Turn(turnplayer, number)
```

By passing in the variables needed to make the object, these functions will update the current game or turn to reflect the turn in progress. Of course, the turn will need a way to add moves, which are done via different types of log functions. For instance, summon_log will add a Move object with "Summon" as its kind and the card name for its detail. The land log will just record "Land" as the move's kind.



```
def summon_log(func):
    global current_turn
    def inner(*args, **kwargs):
        func(*args, **kwargs)
        cardname = args[1]
        print(cardname)
        try:
            summon_check = find_by_name(current_turn.player.board.field, cardname)
            if summon_check:
                    summon = Move("Summon", cardname)
                    current_turn.moves.append(summon)
        except ValueError:
            print("not summoned")
    return inner

def land_log(func):
    global current_turn
    def inner(*args, **kwargs):
        landchecker = current_turn.player.board.playedland
        func(*args, **kwargs)
        if current_turn.player.board.playedland:
            if not landchecker:
                land = Move("Land")
                current_turn.moves.append(land)
    return inner
```

These log functions were one of the most eye-opening things for me during the mentorship program. Obviously, I'd heard that Python was an object oriented programming language, but before my mentor taught me about decorators, it had never occurred to me to think of functions as objects. These decorator functions take in another function as well as all of its arguments, and run the added decorator's code along with the code of the original function. These functions are also called in a unique way which makes it very easy to tell which functions are being decorated.

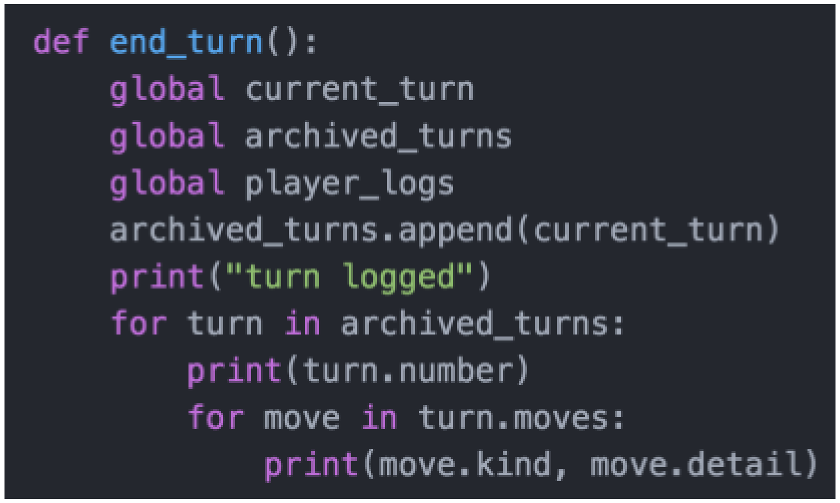


