Deckcompare.com

1 Abstract

This project aimed to build a way in which to take two Hearthstone deck strings and find out what the difference is between them in terms of cards. Using Python, Django and JSON we built a webapp that takes two input strings and returns 3 lists, cards in deck A only, cards in deck B only and cards in both decks. The number of Hearthstone expansions every year meant we tried out new decks and researched meta decks extensively every 3 months. We found no way of easily checking between two decks to see the differences and similarities so we set out to build this ourselves.

Project by: Floris van Rijn (www.github.com/florisvanrijn) & Liam Murphy (www.github.com/liammurphy14)
Repository: https://github.com/liammurphy14/deckcompare-python

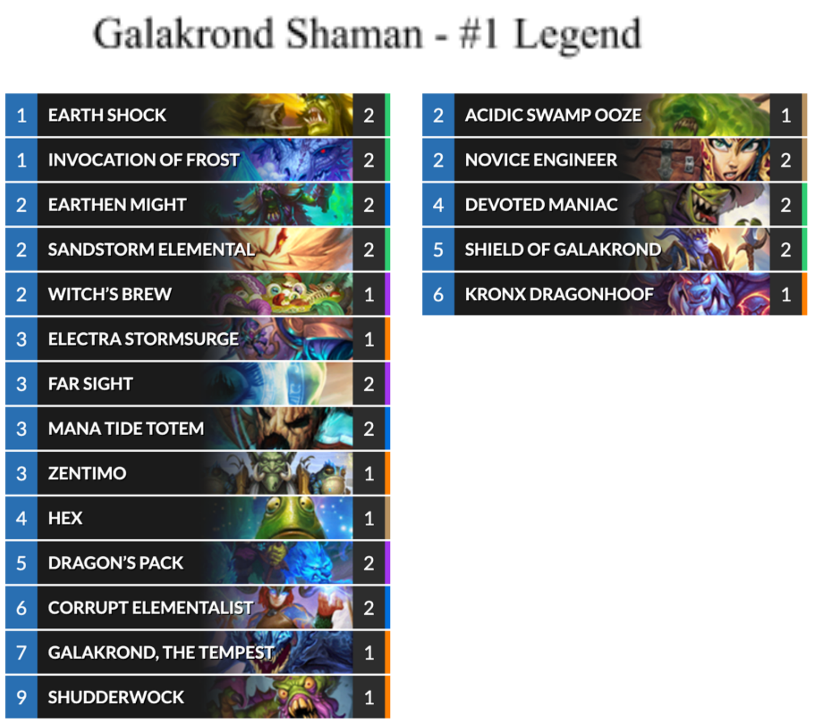
2 Unfamiliar with Hearthstone?

Hearthstone is an online multiplayer card game where two opponents use decks they've built to battle each other in which the winner gains points allowing him/her to climb the competitive ladder.

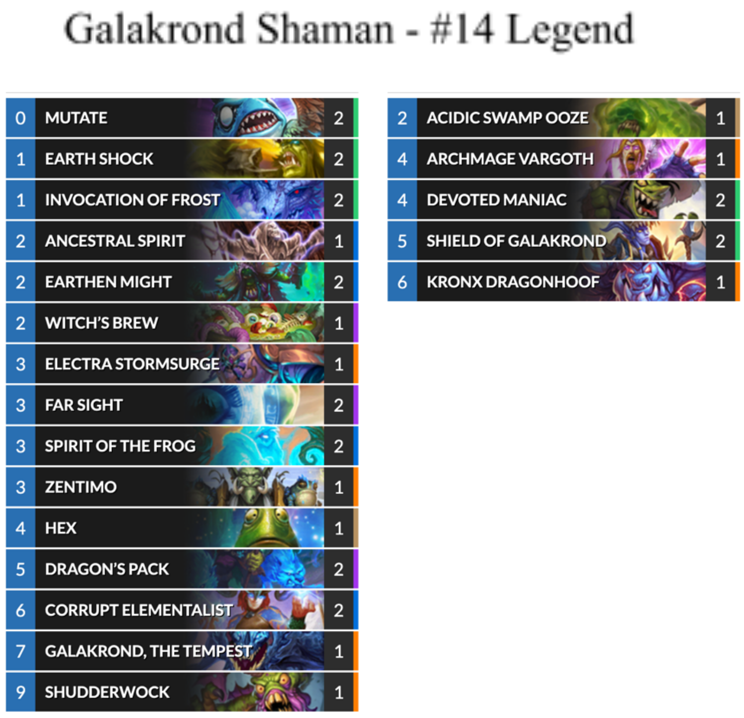
Hearthstone allows you to pick out of 9 heroes with hero specific abilities and cards. Decks consist of 30 cards and can be built out of both class specific cards as well as neutral cards. Hearthstone releases 3 new card sets every year and keeps around 1190 cards as playable in the standard game. The release of a new expansion brings a new game mechanic which often changes how the previous expansions and decks behave. Ultimately this results in the game's 'meta' decks being reinvented and tested heavily every 4 months.

