

## **Python 21 Card Trick Documentation**

### **Introduction**

Having never used Python before being set a project to solve the card trick 21 felt quite a large step for me. However, I was excited to step up to the challenge. After reading through all the learning resources from lectures I realised I knew more than I originally thought.

We were asked to create a python command line application to recreate the card trick “21”. A game where the dealer would deal out 21 cards putting 7 cards in 3 columns side by side. The player would then select a card but not tell the dealer. Instead the player tells the dealer which column it is in. The dealer picks up the cards column by column sandwiching the column the player chose in between the other two. The dealer then re-deals the cards and asks the player where the card is now. After going through this process another two times the dealer picks out the 11<sup>th</sup> card which is, the players card.

When re-creating the game in python, I had to abide to the following rules:

- The player cannot tell the program what their card is as this would defeat the object of the trick.
- The program cannot use any external python libraries.
- All code must be referenced.

### **Creating the game**

**Link to game:** <https://gitlab.cs.cf.ac.uk/c1673107/21-Card-Trick-Python/wikis/home>

### **Core functionality**

Firstly, the game prints out some welcome text informing the user on how to play and what is going to happen. There is then a 3 second break until the next few functions load. A function creates a deck of 52 random cards and a second function takes out 21 cards from this deck. A function is then run to present 3 columns of 7 cards to the user and presents an input field so they can type which column their card is in. A for loop then repeats this process and a final function takes the 11<sup>th</sup> card and presents it to the player. Playing the game is quite simple, as the intro text explains. All the user needs to do is enter 1,2 or 3 as per the card they choose and which column it moves to. The card is automatically guessed at the end to save the user from having to ask for their card to be presented.

### **Going into detail**

The game uses mainly a lot of the content I have learnt in the lectures. It also however incorporates further content from which I have found through multiple learning resources online; these sources are explicitly referenced in the comments of the code and towards the end of this document.

## Printing the welcome text

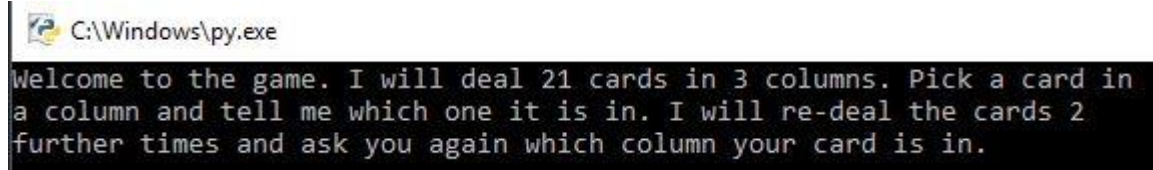
I thought it would be a good idea to inform the player on how to play and give an idea of how the game works. From what I had learnt in lectures I originally just printed using triple quotes as shown below:

```
print("""Welcome to the game. Please choose a card and
      I will try to guess it. I will deal 21 cards in 3 columns.
      Pick a card in a column and tell me which one it is in""")
```

However, what this did was when run in the command prompt, lines 2 and 3 were indented differently to the top line. I had to manually use different lines as to abide by the PEP guidelines of maximum 79 characters. After research, I found on stack overflow a way of using text wrapping. For example:

```
def welcome_text():
    welcome_str = "Welcome to the game. I will deal 21 cards in 3 columns. Pick a card in a column and tell me which one it
    print(textwrap.fill(welcome_str, 70))
    print("-----")
    time.sleep(3)
```

Here I do not have to put my string on different lines the import 'textwrap' from the standard library does it for me. Here I print out the string 'welcome\_str' in lines of maximum character length 70 as shown below:



```
C:\Windows\py.exe
Welcome to the game. I will deal 21 cards in 3 columns. Pick a card in
a column and tell me which one it is in. I will re-deal the cards 2
further times and ask you again which column your card is in.
```

