Title: Card Game Project Writeup

Name: Gabriel Donnan

Number

Contents

[Analysis 3](#_Toc12863164)

[Identification 3](#_Toc12863165)

[Computational Approach 3](#_Toc12863166)

[Research 4](#_Toc12863167)

[Stakeholders 7](#_Toc12863168)

[Interviews 7](#_Toc12863169)

[Hardware Spec 8](#_Toc12863170)

[Software Spec 9](#_Toc12863171)

[Requirements 9](#_Toc12863172)

[Limitations 10](#_Toc12863173)

[Design 11](#_Toc12863174)

[Implementation 12](#_Toc12863175)

[Testing 13](#_Toc12863176)

[Evaluation 14](#_Toc12863177)

# Analysis

## Identification

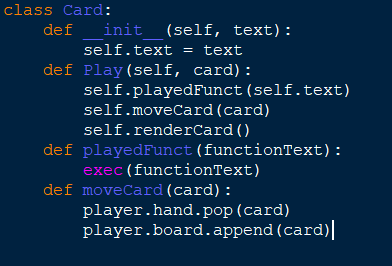
Currently I believe the market for digital card games is relatively bare, with only two main games dominating the market. I believe this leaves room for the creation of a new digital card game, using the simple combat mechanics employed by many card games in the past but using the digital medium to allow for more complicated effects and mechanics that are not amenable to the physical medium, specifically pseudo-random effects such as random generation. As such I believe that creating a game to fill this role would be beneficial to those who may believe the current game market to be stale as it could revitalize an interest in games of this type.

## Computational Approach

The problem of making this is, I believe, best solved by computational methods. This is for a variety of reasons, firstly the game is easily broken down into consecutive logical operations, which is exactly how computers operate, following instructions linearly. For example when a card is played this can easily be broken down into some function calls that can be made, firstly any effect the card might have on play must be enacted so a function called playedFunct() can be called to do this, then the card must be removed from the list of cards in your hand then added to the list of cards on the board, this can be done via a simple function call or even a single line statement which can pop and append the card (though one-liners like this can be disadvantageous when debugging as they can often seem more logically complex than they actually are). And finally the card has to be rendered on the board so a function renderCard(card,position) can be called, allowing the card to become visible on the board. Every facet of the gameplay can be broken down logically as such, as can the menus outside of the game.

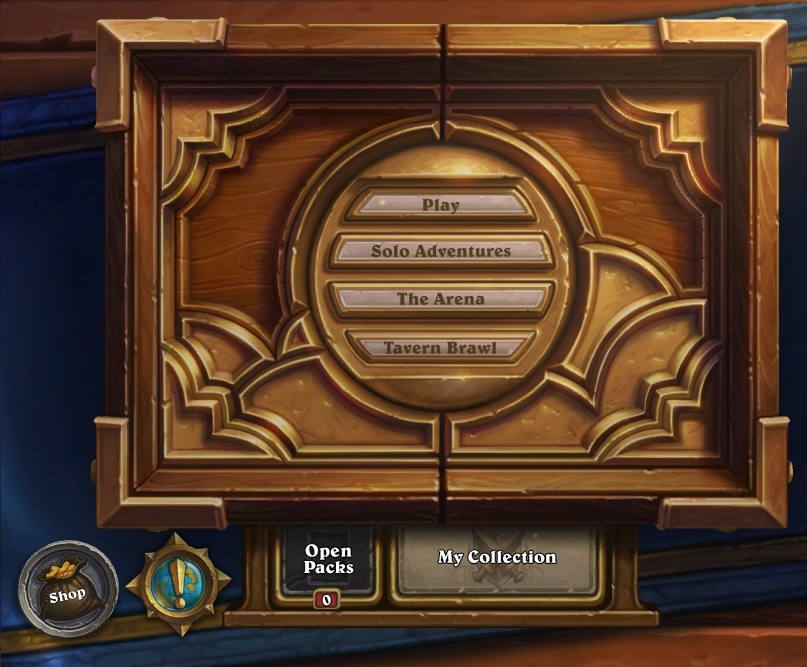
The reinforcement learning based AI which will be playing the game is not only best suited to computers as it can interact linearly with the game, it is impossible to make without them as you would need some other device capable of intelligent decision-making to be playing the game, which is limited to only humans and computers. Using a human here would defeat the purpose of having an AI opponent as the game is meant to be playable by yourself. The AI’s training makes use of iterative self-improvement (iteration being one of the most key computing concepts), performing mathematical and logical operations to achieve this at a speed not physically achievable by non-computational methods. The AI operates on a basic level by taking data from playing the game and optimizing itself so that it can create an output which will be the most likely to secure it the ability to win the game, this satisfies the computational approach as it is simply enacting a mathematical function on the input data to create an output

The game will be made in an object-oriented manner as I find this to be the best method to create the game whilst maintaining an intelligible, logical, code flow. Each card for each player can be instantiated as an object of the class “Card” which allows the cards to have their own unique properties (such as a name, cosmetic text, health, cost, attack, etc.) and were I not using an object oriented structure like this it would be much more difficult to logically allow the game to function. This process can be shown to be effective when using the example of playing a card again as using objects the cards themselves can have a .play() function, which is important as if it was necessary to use a global function the player number would need to be specified as each player has cards and the syntax within this function would be much more complicated, requiring many IF statements to determine which player is playing and what card they’re playing. Once the card played is determined when using objects any effects the cards may have on other cards can be easily enacted as they can directly adjust the properties of the other cards, but when not using an object oriented approach there are no individual card properties to edit, and as such everything would have to be kept in lists and adjusted this way, resulting in much more unreadable code, greater difficulty in adding more cards and inefficient code.

 *Below is a mock-up of the Card class structure needed to play a card:*

## Research

When looking into creating a game such as this I must first look at other similar games to see what inspiration I can derive from them but also more importantly what I can improve on comparatively to them. As part of my research I played and looked at the features of the two most popular online card games (Hearthstone and Magic: The Gathering, Arena) and the recently released DOTA Underlords, which is not a card game but uses an interesting unit buying system which I feel can help solve the problem of card drawing randomness which often negated some amount of the skill involved in gameplay for the first two games I looked into.

 Firstly I looked at Hearthstone, which is a game which is designed to look as smooth and user-friendly as possible, meaning it is an automatic attractor for those newer to the card game community or those who appreciate the polish put in to it.

*The Hearthstone main menu screen*

Hearthstone’s simple user interface is extremely appealing and as such I have decided to use a more simplified visual main menu structure rather than using text or any more complicated layouts which I feel needlessly confuse the user. The game has a pack opening system and an overarching card “collection” which indicates the cards which you do and do not own, and therefore the cards which you can and can not put into decks. I feel this is unfair to the end user and instead have opted to allow each user to have access to all the cards in the game.

