Pontoon AI Designing

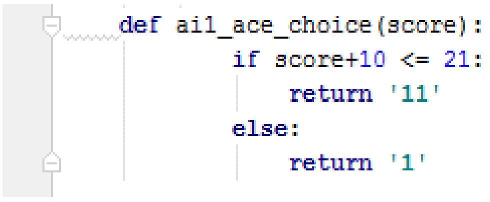
Introduction

This project is based on designing AI's (i.e. something for a player to play the game against) to play the game Pontoon.

Pontoon is similar to 21. Each player receives two cards and sticks or twists. If their current score is over 21, they bust and everyone knows they bust. Otherwise they stick. Aces can count as 1 or 11, and no splits are allowed, and neither is the five card trick rule.

I spent a good lot of time writing the code for the initial set up and drawing random cards which can never be repeated, as well as for a player to input their stick or twist. This can be seen in the list of codes for more details, and isn't too important.

After building the battleground for AI's to verse each other, I started designing my AI's. Note that **I do not want the AI's to cheat**, meaning, I do not want them to look at the cards of the opponent. They must rely only on information a normal player has access to: their own cards, the cards opponents twist on, the number of cards they and opponents have, etc. Secondly, the AI's could win significantly more if they play second and stick immediately if the player busts. This is boring, so I shall ignore this concept until stated otherwise, and assume that no player reveals if they have bust or stuck, just that they have finished. Each AI needed to have at least two parts to them: the choice of whether to stick or twist, and, the choice of whether their ace(s) be valued at 1 or 11. The second choice seems to be very trivial, as one will merely maximise the value of the aces without busting (unless I introduce the rule of five cards beating all other hands except for pontoon, which I don't). This code can be written simply, and is as follows:



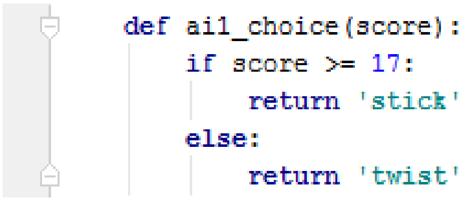
```
def ai1_ace_choice(score):
        if score+10 <= 21:
            return '11'
        else:
            return '1'
```

Here, 'score' will be the current score the AI has. This will most likely be the ace choice for all the AI's I design.

Now we can move on to the more interesting aspect, that is, the choice of stick or twist.

Alpha 1.0

I will call my first AI 'Alpha 1.0', as it is the start of the Greek alphabet, much like this is the start of my AI's. A very simple approach that, perhaps, most players of the game go by is using a 'hit limit'. That is, once the value of their cards reaches a certain limit, N, then they will stick. Much like the code for the ace choice, this is incredibly simple to write, and is as follows:



```
def ai1_choice(score):
    if score >= 17:
        return 'stick'
    else:
        return 'twist'
```

Here we have used 17 as our limit, meaning if your score is 17 or above, then you will stick. But the questions is, what would be the ideal limit? Would using a limit of 17 be better than using 16? A way to find out which limit is best is by competing different AI's with different limits. Note, that all we are proving by doing this is which limit works best ***against other limit.*** I cannot confidently say that the winner of the limits versing each other would be best against other types of AI, as it may be so that a lower limit would work best against one AI and a higher limit better against another (though I somewhat doubt it, but I will be scientific about this.) So, I will create a table array in excel of the results of the different limits playing 100,000 games against each other (which took ~8 seconds to run for each matchup, any more would have been too time consuming). Additionally, I tried limits from 12 to 21, as any lower than 12 should always twist, as it is impossible to bust by twisting. The results are as follows:

		AI1's limit									
	vs.	12	13	14	15	16	17	18	19	20	21
AI2's limit	12	AI1 wins: 44524 AI2 wins: 44833 Draws: 10643									
	13	AI1 wins: 40625 AI2 wins: 49556 Draws: 9819	AI1 wins: 44629 AI2 wins: 44422 Draws: 10949								
	14	AI1 wins: 38070 AI2 wins: 52870 Draws: 9060	AI1 wins: 41005 AI2 wins: 48695 Draws: 10300	AI1 wins: 44036 AI2 wins: 44160 Draws: 11804							
	15	AI1 wins: 37373 AI2 wins: 54337 Draws: 8290	AI1 wins: 39461 AI2 wins: 50806 Draws: 9733	AI1 wins: 41842 AI2 wins: 47062 Draws: 11096	AI1 wins: 43206 AI2 wins: 43665 Draws: 13129						
	16	AI1 wins: 39024 AI2 wins: 53619 Draws: 7357	AI1 wins: 39814 AI2 wins: 51315 Draws: 8871	AI1 wins: 41036 AI2 wins: 48212 Draws: 10752	AI1 wins: 41750 AI2 wins: 45211 Draws: 13039	AI1 wins: 42202 AI2 wins: 42313 Draws: 15485					
	17	AI1 wins: 42252 AI2 wins: 51318 Draws: 6430	AI1 wins: 42380 AI2 wins: 49323 Draws: 8297	AI1 wins: 42481 AI2 wins: 47105 Draws: 10414	AI1 wins: 42231 AI2 wins: 44785 Draws: 12984	AI1 wins: 41448 AI2 wins: 42277 Draws: 16275	AI1 wins: 40165 AI2 wins: 40118 Draws: 19717				
	18	AI1 wins: 48494 AI2 wins: 46206 Draws: 5300	AI1 wins: 47335 AI2 wins: 45261 Draws: 7404	AI1 wins: 46734 AI2 wins: 43264 Draws: 10002	AI1 wins: 45223 AI2 wins: 41612 Draws: 13165	AI1 wins: 42851 AI2 wins: 39996 Draws: 17153	AI1 wins: 40240 AI2 wins: 38507 Draws: 21253	AI1 wins: 37290 AI2 wins: 36607 Draws: 26103			
	19	AI1 wins: 56378 AI2 wins: 39354 Draws: 4268	AI1 wins: 54780 AI2 wins: 38582 Draws: 6638	AI1 wins: 52656 AI2 wins: 37500 Draws: 9844	AI1 wins: 50081 AI2 wins: 36410 Draws: 13509	AI1 wins: 46851 AI2 wins: 35255 Draws: 17894	AI1 wins: 42531 AI2 wins: 34463 Draws: 23006	AI1 wins: 37541 AI2 wins: 33067 Draws: 29392	AI1 wins: 31808 AI2 wins: 32057 Draws: 36135		
	20	AI1 wins: 67201 AI2 wins: 29686 Draws: 3113	AI1 wins: 65036 AI2 wins: 29074 Draws: 5890	AI1 wins: 61873 AI2 wins: 28529 Draws: 9598	AI1 wins: 57994 AI2 wins: 28040 Draws: 13966	AI1 wins: 52938 AI2 wins: 27595 Draws: 19467	AI1 wins: 47509 AI2 wins: 26979 Draws: 25512	AI1 wins: 40988 AI2 wins: 26127 Draws: 32885	AI1 wins: 33423 AI2 wins: 25424 Draws: 41153	AI1 wins: 24809 AI2 wins: 24954 Draws: 50237	
	21	AI1 wins: 86072 AI2 wins: 13376 Draws: 552	AI1 wins: 82815 AI2 wins: 13294 Draws: 3891	AI1 wins: 78589 AI2 wins: 13014 Draws: 8397	AI1 wins: 72920 AI2 wins: 13016 Draws: 14064	AI1 wins: 66410 AI2 wins: 12963 Draws: 20627	AI1 wins: 58928 AI2 wins: 12993 Draws: 28079	AI1 wins: 50625 AI2 wins: 12422 Draws: 36953	AI1 wins: 40448 AI2 wins: 12348 Draws: 47204	AI1 wins: 29076 AI2 wins: 12213 Draws: 58711	AI1 wins: 12111 AI2 wins: 12140 Draws: 75749

For someone who prides himself on his spreadsheet formatting and designing, this certainly wasn't my finest piece of work, but it suffices after you understand what's going on. If not, here's some conclusions I can draw from the results:

Some explanation

- Firstly, I included same limit matchups as a control. We can see that they score extremely similarly, with an error of around a couple hundred. I'll use anything more than plus or minus 700 wins as a significant victory
- It was not necessary to show the other half of the grid, as these would be identical to the other half, but with AI1 and AI2 switched

Result analysis

- The limit of 17 never lost a single matchup, it always won (albeit only slightly at times). It is tempting to say this is the best limit to go by, and it is in a way, but we shan't leap to conclusions. This is no surprise, as this is the limit the dealer uses in blackjack in casinos, thus creating the house advantage
- I was right to be suspicious about some AI's being better against certain AI's than others, as you can see that 15 beat 12 more than 17 beat 12. We can conclude that the best technique is to slightly up your limit over your opponent rather than sticking with 17 all the time
- Using any limit above 17 seems to be an awful idea. When playing in real life, I'd often use 18 as a limit, but it performs abysmally against every limit worse than itself, as does every single limit above 18. 17 is definitely a significant limit in this game, and only a fool would use any limit higher than it
- Furthermore, after looking at the exact number of wins, the best technique is to stick at the lowest limit possible if your opponent is using a limit of 18 or higher
- If your opponent is using a limit lower than 17, then use a limit of two (if they're using a very low limit) or one (if they use 15 or 16) more than them

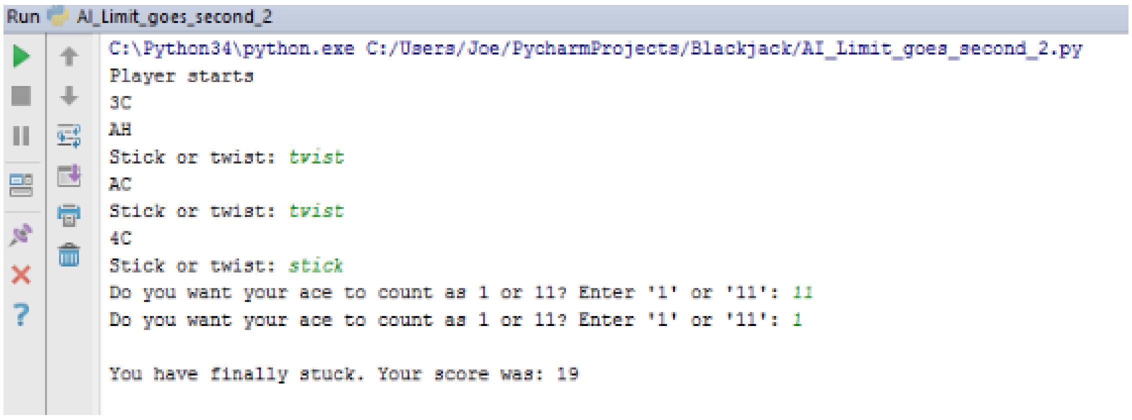
Other comments

- I included draws in these results, as they may come in handy in the future when I become more advanced, but I can draw little conclusions from it at the moment
- I was a little cautious of saying 17 was best, particularly when an error exists and it scored so closely with 16, which did better than it against a lot of the other limits. So, I ran 1,000,000 games to reduce the error, and it confirmed that 17 was indeed the better limit against 16, scoring 413,303 losses and 425,725 wins, and 160,972 draws.

So, we will use 17 as our limit for Alpha 1.0, despite the fact that 15 and 16 often scored better against lower limits (though I don't think anyone in a real game would stick on such low limits), as it won every single game. Our aim is to create an AI which wins more often than not, not to create an AI which wins a lot against bad players. So, I have designed Alpha 1.0.

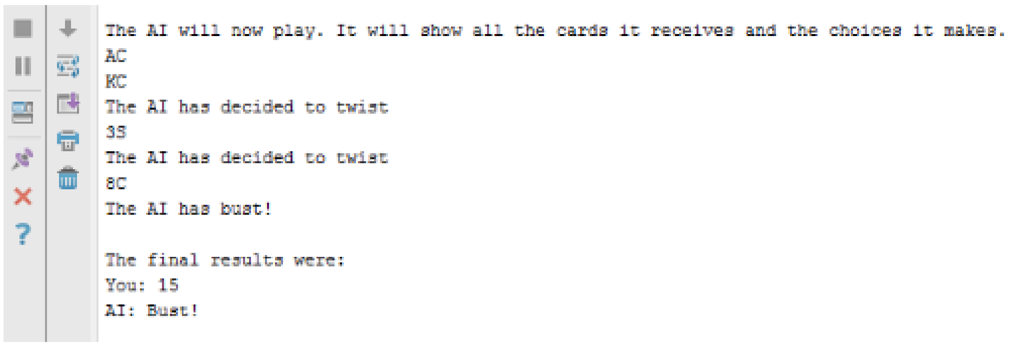
Alpha 2.0

It’s hard to spot dumb things an AI can do in just wins and losses, particularly when you don’t see how it plays the game. So when I played against Alpha 1.0, I noticed a massive flaw in its gameplay. It is best demonstrated by this example:



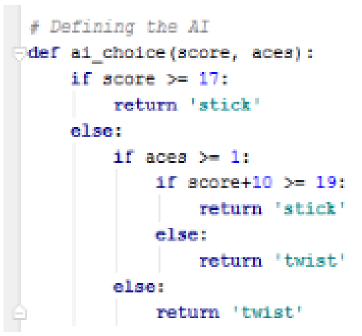
```
C:\Python34\python.exe C:/Users/Joe/PycharmProjects/Blackjack/AI_Limit_goes_second_2.py
Player starts
3C
AH
Stick or twist: twist
AC
Stick or twist: twist
4C
Stick or twist: stick
Do you want your ace to count as 1 or 11? Enter '1' or '11': 11
Do you want your ace to count as 1 or 11? Enter '1' or '11': 1

You have finally stuck. Your score was: 19
```

When presented with lots of aces, I can choose to think of them as 1 or 11 as I play. So I thought of my first score as 4 or 14. Either way, I wanted to twist. Next, I thought of my score as either 5 or 15 or 25, none of which are desirable scores, so I twist again. Now, I have either 9 or 19 or 29. I liked 19, so stuck with that, though perhaps carrying through with 9 would have been a good option. Note that I am not playing the rule where 5 cards beats all but pontoon; that would have changed the concept of my choices very differently. Now let’s see how Alpha 1.0 handles a similar situation.