## Creating a deck of 52 random cards

Originally the first thing I did in the game was create a deck of 52 cards. I defined a function, 'create\_shuffled\_deck' which creates 2 lists, 'cards\_value' and 'cards\_suit'. The next step was to combine the 2 lists and add them to another list called 'deck'. To do this I used my knowledge of for loops and lists from class notes so that for the entire list 'cards\_value', each string from 'card\_suit' would be concatenated to each string in 'card\_value'. This is done by using the method 'append'. I also used indexes to access each card suit. Again, using my class notes on imports, I used the standard library and the random import to then shuffle this deck of 52 cards.

```
def create_shuffled_deck():
    card_value = ["2", "3", "4", "5", "6", "7", "8", "9", "10", "Jack", "Queen", "King", "Ace"]
    card_suit = ["Hearts", "Diamonds", "Spades", "Clubs"]
    deck = []
    for i in range(len(card_value)):
        deck.append(card_value[i] + " of " + card_suit[0])
        deck.append(card_value[i] + " of " + card_suit[1])
        deck.append(card_value[i] + " of " + card_suit[2])
        deck.append(card_value[i] + " of " + card_suit[3])
    random.shuffle(deck)
    return deck
```

## Taking 21 Cards from the deck

Here using my knowledge of lists again created another empty list, 'cards\_21'. I then used a for loop from my class notes to add 21 cards to this list from the list 'deck'. I then realised I would need to then remove each card that is added to remove any chance of duplication. To do this I used the pop method found in my lecture notes. The function when run returns the 21-card list.

```
def take_21_cards():  
    cards_21 = []  
    for i in range(0,21):  
        cards_21.append(deck[i])  
        deck.pop(i)  
    return cards_21
```

## Dealing the cards into 3 columns of 7

The next step was to deal the cards into 3 columns of 7 and ask the user which column their card is in. The original method I used to deal out the cards was a for loop with the enumerate method to give the indexes of the cards, this is shown below.

```
for i, card in enumerate(cards_21):  
    if i%3 == 0:  
        list_one.append(cards_21[i])  
    if i%3 == 1:  
        list_two.append(cards_21[i])  
    if i%3 == 2:  
        list_three.append(cards_21[i])
```

I created this using notes I had from lectures. The for loop would simply go through the list 'cards\_21' and add a card to each list. I wasn't entirely sure at the time how to deal a card to each column. After taking some advice from a friend he pointed me to a previous lecture slide covering the modulus operator. After researching online, I found an example on stack overflow of the use of this operator. The use of list comprehension here means the modulus operator cycles between the list for each successive integer. The use of the if statement determines the result of the modulus operation and deals a card to its respective column accordingly.

```
column_one = cards_21[0::3] # Th  
column_two = cards_21[1::3] # Th  
column_three = cards_21[2::3] #
```

## The new method

However, I wasn't completely made up on this method. On the same stack overflow page, someone referred to the use of slices. I did the research to understand what was going on and concluded that due to its simplicity, I would use it instead of the for loop. This as shown is a much easier way to deal the cards without the use of for loops or if statements.

I first renamed the lists to columns to make it more understandable. I will explain this code by referring to each column. In column one within the slice, the 0 is going to tell the list 'cards\_21' to deal the very first card in the list to 'column\_one'. The 2 colons then inform the program to not add the second or third card. It deals the first card and then skips to the 4<sup>th</sup> card. The 3 at the end is the same for all columns and tells 'cards\_21' to then deal every third card from that point. So, after the first card is dealt to column one, the next card to be dealt will be the 4<sup>th</sup> card, and then the 7<sup>th</sup> card etc.

The other 2 columns work in the same way. For example, in 'column\_two', the number 1 within the slice tells 'cards\_21' to deal the second card to this column and the 3 in 'column three' tells 'cards\_21' to deal the third card to this column.

The difficulty in all this was making sure that a card was dealt to each column row by row. I had many failures in doing this as originally I wasn't thinking and trying to be too simple. I created the whole program and then realised I was dealing 7 cards to a column at a time, not row by row. I solved this problem with originally fixing my for loop using lecture notes by adding [i] and then in the end using the slices.