The game’s concept is relatively simple, you draw 3 or 4 cards at the start of the game (dependant on whether you go first or second), you start at 1 mana crystal, gaining one per turn until you have 10. Mana crystals can be spent by playing cards and will refresh at the start of each player’s turn. There are two main card types, minions and spells. Minions are creatures which are summoned on to the board which can fight for you, either attacking your opponent or their minions. Spells are instant effects which do not necessarily summon creatures but can affect them or either player’s character. When the first player reaches 0 health the game ends.

I have decided to simplify cards into one type instead of using minions and spells. Cards will always summon a creature and can have effects similar to those of hearthstone’s spell cards attached to them, thus meaning the game is more easily understood. I have also opted to use a similar combat style, however unlike in hearthstone, you must always attack the creatures on the board before attacking the opposing player, allowing the game to last for longer and for creature on creature combat to be a larger facet of the game.



*A standard board state in Magic: The Gathering, Arena*

The second game I researched was Magic: The Gathering, Arena. Magic: The Gathering is one of the longest standing card games ever made and its advent into the market of digital card games has given me an opportunity to see what works and what doesn’t comparatively to Hearthstone.

The game functions slightly differently to hearthstone in that instead of mana crystals you gain as the turns advance you have the ability to play “land” cards which allow you to play other cards by spending them each turn. It is also different in the fact that your opponent gets to dictate the path on which your attacking creatures attack, meaning the game is also reliant on bluffs and not knowing your opponent’s decision-making.

I believe that whilst this does promote the skillful aspect of the game that my stakeholders want, bluffing and strategies such as this are almost impossible against an AI opponent as it does not make decisions in the exact same way humans do and as such I am allowing the player full control of their creatures’ attacks. In Magic: The Gathering damage inflicted to creatures is not permanent due to the game originating in physical form so it was nearly impossible to keep track of damage inflicted to so many things. Since my game will be digital the damage inflicted to creatures will last between rounds as computers can keep track of this very easily.

 One feature of Magic: The Gathering that I wish to implement if I have time is the ability to be able to stack like units on top of one another to remove clutter on the board and allow the game to look more visually appealing, which I believe is a key factor in player accruement.

*A standard board state in DOTA: Underlords, with purchased units displayed at the bottom*

The final game at which I looked for my research was DOTA: Underlords, a recently released game based on the popular mod DOTA: Autochess. This was the only game I researched that was not a card game, but rather just a strategy game in a more broad sense to see if I could incorporate any features currently unused by digital card games but applicable to strategy games in general to allow mine to stand out.

One particular feature that stood out to me while researching this game is the unit purchasing system in which each round you are offered a selection of units which you can buy with gold you collect over the course of the game, when applied to a card game instead of drawing a card each turn I feel that this would be extremely beneficial to the skill requirement of the game and thus the long term enjoyment levels that can be achieved. Adding this feature also removes the need for deckbuilding, which becomes simply a game of statistics instead of an interesting facet of the game with which to play due to the easily analysable gameplay of card games.

## Stakeholders

When looking into stakeholders for my game a number of my peers expressed interest in such a game given certain features they wanted were implemented. One particular interested party is a friend of mine named Vishaan who has a background in games that involve longer term strategy so as such a card game was a natural attractor for him. The game perfectly suits his desire for strategizing as there are nigh on infinite possible iterations of the game and its state so a good understanding of strategic thinking is rewarded.

He also enjoys games with a high degree of customization as evidenced by his investment in RPGs with character customization in the past, so the deck customization that is possible appeals to him as well. He hopes that in the way the game is coded it will be easy to add new cards to allow the game to feel fresher as new cards can rejuvenate a game to ensure it’s long term enjoyability. He also prefers games in which the opponents feel more real and as such I have decided to use reinforcement learning AI as opposed to the standard model of using strings of conditional statements to make the decisions the opponent makes less rigid.

Another potential interested party is my younger brother Micah as he has enjoyed digital card games in the past but feels like the current available market for them is too bare for his liking, as such he wants a game with different gameplay to the other two major online card games of Hearthstone and Magic: The Gathering, Arena. In order to achieve high enough difference from these games he feels that the game I am making should have mechanics with more choice to replace the classic method of drawing cards implemented by other games which can lead to a high amount of luck being involved in gameplay as opposed to pure skill so as such I have decided to instead opt for a turn based card buying system instead which, whilst it still has some variance, rewards skill over luck a greater proportion of the time. He has also requested that the game have a screen that will scale to any screen size automatically so it does not take any additional effort to use it on different devices and as such the game will be able to be enjoyed by a wider range of device owners who can now use any device without the game looking any different between them.

The third stakeholder I have identified is another friend of mine who very much enjoys gaming but is not currently very involved in card games or strategy games in general and as a result of this lack of current investment into games of this type he has said he would enjoy it if the game were to have an explanation of the rules included within it to make his learning process easier. In order to fulfill this request I have decided that I will add a basics tutorial section to the game accessible via the main game menu so that the game can be picked up by new players with greater ease.

### Interviews

I conducted short interviews with my stakeholders in order to identify what exactly they wanted to see from the finished product of the game:

My first interviewee was Vishaan, below is a record of the interview that was conducted:

Me: What, for you, makes a game enjoyable?

Vishaan: Well, in my opinion as long as the game is of a type that I like it just has to not be completely reliant on luck but have enough variation so it doesn’t feel stale

Me: How do you think one could tackle the problem of luck versus variance in a card game?

Vishaan: Im not 100% sure, but if I had to guess I would say I would want not to have to build a deck of cards so the games I play don’t always feel the same for each deck

Me: Alright, and what do you feel, if anything, that current card games are doing wrong that I could improve on?

Vishaan: Yeah there is one thing that annoys me about card games right now, most games do this actually and its really offputting. Its that their AIs for enemies don’t feel like they’re actually a challenge and I think if they seemed more human then I could look at them as more of a valid opponent

## Hardware Spec

The game will not be extremely hardware intensive as all the AI’s training will be done on my computer beforehand so the intense training process is not necessary to carry out on the end user’s machine. If someone were to attempt re-training it or training an entirely new neural network to use instead of mine (this can be done relatively easily and would normally be done to adjust the difficulty level of the AI opponent as less training would mean more difficulty and vice versa up to a point) I would recommend a system using:

* Dedicated GPU with no less than 4GB of VRAM
  + GB+ of VRAM would be recommended but can be very expensive
  + A more powerful GPU is more impactful on training performance than a more powerful CPU
* CPU with at least 4 cores with a clock speed of 3GHz
  + Improving the CPU will increase the performance of training with more cores being more impactful than higher clock speed but an upgrade of CPU will almost always be worse than an upgrade of GPU

These hardware specs are important in the event of a retraining in order to attempt to train it in a practical amount of time (i.e. less than a week). The only real requirement when you are not retraining is that the hardware is able to support all the necessary software at a basic level for the computer to run it which, according to Intel (<https://software.intel.com/en-us/distribution-for-python/system-requirements>), Is at least one gigabyte of secondary storage (SSD or HDD are both acceptable as the read/write speed will not affect the performance very much) for the base python and an additional 2 to 3 gigabytes for the rest of the libraries that need to be imported such as pytorch.