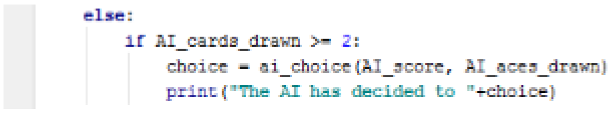
```
The AI will now play. It will show all the cards it receives and the choices it makes.
AC
KC
The AI has decided to twist
3S
The AI has decided to twist
8C
The AI has bust!

The final results were:
You: 15
AI: Bust!
```

It is obvious how dumb it played here. It has a pontoon at the start, it should have stuck with that! But it’s treating aces as 1’s (which I had to do for bust reasons), so it thinks it has a score of 11, so twists, and ultimately, busts. To prevent this, I need to introduce a new part to its stick/twist option, saying that “if you have one (or more) ace(s), then you should stick if you have a high enough score of N”. Now we will have two parameters: the ace being 1 limit (I will call this the hard limit (I’ve later realised ‘hard limit’ has a different meaning in technical gameplay, but it makes sense to me)), and the ace being 11 limit (I will call this the ace limit). This is fairly simple to code for one ace and turns the problem two dimensional. Any more than one may be tricky, and it is fairly uncommon, though I’ll update that in Alpha 2.1. So, I attempt to code the situation of having an ace. Firstly, I needed to redesign the code so that the (AI/player)_aces_drawn is updated after every card is drawn instead of just at the end. Here is an outline of the code:



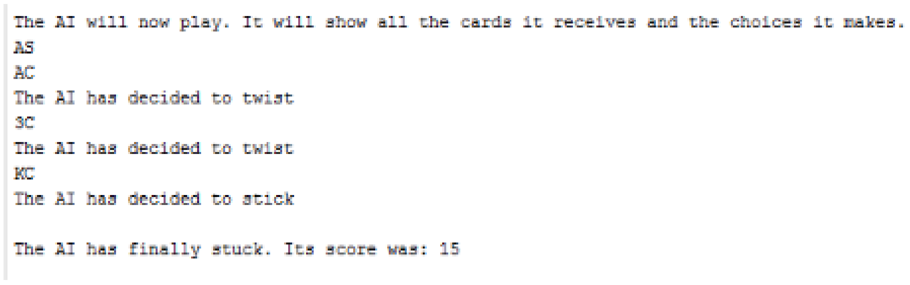
```
# Defining the AI
def ai_choice(score, aces):
    if score >= 17:
        return 'stick'
    else:
        if aces >= 1:
            if score+10 >= 19:
                return 'stick'
            else:
                return 'twist'
        else:
            return 'twist'
```

In words, this means: “if your score with aces as 1’s is greater than equal to 17, then stick. Otherwise, if you have aces, then if you treat one of the aces as 11, if your score is at or over 19, then stick, else twist again.” In the case of two or more aces, we only treat one of the aces as 11. If we were to treat both as 11, we would bust immediately, so it is not worth considering (so Alpha 2.1 will not be as I intended. The case of 2+ aces is trivial). Here, we need to find the ideal values for, what are in this example, 17 and 19. But first, let’s see update the rest of the code and see if Alpha 2.0 plays as we want it to. Here is the updated code further down for Alpha 2.0:



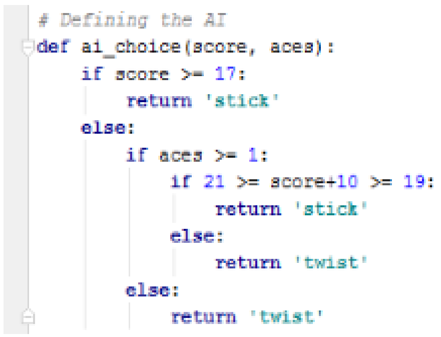
```
        else:
            if AI_cards_drawn >= 2:
                choice = ai_choice(AI_score, AI_aces_drawn)
                print("The AI has decided to "+choice)
```

Fortunately, on my first gameplay, I noticed an error in Alpha 2.0’s coding:



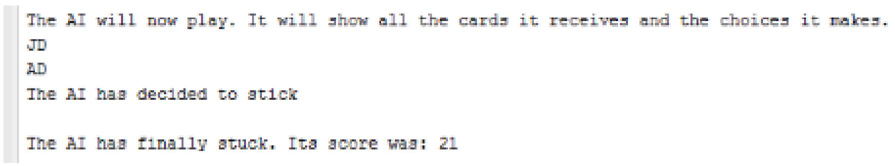
```
The AI will now play. It will show all the cards it receives and the choices it makes.
AS
AC
The AI has decided to twist
3C
The AI has decided to twist
KC
The AI has decided to stick

The AI has finally stuck. Its score was: 15
```

The problem was that Alpha 2.0 thought it had a hand of 25 and would stick at that, but its ace choices knocked it down to 15. The code needs to be rewritten as:



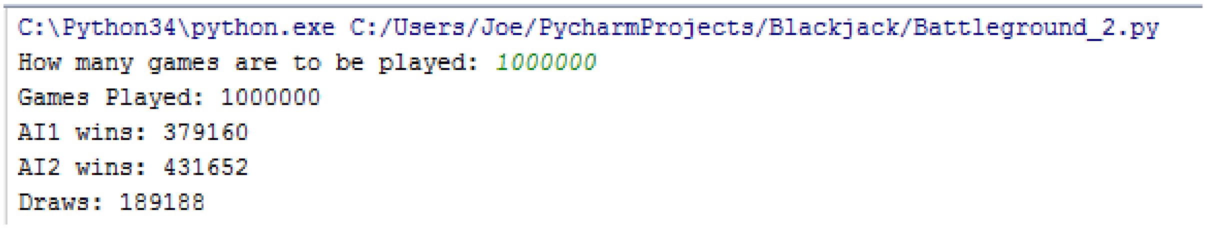
```
# Defining the AI
def ai_choice(score, aces):
    if score >= 17:
        return 'stick'
    else:
        if aces >= 1:
            if 21 >= score+10 >= 19:
                return 'stick'
            else:
                return 'twist'
        else:
            return 'twist'
```

to avoid such a situation (hard to spot, but luckily this happened so early. It’s always important to test code). Now Alpha 2.0 works as I hoped (though this isn’t the perfect example):



```
The AI will now play. It will show all the cards it receives and the choices it makes.
JD
AD
The AI has decided to stick

The AI has finally stuck. Its score was: 21
```

So we have designed the basis of Alpha 2.0, but we need to perfect the limits. Based on the results before, it feels like 17 and 17 would work best. Though from personal experience, I feel like a higher score could be used on the second limit. We will find out experimentally anyhow. Firstly, I would like to demonstrate how Alpha 2.0 beats Alpha 1.0 by using Alpha 2.0 (17, 17) vs. Alpha 1.0 (17). The results are as follows, where AI1 is Alpha 1.0 and AI2 is Alpha 2.0:



```
C:\Python34\python.exe C:/Users/Joe/PycharmProjects/Blackjack/Battleground_2.py
How many games are to be played: 1000000
Games Played: 1000000
AI1 wins: 379160
AI2 wins: 431652
Draws: 189188
```

It's fair to say that Alpha 2.0 is a fantastic improvement on Alpha 1.0. Now, we can perfect Alpha 2.0. I will create an array similar to Alpha 1.0, but I won't bother with trivial numbers. This is a four dimensional problem, and simply testing 15, 16, 17 leads to nearly 40 trials alone. I'll try some values and see if I can narrow it down from my intuition.

After some trialling, the results are getting very close, margins of victory being as little as 500 in 100,000 (consistently, so I can say it is not within the error range). In fact, (17, 19) vs (17, 18) is so close that it ***is*** within the margin of error and I'm going to have to use 1,000,000 games to separate them, and even that isn't going to separate them! By trial and error, I narrowed down the limits. Where the format is *(strong limit, limit with ace(s))*, here are my results:

		AI1's limit								
AI2's limit	vs.	(16,17)	(16,18)	(16,19)	(17,17)	(17,18)	(17,19)	(17,20)	(18,18)	(18,19)
	(16,17)	AI1 wins: 42410 AI2 wins: 42652 Draws: 14938								
	(16,18)	AI1 wins: 42331 AI2 wins: 42718 Draws: 14951	AI1 wins: 42693 AI2 wins: 42536 Draws: 14771							
	(16,19)	AI1 wins: 42265 AI2 wins: 42849 Draws: 14886	AI1 wins: 42720 AI2 wins: 42324 Draws: 14956	AI1 wins: 42483 AI2 wins: 42589 Draws: 14928						
	(17,17)	AI1 wins: 41493 AI2 wins: 42825 Draws: 15682	AI1 wins: 41904 AI2 wins: 42745 Draws: 15351	AI1 wins: 41657 AI2 wins: 42667 Draws: 15676	AI1 wins: 40929 AI2 wins: 40519 Draws: 18552					
	(17,18)	AI1 wins: 41373 AI2 wins: 43149 Draws: 15478	AI1 wins: 41542 AI2 wins: 42987 Draws: 15471	AI1 wins: 41517 AI2 wins: 42798 Draws: 15685	AI1 wins: 40474 AI2 wins: 41045 Draws: 18481	AI1 wins: 40534 AI2 wins: 40783 Draws: 18683				
	(17,19)	AI1 wins: 41264 AI2 wins: 43046 Draws: 15690	AI1 wins: 41458 AI2 wins: 42833 Draws: 15709	AI1 wins: 41868 AI2 wins: 42446 Draws: 15686	AI1 wins: 40417 AI2 wins: 40921 Draws: 18662	AI1 wins: 40680 AI2 wins: 40800 Draws: 18520	AI1 wins: 40799 AI2 wins: 40346 Draws: 18855			
	(17,20)	AI1 wins: 41727 AI2 wins: 42743 Draws: 15530	AI1 wins: 41655 AI2 wins: 42648 Draws: 15697	AI1 wins: 41854 AI2 wins: 42535 Draws: 15611	AI1 wins: 40713 AI2 wins: 40618 Draws: 18670	AI1 wins: 41038 AI2 wins: 40451 Draws: 18511	AI1 wins: 40775 AI2 wins: 40409 Draws: 18816	AI1 wins: 40640 AI2 wins: 40535 Draws: 18825		
	(18,18)	AI1 wins: 42195 AI2 wins: 41385 Draws: 16420	AI1 wins: 42654 AI2 wins: 41105 Draws: 16241	AI1 wins: 42621 AI2 wins: 40874 Draws: 16505	AI1 wins: 40514 AI2 wins: 39522 Draws: 19964	AI1 wins: 40564 AI2 wins: 39468 Draws: 19968	AI1 wins: 40462 AI2 wins: 39583 Draws: 19955	AI1 wins: 40176 AI2 wins: 39363 Draws: 20461	AI1 wins: 38104 AI2 wins: 37728 Draws: 24168	
	(18,19)	AI1 wins: 42365 AI2 wins: 41278 Draws: 16357	AI1 wins: 42447 AI2 wins: 41019 Draws: 16534	AI1 wins: 42705 AI2 wins: 40827 Draws: 16468	AI1 wins: 40385 AI2 wins: 39864 Draws: 19751	AI1 wins: 40608 AI2 wins: 39350 Draws: 20042	AI1 wins: 40419 AI2 wins: 39400 Draws: 20181	AI1 wins: 40153 AI2 wins: 39599 Draws: 20248	AI1 wins: 37605 AI2 wins: 38001 Draws: 24394	AI1 wins: 37889 AI2 wins: 37619 Draws: 24492

The poor formatting really shows this time, colours splashed everywhere, but I'll explain what they all mean:

Some explanation

- Again, I included same limit matchups as a control. The error seems to be even smaller this time, narrowing down to a couple hundred
- It was not necessary to show the other half of the grid, as these would be identical to the other half, but with AI1 and AI2 switched
- I was absolutely confident with every victory I gave. If they are within one hundred of each other, I was sure to run 1,000,000 games between them and see if there was a consistent victory, and there was (aside for one, which I'll come to)

Result analysis

- The most important factor by far was the hard limit. A hard limit of 17 beat every other hard and ace limit. There would be a point where the other limits one, but I narrowed down my range to avoid that, and it is unimportant
- Worryingly, my intuition was right, and higher ace limit is more successful instead of a 17,17 limit which I thought might work based on the previous experiment
- There were two limits which won every single matchup, and they were 17, 18 and 17, 19. When I fought these against each other, they were indistinguishable and lay inside the margin of error

So, we can conclude either 17, 18 or 17, 19 is the ideal limits for Alpha 2.0. I tried running these for 1,000,000 games and they still scored similarly. So, I'm going to have to run them from 10,000,000 games. I will also run 17, 18 vs itself, and 17, 19 vs itself as controls. Each of these matchups will roughly take 10-20 minutes to run, so I won't take these results lightly and I won't be running them again. Here are the results:

AI1 - (17, 18) vs AI2 - (17, 19)

Games Played: 10,000,000
AI1 wins: 4,066,631
AI2 wins: 4,070,513
Draws: 1,862,856

AI1 - (17, 18) vs AI2 - (17, 18)

Games Played: 10,000,000
AI1 wins: 4,071,367
AI2 wins: 4,071,787
Draws: 1,856,846

AI1 - (17, 19) vs AI2 - (17, 19)

Games Played: 10000000
AI1 wins: 4063385
AI2 wins: 4065305
Draws: 1871310

I think it's safe to say that (17,19) is the conclusive winner here. It won by more than both the control tests by at least double. And so, we have designed Alpha 2.0, coincidentally my first initial example guess. Whether this wins more often against lesser opponents like there existed such AI's in Alpha 1.0's design, I don't know, but that isn't too important. What I have designed here as an AI which I cannot significantly beat over a lot of games.