In the beginning of a new expansion many new decks are crafted. Over the next few days the decks that see most play slowly settles into a few decks known as 'meta decks'. These 'strong' decks are refined more and more by casual players and professionals alike. As an average player during this time it's not unusual to have over 10 variants of a new deck saved as you research which cards and tactics best fit your style. This means you spend a lot of time researching variants of a few popular decks. Deck research is often done on websites such as Hearthstone top decks (https://www.hearthstonetopdecks.com/). Between the different versions of decks it can be hard to see immediately what the difference is. The more refined they become, the subtler the difference is and in the end it can be the tweaking of just a single card. Below are two iterations of a popular deck as of December 2019. Finding the variation in cards used is a time consuming process done by hand. The differences in these two decks are that the first deck contains 2x Sandstorm Elemental, 2x Novice Engineer and 2x Mana Tide Totem that the second in turn replaces these with 1x Ancestral Spirit, 1x Archmange Vargoth, 2x Spirit of the Frog and 2x Mutate.

Going through this process ourselves we realised there was no existing tool out there that could take two decks and instantly return the difference between them to the user. This is what we set out to make.


Galakrond Shaman - #1 Legend

1	EARTH SHOCK	2
1	INVOCATION OF FROST	2
2	EARTHEN MIGHT	2
2	SANDSTORM ELEMENTAL	2
2	WITCH'S BREW	1
3	ELECTRA STORMSURGE	1
3	FAR SIGHT	2
3	MANA TIDE TOTEM	2
3	ZENTIMO	1
4	HEX	1
5	DRAGON'S PACK	2
6	CORRUPT ELEMENTALIST	2
7	GALAKROND, THE TEMPEST	1
9	SHUDDERWOCK	1
2	ACIDIC SWAMP OOZE	1
2	NOVICE ENGINEER	2
4	DEVOTED MANIAC	2
5	SHIELD OF GALAKROND	2
6	KRONX DRAGONHOOF	1


Galakrond Shaman - #14 Legend

0	MUTATE	2
1	EARTH SHOCK	2
1	INVOCATION OF FROST	2
2	ANCESTRAL SPIRIT	1
2	EARTHEN MIGHT	2
2	WITCH'S BREW	1
3	ELECTRA STORMSURGE	1
3	FAR SIGHT	2
3	SPIRIT OF THE FROG	2
3	ZENTIMO	1
4	HEX	1
5	DRAGON'S PACK	2
6	CORRUPT ELEMENTALIST	2
7	GALAKROND, THE TEMPEST	1
9	SHUDDERWOCK	1
2	ACIDIC SWAMP OOZE	1
4	ARCHMAGE VARGOTH	1
4	DEVOTED MANIAC	2
5	SHIELD OF GALAKROND	2
6	KRONX DRAGONHOOF	1

3 Deck strings

Hearthstone allows the easy sharing of decks through deck strings and lets you import them into the game to build the decks. Here is an example of a deck string:

`AAECAaoICP4Figfv9wKZ+wLLhQPFmQPjtAPTwAMLnAKBBP8Fsgaw8ALPpQO3rQO5rQP+rgOqrwPQrwMA`

To understand how this results in the cards shown just above as "Galakrond Shaman - #1 Legend", it's best to take a step-by-step approach starting with decoding the string.