```
@jsonlog.summon_log
def summon(self, cardname):
```

Simply putting “@” followed by the decorator function over the function being decorated. In this case, summon_log is made to take in the summon function from the board and edit it to give it logging abilities. What the decorator function does in this case is takes the name of the card the summon function is given and checks if it is on the field after running the function, and creates and adds the move to the current turn’s move log if so. The land_log function uses a similar method by checking whether the board’s playedland value for the turn changes from False to True, before logging the move.

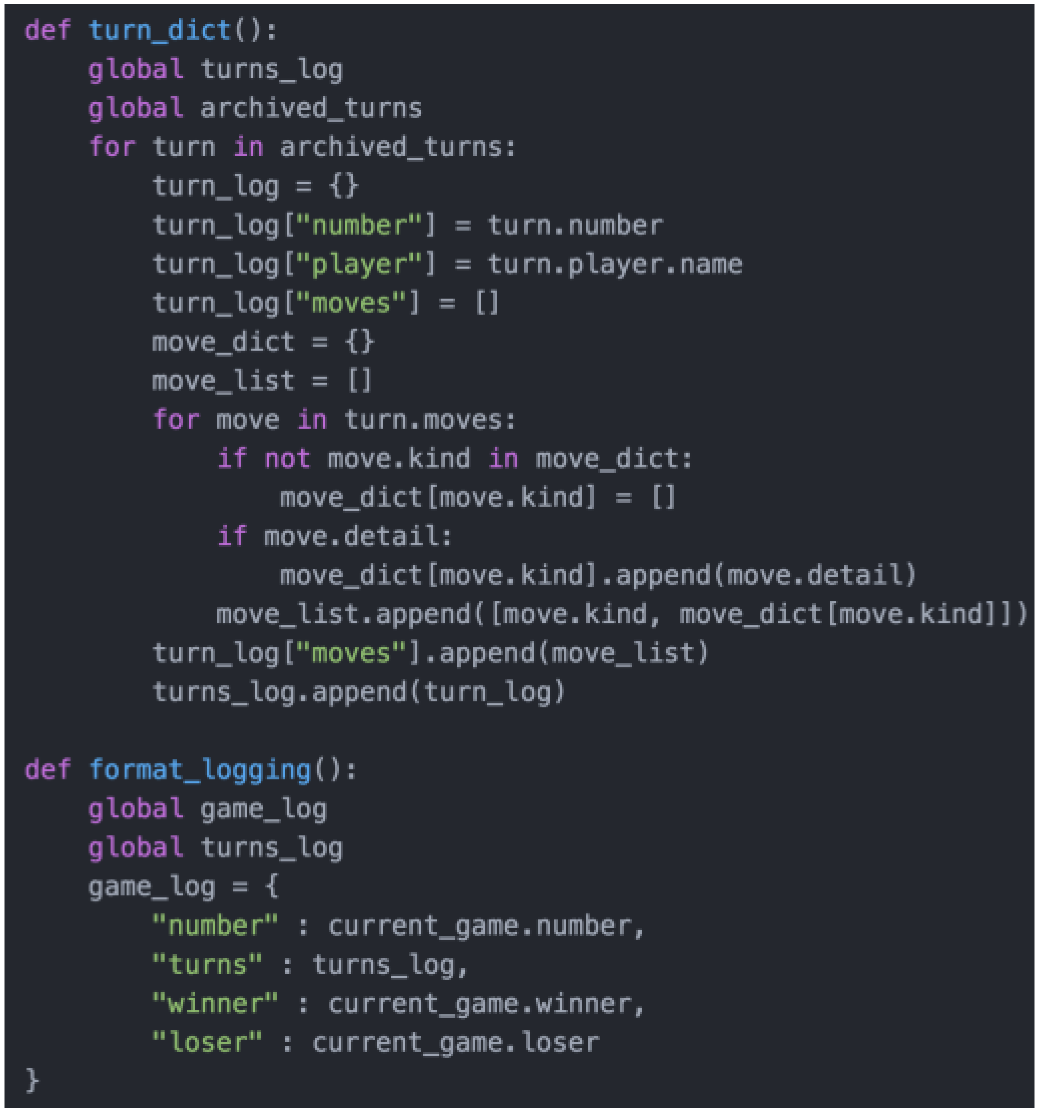
In addition to the summon_log and land_log functions, there are also logs for tapping, attacking, and blocking, which work in the same way. All of these logs are placed using the the same decorator format, run a function and have a way of checking to see if the move happened by checking the status of the card. For the tap log, it checks whether a land switched from untapped to tapped. In the case of the attack and block logs, it checks if the creature’s attacked and blocked attributes change from false to true. In its present form it works pretty well, although I could see a scenario where it messes up due to multiple cards of the same name being played, so these functions are something I’ll want to look what I can change in the next version.

The remaining functions in jsonlog.py help to tie things together. First of, the end_turn function is called at the end of a players turn in the player file and takes care of archiving the current turn before the next turn starts and a new object is assigned to current_turn.



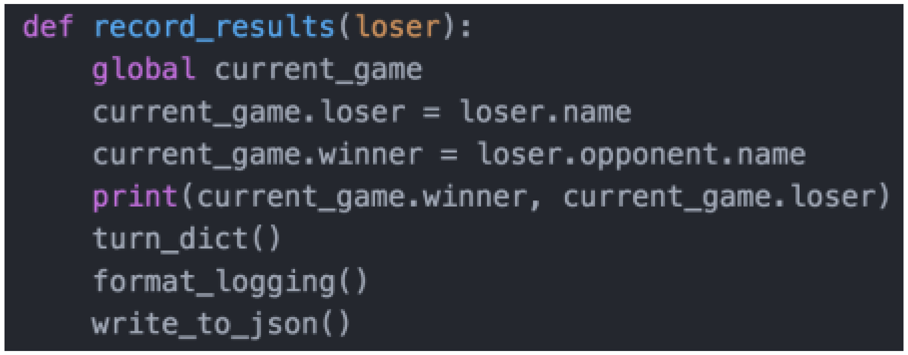
```
def end_turn():
    global current_turn
    global archived_turns
    global player_logs
    archived_turns.append(current_turn)
    print("turn logged")
    for turn in archived_turns:
        print(turn.number)
        for move in turn.moves:
            print(move.kind, move.detail)
```

The next functions take the variables that have been stored and create a dictionary that’s organized to be stored in JSON. The turn_dict function deals with taking each turn in archived_turns and creating a dictionary to organize the turns. Each turn dictionary will have “number”, “player” and “moves” keys. While setting the values for the turn’s number and player values is pretty simple, organizing the moves is a bit more difficult. Initially the move will be set to a list, and then each move will have a dictionary created for it where the move's kind is the key and the value is the detail. A list is created of each kind/detail combination and then appended to the turn_log's "moves" list. Finally, the whole turn_log will be stored in turns_log, which will be a list of all turn dictionaries.



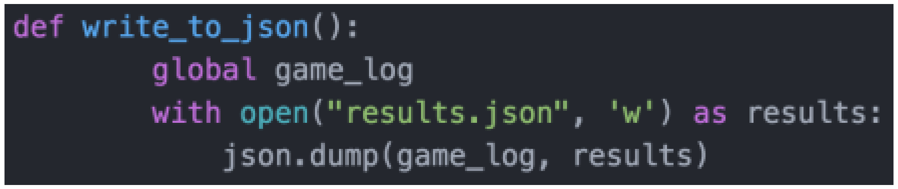
```python
def turn_dict():
    global turns_log
    global archived_turns
    for turn in archived_turns:
        turn_log = {}
        turn_log["number"] = turn.number
        turn_log["player"] = turn.player.name
        turn_log["moves"] = []
        move_dict = {}
        move_list = []
        for move in turn.moves:
            if not move.kind in move_dict:
                move_dict[move.kind] = []
            if move.detail:
                move_dict[move.kind].append(move.detail)
            move_list.append([move.kind, move_dict[move.kind]])
        turn_log["moves"].append(move_list)
        turns_log.append(turn_log)

def format_logging():
    global game_log
    global turns_log
    game_log = {
        "number" : current_game.number,
        "turns" : turns_log,
        "winner" : current_game.winner,
        "loser" : current_game.loser
}
```

While turn_dict creates the dictionary for the turn, format_logging creates the final dictionary that will be stored. It's a pretty simple function that wrap everything up, giving the game dictionary "number", "winner", and "loser" values and storing the turns_log in "turns." These functions are run in order when the record_results function is called.



```
def record_results(loser):
    global current_game
    current_game.loser = loser.name
    current_game.winner = loser.opponent.name
    print(current_game.winner, current_game.loser)
    turn_dict()
    format_logging()
    write_to_json()
```

The record_results function is called in player.py when take_damage is called on a player resulting in their life points running out. Since it's called by the losing player, their name is the one passed in and the winner is the opponent. After calling turn_dict and format_logging, the last function called is write_to_json which takes the dictionary and puts it in results.json.

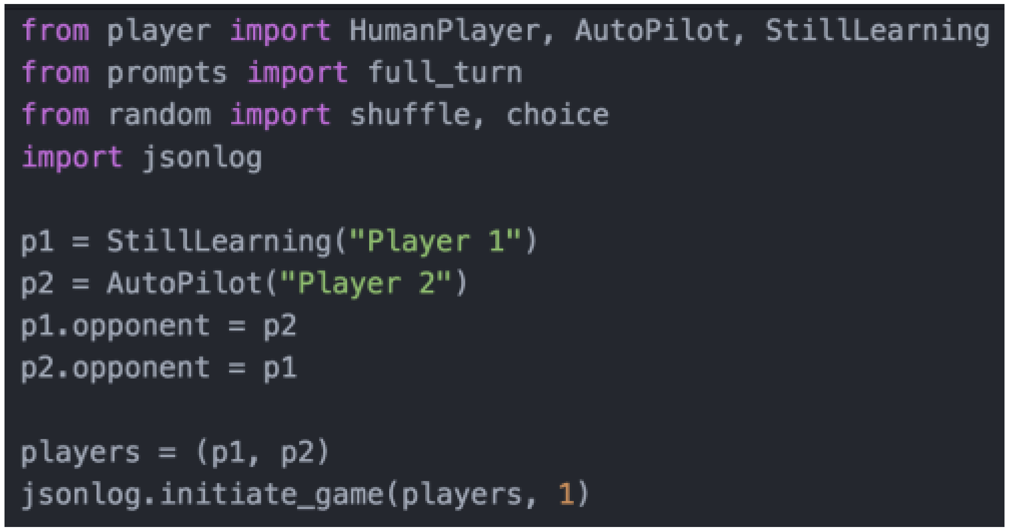