I'm unsure on how to improve on this and create Alpha 3.0. I want to take into account current cards gone that it can see, but this is tricky. My next design will be doing that and it won't follow a limit idea like I have used with Alpha. Perhaps I will combine the two after I have created my next AI in some way.

Magnus 1.0

'Beta' sounded like I was just listing through the Greek alphabet, and held too many connotations with beta versions of software, so I have decided to call my next AI 'Magnus'. No particularly reason why, I read it somewhere and thought it sounded a nice mysterious name for an AI.

My previous AI's have been too focussed on simply the total value of their current hand, and not using all the information presented to them. You can take into account the number on the cards you own, and the cards presented to your opponent (which is what I shall be using in Magnus 1.0), and remove them from the current deck to give you a more accurate understanding of what cards will come next. I'm taking a bit of a risk, but I hypothesise that the statistic "percentage next card will cause a bust" will strongly correlated to winning.

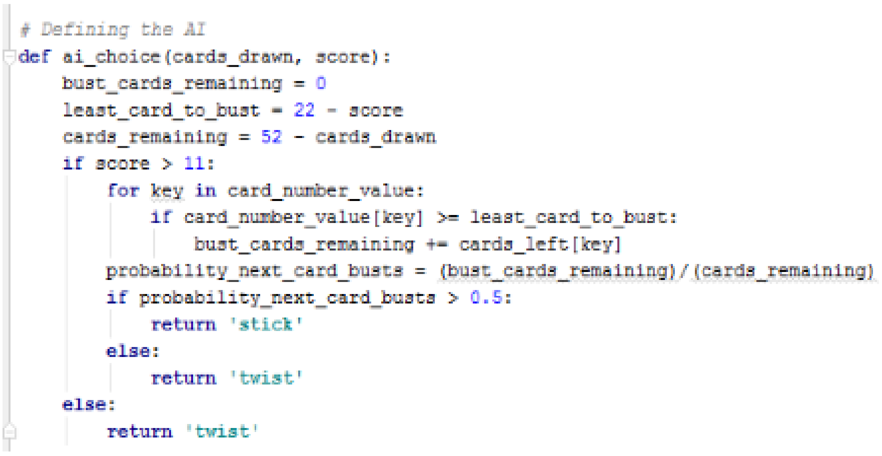
Magnus 1.0 should be smarter than Alpha 1.0 (not Alpha 2.0 because I won't write the ace dimension for Magnus in 1.0) because it takes more into account. Whether or not I will be able to prove it is smarter by designing it such that it wins more often than Alpha is a difficult ask. Alpha could barely top itself with small changes made to it. As much as I'm sure I can make Magnus smart enough to tie with Alpha, whether it can beat it a statistically significant amount is a challenging task.

I was tempted to say that 'one should stick whenever percentage is above 50%, why would you twist if you're more than likely to bust?' But after writing a basic calculator in excel and trialling that, I play fairly conservatively. I received a 10 and a 5, and I was told the probability the next card would bust me was 54% so I should stick, which seems far too conservative, especially after considering the limits of 17 and 19 in the AI's before. So, this again results in finding a magic probability number, which I shall call '$\hat{p}$' (p-hat, an estimation for the ideal probability p), this time a number rather than a limit, which I expect to be between 0.6 and 0.7. In fact, this problem is more difficult than the previous, as probabilities are continuous, whereas the limit was discrete. It is similar to a root finding problem, and I will be using a similar numerical method (interval bisection). Judging by how difficult it was to differentiate between similar AI's in Alpha, I am certain I will reach a point where 1% (for example) will make an impact which lies within the range of error.

But first, I need to write the code for Magnus 1.0 for a general percentage N, stating 'if the probability the next card busts you is above N, then stick, else twist'. This will be more technical to code than Alpha, as I'm having to create newly defined values. Additionally, when the AI plays second, it will now have an advantage, because it will be able to see the dealt cards of the opponent, giving them more information, so table arrays cannot be cut in half now as it they will be non-symmetric (in a mathematical sense, one could consider the 'vs.' as an equivalence relation in Alpha, as it has symmetry, transitivity (in a loose sense) and reflexivity, but now 'vs.' fails to have the symmetric property in Magnus.) I will need to write two different codes for the AI first and second. In fact, it's worth considering these as two different AI's, because their gameplay is quite different, and their optimal numbers will probably be different too. So, I shall write this in two different sections, though the content from here on out will not follow a chronological order.

Magnus 1.0 - First

The first should be easy to write, but the second I will struggle with. Here is the code for the AI playing first:



```
# Defining the AI
def ai_choice(cards_drawn, score):
    bust_cards_remaining = 0
    least_card_to_bust = 22 - score
    cards_remaining = 52 - cards_drawn
    if score > 11:
        for key in card_number_value:
            if card_number_value[key] >= least_card_to_bust:
                bust_cards_remaining += cards_left[key]
        probability_next_card_busts = (bust_cards_remaining)/(cards_remaining)
        if probability_next_card_busts > 0.5:
            return 'stick'
        else:
            return 'twist'
    else:
        return 'twist'
```

Not as self-explanatory as Alpha, so I'll explain. Everything should make sense up to the first if statement. Here, I say that anything with a current score less than 11 should twist, because it is impossible to bust. Otherwise, I include a 'for' loop over the dictionary 'card_suit_value.' Here, I have a dictionary of cards and their value, i.e. one entry will be "J: 10". I loop through every entry in that dictionary, and if the value is greater than the least card to cause a bust, then we add the number of cards left of that value to the count of 'bust_cards_remaining.' Note that this is not cheating. Any player could work this value out by looking at their own hand and being decent at mental maths. Then a simple calculating of bust cards left and cards left shows the probability that the next card you will draw will bust you. Finally, if this is greater than a certain number (in this example I have used 0.5), then the AI shall stick, or if not, it shall twist.

The style of play is extremely similar to Alpha 1.0, essentially using a limit, but this limit should be more sophisticated, taking into account individual card values and the number of cards drawn, rather than the total value alone. The style in which they differ is that Magnus will stick when he receives a lot of small cards, because it means big cards are left and he will more likely bust, whereas Alpha will play on. This scenario is very uncommon, perhaps showing less often than the error range. I trialled a few different limits against Alpha 1.0 (17) and it was really difficult to beat. Is it possible that Magnus simply has a worse tactic than Alpha? The results frighteningly suggest that.

As expected, p̂ is between 0.6 and 0.7, but, also as expected, finding this precisely is an enormous task. Furthermore, finding p̂ is somewhat a redundant task, as I know it will be different in Magnus 2.0 when I take into account the ace problem which lacked in Alpha 1.0 and was improved on in Alpha 2.0 (when I try Magnus 1.0 vs Alpha 2.0, Alpha dismantles Magnus). But I shall find p̂ anyhow. I'm not sure if I can prove Magnus 1.0 is better than Alpha 1.0 through statistics, because it will be so close. Additionally, AI's may not be transitive, so a different p̂ may be ideal against Alpha 1.0 to Alpha 2.0, and again to Magnus itself. The p̂ I shall be finding will be the best against itself, then I may explore those other options later.

The best way to do this is the same as in Alpha 1.0, creating an array, but this time being more precise when I can narrow down my range, much like interval bisection, though being a bit more human about it because I need to take into account error ranges (which will be then bane of finding

Magnus 1.0's p̂ value.) Additionally, unlike Alpha, Magnus has a problem when it plays second, because it struggles to think about the cards that have gone to the first player. Instead, I will start the game again after the first player scores, and put the cards back in the deck, then see who wins. This is justified, and in fact necessary, as I am only finding how Magnus should play first.

I started an array table between 0.6 and 0.7, but it seemed like everyone was beating anyone just a bit smaller than it, and then losing to anyone quite a lot smaller than it. I increased my table (which I should have done at the start, I now know that p̂ is between 0.65 and 0.85 after trying my basic excel sheet, my estimation of p̂ was too low.) Turns out I have been using the wrong .py file all day, so I'm having to start my results again, wasted an entire fucking morning's work.

I start the array again of 1,000,000 games per matchup, instead using large intervals which I will zoom in on when I can narrow down the range:

		AI1's limit			
	vs.	0.6	0.65	0.7	0.75
AI2's limit	0.6	AI1 wins: 417431 AI2 wins: 416430 Draws: 166139			
	0.65	AI1 wins: 409058 AI2 wins: 414586 Draws: 176356	AI1 wins: 403910 AI2 wins: 403008 Draws: 193082		
	0.7	AI1 wins: 413711 AI2 wins: 400618 Draws: 185671	AI1 wins: 401887 AI2 wins: 391766 Draws: 206347	AI1 wins: 380956 AI2 wins: 382369 Draws: 236675	
	0.75	AI1 wins: 414978 AI2 wins: 396021 Draws: 189001	AI1 wins: 402165 AI2 wins: 386619 Draws: 211216	AI1 wins: 378130 AI2 wins: 375234 Draws: 246636	AI1 wins: 369395 AI2 wins: 368621 Draws: 261984

I initially carried this on up to 0.85, but it was clear that wasn't necessary. I've used large intervals here, but I'll narrow it later. I may have found a use for the draws too. The draws increase as the limit does, just like with Alpha. So, we should aim for Magnus to have roughly the same number of draws when it plays with p̂ as Alpha does when it plays with 17. Here is my narrowed range:

		AI1's limit				
	vs.	0.64	0.65	0.66	0.67	0.68
AI2's limit	0.64					
	0.65	AI1 wins: 405028 AI2 wins: 405291 Draws: 189681				
	0.66	AI1 wins: 404273 AI2 wins: 405180 Draws: 190547	AI1 wins: 402961 AI2 wins: 403198 Draws: 193841	AI1 wins: 402420 AI2 wins: 402596 Draws: 194984		
	0.67	AI1 wins: 404063 AI2 wins: 404888 Draws: 191049	AI1 wins: 401807 AI2 wins: 403619 Draws: 194574	AI1 wins: 402368 AI2 wins: 402527 Draws: 195105	AI1 wins: 401484 AI2 wins: 402832 Draws: 195684	
	0.68	AI1 wins: 404597 AI2 wins: 397799 Draws: 197604	AI1 wins: 402597 AI2 wins: 395788 Draws: 201615	AI1 wins: 402586 AI2 wins: 395974 Draws: 201440	AI1 wins: 401548 AI2 wins: 396268 Draws: 202184	AI1 wins: 392118 AI2 wins: 392545 Draws: 215337

I spent **so** long working on this yesterday and it was all for naught, literally 6 hours wasted. I was fucking up codes, incurring errors, missing obvious answers and running hundreds of millions of pointless games. Designing this has been a massive fuck up. But I've solved it now. I was thinking the cause of the sharp drop off at 0.68 was like how 18 dropped off quickly, and it is, but for a different reason. After messing around in the excel sheet, I realised something. When you receive 17 in the first pair to start, the probability the next card busts is exactly 0.68. But in my code I said stick if **strictly above** 0.68, so it kept twisting. So, we can say p̂ is 0.68 **(if we change the code to more than or equal to, which we will take Magnus to be like from now on)** or less. Determining how much less is a bit of an issue, because the errors are so close. For example, 0.66 and 0.67 between 10 million games twice: (0.66 wins: 4,024,522, 0.67 wins: 4,022,901, Draws: 1,952,577) and (0.66 wins: 4,022,325, 0.67 wins: 4,025,642, Draws: 1,952,033), so I can't determine p̂ to any good precision. All I can say is that it is at most 0.68 and perhaps a bit less, maybe to as low as 0.66.

