

**Project 1.3.10 Game Theory**

Introduction

|  |  |
| --- | --- |
| How do people make decisions? How do rewards, punishments, and ethical and moral beliefs affect people’s decisions? How do decisions made in one situation apply to another?  In this project, you will create a decision-making strategy using *Python*. Your strategy will compete against other teams’ strategies in a round-robin tournament of a game called the **Iterative Prisoner’s Dilemma** (IPD). The IPD is a fundamental problem in **game theory**. Social scientists can use game theory to describe and predict people’s behavior. Examples include economic phenomena like stock market fluctuations and political phenomena like revolutions. Nations use game theory to simulate the outcomes of various options when negotiating economic treaties or conducting war. | How do I make a decision? |

Materials

* Computer with Enthought Canopy distribution of *Python*® programming language, Internet browser, and GitHub client software
* GitHub individual account and membership in organizational account's team

Procedure

**Part I: Describe a decision-making algorithm.**

1. Form teams of two as directed by your teacher. Meet or greet each other to practice professional skills. Set team norms.
2. Game theorists use algorithms to describe people’s decisions. Consider your decision to buy lunch each day. If the cafeteria director decides to raise the lunch price above a certain point, you will opt not to buy school lunch. At what price? Are there other factors? Express your algorithm for the decision to buy school lunch in terms of price and other factors. You could use any language you care to, but do not use the computer. Your expression must be conveyed in writing or through a graphical representation.
3. The Iterative Prisoner’s Dilemma is a series of rounds of the **Prisoner’s Dilemma.** The Prisoner’s Dilemma was one of the early algorithmic problems in game theory, first posed in 1950. In the Prisoner’s Dilemma, you and another person have committed a crime together, and you are caught without evidence. The police question you and the other person separately.

* If both of you collude together and refuse to talk the police, you both will go free.
* If you both confess your crime and betray each other, the liability of the crime will be split between you. You both will receive the standard punishment.
* If you stonewall the police, hoping to collude with the other person but that person betrays you, you will receive an unusually severe punishment while they go free and get to keep the stolen goods.
* If you betray the other person while they attempt to collude with you, you will be set free and get a cash payment.

Your result:

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | Your Action | |
|  |  | Collude | Betray |
| Other player’s Action | Collude | **R**eleased  *0 pts* | Set Free  + **T**reat  *+100 pts* |
| Betray | **S**evere Punishment  *-500 pts* | **P**unishment  *-250 pts* |

What is the maximum number of points possible for you in one round of this game? When does it occur?

1. Face off against another player to play a few rounds of the IPD without a computer. On the count of three (you can chant “IPD” together,) each player reveals their decision to collude or betray using an open hand or a closed fist.
2. In the Iterative Prisoner’s Dilemma, many rounds are played, one after another, with the same other player. Betraying in one round will sometimes cause the other player to lose trust and be less likely to collude with you in future rounds. Once players start betraying each other, round after round they end up turning each other in and doing poorly. Considered together with the other player, your outcomes are better if you collude round after round.

The Prisoner’s Dilemma lets social scientists study the conditions under which people will either act selfishly or act in the collective best interests of the group. Think of one situation in which a person in your life could act selfishly or in the interest of a common group, and explain how this relates to the Prisoner’s Dilemma.

1. Our recent capacity to generate, collect, store, and analyze huge amounts of data quickly has caused dramatic changes in all career fields. Simulation was an unimportant tool in social sciences in 1960. Explain why the rise in computational power has changed the career fields in social sciences such that simulation is now a fundamental tool for many social scientists.

**Part II. Explore the Simulation**

1. Make sure you are still in the “testingAB” branch, and then open prisonersDilemma.py in the *Python* code editor.

Execute the code, which imports prisoners’ dilemma player algorithms from other files and defines several functions for running a tournament. The code also simulates a tournament among four of the imported strategies called example0 (“E0”), example1 (“E1”), example2 (“E2”), and example3 (“E3”). Data from a round-robin tournament among four players each using one of these strategies is printed on screen and stored in a data file. Scroll up to examine the three sections of printed output.

-------------------------------------------------------------------------------

Section 0 - Line up

-------------------------------------------------------------------------------

Player 0 (P0): E0, Collude

Always collude.

Player 1 (P1): E1, Betray

Always betray.

Player 2 (P2): E2, Alternate

Collude, then alternate.

Player 3 (P3): E3, Collude but retaliate

Collude first round. Collude, except in a round after getting

a severe punishment.

