

Python Project Report

191158510 - Programming in Engineering

The Evolution of Trust

Sandbox Simulation of the Online Game



Kirtan Premprakash Patel - s2935848

UNIVERSITEIT TWENTE.

Contents

1. Introduction	1
1.1 Prisoner's Dilemma	1
1.2 The Game of Trust	1
1.3 Player Descriptions	3
2. Implementation Method	4
2.1 Ipython Notebook	4
2.2 Utility Scripts	4
2.3 Input File	4
3. Project Work	5
3.1 Algorithm	5
3.2 Variable Description	6
3.2.1 Global Variables	6
3.2.2 Local Variables	6
3.3 Class Diagram	7
3.4 Sequence Diagram	8
4. Testing & Valdition	9
4.1 Test Case 1	9
4.2 Test Case 2	9
4.3 Test Case 3	9
5. Remarks	10
Appendix	11
A. Project Code	11
A.1 PiE.code-mod.py	11
A.2 players.py	13
A.3 matches.py	16
A.4 tournament.py	17
A.5 display.py	18
B. Test Output	19
B.1 Test Case 1	19
B.2 Test Case 2	19
B.3 Test Case 3	20

1 Introduction

1.1 Prisoner's Dilemma

Cooperation and trust are two of the most important elements of any successful human endeavor — people need to work together to accomplish big things, and mutual trust helps to avoid squabbling and backstabbing. Despite their self-evident value, of course, these social bonds between individuals and groups break down all the time in practice. This basic fact of human relationships raises an important question: how can we create social conditions that encourage trust and cooperation? Game theory has been considering this problem for decades, and has produced some valuable and surprisingly clear-cut insights into the matter.

Game Theorist have discussed this question with the help of a game, also known in game theory as the infamous Prisoner's Dilemma. The Prisoner's Dilemma is named after a story where two suspects can either squeal on their partner-in-crime ("cheat"), or stay silent ("cooperate"). Using this game, the following takeaways have been concluded.

Game theory has shown us the three things we need for the evolution of trust:



1. REPEAT INTERACTIONS

Trust keeps a relationship going, but you need the knowledge of possible future repeat interactions *before* trust can evolve.



2. POSSIBLE WIN-WINS

You must be playing a non-zero-sum game, a game where it's at least possible that *both* players can be better off -- a win-win.



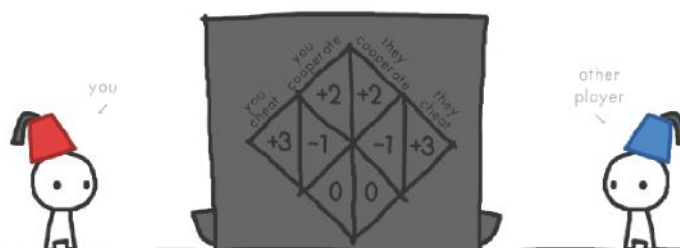
3. LOW MISCOMMUNICATION

If the level of miscommunication is *too* high, trust breaks down. But when there's a little bit of miscommunication, it pays to be *more* forgiving.

1.2 The Game of Trust

The Prisoner's Dilemma has been implemented as a game by the simple means of a coin-machine game. It has an interactive gamified implementation curated Nick Case. It can be found on the following webpage : <https://ncase.me/trust/>

In front of each player is a machine: The players can either choose to COOPERATE (put in coin), or CHEAT (not put in coin). Each action has a consequence on their score, and the scoring matrix (can be modified) is set to a default value.



In the game, both players stand to benefit if they both choose the "cooperate" option, but choosing the "cheat" option when the other player chooses "cooperate" is even more beneficial — and your opponent takes a hit. However, if both players "cheat," neither benefits at all. Consequently, in order for both players to do well, they must resist the temptation to cheat, trust each other, and cooperate.

"The Evolution Of Trust" explores the implications of this game in 3 separate scenarios:

- Individual one-on-one games, with 10 rounds per match
- Round-robin tournaments in which multiple different players each play against each other once in 10-round matches
- Repeated (or "iterated") tournaments, in which numerous players using a variety of strategies face off against each other multiple times; in these repeated tournaments, the 5 lowest-ranking players are eliminated and the 5 best-performing players are duplicated after every round, so that players using the most effective strategies slowly take over the game









In Step 7 of the online game mentioned above, a Sandbox is available in which you can tweak the parameters of the evolution process (Repeated Tournaments) of a group of players. The parameters that can be tweaked are :

POPULATION

PAYOFFS

RULES

Start off with this distribution of players:

 COPYCAT 3	 CHEATER 3
 COOPERATOR 3	 GRUDGER 3
 DETECTIVE 3	 COPYKITTEN 3
 SIMPLETON 3	 RANDOM 4

POPULATION

PAYOFFS

RULES

The payoffs in a one-on-one game are:

	cheat	cooperate
cheat	+3 -1	+2 -1
cooperate	-1 0	+2 0

set default

POPULATION

PAYOFFS

RULES

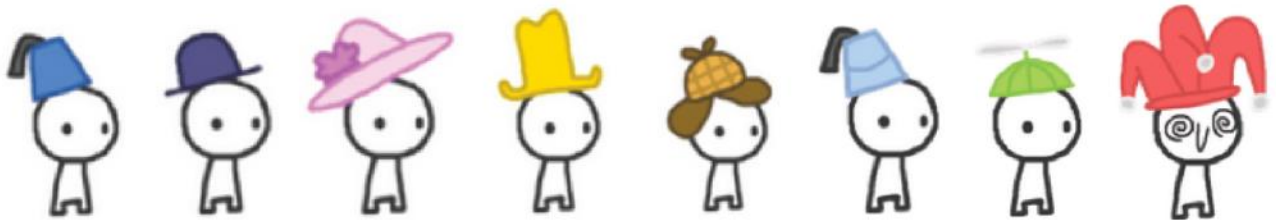
Play 10 rounds per match:

After each tournament, eliminate the bottom 5 players & reproduce the top 5 players:

During each round, there's a 5% chance a player makes a mistake:

1.3 Player Descriptions

There are 8 types of players in the simulation whose behaviours have been mentioned in the game, before the sandbox. The descriptions of these behaviours are mentioned below.



- **Copy Cat (CC)** : Starts with Cooperate. Then, it simply repeats whatever the opponent did in the last round
- **All Cheat (CH)** : Always Cheats
- **All Cooperate (CO)** : Always Cooperates
- **Grudger (GR)** : Starts with Cooperate, and keeps co-operating until the opponent cheats it even once. Afterwards, it always plays cheat
- **Detective (DE)** : Starts with : Cooperate, Cheat, Cooperate, Cooperate. Afterwards, if you ever cheat, it plays like a copycat. Otherwise, it plays like an All Cheat
- **Copy Kitten (CK)** : It is like a Copycat, except it cheats back only after you cheat it twice in a row.
- **Simpleton (ST)** : It starts with cooperate. if you cooperate back, it does the same thing as last move, even if it mistakes. if you cheat back, It does the opposite thing as last move, even if it mistakes.
- **Random (R)** : Just plays Cheat or Cooperate randomly with a 50/50 chance

2 Implementation Method

2.1 Ipython Notebook

The main element of the Project was composed on Kaggle in the Ipython environment (Python Notebook). Python Notebook is a useful tool which helps you segment the code and thus keep a check on proper functioning of the entire program.

Kaggle Ipython Notebook is a web-based interactive environment that combines code, rich text, images, videos, animations, mathematical equations, plots, maps, interactive figures and widgets, and graphical user interfaces, into a single document. The notebooks are saved as structured text files (JSON format), which makes them easily shareable. Thus it was used extensively

2.2 Utility Scripts

Utility scripts can be thought of as small, simple snippets of code written as independent code files and stored within your project environment. Usually, each script is designated to perform a particular task. It contributes a lot to reusability of code and helps in making the code more modular.

They were used widely to use modular scripts in the main operation. In a Python Environment, scripts in the same directory can be imported in each other. To do so in Kaggle, the scripts which I want to import were saved as utility scripts and then imported when required. This lead to a well structured and clutter free code. It even helped in increasing the readability of the code to a great extent.

2.3 Input File

Since there were many variables, a *.txt file has been used to read user inputs rather than the user having to type in the variables. The template of the input file has been defined and needs to be followed. User can change the parameters in the input text file and run the code after saving the file with his input variables.

```
1 <<POPULATION>>
2 CH:6:
3 CO:5:
4 R:0:
5 CC:4:
6 CK:2:
7 GR:5:
8 ST:1:
9 DE:2:
10
11
12 <<PAY-OFFS>>
13 00:+0:
14 01:+3:
15 10:-1:
16 11:+2:
17
18
19 <<RULES>>
20 <ROUNDS/MATCH>
21 RPM:10:
22
23 <MAXIMUM NO. OF TOURNAMENTS>
24 T_MAX:1000:
25
26 <PLAYERS REPLACED EACH TOURNAMENT>
27 PR:3:
28
29 <PROBABILITY OF MISTAKES>
30 M:0.10:
```

3 Project Work

3.1 Algorithm

Algorithm 1 PiE_code-mod.py

Require: *input file*

Ensure: *input file conforms to format*

read file and save variables

run_simulation = True

sim_counter = 0

while (*run_simulation == True*) **do**

current_playground = playground[last]

max_players = max(current_playground)

if (*max_players == Players_Total*) **or** (*sim_counter >= T_MAX*) **then**

get population history of each player from playground history

display result

*write population history in *.txt/ *.csv file*

run_simulation = False

else

new_players_distribution = play_a_tournament(current_playground,parameters)*

add new_players_distribution to playground[]

sim_counter+=1

end if

end while

Algorithm 2 tournament.py >> def play_a_tournament

Require: *population_distribution,parameters**

total_players = sum(population_distribution)

score_matrix = np.zeros(total_players)