So, I will soon trial Alpha 1.0 (17) against Magnus 1.0 (0.68 I'll settle with, where the code is edited to be less than equal to). Whether Magnus will significantly defeat Alpha, I don't know, but I can prove logically that it should do, so the results should be fairly unnecessary.

Magnus was designed for the exact situation for sticking with lots of small cards because the card is likely to be bigger and so bust it. Other than that, Magnus plays identical to Alpha. So the results between the two will be very similar. The theory behind Magnus is perfect. Alpha is too aggressive on 16 and busts when it is obvious only big cards are left in the pack. So, the only way Magnus is different is in hands which sum to 16 (the largest hand which sums to 15 gives a probability of 0.651, and smallest hand to 17 gives 0.68, so only 16 needs to be considered).

Further playing around, it's really hard to get over 0.68 on 16, it could be proven very easily that it requires at least 5 cards, though that proof is not in my interest. Furthermore, the highest probability I can reach on 4 cards is 0.6666 recurring (again, easy to prove mathematically, but not in my interest, I'm interested in perfect AI). These situations are certainly best to stick in than to twist because you're on 16, and I will prove so. I made a realisation of something which became the most developmental aspect of this project: situational gameplay.

Situational Gameplay and the continued development of Mangus

I have designed a code which runs a set number of games, and in this example, if the player is in the situation of a score of 16 with five or more cards, then it saves the number of times this happens and the number of busts. You will see, Magnus will have far less busts, and that is what is important:

Magnus:	Alpha:
Games played: 1,000,000	Games played: 1,000,000
Five card situation games played: 4685	Five card situation games played: 4751
Busts in that situations: 2032	Busts in that situation: 2956

I reran each of these again to see the error on the busts, and it was roughly plus or minus 150, clearly outside the margin of error.

I change the battleground to be similar to this. This kind of five-card-battleground I invented was a fantastic idea. This removes so much error I was having with normal battles, and I can now find p̂ to a greater precision.

Firstly, I need to find the situations that need to be considered. By being open minded and thinking p̂ could be as low as 0.608, this means I need to include situations where we have a score of 15 and 6+ cards, and a score of 16 and 3+ cards. Once I narrow down p̂'s range, I can narrow down the situations. This is easy to edit in the code, so I won't consider it. I have included a table below to explain why I can narrow down ranges the way I do:

		Score							
		14		15		16		17	
		Min	Max	Min	Max	Min	Max	Min	Max
Cards Drawn	2			0.5400	0.5400	0.6000	0.6200	0.6800	0.6800
	3			0.5306	0.5714	0.6122	0.6327		
	4			0.5625	0.5833	0.6250	0.6667		
	5			0.5745	0.5957	0.6383	0.6809		
	6			0.5870	0.6087	0.6522	0.6957		
	7			0.6000	0.6222	0.6889	0.7111		
	8		0.5454	0.6136	0.6364	0.7045	0.7273		
	9			0.6512	0.6512	0.7442	0.7442		

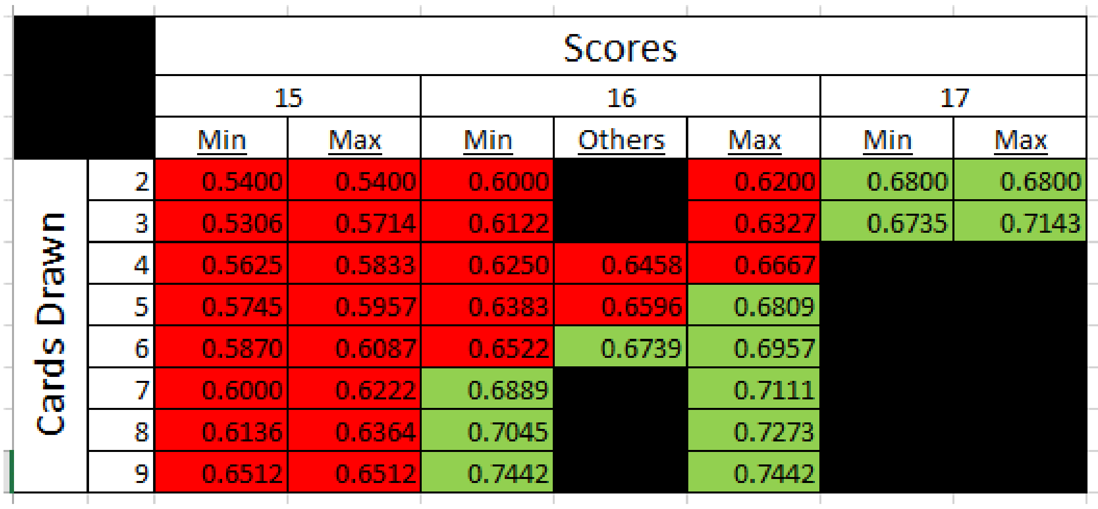
The blacked out situations can either not occur, or aren't important. Firstly, I run 0.63 vs 0.66. Since 0.66 consistently wins (eg. Situation games played: 13130, 0.63 wins: 5437, 0.66 wins: 5922, Draws: 1771), I can narrow down the range to 15 wins and 8 cards, because p̂ is great enough for these situations to play identically to each other and were not situations which made 0.66 beat 0.63. Now, I try 0.64 vs 0.67. Since 0.67 wins (e.g. Situation games played: 130414, 0.64 wins: 51779, 0.67 wins: 52529, Draws: 26106), I can narrow down further, removing 16 wins and 3 cards, and 15 wins and 7/8 cards. Now I can be extremely precise, because I have limited the situations so much. I'm fairly confident p̂ is not near 0.65, so I will remove all 15 score situations from the code, and so I will trial 0.66 vs 0.67 over 10 million games. The results were:

Situation games played: 443759
0.66 wins: 172465
0.67 wins: 174048
Draws: 97246

I reran this again because I'm still unsure on error ranges, and it proved 0.67 won. I can forgot to notice that, since 0.68 is a maximum, I can remove the games 16 of cards 7, 8, or 9. And, after a trial of 0.675 vs 2/3:

Situation games played: 440933
0.675 wins: 172106
2/3 wins: 171311
Draws: 97516

Since 0.675 won, I will remove score 16 and 4 cards from the situations too. Now, things will get gritty. Let's look at the every possible situations:



		Scores						
		15		16			17	
		Min	Max	Min	Others	Max	Min	Max
Cards Drawn	2	0.5400	0.5400	0.6000		0.6200	0.6800	0.6800
	3	0.5306	0.5714	0.6122		0.6327	0.6735	0.7143
	4	0.5625	0.5833	0.6250	0.6458	0.6667		
	5	0.5745	0.5957	0.6383	0.6596	0.6809		
	6	0.5870	0.6087	0.6522	0.6739	0.6957		
	7	0.6000	0.6222	0.6889		0.7111		
	8	0.6136	0.6364	0.7045		0.7273		
	9	0.6512	0.6512	0.7442		0.7442		

Those highlighted in red are ones which we must twist on based on previous matchups, and the green are those above 0.68 which we must stick on. I have included all possible situations in the 16 score in regards to the number of bust cards that have been drawn. So, we are left with one possible situation. Should we stick when we have a score of 16 and 6 cards, 1 of which is a bust card? Let's show only those situations in the code (not hard to do) and do a trial:

Situation games played: 2926
Stick wins: 1344
Twist wins: 1220
Draws: 362

So, it should stick (not a huge win, even after reruns, but the best I can get and it isn't that important). Therefore, 31/46=0.6739… >= p̂ > 0.6667…=2/3. I was quite foolish to think that p̂ would be one value, I should have realised that it would take a range due to the discrete nature of the game.

Finally, I noticed that there was a further problem when having a score of 17 and three cards which were all bust cards. I needed to take this into account too. So, I basically repeated the previous experiment, but with score 17, 3 cards, all busts:

Situation games played: 291
Twist wins: 120
Stick wins: 112
Draws: 59

This result was pretty conclusive after several repeats. So, I can narrow p̂ down even more, saying 31/46=0.6739… > p̂ > 33/49 = 0.6735. So, we have found p̂, and hence completed Magnus.

And now we come to the most disappointing conclusion of the entire project.

Conclusion of Magnus

I ran Magnus with p=0.6735 again Alpha 1.0 (17). I'd ran it plenty of times before with Alpha always winning because I kept making mistakes and spotting them, thinking when I corrected them that Magnus would win. But I can never make Magnus win. It isn't a mistake:

Situation games played: 60855
Magnus wins: 24357
Alpha wins: 27000
Draws: 9498
Magnus busts: 18840
Alpha busts: 19979

The situations played here are the situations in which Alpha and Magnus differ in gameplay. Alpha is better than Magnus. I accepted this when I saw the busts. I said it at the start; that it's possible that that probability isn't perfectly correlated with winning, and I was right, it isn't. This is where I become a true scientist and accept the results and change my theory around it.

How can I explain why I was wrong? The current score is simply more important than the probability the next card will bust you. An unexpected result, but I can't deny it any longer. They're similar, but the score is flat out more important. The result here is a total disproof of my theory, as Alpha can win more than Magnus even though it has more busts (in situations where they differ).

I declare Magnus the greatest coding failure I've ever created. I learnt a lot about coding and pontoon by designing him, but it seems like a waste of time and effort continuing with Magnus. Magnus going second is certainly more interesting, but I don't have the determination to go through with him the way I did with him going first, and I'm quite confident that it wouldn't be as good as Alpha, so why would I go through with it? I spent roughly 20 hours on Magnus and it couldn't beat 4 lines of code in Alpha.

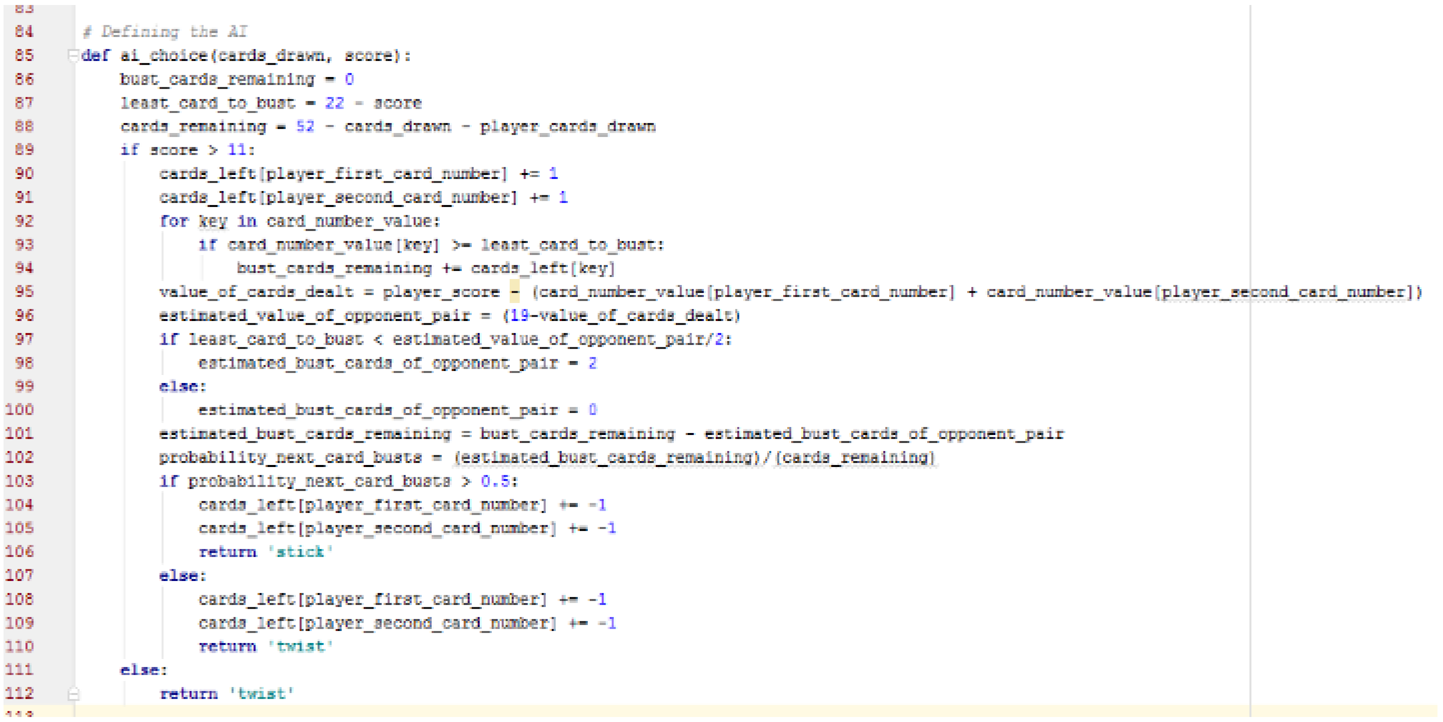
I gained a lot from Magnus though, and I explored specific situations in much more depth, particularly clinch ones (albeit rare). I will probably use these in further developments of Alpha.

