# Portfolio Project: Go Fish

## Introduction

I chose Go Fish for my profile project because it is a relatively simple game, but gives the opportunity for a programmer to add complexities if they want and have time.

Here is a link to go fish rules I used for the project:
https://www.wikihow.com/Play-Go-Fish

The game involves a deck or “pond”, two to four players, each with their own hands. The game goes in circles until one player has no cards left in their hand.

## Design and Implementation

The game goes as follows:

First, the game prompts the player to select how many computer players they would like to play against, and how large their initial hands should be. I did this to make sure the functions, classes, and methods are generalizable.

Then I create a deck, or “pond”, with a pond class object and methods.

To store the hand of each player, Originally I used a two dimensional array, where the first dimension was the player index and the second was the hand. I later refactored the code to use player and deck classes. The player hand and name are player class attributes, and I wrote class methods to handle actions like creating the hand, checking the hand for cards, passing cards, etc…

| Class: Pond |
| --- |
| **Coats** - coats of deck cards<br>**card_number** - 1, 2, …, Q,K, A<br>**Deck** - Stores the cards in the pond |
| **create_deck**() - called at game start<br>**print**() - print deck contents |

| Class: Player |
| --- |
| **Player_number**<br>**Player_name**<br>**Hand** - cards in player hand<br>**Memory** - known cards in other hands |
| **create_hand**<br>**Sort_hand**<br>**Check_for_matches**<br>**Check_memory** - for a card<br>**Remove_card_from_memory**<br>… |

```
27  class Player():
28      def __init__(self, size_of_hands, pond, player_number, number_of_players) ->
        None:
29
30          self.memory = [set() for i in range(0, number_of_players)]
31
32          self.player_number = player_number
33          if player_number == 0:
34              self.player_name = "Player"
35          else:
36              self.player_name = "Computer " + str(player_number)
37
38          self.hand = []
39          self.create_hand(size_of_hands, pond)
40
41      def print_hand(self):
42          print(self.player_name, "hand:", self.hand)
43
44      def print_memory(self):…
46
47      def create_hand(self, size_of_hands, pond):…
50
51      def sort_hand(self):…
53
54      def check_for_matches(self, selected_card):…
60
61      # TODO: refactor this, not good to return 2 values from method
62      def check_memory(self, players, current_player, testing):…
72
73      def remove_card_from_all_memory(self, four_of_kind_card):…
77
```

*Example of Player class*

I also have many functions in a module to support the game. Most of these functions are not placed in classes because they do not logically belong to an existing class. For instance, checking for win conditions, getting the player with the most cards in the hands, etc… Originally the code was very procedural, so I still have a few functions that could be merged into the classes for better code organization, such as putting the four_of_a_kind_check() and go_fish_action() in the Player class.

```
78
79  > def wait_for_player(): ···
83
84
85  > def get_number_of_players(manual=None): ···
96
97
98  > def get_hand_size(manual=None): ···
08
09
10  > def get_card_from_user(card_numbers, hand): ···
24
25
26  > def get_selected_player_from_user(number_of_players, current_player): ···
33
34
35  > def get_largest_hand_player(players, current_player): ···
43
44
45  > def four_of_a_kind_check(players, current_player): ···
61
62
63  > def remove_four_of_kind(players, current_player, four_of_kind_card): ···
76
77
78  > def check_win_condition(players, current_player): ···
85
86
87  > def go_fish_action(pond, players, current_player): ···
96
```