-------------------------------------------------------------------------------

Section 1 - Player vs. Player

-------------------------------------------------------------------------------

Each column shows pts/round earned against each other player:

P0 P1 P2 P3

vs. P0 : 0 100 50 0

vs. P1 : -500 0 -376 -375

vs. P2 : -250 -75 0 -199

vs. P3 : 0 -75 -199 0

TOTAL : -750 -50 -525 -574

-------------------------------------------------------------------------------

Section 2 - Leaderboard

-------------------------------------------------------------------------------

Average points per round:

Team name (P#): Score with strategy name

E1 (P1): -13 points with Betray

E2 (P2): -132 points with Alternate

E3 (P3): -144 points with Collude but retaliate

E0 (P0): -188 points with Collude

1. The variable reports contains these report sections. There is also a Section 3, containing a separate report for each team. The following command at the IPython prompt will display the Section 3 report for player 0.

In []: print reports[3][0]

1. At the IPython prompt, execute the following commands and examine the output.

In []: scores, moves, reports = main\_play([example1, example2, team0])

In []: scores, moves, reports = main\_play([example1]\*3 + [example2]\*2)

1. The function main\_play() takes a single argument: a list. Explain the purpose of that list, based on your examination of the program’s output. Note that multiplying a list and an int produces a list with repeated elements:

[example1]\*3 + [example2]\*2

is equivalent to

[example1, example1, example1, example2, example2]

1. **Examine data about past moves.** The simulation program represents each decision by one of the two characters 'c' or 'b' to collude or betray. Before making the decision each round, each player (or algorithm) can consider what has happened in previous rounds. The algorithms have the previous rounds’ information in the form of a string. For example 'ccb' indicates that the player colluded in the first two rounds and betrayed in the most recent round. Each player can consider two strings: one for their own history and one for their crime partner’s history. Consider the following two histories:

Your history 'ccccc'

Your crime partner’s history 'ccccb'

You can tell that five rounds have been played already. What should be each player’s score, based on their moves?

1. The four players in the simulation shown above used four different algorithms. The *Python* code implementing the algorithms strategies is shown below. Only   
   the algorithm in Example 3 algorithm considers what has occurred in previous rounds. It uses my\_history and their\_history, which are strings as described in Step 14.

|  |  |
| --- | --- |
|  | *# Example 0. Always collude*  **def** move(my\_history, their\_history, my\_score, their\_score):  **return** 'c' |

|  |  |
| --- | --- |
|  | *# Example 1. Always collude*  **def** move(my\_history, their\_history, my\_score, their\_score):  **return** 'c' |

|  |  |
| --- | --- |
|  | *# Example 2. Collude, then alternate*  **def** move(my\_history, their\_history, my\_score, their\_score):  **if** **len**(my\_history)%2 == 0:  **return** 'c'  **else**:  **return** |

|  |  |
| --- | --- |
|  | *# Example 3. Collude but retaliate*  **def** move(my\_history, their\_history, my\_score, their\_score):  **if** **len**(my\_history)==0: *# It's the first round; collude.*  **return** 'c'  **elif** my\_history[-1]=='c' **and** their\_history[-1]=='b':  **return** 'b' *# Betray if severely punished last time,*  **else**:  **return** 'c' *# otherwise collude.* |

A ten-round match between player 0 and player 1 results as follows:

Example 0: 'cccccccccc' Score = -5000

Example 1: 'bbbbbbbbbb' Score = +1000

Analyze the code for these algorithms and record the results you expect from a ten-round match of the IPD between example2 and example3, following the previous example matching example0 to example1.

1. Check the output. Run a tournament that includes example2 and example3 among the strategies. Open the file tournament.txt. The simulation program will have stored that file in the same directory as the simulation program itself. The file probably opens by default in the application Notepad, but any text editor will do. At the top of the file is the record for team 1 vs. team 0, showing their final scores per round, the names for their strategies, and a record of their decisions in b and c strings. When the simulation runs, it runs 100 to 200 rounds of the dilemma between each pair of strategies.

team 1 vs. team 0

100 vs. -500

backstabber vs. loyal

bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb

ccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccccc

Scroll down until you find the data stored for example2 vs. example3. Is it different than your prediction? If so, how?

1. Notice that a strategy’s success is dependent upon the strategies it is collaborating with and competing against. Discuss this with one or more other teams in the class and describe why this occurs.