Magnus 1.0 - Second

I'm having trouble with the AI playing second. The code should be relatively similar, but I'm having trouble calculating the bust cards remaining. This is **far** harder than I thought. I can't keep the exact code because that would cause my AI to cheat and look into the deck, and essentially, see what cards my opponent has. It's those first two cards that are causing the problem. Those two become 'unavailable cards' of which I can only estimate the value based on how many times they twist and what they receive on the twist and whether they stick or bust. They may be 'bust cards' but, unlike the others, will have a negative contribution to my percent. It's getting to the point where the AI needs accept that its opponent is clever and needs to figure out the strategy of its opponent. I'm going to have to assume that the opponent has an 'average card hand' of 19 and work out the expected value of the two cards from that, and take this into account.

A second problem, a more straightforward problem, is that I can't retrieve the value of the first two cards dealt to the opponent. I'm going to have to edit the code itself to save these values because the task is impossible otherwise.

Get ready for this next code, because it's incredibly wordy and took me a long time to figure out:



```
# Defining the AI
def ai_choice(cards_drawn, score):
    bust_cards_remaining = 0
    least_card_to_bust = 22 - score
    cards_remaining = 52 - cards_drawn - player_cards_drawn
    if score > 11:
        cards_left[player_first_card_number] += 1
        cards_left[player_second_card_number] += 1
        for key in card_number_value:
            if card_number_value[key] >= least_card_to_bust:
                bust_cards_remaining += cards_left[key]
        value_of_cards_dealt = player_score - (card_number_value[player_first_card_number] + card_number_value[player_second_card_number])
        estimated_value_of_opponent_pair = (19-value_of_cards_dealt)
        if least_card_to_bust < estimated_value_of_opponent_pair/2:
            estimated_bust_cards_of_opponent_pair = 2
        else:
            estimated_bust_cards_of_opponent_pair = 0
        estimated_bust_cards_remaining = bust_cards_remaining - estimated_bust_cards_of_opponent_pair
        probability_next_card_busts = (estimated_bust_cards_remaining)/(cards_remaining)
        if probability_next_card_busts > 0.5:
            cards_left[player_first_card_number] += -1
            cards_left[player_second_card_number] += -1
            return 'stick'
        else:
            cards_left[player_first_card_number] += -1
            cards_left[player_second_card_number] += -1
            return 'twist'
    else:
        return 'twist'
```

Again, everything should be understandable to the first if statement. Then, it checks the score, and twists for anything lower than 12 for obvious reasons, and starts its algorithm if not. Firstly, it adds back the first and second cards (initial pair) drawn by the player into a temporary deck of the pair and whatever else is in the deck. Then, it adds up all the bust cards that exist in this temporary deck (which remains not cheating, any player could do this). Then, it calculates the value of the cards that have been shown from twisting by the player. Then, it estimates the value of the pair it could not see by using an average card hand of 19. Then, it estimates the number of bust cards the opponent has through a very naïve method of assuming the opponent has a pair and finding if they're both busts or not. Then, it estimates the bust cards remaining in the real life deck by taking the estimated number of bust cards in the pair away from the bust cards in the temporary deck (with them in). Then it calculates the probability by dividing the estimated bust cards remaining by the number of cards remaining in the real deck. Then, if this probability is above the defined value (here 0.5), then it takes the player's initial pair out of the temporary deck and stick, or if it is less, it take's the pair out and twists.

Make sense? I struggle too, and I wrote it. I have no doubt that this code will cause me grief when I start battling AI's. It causes me grief and I haven't even battled AI's.

It's clear to me where Magnus 2.0 will focus on. Lines 96-100 are ***by far*** the most important lines in the code. That 19 representing the average hand value of the opponent is plucked out of thin air somewhat, and I will be able to change this number by versing AI's against each other. Secondly, calculating the estimated number of bust cards of the opponent is done awfully. I mean seriously badly. It's so badly done that it is perhaps worse than just assuming the opponent has 1 bust card. It needs rewriting ASAP, but it's quite a task. The best way to do it is to do it exhaustively, i.e. write out the estimated number of bust cards if your least card to bust is 5 and there estimated pair is 7. This is the most important problem to work on after I'm sure Magnus calculates correctly. But this is a huge task in itself and the code needed filling.

Finally, I need to check Magnus is calculating correctly. After two runs, they were both correct as my paper value. Thank fucking Christ for that because I'd have no clue where to start checking my code.

Now, I need to spend some time working on the maths of estimated how many bust cards the opponent has based on their estimated card value and my least card to bust.

I've just finished writing Magnus 1.0 first, and concluded Magnus is worse than Alpha. So I've come to the conclusion of not going ahead with this project. The probability calculated would be intriguing, but it wouldn't correlate to wins.

I spent many hours designing Magnus to play first, and designing Magnus to play second would have been interesting, but would have taken twice as long as the first, and wouldn't have played as well as Alpha, so I have decided not to continue.

So, I'm sad to say, this project will remain unfinished. Magnus playing second has some very interesting parts to it, such estimating the average hand of the player. I will try improving Alpha or creating a new AI, and I will be using elements of Magnus in them. For example, estimating the average hand of the opponent based on the number of cards they receive and what cards they receive, as well as possibly sticking when reaching a certain number of cards on a certain score. Either way, Magnus is being discontinued.

Veritas 1.0

Alpha was my Greek model, Magnus my Scandinavian, so I needed a Latin name for this one. Hence, Veritas, Latin for truth, has been named. It is also going to be a phantom project, by which I mean, I don't have the time nor effort to continue with AI designing, it's fair to say I'm sick of Pontoon. Perhaps I'll continue it one day, but I'm not currently interesting in doing so.

Perhaps the best thing to come out of Magnus was the idea of situational gameplay. Ideally, I would design Veritas by trialling every single situation using the situational battleground, and then trying alternative AI's sticking and twisting in such a situation and seeing who won the most, then making Veritas be the winner of those. However, that could take fucking days, and I'd like to wrap this project up soon. Instead, I would have confidently said that anything below 15 is twist-worthy and anything above 18 is stick-worthy. Then I would analyse the results in between to see if any are obscure cases not picked up by Alpha.

What are these situations though? Firstly, one aspect is number of cards drawn. Secondly, number of bust cards remaining (though as highlighted in Magnus, this is likely to be unimportant information, as may cards drawn actually). A bit of combinatorics tells me this gives us **a lot** of situations, roughly 100, to consider. So I'm going to have to be clever and cut these situations down as I go along.

How will I use the ace to my advantage here? I'm not quite sure. My current method planned doesn't take into account this option at all. I could rip off Alpha 2.0, but where's the idea of situational gameplay in that? I've drawn a blank here. This is a very important aspect to work on too, as it improved Alpha 1.0 to 2.0 massively, so there's clearly potential here.

I would also liked to have used the idea of estimating the opponent's hand, as discussed in Magnus going second. This would have been a matter of exhausting every possibility and using statistics, and would have been a long, boring mathematical problem.

Perhaps one day I'll build on this AI, but testing all these trials and calculating opponents' expected hands is plain boring.

Concluding Remarks

I have included the menu in the list of codes. This is basically my final user interface, where the player will verse Alpha 2.0. I wanted to make a score tally, but I'm somewhat bored of this, as I said, I'm ready to move on. It doesn't actually work because once imported, it runs it and then doesn't restart when run again. Again, I don't have the determination to work on import statements to get this to work, but it's obvious how it would work.

I've done a lot of work on this project, far more than I intended. I have achieved my initial aims of designing normal game interaction, finding a limit and optimal bust percentage. Furthering the AI became more difficult and I lost interest in constantly checking for errors in such tedious code with very slightly different alterations. I think it's time I need to move on to a different project now, though I've learnt a lot about Pontoon from this, and a lot of python too (though perhaps not so much towards the end). If I want to expand my python knowledge more, this certainly isn't the project to do so.

List of Codes

The list of codes that are referenced are all included here. They have fairly random AI’s/situations/etc. in them, but include comments which will make them all understandable.

AI vs. Player

```python
__author__ = 'Joe'
from random import randint

# Creating the initial setup
cards_left = {'2': 4, '3': 4, '4': 4, '5': 4, '6': 4, '7': 4, '8': 4, '9': 4, '10': 4, 'J': 4, 'Q': 4, 'K': 4, 'A': 4}
Two = ['H', 'D', 'C', 'S']
Three = ['H', 'D', 'C', 'S']
Four = ['H', 'D', 'C', 'S']
Five = ['H', 'D', 'C', 'S']
Six = ['H', 'D', 'C', 'S']
Seven = ['H', 'D', 'C', 'S']
Eight = ['H', 'D', 'C', 'S']
Nine = ['H', 'D', 'C', 'S']
Ten = ['H', 'D', 'C', 'S']
Jack = ['H', 'D', 'C', 'S']
Queen = ['H', 'D', 'C', 'S']
King = ['H', 'D', 'C', 'S']
Ace = ['H', 'D', 'C', 'S']
# A more sophisticated deck introduced to aid finding and fixing card suits
card_deck = {'2': Two, '3': Three, '4': Four, '5': Five, '6': Six, '7': Seven, '8': Eight, '9': Nine, '10': Ten, 'J': Jack, 'Q': Queen, 'K': King, 'A': Ace}
# Defines card values (Ace contributes as 1 to current score to prevent avoidable busts)
card_number_value = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'J': 10, 'Q': 10, 'K': 10, 'A': 1}
# Creates a basic list of card numbers
cards_list = list(card_deck.keys())

# Defining the AI
def ai_choice(score, aces):
    if score >= 17:
        return 'stick'
    else:
        if aces >= 1:
            if 21 >= score >= 19:
                return 'stick'
            else:
                return 'twist'
        else:
            return 'twist'

def ai_ace_choice(score):
        if score+10 <= 21:
            return '11'
        else:
            return '1'

# AI starts to play
print("The AI will play first. It will show all the cards it receives and the choices it makes.")
AI_score = 0
AI_cards_drawn = 0
choice = 'twist'                                                                                                        # To enter the while loop qualifier
while choice == 'twist' or AI_cards_drawn < 2:

    # Randomly generating the card number
    i = randint(0, int(len(cards_list))-1)                                                                              # Generates random number for drawn card's index
    card_drawn_number = cards_list[i]                                                                                   # Defines drawn card's number
    while cards_left[card_drawn_number] == 0:                                                                           # Checks if there are any suits left of that number
```

```
        i = randint(0, int(len(cards_list))-1)                          # If not, regenerates new random number if none left
        card_drawn_number = cards_list[i]                               # And redefines drawn card's number
    cards_left[card_drawn_number] += -1                                 # Decreases drawn card's count by 1

    # Finding the card suit
    for key in card_deck:                                               # For every number in the card deck
        if card_drawn_number == key:                                    # If the card's number matches it, then
            j = randint(0, int(len(card_deck[key]))-1)                  # Generate a random number to be its index in its suit list
            card_drawn_suit = card_deck[key][j]                         # Define drawn card's suit
            card_deck[key].remove(card_drawn_suit)                      # Remove that suit from its suit list
            AI_score += card_number_value[key]                          # Increase current score by correct amount

    AI_cards_drawn += 1                                                 # Increases cards drawn tally by 1

    # Displaying or saving cards
    if AI_cards_drawn == 1:                                             # For first card drawn
        AI_first_card = str(card_drawn_number+card_drawn_suit)          # Don't display but save
    elif AI_cards_drawn == 2:                                           # For second card drawn
        AI_second_card = str(card_drawn_number+card_drawn_suit)         # Don't display but save
        print("The AI has received its first two cards.")               # Display that AI has received its cards
    else:                                                               # For any other card drawn
        print(card_drawn_number+card_drawn_suit)                        # Displays the drawn card (ignore error, it's always defined)
    AI_aces_drawn = 4-len(Ace)                                          # Defines how many aces player has drawn (ignore error)


    # If statement to check if busted and asks stick/twist
    if AI_score > 21:
        print("The AI has bust!")
        print("")
        break
    else:
        if AI_cards_drawn >= 2:                                         # If statement to ensure at least two cards always drawn
            choice = ai_choice(AI_score, AI_aces_drawn)                 # Input of stick or twist (error catching not needed)
            print("The AI has decided to "+choice)                      # Shows the AI's choice of stick or twist
else:
    # Finished drawing cards
    for n in range(1,5,1):                                              # Makes n equal 1, then 2, 3, 4
        if AI_aces_drawn >= n:                                          # If you have N or more aces
            ace_choice = ai_ace_choice(AI_score)                        # Input value wanted (error catching not needed)
            if ace_choice == '11':                                      # If '11' selecting
                AI_score += 10                                          # Add 10 to the original 1 score (to make 11)
            elif ace_choice == '1':                                     # If '1' selecting
                pass                                                    # Add 0 to the original 1 score (to make 1)
        else:                                                           # If you don't have N or more aces
            break                                                       # Don't bother trying higher N's
    print("")
    print("The AI has finished drawing cards")                          # Displays that AI has finished
    print("")

# Player starts the game
print("It is now your turn. Your first two cards are:")
player_score = 0
player_cards_drawn = 0
choice = 'twist'                                                        # To enter the while loop qualifier
while choice == 'twist' or player_cards_drawn < 2:

    # Randomly generating the card number
    i = randint(0, int(len(cards_list))-1)                              # Generates random number for drawn card's index
    card_drawn_number = cards_list[i]                                   # Defines drawn card's number
    while cards_left[card_drawn_number] == 0:                           # Checks if there are any suits left of that number
        i = randint(0, int(len(cards_list))-1)                          # If not, regenerates new random number if none left
        card_drawn_number = cards_list[i]                               # And redefines drawn card's number
    cards_left[card_drawn_number] += -1                                 # Decreases drawn card's count by 1

    # Finding the card suit
    for key in card_deck:                                               # For every number in the card deck
```

```
        if card_drawn_number == key:                                                    # If the card's number matches it, then
            j = randint(0, int(len(card_deck[key]))-1)                                  # Generate a random number to be its index in its suit list
            card_drawn_suit = card_deck[key][j]                                         # Define drawn card's suit
            card_deck[key].remove(card_drawn_suit)                                      # Remove that suit from its suit list
            player_score += card_number_value[key]                                      # Increase current score by correct amount

    player_cards_drawn += 1                                                             # Increases cards drawn tally by 1
    print(card_drawn_number+card_drawn_suit)                                            # Displays the drawn card (ignore error, it's always defined)
    player_aces_drawn = 4-AI_aces_drawn-len(Ace)                                        # Defines how many aces player has drawn


    # If statement to check if busted and asks stick/twist
    if player_score > 21:
        print("You have bust!")
        break
    else:
        if player_cards_drawn >= 2:                                                     # If statement to ensure at least two cards always drawn
            choice = str(input("Stick or twist: ")).lower()                             # Input of stick or twist
            while choice != 'stick' and choice != 'twist':                              # Error catching in stick or twist answer
                choice = str(input("That's not a valid answer, type 'stick' or 'twist' exactly. Enter again: ")).lower()
else:
    # Finished drawing cards
    for n in range(1,5,1):                                                              # Makes n equal 1, then 2, 3, 4
        if player_aces_drawn >= n:                                                      # If you have N or more aces
            ace_choice = str(input("Do you want your ace to count as 1 or 11? Enter '1' or '11': "))    # Input value wanted
            while ace_choice != '1' and ace_choice != '11':                             # Error catching
                ace_choice = str(input("That's not a valid answer, type '1' or '11' exactly. Enter again: "))
            if ace_choice == '11':                                                      # If '11' selecting
                player_score += 10                                                      # Add 10 to the original 1 score (to make 11)
            elif ace_choice == '1':                                                     # If '1' selecting
                pass                                                                    # Add 0 to the original 1 score (to make 1)
        else:                                                                           # If you don't have N or more aces
            break                                                                       # Don't bother trying higher N's
    print("")
    print("You have finally stuck. Your score was: "+str(player_score))                 # Displays player's score

# Final results
print("")
if player_score > 21:                                                                   # Make player score 'Bust!' if bust
    player_score = "Bust!"
if AI_score > 21:                                                                       # Make AI score 'Bust!' if bust
    AI_score = "Bust!"
print("The cards this AI first received were: " + AI_first_card + " and " + AI_second_card)    # Reveals AI's first and second cards
print("The final results were:")                                                        # Display results
print("You: " + str(player_score))
print("AI: " + str(AI_score))
print("")
if player_score == AI_score:                                                            # If equal sscores
    print("You drew.")
elif player_score == "Bust!":                                                           # If you bust (and AI doesnt, caught in previous if statement)
    print("The AI wins.")
elif AI_score == "Bust!":                                                               # If AI bust (and player doesnt, caught in first if statement)
    print("You win!")
elif AI_score > player_score:                                                           # If neither bust and AI scores higher
    print("The AI wins.")
elif player_score > AI_score:                                                           # If neither bust and player scores higher
    print("You win!")
```