3.1 Decoding the deck string

Deck strings are a base64-encoded compact byte string. Where a string is an array of characters (a somewhat abstract concept that can't be stored directly), a byte string is a sequence of bytes. If you know which encoding scheme was used to turn the binary data into an ASCII string then you can easily encode or decode between them.

From an online search we know that deck strings are base64-encoded, and using the following Python code and base64 module we can decode the sequence of characters

```python
import base64

deckcode = "AAECAaoICP4Figfv9wKZ+wLLhQPFmQPjtAPTwAMLnAKBBP8Fsgaw8ALPpQO3rQO5rQP+rgOqrwPQrwMA"
code = base64.b64decode(deckcode)
print(code)
```

Giving us the following hexadecimal code:

```
/Users/florisvanrijn1/Desktop/Python_project/deck_Compare/deck_writeup1.py
b'\x00\x01\x02\x01\xaa\x08\x08\xfe\x05\x8a\x07\xef\xf7\x02\x99\xfb\x02\xcb\x85\x03\xc5\x99\
x03\xe3\xb4\x03\xd3\xc0\x03\x0b\x9c\x02\x81\x04\xff\x05\xb2\x06\xb0\xf0\x02\xcf\xa5\x03\xb7
\xad\x03\xb9\xad\x03\xfe\xae\x03\xaa\xaf\x03\xd0\xaf\x03\x00'
```

Getting it into individual bytes requires one more decoding operation using the following code:

```python
binCode = []
for i in range(len(code)):
    binCode.append(format(code[i], "08b"))

print(binCode)
```

This code inherently knows to split each value of the hexadecimal string based on the backslash '\' and formats it into an eight digit zero-padded (on the left) binary value as indicated by "08b". Looking back, this part of the code wasn't necessary as hexadecimal is already stored as binary under the hood.

Nevertheless, the bincode list of strings results in the following output:

```
Users/florisvanrijn/Desktop/Python_project/deck_Compare/deck_writeup1.py
['00000000', '00000001', '00000010', '00000001', '10101010', '00001000', '00001000',
'11111110', '00000101', '10001010', '00000111', '11101111', '11110111', '00000010',
'10011001', '11111011', '00000010', '11001011', '10000101', '00000011', '11000101',
'10011001', '00000011', '11100011', '10110100', '00000011', '11010011', '11000000',
'00000011', '00001011', '10011100', '00000010', '10000001', '00000100', '11111111',
'00000101', '10110010', '00000110', '10110000', '11110000', '00000010', '11001111',
'10100101', '00000011', '10110111', '10101101', '00000011', '10111001', '10101101',
'00000011', '11111110', '10101110', '00000011', '10101010', '10101111', '00000011',
'11010000', '10101111', '00000011', '00000000']
```

Each deck string follows a standard format for bytes so that it can more easily be turned into the final list of cards:

```
b0 b1 b2 | b3     | b4           | b5     | b6 .. b17      | b18    | b19 .. b42     | b43
0  1  2  | 1      | 1f           | 6      | c7 .. f8       | c      | 8d  .. c       | 0
header   | length | hero payload | length | 1 card payload | length | 2 card payload | footer
```

The header contains the first three bytes. Where the byte0 (b0) is always 0, byte1 (b1) denotes the format version (currently 1) and byte2 (b2) denotes whether the version of the deck is 1,"wild" where all cards are available for use or 2, "standard" where only the most recent subset of cards are useable.

Hero payload (b4)
B3 indicates the length of the hero payload. The format of bites allows for a higher number than 1, however this will always be 1 as a deck is made for 1 hero.

1 card payload (b6 – b17)
B5 indicates the length of the 1 card payload. This is the cards in a deck for which there is only 1 copy.

2 card payload (b19 – b42)
B18 indicates the length of the 2 card payload. This is the cards in a deck for which there are 2 copies.