The system should also have:

* A processor at least as powerful as a single core 1.6GHz processor
  + This is normally the power level of the lowest price point ARM or Intel Atom processors
* Standard peripheral devices required for computers to function normally
  + At least one monitor on which to see the gameplay
  + A keyboard (even 80% size keyboards still work)
  + A mouse

## Software Spec

There are relatively few pieces of required software for the game:

* A python version of 3.0 or more recent
  1. Syntax changes heavily in some common statements like print or input from 2.7 to 3.0 and as such the code would not be able to run on any older versions of python
* The most recent versions of the python libraries I use
  1. Pygame
  2. Pytorch
  3. Random
  4. Math
* An operating system which supports all the libraries I have listed (i.e. the standard operating systems of Windows, most Linux distributions or Mac OS)

## Requirements

The game, upon completion, should have:

* Necessary Features for the game to function:
  1. Card buying system
  2. At least 10 different card types
  3. Cards which trigger effects on:
     + Death
     + Play
     + Attack
     + End of turn
  4. Ability to play cards from your hand onto the board
  5. A limit on the space on your board and in your hand
  6. Tutorial section
  7. Ability to attack using your played cards to damage the opposing player
  8. AI implementation of the second player
  9. Ability to win or lose the game given certain criteria have been fulfilled
  10. Ability to combine cards to make them more powerful
* Important features the game should have but are not necessary for it to function:
  1. A menu with which to navigate, at the very least containing the ability to:
     + Play a game
     + Adjust your settings
     + View the tutorial
  2. Ability to adjust the position of cards on your board
  3. Ability to combine multiple of the same card into a single more powerful version of themselves
  4. Ability to pause the game
  5. Menu of all the cards and their combinations
     + Given enough time this could also be an interactable visual menu, though this is more of a stretch feature
* Features that would be nice to have but less likely to be finished:
  1. The ability to retrain the AI from within the game menu
  2. Different game modes to the standard mode of play
  3. Local multiplayer versus another human player
  4. In-game custom card creation feature
  5. Ability to save the game state and return to it later
  6. Animations for things like playing cards
  7. Passive effects for played cards
  8. Stacking like units

## Limitations

Whilst the game will be feature-rich and strategically in-depth as I can make it, there will of course be limitations to the things that I will have been able to complete or add to the game within the time frame I have to make it.

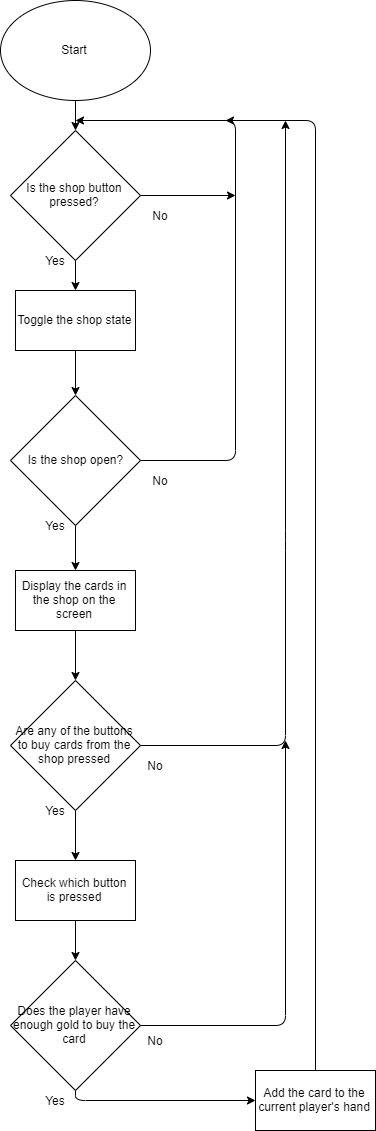
Such limitations include:

* No sound
* No custom animations for different effects
* No online or LAN multiplayer
  1. I might possibly implement local multiplayer, however online or LAN multiplayer is much too complicated to do within such a time frame
* Only English language support

# Design

## Design of required features.

### 1)Card Buying System

This is a system to allow the current player to buy cards from the shop via clicking buttons associated with them in order to be played later.

If mouse.collide(button) and mouse.clicked() then

Shop.open = True // toggles Boolean value

If Shop.open == True then

Cards.display() //Displays cards in the shop

Cards.displayButtons() // Displays the buttons associated with the cards in the shop

For button in Cards.buttons:

If button.get\_pressed() == True and PlayerCurrency > button.card.currency then

Player.buyCard(button.card) // Buys the card which the button is linked to

PlayerCurrency -= button.card.cost // Deducts the cost of the card from the players gold count

Else

DisplayError(“You don’t have enough currency to buy that”)