```
def write_to_json():
        global game_log
        with open("results.json", 'w') as results:
            json.dump(game_log, results)
```

This wraps up jsonlog.py's functionality and most of the first log itself, but as much as I tried to completely separate the log from the game, some of the functions still ended up being tied together. Most of the functions so far have been shown without the program that actually runs them.

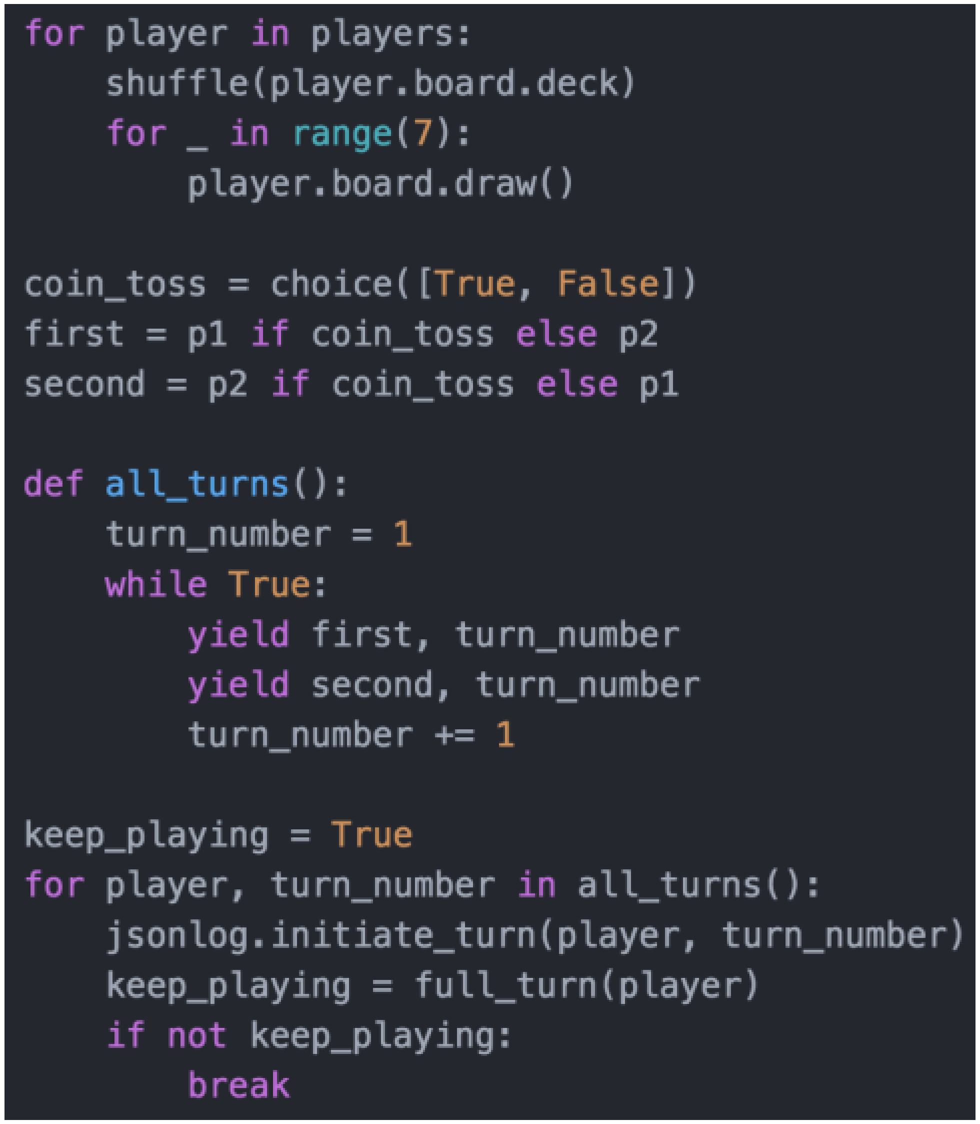
Rungame.py (https://github.com/nickhattwick/PlayingWithCards/blob/master/rungame.py)

Rungame.py is the file that a person will run to kick off the game as well as the logging system. It imports the Player subclasses from player as well as a function called full_turn, which I'll get to shortly. It uses the shuffle and choice functions from the random library and finally imports jsonlog.



```
from player import HumanPlayer, AutoPilot, StillLearning
from prompts import full_turn
from random import shuffle, choice
import jsonlog

p1 = StillLearning("Player 1")
p2 = AutoPilot("Player 2")
p1.opponent = p2
p2.opponent = p1

players = (p1, p2)
jsonlog.initiate_game(players, 1)
```

The first thing it does it set two players from the different classes and calls the log's initiate game function with those players. Currently it's set to have StillLearning face off against the AutoPilot, though in order to play with the human, one of these would just have to be switched with HumanPlayer. Next, it shuffles the players' decks and gives them each a 7-card starting hand and decides the player who will play first.



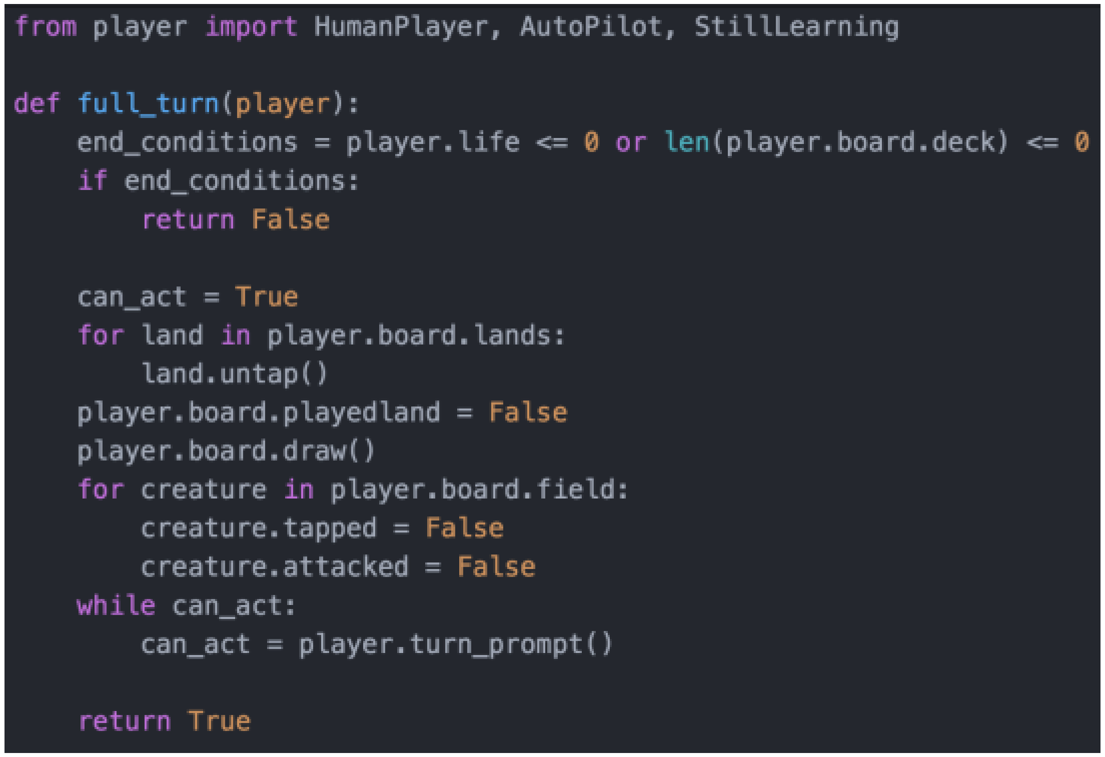
```python
for player in players:
    shuffle(player.board.deck)
    for _ in range(7):
        player.board.draw()