for (*i in range(0,total_players)*) **do**

for (*i in range(0,total_players)*) **do**

find player_type_index for the player at i and j

match_score_card = match(player_index_i, player_index_j, parameters)*

update score for players at i and j

end for

end for

find lowest and highest scores after tournament

reduce count of player_types with lowest scores

increase count of player_types with highest scores

return *new_population*

3.2 Variable Description

3.2.1 Global Variables

These are the variables used in the main code of the project : PiE_code-mod.py

Variable	Type	Description
PG_initial	list	initial distribution of players in the playground
Players_Total	int	value of total number of players
PayOff	list	payoff values of each possible result
RPM	int	number of rounds per match
T_MAX	int	maximum number of tournaments the simulation runs for
PR	int	number of people replaced/duplicated after each match
M	float	value of probability of mistake
Players	dict	relation between player type and their index
Players_index	dict	relation between player index and their type
Population	list	line numbers where the population data is given in the input file
Pay_offs	list	line numbers where the pay-offs data is given in the input file
line_counter	int	count of number of line in the input file
input_file	list	line numbers where the population data is given in the input file
run_simulation	bool	boolean value of whether simulation is to be run or not
sim_counter	int	value of number of times the simulation has run
PG_current	list	distribution of players in the playground in the current simulation
max_players	int	highest value of the number of each type of players
dataframe	DataFrame	highest value of the number of each type of players
new_PG_dist	list	new population distribution after playing a tournament

3.2.2 Local Variables

These are the variables used in the functions and classes called by the main code.

1. players.py

Variable	Type	Description
p	float	random value to predict if a player makes a mistake
mistake	bool	indicates if player has made a mistake or not
initial_choice	string	initial choice (cheat "0", cooperate "1") made by the player
final_choice	string	decision (cheat "0", cooperate "1") of the player after the mistake
binary_result	string	result of the player for that round 00 : player cheated, opponent cheated 01 : player cheated, opponent cooperated 10 : player cooperated, opponent cheated 11 : player cooperated, opponent cooperated
choice	string	choice made by the player according to the player behaviour

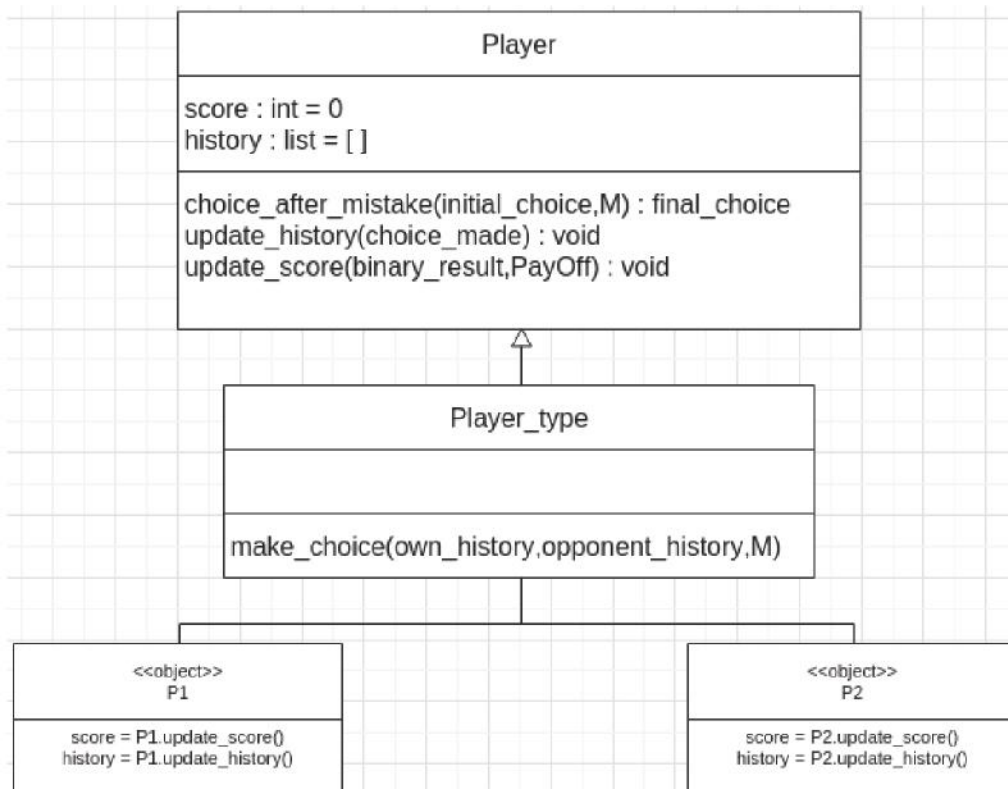
2. matches.py

Variable	Type	Description
result_P1/P2	string	match result of player 1/2
score_card	list	scores of player 1 and 2 in def calculate_score()
Players	dict	relation between player type and their index
Players_index	dict	relation between player index and their type
player_class_1/2	string	string of the class name to be used to initialize P1/P2
scores	list	scores of P1 and P2 in def match()