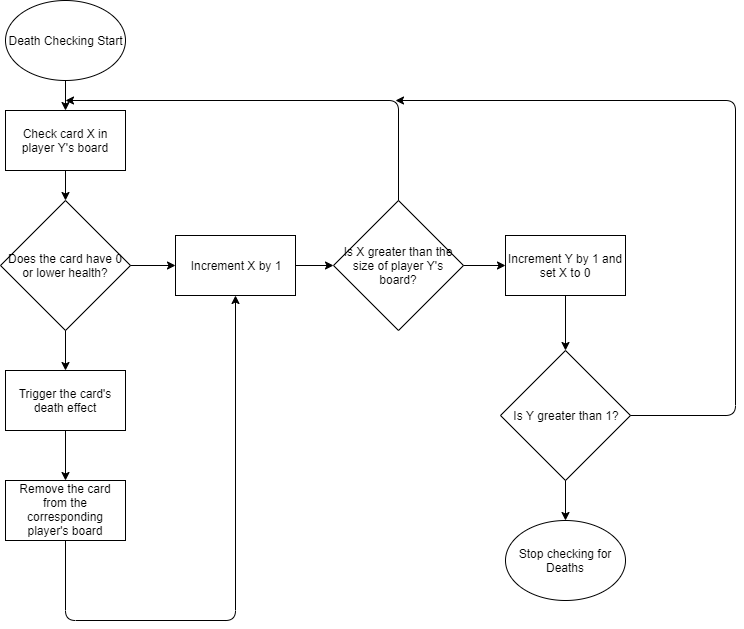
##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Clicking shop button | Check that button functions as required to toggle the shop | The shop goes from not being displayed to being displayed when the button is pressed |
| 2 | Clicking on the card itself when shop is open | Check that clicking on the card in the shop does not buy it, play it if an instance is in your hand or any other such possible bugs | Clicking on the card itself should not do anything, as the buttons associated with the cards are what are used for buying them |
| 3 | Clicking on the buy buttons associated with the cards when you have enough currency to buy them | Check that when you have enough currency to buy the card it can be added to your hand and removed from the shop | The card is removed from the shop and added to your hand, deducting the amount of gold it costs from your balance |
| 4 | Clicking on the buy buttons associated with the cards when you do not have enough currency to buy them | Check that when you do not have enough currency to buy the card it does not get purchased from the shop and an error message is displayed | The card remains in the shop and a copy of it is not added to your hand, an error message is displayed telling you that you do not have enough gold to purchase that card right now |
| 5 | Clicking in the locations in which the buy buttons would be located when the shop is not being displayed | Check that the buttons cannot be interacted with if the shop is invisible | Nothing will happen, the cards in the shop at that location will not be bought |

### 3) Cards which trigger effects at different intervals

#### 1) On Death

This simultaneously checks whether destroyed cards need to be removed from the board and, if so, triggers their effect by calling a class function that all cards have but is changed via polymorphism to the specific card.



For card X in playerBoard[Y]:

If X.health < 1: // Checks if the card is dead

X.onDeath() // Triggers the card’s death effect

playerBoard[Y].remove(X)

if index(X) > length(playerBoard[Y]) // Checks if it is checking outside the range of the current player’s board

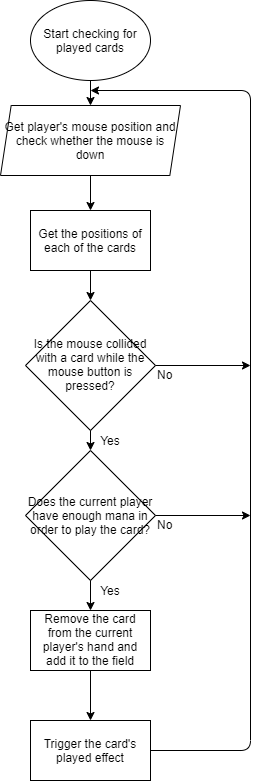
Y++

X = 0

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Checking an empty board | To check if the program tries to remove cards despite lack of any or if there’s an index range error where the max index is 0 | No error will occur, i.e. no cards will erroneously try to be removed and the indices will not exceed their limits |
| 2 | Checking a full board of living cards | To test the edge case where both players’ boards are full, i.e. that no index range error occurs and no cards are removed | No cards are removed as they all are alive (i.e. have health values greater than 0)  No index out of range error occurs. |
| 3 | Checking a full board of dead cards | To test whether the program can effectively remove multiple cards at a time from multiple boards | All cards are removed from both players’ boards |
| 4 | Checking a board with some living and some dead cards | To test whether the program can differentiate between living and dead cards interchangeably | The dead cards are removed from the correct players’ boards and the living cards are ignored |
| 5 | Checking a board with some living and some dead cards of the same name | To test whether the program can differentiate between living and dead instances of the same card and does not remove the living ones erroneously | The living instances of the cards remain untouched and the dead ones are removed, the death effects are triggered only for the cards that died |
| 6 | Checking a board where one player has a full board of cards and the other has an empty board | To check whether the maximum indices of the board length can change in between players | No index out of range error occurs, dead cards are removed and living ones are ignored |
| 7 | Checking a board with intermediate numbers of cards on both players side with some living and some dead | To make sure the program functions outside of purely edge cases | All dead cards are removed and all living cards are ignored |

#### 2)On Play

This checks whether the current player is able to play their card and, if so, triggers the On Play effect unique to that card



For card in playerHand[currentPlayer]:

Position = card.getPosition() // Gets the position of each card

If mouse.touching(Position) and playerMoney > card.cost and mouse.isDown() == True: // Checks if cursor is touching the card and whether the mouse is down. Then subsequently whether the player can afford the card

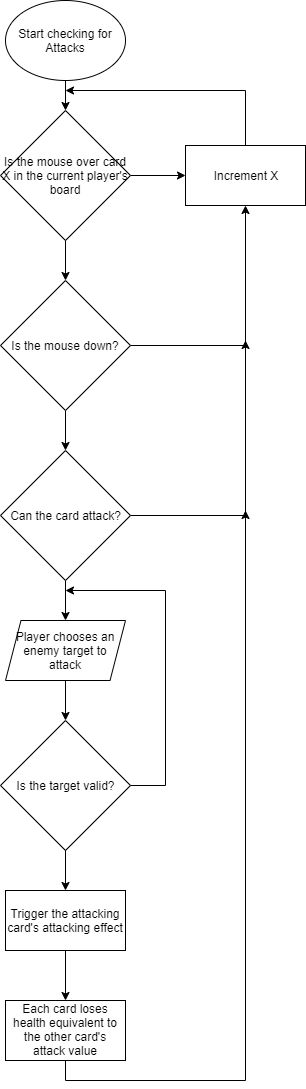
card.play()

card.playedEffect()

##### Test Plan

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Attempting to play a card with insufficient mana | Make sure that the mana cost of the card is enforced | The card will not be played and no mana will be deducted. |
| 2 | Playing a card with sufficient mana | Make sure that the correct number of mana is deducted and the card enters the battlefield while triggering its played effect | The corresponding amount of mana to the card’s cost is deducted from the current player, the card is moved from the current player’s hand to the battlefield and its played effect is triggered. |
| 3 | Playing a card with a played effect that affects the board when other cards are populating the board | Make sure the played effect interacts as intended with the other cards on the board | The played effect will activate on all of the cards on the board as intended (i.e. one that deals damage to all cards on the board will deduct X amount of health from all the other cards once) |
| 4 | Playing a card with a played effect that affects the board when no other cards are populating the board | Make sure that the played effect does not break when there is nothing to interact with | The played effect will change nothing (i.e. one that deals damage to all other cards will not change anything or crash the program as there are no other cards to deal damage to) |
| 5 | Attempting to play a card when the current player’s board is full | Make sure nothing changed when this is attempted | The card will remain in the current player’s hand, will not trigger its played effect and will not end up on the current player’s board. |

#### 3)On Attack

Checks whether a card can attack and whether its target is valid and if it can it triggers its attacking effect.