Finally there's a footer (b43) to indicate the end.

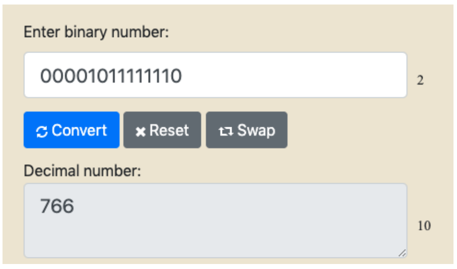
This information will be important when we match the card IDs to the card name and need to return the number of that card in the deck.

3.2 Varint

Hearthstone card IDs are stored as varints. A varint is a way to store larger numbers into bytes using 8bit binary using only the absolute required amount of memory. An 8bit byte can store as maximum value 255 (in binary = 11111111). A varint takes 7 bits of each byte to represent a number and allows you to concatenate several bytes into a larger value. It uses the first bit in every byte to indicate if the next byte is part of the same varint or if it starts a new varint. If the first bit is equal to 1, the next byte also makes up the varint and if it's equal to 0, the next byte doesn't add to the varint. We can explore this better through an example.

After the hero payload, the very first card we have in our deck is Hex.

Looking at the bytes after our hero payload we have the byte `'11111110'`. The first bit of value `'1'` indicates the next byte is also part of this varint, so we take the next byte which equals `'00000101'` and see the first bit is '0', meaning this is the last byte to make up this varint.
Removing the marker bit gives us `'1111110'` and `'0000101'` respectively. The nature of varints is to store the least significant values first meaning we must move `'0000101'` to in front of `'1111110'`, giving us `'00001011111110'`. We then take this value and covert it into decimal (base10).


Enter binary number:
00001011111110
2

Decimal number:
766
10

Cross-checking the value of 766 in our Hearthstone API that stores all the card IDs, we see the ID of 766 is equal to the card Hex.

`"dbfId":766, "name":"Hex"`

Now we have to turn this logic into code such that for every value in our original binary list of strings we make the varints and turn them into decimal values. The following code does exactly that.

```
1.      #decode varints
2.      final = []
3.
4.      i = 0
5.      while i < len(binCode):
6.          group = []
7.
8.          x = 1
9.          while x:
10.             group.append(binCode[i])
11.             if binCode[i][0] == "1":
12.                 i += 1
13.             else:
14.                 x = 0
15.         i += 1
16.
17.         combined = ""
18.         for j in range(len(group)):
19.             combined += group[-j-1][1:]
20.         final.append(combined)
21.         #print(combined)
22.
23.     for i in range(len(final)):
24.         final[i] = int(final[i], 2)
25.
26.     return final
```

Although the logic is already explained above, and the final output that is returned is a list of the card IDs, (also called dbfIDs), it's helpful to explain how the code achieves this.

Starting on line 2 we already have the previously filled in 'binCode' which is equal to the list of binary card ID codes. In a while loop we set a counter equal to 0 and while smaller than the length of the list we initiate a list called group. Nesting another while loop, we set a value of x equal to 1. This is done to be able to take the right amount of bytes to form the varint. While x = 1 we append the binary value to the group list and then check to see what the value of the first bit in the byte is with the code binCode[i][0]. If this value is equal to 1, it means the next byte is part of the varint and we add it to our group[] list. If this value isn't 1 we change the value of x to break out of the while loop and then increase the value of 'i' so next time it starts the process again with the next byte.

Before that starts again we take the list we just formed for group[], which now combines our two values of `'11111110'` and `'00000101'` and start a new 'for loop'. This loop on line 18 iterates over the length of the group equal to 2 and for each value makes a new string called "combined" that is the j-1th value of group[j]. This means that for the first value in group[], equal to group[0] it actually takes the -1th which is the last value. This only adds to the string "combined" the last 7 bits and appends that to a list called final. It does this for all the values before moving into the final for loop on line 23 that transforms the varints into base10 ints..