## Asking the user to choose a column

Once I had successfully managed to create 3 columns and deal 7 cards row by row I had to present the columns to the user. This needed to happen so that the user could see the columns, pick a card and then tell the program where it was. This is shown below.

```
def user_choose():  
    print("Column one: {}".format(column_one))  
    print("-----")  
    print("Column two: {}".format(column_two))  
    print("-----")  
    print("Column three: {}".format(column_three))  
    print("-----")  
    while True:  
        user_column = input("Which column number is your card in: 1, 2 or 3?: ") # Cast to an int  
        user_column = int(user_column)  
        if 1 <= user_column <=3:  
            print("You have chosen column {}".format(user_column))  
            print("-----")  
            return user_column  
            break  
        else:  
            continue  
            print("Not a valid number")
```

This was probably one of the easier parts of the task but very important. I used basic knowledge from lectures in printing out the strings and then formatted the strings using also my lecture notes. Formatting the strings this way allowed me to tell the user what columns they were as well as displaying the cards. For example, the user would see this,

```
-----  
Column one: ['3 of Spades', 'King of Spades', 'Ace of Spades', '3  
-----  
Column two: ['6 of Clubs', '10 of Diamonds', '8 of Diamonds', 'Kir  
-----  
Column three: ['2 of Hearts', '9 of Hearts', '9 of Spades', '4 of  
-----  
Which column number is your card in: 1, 2 or 3?:
```

as opposed to just seeing the cards printed out. This helps to make sure that the user picks the right column for the program to properly function. I printed the lines of hyphens to again make it more readable.

In lectures, we had covered while loops as well as for loops. This next part is the last thing I did in the entire program. This is because I went through the program several times 'bug testing' and realised that if I entered any number such as 10, the program would break. The while loop fixes this. What it does is makes sure that while the number entered is between 1 and 4, 'user\_column' is returned and the function stops ('break'). However, if this if statement isn't satisfied, the function repeats itself ('continue') and the string 'not a valid number' is printed. This means that the user cannot progress until a valid number is entered.

## Sandwiching the columns

```
cards_21 = []  
if user_column == 1:  
    for i, card in enumerate(list_two):  
        cards_21.append(list_two[i])  
  
    for i, card in enumerate(list_one):  
        cards_21.append(list_one[i])  
  
    for i, card in enumerate(list_three):  
        cards_21.append(list_three[i])  
  
if user_column == 2:  
    for i, card in enumerate(list_one):  
        cards_21.append(list_one[i])  
  
    for i, card in enumerate(list_two):  
        cards_21.append(list_two[i])  
  
    for i, card in enumerate(list_three):  
        cards_21.append(list_three[i])  
  
if user_column == 3:  
    for i, card in enumerate(list_one):  
        cards_21.append(list_one[i])  
  
    for i, card in enumerate(list_three):  
        cards_21.append(list_three[i])  
  
    for i, card in enumerate(list_two):  
        cards_21.append(list_two[i])
```

This was my first attempt at sandwiching the columns using my lecture notes. This block of code consists of for loops and if statements. At the beginning the 'cards\_21' list is emptied so that the cards can be collected. According to whatever column is chosen, one of the columns is collected first, then the chosen column is collected and then the third.

For example, if the user chose column one, using the append method, all the cards in column 2 are added to 'cards\_21'. Then all the cards from column 1 are added, and then the cards from column 3. Column 1 is now sandwiched between 2 and 3.

Whilst this worked, it is very inefficient using lots of lines and unnecessary for loops and if statements. After reading back over some of my previous work and notes from lectures I realised there was a much simpler way.

I will explain this new approach on the following page.

## The second approach

```
for i in range(2):
    cards_21 = []
    if user_column == 1:
        cards_21 = column_two + column_one + column_three
    elif user_column == 2:
        cards_21 = column_one + column_two + column_three
    else:
        cards_21 = column_one + column_three + column_two

    column_one = cards_21[0::3]
    column_two = cards_21[1::3]
    column_three = cards_21[2::3]

    user_column = user_choose()
cls()
```

The code above demonstrates my final way of sandwiching the columns. To simplify the program (hereby reducing the lines of codes) I used my knowledge of for loops from lectures to repeat the method without needing to copy and paste several times (as I did originally).