For card X in playerBoard[currentPlayer]:

If mouse touching X and X can attack and mouse down: // Checks if the player is attempting to attack with the card

Attacker = X

For card Y in playerBoard[opposingPlayer]:

If mouse touching Y and Y.isAlive() and mouse down: // Checks whether the targeted card is valid to attack

Attacker.attackEffect() // Triggers the attacking effect of the attacking card

Player.attack(Attacker,Y) //Deducts the attack of each card from each other’s healths and inverts X’s canAttack value to false

|  |  |  |  |
| --- | --- | --- | --- |
| Number | Item | Purpose | Expected Result |
| 1 | Attempt to attack with a card which cannot attack | Make sure that the condition that the card be able to attack for it to be able to trigger its attacking effect be enforced | The card does not attack any other card, the attack effect does not trigger and no health is deducted from either card |
| 2 | Attempt to attack a friendly card with a card that can attack | Make sure the card cannot attack friendly cards | The card does not attack the friendly card, the attack effect does not trigger and no health is deducted from either card |
| 3 |  |  |  |
| 4 |  |  |  |

# Implementation

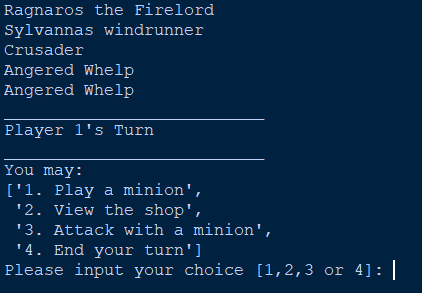
When starting the implementation I initially made the game text based in order to test out functionality instead of jumping straight into a pygame based iteration.

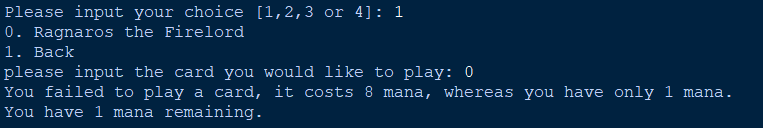
## Game Loop

### First Iteration

1. **while** **not** done:
2. **print**("\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\nPlayer " + str(player.currentPlayer) + "'s Turn\n\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\nYou may:")
3. pprint.pprint(["1. Play a minion","2. View the shop", "3. Attack with a minion", "4. End your turn"])
4. choice = 0
5. **while** choice **not** **in** [1,2,3,4]:
6. choice = int(input("Please input your choice [1,2,3 or 4]: "))
7. **if** choice == 1:
8. counter = 0
9. **for** i **in** player.playerHand[player.currentPlayer-1]:
10. **print**(str(counter) + ". " + i.name)
11. counter += 1
12. **print**(str(counter) + ". Back")
13. playChoice = 999
14. **while** playChoice **not** **in** range(0,len(player.playerHand[player.currentPlayer-1])+1):
15. **try**:
16. playChoice = int(input("please input the card you would like to play: "))
17. **except**:
18. **print**("invalid choice")
19. #print(len(player.playerHand[player.currentPlayer-1]))
20. **if** playChoice != len(player.playerHand[player.currentPlayer-1]):
21. player.play(player.playerMana[player.currentPlayer-1],playChoice)

In the first iteration I created a basic game loop with 4 options, I used a heavily class based implementation, with variables and functions in the player class allowing cards to be played. The variable player.playerHand is a list which stores all the cards that are currently in each player’s hand which I used to choose a card to be played using the player.play() function. If the player chose “1” as their option the names of each card in the player’s hand would be displayed alongside an index and the player would be allowed to choose one of the cards displayed to attempt to play (or choose to go back and not play anything). Once the user selects a valid card to play the game will call the player.play() function which will verify whether the player has enough mana to play the card needed and if so then it will be moved from the player’s hand to the board.





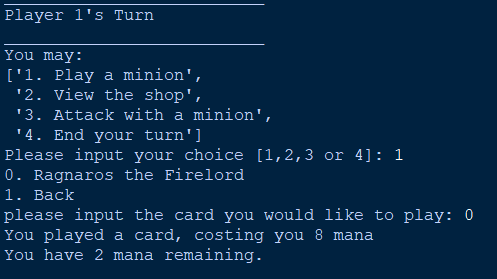
Upon testing out the code for playing cards like this I quickly realized that while in theory it should work correctly I had no way of gaining enough mana to play the cards I needed to play as my endTurn function in the player class had not yet had mana gain implemented.

1. **def** endTurn(self):
2. #Executes the end of turn functions of the cards in play
3. **for** card **in** playerBoard[0]:
4. card.executeFunction(card.endFunc, 1)
5. **for** card **in** playerBoard[1]:
6. card.executeFunction(card.endFunc, 2)
7. #Changes the current player
8. self.playerCurrency[0] += 3
9. self.playerCurrency[1] += 3
10. self.currentPlayer = (self.currentPlayer % 2)+1
11. self.genCards(5)

this was easily changed by simply adding in the lines

1. self.playerMana[0] += 1
2. self.playerMana[1] += 1

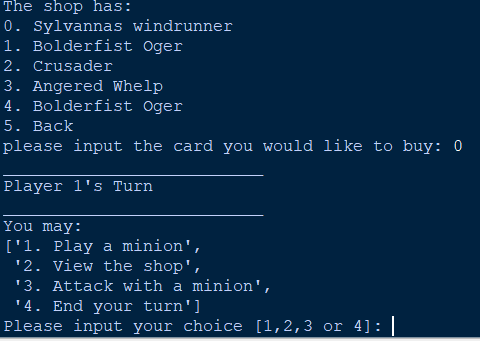
which, while not the desired end result for my mana gain mechanic allowed me to test the playing function thoroughly and confirm that it did in fact function correctly.



For the second section of the main game loop used very similar methods to the first, looping through each card in the shop to display them next to an index from and allowing you to choose which one you would like to purchase by typing in the index of the selection you would like. The try: except: statement in the validation is to make sure the program will not crash if the user inputs a string as it tries to convert the user’s input into an int, which cannot happen if it is not a number.