coin_toss = choice([True, False])
first = p1 if coin_toss else p2
second = p2 if coin_toss else p1

def all_turns():
    turn_number = 1
    while True:
        yield first, turn_number
        yield second, turn_number
        turn_number += 1

keep_playing = True
for player, turn_number in all_turns():
    jsonlog.initiate_turn(player, turn_number)
    keep_playing = full_turn(player)
    if not keep_playing:
        break
```

From there it uses the all_turns functions to generate the variables needed to create a turn object for the log. First it stores a turn_number variable then each time it's called it switches between outputting the player going first and the player going second, along with the turn number. Finally, once both are called, it increases the turn number and loops back to returning the first player.

Next it makes a new variable called keep_playing, sets it to True and runs a "for loop" that will run through all_turns and alternate whose turn it is, switching players whenever keep_playing becomes False. For each player's turn, it calls jsonlog's initiate_turn function passing in the player and turn_number provided by all turns. Next it will call the full_turn for the player by setting the value of keep_playing equal to what full_turn is returning. Eventually full_turn will return False and the loop will move on.

Moving on to the full_turn function stored in prompts.py, this is what calls for players to run their turns and takes care of some of the logic for reseting turns. The first thing full_turn does is set the conditions for it to return False, which are that a player's life points become zero or a player runs out of cards. Using another True/False switch to determine when it Full Turn stop, the program sets can_act equal to True and moves on to the functions that start a player's turn.

First this will untap each of the player's lands, set their playedland value to False so they can play a land again, draws a card for the player and untaps each creature and sets its attacked value (used to check for the log) back to False.

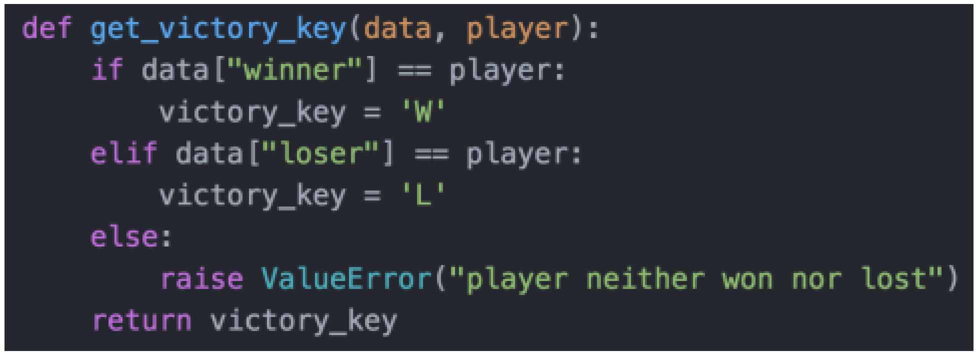

```
from player import HumanPlayer, AutoPilot, StillLearning

def full_turn(player):
    end_conditions = player.life <= 0 or len(player.board.deck) <= 0
    if end_conditions:
        return False

    can_act = True
    for land in player.board.lands:
        land.untap()
    player.board.playedland = False
    player.board.draw()
    for creature in player.board.field:
        creature.tapped = False
        creature.attacked = False
    while can_act:
        can_act = player.turn_prompt()