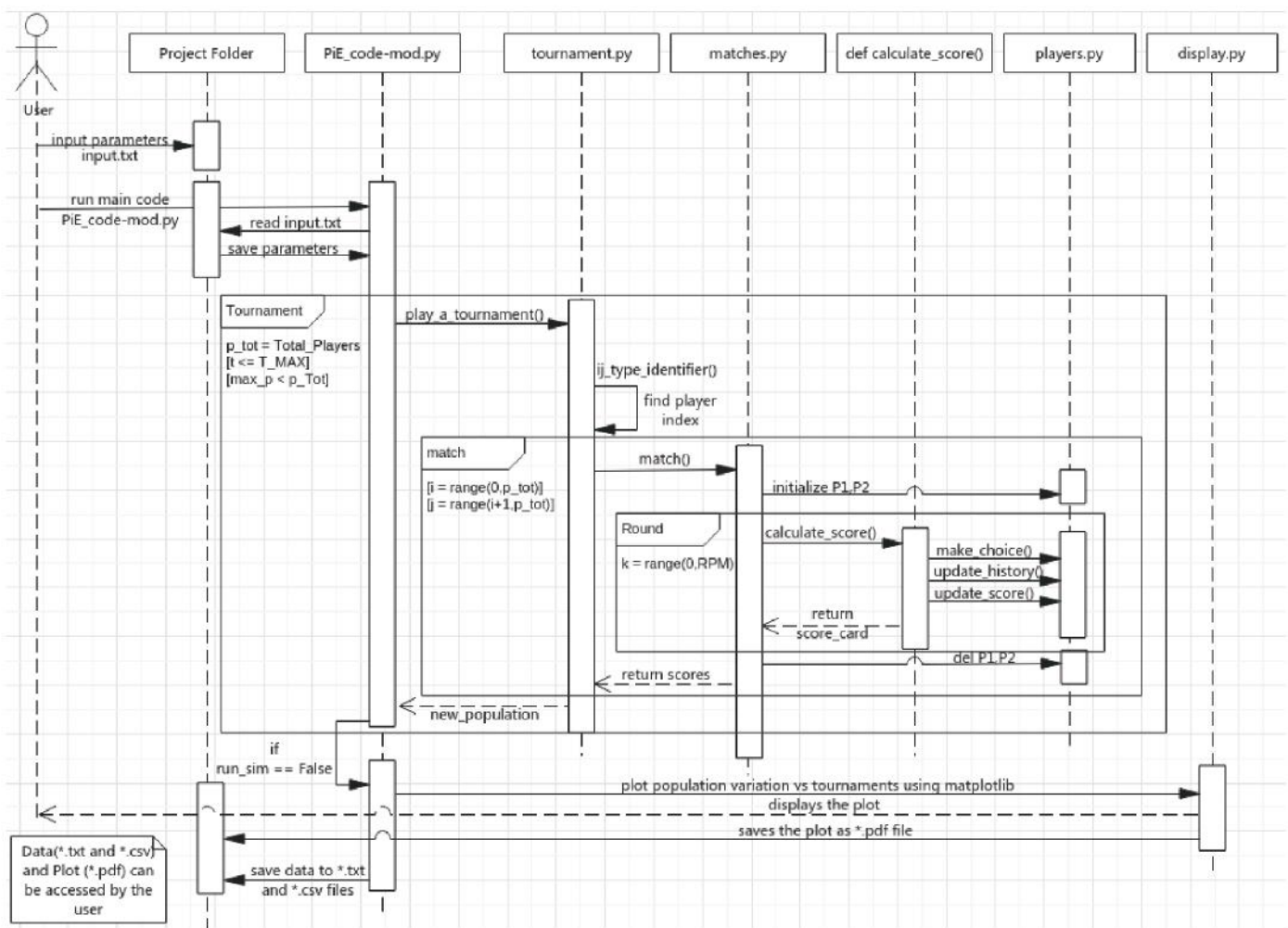
3. tournament.py

Variable	Type	Description
identifier_list	list	to find the type of player according to their player number
index_num	int	index of player type
total_players	int	total number of players
score_matrix	list	score of each player playing the tournament
player_index_1/2	index	index of player type of player 1/2
match_score_card	list	scores of both players after a match
l2h_score_matrix	list	score_matrix arranged from lowest to highest scores
h2l_score_matrix	list	score_matrix arranged from highest to lowest scores
lowest	list	lowest scores of players after the tournament
highest	list	highest scores of players after the tournament
new_population	list	new population distribution after replacement of players
player_population_index	int	index of player type of the given score

3.3 Class Diagram



3.4 Sequence Diagram



4 Testing & Validation

The Game illustrates that with longer duration and lower miscommunication, the CopyKitten is the player type that survives. In smaller time, the Cheater wins. The duration to reach this result depends on the initial population distribution, but the core results are the above mentioned.

What the game is, defines what the players do.
 Our problem today isn't just that people are losing trust,
 it's that our *environment* acts against the evolution of trust.