1. **elif** choice == 2:
2. counter = 0
3. **print**("The shop has:")
4. **for** i **in** player.forSale:
5. **print**(str(counter)+".",i.name)
6. counter += 1
7. **print**(str(counter) + ". Back")
8. purchaseChoice = 255
9. **while** purchaseChoice **not** **in** range(0,len(player.forSale)+1):
10. **try**:
11. purchaseChoice = int(input("please input the card you would like to buy: "))
12. **except**:
13. **print**("invalid choice")
14. **if** purchaseChoice != len(player.forSale):
15. player.buyCard(purchaseChoice)

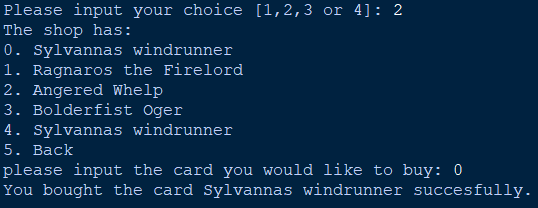
This called a function player.buyCard which validated that the player had sufficient currency to buy the card rather than this validation occurring within the game loop in order to improve readability.



There was, however, an issue. Even though the card was bought there was no indication that anything at all had happened and so I updated the buyCard() function in the player class to include a method of showing what happened.

1. **def** buyCard(self, cardPos):
2. **if** self.playerCurrency[self.currentPlayer-1] >= self.forSale[cardPos].shopCost:
3. #Adds the purchased card to the purchasinc player's hand and removes it from the shop if they have enough currency to buy it
4. self.playerHand[self.currentPlayer-1].append(self.forSale.pop(cardPos))
5. **print**("You bought the card " + self.playerHand[player.currentPlayer-1][len(self.playerHand[player.currentPlayer-1])-1].name + " succesfully.")
6. **else**:
7. **print**("Oops, looks like you dont have enough gold to purchase that card right now!")

The added print statement displaying the name of the latest card in the player’s hand successfully managed to function in the way I intended, displaying the card purchased by the player.



This addition, while not important to the fully pygame-based iteration later was extremely useful in debugging and the function of the first iteration as a useable product.

The third section of the initial game loop is slightly more complicated than the first two

1. **elif** choice == 3:
2. counter = 0
3. **print**("You have " + str(len(player.playerBoard[player.currentPlayer-1])) + " minions, of which:")
4. **for** i **in** player.playerBoard[player.currentPlayer-1]:
5. **if** i.canAttack:
6. **print**(i.name)
7. **print**("Can attack")
8. **for** i **in** player.playerBoard[player.currentPlayer-1]:
9. **print**(str(counter) + ". " + i.name)
10. counter += 1
11. **print**(str(counter) + ". Back")
12. attackChoice = 255
13. **while** attackChoice **not** **in** range(0,len(player.playerBoard[player.currentPlayer-1])+1):
14. **try**:
15. attackChoice = int(input("please input the card you would like to attack with: "))
16. **except**:
17. **print**("invalid choice")
18. enemyChoice = 255
19. **if** player.currentPlayer == 2:
20. playerSwap = 1
21. **else**:
22. playerSwap = 2
23. counter = 0
24. **for** i **in** player.playerBoard[playerSwap-1]:
25. **print**(str(counter) + ". " + i.name)
26. **while** enemyChoice **not** **in** range(0,len(player.playerBoard[playerSwap-1])+1):
27. **try**:
28. enemyChoice = int(input("please input the card you would like to attack: "))
29. **except**:
30. **print**("invalid choice")
31. **if** attackChoice != len(player.playerBoard[player.currentPlayer-1]):
32. player.attack(player.playerBoard[player.currentPlayer-1][attackChoice],player.playerBoard[playerSwap-1][enemyChoice])

This time almost all of the validation was done before the attack function was called as in this text based version higher numbered choices did different things, such as being the back button instead of a valid attack target. This meant that much more validation was needed and as two separate inputs were required, one for the attacking card and one for the card to be attacked, this simply cannot be done later in an efficient manner.

Firstly it loops through all cards for which “canAttack” is True in the current player’s board, displaying the name of each, the player then chooses which of the cards they want to attack with (or alternatively they can select “back” to cancel). When a valid selection has been made here the names of all possible enemy targets are printed and the player can then choose which of them they want to attack. Once this has been selected the player.attack() function is called to make the first selected card attack the second.

[\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Add](\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\Add) testing here\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\

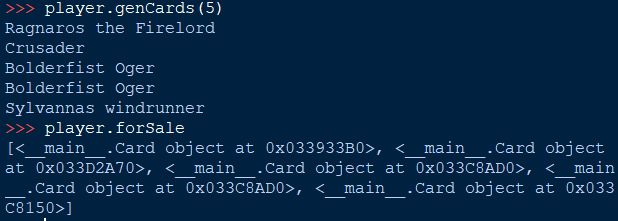
### Player Class

1. **class** Player:
2. playerHealth = [None,None]
3. playerHand = [None,None]
4. playerBoard = [None,None]
5. playerMana = [None,None]
6. playerCurrency = [None,None]
7. **def** \_\_init\_\_(self, cardList):
8. self.playerHealth[0] = 25
9. self.playerHealth[1] = 25
10. self.playerHand[0] = []
11. self.playerBoard[0] = []
12. self.playerBoard[1] = []
13. self.playerHand[1] = []
14. self.playerMana[0] = 10
15. self.playerMana[1] = 10
16. self.globalCardList = cardList
17. self.currentPlayer = 1
18. self.forSale = []
19. self.playerCurrency[0] = 20
20. self.playerCurrency[1] = 20

The player class is what manages most of the gameplay and as such is used many times in the game loop. It is initialized with a health value for each player, what is in each player’s hand and board, each player’s starting mana value and currency value alongside other things. The list “cardList” contained declared versions of each card object so they could be more easily generated by the genCards() function

1. **def** genCards(self, amount):
2. displaylist = []
3. #randomly generates (amount) cards from the globalCardList (list of all possible cards) and prints their name, will display them for purchase later and replace card drawing
4. **for** i **in** range(amount):
5. displaylist.append(random.choice(self.globalCardList))
6. self.forSale = displaylist
7. **for** i **in** self.forSale:
8. **print**(i.name)

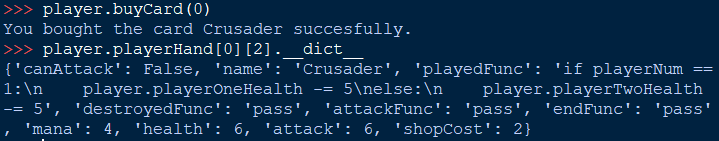
This function randomly chooses (amount) times from the globalCardList class variable and appends them to a list “displayList” which replaces the items in the shop and then prints each item in it once completed.



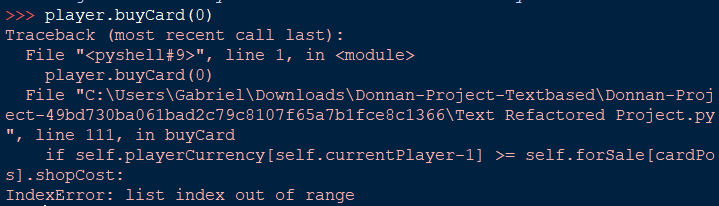
The items in the shop are Card objects as this makes moving them from the shop to your hand and from the hand to the board while acting as they should (i.e. calling played functions) extremely easy.