**Part III: Develop Your Own Algorithm**

1. You will now develop several algorithms to test against each other. Eventually you will select one algorithm to contribute to the classroom tournament. That one algorithm can integrate all of your initial algorithms or other algorithms. Your final algorithm must call at least one other procedure that you create. For example, your function move could use another function that you define called probability\_that\_other player\_will\_collude. With your partner, brainstorm algorithms for making the collude-vs.-betray decision. Recall the ground rules for brainstorming:

* Quantity over quality
* No evaluation of ideas, but piling on is okay
* Record ideas with tag lines of one or a few words

1. Review your team norms for the driver and navigator roles in pair programming.
2. Prepare to develop. Pick three of your ideas to implement in the *Python* code in the simulation program. Each idea will be implemented in a different teamXX file. Describe your ideas in human language and pseudocode.
3. Implement your ideas with code. Use test-driven development as follows.
   1.  Create a test. In Canopy, open the team file assigned to you by your teacher. The teamNN.py file is in the folder for your local repository. Think about how you could develop your algorithm in stages. What is the first stage? In the code at the end of the teamXX file, record whether your code at the end of that first stage should return 'b' or 'c' for one particular set of the four arguments. For example, the following test checks whether move() returns 'b' on the first round.

|  |  |
| --- | --- |
| 51  52  53  54  55  56  57 | *# Test 1*  **if** test\_move(my\_history='',  their\_history='',  my\_score=0,  their\_score=0,  result='b'):  **print** 'Test passed' |

* 1. Pair programming, create code that will pass this test. Execute and revise your code until it passes your test(s).

1. **Commit the changes**. You saved your changes on your computer when you executed the file. However, you did not commit those changes to the cloned repo on your computer. In the GitHub Client, select the Changes tab. You should see uncommitted changes to the repository. Select one of the files that has been changed and you should see lines you deleted (red) or added (green).
   1. Although your changes are saved on your local computer, they are not stored permanently by git. To create a permanent checkpoint of this change, you need to make a commit. Enter a summary (e.g., “Make first move”) and description (e.g., “Always collude on first move. Also created a test to check this.”) and select **Commit to testingAB**.
   2. You now have a record of that change in the testing branch. However, the master branch is still unchanged. Click on the arrow next to testing and switch back to the master branch. You will see that there are no changes to that branch. When you switch to the master branch, Canopy may warn you that the file has changed on disk; the change you made has disappeared! Git actually changed the file on the computer to reflect what is current in the master branch. The change still exists – try changing back to the testing branch. You can see how you can make changes to code without affecting the master version of the code and easily switch between the two.
   3. Go back to and stay on the testing branch. Repeat the process as you develop within the testing branch:
      1. Plan a small task.
      2. Create a test that will check that your code correctly accomplishes the task you plan.
      3. Creating code to pass the test.
      4. Commit the successful code to the testing branch.
2. **Create a program.** Implement three strategies of your design, one strategy in each of three files. Run tournaments in which you observe how your strategies do in competition and collaboration with other possible strategies. Record the results of your simulations. Complete the following for each idea:
   1. Write pseudocode and strategize how you will implement the algorithm in *Python*.
   2. Implement your algorithms in Python in any of the team files. Include team\_name, strategy\_name, and strategy\_description.
   3. To run a tournament with the algorithms you have developed, in the prisoners\_dilemma.py file modify the modules list in lines 44-46 to include any of the team files you modified and any of the example strategies you would like to use in your sample tournament. Then, in the second last line of the file, change main\_play(modules[0:4]) to main\_play(modules). Execute the file to run the sample tournament.
   4. Examine the tournament.txt file or the Section 4 reports to see how your algorithm performed against each other strategy.
   5. Record notes about your thinking as you develop your algorithms further. In your notes, include the performance of your algorithm against other strategies:

|  |  |
| --- | --- |
| You  They Scored  Scored | Your algorithm #1 |
| Always collude |  |
| Always betray |  |
| Use their previous move |  |
|  |  |

1. Decide on one algorithms to contribute to the whole-class tournament. Explain your reasoning for choosing that algorithm.

**Part IV. Contribute Your Strategy to the Class Collaboration**

1. Open the team file that has the strategy you want to contribute to the class tournament.
   1. Choose “Save As” and change the name to the team that your teacher instructed you to use. Choose “Yes” if asked if you want to replace the file.
   2.  In the Github client, create a Commit in the testingAB branch to save the changes you have made.
2. Once you have completed creating and testing your code, you need to merge your code into your Master Branch and then share your code with the class repository through a Pull Request.
   1. You have modified multiple files in your testingAB branch as you have worked to develop your strategy. However, we only want the file with your final strategy. We will use the Git command line to just merge that file from your testing branch into your Master branch.
3.  Switch to the “master” branch of your fork of the IPD Repository.
   1. In the GitHub for Windows Client, Click on the arrow next to “testingAB” and choose “master.”
   2. Canopy will automatically update the code file you have open, showing the original version of the code.
   3. Right-click on the name of the repository and choose “Open in Git Shell.”
4.  Using the Git Shell, you will select the file from the testingAB branch that has your team code in it.
5. Run the following commands:

**git checkout testingAB teamXX**

**git commit**

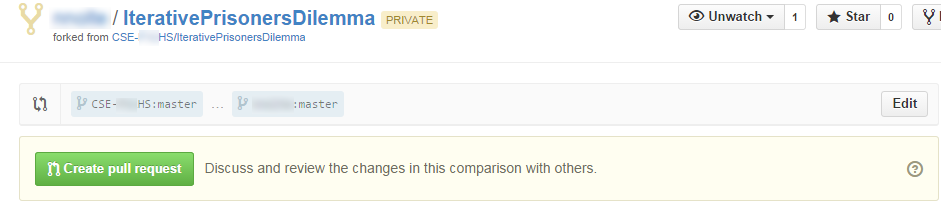
1. A text editor window should open up. Enter a commit message in this window, such as “Merged team code from Testing Branch.” Close the window and choose “Yes” when asked to save the file.
2. Run the following command to push the changes to your remote repository:

**git push**

1. You now should have your IPD strategy within the master branch.
2.  Open the repository on GitHub by either right-clicking on it in the list or directing a Web browser to [github.com](http://github.com/).
3. Create a **Pull Request** so your changes can be merged into the original code.
4. Click on the green symbol as shown to enter the compare and pull request screen:



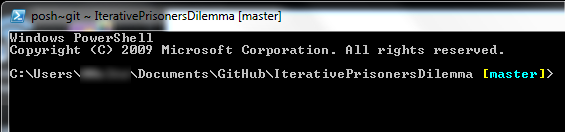
1. The top line shows what two things you are comparing. In this case it should show the master branch of the original repository in your school’s organization account (the base fork) and your master branch in your forked copy of the repository (the head fork). If it does not, click on the “Edit” button to change it.



1. Click on “Create pull request” and add a title and comment, similar to what you did with a commit by adding a commit message and comment. In the pull request type the names of your group members. Finish making the pull requests by again selecting “Create pull request.”
2. Your teacher will now get a message that someone has code they would like to merge into the original repository. If everything was done correctly, your class's code will be easily merged into the class repo's current version (the head commit) on the master branch. The current commit in that master branch will have the original code modified with the code of every other team. If two different developers both make changes on the same line of code, merging the commits requires work in the git shell—skill beyond what you will learn in this course.

Was the teacher able to easily merge your code with the code from others merged before you?

1. After all of the changes for your class have been incorporated you will want to get all the changes made to the class' repo into your copy of the code. To do this, you will need to resync the original fork you made. You will need the URL of the original repository and you will need to use the git shell.
2. Visit the original repository on Github (not your fork) and click on “Clone or Download” and copy the URL.
3. Open the git shell by right-clicking the repository in the GitHub for Windows/Mac Client and choosing “Open in git shell.” A PowerShell window should open, and the path should be the location of your local repository followed by [master].



1. Run the command:

**git remote add upstream PASTE URL HERE**

This adds a pointer to the remote “upstream” version of this repository. Next, run the command:

**git fetch upstream**

This gets the code from the original master branch, which should now have every team's code incorporated. Next, run the command:

**git merge upstream/master**

This command updates your fork to be the same as the original repository.

You should now have all the code and can run the tournament!

**Part V. Run the Iterative Prisoner’s Dilemma Tournament**

1. Once everyone’s code has been merged with the original repo and you have re-synced your fork, you are ready to run the tournament. Make sure you are in the “master” branch of your fork so you have code from all teams. Execute the function and examine the tournament.txt file. Who won? How did your strategy do overall?
2. Against which strategies did your strategy perform well? Which strategies defeated you? Explain why the winning strategy or strategies performed well.
3. Write a strategy in pseudocode that would do better when paired with a team that you fared poorly against in the competition. Explain why you think this new strategy would or would not do well overall.

Conclusion

1. Game theory describes any situation in which two or more people have to make a decision without knowing the other person's decision. Usually, the consequences of our own decision depend on other people's decision. So we have to make our decision based on guesses about what the other person will do.

Describe some situations in which you make a decision based on guesses about other people's decisions.

1. Use the Web to find about how computer simulation has changed the social sciences, including applications in diplomacy, warfare, and economics. Briefly describe your findings.
2. Practice Opportunity for the Create Performance Task.

Each function you create is an abstraction. “Explain how an abstraction you created helped manage the complexity of your program.” (adapted from College Board Create Performance Task Part 2d.)

*Note: This task does not duplicate the content of the College Board task or rubric. The task provided here contains elements that are different than the College Board Performance Tasks and Rubrics. Please reference official College Board materials.*