I once again clear the 'cards\_21' list so that I can re add the cards to it. The if statements are dependent on the user's choice of column. I read through older notes on list comprehension and saw that you can add lists together very simply. In this example, if the user chose column one, all three columns are added to the empty list of 'cards\_21'. However due to how the trick works, whatever column is chosen is the second list to be added. So, for the before example, list 2 is added, then list 1 (the chosen column) and then finally list 3.

The way this works is that now I have a set of 21 cards in 'cards\_21', each list has been added to the end of the one before. Thus, the chosen column and their card is somewhere in the middle of this 21-card list.

The method of dealing the cards (using slices) then splits these 21 cards into 3 columns of 7 cards again and the function which asks the user to choose a column is then called and run.

## Guessing the card

Now the user has been asked 3 times in total where their card is and the method above sorts the cards and re-deals twice. However, it is now important to once again gather the cards into a list of 21. This is because to guess the card the program needs to print out the 11<sup>th</sup> card. The following function 'guess\_card' does exactly this.



```
def guess_card():  
    if user_column == 3:  
        cards_21 = column_two + column_three + column_one  
    elif user_column == 2:  
        cards_21 = column_one + column_two + column_three  
    else:  
        cards_21 = column_two + column_one + column_three  
    user_card = (cards_21[10])  
    print("-----")  
    print("Your card is the {}".format(user_card))  
    print("-----")  
    print("Thank you for playing the game, By Jack Allcock C1673107")  
  
guess_card()
```

I used the exact same method for collecting the cards as explained in the previous section. However, to just reiterate, once this function is run, the cards that have been re-dealt for the 2<sup>nd</sup> time are now collected and put into the list `cards_21`; with the chosen column in the middle. The program then accesses the 11<sup>th</sup> card which will be from the second column as the first seven cards are from 'column\_one'. To do this I defined a new variable 'user\_card' which using an index number 10, will get the 11<sup>th</sup> element. I created this using my lecture knowledge of accessing elements in a list. I then printed a formatted string so that whatever card is printed, is the one they chose. Otherwise I would have to use if statements for every possible card that could be chosen out of 52. So, formatting is a much better way to do this.

## Clearing the screen

As you may have noticed, in several functions and areas of my code I made use of a function 'cls'. I was bug testing the code and realised that when printing to the user the columns, it got very messy and hard to read. I spent a lot of time on the web trying to find a way of resolving this and one of the things that came up was a way of clearing the screen. After reading many different resources I found one on stack overflow. A member of the community had posted one which would work across-platforms and operating systems.

What the code does is that whenever the function is called, all text is cleared. I will give an example of this now (the screenshots are on the next page). After three times of asking the user where their card was in total there were around 9 columns on the screen, each three only separated by a string of hyphens. For readability, each time the user enters which column their card is in the screen clears and the next text loads up.

This wasn't directly required in the specification however after seeing my family test the game, readability and easiness to use was an obvious issue.

Firstly, I added the import 'os'. This allows the function to recognise the operating system on the user's machine.

```
import random
import os
import time
import textwrap

def cls():
    # Clears the function from the client after x amount of seconds - found on Stack Overflow.
    os.system('cls' if os.name=='nt' else 'clear')
```

An if statement in the function is used to check what operating system the machine runs on. So, if it is windows, ('nt' is the registered name for windows) then the module will be cleared with the 'cls' command. However, if it is any other Unix Operating System such as Apple macOS, 'clear' will be used instead.

```
user_column = input("Which column number is your card in: 1, 2 or 3?: ") # Cast to an int so I can print it within a string.
user_column = int(user_column)
if 1 <= user_column <=3:
    print("You have chosen column {}".format(user_column))
    print("-----")
    return user_column
    break
else:
    continue
    print("Not a valid number")

cls()
```

For example, the 'cls' function is called here to clear the screen when the user inputs what column their card is in.