1. **def** buyCard(self, cardPos):
2. **if** self.playerCurrency[self.currentPlayer-1] >= self.forSale[cardPos].shopCost:
3. #Adds the purchased card to the purchasinc player's hand and removes it from the shop if they have enough currency to buy it
4. self.playerHand[self.currentPlayer-1].append(self.forSale.pop(cardPos))
5. **print**("You bought the card " + self.playerHand[player.currentPlayer-1][len(self.playerHand[player.currentPlayer-1])-1].name + " succesfully.")
6. **else**:
7. **print**("Oops, looks like you dont have enough gold to purchase that card right now!")

The buyCard() function follows on logically from the genCards() function, taking in a card position of the item you want to buy in the shop and checking if you have sufficient currency to buy it, if you do then the card is moved from the player.forSale[] list to the player.playerHand[] list for the current player, printing a message depending on whether you are able to successfully do it.



However upon testing I found that if you tried to buy a card larger than the maximum index of the hand an exception would occur (this most often happened when trying to buy the first card after everything had already been bought)



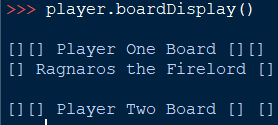
As such I edited the code to include the statement

**if** player.forSale[cardPos]:

at the beginning of the function to check whether the index exists in the forSale list to prevent crashes such as these

1. **def** boardDisplay(self):
2. #Update this with pygame stuff later, simple visualiser for logic for now
3. **print**("\n[][] Player One Board [][]")
4. **for** i **in** self.playerBoard[0]:
5. **print**("[] " + i.name + " []")
6. **print**("\n[][] Player Two Board [] []")
7. **for** i **in** self.playerBoard[1]:
8. **print**("[] " + i.name + " []")

The player.boardDisplay() function was relatively simple and was called after every change in the board state, displaying the names of the cards on each player’s board. It takes no inputs as it’s function is always the same



The attack function is significantly more complicated

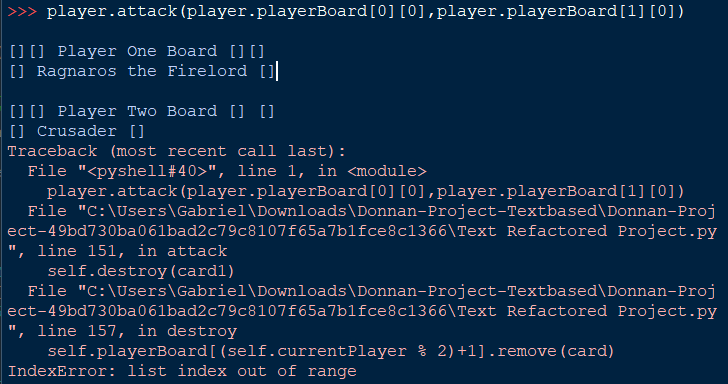
1. **def** attack(self, card1, card2):
2. self.boardDisplay()
3. card1.executeFunction(card1.attackFunc, self.currentPlayer)
4. card2.health -= card1.attack
5. card1.health -= card2.attack
6. **if** self.currentPlayer == 1:
7. **if** card1.health <= 0:
8. self.destroy(card1, 1)
9. **if** card1.health <= 0:
10. self.destroy(card2, 2)
11. **else**:
12. **if** card1.health <= 0:
13. self.destroy(card1, 2)
14. **if** card1.health <= 0:
15. self.destroy(card2, 1)

It takes in the arguments “card1” and “card2”, where “card1” is the attacker and “card2” is defending. For the attacking card it calls the card.executeFunction() function on its attack function, effectively calling a function specific to each card that occurs when they attack affecting the board in different ways depending on their attackFunc variable.

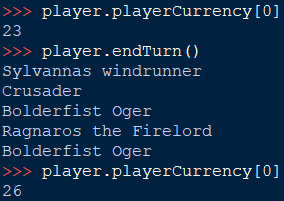
The health of each card is reduced by the attack of the other card and each card’s health is checked after this to call player.destroy() on them if they have less than 1 health.

1. **def** destroy(self,card, player):
2. card.executeFunction(card.destroyedFunc, player)
3. **if** player == 1:
4. (self.playerBoard[0]).remove(card)
5. **else**:
6. (self.playerBoard[1]).remove(card)

Player.destroy() takes a card and player input to remove the card from a specified player’s board and calls card.executeFunction on the card’s destroyedFunc to perform a specific action to each card that happens when they die. The player input is taken so that the program knows which player’s board to remove the card from, in an earlier implementation I did not have this and as such it crashed whenever a friendly card was destroyed on your turn.



The player.endTurn() function is called at the end of each player’s turn to enact the changeover from one player to the other

1. **def** endTurn(self):
2. #Executes the end of turn functions of the cards in play
3. **for** card **in** self.playerBoard[0]:
4. card.executeFunction(card.endFunc, 1)
5. **for** card **in** self.playerBoard[1]:
6. card.executeFunction(card.endFunc, 2)
7. #Changes the current player
8. self.playerCurrency[0] += 3
9. self.playerCurrency[1] += 3
10. self.currentPlayer = (self.currentPlayer % 2)+1
11.     self.genCards(5)

This is done relatively simply by performing the modulus 2 on the self.currentPlayer variable and incrementing by one. It also triggers the endFunc for each card in each player’s board and increases the currency of each player by 3 and calls player.genCards() so that new cards may be bought each round. The function requires no inputs as it is the same in all situations in its functionality.

The final function in the first implementation of the player class was player.play()

1. **def** play(self, mana, cardPos):
2. card\_played = self.playerHand[self.currentPlayer-1][cardPos]
3. #Checks if the card you're trying to play costs too much
4. **if** card\_played.mana <= mana:
5. self.playerBoard[self.currentPlayer-1].append((self.playerHand[self.currentPlayer-1]).pop(cardPos))
6. card\_played.executeFunction(card\_played.playedFunc,1)
7. mana -= card\_played.mana
8. **print**("You played a card, costing you " + str(card\_played.mana) + " mana")
9. **else**:
10. **print**("You failed to play a card, it costs " + str(card\_played.mana) + " mana, whereas you have only " + str(mana) + " mana.")
11. **print**("You have " + str(mana) + " mana remaining.")
12. **return** mana

This takes a mana and cardPos value as inputs and returns a mana value at the end after the card has either been played or the player has failed to play the card. The function uses the cardPos variable to find the position of the card the player wants to play within their hand, this makes playing cards easier when using text inputs as the user can simply input an index to play.