    return True
```

Then lastly, it sets can_act equal to the player's turn_prompt, calling the turn_prompt and letting it run until it returns False. This system takes care of passing the turns back and forth between players. Overall, while most of the game logic is handled elsewhere, rungame.py and prompts.py have an import role of combining this logic into a runnable program.

So far, the code shown and talked about is what runs the entirety of my game and the initial logging of each game. Now let's talk about the second log and the last player object that learns as it goes.

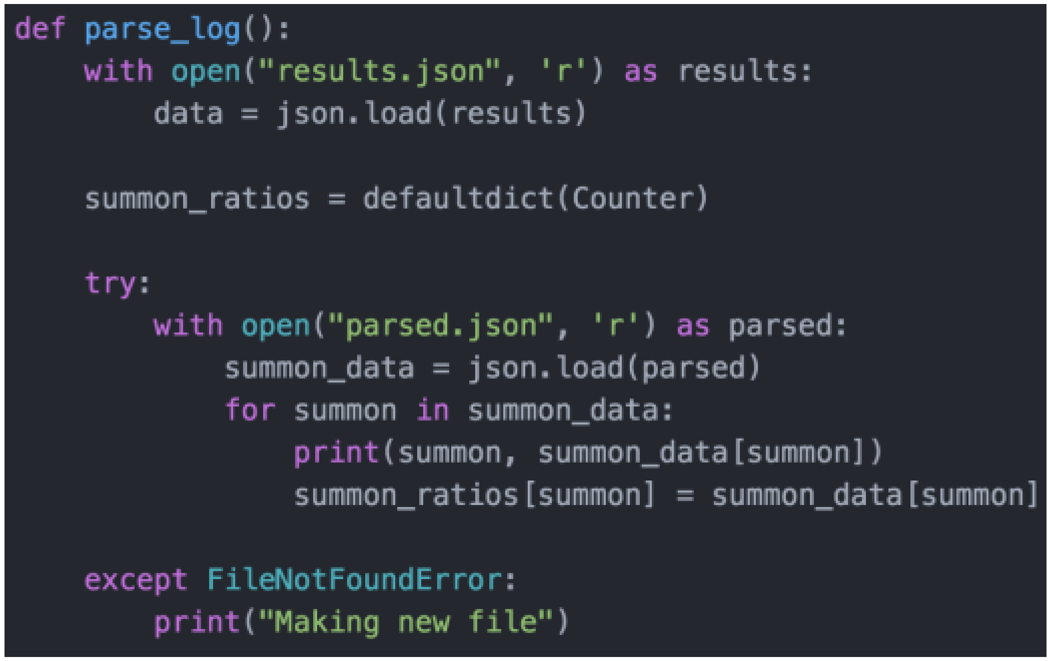
Parsing JSON (https://github.com/nickhattwick/PlayingWithCards/blob/master/jsonlog.py)

The two parts of my current learning player are parsing.py and the StillLearning player object. First let's talk about the parsing file, which takes results.json and updates a dictionary of each creature summoned and its win to loss ratio. Parsing.py consists of two functions. The first is get_victory_key, which takes in a data set and a player and outputs a victory key of 'W' if the data has a winner value that's the same as the player or 'L' if it has a loser value of the player.



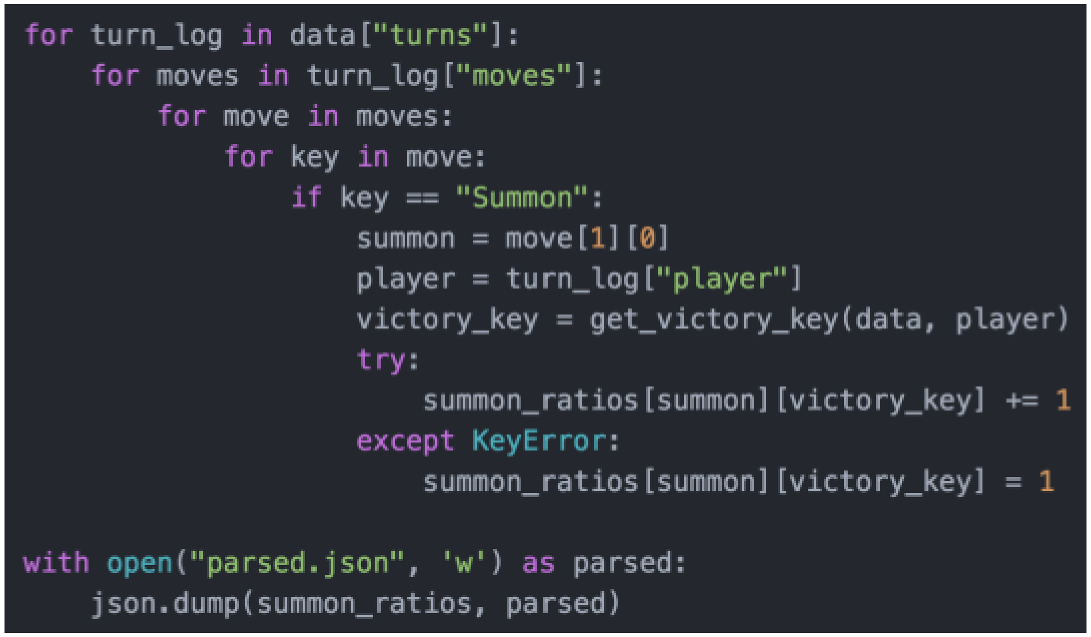
```
def get_victory_key(data, player):
    if data["winner"] == player:
        victory_key = 'W'
    elif data["loser"] == player:
        victory_key = 'L'
    else:
        raise ValueError("player neither won nor lost")
    return victory_key
```

Next up, the function that uses get_victory_key is parse_log which opens results and creates a dictionary called summon_ratios. It then opens parsed.json and copies any existing dictionaries in it into summon_ratios.



```
def parse_log():
    with open("results.json", 'r') as results:
        data = json.load(results)

    summon_ratios = defaultdict(Counter)

    try:
        with open("parsed.json", 'r') as parsed:
            summon_data = json.load(parsed)
            for summon in summon_data:
                print(summon, summon_data[summon])
                summon_ratios[summon] = summon_data[summon]

    except FileNotFoundError:
        print("Making new file")
```

After that, it looks through each turn in results.json and finds each summon that was made. It calls get_victory_key with the player who's turn the summon was made during. Then for each card summoned, it adds one count to the value stored in the summon's key for the corresponding victory_key or creates the key with a value of one if it doesn't exist.

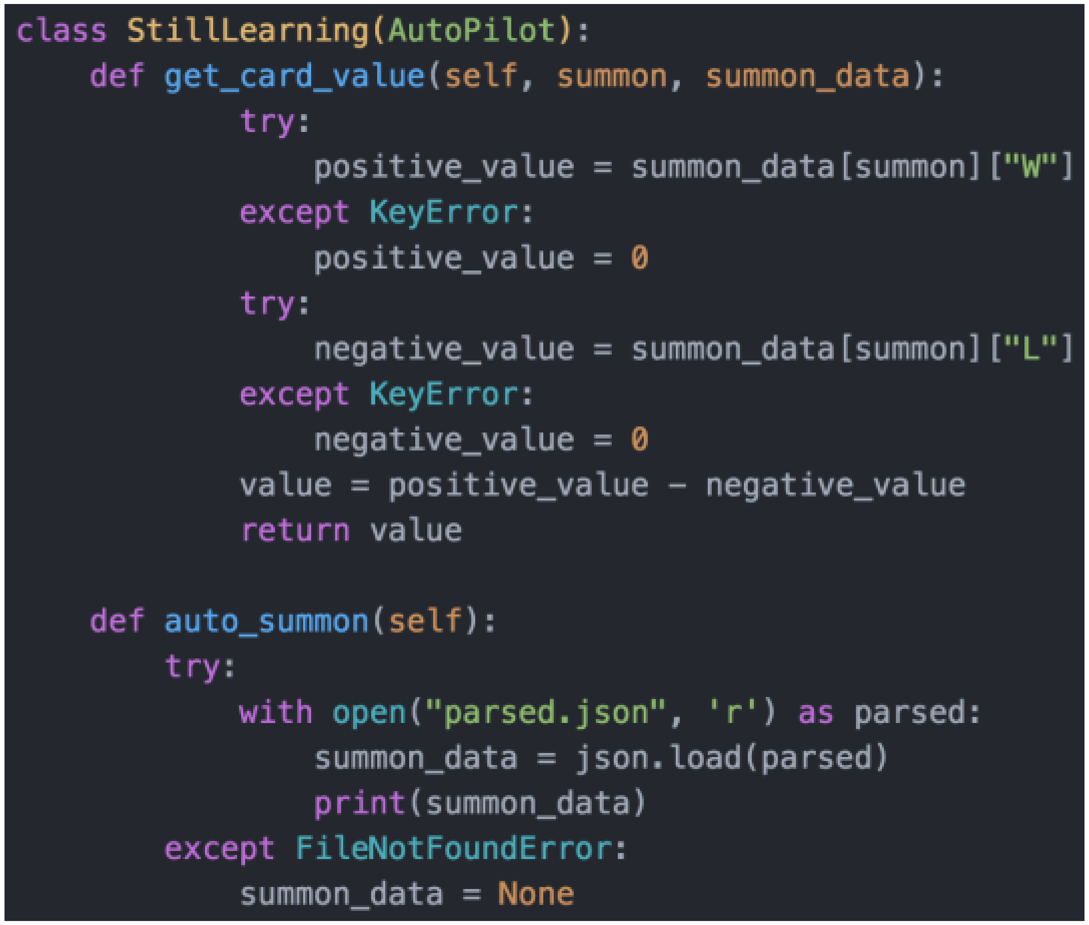