That may seem cynical or naive -- that we're "merely" products of
 our environment -- but as game theory reminds us, we *are* each
 others' environment. **In the short run, the game defines the players.**
But in the long run, it's us players who define the game.

4.1 Test Case 1

Test Case 1														
										Parameters				
	CH	CO	R	CC	CK	GR	ST	DE			RPM	T_MAX	PR	M
Initial Population	5	4	2	3	2	3	4	3			15	1500	3	0.05
Final Population	0	0	0	0	26	0	0	0						
	"00"	"01"	"10"	"11"										
Pay Offs	0	3	-1	2										

4.2 Test Case 2

Test Case 2														
										Parameters				
	CH	CO	R	CC	CK	GR	ST	DE			RPM	T_MAX	PR	M
Initial Population	5	4	2	3	2	3	4	3			10	1500	5	0.1
Final Population	0	0	0	0	0	26	0	0						
	"00"	"01"	"10"	"11"										
Pay Offs	0	3	-2	2										

4.3 Test Case 3

Test Case 3														
										Parameters				
	CH	CO	R	CC	CK	GR	ST	DE			RPM	T_MAX	PR	M
Initial Population	1	2	4	3	2	1	2	3			15	1000	4	0.1
Final Population	18	0	0	0	0	0	0	0						
	"00"	"01"	"10"	"11"										
Pay Offs	0	2	-2	1										

5 Remarks

The Project underwent a lot of rework and the final program has been modified till the point it felt extremely clean and modular. Certain problems that I handled during this project, my learning and takeaways have been mentioned below.

- Initially the program was made in a single file. After having a working prototype, the code was modularized. This can be seen as the modules are tightly bound and have multiple parameters.
- The initial prototype was long and had a lot of redundancies. The length of the code has been reduced significantly but working on ways to remove the redundancies and make a clean and modular piece of code.
- Reading varying numbers from a file was also a task which required some attention. I did not want to restrict the user to input single or double digit player numbers and wanted it to be user friendly.

To resolve this, the input file format has been specified such that the number is to be entered between 2 colons(:) this makes it easy to be read using the *re* library.

- Initially, the problem of 2 different player types having the same score (one of the lowest or highest) was faced. This meant that when we try to find the index of the player, we end up selecting the same player type twice instead of selecting the other one in the second time.

This was resolved : once the highest/lowest score is used to find the player index, the score is set to 0 so that the next search iteration will give the value of the next player index rather than repeating the same player index.

- I learnt a lot about Programming Project Reports and different tools which are used to convey the working of the program. I have tried to use the tools as effectively as possible.
- I would aim to make the used programs as stand-alone as possible which would make it easy to detect issues and bugs if they are spotted.
- Python is an easy-to-code language and using it for this project has helped me a lot. I have extensively used the concept of Classes and this is one of the features of Object Oriented Programming (OOP) that makes it extremely popular.
- Converting certain snippets into C and using Cython will accelerate the code and reduce the runtime. When working with greater lengths of python codes, using compiled snippets or JIT compiling and C-Extensions for Python is essential.

It was a tremendous learning experience, which gave me a lot of insight on Python and Game Theory. I developed a lot of interest in the field and understand how computing and programming can be useful in deciphering many such phenomena with a better understanding.

Appendix

A Project Code

A.1 PiE_code-mod.py

```
#!/usr/bin/env python3

import re
import copy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

# import user defined libraries
from players import *
from matches import *
from tournament import *
from display import *

PG_initial = []
Players_Total = 0
PayOff = []
RPM = 0
TMAX = 0
PR = 0
M = 0
Players = {"CH" : 0, "CO" : 1, "R" : 2, "CC" : 3,
           "CK" : 4, "GR" : 5, "ST" : 6, "DE" : 7}
Players_index = {0 : "CH", 1 : "CO", 2 : "R", 3 : "CC",
                 4 : "CK", 5 : "GR", 6 : "ST", 7 : "DE"}

input_file = open("input.txt", "r")
Population = [2,3,4,5,6,7,8,9]
Pay-offs = [13,14,15,16]

line_counter = 0

for a_line in input_file:
    line_counter+=1

    if line_counter in Population:
        PG_initial.append(int((re.search(':(.*):', a_line)).group(1)))
        continue

    if (line_counter in Pay-offs):
        PayOff.append(int((re.search(':(.*):', a_line)).group(1)))
        continue

    if (line_counter == 21):
        RPM = int((re.search(':(.*):', a_line)).group(1))
        continue

    if (line_counter == 24):
        TMAX = int((re.search(':(.*):', a_line)).group(1))
        continue

    if (line_counter == 27):
        PR = int((re.search(':(.*):', a_line)).group(1))
        continue

    if (line_counter == 30):
        M = float((re.search(':(.*):', a_line)).group(1))
        continue
```

```

input_file.close()

Players_Total = sum(PG_initial)
PG = np.array([PG_initial])

run_simulation = True
sim_counter = 0

while(run_simulation == True):
    PG_current = PG[-1]
    max_players = max(PG_current)

    if ((max_players == Players_Total) or (sim_counter>=TMAX)):
        if (sim_counter==TMAX):
            print("Maximum_Tournament_Limit_Reached")

            dataframe = pd.DataFrame({
                'CH':PG[:,0], 'CO':PG[:,1], 'R':PG[:,2], 'CC':PG[:,3],
                'CK':PG[:,4], 'GR':PG[:,5], 'ST':PG[:,6], 'DE':PG[:,7]
            })

            display_result(dataframe)

            np.savetxt('PG_distribution.csv',PG,delimiter=',')
            np.savetxt('PG_distribution.txt',PG,delimiter=',')
            run_simulation = False

        else :
            new_PG_dist = play_a_tournament(PG_current,M,PR,RPM,PayOff)
            PG = np.vstack([PG,new_PG_dist])
            sim_counter+=1