This system of incrementing a counter and using an while x = 1 can stand to improve from refactorization. Rewriting the code we'd suggest using the following pseudo code:

```
varintGroups = []

for each byte in listOfBytes:
      if byte[0] = 1:
              append byte to current sub-list
      else if byte[0] = 0
              append byte to current sub-list, then close out sub-list and create new one
```

This creates a list of lists where each list is a group of bytes, avoiding the entire iterator as you loop through all the bytes.

So far we've gone from a beginning deck string to having a list of card IDs, number of each card that is in the deck, and the type of deck/hero we're using.

```
[0, 1, 2, 1, 1066, 8, 766, 906, 48111, 48537, 49867, 52421, 55907, 57427, 11, 284,
513, 767, 818, 47152, 53967, 54967, 54969, 55166, 55210, 55248, 0]
```

We can now start to write a core python script that will decode two deck strings and compare them.

3.3 1 and 2 card payload

When comparing two lists of dbfIDs, its not enough to check if a card is in one list and not the other as the difference between two decks can simply be whether there's 1 copy or 2 copies of a card in a deck. This means that for both deck codes we first have to split the dbf IDs into a list of 1 copy cards and 2 copy cards.

Taking our sample list of dbfIDs from our Shaman Galakrond deck we have the following:

```
[0, 1, 2, 1, 1066, 8, 766, 906, 48111, 48537, 49867, 52421, 55907, 57427, 11, 284,
513, 767, 818, 47152, 53967, 54967, 54969, 55166, 55210, 55248, 0]
```

The 5th element in the list indicates the number of 1 card payload cards that are to follow. For our list of ids this means the next 8 values after b5 are 1 card payloads. B14 in our list is equal to 11, and this indicates the number of 2 card copies there are to follow. In total this is equal to the 30 cards that form a deck. b14 in our sample indicates the number of 2 card copies to follow, but this most likely won't be b14 in the next deck string we input due to the varying number of 1 copy cards.

The only certainty we have are the first 6 bytes.

```
b0 b1 b2 | b3     | b4           | b5     | b6 .. b17      | b18    | b19 .. b42     | b43
0  1  2  | 1      | 1f           | 6      | c7 .. f8       | c      | 8d  .. c       | 0
header   | length | hero payload | length | 1 card payload | length | 2 card payload | footer
```

We know that b5 will indicate the amount of 1 card payloads there are, and therefore we know that the number indicating the 2 card payload is located at position 6 + b5.

```python
def deckListBuilder(deckcode):
    #turns deckcode into list of singles and doubles
    deck = decoder.decode(deckcode)

    singles = deck[6:6+deck[5]]
    doubles = deck[7+deck[5]:7+deck[5]+deck[6+deck[5]]]
    cardList = [singles,doubles]

    return cardList
```

This logic is what we use to build the list of 1 copy cards called `singles` taking a slice of the larger deck list.

Extending this logic, `doubles`, takes the values of all the 2 card payloads and is equal to:

```
doubles = deck[7+deck[5]:7+deck[5]+deck[6+deck[5]]]
```

Which appends the values in deck from deck[7+deck[5]] equal in our string to byte15, which is the first id of the 2 card payload, up until 7+deck[5] + deck[6+deck[5]]. This second part of the range is best broken down in parts. 7+deck[5] is equal to the position of the first payload 2 card which for us is byte15. deck[6+deck[5]] is equal to the byte that indicates the number of 2 card payloads there are. In our string this is the position in the list of the last 2 card payload. Working through it, the last value is “7 + 8 + deck[14]” which is equal to “7 + 8 + 11” = 26. b26 in our list is 0.

Both the 1 and 2-card payloads are appended to cardList which returns the following list of lists:

```
[[766, 906, 48111, 48537, 49867, 52421, 55907, 57427], [284, 513, 767, 818, 47152,
53967, 54967, 54969, 55166, 55210, 55248]]
```

At this point we’ve transformed our original deck string of:

```
AAECAaoICP4Figfv9wKZ+wLLhQPFmQPjtAPTwAMLnAKBBP8Fsgaw8ALPpQO3rQO5rQP+rgOqrwPQrwMA
```

Into a list of lists containing a list of the 1 card payloads and a list of the 2 card payloads.

```
[[766, 906, 48111, 48537, 49867, 52421, 55907, 57427], [284, 513, 767, 818, 47152,
53967, 54967, 54969, 55166, 55210, 55248]]
```

The next stage is to take these values for two deck strings and compare them to each other.