```python
for turn_log in data["turns"]:
    for moves in turn_log["moves"]:
        for move in moves:
            for key in move:
                if key == "Summon":
                    summon = move[1][0]
                    player = turn_log["player"]
                    victory_key = get_victory_key(data, player)
                    try:
                        summon_ratios[summon][victory_key] += 1
                    except KeyError:
                        summon_ratios[summon][victory_key] = 1

with open("parsed.json", 'w') as parsed:
    json.dump(summon_ratios, parsed)
```

After going through each summon in the game and placing them accordingly, the program finally overwrites parsed.json with summon_ratios which has now combined the new files summon ratios with ones that were in parsed.json before.

StillLearning (https://github.com/nickhattwick/PlayingWithCards/blob/master/player.py)

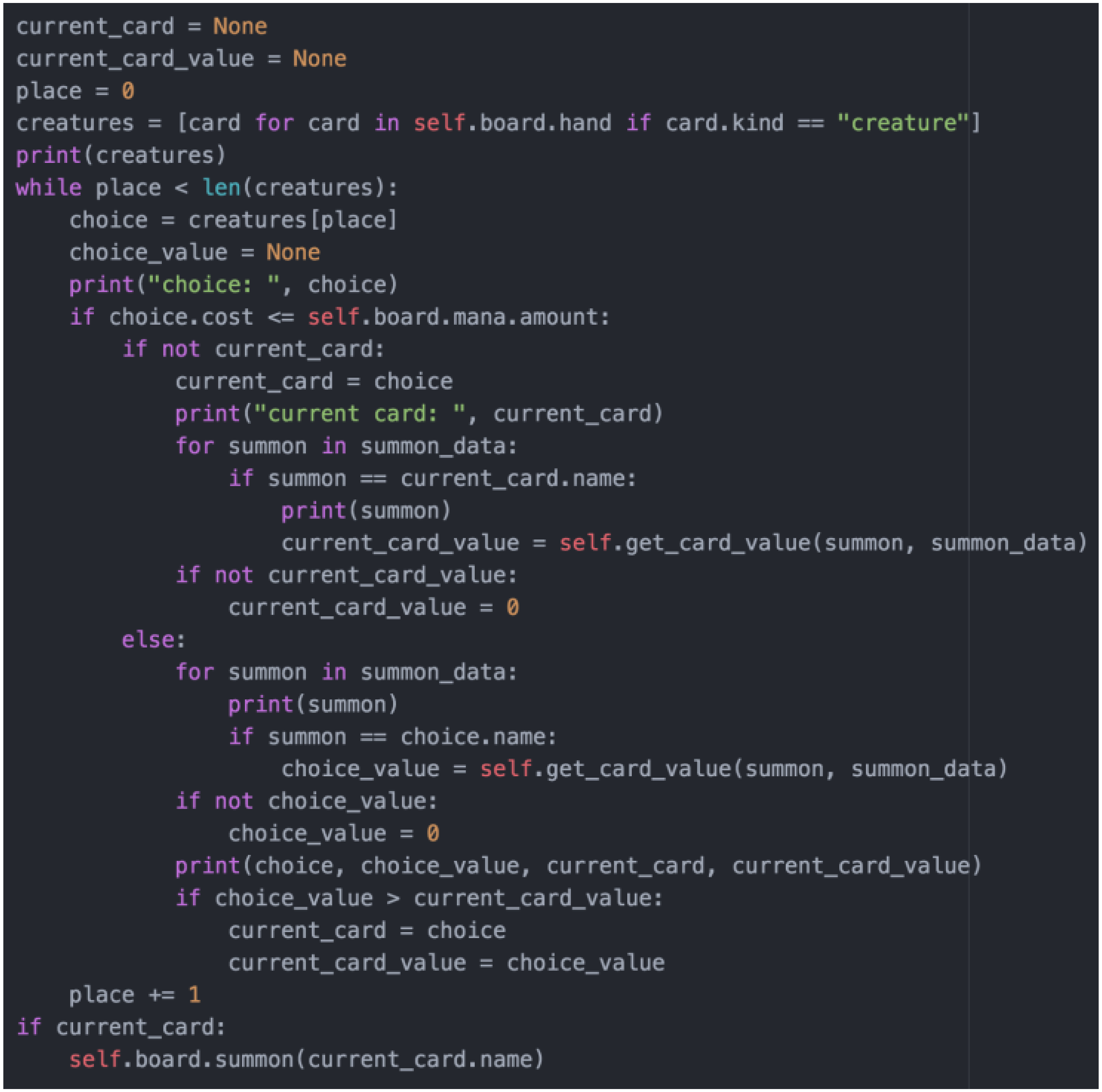
Finally, the StillLearning player object takes in parsed.json and uses it to inform what moves it's going to make. This new player inherits the AutoPilot's function and brings in two new functions. First, it has a function called get_card_value which it uses to rank cards from the parsed.json file. Then it creates a new version of auto_summon to use this card value to determine which card it summons. Then it runs the same turn as AutoPilot, with a change in in the auto_summon function.