*Examples of functions not in classes*

Then the game goes in a loop, starting with the user player, in a while loop until the game end condition is true.  Here is the pseudo code:

```
While not game_end
    If user_player
        Input selected_card and selected_player
        Request selected_player.hand for selected_card

    If computer_player
        Check_cards_in_hand() for card_in_memory
        If match
            request that selected_card from selected_player
        If not match
            Request random_card_in_hand from player_with_largest_hand

    Save requested card in all players memory
    If match: swap_cards_action
        Else go_fish_action

    check_four_of_a_kind()
        If true, give player turn again
        If false, increment player

    check_win_condition()
        If True, game_end = True
```

```
    """Computer Turn"""
    if current_player != 0 or testing:

        print("\n§~~~~ Computer Player {0}'s Turn! ~~~~§".format(
            current_player))

        has_memory = players[current_player].check_memory(
            players, current_player, testing)

        if has_memory is None:
            if testing:
                print("no player cards found in memory.")
            selected_card = random.choice(
                players[current_player].hand)[0]
            selected_player = go_fish_modules.get_largest_hand_player(
                players, current_player)
        else:
            selected_player = has_memory[0]
            selected_card = has_memory[1]
```

*Example of computer turn driver code. You can see VSCode cannot error check "check_memory" and other methods*

```
95
96      """Swap hands and go fish functions"""
97
98      matched_cards = players[selected_player].check_for_matches(
99          selected_card)
100
101     print("Player {0} asks Player {1} for {2}'s".format(
102         current_player, selected_player, selected_card))
103
104     for each in players:
105         each.memory[current_player].add(selected_card)
106
107     if not matched_cards:
108         go_fish_modules.go_fish_action(pond, players, current_player)
109     else:
110         go_fish_modules.swap_cards_action(
111             selected_player, selected_card, current_player, matched_cards, players)
112
113     if testing:
114         players[current_player].print_memory()
115
116     """End turn functions"""
117     for each in players:
118         each.sort_hand()
119
120     four_of_kind_card = None
121     four_of_kind_card = go_fish_modules.four_of_a_kind_check(
122         players, current_player)
123
```

*Example of players swapping cards or go fish driver code*

As the course and my knowledge improved, I moved a lot of the algorithms into functions and methods, and a lot of the player and deck data into classes. I also recognize I have a few nested **for** statements that have poor big O complexity, however the hands and decks are very small, and it would not be worth the effort to index or sort the cards for performance improvements

Besides that, I think if you want to understand the strategy of a game, it's a good idea to program it! I had to make a few decisions on how to add intelligence to the computer players. However there is a large component of chance in the game, so adding more complex intelligence would likely not change the outcome significantly.

## Conclusions

From my project, I learned a few things.

First, it's good to just get started. Try to make good design decisions early, but it's ok to “pass” or “TODO” suboptimal or “nice to have” functionality, and come back to it later. It is actually pretty easy and quick to refactor code (if you do not lose track of it).

Second, I needed to build automated testing into the program design. Since the game is random and fairly long, it is not practical to manually test functionality. Building a “testing” or “debug” mode into the program that does not require user input probably saved me hours of testing. However, I feel like there was a better way to program the testing functionality than putting it straight in the source code.

Third, the parser was not able to do “intellisense” or whatever it's called on many of my classes. For instance when they were part of a function, or in a list. This meant I needed to be very careful when typing method names and arguments, because the IDE could not check them for me.

In hindsight I mostly wish I had implemented test driven development early in the game.