3.4 Comparing cards

To make the comparison of cards easier we first make a list of the combined unique cards across both decks with the following code:

```python
def getAllCards(cardLists):
    #returns list of all unique cards across both decks for iteration
    decks = []
    for card in cardLists:
        if card not in decks:
                decks.append(card)
    return decks
```

From here we start the actual comparison:

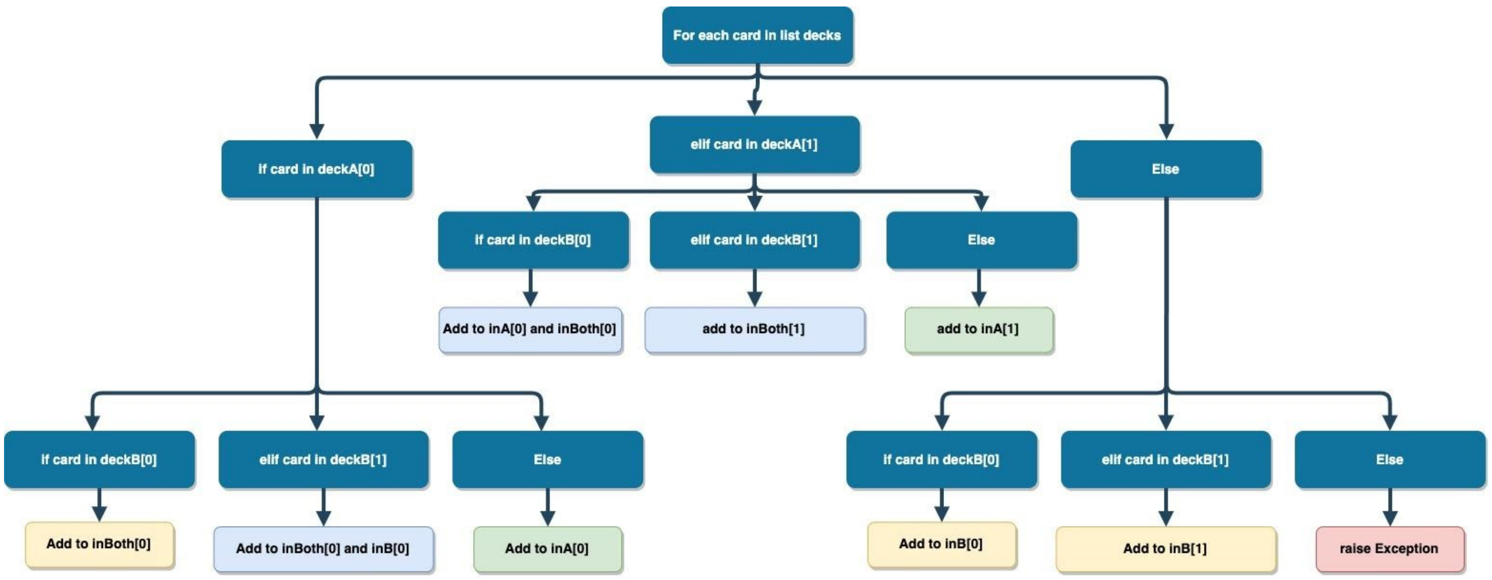
```
def compare(deckcodeA, deckcodeB):
    #runs decoder function to end up with list of IDs
    deckA = deckListBuilder(deckcodeA)
    deckB = deckListBuilder(deckcodeB)

    decks = getAllCards(deckA[0] + deckA[1] + deckB[0] + deckB[1])
    inA = [[],[]]
    inB = [[],[]]
    inBoth = [[],[]]
```

This code first initiates two variables for the deck inputs, deckA and deckB and passes in the value from cardList for each deck string. This is equal to setting their value to `cardList = [singles,doubles]`. We then set a third variable called decks equal to a single list of unique dbfIDs for all the cards in both decks.