Player vs AI

```python
__author__ = 'Joe'
from random import randint

# Creating the initial setup
cards_left = {'2': 4, '3': 4, '4': 4, '5': 4, '6': 4, '7': 4, '8': 4, '9': 4, '10': 4, 'J': 4, 'Q': 4, 'K': 4, 'A': 4}
Two = ['H', 'D', 'C', 'S']
Three = ['H', 'D', 'C', 'S']
Four = ['H', 'D', 'C', 'S']
Five = ['H', 'D', 'C', 'S']
Six = ['H', 'D', 'C', 'S']
Seven = ['H', 'D', 'C', 'S']
Eight = ['H', 'D', 'C', 'S']
Nine = ['H', 'D', 'C', 'S']
Ten = ['H', 'D', 'C', 'S']
Jack = ['H', 'D', 'C', 'S']
Queen = ['H', 'D', 'C', 'S']
King = ['H', 'D', 'C', 'S']
Ace = ['H', 'D', 'C', 'S']
# A more sophisticated deck introduced to aid finding and fixing card suits
card_deck = {'2': Two, '3': Three, '4': Four, '5': Five, '6': Six, '7': Seven, '8': Eight, '9': Nine, '10': Ten, 'J': Jack, 'Q': Queen, 'K': King, 'A': Ace}
# Defines card values (Ace contributes as 1 to current score to prevent avoidable busts)
card_number_value = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'J': 10, 'Q': 10, 'K': 10, 'A': 1}
# Creates a basic list of card numbers
cards_list = list(card_deck.keys())

# Player starts the game
print("You will start first. Here are your first two cards:")
player_score = 0
player_cards_drawn = 0
choice = 'twist'                                                                                                        # To enter the while loop qualifier
while choice == 'twist' or player_cards_drawn < 2:

    # Randomly generating the card number
    i = randint(0, int(len(cards_list))-1)                                                                              # Generates random number for drawn card's index
    card_drawn_number = cards_list[i]                                                                                   # Defines drawn card's number
    while cards_left[card_drawn_number] == 0:                                                                           # Checks if there are any suits left of that number
        i = randint(0, int(len(cards_list))-1)                                                                          # If not, regenerates new random number if none left
        card_drawn_number = cards_list[i]                                                                               # And redefines drawn card's number
    cards_left[card_drawn_number] += -1                                                                                 # Decreases drawn card's count by 1

    # Finding the card suit
    for key in card_deck:                                                                                               # For every number in the card deck
        if card_drawn_number == key:                                                                                    # If the card's number matches it, then
            j = randint(0, int(len(card_deck[key]))-1)                                                                  # Generate a random number to be its index in its suit list
            card_drawn_suit = card_deck[key][j]                                                                         # Define drawn card's suit
            card_deck[key].remove(card_drawn_suit)                                                                      # Remove that suit from its suit list
            player_score += card_number_value[key]                                                                      # Increase current score by correct amount

    player_cards_drawn += 1                                                                                             # Increases cards drawn tally by 1
    print(card_drawn_number+card_drawn_suit)                                                                            # Displays the drawn card (ignore error, it is always defined)
    player_aces_drawn = 4-len(Ace)                                                                                      # Defines how many aces player has drawn
    if player_cards_drawn == 1:                                                                                         # For first card drawn (this part is so the AI Magnus will work)
        player_first_card_number = card_drawn_number                                                                    # Save the number of the card
    elif player_cards_drawn == 2:                                                                                       # For second card drawn
        player_second_card_number = card_drawn_number                                                                   # Save the number of the card



    # If statement to check if busted and asks stick/twist
    if player_score > 21:
        print("You have bust!")
        print("")
        break
    else:
        if player_cards_drawn >= 2:                                                                                     # If statement to ensure at least two cards always drawn
            choice = str(input("Stick or twist: ")).lower()                                                             # Input of stick or twist
            while choice != 'stick' and choice != 'twist':                                                              # Error catching in stick or twist answer
                choice = str(input("That's not a valid answer, type 'stick' or 'twist' exactly. Enter again: ")).lower()
else:
    # Finished drawing cards
    for n in range(1,5,1):                                                                                              # Makes n equal 1, then 2, 3, 4
        if player_aces_drawn >= n:                                                                                      # If you have N or more aces
            ace_choice = str(input("Do you want your ace to count as 1 or 11? Enter '1' or '11': "))                    # Input value wanted
            while ace_choice != '1' and ace_choice != '11':                                                             # Error catching
                ace_choice = str(input("That's not a valid answer, type '1' or '11' exactly. Enter again: "))
            if ace_choice == '11':                                                                                      # If '11' selecting
                player_score += 10                                                                                      # Add 10 to the original 1 score (to make 11)
            elif ace_choice == '1':                                                                                     # If '1' selecting
                pass                                                                                                    # Add 0 to the original 1 score (to make 1)
        else:                                                                                                           # If you don't have N or more aces
            break                                                                                                       # Don't bother trying higher N's
    print("")
    print("You have finally stuck. Your score was: "+str(player_score))                                                 # Displays player's score
    print("")

# Defining the AI
def ai_choice(score, aces):
    if player_score > 21:
        return 'stick'
    else:
        if score >= 17:
            return 'stick'
        else:
            if aces >= 1:
                if 21 >= score >= 19:
                    return 'stick'
                else:
                    return 'twist'
            else:
                return 'twist'

def ai_ace_choice(score):
        if score+10 <= 21:
            return '11'
        else:
            return '1'

# AI starts to play
print("The AI will now play. It will show all the cards it receives and the choices it makes. Here is its first two cards:")
AI_score = 0
AI_cards_drawn = 0
choice = 'twist'                                                                                                        # To enter the while loop qualifier
while choice == 'twist' or AI_cards_drawn < 2:

    # Randomly generating the card number
    i = randint(0, int(len(cards_list))-1)                                                                              # Generates random number for drawn card's index
    card_drawn_number = cards_list[i]                                                                                   # Defines drawn card's number
    while cards_left[card_drawn_number] == 0:                                                                           # Checks if there are any suits left of that number
        i = randint(0, int(len(cards_list))-1)                                                                          # If not, regenerates new random number if none left
        card_drawn_number = cards_list[i]                                                                               # And redefines drawn card's number
    cards_left[card_drawn_number] += -1                                                                                 # Decreases drawn card's count by 1

    # Finding the card suit
    for key in card_deck:                                                                                               # For every number in the card deck
        if card_drawn_number == key:                                                                                    # If the card's number matches it, then
            j = randint(0, int(len(card_deck[key]))-1)                                                                  # Generate a random number to be its index in its suit list
            card_drawn_suit = card_deck[key][j]                                                                         # Define drawn card's suit
            card_deck[key].remove(card_drawn_suit)                                                                      # Remove that suit from its suit list
            AI_score += card_number_value[key]                                                                            # Increase current score by correct amount
    AI_cards_drawn += 1                                                                                                                  # Increases cards drawn tally by 1
    print(card_drawn_number+card_drawn_suit)                                                                                             # Displays the drawn card (ignore error, it is always defined)
    AI_aces_drawn = 4-player_aces_drawn-len(Ace)                                                                                         # Defines how many aces player has drawn (ignore error)

    # If statement to check if busted and asks stick/twist
    if AI_score > 21:
        print("The AI has bust!")
        break
    else:
        if AI_cards_drawn >= 2:                                                                                                          # If statement to ensure at least two cards always drawn
            choice = ai_choice(AI_score, AI_aces_drawn)                                                                                  # Input of stick or twist (error catching not needed)
            print("The AI has decided to "+choice)                                                                                       # Shows the AI's choice of stick or twist
else:
    # Finished drawing cards
    for n in range(1,5,1):                                                                                                               # Makes n equal 1, then 2, 3, 4
        if AI_aces_drawn >= n:                                                                                                           # If you have N or more aces
            ace_choice = ai_ace_choice(AI_score)                                                                                         # Input value wanted (error catching not needed)
            if ace_choice == '11':                                                                                                       # If '11' selecting
                AI_score += 10                                                                                                           # Add 10 to the original 1 score (to make 11)
            elif ace_choice == '1':                                                                                                      # If '1' selecting
                pass                                                                                                                     # Add 0 to the original 1 score (to make 1)
        else:                                                                                                                            # If you don't have N or more aces
            break                                                                                                                        # Don't bother trying higher N's
    print("")
    print("The AI has finally stuck. Its score was: "+str(AI_score))                                                                     # Displays AI's score

# Final results
print("")
if player_score > 21:                                                                                                                    # Make player score 'Bust!' if bust
    player_score = "Bust!"
if AI_score > 21:                                                                                                                        # Make AI score 'Bust!' if bust
    AI_score = "Bust!"
print("The final results were:")                                                                                                         # Display results
print("You: " + str(player_score))
print("AI: " + str(AI_score))
print("")
if player_score == AI_score:                                                                                                             # If equal sscores
    print("You drew.")
elif player_score == "Bust!":                                                                                                            # If you bust (and AI doesnt, caught in previous if statement)
    print("The AI wins.")
elif AI_score == "Bust!":                                                                                                                # If AI bust (and player doesnt, caught in first if statement)
    print("You win!")
elif AI_score > player_score:                                                                                                            # If neither bust and AI scores higher
    print("The AI wins.")
elif player_score > AI_score:                                                                                                            # If neither bust and player scores higher
    print("You win!")
```