### Break in the program

I added a timer to the end of the welcome text. To do this I imported the module 'time' and then set the sleep to 3 seconds (time.sleep(3)). What this does is create a 3 second break once the welcome text has been printed before the next function shows up. This gives the user a bit of time to read the text before other things start loading up on the screen. I found this on the Python Docs.

```
import time
```

```
welcome_str = "Welcome to the game. I will deal 21 cards in 3 c
print(textwrap.fill(welcome_str, 70))
print("-----")
time.sleep(3)
```



## **References**

### **Stack Overflow - Improving my method of dealing out the cards in 3 columns**

*<http://stackoverflow.com/questions/509211/explain-pythons-slice-notation>*

*<http://stackoverflow.com/questions/40303814/python-adding-to-3-lists-row-by-row>*

### **Stack Overflow – text wrapping the welcome string**

*<http://stackoverflow.com/questions/16430200/a-good-way-to-make-long-strings-wrap-to-newline-in-python-3-x>*

### **Stack Overflow – Clearing the terminal**

*<http://stackoverflow.com/questions/2084508/clear-terminal-in-python>*

### **Python Docs – Break in the program**

*<https://docs.python.org/2/library/time.html>*

## **To conclude**

As I stated at the start being new to Python made this task quite a scary one to take on. However, after writing this documentation, it now it feels great knowing that I have overcome the challenge and feel a lot more comfortable with python then I did before; but maybe that's because I am more of a practical learner.

I commented my code fully (not always shown in the screenshots to make picture sizes smaller) so that I could come back and refer to the code in the future. It also means other people reading my code can understand it better, I guess it's a good habit to get into.

I believe there are areas in which I could improve, such as putting all my code into functions to reduce line space and I am sure there are easier methods out there to do what I did. However, I felt that the code in its state now worked best for me. I have also learnt that planning is a lot more important than I first realised. Since I wasn't entirely confident with what I was doing I wrote the whole program originally without functions. This was my original approach because at the time I was writing line by line constantly correcting things and testing. Now I realise this was the wrong thing to do as I had to come back and re write several parts to get into functions.

I have learnt a lot from this experience and have also enjoyed it. I am looking forward to any future projects that I get to undertake in Python.

### Extra Credit Work

After completing the original exercise, I unfortunately think I left it too late to give the extra credit a proper go. This is because I oversimplified the idea of what we had to do in my head. When it came to doing it, I did the whole thing wrong. However, I began to correct it but haven't had time to complete it. Thus, I have left the code for this exercise in triple quotes to avoid program errors.

What I have completed is the following:

```
def relay_cards():
    global pile_one, pile_two, pile_three
    pile_one = cards_11[0:3]
    pile_two = cards_11[4:8]
    pile_three = cards_11[8:12]
    print(pile_one)
    print(pile_two)
    print(pile_three)

def pile_choose():
    pile_input = int(input("Pick a pile between 1 and 3: "))
    return pile_input

relay_cards()
pile_input = pile_choose()

if pile_input == 3:
    print("Removing piles 1 and 2... ")
    pile_one = []
    pile_two = []
    print("The final pile left is: ")
    print(pile_three)
    card_input = int(input("Choose a card, 1, 2, 3 or 4"))
    # if card_input = 1:
    #     print("Card removed")
    #     pile_three.pop(0)
    #     card_input = int(input("Choose a card, 1, 2, 3"))
    # if card_input = 1:
    #     print("Card removed")

# Need to finish above so that each card removed until their card is reached.
```

11 Cards are put together at the end of the guess card function. The cards are then dealt into 3 piles of different sizes. The piles were printed for my own use.

The user is asked to choose a pile. If the pile with their card in is chosen, in this case 3, all other piles are removed. I then didn't get time to finish the next part; where I would have gone through this pile until I got to their card.