```python
class StillLearning(AutoPilot):
    def get_card_value(self, summon, summon_data):
            try:
                positive_value = summon_data[summon]["W"]
            except KeyError:
                positive_value = 0
            try:
                negative_value = summon_data[summon]["L"]
            except KeyError:
                negative_value = 0
            value = positive_value - negative_value
            return value

    def auto_summon(self):
        try:
            with open("parsed.json", 'r') as parsed:
                summon_data = json.load(parsed)
                print(summon_data)
        except FileNotFoundError:
            summon_data = None
```

The get_card_value function is a relatively simple one, it takes in a card being summoned and a list of data. It takes in the existing count in the card's "W" dictionary and subtracts the card's "L" value from it and returns the value.

Next, auto summon starts by opening up parsed.json and loading it into summon data. From there, it sets empty current_card and current_card_value variables and creates a list of the creatures in the players hand. Looping through the creatures in the list, it sets the current creature as a choice. Then it proceeds if the card is one the player has enough Mana to summon. If the player doesn't yet have a current_card then the first creature the player can possibly summon will be put in that slot and use get_card_value to get the current_card_value after searching for the card in summon_data.



```
current_card = None
current_card_value = None
place = 0
creatures = [card for card in self.board.hand if card.kind == "creature"]
print(creatures)
while place < len(creatures):
    choice = creatures[place]
    choice_value = None
    print("choice: ", choice)
    if choice.cost <= self.board.mana.amount:
        if not current_card:
            current_card = choice
            print("current card: ", current_card)
            for summon in summon_data:
                if summon == current_card.name:
                    print(summon)
                    current_card_value = self.get_card_value(summon, summon_data)
            if not current_card_value:
                current_card_value = 0
        else:
            for summon in summon_data:
                print(summon)
                if summon == choice.name:
                    choice_value = self.get_card_value(summon, summon_data)
            if not choice_value:
                choice_value = 0
            print(choice, choice_value, current_card, current_card_value)
            if choice_value > current_card_value:
                current_card = choice
                current_card_value = choice_value
    place += 1
if current_card:
    self.board.summon(current_card.name)
```

If there's already a current_card, then it will calculate the value of each possible summon and replace current_card with the new card and value if they are higher. Finally, once it's thought about each creature in the player's hand it will summon current_card (if there is one).

Currently StillLearning's only learning function is its summon function with the rest of the turn working the same as the AutoPilot, but as the log changes, so will the values the player uses to determine which card to play.

That concludes my full description of the program. In order to run it with StillLearning against the AutoPilot, run rungame.py in Python3 to play a game and create the results log, then run parsing.py in order to update the parsed log that StillLearning uses. You can also run humangame.py in order to play a game using the keyboard to input moves.

Conclusion and Next Steps

Overall, I'm very pleased with how my project came out. As my very first programming project, I made a simplified yet working game along with an AI player that has an updating strategy. I'm also very happy with the organization of the program since it will be much easier to add new features to and try out different learning strategies since I won't have to change much of the code to get new code to work.

So where would I like to go from here? I can see three different ways I'd like to move forward with the program. First, I want to make this game into the full version of Magic: the Gathering including a deck building program to build decks from an already existing JSON database of Magic cards. Second, I want to learn more about Machine Learning and algorithms. What I have right now is an updating strategy, but I haven't come up with the best strategy. Lastly, at some point it would be neat to visualize the game for a human player so they can click and drag card objects.

With the code I've built so far, making the full game should be very doable. The hardest part will likely be converting the card's effects into code the program can run and possibly making a program that knows enough to convert the card text for me. For future machine learning algorithms, I also know how to implement them, but the challenge will be learning more about machine learning and finding the right algorithms to use. The last step of actually visualizing the game will probably take longer depending on what I aim to learn first, as I'd likely look at learning another language in order to do that. Nevertheless, there are lots of things I'd like to learn about, and it's definitely good to have a base program to build on as I learn things that could help.

Thank You

Lastly, I'd like to thank my mentor, Nikhil Sharma as well as Ray Berg and everyone who helped run the ChiPy Mentorship Program. This is the first program I've made and I learned a ton through the program. Every time I meat with my mentor he'd show me different ways of using Python that I wasn't previously familiar with. I now know how importing from different files works, how to read/write files with Python as well as about decorators and using polymorphism with class objects. All in all, I've learned a lot about Python and I had a great mentor helping me out the whole way. Thanks again for everything.

ChiPy Mentorship – Blog 3

December 16, 2016December 16, 2016 NickLeave a comment
Since my last post, I've finished a restructure of my original code and moved and then moved a lot of the game functionality into the new board.py (https://github.com/nickhattwick/PlayingWithCards/blob/master/board.py). Since the player object referred to the player character, it was getting a bit large having all of the games functionality in it as well. Now, the player object gets its own board object to reference, which runs all of the game mechanics, leaving the player object to focus more on the ways that the player interacts with it.

After more debugging, I got that working and then started to work on a logging file for the game. That's currently the project I'm working on. So far I've made decorators that will probably need some debugging, but will go into all of the different moves and record the type of move that was made into a list of the logging file. So far, that's all I have as this week and last have been really busy with final school assignments and I still have a couple papers left to write. I'm a little nervous that I wasn't able to do more, but after Sunday all of the school work will be done and will have a few weeks to work more than I could previously.

I've learned a lot so far, but still have a few things I'd like to do before presentation time. Step one is finishing up the logging system. I plan to do it by using the decorators to append to lists in the jsonlog.py (https://github.com/nickhattwick/PlayingWithCards/blob/master/jsonlog.py) file, and then at the end of the game use a function that will take the variables from those appended lists and make a JSON file of game data. The next step after that will be a file that takes the win results and move and makes a log showing each move and how many times its led to victory. This will create a way of keeping data from logs, while being able to delete specific game logs when there are too many. Finally, a new version of my computer players will reference this list and use those numbers to determine which cards lead to winning more often and then make the best move by those standards, after an initial data set of random moves.

I might start it off by only having the program use this for summoning and then branch it off to do more, as a way of starting small and building up. Though time is starting to run out, so I'm not sure how much I'll be able to do. I definitely want to get into some part of the Machine Learning. Additionally, learning JSON will be important for if/when I bring the full game of Magic into the model. There's already an existing JSON card library that I could draw from and are a lot more things I can do with this in the future. It's been a great experience so far and am looking forward to these next few weeks.

ChiPy Mentorship – Blog 2

November 17, 2016November 21, 2016 NickLeave a comment