```

A.2 players.py

```
import math
import numpy as np

class Player():
    def __init__(self):
        self.score = 0
        self.history = []

    def choice_after_mistake(self, initial_choice, M):
        p = np.random.uniform(0,1)
        #print(p)
        if p > M :
            mistake = False
        else :
            mistake = True

        if mistake == False:
            final_choice = initial_choice
        elif (mistake == True) and initial_choice=="0":
            final_choice = "1"
        elif (mistake == True) and initial_choice=="1":
            final_choice = "0"
        return final_choice

    def update_history(self, choice_made):
        self.history.append(choice_made)

    def update_score(self, binary_result, PayOff):
        if binary_result == "00":
            result = 0
        elif binary_result == "01":
            result = 1
        elif binary_result == "10":
            result = 2
        elif binary_result == "11":
            result = 3
        else :
            print("ERROR!!")
            exit()
        self.score+=PayOff[result]

# All Cheat
class CH(Player):
    def __init__(self):
        Player.__init__(self)

    def make_choice(self, own_history, opponent_history, M):
        choice = "0"
        final_choice = self.choice_after_mistake(choice, M)
        return final_choice

# All Co-operate
class CO(Player):
    def __init__(self):
        Player.__init__(self)

    def make_choice(self, own_history, opponent_history, M):
        choice = "1"
        final_choice = self.choice_after_mistake(choice, M)
        return final_choice
```

```

# Random Player
class R(Player):
    def __init__(self):
        Player.__init__(self)

    def make_choice(self, own_history, opponent_history, M):
        choice = str(math.floor(np.random.uniform(0,1)/0.5))
        final_choice = self.choice_after_mistake(choice, M)
        return final_choice

# Copycat
class CC(Player):
    def __init__(self):
        Player.__init__(self)

    def make_choice(self, own_history, opponent_history, M):
        if (len(opponent_history)==0):
            choice = "1"
        else :
            choice = opponent_history[-1]

        final_choice = self.choice_after_mistake(choice, M)
        return final_choice

# CopyKitten
class CK(Player):
    def __init__(self):
        Player.__init__(self)

    def make_choice(self, own_history, opponent_history, M):
        if (len(opponent_history)==0) or (len(opponent_history)==1):
            choice = "1"
        else :
            if ((opponent_history[-1]=="0") and (opponent_history[-2]=="0")):
                choice = "0"
            else :
                choice = "1"

        final_choice = self.choice_after_mistake(choice, M)
        return final_choice

# Grudger
class GR(Player):
    def __init__(self):
        Player.__init__(self)

    def make_choice(self, own_history, opponent_history, M):
        if (len(opponent_history)==0):
            choice = "1"
        elif ("0" in opponent_history) :
            choice = "0"
        else :
            choice = "1"

        final_choice = self.choice_after_mistake(choice, M)
        return final_choice

```



```

# Simpleton
class ST(Player):
    def __init__(self):
        Player.__init__(self)

    def make_choice(self, own_history, opponent_history, M):
        if (len(opponent_history)==0):
            choice = "1"
        else:
            if (opponent_history[-1] == "1"):
                choice = own_history[-1]
            elif (opponent_history[-1] == "0") :
                if (own_history[-1] == "0"):
                    choice = "1"
                elif (own_history[-1] == "1"):
                    choice = "0"

        final_choice = self.choice_after_mistake(choice, M)
        return final_choice

```

```

# Detective
class DE(Player):
    def __init__(self):
        Player.__init__(self)

    def make_choice(self, own_history, opponent_history, M):
        if (len(opponent_history)<=3):
            if (len(opponent_history)==1):
                choice = "0"
            else :
                choice = "1"
        elif ("0" in opponent_history[0:4]) :
            choice = opponent_history[-1]
        else :
            choice = "0"

        final_choice = self.choice_after_mistake(choice, M)
        return final_choice