Finally we instantiate the empty lists of lists for the cards that will be inA, inB and inBoth and allow for the input of a list of single copy cards and double copy cards.

The brunt of the comparison is done in a long if/elif/else statement that checks for 3 potential situations and within each situation are 3 more if/elif/else statements. The following flowchart shows when the card should be appended to inA, inB, inBoth or a multiple of them.



Walking through the flowchart, the if statement initially checks for the presence of the card in deckA[0] which is the 1 copy cards in the first deck string, then within this it checks whether it's also in the 1 copy card list of deck string B, the 2 copy card list of deck string B or neither of those and just in A. If the card is both in the 1 copy card list of A as well as the 1 copy card list of deckB, then its added to the single card list of inBoth as both decks possess a single copy the card. This same logic is executed for the other cases and results in the following code that executes exactly the flowchart above:

```python
for card in decks:
    if card in deckA[0]:
        if card in deckB[0]:
            #1 in each deck
            inBoth[0].append(card)

        elif card in deckB[1]:
            #1 in A, 2 in B
            inBoth[0].append(card)
            inB[0].append(card)

        else:
            #1 in A
            inA[0].append(card)

    elif card in deckA[1]:
        if card in deckB[0]:
            #2 in A, 1 in B
            inA[0].append(card)
            inBoth[0].append(card)

        elif card in deckB[1]:
            #2 in each deck
            inBoth[1].append(card)

        else:
            #2 in A
            inA[1].append(card)

    else:
        if card in deckB[0]:
            #1 in B
            inB[0].append(card)

        elif card in deckB[1]:
            #2 in B
            inB[1].append(card)

        else:
            raise Exception("You shouldn't be here. Card: " + str(card))
```

At this stage our deck string for both deck A and deck B has been turned into a list of dbfIDs for cards inA, inB and inBoth. However, it's still just a list of IDs, which is of little value to an end user. The final step in our program is to take the list of IDs, map them to their names and return these 3 lists to the user.

```python
with open('cardData', 'r') as f:
    #cardList is a dictionary of dictionaries
    cardList = json.load(f)


def printNames(list):
    for i in range(2):
        #implement exception handling incase json file is out date
        try:
            for j in range(len(list[i])):
                cardName = cardList[str(list[i][j])]['name']
                print(str(i+1) + "x " + cardName)
        except:
            jsonConvert.build()
```

We begin the transformation of dbfIDs to names by reading in a json file taken from a HearthstoneAPI made by https://hearthstonejson.com/. This contains 19 fields of potential data for each card in the game.

We've wrapped the process in a try/except statement. We do this as the frequency of visitors significantly increases the moment an expansion drops. Comparing new decks will result in dbfIDs not currently in our JSON file. If this happens then our code calls the API and downloads the new JSON file into server memory. Expansions are frequent during the year and this shouldn't impact the end-user. After catching the exception the code continues to build the output requested.

Our final output from two deck strings is as follows:

```
---------------------
Cards only in Deck 1:
---------------------
2x Novice Engineer
2x Mana Tide Totem
2x Sandstorm Elemental
---------------------
Cards in both decks:
---------------------
1x Hex
1x Acidic Swamp Ooze
1x Shudderwock
1x Electra Stormsurge
1x Zentimo
1x Witch's Brew
1x Kronx Dragonhoof
1x Galakrond, the Tempest
2x Earth Shock
2x Far Sight
2x Earthen Might
2x Dragon's Pack
2x Corrupt Elementalist
2x Shield of Galakrond
2x Devoted Maniac
2x Invocation of Frost
---------------------
Cards only in Deck 2:
---------------------
1x Ancestral Spirit
1x Archmage Vargoth
2x Spirit of the Frog
2x Mutate
```

This is useful to us offline, however, it isn't complete until its usable as a webapp. To implement this we package the entire program into a Django webapp and make it usable on the domain www.deckcompare.com.