The function then checks the mana input against the mana cost of the card, if the mana input is higher than or equal to the cost of the card the card is popped from the player’s hand and appended to their board, triggering their playedFunc and producing a text output.

### Card class

#### First iteration

1. **class** Card:
2. **def** \_\_init\_\_(self,shopCost,name, mana, attack, health, playedFunc = "pass", destroyedFunc = "pass", attackFunc = "pass", endFunc = "pass"):
3. #for all the func variables the input is a block of text which is passed into generic functions containing only an exec block, this saves me from having to write hundreds of new functions and allows for creations of new cards extremely quickly
4. #The function text defaults to a function that does nothing
5. self.canAttack = False
6. self.name = name
7. self.playedFunc = playedFunc
8. self.destroyedFunc = destroyedFunc
9. self.attackFunc = attackFunc
10. self.endFunc = endFunc
11. self.mana = mana
12. self.health = health
13. self.attack = attack
14. self.shopCost = shopCost

The Card class is for creating objects out of each card so they can be played and interacted with meaningfully with outside of simply being variables. They are initially defined relatively simply, taking the inputs shopCost, name, mana, attack, health, playedFunc, destroyedFunc, attackFunc and endFunc and assigning each to a variable for the object. Each variable ending in Func was made so that no subclasses would have to be created for the cards, instead allowing specific functions to be passed in as text which was passed to an exec() block whenever they need to be called. Each of these was defaulted to “pass” as if they were called without a value the program would crash as it would try and execute an empty variable.

The function to execute the “func” variables is simply one line

1. **def** executeFunction(self,text, playerNum):
2. **exec**(text)

It takes in a variable “playerNum” alongside the text to be executed. This function may appear to do nothing but it actually is used in the cards’ “func” variables when run to determine what side of the board they are on.

The cards were declared as such

1. cards.append(Card(4,"Ragnaros the Firelord" ,8,2,8,playedFunc = """
2. for i in player.playerBoard[player.currentPlayer-1]:
3. i.health -= 8"""))
4. cards.append(Card(3,"Sylvannas windrunner",6,5,5, destroyedFunc = """if playerNum == 1 and player.playerBoard[1]:
5. (player.playerBoard[0]).append((player.playerBoard[1]).pop(random.randint(0,len(player.playerBoard[1])-1)))
6. elif player.playerBoard[0] and playerNum == 2:
7. player.playerBoard[1].append(random.choice(player.playerBoard[0]).pop)"""))

With the “func” variables written as if they were code within multi line strings so that they can be executed. This unfortunately made debugging much harder as if a bug were discovered within one of these variables the error text from the traceback would not be as informative as usual and as such during testing this proved to be a less efficient method of going about the card class.

#### Second Iteration

In the second iteration the cards are implemented each as their own subclass of the overarching “CardBase” class and as such the initialization statements are much simpler

1. **class** CardBase:
2. **def** \_\_init\_\_(self,shopCost,name, mana, attack, health):
3. self.canAttack = False
4. self.name = name
5. self.mana = mana
6. self.health = health
7. self.attack = attack
8. self.shopCost = shopCost

This implementation eliminated all the “func” variables, meaning that significantly fewer variables had to be initialized and that all the functions would be much more easily debugged, however this did come with some side effects in terms of the length of my code, requiring significantly more lines per card declaration.

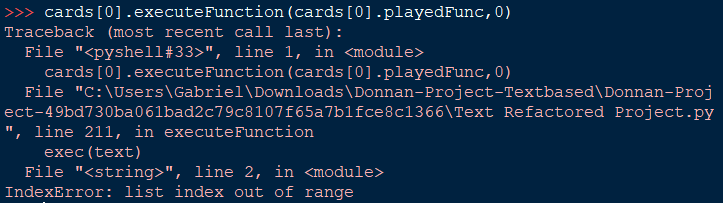
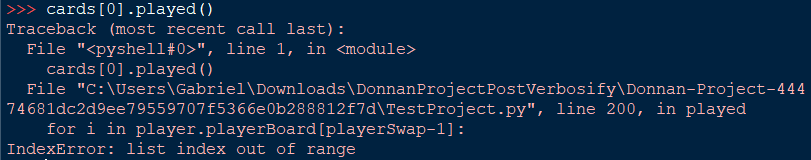
1. **def** played(self):
2. **pass**
4. **def** destroyed(self):
5. **pass**
7. **def** attacking(self):
8. **pass**
10. **def** end(self):
11. **pass**

In the “CardBase” superclass each possible function that the individual cards could have are declared with only “pass” as their code and through polymorphism the subclasses change this when initialized. The purpose of this is so that if a card were not to have a function that triggers when it is destroyed but the code tries to call it it could run the empty function with pass in it instead of crashing the program. For example for the card “Ragnaros” it has no “destroyed” function but the program will continue to run without crashing if this is called, not changing anything.



For the subclasses they each have one simple line in the \_\_init\_\_ function and the subclass nature mostly serves to allow the cards to have polymorphed versions of the CardBase class’s functions.

1. **class** Ragnaros(CardBase):
2. #All init statements for CardBase subclasses are extremely similar, merely passing in the values needed
3. **def** \_\_init\_\_(self):
4. CardBase.\_\_init\_\_(self, 5, "Ragnaros", 8, 2, 8)
6. **def** played(self):
7. #Deals 8 damage to all cards on the opposing side of the board
8. playerSwap = (player.currentPlayer % 2)+1
9. destroyed = []
10. **for** i **in** player.playerBoard[playerSwap-1]:
11. i.health -= 8

It is now much simpler to call the functions of the cards and most importantly proper tracebacks can be created if a bug is discovered in testing. 

Whilst the exec implementation recognises that a list index is out of range it cannot point out where within the string the error ocurred, making debugging significantly more tedious.

Using the new class-subclass implementation also makes the initialisation statements for the cards significantly simpler

1. cards = [
2. Ragnaros(),
3. Sylvannas(),
4. Thaurissan(),
5. Crusader(),
6. Whelp(),
7. Ogre()
8. ]

All the cards can be initialized to a list in a single statement with no parameters needing to be passed in to any of them whereas the func variable implementation required much messier, longer statements such as

1. cards.append(Card(2,"Emperor Thaurissan",6,5,5, endFunc = """if playerNum == 1:
2. for i in player.playerOneHand:
3. i.mana -= 1
4. else:
5. for i in player.playerTwoHand:
6. i.mana -= 1"""))

# Testing

# Evaluation

Analysis

Design

Implementation

Testing

Evaluation