Battleground

```
__author__ = 'Joe'
from random import randint

games_playing = int(input("How many games are to be played: "))
games_played = 0
AI1_wins = 0
AI2_wins = 0
draws = 0
AI1_busts = 0
AI2_busts = 0

while games_played < games_playing:
    # Creating the initial setup
    cards_left = {'2': 4, '3': 4, '4': 4, '5': 4, '6': 4, '7': 4, '8': 4, '9': 4, '10': 4, 'J': 4, 'Q': 4, 'K': 4, 'A': 4}
    Two = ['H', 'D', 'C', 'S']
    Three = ['H', 'D', 'C', 'S']
    Four = ['H', 'D', 'C', 'S']
    Five = ['H', 'D', 'C', 'S']
    Six = ['H', 'D', 'C', 'S']
    Seven = ['H', 'D', 'C', 'S']
    Eight = ['H', 'D', 'C', 'S']
    Nine = ['H', 'D', 'C', 'S']
    Ten = ['H', 'D', 'C', 'S']
    Jack = ['H', 'D', 'C', 'S']
    Queen = ['H', 'D', 'C', 'S']
    King = ['H', 'D', 'C', 'S']
    Ace = ['H', 'D', 'C', 'S']
    # A more sophisticated deck introduced to aid finding and fixing card suits
    card_deck = {'2': Two, '3': Three, '4': Four, '5': Five, '6': Six, '7': Seven, '8': Eight, '9': Nine, '10': Ten, 'J': Jack, 'Q': Queen, 'K': King, 'A': Ace}
    # Defines card values (Ace contributes as 1 to current score to prevent avoidable busts)
    card_number_value = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'J': 10, 'Q': 10, 'K': 10, 'A': 1}
    # Creates a basic list of card numbers
    cards_list = list(card_deck.keys())

    # Defining AI1
    def ai1_choice(cards_drawn, score):
        bust_cards_remaining = 0
        least_card_to_bust = 22 - score
        cards_remaining = 52 - cards_drawn
        if score > 11:
            for key in card_number_value:
                if card_number_value[key] >= least_card_to_bust:
                    bust_cards_remaining += cards_left[key]
            probability_next_card_busts = (bust_cards_remaining)/(cards_remaining)
            if probability_next_card_busts >= 0.6734:
                return 'stick'
            else:
                return 'twist'
        else:
            return 'twist'

    def ai1_ace_choice(score):
        if score+10 <= 21:
            return '11'
        else:
            return '1'

    # AI1 starts to play
    AI1_score = 0
    AI1_cards_drawn = 0
    choice = 'twist'                                                                                           # To enter the while loop qualifier
    while choice == 'twist' or AI1_cards_drawn < 2:
```

```
    # Randomly generating the card number
    i = randint(0, int(len(cards_list))-1)                                                                       # Generates random number for drawn card's index
    card_drawn_number = cards_list[i]                                                                            # Defines drawn card's number
    while cards_left[card_drawn_number] == 0:                                                                    # Checks if there are any suits left of that number
        i = randint(0, int(len(cards_list))-1)                                                                   # If not, regenerates new random number if none left
        card_drawn_number = cards_list[i]                                                                        # And redefines drawn card's number
    cards_left[card_drawn_number] += -1                                                                          # Decreases drawn card's count by 1

    # Finding the card suit
    for key in card_deck:                                                                                        # For every number in the card deck
        if card_drawn_number == key:                                                                             # If the card's number matches it, then
            j = randint(0, int(len(card_deck[key]))-1)                                                           # Generate a random number to be its index in its suit list
            card_drawn_suit = card_deck[key][j]                                                                  # Define drawn card's suit
            card_deck[key].remove(card_drawn_suit)                                                               # Remove that suit from its suit list
            AI1_score += card_number_value[key]                                                                  # Increase current score by correct amount

    AI1_cards_drawn += 1                                                                                         # Increases cards drawn tally by 1
    AI1_aces_drawn = 4-len(Ace)                                                                                  # Defines how many aces player has drawn (ignore error)
    if AI1_cards_drawn == 1:                                                                                     # For first card drawn (this part is so the AI Magnus will work)
        AI1_first_card_number = card_drawn_number                                                                # Save the number of the card
    elif AI1_cards_drawn == 2:                                                                                   # For second card drawn
        AI1_second_card_number = card_drawn_number                                                               # Save the number of the card

    # If statement to check if busted and asks stick/twist
    if AI1_score > 21:
        break
    else:
        if AI1_cards_drawn >= 2:                                                                                 # If statement to ensure at least two cards always drawn
            choice = ai1_choice(AI1_cards_drawn, AI1_score)                                                      # Input of stick or twist (error catching not needed)
else:
    # Finished drawing cards
    for n in range(1,5,1):                                                                                       # Makes n equal 1, then 2, 3, 4
        if AI1_aces_drawn >= n:                                                                                  # If you have N or more aces
            ace_choice = ai1_ace_choice(AI1_score)                                                               # Input value wanted (error catching not needed)
            if ace_choice == '11':                                                                               # If '11' selecting
                AI1_score += 10                                                                                  # Add 10 to the original 1 score (to make 11)
            elif ace_choice == '1':                                                                              # If '1' selecting
                pass                                                                                             # Add 0 to the original 1 score (to make 1)
        else:                                                                                                    # If you don't have N or more aces
            break                                                                                                # Don't bother trying higher N's

# Creating the initial setup
cards_left = {'2': 4, '3': 4, '4': 4, '5': 4, '6': 4, '7': 4, '8': 4, '9': 4, '10': 4, 'J': 4, 'Q': 4, 'K': 4, 'A': 4}
Two = ['H', 'D', 'C', 'S']
Three = ['H', 'D', 'C', 'S']
Four = ['H', 'D', 'C', 'S']
Five = ['H', 'D', 'C', 'S']
Six = ['H', 'D', 'C', 'S']
Seven = ['H', 'D', 'C', 'S']
Eight = ['H', 'D', 'C', 'S']
Nine = ['H', 'D', 'C', 'S']
Ten = ['H', 'D', 'C', 'S']
Jack = ['H', 'D', 'C', 'S']
Queen = ['H', 'D', 'C', 'S']
King = ['H', 'D', 'C', 'S']
Ace = ['H', 'D', 'C', 'S']
# A more sophisticated deck introduced to aid finding and fixing card suits
card_deck = {'2': Two, '3': Three, '4': Four, '5': Five, '6': Six, '7': Seven, '8': Eight, '9': Nine, '10': Ten, 'J': Jack, 'Q': Queen, 'K': King, 'A': Ace}
# Defines card values (Ace contributes as 1 to current score to prevent avoidable busts)
card_number_value = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'J': 10, 'Q': 10, 'K': 10, 'A': 1}
# Creates a basic list of card numbers
cards_list = list(card_deck.keys())

# Defining AI2
def ai2_choice(score):
    if score >= 17:
```

```python
        return 'stick'
    else:
        return 'twist'

def ai2_ace_choice(score):
    if score+10 <= 21:
        return '11'
    else:
        return '1'

# AI2 starts the game
AI2_score = 0
AI2_cards_drawn = 0
choice = 'twist'                                                                                                    # To enter the while loop qualifier
while choice == 'twist' or AI2_cards_drawn < 2:

    # Randomly generating the card number
    i = randint(0, int(len(cards_list))-1)                                                                          # Generates random number for drawn card's index
    card_drawn_number = cards_list[i]                                                                               # Defines drawn card's number
    while cards_left[card_drawn_number] == 0:                                                                       # Checks if there are any suits left of that number
        i = randint(0, int(len(cards_list))-1)                                                                      # If not, regenerates new random number if none left
        card_drawn_number = cards_list[i]                                                                           # And redefines drawn card's number
    cards_left[card_drawn_number] += -1                                                                             # Decreases drawn card's count by 1

    # Finding the card suit
    for key in card_deck:                                                                                           # For every number in the card deck
        if card_drawn_number == key:                                                                                # If the card's number matches it, then
            j = randint(0, int(len(card_deck[key]))-1)                                                              # Generate a random number to be its index in its suit list
            card_drawn_suit = card_deck[key][j]                                                                     # Define drawn card's suit
            card_deck[key].remove(card_drawn_suit)                                                                  # Remove that suit from its suit list
            AI2_score += card_number_value[key]                                                                     # Increase current score by correct amount

    AI2_cards_drawn += 1                                                                                            # Increases cards drawn tally by 1
    AI2_aces_drawn = 4-len(Ace)                                                                                     # Defines how many aces player has drawn

    # If statement to check if busted and asks stick/twist
    if AI2_score > 21:
        break
    else:
        if AI2_cards_drawn >= 2:                                                                                    # If statement to ensure at least two cards always drawn
            choice = ai2_choice(AI2_score)                                                                          # Input of stick or twist
else:
    # Finished drawing cards
    for n in range(1,5,1):                                                                                          # Makes n equal 1, then 2, 3, 4
        if AI2_aces_drawn >= n:                                                                                     # If you have N or more aces
            ace_choice = ai2_ace_choice(AI2_score)                                                                  # Input value wanted
            if ace_choice == '11':                                                                                  # If '11' selecting
                AI2_score += 10                                                                                     # Add 10 to the original 1 score (to make 11)
            elif ace_choice == '1':                                                                                 # If '1' selecting
                pass                                                                                                # Add 0 to the original 1 score (to make 1)
        else:                                                                                                       # If you don't have N or more aces
            break                                                                                                   # Don't bother trying higher N's

# Final results
if AI1_score > 21:                                                                                                  # Make player score 'Bust!' if bust
    AI1_score = "Bust!"
    AI1_busts += 1
if AI2_score > 21:                                                                                                  # Make AI score 'Bust!' if bust
    AI2_score = "Bust!"
    AI2_busts += 1
if AI1_score == AI2_score:                                                                                          # If equal sscores
    draws += 1
elif AI1_score == "Bust!":                                                                                          # If you bust (and AI doesnt, caught in previous if statement)
    AI2_wins += 1
elif AI2_score == "Bust!":                                                                                          # If AI bust (and player doesnt, caught in first if statement)
    AI1_wins += 1
    elif AI2_score > AI1_score:                                                                                    # If neither bust and AI scores higher
        AI2_wins += 1
    elif AI1_score > AI2_score:                                                                                    # If neither bust and player scores higher
        AI1_wins += 1
    games_played += 1                                                                                              # Add 1 to games played

print("Games Played: " + str(games_played))
print("AI1 wins: " + str(AI1_wins))
print("AI2 wins: " + str(AI2_wins))
print("Draws: " + str(draws))
print("AI1 busts: " + str(AI1_busts))
print("AI2 busts: " + str(AI2_busts))
```

Alpha 1.0

```
# Defining AI1
def alpha1_choice(score):
    if score >= 17:
        return 'stick'
    else:
        return 'twist'
def alpha1_ace_choice(score):
    if score+10 <= 21:
        return '11'
    else:
        return '1'
```

Alpha 2.0

```
# Defining AI1
def alpha2_choice(score, aces):
    if score >= 17:
        return 'stick'
    else:
        if aces >= 1:
            if 21 >= score >= 19:
                return 'stick'
            else:
                return 'twist'
        else:
            return 'twist'
def alpha2_ace_choice(score):
    if score+10 <= 21:
        return '11'
    else:
        return '1'
```

Magnus 1.0 First

```python
# Defining the AI
def magnus_choice(cards_drawn, score):
    bust_cards_remaining = 0
    least_card_to_bust = 22 - score
    cards_remaining = 52 - cards_drawn
    if score > 11:
        for key in card_number_value:
            if card_number_value[key] >= least_card_to_bust:
                bust_cards_remaining += cards_left[key]
        probability_next_card_busts = (bust_cards_remaining)/(cards_remaining)
        if probability_next_card_busts >= 0.6735:
            return 'stick'
        else:
            return 'twist'
    else:
        return 'twist'

def magnus_ace_choice(score):
        if score+10 <= 21:
            return '11'
        else:
            return '1'
```