```

A.3 matches.py

```
import numpy as np

from players import *

def calculate_scores(P1,P2,M,RPM,PayOff):

    for i in range(0,RPM):
        P1.choice = P1.make_choice(P1.history ,P2.history ,M)
        P2.choice = P2.make_choice(P2.history ,P1.history ,M)
        P1.update_history( P1.choice)
        P2.update_history( P2.choice)
        result_P1 = P1.choice + P2.choice
        result_P2 = P2.choice + P1.choice
        P1.update_score(result_P1 ,PayOff)
        P2.update_score(result_P2 ,PayOff)

    score_card = []
    score_card.append(P1.score)
    score_card.append(P2.score)

    return score_card

def match(player_index__1 , player_index__2 ,M,RPM, PayOff):

    Players = {"CH" : 0, "CO" : 1, "R" : 2, "CC" : 3,
               "CK" : 4, "GR" : 5, "ST" : 6, "DE" : 7}
    Players_index = {0 : "CH", 1 : "CO", 2 : "R", 3 : "CC",
                     4 : "CK", 5 : "GR", 6 : "ST", 7 : "DE"}

    player_class_1 = Players_index[player_index__1] + "()"
    player_class_2 = Players_index[player_index__2] + "()"

    P1 = eval(player_class_1)
    P2 = eval(player_class_2)

    scores = calculate_scores(P1,P2,M,RPM,PayOff)

    del P1
    del P2

    return scores
```

A.4 tournament.py

```
import math
import copy
import numpy as np

from matches import *

def ij_type_identifier(pop_dist, index_num):
    identifier_list = [0]
    for i in range(1,9):
        identifier_list.append(sum(pop_dist[0:i]))

    for j in range(0,8):
        if ((index_num>=identifier_list[j]) and (index_num<identifier_list[j+1])):
            player_index = j

    return player_index

def play_a_tournament(population_distribution, M, PR, RPM, PayOff):

    total_players = sum(population_distribution)

    score_matrix = np.zeros(total_players)

    for i in range(0, total_players):
        for j in range((i+1), total_players):

            player_index_1 = ij_type_identifier(population_distribution, i)
            player_index_2 = ij_type_identifier(population_distribution, j)
            match_score_card = match(player_index_1, player_index_2, M, RPM, PayOff)
            score_matrix[i] = score_matrix[i] + match_score_card[0]
            score_matrix[j] = score_matrix[j] + match_score_card[1]

    l2h_score_matrix = np.sort(score_matrix)
    lowest = l2h_score_matrix[0:PR]
    h2l_score_matrix = np.flip(l2h_score_matrix)
    highest = h2l_score_matrix[0:PR]

    new_population = copy.copy(population_distribution)

    for element in lowest :
        index = np.where(score_matrix==element)[0][0]
        player_population_index = ij_type_identifier(population_distribution, index)
        score_matrix[index] = 0
        new_population[player_population_index]-=1

    for element in highest :
        index = np.where(score_matrix==element)[0][0]
        player_population_index = ij_type_identifier(population_distribution, index)
        new_population[player_population_index]+=1

    return new_population
```