While I had a working prototype of the game for my last blog post, the base of the code still had a lot of flaws that would make moving forward with the project impossible. In the original game code, the player is forced to be a human and always playing against a computer. The opponent's reaction code is directly intertwined with the code for the players move. If the player attacks, the block code is automatically resolved, and the prompts the player receives, the action the player makes, and the moves the opponent makes are so tied together that it was impossible to separate them without pretty much rewriting the code.

https://github.com/nickhattwick/PlayingWithCards (https://github.com/nickhattwick/PlayingWithCards)

The gamep3.py (https://github.com/nickhattwick/PlayingWithCards/blob/master/gamep3.py) file holds the original game code, while the rest of the folder holds the new game code. Now the rungame.py (https://github.com/nickhattwick/PlayingWithCards/blob/master/rungame.py) file simply sets up the characters and kicks off the sequence drawing from other files, instead of simply having everything run from one file. Once players are created and the first player is decided, rungame.py calls prompts.py (https://github.com/nickhattwick/PlayingWithCards/blob/master/prompts.py) which sets up the turn and conditions for it to continue or end, before passing priority to the player to make their move by calling for the player (https://github.com/nickhattwick/PlayingWithCards/blob/master/player.py)'s turn_prompt function. The three different types of players will each have their own unique function called by this function. A human player will be given text commands, the scripted AI will run a set strategy as before. The third type of player is going to be the one that implements the machine learning aspect by having a different way of choosing from options similar to the player.

There's a little bit, I still want to fix in the code, but it's ready to move on to the machine learning function. My mentor and I have started watching the Intro to Machine Learning (https://www.udacity.com/course/intro-to-machine-learning--ud120) course on Udacity and so far it doesn't seem to be a classification problem or an SVM one. The Decision Tree section seems promising and will hopefully end up being a model I can use for this.

Overall, I'm feeling a lot more confident about the code. Since it is much cleaner and separated into parts so that it can be edited easily without messing up other sections. For the next steps, we're continuing to watch through the Machine Learning course and during our last meeting, Nik showed me how to use decorators and write things into txt files so that the program has a way of keeping a log. Getting that working will definitely be a needed next step.

Other than that, it's time to make the third turn_prompt section and get moving on the machine learning. I've got a ton of learning to do and am a bit nervous, but things should work out. I'm really curious about what the next blog post will be like.

ChiPy Mentorship – Blog 1

October 18, 2016 *NickLeave a comment*
My project for ChiPy's Python Mentorship is an attempt to program a card game like Magic: the Gathering in Python and then to use machine learning techniques in order to have the computer solve for the best possible strategy. In high school and my first couple years of college, I played Magic: the Gathering to the point where I became sick of it. I was intrigued by the tournament aspect and trying to solve an incredibly complex game. While my desire to compete in tournaments or play has pretty much disappeared for me, I've always sort of regretted not being able to "solve" the game. In the last couple years, I've started learning to program and always thought it would be neat to reprogram the game in python. I've been inspired reading about programs like Deep Blue and the recent developments in Go using Google's DeepMind, and have had a desire for a while to learn more about data Science and machine Learning for awhile.

The Project So Far

The first part of the project is coming along smoothly. At present, I have made a simple version of the game where a player can play against a computer "AI". The basic idea of Magic: the Gathering is that both players summon creatures which they use to attack their opponents and take their life points from 20 down to 0. In order to summon creatures, you need to play land cards which roughly act as a clock so that as the game progresses, players can summon stronger creatures.

Presently, the rules for the game I have programmed are as follows.

1. Each player starts with a 40 card deck made up of lands and creatures, 20 life points, and an empty field, hand and discard pile.
2. Both players shuffle and draw 7 cards from their respective decks.
3. During each players turn they can play land and summon creatures whose costs total up to the number of lands the player has.
4. Players can attack the opponent with the creatures they summon, and the attacking player's opponent can block with one of their creatures. The creature with lower power will be destroyed (in case of a tie, both are destroyed).
5. Each creature can only block once per turn, and only if they didn't attack during their controller's last turn.
6. If a creature's attack is unblocked, the player it's attacking loses life equal to its power.
7. If a players life points hit zero, they lose.

After a lot of testing, I have a fully functioning game. The code here is a text based version of the game I described that runs in Python 3. Typing quit on your turn's prompt screen will quit the program.

https://github.com/nickhattwick/PlayingWithCards/blob/master/gamep3.py
(https://github.com/nickhattwick/PlayingWithCards/blob/master/gamep3.py)

(https://github.com/nickhattwick/PlayingWithCards/blob/master/gamep3.py)
(https://github.com/nickhattwick/PlayingWithCards/blob/master/gamep3.py)

Next Steps

From here there a few ways that I want to improve and continue my project. One thing I'd like to do eventually is make the game more complex and closer to Magic. Creatures in Magic have various aspects such as cost, power, toughness, and name, as well as special abilities unique to each creature. In my version, the first for variables are represented by a single number and there are no special effects that manipulate the game outside of the basic rules described above. The next step I want to do to improve the game's complexity is to make some spell cards which will do various things such as destroying a creature, drawing extra cards, or powering up a creature. I've started to work on that, though adding complexity to the game is probably not the next step in the project.

Before that, I'd like to start doing some basic machine learning. This is the part of the project I'm most nervous and excited about as it's something that I've been trying to learn about for awhile and one of the main topics I hoped to work on in this mentorship program. I've tried watching tutorials on machine learning and data science in the past, but realized that I was lacking the mathematical knowledge I needed to make use of it. Even if data science could show me shortcuts for different equations and algorithms, it wouldn't do a lot of good if I didn't understand the equations and what I was using them for.

One of the last things I was working on before the mentorship program started was going through statistics tutorials and starting to work on calculus with MIT's Open Course Ware and I feel prepared to tackle machine learning again. The project I've done so far is roughly the extend of my current knowledge of programming, so I'm diving into new territory one way or another. So, the next step is to set the stage to start testing out some machine learning algorithms, which means that I'll be reworking the code in my current game to be more versatile. At present, the game is made so that a player plays against computer that uses set moves. While I'm pleased with the game, the computer can't test out and see the effects of them.

My mentor suggested we use "controllers" and keep the basic game code the same. The game logic will be a program that will call out to a player controller or a computer controller for it's input. This will help to clean the program up as well as allow for computer versus computer matches which will be essential for having the computer get in a lot of games and test data. Currently, I'm working separating the code out of one file. For the future if the main game imports a deck from another file it will open up a lot more possibilities. For now, I'm restructuring the game I have to be ready for the next step. I am currently working on that in here.

https://github.com/nickhattwick/PlayingWithCards/tree/master/splittest (https://github.com/nickhattwick/PlayingWithCards/tree/master/splittest)

Overall, things are going well. I'm getting in to new territory, but learning is why I applied for this program. I owe a huge thanks to my mentor, Nik, for taking the time to help me on this project, showing me better way of doing things, being willing to learn these new topics with me, and teaching me a lot more about Python. I've already learned a lot from this program and look forward to learning.