Situational Battleground

```python
__author__ = 'Joe'
from random import randint

games_playing = int(input("How many games are to be played: "))
five_card_games_played = 0
games_played = 0
draws = 0
AI1_wins = 0
AI2_wins = 0
AI1_busts = 0
AI2_busts = 0

while games_played < games_playing:

    situation1 = False
    situation2 = False

    # Creating the initial setup
    cards_left = {'2': 4, '3': 4, '4': 4, '5': 4, '6': 4, '7': 4, '8': 4, '9': 4, '10': 4, 'J': 4, 'Q': 4, 'K': 4, 'A': 4}
    Two = ['H', 'D', 'C', 'S']
    Three = ['H', 'D', 'C', 'S']
    Four = ['H', 'D', 'C', 'S']
    Five = ['H', 'D', 'C', 'S']
    Six = ['H', 'D', 'C', 'S']
    Seven = ['H', 'D', 'C', 'S']
    Eight = ['H', 'D', 'C', 'S']
    Nine = ['H', 'D', 'C', 'S']
    Ten = ['H', 'D', 'C', 'S']
    Jack = ['H', 'D', 'C', 'S']
    Queen = ['H', 'D', 'C', 'S']
    King = ['H', 'D', 'C', 'S']
    Ace = ['H', 'D', 'C', 'S']
    # A more sophisticated deck introduced to aid finding and fixing card suits
    card_deck = {'2': Two, '3': Three, '4': Four, '5': Five, '6': Six, '7': Seven, '8': Eight, '9': Nine, '10': Ten, 'J': Jack, 'Q': Queen, 'K': King, 'A': Ace}
    # Defines card values (Ace contributes as 1 to current score to prevent avoidable busts)
    card_number_value = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'J': 10, 'Q': 10, 'K': 10, 'A': 1}
    # Creates a basic list of card numbers
    cards_list = list(card_deck.keys())

    # Defining AI1
    def ai_choice(cards_drawn, score):
        if score == 17 and cards_drawn == 3 and len(Six) == 2 and len(Five) == 3:
                return 'twist'
        if score >= 17:
            return 'stick'
        else:
            return 'twist'

    def ai1_ace_choice(score):
        if score+10 <= 21:
            return '11'
        else:
            return '1'

    # AI1 starts to play
    AI1_score = 0
    AI1_cards_drawn = 0
    choice = 'twist'                                                                                                    # To enter the while loop qualifier
    while choice == 'twist' or AI1_cards_drawn < 2:

        # Randomly generating the card number
        i = randint(0, int(len(cards_list))-1)                                                                          # Generates random number for drawn card's index
        card_drawn_number = cards_list[i]                                                                               # Defines drawn card's number
        while cards_left[card_drawn_number] == 0:                                                                       # Checks if there are any suits left of that number
            i = randint(0, int(len(cards_list))-1)                                                                      # If not, regenerates new random number if none left
            card_drawn_number = cards_list[i]                                                                           # And redefines drawn card's number
        cards_left[card_drawn_number] += -1                                                                             # Decreases drawn card's count by 1

        # Finding the card suit
        for key in card_deck:                                                                                           # For every number in the card deck
            if card_drawn_number == key:                                                                                # If the card's number matches it, then
                j = randint(0, int(len(card_deck[key]))-1)                                                              # Generate a random number to be its index in its suit list
                card_drawn_suit = card_deck[key][j]                                                                     # Define drawn card's suit
                card_deck[key].remove(card_drawn_suit)                                                                  # Remove that suit from its suit list
                AI1_score += card_number_value[key]                                                                     # Increase current score by correct amount

        AI1_cards_drawn += 1                                                                                            # Increases cards drawn tally by 1
        AI1_aces_drawn = 4-len(Ace)                                                                                     # Defines how many aces player has drawn (ignore error)
        if AI1_cards_drawn == 1:                                                                                        # For first card drawn (this part is so the AI Magnus will work)
            AI1_first_card_number = card_drawn_number                                                                   # Save the number of the card
        elif AI1_cards_drawn == 2:                                                                                      # For second card drawn
            AI1_second_card_number = card_drawn_number                                                                  # Save the number of the card


        # If statement to check if busted and asks stick/twist
        if AI1_score > 21:
            AI1_score = "Bust!"
            break
        else:
            if (AI1_score == 17 and AI1_cards_drawn == 3 and len(Six) == 2 and len(Five) == 3):
                situation1 = True
            if AI1_cards_drawn >= 2:                                                                                    # If statement to ensure at least two cards always drawn
                choice = ai_choice(AI1_cards_drawn, AI1_score)                                                          # Input of stick or twist (error catching not needed)
    else:
        # Finished drawing cards
        for n in range(1,5,1):                                                                                          # Makes n equal 1, then 2, 3, 4
            if AI1_aces_drawn >= n:                                                                                     # If you have N or more aces
                ace_choice = ai1_ace_choice(AI1_score)                                                                  # Input value wanted (error catching not needed)
                if ace_choice == '11':                                                                                  # If '11' selecting
                    AI1_score += 10                                                                                     # Add 10 to the original 1 score (to make 11)
                elif ace_choice == '1':                                                                                 # If '1' selecting
                    pass                                                                                                # Add 0 to the original 1 score (to make 1)
            else:                                                                                                       # If you don't have N or more aces
                break                                                                                                   # Don't bother trying higher N's

    # Creating the initial setup
    cards_left = {'2': 4, '3': 4, '4': 4, '5': 4, '6': 4, '7': 4, '8': 4, '9': 4, '10': 4, 'J': 4, 'Q': 4, 'K': 4, 'A': 4}
    Two = ['H', 'D', 'C', 'S']
    Three = ['H', 'D', 'C', 'S']
    Four = ['H', 'D', 'C', 'S']
    Five = ['H', 'D', 'C', 'S']
    Six = ['H', 'D', 'C', 'S']
    Seven = ['H', 'D', 'C', 'S']
    Eight = ['H', 'D', 'C', 'S']
    Nine = ['H', 'D', 'C', 'S']
    Ten = ['H', 'D', 'C', 'S']
    Jack = ['H', 'D', 'C', 'S']
    Queen = ['H', 'D', 'C', 'S']
    King = ['H', 'D', 'C', 'S']
    Ace = ['H', 'D', 'C', 'S']
    # A more sophisticated deck introduced to aid finding and fixing card suits
    card_deck = {'2': Two, '3': Three, '4': Four, '5': Five, '6': Six, '7': Seven, '8': Eight, '9': Nine, '10': Ten, 'J': Jack, 'Q': Queen, 'K': King, 'A': Ace}
    # Defines card values (Ace contributes as 1 to current score to prevent avoidable busts)
    card_number_value = {'2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10': 10, 'J': 10, 'Q': 10, 'K': 10, 'A': 1}
    # Creates a basic list of card numbers
    cards_list = list(card_deck.keys())

    # Defining AI2
    def ai2_choice(score):
        if score >= 17:
            return 'stick'
        else:
            return 'twist'

    def ai2_ace_choice(score):
        if score+10 <= 21:
            return '11'
        else:
            return '1'

    # AI2 starts to play
    AI2_score = 0
    AI2_cards_drawn = 0
    choice = 'twist'                                                                                                    # To enter the while loop qualifier
    while choice == 'twist' or AI2_cards_drawn < 2:

        # Randomly generating the card number
        i = randint(0, int(len(cards_list))-1)                                                                          # Generates random number for drawn card's index
        card_drawn_number = cards_list[i]                                                                               # Defines drawn card's number
        while cards_left[card_drawn_number] == 0:                                                                       # Checks if there are any suits left of that number
            i = randint(0, int(len(cards_list))-1)                                                                      # If not, regenerates new random number if none left
            card_drawn_number = cards_list[i]                                                                           # And redefines drawn card's number
        cards_left[card_drawn_number] += -1                                                                             # Decreases drawn card's count by 1

        # Finding the card suit
        for key in card_deck:                                                                                           # For every number in the card deck
            if card_drawn_number == key:                                                                                # If the card's number matches it, then
                j = randint(0, int(len(card_deck[key]))-1)                                                              # Generate a random number to be its index in its suit list
                card_drawn_suit = card_deck[key][j]                                                                     # Define drawn card's suit
                card_deck[key].remove(card_drawn_suit)                                                                  # Remove that suit from its suit list
                AI2_score += card_number_value[key]                                                                       # Increase current score by correct amount

        AI2_cards_drawn += 1                                                                                            # Increases cards drawn tally by 1
        AI2_aces_drawn = 4-len(Ace)                                                                                     # Defines how many aces player has drawn (ignore error)
        if AI1_cards_drawn == 1:                                                                                        # For first card drawn (this part is so the AI Magnus will work)
            AI2_first_card_number = card_drawn_number                                                                   # Save the number of the card
        elif AI1_cards_drawn == 2:                                                                                      # For second card drawn
            AI2_second_card_number = card_drawn_number                                                                  # Save the number of the card


        # If statement to check if busted and asks stick/twist
        if AI2_score > 21:
            AI2_score = "Bust!"
            break
        else:
            if (AI1_score == 17 and AI1_cards_drawn == 3 and len(Six) == 2 and len(Five) == 3):
                situation1 = True
            if AI2_cards_drawn >= 2:                                                                                    # If statement to ensure at least two cards always drawn
                choice = ai2_choice(AI2_score)                                                                          # Input of stick or twist (error catching not needed)
    else:
        # Finished drawing cards
        for n in range(1,5,1):                                                                                          # Makes n equal 1, then 2, 3, 4
            if AI2_aces_drawn >= n:                                                                                     # If you have N or more aces
                ace_choice = ai2_ace_choice(AI2_score)                                                                  # Input value wanted (error catching not needed)
                if ace_choice == '11':                                                                                  # If '11' selecting
                    AI1_score += 10                                                                                     # Add 10 to the original 1 score (to make 11)
                elif ace_choice == '1':                                                                                 # If '1' selecting
                    pass                                                                                                # Add 0 to the original 1 score (to make 1)
            else:                                                                                                       # If you don't have N or more aces
                break

    if situation1 is True or situation2 is True:
        five_card_games_played += 1
        if AI1_score == "Bust!":
            AI1_busts += 1
        if AI2_score == "Bust!":
            AI2_busts += 1
        if AI1_score == AI2_score:                                                                                  # If equal sscores
            draws += 1
        elif AI1_score == "Bust!":                                                                                  # If you bust (and AI doesnt, caught in previous if statement)
            AI2_wins += 1
        elif AI2_score == "Bust!":                                                                                  # If AI bust (and player doesnt, caught in first if statement)
            AI1_wins += 1
        elif AI2_score > AI1_score:                                                                                 # If neither bust and AI scores higher
            AI2_wins += 1
        elif AI1_score > AI2_score:                                                                                 # If neither bust and player scores higher
            AI1_wins += 1
        games_played += 1

    games_played += 1

print("Games played: "+str(games_played))
print("Situation games played: "+str(five_card_games_played))
print("AI1 wins: "+str(AI1_wins))
print("AI2 wins: "+str(AI2_wins))
print("Draws: "+str(draws))
print("AI1 busts: "+str(AI1_busts))
print("AI2 busts: "+str(AI2_busts))
```

Magnus 1.0 Second

```
def magnus_choice(cards_drawn, score):
    bust_cards_remaining = 0
    least_card_to_bust = 22 - score
    cards_remaining = 52 - cards_drawn - player_cards_drawn
    if score > 11:
        cards_left[player_first_card_number] += 1
        cards_left[player_second_card_number] += 1
        for key in card_number_value:
            if card_number_value[key] >= least_card_to_bust:
                bust_cards_remaining += cards_left[key]
        value_of_cards_dealt = player_score - (card_number_value[player_first_card_number] +
card_number_value[player_second_card_number])
        estimated_value_of_opponent_pair = (19-value_of_cards_dealt)
        if least_card_to_bust < estimated_value_of_opponent_pair/2:
            estimated_bust_cards_of_opponent_pair = 2
        else:
            estimated_bust_cards_of_opponent_pair = 0
        estimated_bust_cards_remaining = bust_cards_remaining - estimated_bust_cards_of_opponent_pair
        probability_next_card_busts = (estimated_bust_cards_remaining)/(cards_remaining)
        if probability_next_card_busts > 0.5:
            cards_left[player_first_card_number] += -1
            cards_left[player_second_card_number] += -1
            return 'stick'
        else:
            cards_left[player_first_card_number] += -1
            cards_left[player_second_card_number] += -1
            return 'twist'
    else:
        return 'twist'
def magnus_ace_choice(score):
        if score+10 <= 21:
            return '11'
        else:
            return '1'
```

Menu

```python
__author__ = 'Joe'

again = 'y'
print("RULES")
print("")
print("This is a system which allows you to play pontoon against an AI.")
print("Pontoon is similar to blackjack. Each player receives two cards, then chooses to draw another or stick with the score they have.")
print("If they score over 21, they automatically lose. ")
print("Aces count as a score of either 1 or 11 to be chosen by the player once they declare they have stopped drawing cards.")
print("There are no splits, and the five-card-trick rule is not active.")
print("")
while again == 'y':
    start_choice = str(input("Do you want to play first? Enter (y/n): "))
    while start_choice != 'y' and start_choice != 'n':
        start_choice = str(input("That's not a valid answer, type 'y' or 'n' exactly. Enter again: "))
    if start_choice == 'y':
        print("")
        import AI_goes_second
    else:
        print("")
        import AI_goes_first
    print("")
    again = str(input("Do you want to play again? Enter (y/n): "))
    while again != 'y' and start_choice != 'n':
        again = str(input("That's not a valid answer, type 'y' or 'n' exactly. Enter again: "))
else:
    print("")
    print("You have finished playing pontoon with the AI.")
```