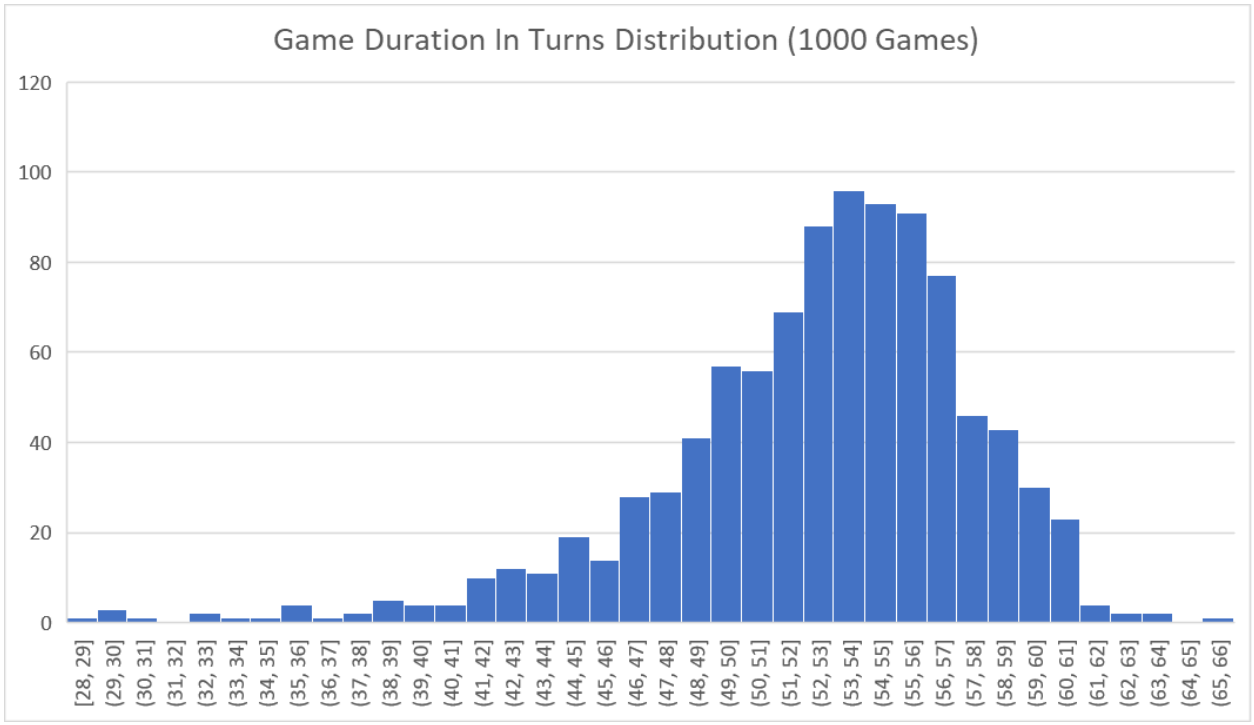
I also refactored the code significantly into OOP after developing a functional or procedural version of the game. However I think that was a very good learning exercise. Probably not something I would do again though!

I think the memory feature I implemented really made the computer players intelligent, and challenging. Maybe too challenging. They have perfect memory of who has what card, because players can only ask for cards they have in their hand. In the future, I would make a “Easy”, “Medium”, and “Challenging” mode, by writing logic to drop cards out of the computer memory, or only storing them if they have that card in their hand. I think both of those are normal human behavior.

I also think I would create cards with a different structure. I used a tuple such as ‘A’, ‘C’ for ace of clubs. However this made sorting problematic. I should have created a data structure that also included a card “value” such as 11 for Jack, 12 for Queen, etc… This would have made sorted hands look nicer, and may have made my code easier. A dictionary for each card may be worked like

```
{Card_value: Jack,
Card_number: 11,
Card_suit: Club,}
```

I would also continue to refactor the OOP structure of the code, or rewrite it starting with a stronger OOP design.


Game Duration In Turns Distribution (1000 Games)
120
100
80
60
40
20
0
[28, 29]
(29, 30]
(30, 31]
(31, 32]
(32, 33]
(33, 34]
(34, 35]
(35, 36]
(36, 37]
(37, 38]
(38, 39]
(39, 40]
(40, 41]
(41, 42]
(42, 43]
(43, 44]
(44, 45]
(45, 46]
(46, 47]
(47, 48]
(48, 49]
(49, 50]
(50, 51]
(51, 52]
(52, 53]
(53, 54]
(54, 55]
(55, 56]
(56, 57]
(57, 58]
(58, 59]
(59, 60]
(60, 61]
(61, 62]
(62, 63]
(63, 64]
(64, 65]
(65, 66]