A.5 display.py

```
import pandas as pd
import matplotlib.pyplot as plt

def display_result(input_dataframe):

    # https://queirozf.com/entries/pandas-dataframe-plot-examples-with-matplotlib-pyplot

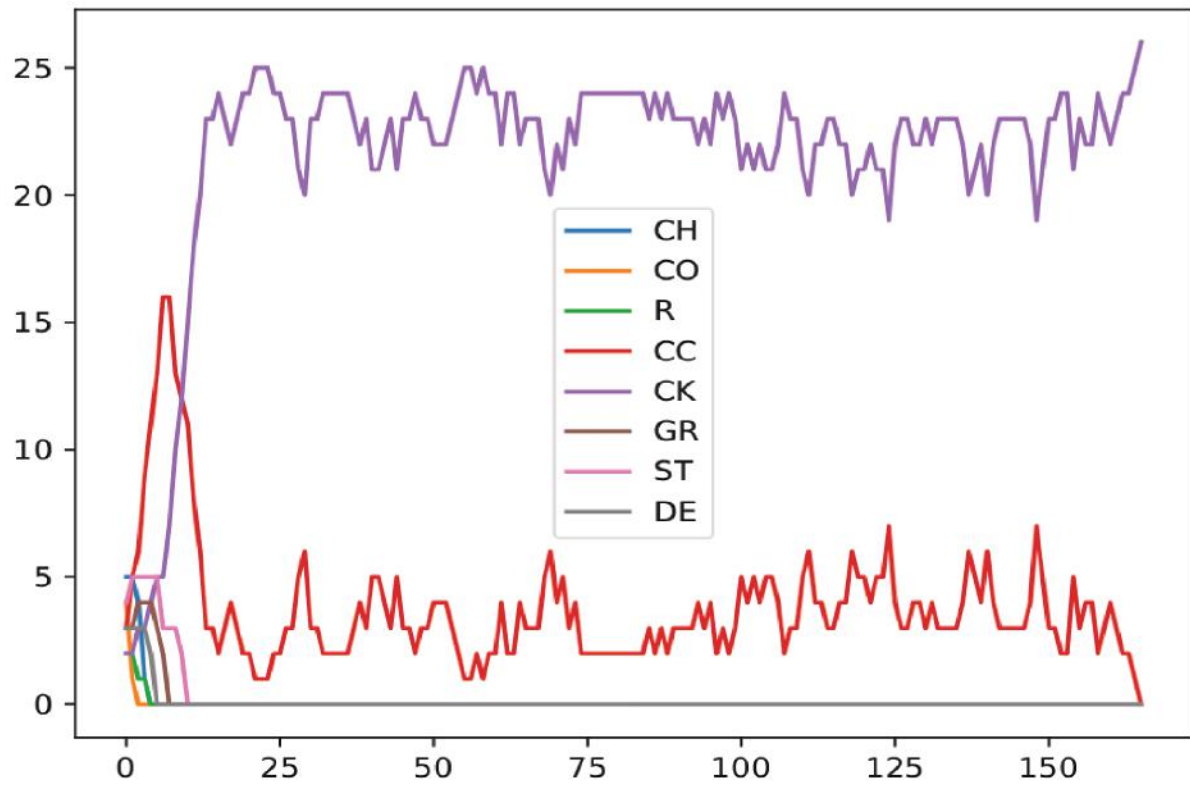
    players_type = [ 'CH', 'CO', 'R', 'CC', 'CK', 'GR', 'ST', 'DE' ]
    ax = plt.gca()
    for element in players_type:
        input_dataframe.plot(kind='line', y = element, linestyle = '-', marker=',', ax=ax)

    plt.savefig("result.pdf", format="pdf", bbox_inches="tight")
    plt.xlabel('No._of_Tournaments')
    plt.ylabel('Number_of_Players')
    plt.show()

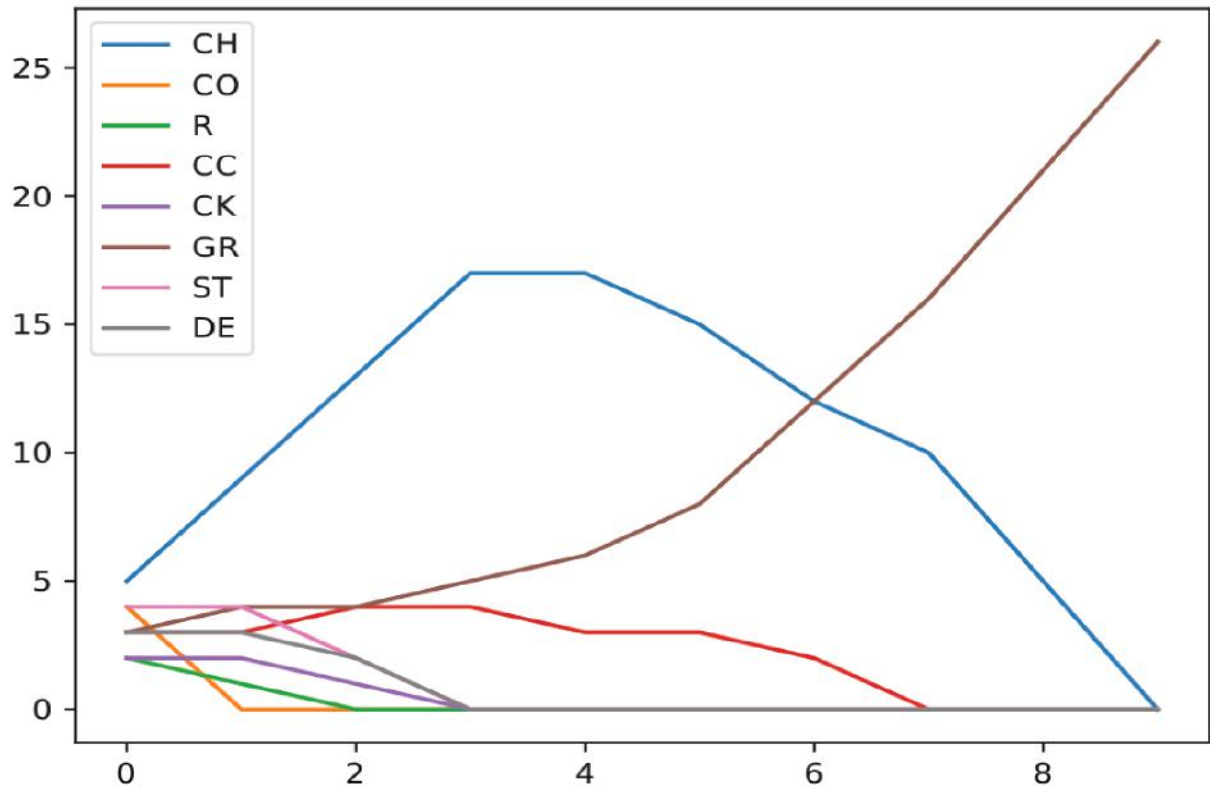
    return 0
```


B Test Output

B.1 Test Case 1



B.2 Test Case 2



B.3 Test Case 3

